计 算 机 科 学 丛 书

CRC Press

操作系统设计

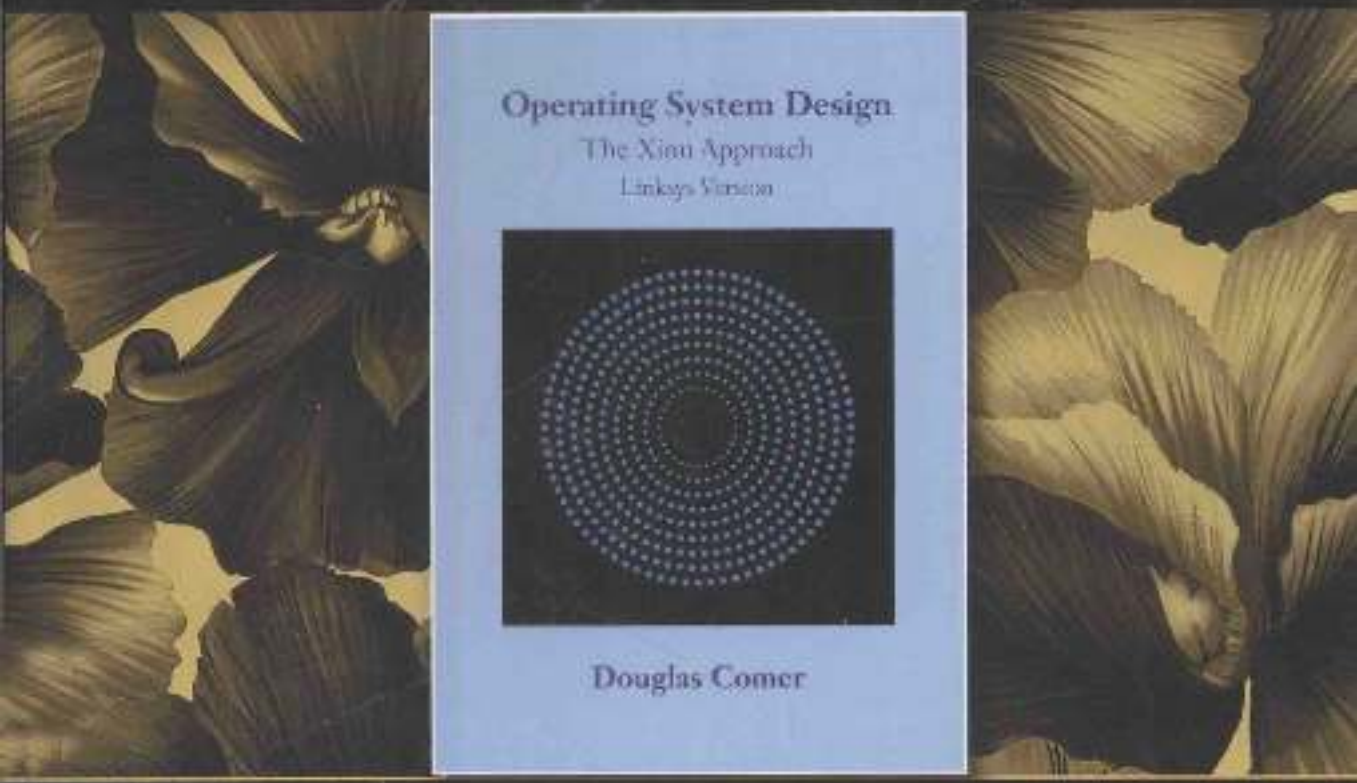
Xinu方法

(美) Douglas Comer 著
普度大学

邹恒明 周亮 曹浩 刘正邦 等译
上海交通大学

Operating System Design

The Xinu Approach, Linksys Version



机械工业出版社
China Machine Press

操作系统设计 Xinu方法

Operating System Design The Xinu Approach, Linksys Version

本书以Xinu（一个小型简洁的操作系统）为例，全面介绍操作系统设计方面的知识。本书着重讨论用于嵌入式设备的微内核操作系统，采用的方法是在现有的操作系统课程中纳入更多的嵌入式处理内容，而非引入一门教读者如何在嵌入式系统上编程的新课程。

本书从底层机器开始，一步步地设计和实现一个小型但优雅的操作系统Xinu，指导读者通过实用、简单的原语来构造传统的基于进程的操作系统。本书回顾了主要的系统组件，并利用分层设计范式，以一种有序、易于理解的方式组织内容。

作者的网站www.xinu.cs.purdue.edu提供了便于学生搭建实验环境的软件和资料。

本书特点

- 解释每个操作系统抽象的产生，展示如何通过简洁高效的设计来组织这些抽象。
- 层层剥离系统的每一层，从原始硬件到可运行的操作系统。
- 涵盖系统的每一部分，这样读者看到的不是一两个部分如何交互，而是整个系统如何组合在一起。
- 提供文中描述的所有部分的源代码，方便读者检查、修改、工具化、调整、扩展或者将其移植到其他架构。
- 阐明操作系统的每一部分是如何满足设计的，以帮助读者理解可选的设计方案。

作者简介



Douglas Comer 美国普度大学计算机系杰出教授，国际公认的计算机网络、TCP/IP协议、Internet和操作系统设计方面的专家。Comer出版了多部优秀的教材和专著，被翻译成16种语言，并广泛用于世界各地的工业界和学术界。Comer教授划时代的三卷巨著《Internetworking with TCP/IP》对网络和网络教育产生了革命性的影响。Comer博士是ACM院士、普度教育学院院士。





上架指南：计算机/操作系统
ISBN 978-7-111-42826-8
9 787111 428268 >
定价：79.00元

CRC Press
客服电话：(010) 88378991, 88361066
购书电话：(010) 68326294, 88379649, 68995259
投稿电话：(010) 88379604
数字阅读：www.hzmedia.com.cn
学习网站：www.hzbook.com
华章网站：www.china-pub.com

计 算 机 科 学 丛 书

操作系统设计

Xinu方法

(美) Douglas Comer 著 邹恒明 周亮 曹浩 刘正邦 等译
普渡大学 上海交通大学

Operating System Design

The Xinu Approach, Linksys Version



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

操作系统设计：Xinu 方法了（美）科默（Comer，D.）著；邹恒明等译．北京：机械工业出版社，2013.10
（计算机科学丛书）
书名原文：Operating System Design：The Xinu Approach，Linksys Version
ISBN 978-7-111-42826-8

Ⅰ.操… Ⅱ.①科… ②邹… Ⅲ.操作系统-程序设计 Ⅳ.TP316

中国版本图书馆 CIP 数据核字（2013）第 123969 号

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2012-4866

Operating System Design：The Xinu Approach（Linksys Version）by Douglas Comer（ISBN：978-1-4398-8109-5）.
Copyright © 2012 by Taylor & Francis Group LLC.
Authorized translation from the English language edition published by CRC Press，part of Taylor & Francis Group LLC. All rights reserved.
China Machine Press is authorized to publish and distribute exclusively the Chinese（Simplified Characters）language edition. This edition is authorized for sale in the People's Republic of China only（excluding Hong Kong，Macao SAR and Taiwan）. No part of this publication may be reproduced or distributed in any form or by any means，or stored in a database or retrieval system，without the prior written permission of the publisher.
Copies of this book sold without a Taylor & Francis sticker on the cover are unauthorized and illegal.

本书原版由 Taylor & Francis 出版集团旗下 CRC 出版公司出版，并经授权翻译出版。版权所有，侵权必究。
本书中文简体字翻译版授权由机械工业出版社独家出版并限在中国大陆地区销售。未经出版者书面许可，不得以任何方式复制或抄袭本书的任何内容。
本书封面贴有 Taylor & Francis 公司防伪标签，无标签者不得销售。

本书对操作系统的内存管理、进程管理、进程协调和同步、进程间通信、实时时钟管理、设备无关的 I/O、设备驱动、网络协议、文件系统等进行了详细的介绍，并利用分层的设计模式，以一种有序、易于理解的方式来阐述这些内容。本书以 Xinu 操作系统为系统设计的样板和模式，从一个模机开始，一步一步地设计和实现一个小型但优雅的操作系统。本书的样本代码可以运行在 Linksys E2100L 无线路由器上。

本书适用于高年级的本科生或低年级的研究生，也适用于那些拟了解操作系统的计算机从业人员。学习本书前，学生需要具备基本的程序设计能力，应当理解基本的数据结构，包括链表、线和队列，并且应当用 C 语言写过程序。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：盛思源
燕城市京瑞印刷有限公司印刷
2014 年 1 月第 1 版第 1 次印刷
185mm × 260mm · 23.5 印张
标准书号：ISBN 978-7-111-42826-8
定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的浪潮中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅确划了研究的范畴，还展示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减损。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育
华章科技图书出版中心

译者序

Operating System Design: The Xinu Approach, Linksys Version

——像小孩那样学习

我实在告诉你们，你们若不回转，变成小孩的样式，断不得进天国。

——马太福音 18：3

这是耶稣教训门徒所说的话。虽然讲的是天国的事情，但不经意间已指出了一条学习的路径。众所周知，小孩接受新鲜事物的能力非常强，小孩能够随着时间的推移而不知不觉地学会说话和日常的生活规则，而成人学习一门新语言或一种新技能则通常较为困难。

小孩对语言或技能的掌握是在成人所说所行的点滴中先行模仿然后再悟出的，也就是通过模仿从细节中理出了头绪。小孩的这种学习方法是一种典型的先实际后理解、先细节后全景、从只见树木到领见森林的转变。这与成人所惯常采用的学习方式有很大不同。成人采用的学习方法通常是先理论后实践、先全貌后细节、先森林后树木式的由高至低逐步推进的方式。这在大学的“操作系统”课程学习中体现得十分明显。一般大学的普遍做法是先介绍操作系统的全貌，然后介绍操作系统的个体组成部分，再附加一些动手实验，所谓通过实验加深对理论的理解（其他课程的学习方式也大同小异）。此种模式是一种典型的自顶向下的演绎模式，由于其由来已久，人们已经习以为常，并没有想过有什么不妥。但操作系统及许多大学课程教学中所呈现的事实是，很多人对其感到枯燥难懂，难以掌握。

既然如此，为什么不试试另一种学习模式，即先动手后理解、先细节后抽象、由树木到森林，像小孩一样来学习操作系统或其他专业课程呢？对于操作系统来说，完全可以采用由底至上的方法，先不讲操作系统理论和模型，而是先来实际设计一个计算机的内存管理系统，在动手实践中逐渐抽象出规律，然后上升为理论和模型。在讨论完内存管理之后再来阐述如何调度程序，这样逐次推进，最后搭起整个操作系统。这就像把一大堆积木呈现给读者，先搭起一个长方体，然后搭起几个轮子，再搭上一些窗户，最终，一辆汽车就形成了。

本书所采取的正是由底至上、由动手到理论、由具体到抽象、由细节到整体、由树木到森林的学习模式来阐述操作系统的原理和设计。该书以细节为核心，以设计与实现为手段，通过设计和实现操作系统的各种功能来引入操作系统的原理和模型，并以设计人员在构建一个操作系统时所遵循的工作次序来组织全书。该书从一个裸机开始，一步一步地设计和实现一个小型但优雅的操作系统——Xinu。本书每一章描述设计 Xinu 系统架构里的一个组件，并提供示例软件来演示该层架构所提供的功能。这种与传统模式的截然不同让人一开始有些反感，但在体会了此种方式的美妙后又会恍然大悟。

该书的最大特点是实践性强，以与实际情况非常贴切的场景为背景，以动手操作为推手，以实际代码为讨论对象，将操作系统的各种实战原理和模式娓娓道来，易于理解和消化。读者只要顺着书的讲解，并配合运行和分析其附带的代码，即可掌握操作系统的设计和实现。该书在美国普渡大学所受到的青睐从一个侧面佐证了这一点。

本书适用于高年级的本科生或低年级的研究生，也适用于那些想了解操作系统的计算机从业人员。鉴于该书的篇幅和所涉及的具体动手实现，建议分配 4 个学分或以上来组织本门课程。另外，考虑到并不是所有人都喜爱先细节后抽象的模式，建议与一本优良的阐述操作系统原理的书结合进行讲授，效果或将更加令人满意。

本书由上海交通大学 2012 年春季高级操作系统课程组集体翻译，因人员众多，就不一一列出。在此特向对本书翻译提供帮助的人员表示感谢。

鉴于译者水平所限，书中纰漏甚至错误在所难免，还望读者不吝指正。

邹恒明

2013 年 5 月于美国达拉斯

前言

Operating System Design: The Xinu Approach, Linksys Version

建造计算机操作系统有点像编织地毯。这两种工作的最终成品都是一个和谐一致、大型、复杂的人造系统。在每一种情况下，最后的人造成品都是由细微但却精巧的步骤所构造。在编织细节时，细节是至关重要的，因为一点点不协调的瑕疵都很容易观察到。就像编织里的线面一样，加入到操作系统里的每个新组件都需要与整体的设计相协调。从这个角度看，将不同片段组装起来的机械加工只是整个建造过程中的一小部分，一个大师级的产品必须以某个模式为基本，所有参与系统设计的工作人员都必须遵守这种模式。

有讽刺意味的是，现有的操作系统教材或课程很少对底层的模式和原理进行解释，而这些模式和原理正是操作系统构造的基础。在学生看来：操作系统似乎是一个暗箱，而现有的教材则加强了这种误解，因为这些教材所解释的不过是操作系统的功能，其关注的也只是操作系统各种能力的使用。更为重要的是，学生在学习操作系统时采取的是从操作系统外面来察看的方式，从而常常导致这样一种感觉：认为操作系统由一组抽象的界面所组成，这些界面下的功能由一大堆晦涩神秘的代码连接在一起，而这些神秘的代码本身还包含着许多与机器硬件直接相关的、无规律可寻的奇技巧术。

令人惊奇的是，学生一旦从大学毕业，就马上觉得与操作系统有关的工作已经结束，自己不再需要理解或学习操作系统，因为由商业公司和开源社区所构造的现有操作系统足以应付各种需要，没有自己什么事情了。但没有什么比这种想法离真理更远了。有讽刺意味的是，尽管为个人计算机设计传统操作系统的公司数量比以前更少了，但社会和行业对操作系统技能的需求却在增长，许多公司雇佣大学生来从事操作系统方面的工作。社会上这些对操作系统技能的需求来源于更便宜的微处理器，这些便宜的微处理器嵌入在智能手机、视频游戏、iPod、Internet 路由器、线缆和机顶盒以及打印机中。

在与嵌入式系统打交道时，有关原理和结构的知识非常关键，因为程序员可能需要在现有的操作系统内部构造某种或某个新的机制，或者对现有操作系统进行修改以便可以在新的硬件平台上运行。而且，为嵌入式设备编写应用程序时需要理解下层的操作系统。如果不理解操作系统设计的各种细微之处，则不可能充分利用这些小型嵌入式处理器的能力。

本书的目的是揭开操作系统设计中的神秘感，将方方面面的材料整合为一个系统化的整体。本书对操作系统的主要系统组件进行了详细阐述，并以一种层次架构的设计范式来组织这些组件，从而以一种有序、可理解的方式来展开这些内容。与其他评述性书籍不同的是，本书并不尽可能多地提供不同方案，呈现给读者的将是一个基于传统过程的、使用实际的、直截了当的原语所构造的操作系统。本书从一个裸机开始，一步一步地设计和实现一个小型但优雅的操作系统。这个称为 Xinu 的操作系统将成为系统设计的样板和模式。

虽然 Xinu 操作系统的规模较小，可以完全容纳在本书中，但该系统却包括了构成一个普通操作系统的全部组件：内存管理、进程管理、进程协同和同步、进程间通信、实时时钟管理、设备独立的输入输出、设备驱动、网络协议和一个文件系统。本书将这些组件组织成一个层次架构，使它们之间的相互连接清晰可见，设计过程浅显易懂。尽管规模小，但 Xinu 却拥有大型系统的能力。此外，Xinu 并不是一个玩具系统，它在很多商业产品中得到了应用。使用该系统的厂商包括 Mitsubishi、Lexmark、HP、IBM、Woodward（woodward.com）、Barrand Software 和 Mucness 公司。学生通过本书可以学到的重要一课是：不管是小型嵌入式系统还是大型系统，好的系统设计都一样重要，一个系统的大部分能力都来自于良好的抽象。

本书所覆盖的所有议题都以一种特定的次序排列，这种次序就是设计人员在构建操作系统时所遵守的工作次序。本书每一章描述设计架构里的一个组件，并提供示例软件来演示由该层架构所提供的功能。使用这种方式具有如下几种优点：第一，每一章所解释的操作系统的功能子集均比上一章所讨论的功能子集更大，这种安排使我们在考虑一层特定架构的设计和实现时不用关心后续层面的实现。

第二，每一章的细节描述在第一次阅读时可以跳过去。读者只需要理解该层所提供的服务即可，而不是这些服务是如何实现的。第三，如果按次序阅读本书，读者可以先理解一个功能，然后在后面看到该功能是如何被后续部分所使用的。第四，有智力挑战的议题（如对并发的支持）出现在书的较前面，高层次的操作系统服务则出现在后面。在本书中，读者将看到大部分核心的功能仅仅只用几行代码就可以完成，这样我们就可以将大部分的代码（网络和文件系统）放到书的较后面，在读者已经做出了充分的思想准备后再进行讲解。

如前所述，与其他关于操作系统的许多书不一样的是，本书并不试图对每个系统组件的每种实现方案进行评估，也不对现有的商业系统进行综述。而是选择对一组使用最广泛的操作系统原语的实现细节进行阐述。例如，在讨论进程协调的一章，我们解释的是信号量（使用最广泛的进程协调原语）原语，而对其他原语（如监视器）的讨论则放到练习里。我们的目的是展示如何将原语在传统的硬件上实现，消除神秘。学生一旦理解了一组特定原语的魔力，其他原语的实现也就容易掌握了。

本书的示例代码可以运行在 Linksys E2100L 无线路由器上，该无线路由器在零售商店里就可以买到。只不过，我们并不是将 Linksys 硬件作为一个无线路由器来使用。我们的做法是，打开 Linksys 设备，将一根串行线连接到其控制端口，使用该串行线来中断 Linksys 正常的启动过程，并通过输入命令来迫使 Linksys 硬件下载和运行一个 Xinu 操作系统副本。也就是说，我们基本上忽略供应商所提供的软件，而是对其底层的硬件进行控制来运行 Xinu。

本书适用于高年级的本科生或者研究生，也适用于那些想了解操作系统的计算机从业人员。在本书所提供的全部材料里，虽然没有任何议题的难度达到不能理解的程度，但学习本书的全部内容可能需要超过一学期的时间。本科生里很少有学生能够熟练地阅读串行程序，而理解运行时环境的细节或机器架构的学生就更少了。因此，必须对学生进行仔细引导，以使他们可以掌握进程管理和进程同步的知识。如果时间有限，我推荐覆盖的内容包括第 1 章～第 7 章（进程管理），第 9 章（基本的内存管理），第 12 章（中断处理），第 13 章（时钟管理），第 14 章（设备无关的 I/O）和第 19 章（文件系统）；此外，对于一个完整学期的本科生课程来说，讨论第 20 章的远程文件系统等基本的远程访问议题也很重要。对于研究生课程来说，学生应当完整地阅读整本书，课堂讨论则应该专注于一些微妙的细节，各种折中和不同实现方案的比较。不管是本科生课程还是研究生课程，都应该包括的两个议题是：1）在初始化阶段，当一个运行中的程序转化为一个进程时所发生的各种改变；2）当输入行里的字符序列作为一个字符串变量传递给命令进程时，在操作系统先里所发生的转化。

在所有情况下，如果学生能够在实验室中对系统进行动手实验，则学习的效果将大幅提高。理想的状态下，学生可以在课程的最初几天或几个星期开始使用这个系统，然后再试图理解系统的内部结构。本书第 1 章提供了几个例子和一些能够引起学生兴趣的实验（令人吃惊的是，很多学生在学习过操作系统课程后，却没有写过一个并发程序或使用过操作系统功能）。

如果要在一个学期内覆盖本书的大部分内容，则要求很快的进度，而这在本科生课程里难以达到。此时，选择略去哪些内容将很大程度上取决于选修本课程的学生的背景。在系统课程里，我们需要课堂讲解时间来帮助学生理解动机和细节。如果学生修过的“数据结构”课程里对内存管理和表处理进行过讨论，则本书第 4 章和第 9 章的内容可以略过。如果学生在将来会选修网络方面的课程，则第 17 章的网络协议内容也可以跳过。此外，本书包括一章远程磁盘系统和一章远程文件系统，这两章的内容存在一些相似之处，可以略过一章。相对来说，远程磁盘系统一章的内容可能更加贴切，因为该章引入了磁盘块缓存的议题，而该议题对于许多操作系统来说都非常重要。

在研究生课程里，课堂时间可以用来讨论动机、原理、折中、不同原语集和不同的实现方案比较。学生在本课程学习结束后，应当对进程模型、中断和进程之间的关系有一个深刻的理解，同时也将具备理解、创建和修改系统组件的能力。学生应当在大脑中建立起了整个系统的完整概念模型，并且知道所有的组件之间是如何交互协作的。

我推荐在各个层面上设计程序设计实验。本书的许多练习都推荐对代码进行修改或者测量，或者尝试不同的实现方案。相关的软件可在下面的网站上免费下载，该网站上还列有如何创建一个 Linksys

实验平台的指令：www.xinu.cs.purdue.edu。

因为 Linksys 的硬件非常便宜，所以构建一个实验的成本很低。此外，我们也有用于其他硬件平台的软件版本，这些版本包括 x86 和 ARM 的一个功能有限的版本。

本书中的许多练习都建议进行改进、实验和不同实现，但是也可以设计大型实验项目。可以用于不同硬件平台的大型实验例子包括：虚拟内存系统、不同计算机之间指令执行的同步机制、虚拟网络的设计等。例如，普度大学的一些学生就将 Xinu 操作系统移植到了各种处理器上，或者为各种 I/O 设备编写了设备驱动程序。

学习本书前，学生需要具备基本的程序设计能力。学生应当理解基本的数据结构，这些基本结构包括链表、栈和队列，并且应当用 C 语言写过程序。

最后，我鼓励设计人员尽可能使用高级程序设计语言，仅在必要的情况下才使用汇编语言。根据这种原则，Xinu 操作系统的大部分都是用 C 语言编写的。少数一些与机器相关的功能，如上下文切换和中断分配器的最底层功能，则是用汇编语言写成的。所有的汇编语言代码都附有解释和注释，使读者无需学习汇编语言的细节就可以理解这些代码。此外，我们还提供用于其他平台的 Xinu 版本，这样就可以对在各种平台上实现 Xinu 操作系统的成本进行比较。例如，我们可以对在 MIPS 处理器上实现 Xinu 所需要的代码量和在其他处理器架构（如 x86）上实现 Xinu 所需要的代码量进行比较。

本书的成书要归功于我过去在商业操作系统上所获得的各种经验，这些经验有好也有坏。虽然 Xinu 操作系统与现有的操作系统在内部机制上并不相同，但其基本的思想却并不新颖。另外，虽然 Xinu 系统里的许多概念和名称都来自于 UNIX 系统，但读者应当注意，这两个系统里的许多函数所使用的参数和内部结构有巨大的不同。因此，为一个系统所写的应用程序在未经修改的情况下不能在另一个平台上运行。

我感谢为 Xinu 项目贡献了思想、辛劳和激情的所有人的帮助。在过去的岁月里，普度大学的许多研究生都从事过本系统的工作，他们为 Xinu 进行过移植，写过设备驱动。从原始的系统版本开始，Xinu 到目前已经走过了 30 多年的历程。本书的 Xinu 版本是原始版本的一个完全重写，但却保留了原始设计的优雅。Dennis Brylow 将 Xinu 移植到了 Linksys 平台，并且创建了许多底层的构件，包括启动代码、上下文切换和 Ethernet 驱动。Dennis 还设计了重启机制，并应用在普度大学的实验室里。另外，我特别要感谢我的妻子和我的合作伙伴 Christine，她的仔细编辑和建议让本书改善良多。

Douglas E. Comer

2011 年 8 月

关于作者

Operating System Design: The Xinu Approach, Linksys Version

Douglas Comer 是美国普度大学（Purdue University）计算机系的杰出教授，国际公认的计算机网络、TCP/IP 协议、Internet 和操作系统设计的专家。Comer 发表论文无数，出版专著多部，是一位为研究和教育而开发课程体系和实验项目的先驱。

作为一个多产作家，Comer 博士的书被翻译成 16 种语言，广泛应用于全世界的计算机科学、工程和工商行政管理等院系和行业。Comer 博士划时代的三卷巨著《Internetworking with TCP/IP》对网络和网络教育产生了革命性的影响。他所编写的教科书和富有创意的实验手册已经塑造和继续塑造着研究生和本科生的教学课程体系。

Comer 博士所写书籍的精确和洞见反映出他在计算机系统领域的广泛背景。他的研究横跨硬件和软件。他创建了一个完整的操作系统 Xinu，编写了设备驱动程序，为传统计算机和网络处理器实现了网络协议软件。Comer 博士的研究成果已经应用到工业界的各种产品中。

Comer 博士创建和主讲的课程包括“网络协议”、“操作系统”、“计算机体系架构”，其听众既有大学生和学术界同仁，也有工业界的工程师。他创新的教育实验让他和他的学生能够设计和实现大型、复杂系统的原型，并对结果原型的性能进行度量。Comer 博士长期在公司、大学和会议上讲课和演说，还为工业界提供咨询服务，以帮助他们设计计算机网络和系统。

二十多年来，Comer 教授担任研究期刊《Software — Practice and Experience》主编。在普度大学停薪留职期间，他在思科公司（Cisco）担任研究副总裁。Comer 博士是 ACM 院士，普度教育学院院士和无数奖项的获得者，其中包括 Usenix 终身成就奖。

关于 Comer 博士的更多信息可在下面网站找到：www.cs.purdue.edu/people/comer。

关于 Comer 博士所著书籍的更多信息可在下面网站找到：www.comerbooks.com。

出版者的话
译者序
前言
关于作者

第1章 引言和概述 …… 1
1.1 操作系统 …… 1
1.2 本书的研究方法 …… 1
1.3 分层设计 …… 2
1.4 Xinu 操作系统 …… 3
1.5 操作系统不是什么 …… 3
1.6 从外面看操作系统 …… 4
1.7 其他章节概要 …… 4
1.8 观点 …… 5
1.9 总结 …… 5
练习 …… 5
第2章 并发执行与操作系统服务 …… 6
2.1 引言 …… 6
2.2 多活动的编程模型 …… 6
2.3 操作系统服务 …… 7
2.4 并发处理的概念和术语 …… 7
2.5 单行程序和并发程序的区别 …… 8
2.6 多进程共享同一段代码 …… 9
2.7 进程退出与进程终止 …… 11
2.8 共享内存，竞争条件和同步 …… 11
2.9 信号量与互斥 …… 14
2.10 Xinu 中的类型命名方法 …… 15
2.11 使用 Kputc 和 Kprintf 进行操作系统的调试 …… 16
2.12 观点 …… 16
2.13 总结 …… 16
练习 …… 17
第3章 硬件和运行时环境概览 …… 18
3.1 引言 …… 18
3.2 IC21001. 的物理和逻辑结构 …… 18
3.3 处理器结构和寄存器 …… 19
3.4 总线操作：获取－存储范式 …… 19
3.5 直接内存访问 …… 19
3.6 总线地址空间 …… 20
3.7 内核段 KSEG0 和 KSEG1 的内容 …… 20
3.8 总线启动的静态配置 …… 21
3.9 调用约定和运行时栈 …… 21
3.10 中断和中断处理 …… 22
3.11 异常处理 …… 23
3.12 计时器硬件 …… 23
3.13 串行通信 …… 24
3.14 轮询与中断驱动 I/O …… 24
3.15 内存缓存和 KSEG1 …… 24
3.16 存储布局 …… 24
3.17 内存保护 …… 25
3.18 观点 …… 25
练习 …… 25
第4章 链表与队列操作 …… 26
4.1 引言 …… 26
4.2 用于进程链表的统一数据结构 …… 26
4.3 简洁的链表数据结构 …… 27
4.4 队列数据结构的实现 …… 28
4.5 内联队列操作函数 …… 29
4.6 获取链表中进程的基础函数 …… 29
4.7 FIFO 队列操作 …… 30
4.8 优先级队列的操作 …… 32
4.9 链表初始化 …… 33
4.10 观点 …… 34
4.11 总结 …… 34
练习 …… 35
第5章 调度和上下文切换 …… 36
5.1 引言 …… 36
5.2 进程表 …… 36
5.3 进程状态 …… 38
5.4 就绪和当前状态 …… 38
5.5 调度策略 …… 38
5.6 调度的实现 …… 39
5.7 上下文切换的实现 …… 41
5.8 内存中保存的状态 …… 41

5.9 在 MIPS 处理器上切换上下文 …… 41
5.10 重新启动进程执行的地址 …… 43
5.11 并发执行和 null 进程 …… 44
5.12 使进程准备执行和调度不变式 …… 44
5.13 推迟重新调度 …… 45
5.14 其他进程调度算法 …… 47
5.15 观点 …… 47
5.16 总结 …… 47
练习 …… 47
第 6 章 更多进程管理 …… 49
6.1 引言 …… 49
6.2 进程挂起和恢复 …… 49
6.3 自我挂起和信息隐藏 …… 49
6.4 系统调用的概念 …… 50
6.5 禁止中断和恢复中断 …… 51
6.6 系统调用模板 …… 51
6.7 系统调用返回 SYSERR 和 OK 值 …… 51
6.8 挂起的实现 …… 52
6.9 挂起当前进程 …… 53
6.10 suspend 函数的返回值 …… 53
6.11 进程终止和进程退出 …… 54
6.12 进程创建 …… 56
6.13 其他进程管理函数 …… 59
6.14 总结 …… 60
练习 …… 61
第 7 章 协调并发进程 …… 62
7.1 引言 …… 62
7.2 进程同步的必要性 …… 62
7.3 计数信号量的概念 …… 63
7.4 避免忙等待 …… 63
7.5 信号量策略和进程选择 …… 63
7.6 等待状态 …… 64
7.7 信号量数据结构 …… 64
7.8 系统调用 wait …… 65
7.9 系统调用 signal …… 66
7.10 静态和动态信号量分配 …… 66
7.11 动态信号量的实现示例 …… 67
7.12 信号量删除 …… 68
7.13 信号量重置 …… 69
7.14 多核处理器之间的协调 …… 69
7.15 观点 …… 70
7.16 总结 …… 70
练习 …… 71
第 8 章 消息传递 …… 72
8.1 引言 …… 72
8.2 两种类型的消息传递服务 …… 72
8.3 消息使用资源的限制 …… 72
8.4 消息传递函数和状态转换 …… 73
8.5 send 的实现 …… 73
8.6 receive 的实现 …… 74
8.7 非阻塞消息接收的实现 …… 75
8.8 观点 …… 75
8.9 总结 …… 75
练习 …… 76
第 9 章 基本内存管理 …… 77
9.1 引言 …… 77
9.2 内存的类型 …… 77
9.3 重量级进程的定义 …… 77
9.4 小型嵌入式系统的内存管理 …… 78
9.5 程序段和内存区域 …… 78
9.6 嵌入式系统中的动态内存分配 …… 79
9.7 低层内存管理器的设计 …… 79
9.8 分配策略和内存持久性 …… 80
9.9 追踪空闲内存 …… 80
9.10 低层内存管理的实现 …… 80
9.11 分配非存储 …… 82
9.12 分配栈存储 …… 83
9.13 释放堆和栈存储 …… 84
9.14 观点 …… 86
9.15 总结 …… 87
练习 …… 87
第 10 章 高级内存管理和虚拟内存 …… 88
10.1 引言 …… 88
10.2 分区空间分配 …… 88
10.3 缓冲池 …… 88
10.4 分配缓冲区 …… 89
10.5 将缓冲区返回给缓冲池 …… 90
10.6 创建缓冲池 …… 91
10.7 初始化缓冲池表 …… 93
10.8 虚拟内存和内存复用 …… 93
10.9 实地址空间和虚地址空间 …… 93
10.10 支持按需换页的硬件 …… 94

10.11 使用页表的地址翻译 …… 95
10.12 页表项中的元数据 …… 95
10.13 按需换页以及设计上的问题 …… 95
10.14 页面替换和全局时钟算法 …… 96
10.15 观点 …… 97
10.16 总结 …… 97
练习 …… 97
第 11 章 高层消息传递 …… 98
11.1 引言 …… 98
11.2 进程间通信端口 …… 98
11.3 端口实现 …… 98
11.4 端口表初始化 …… 99
11.5 端口创建 …… 100
11.6 向端口发送消息 …… 101
11.7 从端口接收消息 …… 102
11.8 端口的删除和重置 …… 103
11.9 观点 …… 106
11.10 总结 …… 106
练习 …… 106
第 12 章 中断处理 …… 107
12.1 引言 …… 107
12.2 中断的优点 …… 107
12.3 中断分配 …… 107
12.4 中断向量 …… 107
12.5 中断向量号的分配 …… 108
12.6 硬件中断 …… 108
12.7 中断请求的局限性和中断多路复用 …… 109
12.8 中断软件和分配 …… 109
12.9 中断分配器底层部分 …… 110
12.10 中断分配器高层部分 …… 112
12.11 禁止中断 …… 114
12.12 函数中中断代码引起的限制 …… 115
12.13 中断过程中重新调度的必要性 …… 115
12.14 中断过程中的重新调度 …… 115
12.15 观点 …… 116
12.16 总结 …… 116
练习 …… 117
第 13 章 实时时钟管理 …… 118
13.1 引言 …… 118
13.2 定时事件 …… 118
13.3 实时时钟和计时器硬件 …… 118
13.4 处理实时时钟中断 …… 119
13.5 延时与抢占 …… 119
13.6 使用计时器来模拟实时时钟 …… 120
13.7 抢占的实现 …… 120
13.8 使用增量链表对延迟进行有效管理 …… 120
13.9 增量链表的实现 …… 121
13.10 将进程转入睡眠 …… 122
13.11 定时消息接收 …… 124
13.12 唤醒睡眠进程 …… 127
13.13 时钟中断处理 …… 127
13.14 时钟初始化 …… 128
13.15 间隔计时器管理 …… 129
13.16 观点 …… 130
13.17 总结 …… 130
练习 …… 130
第 14 章 设备无关的 I/O …… 132
14.1 引言 …… 132
14.2 I/O 和设备驱动的概念结构 …… 132
14.3 接口抽象和驱动抽象 …… 133
14.4 I/O 接口的一个示例 …… 134
14.5 打开-读-写-关闭范式 …… 134
14.6 绑定 I/O 操作和设备名 …… 134
14.7 Xinu 中的设备名 …… 135
14.8 设备转换表概念 …… 135
14.9 设备和共享驱动的多个副本 …… 136
14.10 高层 I/O 操作的实现 …… 138
14.11 其他高层 I/O 函数 …… 138
14.12 打开、关闭和引用计数 …… 141
14.13 devtab 中的字条目和错误条目 …… 143
14.14 I/O 系统的初始化 …… 143
14.15 观点 …… 146
14.16 总结 …… 147
练习 …… 147
第 15 章 设备驱动示例 …… 148
15.1 引言 …… 148
15.2 tty 抽象 …… 148
15.3 tty 设备驱动的组成 …… 149
15.4 请求队列和缓冲区 …… 149
15.5 上半部和下半部的同步 …… 150

15.6 硬件缓冲区和驱动设计 …… 151
15.7 tty 控制块和数据声明 …… 151
15.8 次设备号 …… 153
15.9 上半部 tty 字符输入（ttyGetc） … 153
15.10 通用上半部 tty 输入（ttyRead） …… 154
15.11 上半部 tty 字符输出（ttyPutc） … 155
15.12 开始输出（ttyKickOut） …… 156
15.13 上半部 tty 多字符输出（ttyWrite） …… 157
15.14 下半部 tty 驱动函数（ttyInterrupt） …… 157
15.15 输出中断处理（ttyInter_out） …… 159
15.16 tty 输入处理（tty Inter-in） …… 161
15.17 tty 控制块初始化（ttyInit） …… 166
15.18 设备驱动控制 …… 168
15.19 观点 …… 169
15.20 总结 …… 169
练习 …… 169
第 16 章 DMA 设备和驱动（以太网） …… 171
16.1 引言 …… 171
16.2 直接内存访问和缓冲区 …… 171
16.3 多缓冲区和环 …… 171
16.4 使用 DMA 的以太网驱动例子 …… 172
16.5 设备的硬件定义和常量 …… 172
16.6 环和内存缓冲区 …… 174
16.7 以太网控制块的定义 …… 175
16.8 设备和驱动初始化 …… 177
16.9 分配输入缓冲区 …… 181
16.10 从以太网设备中读取数据包 …… 182
16.11 向以太网设备中写入数据包 …… 183
16.12 以太网设备的中断处理 …… 185
16.13 以太网控制函数 …… 187
16.14 观点 …… 189
16.15 总结 …… 189
练习 …… 189
第 17 章 最小互联网协议栈 …… 190
17.1 引言 …… 190
17.2 所需的功能 …… 190
17.3 同步对话、超时和进程 …… 191
17.4 ARP 函数 …… 192
17.5 网络数据包的定义 …… 198
17.6 网络输入进程 …… 199
17.7 UDP 表的定义 …… 202
17.8 UDP 函数 …… 203
17.9 互联网控制报文协议 …… 210
17.10 动态主机配置协议 …… 211
17.11 观点 …… 214
17.12 总结 …… 214
练习 …… 214
第 18 章 远程磁盘驱动 …… 215
18.1 引言 …… 215
18.2 磁盘抽象 …… 215
18.3 磁盘操作驱动支持 …… 215
18.4 块传输和高层 I/O 函数 …… 215
18.5 远程磁盘范式 …… 216
18.6 磁盘操作的语义 …… 216
18.7 驱动数据结构的定义 …… 217
18.8 驱动初始化（rdsInit） …… 221
18.9 上半部打开函数（rdsOpen） …… 223
18.10 远程通信函数（rdscomm） …… 224
18.11 上半部写函数（rdsWrite） …… 226
18.12 上半部读函数（rdsRead） …… 228
18.13 刷新挂起的请求 …… 231
18.14 上半部控制函数（rdsControl） …… 231
18.15 分配磁盘缓冲区（rdsbufalloc） … 233
18.16 上半部关闭函数（rdsCluse） …… 234
18.17 下半部通信进程（rdsprocess） …… 235
18.18 观点 …… 239
18.19 总结 …… 239
练习 …… 240
第 19 章 文件系统 …… 241
19.1 文件系统是什么 …… 241
19.2 文件操作的示例集合 …… 241
19.3 本地文件系统的设计 …… 242
19.4 Xinu 文件系统的数据结构 …… 242
19.5 索引管理器的实现 …… 243
19.6 清空索引块（lfibclear） …… 246
19.7 获取索引块（lfibget） …… 247
19.8 存储索引块（lfibput） …… 247
19.9 从空闲链表中分配索引块（lfiballoc） …… 248

19.10 从空闲链表中分配数据块（lfdbalIoc） …… 249
19.11 使用设备无关的 I/O 函数的文件操作 …… 250
19.12 文件系统的设备设置和函数名称 …… 251
19.13 本地文件系统打开函数（lfsOpen） …… 251
19.14 关闭文件伪设备（lfClose） …… 256
19.15 刷新缓息中的数据（lfflush） …… 256
19.16 文件的批量传输函数（lfWrite、lfRead） …… 257
19.17 在文件中寄找一个新位置（lfSeek） …… 258
19.18 从文件中提取一个字节（lfGetc） …… 259
19.19 改变文件中的一个字节（lfPutc） …… 260
19.20 载入索引块和数据块（lfsetup） … 261
19.21 主文件系统设备的初始化（lfsInit） …… 264
19.22 伪设备的初始化（lfInit） …… 264
19.23 文件截断（lftruncate） …… 265
19.24 初始文件系统的创建（lfscreate） …… 267
19.25 观点 …… 269
19.26 总结 …… 269
练习 …… 269
第 20 章 远程文件机制 …… 270
20.1 引言 …… 270
20.2 远程文件访问 …… 270
20.3 远程文件语义 …… 270
20.4 远程文件设计和消息 …… 271
20.5 远程文件服务器通信 …… 276
20.6 发送一个基本消息 …… 278
20.7 网络字节序 …… 279
20.8 使用设备范式的远程文件系统 …… 279
20.9 打开远程文件 …… 280
20.10 检查文件模式 …… 282
20.11 关闭远程文件 …… 283
20.12 读远程文件 …… 284
20.13 写远程文件 …… 286
20.14 远程文件的定位 …… 288
20.15 远程文件单字符 I/O …… 288
20.16 远程文件系统控制函数 …… 289
20.17 初始化远程文件数据结构 …… 292
20.18 观点 …… 293
20.19 总结 …… 293
练习 …… 294
第 21 章 句法名字空间 …… 295
21.1 引言 …… 295
21.2 透明与名字空间的抽象 …… 295
21.3 多种命名方案 …… 295
21.4 命名系统设计的其他方案 …… 296
21.5 基于句法的名字空间 …… 296
21.6 模式和替换 …… 297
21.7 前缀模式 …… 297
21.8 名字空间的实现 …… 297
21.9 名字空间的数据结构和常量 …… 297
21.10 增加名字空间前缀表的映射 …… 298
21.11 使用前缀表进行名字映射 …… 299
21.12 打开命名文件 …… 302
21.13 名字空间初始化 …… 303
21.14 对前缀表中的项进行排序 …… 305
21.15 选择一个逻辑名字空间 …… 305
21.16 默认层次和空前缀 …… 305
21.17 额外的对象操作函数 …… 306
21.18 名字空间方法的优点和限制 …… 306
21.19 广义模式 …… 307
21.20 观点 …… 307
21.21 总结 …… 308
练习 …… 308
第 22 章 系统初始化 …… 309
22.1 引言 …… 309
22.2 引导程序：从头开始 …… 309
22.3 操作系统初始化 …… 309
22.4 在 i21001 上启动一个可选的映像 …… 310
22.5 Xinu 初始化 …… 310
22.6 系统启动 …… 312
22.7 从程序转化为进程 …… 316
22.8 观点 …… 316

22.9 总结 …… 316
练习 …… 316
第 23 章 异常处理 …… 317
23.1 引言 …… 317
23.2 异常、陷阱和恶意中断 …… 317
23.3 panic 的实现 …… 317
23.4 观点 …… 318
23.5 总结 …… 318
练习 …… 318
第 24 章 系统配置 …… 319
24.1 引言 …… 319
24.2 多重配置的需求 …… 319
24.3 Xinu 系统配置 …… 319
24.4 Xinu 配置文件的内容 …… 320
24.5 计算次设备号 …… 321
24.6 配置 Xinu 系统的步骤 …… 322
24.7 观点 …… 322
24.8 总结 …… 322
练习 …… 322
第 25 章 一个用户接口例子：Xinu 壳 …… 323
25.1 引言 …… 323
25.2 用户接口 …… 323
25.3 命令和设计原则 …… 323
25.4 一个简化壳的设计决策 …… 324
25.5 壳的组织和操作 …… 324
25.6 词法符号的定义 …… 324
25.7 命令行语法的定义 …… 325
25.8 Xinu 壳的实现 …… 325
25.9 符号的存储 …… 327
25.10 词法分析器代码 …… 327
25.11 命令解释器的核心 …… 330
25.12 命令名查询和内部处理 …… 336
25.13 传给命令的参数 …… 336
25.14 向外部命令传递参数 …… 337
25.15 I/O 重定向 …… 339
25.16 示例命令函数（sleep） …… 340
25.17 观点 …… 341
25.18 总结 …… 341
练习 …… 342
附录 1 操作系统移植 …… 343
附录 2 Xinu 设计注解 …… 349
索引 …… 352

引言和概述

我们的小小系统已有灵光的时刻。

——Alfred, Lord Tennyson

1.1 操作系统

每一个智能设备和计算机系统中都隐藏着这么一类软件，它们控制着处理信息、管理资源以及与显示屏、网络、磁盘和打印机等设备通信的工作。总的来说，这些进行控制和协调工作的代码通常叫做执行器、监视器、任务管理器，或者内核，而我们将使用一个更宽泛的术语操作系统。

计算机操作系统是人类创造的最复杂的物体之一：计算机操作系统允许多个计算进程和用户同时共享一个CPU，保护数据免受未经授权的访问，并保持独立输入/输出（I/O）设备的正确运行。操作系统提供的高级服务都是通过向复杂的硬件发送一系列详细的命令实现的。有趣的是，操作系统并不是从外部控制电脑的独立机制——它还包括一些软件，这些软件由执行应用程序的同一处理器执行。事实上，当处理器运行应用程序的时候，处理器是不能执行操作系统的，反之亦然。

保证操作系统总在应用程序运行结束后重新夺回控制权的安排机制使得操作系统的设计变得非常复杂。操作系统最令人印象深刻的方面来自于服务和硬件之间的不同：操作系统在低级的硬件上提供高级服务。随着本书内容的推进，读者就会理解系统软件处理单中行接口这样简单的设备需要做的事情。而其中的哲学原理很简单：操作系统应该提供让编程更加容易的抽象，而不是反映底层硬件设备的抽象。因此，我们得出结论：

> 设计操作系统时，应该隐藏或是给硬件细节，并创建一个为应用程序提供高级服务的抽象机器。

操作系统的设计并不是人们所熟知的工艺。最初，由于计算机的缺乏和价格的昂贵，只有少数程序员有从事操作系统相关工作的机会。而现在，由于先进的微电子技术降低了制造成本使得微处理器不再昂贵，操作系统便成为一种商品，与此同时也只有少数程序员从事操作系统方面的工作。有趣的是，由于微处理器变得非常便宜，大多数电子设备都是从可编程处理器构建得到，而不是从离散的逻辑构建得到。因此，设计与实现微机和微控制器的软件系统不再是专家的专利，它已成为一个称职的系统程序员必须能胜任的技术。

幸运的是，随着生产新机器的技术的发展，我们对于操作系统的理解也在不断提高。研究人员已经找出了根本问题，制定了设计原则，定义了基本的组件，并设计了组件一起工作的机制。更重要的是，研究人员还定义了一系列的抽象，如文件和当前进程（这些抽象对于所有的操作系统都是相同的），并且已经找到了实现这些抽象的有效方式。最后，我们知道了如何将操作系统的不同组件组织成一个有意义的系统设计与实现。

同早期系统相比，现代操作系统是简洁的、可移植的。设计良好的系统都遵循着将软件分割成一系列基本组件的基本设计模式。因此，现代系统就变得更容易理解和修改。相比早期的系统其处理开销也比较小。

供应商出售的大型商业操作系统通常包括很多额外的软件组件。例如，一个典型的操作系统软件发行版包括编译器、连接器、装载程序、库函数和一系列的应用程序。为了区分这些额外的软件和一个基本的操作系统，我们有时会用内核指代常驻在内存中并且提供诸如并发进程支持等关键性服务的代码。在本书中，操作系统这个术语指的就是内核，而不包括其他附加的功能。一个最小化内核功能的设计有时称为微内核设计。我们的讨论就将集中在微内核上。

1.2 本书的研究方法

本书讲解了如何构建、设计并且实现操作系统的内核。书中使用了工程学方法，而不是仅仅罗列

操作系统的特性和抽象地对其进行描述。这种方法向我们展示了每一个抽象是如何建立的，以及如何将这些抽象组织成一个优雅、高效的设计。

这种工程学方法有两个优势。第一，因为本书的内容涵盖操作系统的每一部分，所以读者会看到整个系统如何融合在一起，而不仅仅是一两个部分之间如何交互。第二，由于读者可以得到书中描述的所有部分的源代码，所以任何部分的实现都没有什么神秘的地方——读者可以获得一份系统的副本来检查、修改、工具化、测量、扩展或者将其移植到其他架构。在本书的最后，读者会看到操作系统的每个部分是如何满足设计需求的，以帮助读者理解可选的设计方案。

本书的关注点是实现，这意味着代码是本书的一个重要组成部分。事实上，代码是讨论的核心，必须通过阅读和学习所罗列的程序来欣赏其中的微妙之处和工程中的细节。例子代码都非常精简，这意味着读者可以集中精力在概念的理解上而不需要费力地阅读许多页的代码。但某些练习建议的改进或修改需要读者深入细节或者找到其他方案。熟练的程序员会找到更多方法来改进和扩展我们的系统。

1.3 分层设计

如果设计得好，操作系统的内部可以如最高级的程序一样优雅、简洁。为了达到优雅的目标，本书所描述的设计，将系统功能划分为8个大类，并将这些组件组织成多个层次。系统的每层提供了一个抽象的服务，该服务又通过低层提供的抽象服务实现。该方法的特点很明显：逐渐变大的层次子集可以形成更加强大的系统。我们将会看到如何利用分层方法来帮助设计者降低操作系统的复杂性。

该方法的另一个重要特点体现在运行时的效率上——设计者可以在不引入额外开销的情况下将操作系统的片段构建为一个层次结构。不过，该方法不同于传统的分层系统。在传统的分层系统中，K层的函数只能调用$K-1$层的函数。在这种多层次的方法中，层次只为设计者提供了一个概念模型——在运行时，高层的函数可以直接调用低层的任何函数。我们将看到，直接调用使得整个系统更有效率。

图1-1说明在本书中所使用的层次结构，给出了我们将讨论的组件的预览，并展示了其中所有片
3 段的结构。

分层结构的核心是计算机硬件。虽然硬件不是操作系统本身的一部分，但现代硬件具有的特性允许其与操作系统紧密集成在一起。因此，我们可以认为硬件是层次结构中的第0层。

从硬件开始往上，操作系统的每个层次都会提供更强大的原语，从而为应用程序屏蔽原始的硬件。内存管理层控制和分配内存。进程管理层是操
4 作系统最基础的组成部分，它包括进程调度和上下文切换。接下来一层的功能包含了进程管理的其余部分内容，包括创建、杀死、挂起和恢复进程。进程管理的上层是进程协调组件，它实现了信号量。实时时钟管理的功能包含在接下来的一个层次中，它允许应用软件在一定的时间内推迟响应。实时时钟上面的一层是与设备无关的I/O程序层，它提供我们所熟悉的服务，如读（read）和写（write）操作。在设备程序之上的一层实现网络通信，更上面的一层则实现了文件系统。

系统的内部组织不应该与系统提供的服务相混淆。虽然将组件组织成不同的层次可以简化设计和实

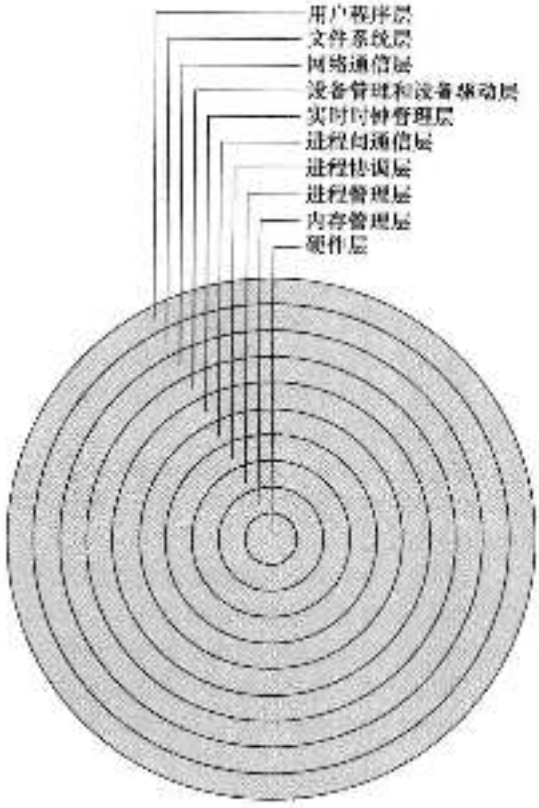


图1-1 在本书中使用的多层次结构

现更加简洁，但最终的层次结构并不能在运行时限制系统调用。也就是说，一旦系统构建完成，各个层次的基础设施都可以暴露给应用程序。例如，应用程序可以调用信号量函数，如可以像调用外层的 poke 函数一样很容易地调用进程间协调层的 wait 和 signal 函数。因此，多层次结构仅仅描述了内部的实现，并不会限制系统所提供的服务。

1.4　Xinu 操作系统

本书中所用的例子都是来自于 Xinu[⊖] 操作系统。Xinu 是一个小型而优雅的系统，主要用在如手机或 MP3 播放器等嵌入式环境中。在通常情况下，在系统启动时，Xinu 和一系列固化的应用程序会一起加载到内存中。当然，如果内存受到限制或者硬件体系结构要求指令使用（与数据）分离的内存，那么 Xinu 就可以在闪存中或者其他只读存储器中执行。然而，在一个典型的系统中，在主内存中执行系统会得到更高的性能。

Xinu 不是一个玩具，它是一个功能强大的操作系统，已在商业产品中使用。例如，Xinu 已使用在 Williams/Bally（主要制造商）旗下出售的弹球游戏中，Woodward 公司使用 Xinu 系统来控制大型燃气/蒸汽和柴油/蒸汽涡轮发动机，Lexmark 公司将 Xinu 作为该公司许多打印机的操作系统。在不同情况下，当设备启动时，硬件加载包含 Xinu 系统的内存映像。

Xinu 包含了操作系统最基础的组件：进程、内存、计时器管理机制、进程间通信设施、设备无关的 I/O 功能和 Internet 协议软件。Xinu 系统可以控制 I/O 设备并且执行一些基本的操作，如从键盘或面板读取按键、在输出设备上显示字符、管理多个并发的计算、控制计时器、在计算任务之间传递消息、并且允许应用程序访问 Internet。

Xinu 系统说明了分层设计是如何应用在实践中的。同时它也展示了这些操作系统的片段是如何作为一个统一整体运行的，以及一个操作系统是如何将这些服务提供给应用程序的。

1.5　操作系统不是什么

在进入操作系统的设计之前，我们需要在学习什么上达成共识。令人惊讶的是，许多程序员都没有一个正确直观的操作系统定义。导致这种问题的原因可能是供应商和计算机专业人员经常将操作系统这个术语泛指由供应商提供的所有软件以及操作系统本身，或者也有可能是很少有程序员直接使用操作系统所提供的服务。在任何情况下，通过排除那些人们熟知的并非操作系统内核的部分，我们就可以很快地给出一个操作系统的定义。

第一，操作系统不是一种编程语言或编译器。当然，一个操作系统必须由某种编程语言编写，并且在设计编程语言时也要考虑到操作系统的特点和特性。软件供应商可能提供多种编译器，将它们集成在操作系统中，由此带来了很多的困惑。然而，操作系统并不依赖于任何一种语言的特性——我们将会看到，可以用一种常规的语言和一个常规的编译器来构建操作系统。

第二，操作系统不是一个窗口系统或浏览器。许多计算机和电子设备都有一个屏幕用来显示图形，而且复杂的系统允许应用程序创建和控制多个独立的窗口。虽然窗口机制本身依赖于操作系统，但是窗口系统可以在无需支持操作系统的情况下被更换掉。

第三，操作系统不是一个命令解释器。嵌入式系统通常包括一个命令行界面（Command Line Interface，CLI），有些嵌入式系统依赖 CLI 来进行所有的控制操作。然而，在一个现代的操作系统中，命令解释器只是一个应用程序，并且在无需修改底层系统的情况下就可以更改解释器。

第四，操作系统不是一个函数库或方法库。应用程序基本上都需要使用库函数，比如发送电子邮件、处理文档、提供数据库访问，或者通过因特网（Internet）来进行通信，并且库函数中的程序往往可以提供一些有意义的功能。尽管许多库函数使用了操作系统的服务，但底层操作系统仍然是独立于库函数的。

⊖ Xinu 这个名字表示 Xinu Is Not Unix，我们可以发现，Xinu 的内部结构与 UNIX（或 Linux）的内部结构截然不同，Xinu 更小，设计更加先进，更易于理解。

第五，操作系统不是计算机开机后最先运行的代码。相反，计算机中包含固件（例如保存在非易失存储器中的程序），它负责初始化各种硬件，将操作系统副本加载到内存中，然后跳转到操作系统开始执行的地方。例如，在个人计算机（Personal Computer，PC）中，这种固件称为基本输入输出系统
6 (Basic Input Output System，BIOS)。

1.6 从外面看操作系统

操作系统的本质在于它给应用程序提供的服务。应用程序通过系统调用来访问操作系统服务。在源代码中，系统调用看起来与普通的函数调用相似。然而，当运行系统调用时，系统调用将控制权转交给操作系统以执行应用程序请求的服务。作为一个集合，系统调用在应用程序和底层操作系统之间建立一个精心设计的边界，这个边界称为应用程序接口（Application Program Interface，API）。API 在定义系统服务的同时也定义了应用程序如何使用这些服务的细节。

为了了解操作系统的内部，首先必须了解其 API 的特点和应用程序如何使用这些服务。本章会介绍一些基本的服务，并用 Xinu 操作系统中的例子来阐明其中的概念。例如，Xinu 的 putc 函数向一个特殊的 I/O 设备写入单个字符。putc 需要两个参数：设备标识符和所要写入的字符。文件 ex1.c 中有一个 C 程序，该程序在 Xinu 中运行后会在控制台显示“hi”：

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  --  write "hi" on the console
 *------------------------------------------------------------------------
 */
void	main(void)
{
	putc(CONSOLE, 'h'); putc(CONSOLE, 'i');
	putc(CONSOLE, '\r'); putc(CONSOLE, '\n');
}
```

这段代码中包含一些 Xinu 使用的约定。源代码中的语句#include <xinu.h>插入了一系列声明，使程序可以引用操作系统的函数。例如，在 Xinu 的全局文件中定义了符号常量 CONSOLE 来对应一个控制台串行设备，通过它，程序员可以与嵌入式系统进行交互。接下来，我们会看到 xinu.h 文件中还包括其他的 include 文件，并学习类似 CONSOLE 这样的名字如何成为一个设备的代名词。现在只需要知道，include 语句必须出现在所有 Xinu 应用程序中就足够了。

要支持与嵌入式系统的通信（例如，为了调试），嵌入式系统上的串行设备需要与常规计算机的
7 终端应用程序连接。每次用户按下计算机键盘上的一个键时，终端应用程序通过串行线路向嵌入式系统发送击键信号。类似地，每次嵌入式系统向串行设备发送一个字符时，终端应用程序需要在用户的屏幕上显示该字符。因此，控制台实际上为嵌入式系统与外界之间的双向通信。

上面所列出的主程序向控制台串行设备写入了 4 个字符：“h”、“i”、回车符和换行符。后两个是控制字符，它们负责将光标移动到下一行的开头。当程序发送控制字符时，Xinu 不执行任何特殊操作——控制字符仅仅像字母数字字符一样被传送到串行设备上。在该例子中，引入控制字符是用来说明 putc 不是行导向的。在 Xinu 系统中，程序员需要自己换行。

上述源文件介绍了两个贯穿全书的重要约定。首先，文件的开始是一行包含文件名称（ex1.c）的注释。如果源文件包含多个函数，那么每个函数名需要出现在该注释行中。如果知道文件的名称，可以帮助我们在 Xinu 的机器可读副本中定位这些文件。其次，文件中需要有一个注释块来标识程序的开始（main）。如果文件中的每个方法前面都有一个注释块，那么就可以很容易地定位这些方法。

1.7 其他章节概要

本书其余章节的内容是根据操作系统的设计层次进行组织的，该设计遵循如图 1-1 所示的多层次

的组织方式。第 2 章介绍并发编程和操作系统所提供的服务。接下来的章节顺序大致与设计和构建的操作系统顺序相同：从最内层向最外层的顺序。每章介绍了一个层次在系统中所扮演的角色，介绍其中新的抽象并解释说明源代码中的细节。总之，这些章节描述了一个完整的、可工作的系统，同时解释了在这个简洁优雅的设计中不同的组件是如何组织起来的。

虽然自底向上的方法开始看起来有些别扭，但它展示了操作系统设计师是如何建立操作系统的。系统的整体结构将在第 9 章开始变得明确。在第 15 章结束后，读者将会理解一个能够支持并发程序的最小内核。第 20 章介绍系统的远程文件访问。到第 23 章时，整个设计将包含一套完整的操作系统功能。

1.8　观点

为什么要学习操作系统？虽然商业系统广泛使用，但只有相对较少的程序员编写操作系统代码。然而，即使是在小型嵌入式系统中，应用程序也要运行在操作系统之上并使用它所提供的服务。因此，了解操作系统内部如何工作能够帮助程序员领会并发处理的精妙并在使用有关系统服务时做出明智的选择。 8

1.9　总结

操作系统并不是一种语言、编译器、窗口系统、浏览器、命令解释程序，或者程序库。由于大多数应用程序都要使用操作系统的服务，所以程序员需要了解操作系统中的原则。在可编程电子设备上工作的程序员需要了解操作系统的设计。

本书采取了一个切实可行的方法，以 Xinu 系统为例来阐述操作系统概念，而不是抽象地描述组件和操作系统的特性。虽然 Xinu 小而优雅，但它并不是一个玩具——它已在商业产品获得使用。Xinu 遵循了多层次的设计原则，在这种设计中系统的软件组件被组织成 8 个概念层次。从原始的硬件开始到一个可工作的操作系统，本书对操作系统的每个层次都进行了详细说明。

练习

1.1　操作系统是否应该使硬件设施对应用程序可用？为什么是或者不是？

1.2　举例说明使用一个真实的操作系统有什么优势。

1.3　构成操作系统的 8 个重要组件是什么？

1.4　在 Xinu 的多层次结构中，文件系统功能是否依赖于进程管理功能？进程管理功能是否依赖于文件系统功能？解释原因。

1.5　探索你最喜欢的操作系统的系统调用，并编写一个程序来使用它们。

1.6　各种编程语言在设计时结合了一些操作系统中的概念。例如进程和进程同步原语。选择一种编程语言，列出该语言提供的功能清单。

1.7　搜索网络，列出还在使用的主流的商用操作系统。

1.8　比较 Linux 和新版的 Windows 操作系统的功能。其中一个系统支持的功能，在另一个系统中也支持吗？

1.9　应用程序可用的操作系统功能集合称为应用程序接口（API）或者系统调用接口。选择两个操作系统，计算每个操作系统提供的功能数，并进行比较。

1.10　在 1.9 题的基础上，找出一个系统有而另一个系统没有的功能。描述这些功能的目的和重要性。

1.11　操作系统有多大？选择一个系统，找出其内核的代码行数。 9

并发执行与操作系统服务

有一篇有关IBM PC上的新型操作系统的文章是这样说的：真正的并发是当你唤起并使用另一个程序时，当前的程序实际上仍然在执行。这种能力可能比常人所能认识到的更加惊异，但对普通人却用处甚少。你运行过多少个执行时间超过几秒的程序呢？

——《纽约时报》，1989年4月25日

2.1 引言

本章讨论操作系统为应用程序提供的并发编程环境。首先描述了并发执行的模型，并说明了为什么并发执行的应用程序需要协调和同步的机制。然后介绍了进程和信号量等基本概念，并解释了应用程序如何使用这些概念。

本章并不抽象空洞地叙述操作系统，而是引用Xinu系统上的具体例子来阐述诸如并发、同步等概念。本章包含了一些适俗易懂的程序，尽管它们都只有寥寥几行代码，却能体现并发执行的本质。以
11 后的章节将就一个操作系统如何实现上述每种概念展开详细讨论。

2.2 多活动的编程模型

现在，即便是小型计算设备，都具备同时处理多项任务的能力。比如，当手机接通了语音来电后，仍然可以显示时间，监听其他来电，或者允许用户调整音量。更复杂的计算机系统允许一个用户运行多个同时执行的应用程序。但问题也就产生了：在这类系统中，应当如何组织软件？有三种基本方案可供使用：

- 同步事件循环。
- 异步事件处理程序。
- 并发执行。

同步事件循环（synchronous event loop）：术语同步指的是经过协调的多个事件。同步事件循环使用一个大循环来处理事件协调。在该循环的给定迭代期间，代码检查每一个可能的活动并调用恰当的处理程序。因此，代码结构大致如下：

```
while (1) {  /* synchronous loop runs forever */
        Update time-of-day clock;
        if (screen timeout has expired) {
                turn off the screen;
        }
        if (volume button is being pushed) {
                adjust volume;
        }
        if (text message has arrived) {
                Display notification for user;
        }
        ...
}
```

异步事件处理程序（asynchronous event handler）：第二种方案用于这样一类系统，其硬件在配置后可为每一个事件调用一个处理程序（handler）。比如，调整音量的代码可能放置在内存中位置100，可以把硬件配置为：当按下“音量”按钮时，控制转移至位置100。类似地，还可把硬件配置为：当一条文本消息到达时，控制转移至位置200，以此类推。程序员需要为每一个事件分别写一段独立的代码，并利用全局变量来协调它们的交互。例如，当用户按下“静音”按钮后，与静音事件相关联的代码就会将音频关掉并将该状态记录至一个全局变量中。此后，当一个用户想要调整音量时，与“音
12 量”按钮相关联的代码会检查这个全局变量，打开音频，并更改这个全局变量来反映音频打开的状态。

并发执行（concurrent execution）用于组织多项活动的第二种架构，也是最为重要的。在这种架构下，软件被组织为一组并发运行的程序。该模型有时称为运行至结束（run-to-completion）模型，因为每一项计算似乎都能一直运行下去，直到它自己选择结束为止。从程序员角度看，并发执行是令人愉悦的。与同步或异步事件相比，并发执行更为强大、更易于理解、更不易出错。

接下来的小节将简述操作系统为并发性提供的必要支持和并发模型的特征。以后的章节还将深入剖析支持并发编程模型的潜在操作系统机制和相关函数。

2.3　操作系统服务

操作系统提供的主要服务是什么？尽管各个操作系统的具体情况千变万化，但大多数系统都提供一组共通的基本服务。这些服务（以及讨论它们的章节）如下：

- 执行并发支持（第 5、6 章）
- 进程同步设施（第 7 章）
- 进程间通信机制（第 8 章）
- 应用程序间的保护（第 9、10 章）
- 地址空间和虚拟内存的管理（第 10 章）
- I/O 设备的高层接口（第 12～14 章）
- 网络通信（第 17 章）
- 文件系统和文件访问设施（第 19～21 章）

并发执行在操作系统中处于核心地位，我们将看到并发性的确会影响操作系统中的每一部分代码。因此，我们将首先研究操作系统为并发所提供的设施，然后借用并发性来说明应用程序如何调用操作系统服务。

2.4　并发处理的概念和术语

传统程序是串行的（sequential），因为程序员可以想象计算机是逐条语句地执行这段代码。在任何时刻，机器只能执行一条语句。操作系统支持一种扩展的计算概念，称为并发处理（concurrent processing）。并发处理意味着在同一个时间可以进行多项计算[⊖]。

许多与并发处理相关的问题随之而来。很容易想象：N 个独立的程序正同时在 N 个处理器或 N 个核上执行，但要想象这组独立的计算正在一台处理单元少于 N 个的计算机上进行却并不容易。即使计算机有一个核，并发计算是否可行？如果多项计算同时进行，系统又该如何防止一个程序与其他程序发生干扰？程序间如何协调以保证在给定时间为一个输入输出设备的控制权只为一个程序所拥有？ 13

尽管多数 CPU 确实已经蕴含了一定程度的并行性，但最显而易见的并发形式——多个独立的应用程序同时执行，却仍是一个大幻觉。为创造出并发执行的错觉，操作系统使用了一种称为多道编程（multiprogramming）的技巧——操作系统在多个程序间切换可用的处理器，允许一个处理器用仅有的几毫秒执行一个程序，随后转而处理别的程序。从人的角度看，程序似乎是在并发地执行。多道编程构成了大多数操作系统的基础。唯一的例外是那些操作基础设备的系统，如简化的电视遥控器和安全至上的嵌入式系统，又如飞行器上的航空电子设备和医疗设备控制器，它们使用一个同步事件循环来确保苛刻的时间限制得到绝对的满足。

支持多道编程的系统可以分成两大类：分时、实时。

分时（timesharing）　分时系统给予所有计算以相同的优先级，并且允许计算在任何时间开始或终止。因为分时系统允许动态地创建一项计算，所以这些系统在面向人类用户的计算机上颇受欢迎。分时系统允许用户在使用浏览器浏览网页的同时，运行一个电子邮件应用程序，或者运行一个后台应用程序播放音乐。分时系统最主要的特征是：一项计算所获得的处理器时间与这个系统的负载呈反比例

⊖　这里的“计算”指的是一项计算任务，下同。——译者注

关系——如果有 N 项计算正在进行，那么每项计算大约会获得可用 CPU 周期的 $1/N$。因此，随着越来越多计算的出现，每项计算的速率将会不可避免地降低。

实时（real-time） 因为实时系统的设计要求就是要满足（可预测的）性能约束，所以不会同等地对待所有计算。相反，实时系统为每项计算分配一个优先级，并且需要非常小心谨慎地调度处理器以确保每项计算满足所要求的执行计划。实时系统最主要的特征是：它总是将 CPU 分配给最高优先级的任务，即使其他任务正在等待。比如，通过提高语音传输任务的优先级，手机中的实时系统可以确保会话不被中断，即使此时用户运行了天气查看程序或游戏程序。

多道编程系统的设计者提出了大量的术语来表示一项计算，包括进程（process）、任务（task）、作业（job）和控制线程（thread of control）。术语进程或作业经常意味着一项自包含的计算，与其他计算相互独立。一个进程通常占用内存中一块单独的区域，并且操作系统会阻止一个进程访问一块已经分配给另一个进程的内存。术语任务指的是一个静态声明过程，即程序员使用一种编程语言以类似函数声明的方式来声明一个进程。术语线程指的是一类过程，它们与别的线程共享同一个地址空间。共享内存意味着一组线程的成员可以高效地交换信息。早期的科技文献中常用术语过程来表示通常意义下的并发执行。UNIX 操作系统普及了这样一种观念：每个进程都占用了一块独立的地址空间。Mach 系统引入了一种二级并发编程方案，其中操作系统允许用户创建一个或多个进程，每个进程都运行于独立的内存区域中，并进一步允许用户在一个给定进程中创建多个控制线程。Linux 遵循的是 Mach 模型。我们使用首字母大写的单词 Process 来表示 Linux 风格的进程。

[14]

由于 Xinu 是为嵌入式环境设计的，所以它允许进程共享同一个地址空间。确切地讲，Xinu 的过程遵循线程模型。然而，因为术语进程已经广为接受，所以本书中不失一般性地将其术语为一项并发计算。

2.5 节将通过对几个应用实例的研究来帮助读者区分并发执行和串行执行。正如我们将看到的，这种差异在操作系统设计中扮演了核心角色——操作系统的每一部分都必须以支持并发执行为目标进行构建。

2.5 串行程序和并发程序的区别

当程序员创建一个传统的（串行的）程序时，他会想象一个处理器在没有中断和干扰的情况下按部就班地执行这个程序。然而编写并发程序的代码时，程序员则必须采取一种完全不同的思维：想象多项计算同时执行。操作系统的内部代码就是适应并发性极好的例子。在任意给定时刻，可能有多个进程在执行。最简单的情况下，每个进程执行的应用程序代码不会被其他过程同时执行。然而，操作系统的设计者必须事先计划好这样的情形：多个进程同时调用同一个操作系统函数，甚至执行同一条指令。更复杂的问题在于，操作系统可能会在任意时间进行进程切换。在多道编程系统中，相对计算速度无法保证。

要设计出能够在并发环境下正确执行的代码不失为一项严峻的智力挑战，因为程序员必须确保无论执行什么操作系统代码或者以何种顺序执行，所有的进程都能够相互合作。我们将会看到并发执行的观念是如何影响操作系统的每一行代码。

为了理解并发环境下的应用程序（如何工作），考虑 Xinu 模型。当 Xinu 启动时，它创建一个过程，并开始执行主程序。这个最初的进程能够继续独立执行，或者创建新的进程。当创建一个新进程时，原来的进程仍继续执行，并且两者并发地执行。无论原进程还是新进程，都可以再创建其他的并发执行的进程。

[15]

比如，考虑一个创建两个过程的并发应用程序。每个进程通过控制台串行设备发送字符：第一个进程发送字母 A，而第二个发送字母 B。文件 ex2.c 包含了源代码。由一个主程序，两个函数 sndA 和 sndB 组成。

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);
```

```
/*------------------------------------------------------------------------
 * main  --  example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void	main(void)
{
	resume( create(sndA, 1024, 20, "process 1", 0) );
	resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  --  repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void	sndA(void)
{
	while( 1 )
		putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  --  repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void	sndB(void)
{
	while( 1 )
		putc(CONSOLE, 'B');
}
```

16

在这段代码中，主程序从不直接调用另外两个函数。相反，主程序调用了两个操作系统函数，create 和 resume。每一次调用 create 都会创建一个新的进程，并从第一个参数指定的地址开始执行指令。在这个例子中，对 create 的第一次调用传递了函数 sndA 的地址，第二次调用传递了函数 sndB 的地址[⊖]。因此，这段代码创建了一个进程来执行 sndA 和另一个进程来执行 sndB。create 建立了一个准备执行但暂时挂起的进程，并返回一个称为进程标识符（process identifier）或进程 ID（process ID）的整数值。操作系统使用进程 ID 来辨别新创建的进程，应用程序使用进程 ID 来引用该进程。在这个例子中，主程序将 create 返回的 ID 作为参数传递给了 resume。resume 启动了（解除挂起）这个进程，允许该进程开始执行。普通函数调用与进程创建（系统调用）的区别在于：

> 普通函数调用在被调用的函数完成之前不会返回，而进程创建函数 create 和 resume 在启动一个新进程后立即返回，这将使已有的进程与新进程并发地执行。

在 Xinu 中，所有进程都是并发执行的，即一个给定进程是独立于其他进程而持续执行的，除非程序员显式控制进程间的交互。在这个例子中，第一个新进程执行函数 sndA 中的代码，不断发送字母 A；而第二个新进程执行函数 sndB 中的代码，不断发送字母 B。由于进程是并发执行的，所以输出结果是许多 A 和 B 的混合。

那么主程序将发生什么？记住在一个操作系统中，每一项计算都对应于一个进程。因此，我们应该问："执行主程序的进程发生了什么？"因为控制已经到达了主程序的末尾，所以执行主程序的进程在第二次调用 resume 以后将会退出。它的退出不会影响到新创建的进程——它们将继续不停地发送 A 和 B。后面小节将对进程终止做详细讨论。

2.6 多进程共享同一段代码

文件 ex2.c 中的例子描述的是每个进程执行独立的函数。然而，还有可能多个进程执行同一个函

⊖ create 的其他参数指定了所需要的栈空间、调度优先级、进程的名称、传递给该进程的参数个数，以及传递给该进程的参数值（如果有的话）。我们将在以后讲到这些细节。

数。令多个进程共享代码对于内存较小的嵌入式系统非常重要。文件 ex3.c 中的程序是一个多进程共
[17] 享代码的实例。

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main --- example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  --  output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch                      /* character to emit continuously */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

正如前面的例子所示，一个进程开始执行主程序。这个进程两次调用 create 来启动两个新进程，它们均执行函数 sndch 的代码。create 调用的最后两个参数指引了 create 将传递一个参数给新创建的进程以及该参数的值。因此，第一个进程收到字符 A 作为参数，而第二个进程收到字符 B。

尽管它们执行的是同一份代码，但是两个进程能够在相互不影响的前提下并发地执行。特别地，每个进程都拥有一份自己的参数和局部变量副本。因此，一个进程产生 A，而另一个进程产生 B。这个问题的关键点在于：

> 一个程序向可被单个控制进程执行的代码组成。相比之下，并发进程并不是唯一地关联
[18] 到一段代码，多个进程可以同时执行同一份代码。

这些例子暗示了设计一个操作系统所涉及的诸多困难。设计者不仅必须保证每一段代码在独立运行时的正确性，还要保证多个进程可以在没有互相干扰的情况下并发地执行一段给定的代码。

尽管进程间可以共享代码和全局变量，但每个进程必须有一份私有的局部变量副本。为了理解这样做的原因，可以考虑如果所有进程共享了每个变量将会造成何等的混乱。举例来说，假设两个进程尝试使用一个共享变量作为 for 循环的循环变量（index），一个进程可能会在另一个进程执行循环体期间修改其值。为避免这种干扰，操作系统为每个进程创建了一组独立的局部变量。

函数 create 还为每个进程分配了一组独立的参数，正如文件 ex3.c 中的实例所展示的。在 create 调用中，最后两个参数分别指明了后面的参数个数（实例中为 1），以及操作系统传递给新创建进程的值。在这段代码中，第一个新进程以字符 A 作为参数，进程开始执行时，形式参数 ch 设为 A。第二个新进程开始执行时，ch 设为 B。因此，输出中包含的是两种字母的混合。这个例子指出了串行编程与并发编程模型的一个重大区别。

> 局部变量、函数参数以及函数调用栈的存储空间是与执行一个函数的进程相关联，而不是与该函数的代码相关联。

重点在于：操作系统必须为每个进程分配额外的存储空间，即使一个进程与其他进程共享了同样

的代码。因此，可用的内存数限制了能够创建的进程数量。

2.7　进程退出与进程终止

文件 ex3.c 中实例中的并发程序由三个进程组成：初始进程和两个通过系统调用 create 启动的进程。前文提到当控制到达主程序代码末尾时，初始进程停止执行。我们使用术语进程退出（process exit）来描述这种情形。每个进程从一个函数的开头开始执行。一个进程既可以因为到达函数的末尾而退出，也可以通过在它的初始函数中执行一条返回语句而退出。一旦一个进程退出了，它就从系统中消失了。正在进行中的计算简单地少了一项。

不要把进程退出（系统调用）与普通函数调用或者返回函数调用相混淆。正如一个串行程序，每个进程都有自己的函数调用栈。无论何时执行一个调用，被调用函数的活动记录（activation record）都会压入栈中。无论何时调用返回，该函数的活动记录都会从栈中弹出。当一个进程把最后一条活动记录（对应于进程启动时最顶层的函数）从栈中弹出时，该进程退出。 [19]

系统例程 kill 提供了一种终止（terminate）进程的机制，无需等待进程退出。从某种意义上说，kill 是 create 的逆操作——kill 接受一个进程 ID 作为参数，并立即将该进程移除。可以在任意时刻、任意函数嵌套层将一个进程终止。当进程终止时，它将立即停止执行，所有分配给该进程的局部变量均消失。事实上，该进程的整个函数调用栈都被移除了。

一个进程可以通过终止自己来退出，这同终止别的进程一样容易。若要这样做，进程可以通过系统调用 getpid 获得自己的进程 ID，然后调用 kill 来请求终止：

```
kill( getpid() );
```

如果当前进程这样终止，那么对 kill 的调用将永远不会返回，因为调用它的进程已经退出了。

2.8　共享内存、竞争条件和同步

在 Xinu 中，每个进程都有它自己的局部变量、函数参数和函数调用副本。但所有进程共享一组全局（外部）变量。数据共享有时候很方便，但也很危险，特别是对那些不习惯写并发程序的程序员。比如，考虑两个并发进程，两者均递增一个共享整数 n。就底层硬件来说，递增一个整数需要三个步骤：

- 把内存中变量 n 的值载入一个寄存器。
- 递增该寄存器中的值。
- 将寄存器中的值写回内存中的变量 n。

因为操作系统可以选择在任意时刻从一个进程切换到另一个进程，所以可能存在潜在的*竞争条件*（race condition）：两个进程同时尝试递增 n。进程 1 可能会首先启动并将 n 的值载入一个寄存器。但就在那一刻，操作系统切换到了进程 2，它载入了 n，递增寄存器，并写回结果。不幸的是，当操作系统切换回进程 1 继续执行时，该寄存器中存放的还是 n 原来的值。进程 1 递增了原来的 n 值并将结果写回内存，覆盖了进程 2 放入内存的值。

为明白共享是如何进行的，考虑文件 ex4.c 中的代码。该文件包含两个进程的代码，它们通过一个共享整数 n[⊖] 进行通信。一个进程不断递增这个整数，而另一个进程不断打印出它的值。 [20]

```
/* ex4.c - main, produce, consume */

#include <xinu.h>

void    produce(void), consume(void);

int32   n = 0;          /* external variables are shared by all processes */
```

⊖ 代码中使用类型名称 int32 来说明变量 n 是一个 32 位整数，以后将会解释类型命名的传统做法。

```
/*-------------------------------------------------------------------------
 *  main  --  example of unsynchronized producer and consumer processes
 *-------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(consume, 1024, 20, "cons", 0) );
        resume( create(produce, 1024, 20, "prod", 0) );
}

/*-------------------------------------------------------------------------
 * produce  --  increment n 2000 times and exit
 *-------------------------------------------------------------------------
 */
void    produce(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                n++;
}

/*-------------------------------------------------------------------------
 * consume  --  print n 2000 times and exit
 *-------------------------------------------------------------------------
 */
void    consume(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

在上述代码中，全局变量 n 是一个初始值为 0 的共享整数。执行 produce 的进程迭代 2000 次，递增 n。我们将这个进程称为生产者（producer）。执行 consume 的进程同样迭代 2000 次，它用十进制方式显示 n 的值。我们将该进程称为消费者（consumer）。

[21]

运行文件 e04.c——它的输出可能会使你惊讶不已。大多数程序员猜想消费者将至少打印出一些，也许全部 0～2000 的值，可是它没有。在一次典型的运行中，n 在前几行中值为 0。在这之后，它的值变为 2000[⊖]。即使这两个进程并发运行，但它们并不要求在每次迭代期间获得相同数量的 CPU 时间。消费者进程必须对输出进行格式化并与一行输出，这是一个需要数百条机器指令的操作。尽管格式化操作的代价十分昂贵，但它不能控制时序。输出操作可以。消费者很快填满了可用的输出缓冲区，然后必须等待输出设备以便把字符发送到控制台，然后它才能继续运行。当消费者等待时，生产者运行。由于生产者每次迭代只需执行少量的机器指令，所以它甚至可以在控制台设备发送完一行字符之前，就运行完整个循环并退出。当消费者再次恢复执行时，它发现 n 的值为 2000。

通过独立的进程进行数据的生产与消费是十分常见的。问题是：程序员应该如何同步生产者和消费者进程，从而使消费者可以接收每个生产出来的数据值？显然，生产者必须等待消费者访问了当前数据项后才能产生另一个数据项。同样，消费者必须等待生产者制造完下一个数据项。为使这两个进程能够正确协作，必须仔细地设计出一种同步机制。关键重要的约束是：

> 在一个并发编程系统中，进程不应该在等待其他进程时仍然占用 CPU。

进程在等待另一个进程时仍然执行指令，可以认为它陷入了忙等待（busy waiting）。为了理解禁止

⊖ 这个例子假设运行在 32 位体系结构上，每一次操作都会影响整个 32 位整数。当运行在 8 位体系结构上时，n 的某些字节可能会先于其他字节得到更新。

正等待的原因，不妨考虑如下实现。如果一个进程在等待时使用 CPU，那么这个 CPU 就不能执行别的进程。在最好的情况下，计算只是被不必要地推迟了；在最坏的情况下，正在等待的进程将会耗尽单 CPU 处理器的所有可用的 CPU 时间，并永久等待下去。

许多操作系统都包括了多个协调函数，以供应用程序使用以避免忙等待。Xinu 提供了一个信号量（semaphore）抽象——该系统提供了一组系统调用，允许应用程序操作并动态创建信号量。一个信号量系统由一个整数值（在信号量创建时初始化）和一组（零个或多个在该信号量上等待的）进程构成。系统调用 wait 递减信号量的值，当结果为负时，调用 wait 的进程将被加入到等待进程集合中。系统调用 signal 执行相反的操作，将信号量递增并允许等待进程之一继续运行（如果有的话）。为实现同步，生产者和消费者需要两个信号量：一个用于使消费者等待，另一个用于使生产者等待。在 Xinu 中，信号量可以通过系统调用 semcreate 动态创建，它接收给定的初始计数作为参数，并返回一个与该信号量相关联的整数标识符。 22

考虑文件 ex5.c 中的例子。主进程创建了两个信号量，consumed 和 produced，并将它们作为参数传递给它创建的进程。因为信号量 produced 的初始计数为 1，所以 cons2 中第一次调用 wait 将不会阻塞。因此，消费者可以自由地打印出 n 的初始值。然而，信号量 consumed 的初始计数为 0，因此在 prod2 中第一次调用 wait 会阻塞。实际上，生产者会在递增 n 之前等待信号量 consumed 以确保消费者已经打印了当前的 n。当执行这个例子时，生产者和消费者进行协调，消费者会打印从 0～1999 的所有 n 值。

```
/* ex5.c - main, prod2, cons2 */

#include <xinu.h>

void    prod2(sid32, sid32), cons2(sid32, sid32);

int32   n = 0;                  /* n assigned an initial value of zero  */

/*------------------------------------------------------------------------
 *  main  --  producer and consumer processes synchronized with semaphores
 *------------------------------------------------------------------------
 */
void    main(void)
{
        sid32   produced, consumed;

        consumed = semcreate(0);
        produced = semcreate(1);
        resume( create(cons2, 1024, 20, "cons", 2, consumed, produced) );
        resume( create(prod2, 1024, 20, "prod", 2, consumed, produced) );
}

/*------------------------------------------------------------------------
 * prod2  --  increment n 2000 times, waiting for it to be consumed
 *------------------------------------------------------------------------
 */
void    prod2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;
        for( i=1 ; i<=2000 ; i++ ) {
                wait(consumed);
                n++;
                signal(produced);
        }
}
```

```
/*------------------------------------------------------------------------
 * cons2  --  print a 2000 times, waiting for it to be produced
 *------------------------------------------------------------------------
 */
void    cons2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ ) {
                wait(produced);
                printf("a is %d \n", a);
                signal(consumed);
        }
}
```

2.9 信号量与互斥

信号量还提供了另一种重要用途，互斥（mutual exclusion）。两个或更多进程在操作时需要互斥，以保证在某一时刻它们中只有一个进程能够得到对一个共享资源的访问权。比如，假设有两个正在执行的进程，每个都需要向一个共享链表中插入数据项。如果它们并发地访问这个链表，指针就有可能被错误地设置。生产者-消费者同步无法解决这个问题，因为这两个进程并不是交替地访问。相反，需要有一种机制允许任一进程能够在任何时间访问这个链表，同时必须确保互斥。即一个进程在另一个进程完成前将一直等待。

为了对链表这样的共享资源提供互斥保护，进程创建一个初始计数为1的信号量。在访问这个共享资源之前，一个进程在该信号量上调用 wait，并在完成访问之后调用 signal。可以把 wait 和 signal 调用分别放置在相关过程（用于执行更新操作）的开头和末尾，或者放置在访问共享资源的那几行代码附近。使用术语临界区（critical section）来表示那些不能被多个进程同时执行的代码。

23
;
24

例如，exb.c 中定义了一个函数，这个函数给一个由多个进程共享的数组增加一个元素，这个元素的值为 item。这段代码的临界区为下面一行：

```
shared[n++] = item;
```

这一行中引用了数组并且增加了元素的个数，因此互斥代码只需要写在这一行的前后。由于这个例子中临界区在函数 additem 之中，所以互斥所需要的 wait 和 signal 函数调用放在这个函数的开始和结尾处。

additem 中的代码在访问数组之前就调用信号量 mutex 上的 wait 函数，当访问结束之后，调用这个信号量上的 signal 函数外。除了这个函数外，程序中还有三个全局变量的声明：数组 ary，数组的索引 n 和用于实现互斥的信号量的标识符 mutex。

```
/* exb.c - additem */

#include <xinu.h>

sid32   mutex;                  /* assume initialised with semcreate    */
int32   shared[100];            /* an array shared by many processes    */
int32   n = 0;                  /* count of items in the array          */

/*------------------------------------------------------------------------
 * additem  --  obtain exclusive access to array ary and add an item to it
 *------------------------------------------------------------------------
 */
void    additem(
          int32         item    /* item to add to array ary             */
```

```
        }
{
        wait(mutex);
        shared[n++] = item;
        signal(mutex);
}
```

这段代码假设了全局变量 mutex 在 additem 调用之前就已经是一个合法的信号量标识符。也就是说，在初始化的时候执行了下面的函数：

```
mutex = screate(1);
```

ex6.c 展示了串行程序和并发程序最后一个不同点。在串行程序中，一个函数对一个数据结构的访问是与其他函数隔离开的，程序员可以通过封装变量的修改操作方法来保证系统的安全性——只需要 25
检查很小一部分代码的正确性就可以保证数据结构的正确性，因为程序中其他的部分不会破坏数据的一致性。而在并发环境下，仅仅把数据修改操作隔离起来是不够的，程序员必须确保数据修改操作是互斥的，否则其他进程很可能在同一时刻执行同一个函数，从而影响数据的一致性。

2.10 Xinu 中的类型命名方法

上述代码中的数据声明是本书中的范例。比如，信号量标识符的类型名称为 sid32，本节来解释这样命名的原因。

C 语言编程中有两个很重要的问题。什么时候应该定义新的类型名称？怎样选定一个类型名称？要想回答这两个问题，必须首先清楚类型名称在语言概念中的两个角色。

- 空间：一个类型定义了存储一个变量所需要的存储空间以及什么样的数值可以赋值给这个变量。
- 用途：一个类型定义了变量的抽象含义，可以帮助程序员了解如何使用这个类型的变量。

空间 在嵌入式系统中，变量的存储空间特别重要，因为程序员设计的数据结构必须保证内存的高效使用。此外，如果在设计变量的存储空间的大小时没有考虑底层硬件的实现细节，可能会导致意想不到的性能开销（比如，大整数的算数运算可能需要多步才能完成）。不幸的是，C 语言中预定义的数据类型并没有明确数据存储空间的大小，比如 int、short 和 long，而这些数据类型的存储空间的实际大小是由底层的计算机架构来决定的。比如，在一台计算机上一个 long 类型的整数可能需要 32 位的存储空间，而在另外一台计算机上就可能需要 64 位的存储空间。因此为了保证自己定义的变量有精确的存储空间大小，程序员必须定义和使用这样的类型名称 int32 来声明数据的大小。

用途 使用类型最初的目的是定义一个变量的使用目的（也就是说，告诉人们这个变量是用来做什么的）。比如，尽管信号量的标识符是一个整数，但是给这个变量一个类似 semaphore 的类型名称，可以让代码的读者能够清楚地认识到这个变量表示的是一个信号量的标识符，并且这个变量只能用在信号量标识符的应用场合（比如，作为一个信号量操作函数的参数）。这样，尽管这个变量存储的是整数，但由于它的类型名称是 semaphore，所以我们不能把这个变量作为一个算数表达式中的临时变量，也不能把它用来存储进程号或者设备号。

头（include）文件让 C 语言中的类型声明变得更加复杂。理论上，每个头文件应该只包含一个模块的类型、常量和变量声明。这样，如果需要寻找进程标识符的类型，就只需要查找那些定义了进程相关元素的头文件。然而，在一个操作系统中，模块之间存在着大量的交叉引用。比如，信号量的头文件引用了进程的头文件，进程头文件也引用了信号量的头文件。

在 Xinu 中，采用了一种新的方法，既可以定义一个类型的空间也能定义一个类型的用途。比如，对于一些 C 语言本身的基本数据类型，如 char、short、int 和 long，在该方法中的定义如图 2-1 所示。

类型	含义
byte	无符号8位变量
bool8	表示布尔变量的8位变量
int16	16位有符号整数
uint16	16位无符号整数
int32	32位有符号整数
uint32	32位无符号整数

图 2-1 Xinu 中整数的基本类型

对于那些与操作系统抽象概念相对应的类型，类型的名称由助记符和数字组成，助记符说明了类型的作用，而数字则表示变量存储空间的大小。因此，一个定义了信号量标识符并且存储空间为32位整数的类型的名称为sid32，一个定义了队列标识符并且存储空间为16为整数的类型的名称为qid16。

为了防止模块之间的交叉引用，引入了一个头文件kernel.h，其中包含所有数据类型的定义，包括图2-1中的数据类型。因此，每个源文件在使用任何类型之前必须引用kernel.h。实际上，kernel.h文件必须在其他模块的头文件之前被引用。为了方便，我们在Xinu中使用了xinu.h头文件，这个头文件以正确的顺序引用了Xinu中所有的头文件，我们只要在我们的源文件中引用这个头文件即可。

2.11 使用Kputc和Kprintf进行操作系统的调试

本章中的例子使用了Xinu的函数putc和printf将整出显示在控制台（CONSOLE）上。尽管当操作系统开发出来并通过测试后，这样的函数可以正常运行，但是在构建或者调试时，一般不使用这些函数，因为它们正确运行的前提是操作系统中很多模块都已经正常运行。那么操作系统的设计者在调试中使用什么呢？

答案是轮询I/O。操作系统的设计者创建一个特殊的I/O函数，这个I/O函数不需要中断就可以工作。仿照UNIX传统的命名方式，我们把这个函数叫做kputc（也就是，putc的操作系统的内核版本）。
27 kputc接收一个字符c作为参数，然后进行以下四个步骤：

- 禁止中断。
- 等待控制台（CONSOLE）的串行设备空闲。
- 向串行设备发送字符c。
- 恢复中断到之前的状态。

因此，当程序员调用kputc时，所有其他的处理过程都暂停运行直到字符显示出来，一旦字符显示出来，其他处理过程再恢复工作。这种机制的核心思想是，操作系统不需要运行就可以使用，因为kputc直接操作底层的硬件。

一旦有了kputc，实现一个格式化输出的函数就很简单了。同样，仿照UNIX传统命名方式，我们把这个函数叫做kprintf。基本上，kprintf和printf的操作基本相同，只是kprintf调用kputc，而printf调用putc。[⊖]

尽管理解轮询I/O的具体细节并不重要，但是操作系统调试的本质就是使用轮询I/O：

> 当需要修改或者扩展操作系统的时候，应该使用kprintf打印调试信息而不使用printf。

2.12 观点

并发处理是操作系统中最强大的抽象之一。它使编程变得简单，并且更不容易出错，同时在很多情况下，并发系统的整体性能比那些手动切换任务的系统更好。正是由于性能上的优势，并发执行很快成为了绝大多数程序设计的首选。

2.13 总结

想要了解操作系统，首先需要了解它提供给应用程序的服务。操作系统提供的不是传统的串行编程环境，而是一种多线程并发执行环境。在我们所介绍的操作系统中，与大多数操作系统相类似，进程可以在操作系统运行的时候创建和终止，多个进程可以同时执行不同的函数，也可以同时执行一个函数。在并发环境中，代表一个进程的是参数的存储，局部变量和函数调用栈而不是运程执行的代码。

进程通过信号量这样的原语进行同步以便协作执行，两种比较常见的协作模式是生产者和消费者
28 模型以及互斥。

⊖ 调试操作系统的代码特别困难，因为禁止中断可以改变一个系统的执行顺序（比如，阻止时钟中断响应），因此，使用kprintf的行数会和预期不同。

练习

2.1 什么是 API？API 是如何定义的？

2.2 多道编程指的是什么？

2.3 列举两种多道编程的类型，并说明它们各自的特点。

2.4 进程、任务和线程，各自有什么特点？

2.5 进程的标识符有什么作用？

2.6 调用函数 X 与调用函数 create 启动进程来执行函数 X 有什么不同？

2.7 ex3.c 中使用了 3 个进程，修改代码使得只用两个进程就得到同样的结果。

2.8 反复测试 ex4.c 中的程序，它每次都打印相同个数的 0 吗？它会打印出不是 0 或者 2000 的值吗？

2.9 在 Xinu 中，全局变量和局部变量有什么区别？

2.10 为什么程序员需要避免忙等待？

2.11 假设有 3 个进程同时调用 ex6.c 中的 additem 函数，解释下它们执行的步骤以及每一步中信号量的值。

2.12 修改 ex5.c 中生产者－消费者的代码，使用一个 15 个空槽的缓冲区，按以下方式同步生产者和消费者：生产者可以生产最多 15 个值，然后进入阻塞，消费者获得缓冲区中所有的值，然后进入阻塞。也就是说，生产者按顺序访问缓冲区，写入整数值 1，2，…，当填满最后一个缓冲槽的时候，生产者返回缓冲区的开始处，然后消费者遍历所有的数值并且把它们打印到控制台。需要几个信号量？

2.13 在 ex5.c 中，信号量 produced 初始化为 1，重写代码时信号量 produced 初始化为 0，并且生产者在开始迭代的时候就释放信号量，这会影响输出吗？

2.14 找到一个你可以访问平台的串口的文档（或者控制台硬件），描述轮询 I/O 函数 kputc() 是如何使用该设备的。

硬件和运行时环境概览

一台机器可以做50个手无人做的工作，没有机器可以做一个非凡的人所做的工作。

——Elbert Hubbard

3.1 引言

因为操作系统需要处理设备、处理器和内存的细节，所以操作系统在设计中不可避免地涉及有关底层硬件能力和功能的知识。本书介绍的系统运行在一个很小的嵌入式硬件系统平台上——E2100L Linksys 无线路由器。我们使用 Linksys 路由器，因为它很简洁，有一个通用的指令集，容易获得并且价格低廉。这个系统小到足以让读者能够理解几乎所有硬件的工作方式，同时又足够的复杂，能够让我们解释操作系统如何工作在一个通用的系统上。最后，程序员可以很方便地使用 E2100L 来下载和运行代码，不需要复杂的硬件环境，也不需要替换 ROM 芯片。

本章介绍 Linksys 硬件，指令处理器、内存和设备的相关知识，解释体系结构、内存地址空间、运行时栈、中断机制和设备寻址等概念。尽管介绍的是与 E2100L 有关的细节，但这些基本的概念可以广泛应用到几乎所有的计算机设备上。

31

3.2 E2100L 的物理和逻辑结构

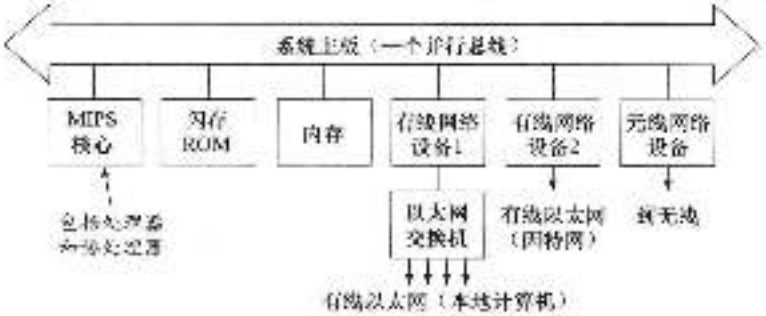
物理上，一个 Linksys 路由器是一个独立的使用分离式电源的小盒子，由于 Linksys 品牌将 E2100L 的市场定位为消费产品，所以能够买到的 E2100L 都是完全组装好的。而且几乎所有的主要功能组件都放在一个超大规模集成电路芯片上，这种方式又称为片上系统（System On a Chip，SoC）。

尽管 E2100L 电路板有一个串行接口的引出，但是它并不提供外部连接。在串口可以使用之前，可以用桌前的串口转换器连接到板上的引脚。串口连接的信息，以及有关如何附加一个串口连接器的指导可以在下面的网址中找到：

http://www.xinu.cs.purdue.edu

逻辑上，E2100L 和大多数通用计算机具有同样的整体架构。片上系统的组件包括处理器、协处理器、内存接口和 I/O 设备接口，其中 I/O 设备主要用于连接有线网络和无线网络设备。

有线网络接口连接到一个具有四个 RJ-45 端口的控制器上，这个控制器可以看做一个集线器，它通过 RJ-45 来连接本地计算机[1]；另一个有线网络接口用于连接网桥间，它只有一个 RJ-45 端口。系统主板，包括数内部总线，提供了系统内部组件相互交互的机制。图 3-1 展示了系统的逻辑结构。



32

图 3-1 E2100L 主要组件的逻辑结构

[1] 芯片上的组件可以支持比板子实际提供的更多的接口。本章和全书的余下章节所指的是默认的硬件配置，在系统重启后，如果没有发现额外的串口初始化信息，那么使用的就是默认的配置。

3.3 节介绍 E2100L 中有关操作系统的特性。从现在开始，我们关注每个组件的整体设计以及组件是如何相互配合的。后面的章节将讨论其他的细节，解释操作系统如何与硬件进行交互，并提供相关例子。

3.3　处理器结构和寄存器

和很多嵌入式系统相类似，E2100L 使用了 RISC[⊖] 处理器。RISC 处理器实现的是 MIPS 指令集。除去一些特殊情况外，操作系统不需要关注指令集，因为编译器负责生成必要的代码。

处理器包括 32 个通用寄存器，每个寄存器有 32 位长并且可以存储一个整数、一个地址或者 4 个 8 位字符。和大多数 RISC 处理器一样，很多 MIPS 操作把寄存器作为参数，将运算结果放在寄存器中。尽管这些寄存器是通用的，但每个寄存器会被编译器指派不同的用途。比如，一个寄存器用做栈指针，一旦发生函数调用，需要给某个活动记录分配空间，栈指针的值就发生变化。图 3-2 列出了寄存器的名字和它们的含义。

名称	含义
0	总是0
AT	汇编器临时变量（保留给汇编器使用）
V0和V1	函数调用的返回值
A0 - A3	参数寄存器，用来存放函数调用时的前四个实参
T0 - T9	函数调用前后的临时变量（非保留）
S0 - S9	函数调用前后的保存变量（保留的）
K0和K1	中断硬件使用的内核寄存器
SP	栈指针（栈向下生长）
RA	函数调用结束的返回地址

图 3-2　通用寄存器及其作用

协处理器包含一组特殊目的的寄存器，可以用来支持很多额外的功能。比如，因为 RISC 处理器不能在一个时钟周期内完成除法运算，所以协处理器用来处理 64 位除法。因此协处理器有一对寄存器用来存储 64 位的值，一个寄存器存放 32 位，另一个寄存器存放高 32 位。类似地，协处理器里还有存放当前中断屏蔽字（判断是否允许中断）的寄存器、存放中断或异常返回地址的寄存器、存放调试信息的寄存器。第 12 章将说明操作系统如何使用特殊的协处理器寄存器。

3.4　总线操作：获取 - 存储范式

总线，又称为系统主板，它为处理器和其他组件之间提供了基本的通信路径。其他组件包括内存、I/O 设备以及接口控制器。和大多数的计算机系统总线一样，系统主板使用获取 - 存储范式（fetch-store paradigm）。比如，当处理器需要访问内存的时候，它在总线上放置一个地址，然后发送一个获取请求去获得对应的值。当内存硬件接收到请求时，它在内存中查找对应的地址，将数据值放在总线上，然后告知处理器值已经准备好。类似地，如果要在内存中存储一个值，处理器需要在总线上放置一个地址和一个值，然后发送一个存储请求，内存硬件接收到请求后，提取要存储的值并且将其存放在地址指定的内存位置。总线硬件负责处理获取 - 存储范式中的细节问题，包括通知处理器或者其他使用总线来进行交流或控制的组件。我们可以看到，操作系统在不了解这些细节的情况下，就可以使用总线。

系统使用内存映射 I/O（就是说，每个 I/O 设备都分配了总线空间上的一些地址），处理器使用与访问内存相同的获取 - 存储范式来访问 I/O 设备。这样，访问 I/O 设备就好像访问数据一样，首先处理器计算出设备的地址，然后访问设备，在访问的过程中，处理器或者向该地址存放一个值，或者从该地址中获得一个值放入寄存器中。

3.5　直接内存访问

E2001L 上的一些 I/O 设备提供了直接内存访问（Direct Memory Access，DMA）功能，这些设备的硬件可以使用总线与内存直接通信。DMA 的目的是让 I/O 操作更快，因为有了 DMA，就不需要频繁地

⊖ RISC 表示 Reduced Instruction Set Computer，精简指令集计算机。

中断 CPU，也不需要 CPU 控制数据传输。相反，在 DMA 中，处理器发给 I/O 设备一系列操作，随后
[34] 设备依次执行这些操作，这样就可以让处理器在设备执行操作的过程中继续执行代码了。

例如，考虑 DMA 在网络设备上的应用。为了接收数据包，操作系统在内存中分配了一块缓冲区，然后启动网络设备。当数据包到达时，设备硬件直接把数据包的内容放到位于内存中的缓冲区里，并中断处理器。为了发送数据，操作系统将数据包放在内存缓冲区中，然后启动该网络设备。该网络设备随后直接从内存缓冲区中获得数据并将数据包发向网络。

除了单个数据包的传输外，E2100L 的 DMA 硬件允许处理器请求多个操作。实际上，处理器创建了要发送的数据包的链表和存放到达数据包的缓冲区的链表。网络接口硬件使用这些链表发送和接收数据，不需要处理器在每个操作完成之后重新启动设备。只要处理器接收数据包的速度大于数据包到达的速度，就可以将新的缓冲区放入链表中，网络硬件就可以继续接收数据包。同样，只要处理器持续地产生数据包并把它们加入到链表中，网络硬件就会持续地发送它们。接下来的章节会解释更多有关 DMA 的细节，并且给出操作系统中的硬件驱动如何分配 I/O 缓冲区和控制 DMA 操作的例子。



图 3-3 地址空间分为用户空间和内核空间，内核空间又细分为段

3.6 总线地址空间

系统主板使用 32 位的总线地址空间，地址空间的范围从 0x00000000～0xFFFFFFFF。地址空间中的有些地址对应的是内存，有些地址对应的是闪存，而另一些是 I/O 设备。为了适应操作系统需要的内存保护和虚拟内存机制，将地址空间分为两部分：低端地址（地址 0x00000000 0x7FFFFFFF）为用户空间，而高端地址（地址 0x80000000 0xFFFFFFFF）为内核空间，内核空间又进一步细分为段。图 3-3 给出了这种组织的示意图。

E2100L 中的内存以 8 位字节为最小的可寻址单元。在 C 语言中我们用术语字符来代替字节，因为每个字节都可以存放一个 ASCII 字节。尽管 32 位总线可以寻址的空间是 4G 字节，但 E2100L 没有 4G 字节的内存。相反，物理内存只占据 16M 字节并且在地址空间中重
[35] 复出现。3.7 节会解释重复的原因。

3.7 内核段 KSEG0 和 KSEG1 的内容

内核段 KSEG0 和 KSEG1 是操作系统的基础，它们具有特别的含义。操作系统放在 KSEG0 中，这个段的最高地址保留给 I/O 设备使用，低端地址对应物理内存，图 3-4 中显示了 KSEG0 的布局。

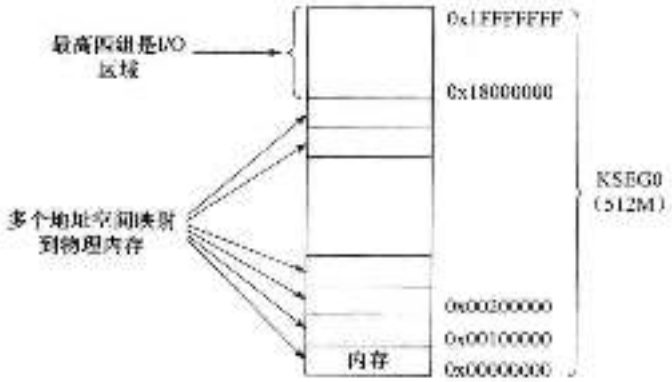


图 3-4 KSEG0 的地址空间

尽管 KSEG0 包含了 512M 字节的地址，但 E2100L 的物理内存只有 16M 字节。因此在地址空间中，物理内存表现的会重复出现。例如，物理内存占据了 0x00000～0x00FFFFF 的相对地址，而同样的物

理内存也占据着物理地址空间 0x0010000 ~ 0x001FFFFF。也就是说，在将这部分地址映射到物理内存的时候，硬件忽略了地址的高位[⊖]。

最重要的一点是：

> 引用超出物理内存大小的地址不一定会产生一个错误，这是由于地址的高位被忽略了，因此地址空间可以被映射到合法的物理内存位置，就好像物理内存重复一样。

3.8　总线启动的静态配置

桌面计算机包含了复杂的总线硬件，让处理器能够检测到连接在总线上的所有硬件。每一个组件都分配一个唯一的标识符来确定供应商和设备类型。在启动时，操作系统首先探测总线，判断哪一个组件为当前组件，并要求所有的组件都返回它们的标识符。该机制使操作系统能够动态地进行自我配置，同时使同一系统具有运行在不同硬件平台上的能力。

与桌面计算机系统不同，嵌入式系统通常使用静态配置的方式。也就是说，在操作系统设计的时候所有的硬件配置就应该确定了，操作系统在启动的时候将不再动态探测或重新配置。第 24 章将讨论系统配置的更多细节内容，并提供一个静态配置的实例。

3.9　调用约定和运行时栈

函数调用是操作系统很重要的一个方面。应用程序通过函数调用的方式来使用操作系统提供的服务，例如创建进程或进行 I/O 操作。当操作系统在进程间切换时，其必须管理这些独立的函数调用。下面将定义一些与函数调用有关的重要概念。

调用约定　函数调用及其返回过程中所需要的步骤称为调用约定。使用约定这个术语是因为在此过程中硬件对于细节并不了解。相反，硬件设计对于一些可能的方法进行了一些约束，并留给编译器的开发者很多选择。可以看到，因为操作系统通过调用函数来触发中断并从一个进程切换到另一个进程，所以它必须理解并支持与编译器相同的约定。

运行时栈　静态域的语言，比如 C 语言，需要使用运行时栈来存储与函数调用有关的状态。编译器在栈中为已经调用的函数分配了足够的空间以保存活动记录。这个分配的空间就是栈桢。每个活动记录包含与调用有关的局部变量空间、临时存储空间、返回地址和其他一些五花八门的东西。约定还规定一个栈向下增长时，内存地址逐渐减小。因此，当一个函数调用发生时，运行时栈在内存中向下增长来保存函数调用的活动记录。

参数　当一个函数被调用时，调用者必须根据参数列表提供一组真实的参数。在大多数 RISC 架构中有固定数量的参数通过寄存器传输，剩余的参数通过内存传输。示例代码中使用 A0 ~ A3 寄存器传递前 4 个参数；超过 4 个的参数作为活动记录放在栈中。

栈桢的内容　编译器需要计算栈桢所用的空间，并为每一个局部变量分配空间。然而，操作系统需要知道参数具体是怎么存放的。在示例代码中，每一个栈桢中最高位的字是用来存放返回地址的，最低的 4 个字是参数保留空间，在参数保留空间之上是除去前 4 个参数以外的一系列参数。图 3-5 解释了这一格式。

有趣的是，在函数 f 中占据了最低 4 个字的参数保留空间却没有被函数 f 所使用到。相反的，这个区域被 f 调用的函数所使用。如图 3-5 所示，如果函数 X 调用函数 Y，那么由函数 X 在栈中开辟的最低 4 个字的空间将被 Y 使用。

我们可以通过观察向 4 个参数处由寄存器 A0 ~ A3 传递的这一现象来理解参数保留区域的必要性。例如，假设函数 X 调用了函数 Y，并传递参数 q。那么当函数 Y 开始运行时，寄存器 A0 将保存 q 的值。如果在函数 Y 中又调用了函数 Z 并传递了参数 r，那么这时候 A0 将用于保存 r 的值，函数 Y 必须覆盖为 X 保留的 q 值。为了保证 q 的值不丢失，Y 在调用 Z 之前将 q 的值保存到栈中，并在 Z 返回之后从

⊖ 虽然在嵌入式系统中有很多忽略地址高位的情况发生，但多数计算机系统还是将地址空间严格控制在物理内存的范围之内，而将访问其他的地址视为错误。

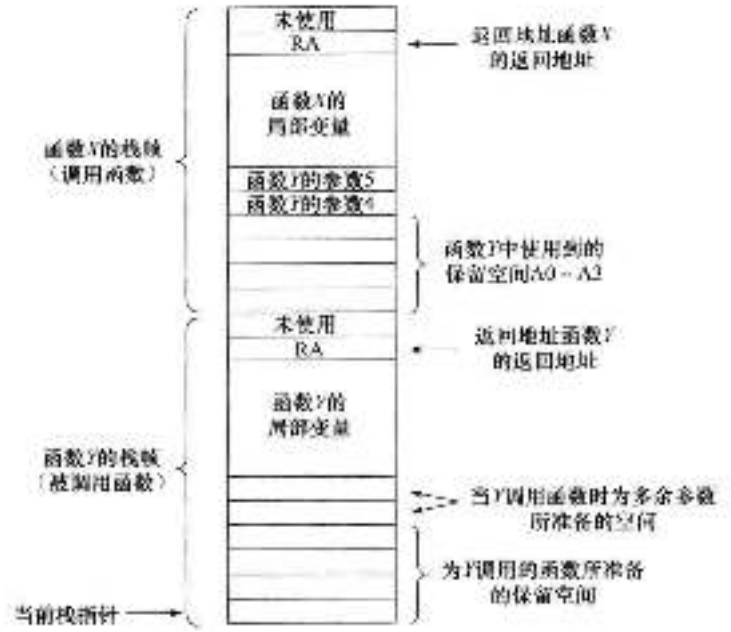


图3-5 当函数X调用函数Y时两个栈帧的样式

栈中恢复这些值。理论上，Y可以使用内存中的任意位置保存A0中的值——在调用结构中使用保留参数空间只是与编译器所使用的约定一致。

作为进一步的约定，栈指针寄存器是不允许使用的。相反，编译器计算每一个函数所需要的栈空间（包含4个字的参数保留空间），并在函数被调用时使代码对应的栈指针减少正确的大小。因为RISC处理器并不包含将值压栈的指令，所以编译器可以通过一个从栈指针固定的偏移量计算出每一个局部变量的地址。

总结：

> 当引用局部变量或运行栈中的参数时，操作系统将遵从编译器的调用约定。在贯穿本书
32 的示例中，所有引用都是以栈指针偏移量的形式给出。

3.10 中断和中断处理

现代计算机提供了一种机制使得外部的I/O设备能够中断正在进行的计算。通常，处理器有一个类似的异常机制，在异常或错误发生时通知软件（例如，应用程序尝试除以0或请求虚拟内存中的页表）。从操作系统的角度看，中断是基础，因为它允许CPU同时进行计算并处理I/O操作[⊖]。

任何连接在总线上的I/O设备在需要服务时都可以向处理器发送中断请求。为了实现上述功能，这些设备会在总线的控制线上放置一个信号。在常规的获取－处理循环中，处理器中的硬件对控制线进行监控，并在控制线有信号时初始化中断处理过程。以RISC处理器为例，主处理器并不包含处理中断的硬件。相反，协处理器会代替主处理器来与总线交换信息并处理中断过程。

例如，当一个设备发生中断时，MIPS处理器执行以下三个重要步骤：

- 设置控制位，不允许新的中断。
- 记录将要执行的指令地址。
- 跳转至保留位置0x80000180。

第一步保证当系统处理一个设备产生的中断时，不会被其他设备再次中断。第二步为操作系统提供了一种在处理完中断后返回继续执行普通代码的方式。第三步使得无论中断在何时发生，操作系统

⊖ 后续章节将介绍操作系统如何管理中断的过程，并且解释用户发起的高层I/O请求操作如何与底层的设备硬件相对发生联系。

都能够获得控制权。在中断发生前，操作系统必须在保留位置（0x80000180）存储中断处理代码。编译器和加载器从0x80010000（一个中断处理程序的位置）启动操作程序，从而保证在保留位置的中断处理程序不会受到操作系统中其他值的影响。在系统启动时，我们的示例系统会在保留区域存储一段指令，无论中断何时发生，这段指令都可以使处理器跳转至操作系统。

当操作系统处理个域数据或I/O队列时，它必须防止中断的发生。这种防止机制由硬件提供。例如，协处理器中的一个寄存器被用做操作系统的中断屏蔽，用来指定哪些设备被允许产生中断。这个屏蔽的每一位都对应于系统中的某一个中断源。中断源可以是系统总线上的设备、内部定时器，或者特殊处理器操作码的执行。所有位的初值均为0，表示所有的源都不可以产生中断。当操作系统启动I/O设备时，操作系统会将相应的屏蔽位置1，从而使该设备可以产生中断。

除了每个设备的对应位外，中断屏蔽还包含一个全局中断状态位。如果将它设置为0，无论设备 [40]
本身的位是0还是1，中断都不会发生。在后面的章节将看到函数disable和restore操作全局状态位，从而使操作系统能够暂时禁止所有中断，以后再恢复中断。

图3-6列出了与中断有关的协处理器寄存器并介绍了每一个的设计目的。

寄存器	功能
STATUS	表示中断状态，包含一个EXL位指定当前中断是否已处理、一个全局位表示所有的中断是否被禁止，以及每个中断源对应一个中断位
CAUSE	表示中断或异常发生源的唯一标识位
EPC	异常程序计数器，用于记录异常处理完成后应该返回的正常程序的位置

图3-6 与中断有关的协处理器寄存器

处理器通过访问协处理器控制寄存器的方式来判断哪一个设备发生了中断，然后与其进行交互。一旦中断处理完成，处理器可以运行中断返回指令，恢复普通代码的执行。之后的章节将提供中断处理的示例以及处理器与不同设备交互的步骤。

3.11 异常处理

尽管异常是由处理器而不是独立的I/O设备产生，但MIPS硬件将异常处理与中断处理合二为一。也就是说，异常处理和中断处理使用相同的硬件步骤来完成：设置EXL位来防止进一步的中断，在EPC寄存器中记录产生异常的代码地址，跳转至0x80000180。

处理器处理中断和异常还是存在细微的差异。当中断发生时，处理器总是处在刚执行完当前指令，正准备执行下一条指令的时刻。因此，EPC寄存器存储的是下一条待执行的指令。当异常发生时，当前指令正在执行，因此EPC寄存器记录正在执行的指令地址。当处理器从异常返回后，这条指令将被 [41]
重新执行。例如，如果发生页错误，异常处理程序会从内存中读取缺失页，返回发生异常的点并重行执行导致页错误的指令。

3.12 计时器硬件

除了外部I/O设备外，E2100L硬件还包括计时器设备。当它结束时，定时器就会产生一个中断，也就是说，如果计时器中断是允许的，那么处理器就必须准备好处理该中断。

在一些嵌入式操作系统中，所有的计时器函数都是通过实时硬件时钟有规律地产生时钟中断来实现的（比如，每秒60次中断）。然而，在E2100L上定时器包含两个处理器可读的寄存器：

- 计数器：给计数寄存器设置一个初始值。
- 限制器：限制寄存器用于指定等待时间。

硬件使用CPU时钟作为计时依据，每个时钟周期都会对计数寄存器加一。当计数器的值达到限制器中的值时，将触发计时器中断。

实时时钟方法与E2100L定时器机制各有优势。E2100L的主要优势是产生更少的中断。与实时时

钟滴答产生中断不同，定时器只在预设的超时发生时触发中断。实时时钟的优势是它能够将中断直接与实时时间相联系，而不需要借助处理器时钟。当然，如果你知道处理器频率，那么将其转换为真实时间是可能的。不幸的是，这种计算依赖计算机 CPU 速度，也就是说，操作系统不调整换算使用的常数，就不能直接移植到更快的处理器上。

3.13 串行通信

串行通信广泛存在于现存的 I/O 设备中，这一技术已经使用了数十年。E2100L 包含一个 RS-232 串行通信设备作为系统控制台。与大多数串行设备一样，E2100L 能够处理输入与输出（也就是说，同时包含发送与接收字符）。当中断发生时，处理器必须检查设备寄存器以确定输出端的字符发送和输入端的字符接收是否完成。第 15 章将详细讲述串行中断。

[42]

3.14 轮询与中断驱动 I/O

大多数由操作系统驱动的 I/O 设备都使用中断机制。操作系统先与一个设备交互并发起一个操作（输入或者输出），然后再完成计算。当 I/O 操作完成时，设备中断处理器，操作系统可以选择启动另一个操作。

尽管中断机制优化了并发性并允许多设备并行处理计算，但中断机制并不是在任何情况下都能使用。例如，如果在操作系统初始化中断和 I/O 之前需要向用户显示一个欢迎界面，又比如一个程序员需要测试新的 I/O 代码时，上述两种情况下中断机制都是无法使用的。

替代中断驱动 I/O 的方法称为轮询 I/O。当处理器启动了一个 I/O 操作，但并不能够进行中断时，可以使用轮询 I/O。操作系统进入一个循环，并不断地检查设备状态寄存器来判断操作是否结束。在第 2 章讨论 kputc 和 kprintf 的时候已经看到过操作系统设计如何使用轮询 I/O 机制的例子。

3.15 内存缓存和 KSEG1

记得尽管在地址空间中 KSEG1 在 KSEG0 之后，但是在内存空间中它们是重复的。所以，KSEG1 的第一个字节指向了具有相同物理内存地址的 KSEG0 的第一个字节。

尽管看起来是重复的，但是硬件强调了这两个内存段差别的重要性：

> 当处理器引用 KSEG0 的地址时，将引用传送到系统（也就是系统主板）之前，首先送入 L1 内存缓存；当处理器引用 KSEG1 的地址时，将它直接传送到总线。

对于普通的数据引用，内存缓存提供了重要的优化——如果处理器短时间内对同一物理地址进行了多次引用，那么缓存硬件返回值比从内存引用要快得多。然而，在处理 I/O 时，缓存会产生不正确的结果。例如，一段代码使用轮询 I/O 机制并检查设备的状态。如果缓存保存了访问设备之前的状态值，那么处理器就不会得到当前设备状态的准确值。所以，操作系统遵守了以下简明的准则：

[43]

> 为了避免从内存缓存中获取旧的值，每一个 I/O 引用（包括 DMA 指定的地址）必须使用 KSEG1。

3.16 存储布局

当 C 编译器编译一个程序时，它将结果镜像切分成 4 个内存段：

- 代码段
- 数据段
- bss 段
- 栈段

代码段包含了主程序的源码和所有函数，占据最低的地址空间。数据段包括所有初始化数据，占据文本段地址空间之后的区域。非初始化数据段称为 bss 段，它紧接在数据段之后。最后，栈段占据了地址空间的最高部分并向低地址增长。图 3-7 说明了上述概念结构：

图中 etext、edata 和 end 表示加载器加入目标程序的全局变量。它们的名字相应地初始化在代码、数据和 bss 段上。因此，一个正在运行的程序可以通过计算在 bss 段底部的 end 和栈段栈顶地址（即 SP

指针）的差值得出程序剩余的内存容量。

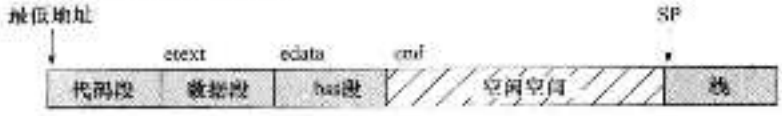


图 3-7　C 编译器创建的内存段示意图

第 9 章将阐述多处理器的内存分配机制。尽管所有处理器共享代码、数据和 bss 段，但是所有的处理器必须分配独立的栈段。如果 3 个处理器，那么栈的分配就如图 3-8 所示从最高的内存地址连续向下分配。

如图 3-8 所示，每一个处理器都有自己的栈指针。在任何一个给定的时刻，处理器 i 的栈指针必须指向一个分配给第 i 个处理器的栈空间。后面的章节将详细介绍这一概念。



图 3-8　3 个处理器情况下内存栈段示意图

3.17　内存保护

E2100L 的内存硬件有多段机制可以为操作系统提供保护。应用程序可以设置为用户态，从而不能读、写内核段的内存空间。当应用程序调用一个系统调用时，控制权将交给内核，系统权限将提升至内核态直到调用返回。理解这一保护机制的关键是控制权的传递只能在操作系统设计者指定的入口进行。这样，设计者才能保证应用程序获得严格控制的服务。

像大部分的嵌入式操作系统一样，我们的示例系统降低了内存保护的复杂度，避免了常规的运行时内存保护。相反，代码将完全运行在 KSEG0 上，并且所有进程都以内核权限运行。缺少保护也就意味着程序员需要非常小心，因为任何进程都能够访问内存的任何位置，包括分配给操作系统和其他处理器栈的空间。如果一个进程溢出了分配给它的内存区域，那么进程的运行时栈将覆盖其他进程栈的数据。后面章节将讨论一个技术软件，它可以用来帮助探测溢出。

3.18　观点

对处理器和 I/O 设备来说，它们的硬件规格说明包含了太多的细节以至于难以学习。幸运的是，处理器的许多不同都是表面的——基础的概念在大多数硬件平台中都是通用的。因此，在学习这些硬件时，应该注重整体架构和设计原则而不是小的细节。

练习

3.1　有些系统使用可编程的中断地址机制，允许系统选择当中断发生时处理应该跳转的地址。请问可编程中断机制的优势是什么？

3.2　DMA 有引入未知错误的可能性。如果一个 DMA 操作从最高内存地址小于 N 字节的内存单元开始传输 N 字节会发生什么情况？

3.3　阅读有关使用多级中断的硬件的文章。当操作系统正在处理其他级中断的时候，在某一级别的中断能不能够打断操作系统？请解释原因。

3.4　图 3-7 中所展示的内存布局的优点是什么？它是不是有缺点？在什么方面其他的布局是有用的？

3.5　嵌入式硬件经常包含多个独立的计时器，每一个计时器都有自己的中断源。为什么说多计时器是有用的？一个只有单一计时器的系统能不能达到多计时器的所有功能？请解释原因。

3.6　如果你熟悉汇编语言，请阅读用于允许函数递归调用的调用规约。建立一个使用递归调用的函数，并阐述你的函数能够正确地运行。

链表与队列操作

如果某一天需要找出一个扫烟囱者，我例证手就准备了一份小小的链表……

——W. S. Gilbert

4.1 引言

链表操作是操作系统中相当基础的操作，并且遍及其每一个组件。链表数据结构使得系统可以高效地管理一系列的对象而不需要搜索或复制。就像我们将看到的那样，在进程管理中这是尤为重要的。

本章介绍了一系列构成链表操作的核心函数。这些函数体现了一个统一的方法——操作系统的不同层次都使用统一的数据结构和统一的结点集合来维护各个进程。我们将看到这些数据结构和函数如何来创建一个新链表，如何在队尾插入一个项，如何在有序队列中插入一个项，如何移除队列首项，以及如何从队列的任意位置移除某项[⊖]。

链表函数相当易于理解，因为系统假设同一时间只有一个进程访问一个链表函数。因此，读者可以将代码视为一个串行程序——没有必要担心来自其他进程的干预。此外，示例代码介绍了多个贯穿全书的编程约定。

[49]

4.2 用于进程链表的统一数据结构

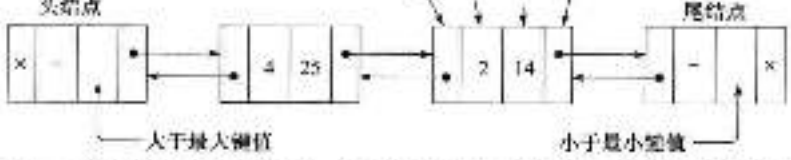
进程管理器管理着进程。尽管任意时刻一个进程只出现在一个链表中，但是进程管理器频繁地将进程从一个链表转移到另一个链表。事实上，进程管理器并不存储进程的所有细节。相反，进程管理器仅仅保存进程的 ID，一个用来表示进程的非负整数。出于方便的考虑，我们将交替使用术语进程和进程 ID。

Xinu 的早期版本有很多进程链表，每个链表都有自己的数据结构。一些由先进先出（FIFO）队列组成，另一些由键值排序。一些链表是单向链接的；另一些则需要双向链接来保证某一项可以在链表中的任意位置高效地插入或删除。当需求被形式化之后，就会发现将进程链表集中到某个单一的数据结构将会大大缩减代码量并减少特殊分支。也就是说，不是用 6 个分开的链表操作函数，而是用单一的一个函数集合来处理所有情况。

为了适应所有情况，我们选择了一个拥有下列属性的代表。

- 所有的链表都是双向链表，也就是说，一个结点既指向前驱结点也指向后继结点。
- 每个结点都存储一个键值和一个进程 ID，尽管键值并不在 FIFO 链表中使用。
- 每个链表都有头、尾结点，头、尾结点占用和其他结点相同的内存。
- 非先进先出队列是以降序排列的。头结点的键值是最大的，尾结点的键值是最小的。

图 4-1 解释了链表数据结构的基本概念，图中链表是包含两个项的链表。



[50]

图 4-1 一个双向链表的概念结构，它包含了进程 4 和进程 2，键值分别为 25 和 14

⊖ 尽管链表操作通常在“数据结构”的课程中提及，但是我们仍然讨论这个话题，因为数据结构非比寻常，它组成了操作系统的关键部分。

正如我期的那样，尾结点的后继和头结点的前驱结点都是空的（null）。当一个链表为空时，头结点的后继为尾结点，尾结点的前驱是头结点，如图 4-2 所示。

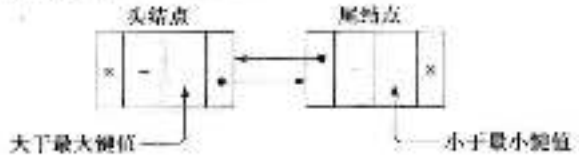


图 4-2　一个空链表

4.3　简洁的链表数据结构

嵌入式系统中的一个关键设计目标就是减少内存的使用。不同于使用传统的链表实现，Xinu 在两个方面优化了内存需求：

- 相对指针
- 隐式数据结构

为了理解优化，我们需要知道大多数操作系统设置了一个进程数量上限。在 Xinu 中，常量 NPROC 指定了这个上限。进程标识符的范围是 0 ~ NPROC − 1。在大多数嵌入式系统中，NPROC 相当小（小于 50）。我们之后将看到一个较小的限制会使优化工作更好。

相对指针　为了理解相对指针的设计动机，我们考虑传统指针占用的空间。在 32 位架构中，每个指针占用 4 字节。如果系统有小于 50 个的结点，那么指针需要的空间可以由连续的内存以及 0 ~ 49 的数值代替。也就是说，可以将结点分配给一个数组，数组的索引可以代替结点的指针。

隐式数据结构　第二种优化关注于从所有结点中省略进程的 ID 字段。此类省略是可行的，因为：

> 一个进程在任一时刻只会在一个链表中。

为了省略进程 ID，使用一个数组并且用第 i 个元素代表进程 ID i。因此，在链表中插入结点 3 就是放入了进程 3。因此，结点的相对路径和存放进程的 ID 一致。 51

图 4-3 说明了图 4-1 中的链表如何被采用了相对指针和隐式标识符的数组来代替。数组的每个项有三个字段：键值、前驱结点的索引、后继结点的索引。头结点的索引为 60，尾结点的索引为 61。

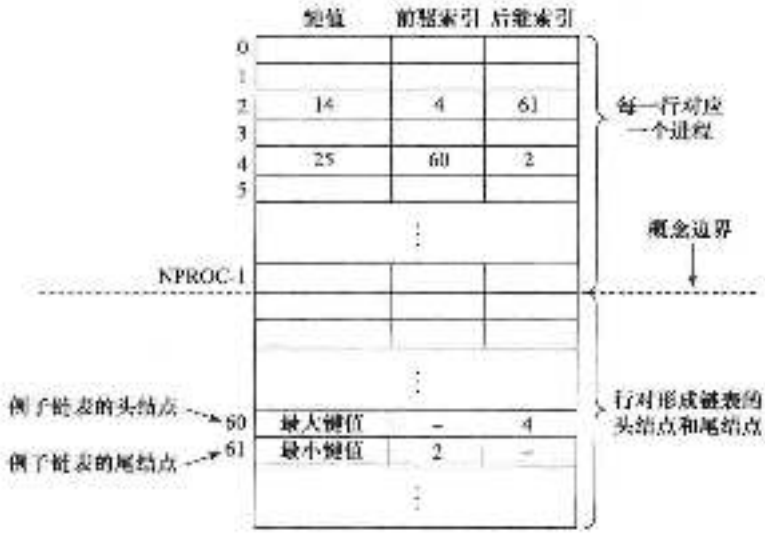


图 4-3　图 4-1 中的链表的队列数组

因为后继和前驱字段包含的是相对指针（如数组索引），所以字段的大小取决于数组的大小。例如，如果一个数组包含小于 256 个元素，那么一个字节就能满足所有相对指针的需求。

Xinu 使用术语队列来表示数组。理解这个数据结构的关键是观察元素中小于索引 NPROC 的与大于它的索引的区别。位置 0 – NPROC – 1 的每一个对应于系统中的一个进程。NPROC 以及更大的索引被用来保存头尾结点的。这个数据结构可行的原因是头、尾结点在编译阶段就知道它们的索引号，
52 并且一个进程在任一时刻只能出现在一个链表中。

4.4 队列数据结构的实现

为了将进程 i 放入队列中，索引为 i 的结点要加入链表中。仔细看看下面代码会让我们的思路变得清晰。在 Xinu 中，图 4-3 中的队列表命名为 queuetab，用来存放 qentry 的数组。文件 queue.h 包含了 queuetab 和 qentry 的声明：

```
/* queue.h - firstid, firstkey, isempty, lastkey, nonempty                */

/* Queue structure declarations, constants, and inline functions          */

/* Default # of queue entries: 1 per process plus 2 for ready list plus */
/*                             2 for sleep list plus 2 per semaphore      */
#ifndef NQENT
#define NQENT   (NPROC + 4 + NSEM + NSEM)
#endif

#define EMPTY   (-1)            /* null value for qnext or qprev index  */
#define MAXKEY  0x7FFFFFFF      /* max key that can be stored in queue  */
#define MINKEY  0x80000000      /* min key that can be stored in queue  */

struct  qentry  {               /* one per process plus two per list    */
        int32   qkey;           /* key on which the queue is ordered    */
        qid16   qnext;          /* index of next process or tail        */
        qid16   qprev;          /* index of previous process or head    */
};

extern  struct qentry   queuetab[];

/* Inline queue manipulation functions */

#define queuehead(q)    (q)
#define queuetail(q)    ((q) + 1)
#define firstid(q)      (queuetab[queuehead(q)].qnext)
#define lastid(q)       (queuetab[queuetail(q)].qprev)
#define isempty(q)      (firstid(q) >= NPROC)
#define nonempty(q)     (firstid(q) <  NPROC)
#define firstkey(q)     (queuetab[firstid(q)].qkey)
#define lastkey(q)      (queuetab[ lastid(q)].qkey)

/* Inline to check queue id assumes interrupts are disabled */

#define isbadqid(x)     (((int32)(x) < 0) || (int32)(x) >= NQENT-1)
/* Queue function prototypes */

pid32   getfirst(qid16);
pid32   getlast(qid16);
pid32   getitem(pid32);
pid32   enqueue(pid32, qid16);
pid32   dequeue(qid16);
status  insert(pid32, qid16, int);
status  insertd(pid32, qid16, int);
53 qid16   newqueue(void);
```

queuetab 数组包含了 NQENT 个表项。如图 4-3 中所示，一个重要边界出现在 NPROC - 1 和 NPROC 之间。小于该边界的每个元素对应于一个进程 ID，元素 queuetab[NPROC] ~ queuetab[NQENT] 对应于链表的头或尾。

queue.h 引入了多种 C 语言的特性和本书中所使用的编程习惯。因为以 .h 结尾的文件将会被其他程序引用（“h”代表头文件）。此类文件经常包含一些全局数据结构的声明、符号常量，以及用于操作数据结构的内联函数（宏）。queue.h 文件定义 queuetab 为一个外部变量（全局变量），意味着每个进程都可以访问这个数组。该文件还定义了数据结构所使用的符号常量，如 EMPTY 用于定义空链表。

符号常量 NQENT 定义 queuetab 数组中的表项总数，该定义为条件定义。语句#ifndef NQENT 的意思是“仅当 NQENT 没有定义时，将到#endif 的代码进行编译”。NQENT 被定义的值为，

$$NPROC + 4 + NSEM + NSEM$$

在 queuetab 中为 NPROC 个进程，NSEM 个信号量链表的头结点和尾结点、一个就绪链表、一个睡眠链表分配足够的入口。使用条件编译使 queuetab 数组可以改变大小而不需要修改 .h 文件。

queuetab 中数组每个表项的内容都是由结构 qentry 定义的。该文件只包含一个 queuetab 数组中的元素声明。第 22 章解释这些数据结构在系统启动的时候如何初始化。字段 qnext 提供了链表中下一个结点的相对地址，字段 qprev 指向前一个结点，字段 qkey 包含了该结点的一个修改键值。当一个字段，如前驱或后继指针，没有包含一个可用的索引值时，这个字段被赋值为 EMPTY。 54

4.5　内联队列操作函数

函数 isempty 和 nonempty 是检查链表是否为空的函数（布尔函数），它将链表的头结点作为参数。通过检查链表中的第一个结点是否是一个进程或者是尾结点，isempty 确定了一个链表是否为空；nonempty 的作用与 isempty 相反。记住，一个项只有当它的索引小于 NPROC 时才会被处理。

其他的内联函数也相当易于理解。firstkey、lastkey 和 firstid 返回链表中第一个进程的键值、最后一个进程的键值，或者第一个进程的 queuetab 索引。通常，这些函数用在非空链表上。

4.6　获取链表中进程的基础函数

如何从链表中获取进程[⊖]？如前所述，从 FIFO 队列的头部获取结点会使存在时间最长的结点被移除。对于一个优先级队列而言，从头开始寻找将会产生一个优先级最高的结点。同样，从尾部开始会产生一个优先级最低的结点。因此，我们可以构造三个简单有效的处理函数：

- getfirst：获取头结点。
- getlast：获取尾结点。
- getitem：获取任意指针位置的进程。

这三个函数的代码可以在 getitem.c 中找到： 55

```
/* getitem.c - getfirst, getlast, getitem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getfirst  -  Remove a process from the front of a queue
 *------------------------------------------------------------------------
 */
pid32   getfirst(
          qid16         q               /* ID of queue from which to    */
        )                               /* remove a process (assumed    */
                                        /* valid with no check)         */
```

⊖ 我们将在后面考虑向链表插入一个元素。

```
{
        pid32   head;

        if (isempty(q)) {
                return EMPTY;
        }

        head = queuehead(q);
        return getitem(queuetab[head].qnext);
}

/*------------------------------------------------------------------------
 *  getlast  -  Remove a process from end of queue
 *------------------------------------------------------------------------
 */
pid32   getlast(
          qid16         q               /* ID of queue from which to    */
        )                               /* remove a process (assumed    */
                                        /* valid with no check)         */
{
        pid32 tail;

        if (isempty(q)) {
                return EMPTY;
        }

        tail = queuetail(q);
        return getitem(queuetab[tail].qprev);
}

/*------------------------------------------------------------------------
 *  getitem  -  Remove a process from an arbitrary point in a queue
 *------------------------------------------------------------------------
 */
pid32   getitem(
          pid32         pid             /* ID of process to remove      */
        )
{
        pid32   prev, next;

        next = queuetab[pid].qnext;     /* following node in list       */
        prev = queuetab[pid].qprev;     /* previous node in list        */
        queuetab[prev].qnext = next;
        queuetab[next].qprev = prev;
        return pid;
}
```

getfirst 接收一个队列 ID 作为参数，验证该参数确定一个非空链表，找到该链表的头结点，然后调用 getitem 从链表中获取进程。同样，getlast 接收一个队列 ID 作为参数，从队尾开始寻找，重复上述过程。这两个函数都返回进程的 ID。

getitem 接收进程 ID 作为参数，从链表中找到这个进程。获取过程包括将原来的前驱结点与后继结点相互连接，当目标进程从链表中完全删除时，getitem 返回该进程的 ID。

4.7 FIFO 队列操作

我们将看到进程管理器的很多链表是由先进先出队列（FIFO）组成的。也就是说，一个新的结点插入链表的尾部，每个结点都是从链表的头部被移除的。例如，调度器可以使用先进先出队列来实现循环调度，将当前进程放入链表的尾部，并切换到链表头的进程。

文件 queue.C 中的函数 enqueue 和 dequeue 实现的是链表上的 FIFO 操作。因为每个链表都有头、尾结点，所以插入和提取的效率都很高。例如，enqueue 将在尾结点前面插入一个项，而 dequeue 则在头结点后面移除一个项。dequeue 接收一个参数作为链表的 ID，enqueue 通过接收两个参数，分别是进程 ID 和要插入链表的链表 ID。

56 / 57

```
/* queue.c - enqueue, dequeue */

#include <xinu.h>

struct qentry	queuetab[NQENT];	/* table of process queues	*/

/*------------------------------------------------------------------------
 *  enqueue  -  Insert a process at the tail of a queue
 *------------------------------------------------------------------------
 */
pid32	enqueue(
	  pid32		pid,		/* ID of process to insert	*/
	  qid16		q		/* ID of queue to use		*/
	)
{
	int32	tail, prev;		/* tail & previous node indexes */

	if (isbadqid(q) || isbadpid(pid)) {
		return SYSERR;
	}

	tail = queuetail(q);
	prev = queuetab[tail].qprev;

	queuetab[pid].qnext  = tail;	/* insert just before tail node */
	queuetab[pid].qprev  = prev;
	queuetab[prev].qnext = pid;
	queuetab[tail].qprev = pid;
	return pid;
}

/*------------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------------
 */
pid32	dequeue(
	  qid16		q		/* ID queue to use		*/
	)
{
	pid32	pid;			/* ID of process removed	*/

	if (isbadqid(q)) {
		return SYSERR;
	} else if (isempty(q)) {
		return EMPTY;
	}

	pid = getfirst(q);
	queuetab[pid].qprev = EMPTY;
	queuetab[pid].qnext = EMPTY;
	return pid;
}
```

函数 enqueue 调用 isbadpid 来检查参数是否是一个合法的进程 ID。第 5 章将说明 isbadpid 由内联函

数组成，它检查 ID 是否在正确的值域内并且对应该 ID 的进程是否存在。

文件 queue.c 包含 xinu.h，它包含了完整的 Xinu 引用文件：

```
/* xinu.h - include all system header files */

#include <kernel.h>
#include <conf.h>
#include <process.h>
#include <queue.h>
#include <sched.h>
#include <semaphore.h>
#include <memory.h>
#include <bufpool.h>
#include <clock.h>
#include <mark.h>
#include <ports.h>
#include <uart.h>
#include <tty.h>
#include <device.h>
#include <interrupt.h>
#include <file.h>
#include <rfilesys.h>
#include <rdisksys.h>
#include <lfilesys.h>
#include <ag71xx.h>
#include <ether.h>
#include <mips.h>
#include <nvram.h>
#include <gpio.h>
#include <net.h>
#include <arp.h>
#include <udp.h>
#include <dhcp.h>
#include <icmp.h>
#include <name.h>
#include <shell.h>
#include <date.h>
#include <prototypes.h>
```

将分散的头文件整合为一个单一的头文件对程序员来说非常有用，因为这样做确保所有相关的定义都是可用的，并且还能保证这些分散的头文件处于一个合理的顺序。在后面的章节中，我们将看到这些头文件的内容。

4.8 优先级队列的操作

进程管理器通常需要从进程的集合中选择一个优先级最高的进程。在我们的示例系统中，优先级是分配给进程的一个整数。通常，查找具有最高优先级进程的任务经常与插入与删除结点进行比较。因此，管理进程列表的数据结构应该这样设计：查找最高优先级进程的操作应该比插入和删除操作更为高效。

许多数据结构被设计成能够存储以优先级方式访问的集合。任何一个这样的数据结构都称为优先级队列。我们的示例系统使用线性链表来存储优先级队列，其中进程的优先级就是链表中的键值。因为链表是按键值降序排列的，所以最高优先级的进程总能在链表的头中找到。因此，找到最高优先级进程的开销为常数时间。插入是一个开销更大的操作，因为必须搜索链表来决定在哪个位置插入。

在小型嵌入式系统中，通常一个队列中只有 2～3 个进程，因此线性链表就足够了。对一个大型系

使而言，要么会有大量的元素存储于优先级队列中，要么插入操作的数目远大于元素获取操作，此时线性链表就会显得效率很低。后面的例子将会更深入地指出这一点。

从有序链表中删除并不难：将第一个结点从链表中移除。当插入一个元素时，必须保持链表的顺序。该函数需要三个参数：要插入进程的 ID、要插入队列的 ID，以及进程的整数优先级。插入使用 queuetab 中的 qkey 字段来存放进程的优先级。为了在链表中找到正确的位置，插入操作搜索比待插入元素的键值小的元素。在搜索时，整数 curr 遍历整个链表。循环最终必定终止，因为尾结点的键值比最小的有效键值小。一旦找到正确的位置，插入操作通过改变必要的指针来加入新结点。

58
|
61

```
/* insert.c - insert */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insert  -  Insert a process into a queue in descending key order
 *------------------------------------------------------------------------
 */
status  insert(
        pid32       pid,            /* ID of process to insert      */
        qid16       q,              /* ID of queue to use           */
        int32       key             /* key for the inserted process */
        )
{
        int16   curr;               /* runs through items in a queue*/
        int16   prev;               /* holds previous node index    */

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        curr = firstid(q);
        while (queuetab[curr].qkey >= key) {
                curr = queuetab[curr].qnext;
        }

        /* Insert process between curr node and previous node */

        prev = queuetab[curr].qprev;    /* get index of previous node   */
        queuetab[pid].qnext = curr;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[curr].qprev = pid;
        return OK;
}
```

4.9 链表初始化

上述过程都假设，链表即使为空，也要初始化。现在考虑创建一个空链表的代码。之所以在本章末尾考虑创建空链表的代码，是因为这体现了设计过程中的一个要点：

> 初始化是设计中的最后一步。

这可能看起来很奇怪，因为设计者不可能延后初始化这个步骤。然而，一般的设计范式是这样的：第一，设计系统需要的数据结构；第二，规划如何初始化这个数据结构。将“能备状态”与“瞬间状态”区分开来，有助于我们关注设计者最重要的意图，避免因为简单的初始化问题而牺牲优秀的设计。

queuetab 数据结构中表项的初始化是按需进行的。运行中的进程调用函数 newqueue 来创建一个新的链表。系统维护一个全局指针指向下一个没有分配的 queuetab 元素。

理论上，链表中的头、尾结点可以被任何没有使用过的 queuetab 的表项初始化。实际上，找到任

意一个位置需要调用者存储两个项：链表的头、尾索引。为了优化存储，我们定义以下规则：

> 链表 X 的头、尾结点是由 queuetab 数组中的连续位置分配的，链表 X 的唯一标识符是头结点的索引。

在代码中，newqueue 给 queuetab 数组中分配了一对相邻的位置，并且通过将头结点的后继结点设为尾结点，尾结点的前驱结点设为头结点的方式将链表初始化。newqueue 将 EMPTY 赋给一个没有使用过的指针（即，尾结点的后继结点和头结点的前驱结点）。当初始化一个链表时，newqueue 也设置头结点和尾结点的键值字段，分别赋予最大整数值与最小整数值，这两个数值都不会作为键值使用。只需要一个分配函数，因为可以用链表来实现 FIFO 队列或优先级队列。

一旦结束初始化，newqueue 给调用者返回链表头结点的索引。调用者只需要存储一个值，因为尾
62 结点 ID 可以通过将头结点的键值 +1 计算出来。

```
/* newqueue.c - newqueue */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  newqueue  -  Allocate and initialize a queue in the global queue table
 *------------------------------------------------------------------------
 */
qid16	newqueue(void)
{
	static qid16	nextqid=NQENT;/* next list in queuetab to use	*/
	qid16		q;		/* ID of allocated queue	*/

	q = nextqid;
	if (q > NQENT) {		/* check for table overflow	*/
		return SYSERR;
	}

	nextqid += 2;			/* increment index for next call*/

	/* Initialize head and tail nodes to form an empty queue */

	queuetab[queuehead(q)].qnext = queuetail(q);
	queuetab[queuehead(q)].qprev = EMPTY;
	queuetab[queuehead(q)].qkey  = MAXKEY;
	queuetab[queuetail(q)].qnext = EMPTY;
	queuetab[queuetail(q)].qprev = queuehead(q);
	queuetab[queuetail(q)].qkey  = MINKEY;
	return q;
}
```

4.10 观点

使用简单的数据结构来处理链表后，就能够复用通用的链表维护函数。这样做可以减少代码冗余。使用隐式数据结构来减少内存的使用。对小型的嵌入式系统而言，简洁的代码和数据是非常必要的。对于有着充足空间的系统而言呢？不恰当的设计会导致一个软件占用所有它可以占用的空间，从而导
63 致内存不足。因此在设计的时候，考虑周全是非常必要的。

4.11 总结

本章描述了进程管理中的链表函数。在我们的示例系统中，进程链表是一个单独的、统一的数据结构——queuetab 数组。操作这些进程链表的函数可以创建 FIFO 队列或者优先级队列。所有的链表都有统一的格式：它们是双向的，每个链表都有一个头和一个尾，每个结点都有一个整数键值。当链表是优先级队列时，键值就会被使用。而当链表是 FIFO 队列时，则键值会被忽略。

为了减少所需要的空间，Xinu 使用相对指针和一个隐式数据结构。链表中的一个结点要么表示一个进程，要么表示链表头或尾。

练习

4.1　队列数据结构是怎么隐式定义的？

4.2　如果优先级的值从 -8 ~ 8，为了在 queuetab 中存储每个键值，需要多少位？

4.3　创建一个单独的一组函数，允许创建单链表，然后将元素插入 FIFO 或者优先级队列中。这样做比一级的做法增加多少内存，这样做是否可以降低 CPU 的使用。请解释。

4.4　insert 对所有的键值都正常工作吗？如果不是，是哪些键值导致了灾难的后果？

4.5　使用指针而不使用数组索引来维护链表，请问内存开销和 CPU 处理时间有何变化？

4.6　比较使用指针和数组索引的两种情况下，isempty 实现的复杂度。

4.7　大型系统有时使用堆来实现优先级队列。什么是堆？当长度为 1 ~ 3 的时候，它与双向有序链表相比，谁的代价更大？

4.8　函数 getfirst、getlast、gettail 并不检查它们的参数是否是一个合法的 ID。修改这些代码，加入校验。

4.9　将下标转换为内存地址的操作可能使用乘法实现。填充 qentry 为 2 字节的幂，然后检查新编译后的代码是否使用位移操作而不是乘法。

4.10　根据之前的练习，检查填充的和未填充数据结构在增加、删除时的区别。

4.11　在严格字对齐的架构（比如，MIPS）上，如果一个结构中包含了不是 4 字节整数倍的数据，编译器就会产生带有额外移位的代码。请尝试改变 qentry 的字段，使数据能够按机器字对齐。同时讨论对队列表的大小以及获取元素的代码所带来的影响。

4.12　修改 newqueue，检查由于试图分配多于 NQENT 个元素而引起的错误。 64

第5章

Operating System Design: The Xinu Approach, Linksys Version

调度和上下文切换

具有强大的执行力，能够将梦想变为现实的工作，才能够称为真正的工作。

——Max Jacob

5.1 引言

操作系统通过在计算时频繁切换处理器给人以并行执行的幻觉。由于运算速度远快于人的反应，其带来的影响就是——多个任务看上去就像同时处理一样。

上下文切换，就是停止当前进程，保存足够的信息以便可以在稍后将该进程重启，然后启动另一个进程。这一过程困难的地方在于上下文切换的时候，CPU 不会停止——我们需要连续地让 CPU 从当前代码转移到新的进程上。

本章介绍上下文切换的基本机制，说明操作系统如何保存当前进程的信息，选择下一个需要运行的进程，然后把控制权交给下一个进程。本章介绍的内容包括如何记录没有正在运行进程的数据结构，和上下文切换是怎么使用这些结构的。目前，我们暂时忽略何时以及为何要使用上下文切换来改变进程的问题。这些问题将在后续章节解答。

5.2 进程表

操作系统将所有与进程相关的信息记录在进程表中。进程表为当前所有存在的进程保存一个进程表项。当一个进程被创建的时候，我们需要在进程表中分配一个进程表项，当一个进程结束的时候删除该进程表项。由于在任一时刻有三只有一个进程在执行，所以进程表中也只有一个进程表项对应了活动进程——进程表中保存的状态信息对于正在执行的进程来说是过时的。进程表中其他的每一项包含了暂时停止执行的进程的相关信息。为了改变上下文，操作系统将当前正在运行的进程的信息保存在进程表中，然后从进程表中恢复将要执行的进程。

哪些信息需要保存在进程表中？在新进程运行时，系统必须保存那些可能被破坏的值。比如，栈。因为每个进程有自己单独的栈空间，所以每个栈不需要保存，然而，在进程运行的时候，它会改变硬件栈的指针寄存器。因此，栈指针的内容必须在进程被暂停的时候保存，必须在进程重新执行时恢复。类似地，通用寄存器的值也要保存和恢复。除了硬件信息外，操作系统也在进程表中保存元信息。我们将会看到操作系统如何使用这些元信息实现进程计数、错误避免和其他管理任务。例如，多用户系统的进程表将记录每个进程 ID 属于哪个用户。类似地，如果操作系统限制进程可以调用的内存空间，这个限制也会保存在进程表中。我们将在后续的章节中讲述操作进程的系统函数并详细阐述进程表的细节。

在我们的示例操作系统中，进程表 proctab 由一个 NPROC 个元素的数组组成。proctab 中的每个元素记录了一个进程所需要的信息。图 5-1 列出了进程表的主要元素。

字段	目的
prstate	进程的当前状态（比如，进程当前正在执行还是正在等待）
prprio	进程的调度优先级
prstkptr	当进程不运行时的栈指针的值
prstkbase	内存中进程栈的最高地址
prstklen	进程栈的最大值
prname	为了能够让人识别进程所分配的进程名字

图 5-1 Xinu 进程表的主要元素

在整个操作系统中，每个进程由一个整数 ID 唯一标识。以下规则给出了进程 ID 和进程表之间的关系。

进程能够通过其进程 ID 来引用，在进程表中，保存了进程状态信息的表项的索引就是进程的 ID。

作为一个例子，考虑代码如何找到一个进程的信息。关于 ID 为 3 的进程信息可以在 proctab[3] 中找到，同样，ID 为 5 的进程信息可以在 proctab[5] 中找到。使用数值索引作为 ID 使得查找信息变得非常高效。

proctab 中的每个表项都定义为 procent 结构。结构 procent 和其他与进程有关的声明在 process.h 中。进程表中的其他字段包含了操作系统管理进程所需要的信息（比如，进程结束时需要释放的进程的栈内存的信息）。其他字段只是用于调试。比如，prname 字段包括了进程名称字符串，这个字段只有在调试的时候才会用到。

69

```
/* process.h - isbadpid */

/* Maximum number of processes in the system */

#ifndef NPROC
#define NPROC           8
#endif

/* Process state constants */

#define PR_FREE         0       /* process table entry is unused        */
#define PR_CURR         1       /* process is currently running         */
#define PR_READY        2       /* process is on ready queue            */
#define PR_RECV         3       /* process waiting for message          */
#define PR_SLEEP        4       /* process is sleeping                  */
#define PR_SUSP         5       /* process is suspended                 */
#define PR_WAIT         6       /* process is on semaphore queue        */
#define PR_RECTIM       7       /* process is receiving with timeout    */

/* Miscellaneous process definitions */

#define PNMLEN          16      /* Length of process "name"             */
#define NULLPROC        0       /* ID of the null process               */

/* Process initialization constants */

#define INITSTK         65536   /* initial process stack size           */
#define INITPRIO        20      /* initial process priority             */
#define INITRET         userret /* address to which process returns     */

/* Reschedule constants for ready */

#define RESCHED_YES     1       /* call to ready should reschedule      */
#define RESCHED_NO      0       /* call to ready should not reschedule  */

/* Inline code to check process ID (assumes interrupts are disabled)    */

#define isbadpid(x)     ( ((pid32)(x) < 0) || \
                          ((pid32)(x) >= NPROC) || \
                          (proctab[(x)].prstate == PR_FREE))

/* Number of device descriptors a process can have open */

#define NDESC           5       /* must be odd to make procent 4N bytes */
```

```
/* Definition of the process table (multiple of 32 bits) */

struct procent {                /* entry in the process table          */
        uint16  prstate;        /* process state: PR_CURR, etc.        */
        pri16   prprio;         /* process priority                    */
        char    *prstkptr;      /* saved stack pointer                 */
        char    *prstkbase;     /* base of run time stack              */
        uint32  prstklen;       /* stack length in bytes               */
        char    prname[PNMLEN]; /* process name                        */
        uint32  prsem;          /* semaphore on which process waits    */
        pid32   prparent;       /* id of the creating process          */
        umsg32  prmsg;          /* message sent to this process        */
        bool8   prhasmsg;       /* nonzero iff msg is valid            */
        int16   prdesc[NDESC];  /* device descriptors for process      */
};

/* Marker for the top of a process stack (used to help detect overflow) */
#define STACKMAGIC      0x0A0AAAA9

extern  struct  procent proctab[];
extern  int32   prcount;        /* currently active processes          */
extern  pid32   currpid;        /* currently executing process         */
```

5.3 进程状态

为了准确记录进程正在做什么并检验进程操作的有效性，系统给每个进程赋予一个状态。在设计过程中，操作系统设计者定义所有可能的状态。因为很多对进程进行操作的系统函数需要使用这些状态去判定操作是否有效，进程状态必须在系统实现前完整定义。

Xinu 使用进程表中的 prstate 字段记录每个进程的状态信息。系统定义了 7 个有效状态，每个状态都有自己的符号常量。系统同时定义了一个额外的常量用于标识未被使用的进程表项（即没有进程会使用该进程表项）。文件 process.h 包含有相关的定义。图 5-2 列出了所有的状态符号以及每个符号的意义。

常量	意义
PR_FREE	进程表中的表项未被使用（非实际的进程状态）
PR_CURR	进程当前正在执行
PR_READY	进程就绪
PR_RECV	进程正在等待消息
PR_SLEEP	进程正在等待计时器
PR_SUSP	进程处于挂起状态
PR_WAIT	进程正在等待信号量
PR_RECTIM	进程正在等待计时器或消息，无论哪个先发生

图 5-2 表示进程状态的 7 个符号

因为 Xinu 是作为一个嵌入式系统而设计和运行的，所以 Xinu 总是将所有进程的代码和数据都保存在内存中。在大型操作系统中，系统可以将不运行的进程放入二级存储介质中。因此，在那些系统中，进程状态需要表述该进程是在内存中还是在磁盘上。

70
-
71

5.4 就绪和当前状态

后面章节将具体介绍每个进程状态，并三说明如何以及为什么系统函数要改变进程的状态。本章下面几节关注就绪和当前进程状态。

几乎每个操作系统都包含就绪和当前进程状态。一个进程处于就绪状态，如果该进程已经为 CPU 服务做好了准备（即有资格）但目前还没有被执行。而一个正在接收 CPU 服务的进程称为当前进程。

5.5 调度策略

从当前执行的进程转换到另外一个进程，需要两个步骤：从有资格使用 CPU 的进程中挑选一个，然后将 CPU 的控制权交给该进程。执行选择进程策略的软件称为调度器。在 Xinu，函数 resched 使用下面的签名调度策略选择进程：

在任何时候，执行有资格获得 CPU 服务的优先级最高的进程。在优先级相同的情况下，采用时间轮转调度（round-robin）策略。

72

调度器要注意两个方面：

- 当前执行的进程包括在有资格进程的集合中。因此，如果进程 p 正在执行，并且它的优先级比其他任何进程的优先级高，则进程 p 将继续执行。
- 术语轮转调度是指这样的情景。一组包含相同优先级的 k 个进程，这些进程的优先级比其他进程高。轮转调度策略让这组进程中每个成员能够依次获得服务，所以每个成员都会在其他成员获得第二轮执行之前执行。

5.6　调度的实现

理解调度器的关键在于明白调度器仅仅是一个函数。也就是说，操作系统的调度器不是一个从进程拿出 CPU 将其转移到另一个进程的主动代理。相反，某个执行中的进程调用调度器函数[⊖]。

调度器是由执行进程放弃 CPU 时所调用的函数组成。

进程的优先级由一个正整数组成，并且给定进程的优先级存储在进程表项的 prprio 字段。用户给每个进程分配一个优先级来控制进程如何选择接收 CPU 服务。市场上还有许多复杂的调度策略，如观察每个进程的行为，动态地调整优先级的调度器。但是，对于大多数嵌入式系统来说，进程的优先级相对保持静态（典型情况下，进程的优先级在进程创建后就不再改变）。

为了快速地选择新进程，我们的示例系统将所有就绪状态的进程存储在一个链表中，称为就绪链表。在就绪链表中的进程按进程优先级降序排列。因此，最高优先级进程在链表头的位置可以快速地被访问。

在我们的示例代码中，就绪链表存储在第 4 章描述的 queuetab 数组中，调度器使用第 4 章中的函数对链表进行更新和访问。也就是说，就绪链表中每个元素的键值由该元素对应进程的优先级组成。全局变量 readylist 包含对应就绪链表的队列 ID。

73

是否应该将当前进程保持在就绪链表中？答案依赖于具体的实现。整个系统一致遵循的每种解决方案都是可行的。Xinu 实现如下的调度策略：

当前进程不出现在就绪链表中。为了能够快速访问当前进程，当前进程的 ID 存储在一个全局整数变量 currpid 中。

考虑 CPU 从一个进程切换到另一个进程的情况。当前正在执行的进程放弃 CPU。通常，刚才正在执行的进程能够继续使用 CPU。在这种情况下，调度器必须把当前进程的状态改成 PR_READY 并将该进程插入到就绪链表中，使得它在稍后能再一次获得 CPU 服务。但是，如果当前进程不再准备继续执行，则该进程不能放在就绪链表中。

那么调度器如何决定是否将当前进程移到就绪链表呢？在 Xinu 中，调度器不是显式地接收参数来决定如何处理当前进程。相反，该函数使用一个隐式参数：如果当前进程不再保持就绪状态，那么在调用 resched 函数之前，当前进程的 pstate 字段必须设置成所需的下一个状态。无论何时准备切换到新进程，resched 检查当前进程的 pstate 字段。如果进程的状态依旧是 PR_CURR，则 resched 认为进程应该继续准备执行，将进程移到就绪链表中。否则，resched 认为进程的下一个状态已经选定。第 6 章将给出一个具体例子。

除了将当前进程移到就绪链表中外，resched 完成调度的所有细节和上下文切换，除了保存和恢复机器寄存器之外（这不能直接通过像 C 语言这样的高级语言实现）。resched 选择一个新的进程运行，在进程表中更新该进程的表项，将新进程从就绪链表中移除，使其成为当前进程，并且更新 currpid，同时也更新抢占计数器，这些我们将在稍后讨论。最后，resched 调用 ctxsw 保存当前进程的硬件寄存器，从新进程中恢复硬件寄存器。这部分的源代码可以从 resched.c 文件中找到：

⊖　后续章节会解释一个进程如何以及为什么会调用调度器。

```
/* resched.c - resched */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)           /* assumes interrupts are disabled      */
{
        struct procent *ptold;  /* ptr to table entry for old process   */
        struct procent *ptnew;  /* ptr to table entry for new process   */

        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
                Defer.attempt = TRUE;
                return;
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];

        if (ptold->prstate == PR_CURR) {  /* process remains running */
                if (ptold->prprio > firstkey(readylist)) {
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
                insert(currpid, readylist, ptold->prprio);
        }

        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
        ptnew = &proctab[currpid];
        ptnew->prstate = PR_CURR;
        preempt = QUANTUM;              /* reset time slice for process */
        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

        /* Old process returns here when resumed */

        return;
}
```

resched 开始工作时检查全局变量 Defer.ndefers，决定是否需要推迟重新调度。如果需要推迟重新调度，resched 通过设置全局变量 Defer.attempt 来表明在推迟的时间段里尝试过调度，并且返回到 resched 的调用者。提供推迟重新调度是为了调节设备驱动适应那些在一个中断上需要驱动服务多个设备的硬件。到目前为止，只要理解调度可以被暂时推迟即可。

一旦通过推迟条件测试，resched 就检查一个隐式参数：当前进程的状态。如果进程状态变量包含 PR_CURR 并且当前进程的优先级在系统中是最高的，则 resched 返回，当前进程继续运行。如果进程状态表明当前进程应该继续使用 CPU，但是当前进程不具有最高优先级，则 resched 将当前进程放入就绪链表。然后，resched 取出就绪链表首部的进程（最高优先级的进程）执行上下文切换。

因为每个并发进程都有自己的指令指针，想象上下文切换如何发生可能是非常困难的。为了

说明并发性是如何操作的。设想进程 P_1 正在运行并且调用 resched。如果 resched 选择切换到进程 P_2，则进程 P_1 将在调用 ctxsw 处停止。但是，进程 P_1 能够运行，并且可以执行任意的代码。稍后，当 resched 切换回 P_1 时，执行点将在离开处——调用 ctxsw 处恢复。P_1 的执行位置不会改变，因为使用 CPU 的是 P_2。当进程 P_1 运行，对 ctxsw 的调用将返回到 resched。后面章节将进一步考虑相关的细节。

5.7 上下文切换的实现

因为寄存器和硬件状态不能直接通过高级语言来进行操作，所以 resched 调用一个用汇编语言编写的函数 ctxsw 来执行从一个进程到另一个进程的上下文切换。当然，ctxsw 的代码依赖于具体的机器。最后一步包括重新设置程序计数器（即，跳转到新进程的位置）。在 Xinu 中，程序的所有部分都保留在内存中，因此新进程的文本段也在内存中。关键之处在于操作系统必须在跳转进入新进程之前加载新进程的所有状态变量。一些体系结构包含用于上下文切换的两条原子指令：一条将处理器状态信息存储在连续的内存单元中，另外一条将处理器状态信息从连续的内存单元中加载。在这种体系结构中，上下文切换代码执行一条指令将处理器状态保存在当前进程栈中，用另一条指令加载新进程的处理器状态。当然每条指令都花费多条指令周期。RISC 体系结构用较长的指令序列来实现 ctxsw，但是每条指令都能快速执行。

5.8 内存中保存的状态

为了理解 ctxsw 如何保存处理器状态，设想系统中的内存有三个活动进程：其中两个是就绪状态，一个正在运行。每个进程都拥有一个栈，当前正在执行的进程使用它自己的栈。当执行进程调用一个函数时，它必须在栈上分配空间来存储参数和局部变量。当它从函数返回时，这些变量将从栈上释放。 76
如果上下文切换函数在进程栈上为进程保存机器状态，那么我们将会找到它们何时正在执行，另外两个进程由于在它们最后运行时执行了上下文切换，所以它们将变量压入它们的栈中。图 5-3 说明了这种情况。

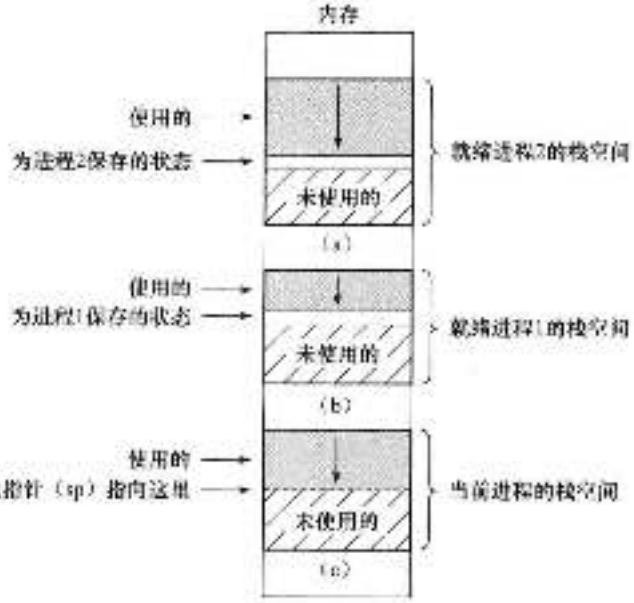


图 5-3 a）内存中栈的解释，b）处于就绪链表中的进程，c）正在执行的进程

5.9 在 MIPS 处理器上切换上下文

在 RISC 体系结构上，例如 MIPS，每条指令只能读写一个寄存器。因此，保存 N 个寄存器需要执

行 N 条指令。像大多数操作系统一样，Xinu 通常把状态保存在进程栈中的惯例。因此，我们的 MIPS 版本的 ctxsw 执行 4 个基本步骤：

- 当调用 ctxsw 时，执行一系列顺序指令将处理器寄存器的内容保存在进程的栈中。
- 将当前进程栈指针保存到进程表项中，加载“新”进程的栈指针。
- 执行一系列顺序指令将以前保存在新进程栈中的值重新加载到处理器寄存器。
- 跳转到新进程恢复执行点之处继续执行。

77

因为上下文切换涉及对处理器寄存器的直接操作，所以代码都用汇编语言编写。为了保存处理器寄存器的内容，ctxsw 在栈上分配 CONTEXT 字节空间。该空间足够容纳所有保存的寄存器值。然后 ctxsw 在分配的空间上按字顺序保存寄存器值。

第二步，直接保存和恢复栈指针，只需要两条指令。ctxsw 接收两个指针作为参数：当前进程栈指针在进程表中保存的单元地址和进程表中包含新的栈指针的地址。因此，一条指令保存栈指针到第一个参数给定的位置，另一条指令从第二个参数给定的地址加载栈指针。

第二步结束后，栈指针指向新进程的栈。ctxsw 将保存在进程栈上的值取出，载入处理器寄存器中。一旦这些值被加载，ctxsw 就从栈中移除 CONTEXT 字节。文件 ctxsw.S 包含相关代码。

```
/* ctxsw.S - ctxsw */

#include <mips.h>

.text
        .align 4
        .globl  ctxsw

/*------------------------------------------------------------------------
 *  ctxsw     Switch from one process context to another
 *------------------------------------------------------------------------
 */
        .ent ctxsw
ctxsw:
        /* Build context record on the current process' stack */

        addiu   sp, sp, -CONTEXT
        sw      ra, CONTEXT-4(sp)
        sw      ra, CONTEXT-8(sp)

        /* Save callee-save (non-volatile) registers */

        sw      s0, S0_CON(sp)
        sw      s1, S1_CON(sp)
        sw      s2, S2_CON(sp)
        sw      s3, S3_CON(sp)
        sw      s4, S4_CON(sp)
        sw      s5, S5_CON(sp)
        sw      s6, S6_CON(sp)
        sw      s7, S7_CON(sp)
        sw      s8, S8_CON(sp)
        sw      s9, S9_CON(sp)

        /* Save outgoing process' stack pointer */

        sw      sp, 0(a0)

        /* Load incoming process' stack pointer */

        lw      sp, 0(a1)
```

```
/* At this point, we have switched from the run-time stack */
/*    of the outgoing process to the incoming process      */

/* Restore called save (non-volatile) registers from new stack  */

lw      s0, S0_CON(sp)
lw      s1, S1_CON(sp)
lw      s2, S2_CON(sp)
lw      s3, S3_CON(sp)
lw      s4, S4_CON(sp)
lw      s5, S5_CON(sp)
lw      s6, S6_CON(sp)
lw      s7, S7_CON(sp)
lw      s8, S8_CON(sp)
lw      s9, S9_CON(sp)

/* Restore argument registers for the new process */

lw      a0, CONTEXT(sp)
lw      a1, CONTEXT+4(sp)
lw      a2, CONTEXT+8(sp)
lw      a3, CONTEXT+12(sp)

/* Remove context record from the new process' stack */

lw      v0, CONTEXT-4(sp)
lw      ra, CONTEXT-8(sp)
        addiu   sp, sp, CONTEXT

        /* If this is a newly created process, ensure */
        /*    it starts with interrupts enabled       */

        beq     v0, ra, ctxdone
        mfc0    v1, CP0_STATUS
        ori     v1, v1, STATUS_IE
        mtc0    v1, CP0_STATUS

ctxdone:
        jr      v0
        .end ctxsw
```

正如代码注释中所说明的，新创建的进程应该以允许中断的形式执行；其他进程从对上下文切换的调用处返回，并且以禁止中断的形式执行 resched。因为 MIPS 使用协处理器处理中断，所以上下文切换时必须使用协处理器显式地允许中断。我们的实现使用了一个技巧：当创建进程时，保存在 CONTEXT－4 单元(sp) 和 CONTEXT－8 单元(sp) 的值不同，而对于其他进程，它们是相同的。在最后一步，ctxsw 比较两个值，如果它们不相同，则允许中断。

5.10 重新启动进程执行的地址

上下文切换产生的一个潜在问题是：在执行上下文切换期间，处理器将继续执行。这意味着寄存器会可以改变的。因此，代码必须保证一旦某个寄存器已经被保存，该寄存器就不会发生改变（否则，当进程重新开始运行时会丢失该值）。程序计数器呈现出特殊的困境。因为保存该值意味着当进程最新执行时，会从代码中指令指针所指的位置处继续执行。如果当指令指针保存时上下文切换还没有完成，那么进程将会在上下文切换还没有发生着重新执行。ctxsw 的代码说明了如何解决这个问题：选择进程恢复执行时的地址，而不是在 ctxsw 运行时保存程序计数器的值。

为了理解如何选取一个合适的地址，考虑一个正在执行的进程。进程调用 resched，然后再到后

78 ~ 80

ctxsw。代码保存一个好像进程刚从调用 ctxsw 的过程返回所用的地址，而不是当试保存 ctxsw 中程序计数器的值。也就是说，这个保存在程序计数器中的值是返回地址，把 ctxsw 当做普通的过程调用时会返回的地址。因此：

当进程重新运行时，进程在调用 ctxsw 后的位置开始执行。

所有调用 resched 的进程执行上下文切换，resched 调用 ctxsw。因此，如果某个进程在任意时刻释放系统并睡眠内存，那么为每一个就绪进程保存的程序计数器都具有相同的值——在 resched 调用 ctxsw 后的那个地址。但是，每个进程都拥有自己的函数调用栈，这意味着当一个进程恢复执行并从 resched 返回时，返回能够跳转至不同的调用者之后。

函数返回的概念是保持系统干净设计的一个重要成分。函数调用向下处理，通过系统的各个层次，每个调用都返回。为了强制每一层的设计，调度器 resched、上下文切换 ctxsw，都类似其他过程调用和返回。概括来讲：

在 Xinu 中，每个函数，包括调度器和上下文切换，最终都返回到它的调用者。

当然，重新调度允许其他进程执行，并且可能执行任意长时间（依赖于进程优先级）。因此，在调用 resched，调度返回和进程重新执行之间可能有比较大的一段延时。

5.11 并发执行和 null 进程

并发执行抽象是完全和绝对的。就是说，操作系统将所有的计算视为进程的一部分——CPU 没有其他的办法临时停止执行进程而去执行另外一小段单独的代码。调度器的设计反映了下面的准则：调度器的唯一功能就是使处理器在正在执行和一系列准备运行的进程之间切换。调度器不能执行进程之外的任何代码，也不能创建新的进程。图 5-4 说明了进程可能的状态转换。

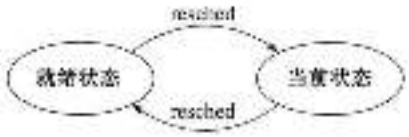


图 5-4 进程在就绪状态和当前状态之间切换的状态转换图

81

我们将看到一个进程不会总是处在准备执行状态。例如，当进程等待 I/O 操作完成或者需要使用正在被使用的共享资源时，就会停止执行。那么，如果所有的进程都在等待 I/O 操作会发生什么？由于代码在设计时假设在任何时候都至少有一个合适的进程可以调度运行，所以 resched 将会失败。当当前正在执行的进程被阻塞时，resched 从就绪链表中取出第一个进程而不检测链表是否为空。如果链表为空，就会造成错误结果。综上所述：

因为操作系统只能将 CPU 从一个进程切换到另外一个进程，所以在任何时候都至少有一个进程准备执行。

为了保证至少有一个进程总是准备执行，Xinu 使用标准的技术：它在系统启动时创建一个称为 null 进程的额外进程。null 进程的进程 ID 为 0 和优先级为 0（比任何进程的优先级都低）。null 进程的代码包含无限循环，将在第 22 章中说明。由于所有其他的进程都必须有高于 0 的进程优先级，所以调度器只有当不存在其他准备运行的进程时才会切换到 null 进程。本质上，当其他所有进程都被阻塞时（如等待 I/O 操作完成），操作系统才会将 CPU 切换到 null 进程[⊖]。

5.12 使进程准备执行和调度不变式

当 resched 需要将当前进程移入就绪链表时，它直接对链表进行操作。由于使进程准备获得 CPU 服务发生的非常频繁，我们设计一个函数来执行该操作。该函数称为 ready。

ready 需要两个参数：进程 ID 和一个控制是否需要调用 resched 的布尔变量。通过使用符号常量 RESCHED_YES 和 RESCHED_NO，调用 ready 的目的将更加清晰。

考虑第二个参数。我们的调度策略决定在任何时刻最高优先级的可执行进程必须获得执行。因此，

⊖ 某些处理器包含特殊的指令，这些指令在 null 进程中停止 CPU 执行直到发生中断。使用这些特殊指令能减少 CPU 的电能消耗。

如果最高优先级进程处于就绪链表中，ready 就应该调用 resched 来保证调度策略的正确性。我们说每个操作系统函数应该维护一个调度不变式：这个函数假设当该函数被调用时，最高优先级的进程正在执行，并且保证当该函数退出时，最高优先级的进程依旧在执行。因此，如果一个函数改变了进程的状态，那么该函数必须调用 resched 来重新建立调度不变式。ready 是一个例外。为了理解其中的原因，必须知道有些操作系统的函数将多个进程移入就绪链表（例如，多个计时器事件可能发生在同一时刻）。在此过程中重新调度可能造成一些未完成的操作。解决方案是通过暂时地推迟调度策略，允许多个使用 RESCHED_NO 参数的 ready 调用。一旦将这些进程加入到就绪链表中，就必须调用一次 resched 重新建立进程调度状态，以使最高优先级的进程处于执行状态。我们将在第 7 章看到使用 ready 的例子。此时，明白当第二个参数为 RESCHED_NO 时 ready 怎样避免重新调度就足够了。文件 ready.c 包含了该段代码。 [82]

```
/* ready.c - ready */

#include <xinu.h>

qid16	readylist;			/* index of ready list		*/

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status	ready(
	  pid32		pid,		/* ID of process to make ready	*/
	  bool8		resch		/* reschedule afterward?	*/
	)
{
	register struct procent *prptr;

	if (isbadpid(pid)) {
		return(SYSERR);
	}

	/* Set process state to indicate ready and add to ready list */

	prptr = &proctab[pid];
	prptr->prstate = PR_READY;
	insert(pid, readylist, prptr->prprio);

	if (resch == RESCHED_YES) {
		resched();
	}
	return OK;
}
```

5.13　推迟重新调度

resched 使用全局变量 Defer.ndefers 来决定重新调度是否为 deferred（推迟）状态[⊖]。函数 sched_cntl 提供了一个接口用来控制推迟。调用者像这样使用该函数： [83]

```
sched_cntl(DEFER_START);
```

来推迟重新调度，并使用：

```
sched_cntl(DEFER_STOP);
```

来结束推迟。在某些系统中，中断处理器能够被更高优先级的设备所中断，为了适应这些系统，使用 Defer.ndefers 作为一个参考计数器。当一个处理器请求推迟开始的时候，计数增加；而当一个过程结

⊖　参见 5.6 节中的 resched 代码。

变量推迟阶段的时候，计数减少。当计数变为0的时候，sched_cntl检查全局变量Defer.ndefers来判断在推迟期间resched是否被调用过。如果被调用过，在返回调用者之前，sched_cntl调用resched。在文件sched.h中，定义了一些用于推迟的常量和变量。

```
/* sched.h */

/* Constants and variables related to deferred rescheduling */

#define DEFER_START     1       /* start deferred rescheduling          */
#define DEFER_STOP      2       /* stop  deferred rescheduling          */

/* Structure that collects items related to deferred rescheduling       */

struct  defer   {
        int32   ndefers;        /* number of outstanding defers         */
        bool8   attempt;        /* was resched called during the        */
                                /*   deferral period?                   */
};

extern  struct  defer   Defer;
```

文件sched_cntl.c中包含了用于控制推迟的代码。

```
/* sched_cntl.c - sched_cntl */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  sched_cntl  -  control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  sched_cntl(             /* assumes interrupts are disabled      */
          int32 def             /* either DEFER_START or DEFER_STOP     */
        )
{

        switch (def) {

            /* Process request to defer:                                */
            /*  1) Increment count of outstanding deferral requests     */
            /*  2) If this is the start of deferral, initialize Boolean */
            /*          to indicate whether resched has been called     */

            case DEFER_START:
                if (Defer.ndefers++ == 0) {     /* increment deferrals  */
                        Defer.attempt = FALSE;  /* no attempts so far   */
                }
                return OK;

            /* Process request to stop deferring:                       */
            /*  1) Decrement count of outstanding deferral requests     */
            /*  2) If last deferral ends, make up for any calls to      */
            /*          resched that were missed during the deferral    */

            case DEFER_STOP:
                if (Defer.ndefers <= 0) {       /* none outstanding */
                        return SYSERR;
                }
```

```
            if ( --Defer.ndefers == 0) {        /* end deferral period  */
                if (Defer.attempt) {            /* resched was called   */
                    resched();                  /*   during deferral    */
                }
            }
            return OK;

        default:
            return SYSERR;
        }
    }
```

5.14 其他进程调度算法

进程调度曾经是操作系统中的一个重要课题，研究人员已经提出来许多调度算法用来替代像 Xinu 所用的轮转调度算法。例如，有一种策略衡量进程执行的 I/O 量，并把 CPU 交给花费最多时间执行 I/O 的那个进程。支持者认为，因为 I/O 设备的速度比处理器慢，所以选择一个执行 I/O 的进程能够增加系统的总吞吐量。

[85]

由于调度的方法有限，测验 Xinu 中的调度策略是很容易的。改变 resched 和 ready 会改变基本的调度器。

5.15 观点

调度和上下文切换最有趣的方面在于，它们嵌入正常计算中成为其中的一部分。也就是说，与操作系统及用户进程分开实现不同，这里的操作系统代码是进程本身执行的。因此，如果系统没有额外的进程可以停止处理器执行一个应用程序并把它移动到另一个处理器上，调度和上下文切换就会作为函数调用的副作用而存在。

我们将会看到，使用进程执行操作系统代码会影响到设计。当程序员编写操作系统函数时，程序员必须适应并发进程的执行。同样，使用进程执行操作系统代码还会影响系统如何与 I/O 设备交互，以及如何处理中断。

5.16 总结

调度和上下文切换形成了并发执行的基础。调度指的是从可以执行的进程中挑选一个。上下文切换包含了停止一个进程并开始一个新的进程。为了跟踪进程，系统使用一个称为进程表的全局数据结构。每当它暂时挂起一个进程时，上下文切换为进程保存进程栈中的处理器状态，并在进程表中放置一个栈指针。当重新启动一个进程时，上下文切换从进程栈中重新加载处理器状态信息，并从上下文切换函数调用的返回点恢复执行进程。

为了确定一个操作何时是允许的，每个进程都分配了一个状态（state）。正在使用 CPU 的进程分配到的是当前（current）状态，可使用 CPU 但并不是当前执行的进程分配的是就绪（ready）状态。因为任何时间都必须保证至少有一个进程能够执行，所以操作系统在启动时创建了一个称为空进程（null process）的额外进程。空进程的优先级为 0，而所有其他进程的优先级均大于 0。因此，空进程只有在没有其他可运行的进程时才运行。

本章介绍了三个在当前和就绪状态之间进行切换的函数。resched 执行调度，ctxsw 执行上下文切换，ready 确保进程有资格执行。

[86]

练习

5.1 如果操作系统一共有 N 个进程，那么在给定时间内有多少进程可以处于就绪链表中？请给出说明。

5.2 操作系统的函数如何知道在给定的时间内哪个进程正在执行？

5.3 重写 resched 方法，要有一个明确的参数表明正在执行进程的处理结果，并检查生成的汇编代码以确定在每种情况下执行的指令数。

5.4 上下文切换期间执行的基本步骤是什么？

5.5 研究另一个硬件架构（例如，在 Intel x86）并确定什么样的信息需要在上下文切换期间保存下来。

5.6 需要多少内存来存储一个 MIPS 处理器的处理器状态？哪些寄存器必须保存？为什么？处理器的标准调用规约如何影响结果？

5.7 假设进程 A 已在就绪队列中。当进程 A 变成当前状态时，执行将从哪里开始？

5.8 为什么需要一个空进程？

5.9 考虑对存储处理器状态的代码进行修改，将其存储在进程表中而不是进程栈中（即假设进程表项中包含一个数组，保存寄存器的内容）。每一种方法的优点是什么？

5.10 在前面的练习中，在进程表中保存寄存器信息增加还是减少了上下文切换期间执行的指令数？

5.11 设计一个双核处理器（即包含两个可以并行执行的独立 CPU 的处理器）的调度策略。

5.12 扩展前面的练习：说明在一个核上执行 resched 可能需要改变正在运行在该核上的进程（注：许多双核处理器的操作系统通过指定所有的操作系统函数（包括调度）来避免这个问题，以此来运行其中一个核上的进程）。

5.13 代码包含了两个用于推迟重新调度的机制：ready 有一个允许调用者避免重新调度的参数，sched_ctl 设置了一个全局的位使 resched 忽略调用。为什么要包含这两种机制？提示：比较效率和开销。

87

更多进程管理

当人们停止思想之时，正是科学死亡之时。

——佚名

6.1 引言

第5章讨论了并发执行的抽象和执行进程。本章介绍操作系统如何在一个表中存储进程的信息，以及如何为每个进程分配一个状态。第5章还介绍了调度和上下文切换的概念。它展示了调度者如何完成一个调度决策，以及进程如何在就绪状态和当前状态之间切换。

本章则拓展我们对于进程管理的知识。本章介绍如何产生一个新的进程，以及当进程退出时会发生什么。本章还检查允许一个进程暂时挂起的进程状态，并探讨在当前、就绪和挂起状态之间切换进程的方法。

6.2 进程挂起和恢复

操作系统函数有时需要暂时停止一个进程的执行，并在以后恢复其执行。我们说已停止的进程处于"假死"状态。假死是很有必要的，例如，当一个进程等待几个重启条件之一，但并不知道哪个条件将会最先满足。

实现操作系统功能的第一步是定义一组操作。在假死情况下，理论上仅需两个操作就可以提供所有需要的功能：

- 挂起：一个进程并将其设置为假死（即让进程无资格使用CPU）。
- 恢复：继续执行之前挂起的进程（即让进程有资格使用CPU）。

因为没有资格使用CPU，所以一个挂起的进程不能保持就绪或当前状态。因此，必须发明一个新的状态。我们将这个状态叫做挂起，然后在状态图中加入这个新状态并将它和某些转换关联起来。图6-1展示了扩展的状态图，总结了挂起和恢复如何影响进程的状态。结果图说明了在就绪、当前和挂起状态之间的可能转换。



图6-1 当前、就绪和挂起状态之间的转换

6.3 自我挂起和信息隐藏

尽管图6-1中的每个状态都有一个标签指定了一个特定函数，但进程挂起和进程管理还是存在一个重要区别：挂起允许一个进程挂起另一个，而并不仅仅是作用在当前进程上。恢复必须允许一个正在执行的进程恢复一个之前被挂起的进程。因此，挂起和恢复均需要指定应该执行进程的进程ID。

一个进程可以挂起自己吗？是的。为了这么做，进程必须获取自身进程ID并将其作为参数传递给挂起函数。由于全局变量currpid记录了当前正在执行的进程，所以一个自我挂起可以如下实现：


```
suspend(currpid);
```

然而，一个设计良好的操作系统应遵循信息隐藏的原则：实现细节一般不暴露。因此，Xinu并不允许进程直接接触全局变量，如currpid，而是调用一个叫做getpid的函数来获取进程ID。因此，要挂起自身，进程这样调用：

```
suspend(getpid());
```

在上述例子里getpid的实现仅仅返回了currpid的值，这看起来似乎没什么必要。然而，当考虑改

动操作系统时，信息隐藏的好处就会体现出来。如果所有进程都调用 getpid，设计者就可以改变进程 ID 的存储位置和存储方式的细节，而不需要改变其他代码。

这里的重点在于：

> 好的系统设计遵循信息隐藏的原则，除非实在有必要，否则实现细节是不暴露的。隐藏这些细节使得可以在不重写调用函数的情况下更改函数的实现。

6.4 系统调用的概念

理论上，进程恢复直接用了。进程必须处于就绪状态下并插入就绪链表的正确位置。由于第 5 章描述的 ready 函数执行这两个任务，所以看起来恢复好像是不需要的。然而，实际上，恢复增加了一层额外的保护：它不对调用者或参数正确性做假设，任意一个进程可以在任意时间以任意参数调用恢复。

我们使用术语系统调用来区分像恢复这样的函数和像就绪这样的内部函数。一般来说，我们认为一组系统调用就是定义了一种从外部观察操作系统的视角——应用程序进程调用系统调用来获取服务。除了增加一层保护外，系统调用接口是另一个信息隐藏的例子：应用程序对内部实现并不知晓，仅仅使用一组系统调用来获取服务。我们将会看到系统调用和其他方法的区别贯穿于整个操作系统。下面总结一下：

> 系统调用为应用程序定义了操作系统服务，它保护系统以避免非法使用，并隐藏或是实现的信息。

91

为了提供保护，系统调用处理上层函数所不做的三个方面的计算：

- 检查所有参数。
- 确保修改会保留全局数据结构状态的一致性。
- 向调用者报告成功或失败。

本质上，系统调用并不对调用它的进程做任何假定。因此，系统调用会检查每个参数，而不是假定调用者提供了正确而有意义的参数。更重要的是，许多系统调用会改变操作系统的数据结构，如进程表和用队列存储的进程链表。系统调用必须确保没有其他进程试图同时改变数据结构，否则就会产生不一致性。因为系统调用不能假定在何种情况下它将被调用，所以必须采取措施来防止其他更改数据结构的进程并发执行。这包含两个方法：

- 避免任何自动让出 CPU 的函数调用。
- 禁用中断以防止被迫让出 CPU。

为了防止自动让出 CPU，系统调用必须避免直接或间接地调用 resched。也就是说，当正在进行修改时，系统调用不能直接调用 resched 或者任何调用 resched 的函数。为了防止被迫让出 CPU，系统调用禁止中断直至修改完成。在第 12 章中，我们将明白其中的原因：硬件中断可以导致重新调度，因为某些中断例程会调用 resched。

一个示例系统调将有助于阐明这两个方面，看一下文件 resume.c 中的代码：

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
          pid32          pid             /* ID of process to unsuspend   */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
```

```
        pri16   prio;                   /* priority to return           */

        mask = disable();
        pptr = &proctab[pid];
        if (isbadpid(pid) || (pptr->prstate != PR_SUSP)) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prio = pptr->pxpriority;        /* record priority to return    */
        ready(pid, RESCHED_YES);
        restore(mask);
        return prio;
}
```

6.5 禁止中断和恢复中断

恢复中的代码检查参数 pid，以确保调用者提供了一个有效的进程 ID 并且指定的进程处于挂起状态。然而，在执行任何计算之前，恢复首先保证中断不会发生（即除非恢复调用操作系统函数导致上下文切换，否则上下文切换不会发生）。为了控制中断，恢复使用这样一对函数[⊖]：

- 函数 disable 禁止中断并且返回先前的中断状态给调用者。
- 函数 restore 从先前保存的值中重新载入中断状态。

恢复在入口处立即禁止中断。恢复可以在两种情况下返回：因为检测到一个错误或者因为恢复成功完成操作请求。无论哪种情况，恢复都必须在返回前调用还原（restore），以重置中断状态，使调用者使用的值与调用开始时相同。

没有编写操作系统代码经验的程序员往往期望系统调用返回前启用中断。然而，还原更具一般性。因为它还原了中断，而不是简单地启用它们，无论在调用期间中断是启用的还是禁止的，恢复都能正确工作。一方面，如果一个函数调用恢复时禁止了中断，那么调用已将以中断禁止的状态返回。另一方面，如果一个函数调用恢复时启用了中断，那么调用也将以中断启用的状态返回。

92
:
93

> 系统调用必须禁止中断来防止其他进程改变全局数据结构，使用一个禁止/还原范例来增加一致性。

6.6 系统调用模板

我们可以从另一个角度来思考中断处理，将注意力集中到系统函数必须维护的不变性上：

> 操作系统函数必须总是以它被调用时一样的中断状态返回给其调用者。

为了确保这个不变性，操作系统函数遵循图 6-2 所列的适用方法。

94

6.7 系统调用返回 SYSERR 和 OK 值

我们将看到在些系统调用返回一个和正在执行的函数相关的值，另一些仅仅返回一个状态指示调用成功。恢复是前者的一个例子：它返回被恢复进程的优先级。在恢复的例子中，必须注意在调用恢复之前记录优先级，因为要恢复过程的优先级可能高于当前执行的进程。因此，一旦就绪指定的进程放置到就绪链表中并调用 resched 时，就可能开始执行新的进程。事实上，任意延迟都可能发生在恢复调用就绪和调用执行之间。在延迟期间，可以执行任意数量的其他进程，并且这些进程可能会终止。因此，为了确保在恢复时，返回的优先级反映了被恢复进程的优先级，恢复在调用就绪之前准备一个副本在局部变量 prio 中。恢复使用该局部副本作为返回值。

⊖ 第 12 章给出中断处理的细节。

```
syscall function_name(args) {
        intmask mask;                   /* 中断信息                */
        mask = disable();               /* 在函数开始时禁止中断*/
        if ( args are incorrect ) {
                restore(mask);          /* 在返回错误前还原中断*/
                return(SYSERR);
        }
        ...other processing...
        if ( an error occurs ) {
                restore(mask);          /* 在返回错误前还原中断*/
                return(SYSERR);
        }
        ...more processing...
        restore(mask);                  /* 在正常返回前还原中断*/
        return( appropriate value );
}
```

图6-2 操作系统函数的通用一般形式的图例

Xinu 定义了两个常量作为整个系统中使用的返回值。函数返回 SYSERR，表明在处理过程中发生了错误。也就是说，如果参数不正确（如超出可接受的范围）或请求的操作无法成功完成，系统函数返回 SYSERR。有的函数（如就绪）不会计算一个特定的返回值，而使用常量 OK 表明操作成功。

6.8 挂起的实现

如图6-1所示，suspend 函数只能用于处于运行或者就绪状态的进程。对于就绪进程，挂起操作的实现并不复杂，只需要从就绪链表中移除该进程，并且修改进程状态为挂起。在从就绪链表中删除该进程并修改进程状态为 PR_SUSP 之后，suspend 函数将会恢复中断并且返回到调用者。被挂起的进程不会占用 CPU 直到其再次恢复执行为止。

挂起当前正在运行的进程同样容易。按照在第5章中列出的步骤，suspend 函数必须修改当前正在运行进程的状态为 PR_SUSP，调用 reached 函数。也就是说，suspend 函数将当前正在运行进程的状态设置为预期的下一个状态。

95 在 suspend.c 文件中能够找到 suspend 函数的实现代码。

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
        pid32           pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        pri16   prio;                   /* priority to return           */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)) {
```

```
        restore(mask);
        return SYSERR;
    }

    /* Only suspend a process that is current or ready */

    prptr = &proctab[pid];
    if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
        restore(mask);
        return SYSERR;
    }
    if (prptr->prstate == PR_READY) {
        getitem(pid);                   /* remove a ready process   */
                                        /* from the ready list      */
        prptr->prstate = PR_SUSP;
    } else {
        prptr->prstate = PR_SUSP;       /* mark the current process */
        resched();                      /* suspended and reschedule */
    }
    prio = prptr->prprio;
    restore(mask);
    return prio;
}
```

和 resume 函数一样，suspend 函数也是一个系统调用。这就意味着，当该函数被调用的时候将禁止中断。此外，suspend 函数还检查参数 pid 是否为合法的进程标识号。因为挂起操作只能用于就绪或当前进程，所以代码还需要检查进程状态是否为这两种状态之一。如果发现了错误，suspend 函数就恢复中断并返回 SYSERR 给调用者。 96

6.9 挂起当前进程

在挂起当前执行进程的代码中，有两点值得关注。第一，当前执行的进程至少会暂时停止执行。因此当前进程在挂起之前，必须预先安排好其他进程在以后继续执行（否则当前进程将会永远处于挂起状态）。第二，由于当前进程将挂起，所以它必须允许其他进程执行。因此，在挂起当前进程的时候，suspend 函数必须调用 resched 函数。其中的关键在于，当一个进程挂起自己的时候，进程将会继续执行直到 resched 函数选择了其他进程并且完成了上下文切换。

需要注意的是，当进程挂起时，resched 函数并不会将该进程放置在就绪链表中。实际上，挂起的进程也并不放置在一个类似于就绪链表那样的挂起链表中。就绪进程存放在一个有序链表中只是为了在重新调度的时候加快对高优先级进程的搜索。因为系统在寻找进程继续执行的时候不会考虑挂起的进程，所以也就没有必要把挂起的进程用链表进行维护。因此，在挂起进程之前，程序员必须安排一种方法使进程以后能继续执行。

6.10 suspend 函数的返回值

suspend 函数和 resume 函数一样，返回挂起进程的优先级给调用者。对于一个就绪进程，返回值将反映在 suspend 函数调用时该进程的优先级（一旦 suspend 函数禁止中断，任何其他进程都不能修改优先级，所以能够在 suspend 函数恢复中断前的任何时候记录优先级）。然而，对于当前正在运行的进程，问题就出现了：suspend 函数返回的优先级是 suspend 函数调用时该进程的优先级还是该进程在以后被继续执行后所拥有的优先级（换言之，在 suspend 返回之后）。从代码的角度而言，问题就是对优先级的记录必须在调用 resched 之前还是之后（上面的代码在调用之后记录）。

为了理解为何返回进程继续执行之后的优先级，可以考虑优先级如何用来传递信息。例如，假设进程需要挂起直到某两个事件中的任何一个发生。程序员可以赋予每个事件一个唯一的优先级数值（例如，25 和 26），并且在与两个事件关联的对 resume 函数调用中分别将优先级设定为对应的

值。之后，挂起的进程在返回继续执行之后就可以通过返回的优先级确定哪个事件的触发导致其继续执行：

```
newprio = suspend( getpid() );
if (newprio == 25) {
        ... event 1 occurred...
} else {
        ... event 2 occurred...
}
```

6.11 进程终止和进程退出

尽管 suspend 函数临时冻结了进程状态，但是 suspend 保存了进程的相关信息，所以进程仍然可以在之后被唤醒。另一个系统调用 kill 通过从系统中完全移除进程来实现进程的终止。一旦进程被杀死，它将无法被重新运行，因为 kill 已经彻底清除了该进程的所有记录并且释放了该进程的进程表中的表项。

kill 所进行的操作取决于进程状态。在编写代码之前，设计者需要考虑每个可能出现的进程状态和在该状态下终止进程可能产生的情况。例如，我们将会看到，处于就绪、睡眠、或者等待状态的进程存放在一个用链表实现的队列数据结构中，这也就意味着 kill 必须先让该进程出队。在第 7 章中，我们将会看到如果一个进程正在等待一个信号量，那么 kill 必须调整该信号量的计数。在我们检查了进程状态和那些控制状态的函数之后，以上情况都将变得清楚明了。目前，理解 kill 系统调用的整体结构和处理处于当前和就绪状态的进程的方法就已经足够了。在 kill.c 文件中可以找到 kill 系统调用的代码。

kill 首先检查输入参数 pid 以确保该参数对应一个合法的进程而不是空进程（空进程不能被杀死，因为它必须保持运行状态）。接下，kill 减小 prcount 变量，该个局变量记录活动的用户进程数。之后调用 freestk 函数释放分配给进程栈的内存空间。剩下的操作取决于进程状态。对于一个处于就绪状态的进程，kill 将会从就绪链表中移除该进程并在进程表中将该进程对应表项的状态设置为 PR_FREE，从而释放该进程的进程表项。由于该进程从就绪链表中移除了，所以在重新调度的时候该进程就不会被选中。因为在进程表中该进程对应的状态为 PR_FREE，所以它在进程表中的条目就可以被回收重用了。

现在，让我们来考虑一下当 kill 需要终止当前正在执行的进程时会发生什么。我们称之为进程退出（exit）。与之前一样，kill 首先检查参数并且减小活动进程数。如果当前进程正好是最后一个用户进程，那么减小 prcount 变量将会使之变为 0，在这种情况下 kill 将调用 xdone 函数，这将在之后进行解释。在将当前进程状态标记为释放之后，kill 调用 resched 函数将 CPU 控制权转交给其他的就绪进程。

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  kill  -  Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
        pid32       pid             /* ID of process to kill        */
        )
{
        intmask mask;               /* saved interrupt mask         */
        struct  procent *prptr;     /* ptr to process' table entry  */
        int32   i;                  /* index into descriptors       */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)
```

```
        || (!prptr = &proctab[pid])->prstate) == PR_FREE) {
            restore(mask);
            return SYSERR;
        }

        if (--prcount <= 1) {           /* last user process completes  */
            xdone();
        }

        send(prptr->prparent, pid);
        for (i=0; i<3; i++) {
            close(prptr->prdesc[i]);
        }
        freestk(prptr->prstkbase, prptr->prstklen);

        switch (prptr->prstate) {
        case PR_CURR:
            prptr->prstate = PR_FREE;       /* suicide */
            resched();

        case PR_SLEEP:
        case PR_RECTIM:
            unsleep(pid);
            prptr->prstate = PR_FREE;
            break;

        case PR_WAIT:
            semtab[prptr->prsem].scount++;
            /* fall through */

        case PR_READY:
            getitem(pid);           /* remove from queue */
            /* fall through */

        default:
            prptr->prstate = PR_FREE;
        }

        restore(mask);
        return OK;
}
```

当最后一个用户进程退出之后，kill 调用 xdone 函数。在某些系统中，xdone 将会关闭设备。在我们的例子中，xdone 仅仅在控制台上打印一条消息，关闭标志系统运行的 LED 屏幕，并且将处理器停机。该段代码能够在 xdone.c 中找到。

```
/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  xdone  -  Print system completion message as last thread exits
 *------------------------------------------------------------------------
 */
void    xdone(void)
{
        kprintf("\r\n\r\nAll user processes have completed.\r\n\r\n");
        qpinBOOFF(SPIO_LED_CISCOWRT);  /* turn off LED "run" light     */
        halt();                        /* halt the processor           */
}
```

为什么 kill 函数一定要调用 xdone？这么做看似没有必要，因为 xdone 中的代码非常容易就加入到 kill 函数中。用一个函数将这些操作包装起来是出于功能分解的考虑。这样，程序员就可以通过修改 xdone 函数来修改所有进程退出之后所做的操作而不需要修改 kill 函数本身。

现在还有一个更严重的问题，在最后一个用户进程从系统中移除之前，xdone 函数就被调用了。为了理解这个问题，考虑这样一个容错设计，当所有进程退出之后，调用 xdone 函数将自动重启进程。在当前实现中，当 xdone 被调用的时候，进程表中仍然有一个进程表项被使用（从而可能导致重启之后系统中已经有一个用户进程）。在本章练习中将会考虑另外一种备选实现。

99
~
100

6.12 进程创建

正如我们所看到的那样，进程是动态的——进程能够在任何时刻创建。系统调用 create 启动一个新的、独立的进程。本质上，create 创建进程的映像就好像正在运行的进程被停止了一样。一旦映像被创建并且进程放置到就绪链表中，ctxsw 就能够切换到该进程。

让我们看看 create.c 的源代码以解释其中的细节。create 函数使用 newpid 函数从进程表中获取一个空闲（未使用）的条目。一旦找到了空闲的条目，create 函数将会为新进程的栈分配空间，并且填充进程表中对应表项的信息。create 函数通过调用 getstk 函数来为栈分配空间（第 9 章将会讨论内存分配）。

create 函数的第一个参数指明了进程开始执行的函数的初始地址。create 函数在进程栈空间上形成一个被保存的上下文环境，就好像指定函数被调用了一样[⊖]。因此，我们将对进程上下文的初始配置称为一次伪调用。为了制造这样一个伪调用，create 函数在进程栈上为上下文分配空间。保存寄存器的初始值，在进程表的对应表项中保存栈指针。当 ctxsw 切换到该进程的时候，新的进程将开始执行指定的函数，遵循通用的调用规范来访问参数并为局部变量分配空间。简而言之，进程的初始函数就像在之前被调用了一样。

那么 create 函数要用什么作为伪调用的返回地址呢？返回地址的值决定了系统在进程从初始（顶层）函数返回之后所要执行的动作。我们的示例系统遵循一个著名的范式：

> 如果一个进程从它开始执行的初始（顶层）函数返回，那么该进程将退出。

更精确地说，我们必须能够区别从函数自身返回和从初始调用返回。这两者之所以有区别，是因为 C 语言允许函数的递归调用。如果一个进程开始执行函数 X，并且在 X 中递归调用 X，那么第一个返回仅仅将第一层递归的栈帧弹出栈并返回到初始调用栈帧中，而不会导致进程退出。如果进程再一次返回（或者到达了 X 函数的末尾）而不再产生其他调用，进程将退出。

为了在初始调用返回的时候退出进程，create 函数将 userret 函数的地址作为伪调用的返回地址。代码使用符号常量 INITRET 符号来表示 userret 函数名[⊜]。如果在初始调用中进程执行到了指定函数末尾或者显式调用了 return，那么控制权将转交给 userret 函数。userret 函数通过调用 kill 函数来终止当前进程。下面列出的 userret.c 文件包含了具体代码。

create 函数还会填充进程表中的表项。特别地，create 函数将新创建的进程状态置为 PR_SUSP，即挂起，而不是就绪。最后，create 函数返回新创建进程的进程 ID。在进程开始执行之前，必须将该进程恢复。

101

许多进程初始化的细节取决于 C 语言运行环境——如果不知道细节不能写出启动进程的代码。例如，create 函数将初始参数放置在寄存器中，并且将连续参数放置在栈中。将参数压栈的那部分代码也许比较难以理解，因为 create 函数直接从它自己的栈空间中复制参数到分配给新进程的栈空间上。为了实现这一功能，create 函数在它自己的栈上找到参数的地址，并且使用指针运算将参数地址移到链表中。这些细节取决于处理器架构和所使用的编译器。

```
/* create.c  - create, newpid */

#include <xinu.h>
#include <mips.h>
```

⊖ （即为指定的函数构造一个栈帧）。——译者注。

⊜ 使用符号常量能够通过修改配置文件而不是代码来完成修改。

```
static  pid32   newpid(void);

/*------------------------------------------------------------------------
 *  create, newpid  -  Create a process to start running a procedure
 *------------------------------------------------------------------------
 */
pid32   create(
          void          *funcaddr,      /* address of function to run   */
          uint32        ssize,          /* stack size in bytes          */
          pri16         priority,       /* process priority             */
          char          *name,          /* process name (for debugging) */
          uint32        nargs,          /* number of args that follow   */
          ...
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        uint32  *saddr;                 /* stack address                */
        uint32  *savargs;               /* pointer to arg save area     */
        pid32   pid;                    /* ID of newly created process  */
        uint32  *ap;                    /* points to list of var args   */
        int32   pad;                    /* padding needed for arg area  */
        uint32  i;
        void    INITRET(void);

        mask = disable();

        if (   (ssize < MINSTK)
            || (priority <= 0)
            || ((int32)(pid = newpid()) == (int32) SYSERR)
            || ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR) ) {
        restore(mask);
        return SYSERR;
}

prcount++;
prptr = &proctab[pid];

/* Initialize process table entry for new process */

prptr->prstate = PR_SUSP;       /* initial state is suspended   */
prptr->prprio = priority;
prptr->prstkptr = (char *)saddr;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;

/* Set up initial device descriptors for the shell              */

prptr->prdesc[0] = CONSOLE;     /* stdin  is CONSOLE device     */
prptr->prdesc[1] = CONSOLE;     /* stdout is CONSOLE device     */
prptr->prdesc[2] = CONSOLE;     /* stderr is CONSOLE device     */

/* Initialize stack as if the process was called                */
```

```
*saddr = STACKMAGIC;
*--saddr = pid;
*--saddr = (uint32)prptr->prstklen;
*--saddr = (uint32)prptr->prstkbase - prptr->prstklen
                                    + sizeof(int);

if (nargs == 0) {                   /* compute padding              */
        pad = 4;
} else if ((nargs%4) == 0) {        /* pad for A0 - A3              */
        pad = 0;
} else {
        pad = 4 - (nargs % 4);
}

for (i = 0; i < pad; i++) {         /* pad stack by inserting zeroes*/
        *--saddr = 0;
}
        for (i = nargs; i > 0; i--) {   /* reserve space for arguments  */
                *--saddr = 0;
        }
        savargs = saddr;                /* loc. of args on the stack    */

        /* Build the context record that ctxsw expects                  */

        for (i = (CONTEXT_WORDS); i > 0; i--) {
                *--saddr = 0;
        }
        prptr->prstkptr = (char *)saddr;

        /* Address of process entry point                               */

        saddr[(CONTEXT_WORDS) - 1] = (uint32) funcaddr;

        /* Return address value                                         */

        saddr[(CONTEXT_WORDS) - 2] = (uint32) INITRET;

        /* Copy arguments into activation record                        */

        ap = (uint32 *)(&nargs + 1);    /* machine dependent code to    */
        for (i = 0; i < nargs; i++) {   /* copy args onto process stack */
                *savargs++ = *ap++;
        }
        restore(mask);
        return pid;
}

/*------------------------------------------------------------------------
 *  newpid  -  Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local   pid32   newpid(void)
{
        uint32  i;                      /* iterate through all processes*/
        static  pid32 nextpid = 1;      /* position in table to try or  */
                                        /*  one beyond end of table     */

        /* Check all NPROC slots */
```

```
	for (i = 0; i < NPROC; i++) {
		nextpid %= NPROC;	/* wrap around to beginning */
			if (proctab[nextpid].prstate == PR_FREE) {
				return nextpid++;
			} else {
				nextpid++;
			}
		}
		return (pid32) SYSERR;
	}

	/* userret.c - userret */

	#include <xinu.h>

	/*------------------------------------------------------------------------
	 *  userret  -  Called when a process returns from the top-level function
	 *------------------------------------------------------------------------
	 */
	void	userret(void)
	{
		kill(getpid());			/* force process exit */
	}
```

create 函数在进程状态图中引入了一次初始状态转移：一个新创建的进程初始时处于挂起状态。图 6-3 说明了这样的状态转移。

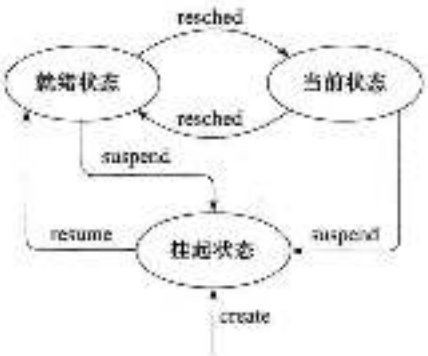


图 6-3 进程状态图，显示了新创建的进程初始化为挂起状态

102 ~ 105

6.13 其他进程管理函数

另外还有三个系统调用可以用来管理进程：getpid，getprio 和 chprio。正如名字一样，getpid 使得当前进程能够获得它的进程 ID，getprio 允许调用者获得任意进程的调度优先级。另外一个系统调用 chprio 允许一个进程修改任意进程的优先级。三个系统调用的实现都简单易懂。例如，让我们看一下 getprio 系统调用的代码。在完成参数检查之后，getprio 从进程表中提取指定进程的调度优先级，并且将优先级返回给调用者。

```
	/* getprio.c - getprio */

	#include <xinu.h>

	/*------------------------------------------------------------------------
	 *  getprio  -  Return the scheduling priority of a process
	 *------------------------------------------------------------------------
	 */
	syscall getprio(
		  pid32		pid		/* process ID		*/
		)
	{
		intmask mask;			/* saved interrupt mask		*/
		uint32	prio;			/* priority to return		*/

		mask = disable();
		if (isbadpid(pid)) {
			restore(mask);
```

```
		return SYSERR;
	}
	prio = proctab[pid].prprio;
	restore(mask);
	return prio;
}
```

由于全局变量 currpid 包含了当前正在运行的进程 ID，所以 getpid 的实现代码很简单：

```
/* getpid.c - getpid */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getpid  -  Return the ID of the currently executing process
 *------------------------------------------------------------------------
 */
pid32	getpid(void)
{
	return (currpid);
}
```

chprio 函数可以修改任何进程的调度优先级。该函数的代码在 chprio.c 中。

```
/* chprio.c - chprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  chprio  -  Change the scheduling priority of a process
 *------------------------------------------------------------------------
 */
pri16	chprio(
	  pid32		pid,		/* ID of process to change	*/
	  pri16		newprio		/* new priority			*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct	procent *prptr;		/* ptr to process' table entry	*/
	pri16	oldprio;		/* priority to return		*/

	mask = disable();
	if (isbadpid(pid)) {
		restore(mask);
		return (pri16) SYSERR;
	}
	prptr = &proctab[pid];
	oldprio = prptr->prprio;
	prptr->prprio = newprio;
	restore(mask);
	return oldprio;
}
```

chprio 函数在修改指定进程在进程表中的优先级之前，首先检查进程 ID 以保证该进程存在。在练习中，你将会看到这段实现代码有两个疏忽之处。

106 / 107

6.14 总结

为了扩展对并发执行的支持，本章在调度器和上下文切换的层面上添加了一层进程管理。新的层面包括挂起、恢复执行创建新进程和杀死。本章还分析了另外三个函数，它们分别是获取当前进程 ID 的 getpid 函数、获取任意进程调度优先级的 getprio 函数以及修改任意进程调度优先级的 chprio 函数。尽管代码很简洁，但到现在为止，这些代码已经构成了一个基本的进程管理器。使用适当的初始化和

其他辅助例程，这个基本的进程管理器可以让多个并发的计算任务在一个处理器上多道并发。

create 函数创建一个新进程，并将进程置于挂起状态。create 函数还为新进程分配栈空间并将一些必需的数值放在栈和进程表中，从而使得上下文切换函数 ctxsw 能够切换到该进程并开始执行。栈中的上下文信息被设置成一个伪调用的形式，就好像该进程是在 userret 函数之前被调用的一样。当进程从顶层函数返回时，控制权就交给 userret 函数，该函数将调用 kill 函数来终止进程。

练习

6.1 正如本章所述，当挂起的进程被其他进程通过 resume 唤醒之后，它的优先级可以设置为指定且唯一的值，从而可以知道哪些事件触发了唤醒操作。使用这种方法来创建一个进程并挂起该进程，然后判断另外两个进程中哪个进程首先唤醒了它。

6.2 为什么 create 创建的伪调用在进程退出之后的返回地址要设置为 userret 函数而不直接设置为 kill 函数？

6.3 全局变量 pcount 表示当前活动的用户进程数。请仔细考虑 kill 中的代码并思考 pcount 所表示的进程数是否包括空进程？

6.4 正如本章所述，kill 在最后一个进程终止之前调用 xdone。修改整个系统使得空进程而非用户进程的数量并且在所有进程完成之后调用 xdone。

6.5 在 6.4 题中，新的实现在 xdone 函数上附加了什么当前实现中没有的限制？

6.6 在某些硬件体系架构上，在应用程序进行系统调用时会使用一条特殊的指令。调查这样的架构，并描述一个系统调用是如何传递到对应的操作系统函数的。

6.7 create 条件将创建的进程设置为挂起状态而不是运行状态，为什么这么做？

6.8 resume 函数在其与 ready 之前，将被唤醒进程的优先级保存在一个局部变量中。请说明如果在调用 ready 之后再引用 proptr->pprio，resume 函数返回的优先级可能是被唤醒进程从未拥有过的（即使是在被唤醒之后）。

6.9 在函数 newpid 中，全局整型变量 nextproc 用来在进程表中寻找一个空闲的表项。从之前停下来的地方开始搜索空闲表项避免了每次都遍历之前用过的表项。请说明这种技术在嵌入式系统上是否有价值。

6.10 函数 chprio 有两个设计缺陷。第一个缺陷是代码并不确保输入的新优先级数值是一个正整数。请说明如果一个进程的优先级被设置为 -1 会怎么样。

6.11 chprio 第二个设计缺陷是它违反了一个基本的设计原则。请识别出这个缺陷，描述其可能产生的后果并修复它。

108

109

第7章

Operating System Design: The Xinu Approach, Linksys Version

协调并发进程

未来属于那些懂得如何等待的人。

——俄罗斯谚语

7.1 引言

前面两章节介绍了进程管理器的部分内容，包括进程调度、上下文切换和创建、终止进程。本章将继续探究进程管理中如何协调彼此同步相互独立的不同进程。在本章中，除了解释进程协同的出发点和实现之外，还会对多处理器（多核芯片）上的协调问题做相关的阐述。

第8章通过介绍操作系统底层的消息传递机制，对进程管理的内容进行拓展。后续的章节则介绍同步机制在输入输出中的应用。

7.2 进程同步的必要性

并发执行的进程需要相互协作来实现共享全局资源。特别地，操作系统的设计者必须保证在任何时候只有一个进程会试图改变一个指定变量的值。以进程表为例，当创建一个新进程时，需要在进程表
111 中为之分配空间并写入相应的值。如果两个进程都要创建新进程，系统就必须保证在某一时间只能有一个执行 create，否则就会产生错误。

第6章展示了一种可以用来保证一个进程不受其他进程影响的方法，即调用一个禁止中断函数。这样也可以避免使用那些调用了 resched 的函数的影响。像 suspend、resume、create 和 kill 这些系统调用都使用了这种方法。

当进程需要确保不受外界影响的时候，为什么不使用上述同样的解决方案呢？原因在于禁止中断对整个操作系统都有着负面影响：禁止中断使操作系统中只有一个进程在执行，其他所有活动都停止，而且还限制了进程的行为。尤为重要的是，在中断禁止期间输入/输出操作无法执行。后面我们还会发现禁止中断太久会导致很多问题（例如，在中断禁止期间，如果网络不断有数据包到达，那么网络接口会丢弃这些数据包）。因此，我们需要一种更为通用的协调机制，使得在不长时间禁止中断的前提下，允许进程之间协调数据项的使用，并且这种协调不会影响其他进程，不会限制其他进程的运行。例如，一个进程能够在对某一个大数据结构进行格式化并打印的同时禁止其他进程对它进行修改，同时不影响那些不会访问这个数据结构的进程的运行。这种机制必须是透明的：程序员应该能够理解进程协调的结果。最后，我们讨论的这种同步机制必须包括以下内容：

- 允许一部分进程为访问某一资源进行竞争
- 提供一种策略来保证竞争的公平性。

第一点确保进程协调是一种局部行为：只限定那些为了同一资源而竞争的进程，使其等待，而非禁止所有中断。操作系统的其余部分均可不受影响地正常运行。第二点确保如果有 K 个进程试图访问某一资源，则最终这 K 个进程都可以成功访问（也就是说，没有进程会被饿死）。

在第2章中，我们已经介绍了解决这一问题的基本机制——计数信号量，并给出了进程之间通过信号量来协调的例子。正如第2章所说，信号量非常优雅地解决了两个问题：

- 互斥。
- 生产者-消费者交互。

互斥 术语*互斥*是指确保一系列进程中同一时间只有一个进程运行的情况。不仅仅指共享数据，
112 还包括共享任意的资源，例如共享输入输出设备的情况。

生产者-消费者交互 术语生产者-消费者交互是指进程之间进行数据项交换的情况。其最简单的形式就是一个生成数据项的进程作为生产者，另一个接收数据项的进程作为消费者。更复杂的情况

下，作为生产者和消费者的进程都可以是多个。这时协调的关键在于任何生成的数据项都只能被一个消费者所读取（即没有数据项丢失或重复读取）。

无论哪种形式的并程协调问题都在操作系统中广泛存在。例如，考虑将一组远程发出的消息显示在控制台上，则控制台必须协调进程确保需要显示字符到达的速度低于硬件显示的速度。需要显示的字符存放在内存中的缓冲区里。如果缓冲区满了，则发出消息的进程必须阻塞等待直到缓冲区可用，而如果缓冲区为空，则显示器停止显示消息。这其中的关键在于当消费者不能接收数据时，生产者必须阻塞等待；而当生产者不再产生数据时，消费者也同样要阻塞等待。

7.3 计数信号量的概念

解决上述问题的计数信号量机制有着相当优雅的实现。从概念上说，信号量 s 由一个整数和一系列阻塞进程构成。当信号量创建之后，并程对信号量使用 wait 和 signal 两个函数。当一个进程调用 wait 函数时，信号量的值减 1；调用 signal 函数时，则信号量的值加 1。如果一个并程调用 wait 函数时，信号量的值变为负值，则这个进程将会阻塞，并被放到信号量的阻塞进程集合中。从运行的角度来讲，对 wait 的调用暂时不会返回。阻塞在信号量上的进程只有当有其他进程调用 signal 函数增加了信号量的值时才能继续执行。也就是说，当调用 signal 函数时，有进程被阻塞等待信息量，则阻塞进程之一变为就绪状态并执行。当然，程序员在使用信号量的时候必须注意：如果没有进程调用 signal，则阻塞的并程会一直等待下去。

7.4 避免忙等待

进程在等待信号量时应该做什么呢？当进程对信号量的减 1 后，似乎就在反复检查信号量的值直到其为正值。对于单 CPU 的系统来说，这种忙等待是不可接受的，因为如果这样就占用了其他进程的 CPU 资源，而如果其他进程都无法获得 CPU 资源，那么就没有进程可以调用 signal 函数来终止前一个进程的等待状态。因此，操作系统必须避免忙等待。在实现信号量时，我们应该遵循一个非常重要的原则：

> 当一个进程等待信号量时，该进程不应该执行任何指令。

7.5 信号量策略和进程选择

为了在实现信号量时避免忙等待，操作系统为每个信号量关联了一个进程等表。只有当前进程可以选择等待信号量。当一个进程等待信号量 s 时，系统将信号量 s 对应的值减 1，如果变为负值，则该进程阻塞。操作系统将该进程放入信号量关联的并程链表中，将其状态改为非当前进程，然后调用 reshed 函数让其他进程运行。

接着，当有进程在信号量 s 上调用 signal 函数时，s 对应的值相应增加。同时，signal 函数检查 s 关联的进程链表，如果链表非空（即至少有一个进程等待该信号量），则 signal 函数就从等表中取出一个进程，并放回到就绪进程链表中。

问题随之而来：如果多个进程在等待，signal 函数会选择哪一个进程？常见的策略有以下几种：

- 最高调度优先级策略。
- 最长等待时间策略。
- 随机策略。
- 先到先服务策略。

尽管看起来很合理，选择最高优先级的进程违反了基本原则之一：竞争的公平性。考虑一些低优先级的进程和一些高优先级的进程共享某一资源，每个并程重复地使用该共享资源，如果信号量系统一直选择高优先级进程，则高优先级进程会一直获得 CPU 资源，低优先级进程会一直阻塞下去。

选择最长等待时间的进程可能会导致优先级反转，即高优先级进程会阻塞而低优先级进程能够执行。后面的练习中还要到了这种策略会导致的另一个同步问题。避免这些问题的方法之一是在等待的

进程中随机选择一个运行。随机策略的主要缺点在于其消耗的资源，例如随机数的生成需要执行乘法运算，而这需要占用不少计算资源。

综上所述，很多实现中采用了先到先服务的策略，即操作系统使用一个队列来存储等待给定信号量的进程。当一个进程调用 wait 函数时，将它加到该队列的末尾并阻塞；而当有进程调用 signal 函数
[114] 时，队列头部的进程将会结束阻塞并开始运行。最终策略为：

信号量进程选择策略：如果一个或多个进程等待信号量 s，当有进程调用 signal 函数时，到达时间最久的进程将从阻塞变为就绪状态。

7.6 等待状态

当一个进程等待信号量时，我们应该为其设置怎样的状态呢？由于进程既没有使用 CPU 又不具备运行的条件，所以我们不能将其设置为当前状态或者就绪状态。而第6章中介绍的挂起状态在这里也不能使用，因为使进程进入或者离开挂起状态的 suspend 和 resume 函数都与信号量无关。更为重要的是，等待信号量的进程都在一个链表中，而挂起的进程显然不在其中，这个差别在调用 kill 函数终止某一进程时至关重要。由于已有的状态无法精确地概括等待信号量的进程的状态，所以我们必须使用一个新的状态——等待，在代码中用 PR_WAIT 的符号来表
[115] 示。图7-1是扩展的进程状态转换图。

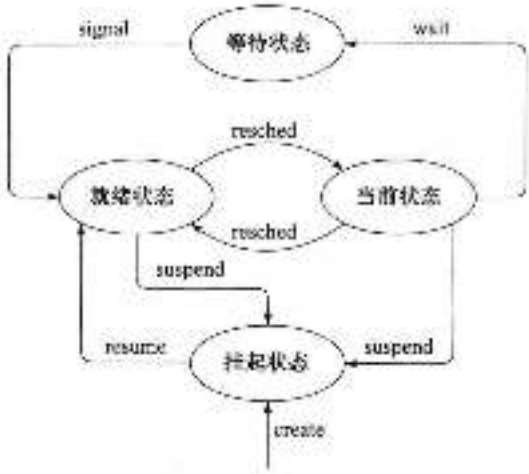


图7-1 包括等待状态的状态转换图

7.7 信号量数据结构

在示例操作系统中，信号量信息存储在一个全局信号量表 semtab 中，其中的每一条表项都对应一个信号量，包含一个整数值和一个用来存放等待进程的队列 ID。表项由结构体 sentry 来定义，具体细节在 semaphore.h 文件中。

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM            45      /* number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE  0               /* semaphore table entry is available   */
#define S_USED  1               /* semaphore table entry is in use      */

/* Semaphore table entry */
struct  sentry  {
        byte    sstate;         /* whether entry is S_FREE or S_USED    */
        int32   scount;         /* count for the semaphore              */
        qid16   squeue;         /* queue of processes that are waiting  */
                                /*      on the semaphore                */
};

extern  struct  sentry semtab[];

#define isbadsem(s)     (int32)(s) < 0 || (s) >= NSEM)
```

在结构体 sentry 中，scount 字段为信号量的当前整数值，等待该信号量的进程链表存储在队列中，squeue 字段给出了指定信号量的队列头部的地址，状态字段 sstate 则表示当前记录是否已使用（已分

配空间）或为空（未分配空间）。

在整个操作系统中，信号量均通过一个整数 ID 来识别。信号量的 ID 同样是为了提高查询的效率——信号量表由一个数组来实现，每个信号量的 ID 就是其在数组中的索引。总的来说：

信号量由其在全局信号量表 semtab 中的索引来进行识别。 116

7.8 系统调用 wait

信号量的两个主要操作为 wait 和 signal。wait 函数减少信号量的值。当信号量的值为非负时，wait 函数直接返回。本质上，对非负值的信号量调用 wait 函数的进程自愿交出其对 CPU 的控制权。也就是说，wait 函数将调用自己的进程放入信号量的等待队列中，并将其状态改为 PR_WAIT，然后调用 resched 函数切换到另一个就绪的进程。前面讲过，维护等待队列的策略为先进先出，即新到达的进程放入队尾。wait.c 文件的代码如下。

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
	  sid32		sem		/* semaphore on which to wait	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct  procent *prptr;		/* ptr to process' table entry	*/
	struct  sentry *semptr;		/* ptr to semaphore table entry */

	mask = disable();
	if (isbadsem(sem)) {
		restore(mask);
		return SYSERR;
	}

	semptr = &semtab[sem];
	if (semptr->sstate == S_FREE) {
		restore(mask);
		return SYSERR;
	}

	if (--(semptr->scount) < 0) {		/* if caller must block */
		prptr = &proctab[currpid];
		prptr->prstate = PR_WAIT;	/* set state to waiting */
		prptr->prsem = sem;		/* record semaphore ID	*/
		enqueue(currpid,semptr->squeue);/* enqueue on semaphore */
		resched();			/*   and reschedule	*/
	}
	restore(mask);
	return OK;
}
```

一个进程一旦进入信号量的等待队列便保持在等待状态（不具备执行的条件）直到该进程到达队列头部且有另一个进程调用 signal 为止。当 signal 调用将一个进程放回到就绪链表中时，该进程就具备了使用 CPU 的条件，并最终继续执行。从等待进程的角度来看，对 ctxsw 的调用会变为对 resched 的调用，对 resched 的调用会变为对 wait 的调用，而对 wait 的调用则会最终回到其调用前的状态。

7.9 系统调用 signal

signal 函数接受一个信号量的 ID 作为参数，增加该信号量的值，如果有进程等待该信号量，则将等待队列中的第一个进程置为就绪状态。为什么 signal 会在信号量值为负的时候将一个进程置为就绪状态？为什么 wait 并不总是将调用自己的进程放入信号量的等待队列中？这些看起来难以理解，其实原因都很容易理解，并且很容易实现。不管信号量的值为多少，wait 和 signal 都遵循下面的不变式：

信号量不变式：信号量值非负。意味着其等待队列为空；信号量值为 $-N$，则等待队列中有 N 个等待的进程。

本质上，信号量值为 N 意味着进程在该信号量上调用 wait 函数 N 次都不会发生阻塞，第 $N+1$ 次调用时才会阻塞。由于 wait 和 signal 都改变信号量的值，所以这两个函数都必须调整等待队列的长度来符合上面的不变式。当 wait 减少信号量的值使其变为负值时，当前进程就会加入到等待队列中；而 signal 则增加信号量的值，并在等待队列非空时将队首的进程取出。

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
syscall signal(
          sid32         sem             /* id of semaphore to signal    */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  sentry *semptr;         /* ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        if ((semptr->scount++) < 0) {   /* release a waiting process */
                ready(dequeue(semptr->squeue), RESCHED_YES);
        }
        restore(mask);
        return OK;
}
```

7.10 静态和动态信号量分配

操作系统设计者需要从下面两种方法中选择一种来进行信号量的分配：

- 静态分配：程序员在编译时定义一个固定的信号量集合，这个集合在系统运行的时候不变。
- 动态分配：操作系统包含按需创建和销毁信号量的函数。

静态分配的优点在于节约存储空间，减少 CPU 的负担——系统只需保留信号量所需的内存，无需创建和销毁信号量的函数。因此小型的嵌入式系统采用静态分配的方式。

动态分配的主要优点在于其能够在运行时适应新的用途。例如，动态分配的方案允许用户启动一个应用程序来创建一个信号量，也可以关闭这个应用程序然后启动另一个。所以大型的嵌入式系统和

大部分大型操作系统都支持包括信号量在内的资源动态分配。7.11 节中我们会看到动态分配机制也不需要很多额外的代码来实现。

117
:
119

7.11 动态信号量的实现示例

Xinu 实现了部分形式的动态分配：进程可以动态创建信号量，某一进程可以创建多个信号量，但信号量的总数不超过预定义的上限。另外，为了最小化分配的开销，操作系统在启动时就为每一个信号量预分配了一个队列。所以当进程创建信号量时只需要做很少的事情。

系统调用 semcreate 和 semdelete 分别处理信号量的动态分配和回收。semcreate 接收信号量的初始值为参数，创建一个信号量并把初始值赋给它，然后返回该信号量的 ID。为了符合信号量不变式，这里的初始值必须为非负值。同此，semcreate 首先检查参数是否合法。如果参数合法，semcreate 调用 newsem 遍历信号量表 semtab 中所有 NSEM 条记录，找到一条未使用的记录并设置初始值。如果没有可用的记录，则 newsem 返回 SYSERR；否则，newsem 将找到的记录的状态置为 S_USED 并返回其在表中的索引。

一旦信号量表中的表项已经分配好，semcreate 就只需要设置初始值并返回信号量的索引给调用者。用来存储等待进程队列的头、尾部已经在操作系统启动时分配好了。semcreate.c 文件中包含了 newsem 和 semcreate 的代码。需要注意的是，代码中使用了静态的索引变量 nextsem 来优化对信号量表的查找（使查找可以从上一次查找结束的地方开始）。

```
/* semcreate.c - semcreate, newsem */

#include <xinu.h>

local	sid32	newsem(void);

/*------------------------------------------------------------------------
 *  semcreate  -  create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32	semcreate(
	  int32		count		/* initial semaphore count	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	sid32	sem;			/* semaphore ID to return	*/

	mask = disable();

	if (count < 0 || (sem=newsem())==SYSERR) {
		restore(mask);
		return SYSERR;
	}
	semtab[sem].scount = count;	/* initialize table entry	*/

	restore(mask);
	return sem;
}

/*------------------------------------------------------------------------
 *  newsem  -  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local	sid32	newsem(void)
{
	static	sid32	nextsem = 0;	/* next semaphore index to try	*/
```

```
        sid32   sem;                    /* semaphore ID to return       */
        int32   i;                      /* iterate through # entries    */

        for (i=0 ; i<NSEM ; i++) {
                sem = nextsem++;
                if (nextsem >= NSEM)
                        nextsem = 0;
                if (semtab[sem].sstate == S_FREE) {
                        semtab[sem].sstate = S_USED;
                        return sem;
                }
        }
        return SYSERR;
}
```

7.12 信号量删除

semdelete 与 semcreate 的行为相反，它以信号量的索引为参数，释放其在信号量表中的表项资源供后续使用。回收信号量分为三个步骤：1）semdelete 验证参数是否为合法的信号量 ID，参数对应的记录是否为已使用状态；2）semdelete 将记录的状态置为 S_FREE，表示该记录可以重用；3）semdelete 遍历等待进程的队列，将其中的每个进程都置为就绪状态。semdelete.c 文件的代码如下：

120
-
121

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  --  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
        sid32           sem             /* ID of semaphore to delete    */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  sentry *semptr;         /* ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        semptr->sstate = S_FREE;

        while (semptr->scount++ < 0) {  /* free all waiting processes   */
                ready(getfirst(semptr->squeue), RESCHED_NO);
        }
        resched();
        restore(mask);
        return OK;
}
```

如果信号量回收后等待队列中还有进程，操作系统就必须对每个进程进行处理。在示例实现中，semdelete 将等待队列中的所有进程都置为就绪状态，就像有进程调用 signal 一样允许这些进程继续运

行。这种实现只是所有策略中的一种。例如，有些操作系统认定如果仍有进程等待指定的信号量，那么回收这一信号量就会产生错误。我们会在后面的练习中讨论其他策略。 [122]

需要注意的是，上述将等待进程置为就绪的代码并没有在进程放入就绪链表后直接重新进行调度。相反，每一个调用 ready 的操作都注明了 RESCHED_NO。当所有等待进程都移入就绪链表之后，代码才显式地调用 resched 来重新建立调度不变式。

7.13 信号量重置

在时候重置一个信号量的数量很方便，不会带来删除一个旧信号量和申请一个新信号量的开销。系统调用 semreset 重置了一个信号量的数量，如下面的文件 semreset.c 所示。

```
/* semreset.c - semreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  semreset  -  reset a semaphore's count and release waiting processes
 *------------------------------------------------------------------------
 */
syscall semreset(
        sid32       sem,            /* ID of semaphore to reset     */
        int32       count           /* new count (must be >= 0)     */
        )
{
        intmask mask;               /* saved interrupt mask         */
        struct  sentry *semptr;     /* ptr to semaphore table entry */
        qid16   semqueue;           /* semaphore's process queue ID */
        pid32   pid;                /* ID of a waiting process      */

        mask = disable();

        if (count < 0 || isbadsem(sem) || semtab[sem].sstate==S_FREE) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        semqueue = semptr->squeue;      /* Free any waiting processes */
        while ((pid=getfirst(semqueue)) != EMPTY)
                ready(pid,RESCHED_NO);
        semptr->scount = count;         /* reset count as specified */
        resched();
        restore(mask);
        return(OK);
}
```

semreset 必须保持信号量不变式。通用目标的解决方案允许调用者定义任意的信号量数量，但是我们的实现并没有这么做，而是采用了一个要求新的数量为非负的简化方案。这样做的结果是，一只信号量的数量改变，等待进程的队列将为空。和 semdelete 一样，semreset 必须确定已经没有进程等待这个信号量了。因此，在检查了其变量并已验证信号量存在后，semreset 迭代等待进程的链表，从这个信号量队列上移除每个进程，并让这个进程准备执行。一旦所有的等待进程都被移除，semreset 就分配新的值给信号量，重新进行调度（在等待进程有更高优先级的情况下），并返回给它的调用者。

7.14 多核处理器之间的协调

以上描述的信号量系统能够在一个单核计算机上良好地运行。但是，许多现代的处理器芯片是多核的。一个核常常用来运行操作系统的功能，而其他核用来执行用户应用程序。在这样的系统里，仅

仅使用操作系统提供的信号量是不够的。欲知为何是这样，不妨考虑在第二个核上运行这样一个应用程序会发生些什么：这个应用程序需要独占地访问某一块特定内存。此应用进程调用 wait，并将请求发送到核1上的操作系统。核2必须中断核1来完成请求。此外，当运行操作系统函数时，核1已经禁止中断，这使信号量无法使用。

有些多处理器系统提供称为旋锁（spin lock）的硬件原语，它允许多个处理器来竞争互斥访问。硬件定义了一系列数量为 K 的旋锁（K 应该小于1024）。理论上，每个旋锁都是一个单独的位，并且都初始化为0。每个处理器包括一个称为 test-and-set 的特定指令，它表现为两个原子指令：设置旋锁为1并返回在这个操作前的旋锁值。硬件的原子性保证表明，如果两个或者更多的处理器试图同时设置一个给定的旋锁，那么其中一个会收到先前的值0，其他的会收到1。一旦完成，获得锁的处理器设置其值为0，来允许其他处理器获得该锁。

123 ↑ 124

下面讲解旋锁的工作原理。假设两个处理器需要独占地访问一块共享数据项，并且在使用旋锁5。当一个处理器想要互斥地获得访问时，这个处理器执行下面的循环[⊖]：

```
while (test_and_set(5)) {
    ;
}
```

这个循环重复使用 test-and-set 指令来设置旋锁5。如果这个锁在指令执行前设置，这个指令将返回1，并且这个循环会继续下去。如果在指令执行前这个锁没有被设置，硬件返回0，循环终止。如果多个处理器试图同时设置旋锁5，这个硬件保证只允许一个能访问。因此，test-and-set 类似于 wait 指令。

一旦一个处理器使用完共享数据，这个进程就执行一条清除旋锁的指令：

```
clear(5);
```

在一个没有旋锁硬件的多处理器机器上，厂商会让机器包含生成旋锁的指令。例如，Intel 多核处理器依赖于内存中的原子 compare-and-swap 指令。如果多个核试图同时执行这个指令，那么其中一个会成功，其他的会失败。程序员能够使用这些指令来建立等价的旋锁。

因为直到访问被允许，处理器会阻塞在一个循环中，所以看起来旋锁似乎显得很费。但是，当两个处理器同时竞争一个旋锁的可能性很低时，这个机制就比系统调用更有效。因此，程序员应当注意何时使用旋锁，何时使用系统调用。

7.15 观点

计数信号量的概念很重要，主要有两个原因。第一，它提供了一个强大的机制，能够用来控制互斥和生产者－消费者同步，而它们是进程协调的两个主要范式。第二，它的实现相当紧凑并且足够高效。回顾函数 wait 和 signal，我们可以发现它们正是得益于计数信号量只占用了很小空间。如果移除用来测试参数的代码和返回结果，那么就只剩下了寥寥数行代码。和我们检测其他抽象概念的实现时一样，以下这点变得越来越显著：尽管它们很重要，但实现计数信号量只需要极少的代码。

7.16 总结

禁止中断会阻止除了当前进程以外的所有行为。但操作系统并没有这么做，而是提供了同步原语

125

来允许进程的子集在不影响其他进程的情况下进行协作。其中计数信号量允许进程在不使用忙等待的情况下进行协作。每个信号量包括一个整数值加上一个进程队列。这个信号量依赖于一个定义为非负数 N 的不变量，它表示这个队列包括 N 个进程。

signal 和 wait 这两个基本原语允许调用者增加或者减少信号量值。如果 wait 调用使这个信号量值为负，调用进程就会被设置为等待状态，并且 CPU 让另一个进程执行。本质上，一个等待某个信号量的进程会自行让自己加入到等待这个信号量的队列中去，并且调用 reschd 来允许其他进程执行。

静态或者动态分配能够用于信号量。在示例代码里，semcreate 和 semdelete 函数用来允许动态分

⊖ 由于使用到了硬件指令，test-and-set 的代码经常使用汇编来编写。这里为清楚起见使用了伪代码。

配。如果进程等待时一个信号量被释放，那么这些进程应当被处理。

多处理器使用称为旋锁的互斥机制。尽管需要一个处理器重复测试是否能够访问，但是相比于一个处理器中断另一个处理器来进行系统调用的方法，自旋锁还是更有效的。

练习

7.1　本章表明如果有进程仍然在排队等待信号量，有些操作系统就认为信号量的删除会出错。重写 semdelete 使每删除一个信号量时返回 SYSERR。

7.2　考虑用延期作为本章中所示的信号量删除机制的一种替代。也就是，重写 semdelete，使得在所有进程都执行 signal 前，用一个延期状态替代一个删除的信号量。修改 signal，使得当最后等待的进程从队列中移除的时候，才释放信号量表表项。

7.3　在前面的练习中，延迟删除是否有一些出乎意料的副作用呢？尝试解释之。

7.4　作为活动信号量延迟删除的进一步的替代，修改 wait，使得当调用进程等待时，如果信号量被删除，则返回一个 DELETED 值。（为 DELETED 选择一个不同于 SYSERR 和 OK 的值。）一个进程如何决定它等待的信号量是否被删除了呢？注意：高优先级的进程能够在任意时刻执行。因此，低优先级的进程准备就绪后，高优先级的进程能够获得 CPU，并产生一个新的信号量。在低优先级的进程完成 wait 操作前，高优先级进程重新使用信号量表表项。提示：考虑增加一个顺序字段到信号量表表项中。

7.5　不是定位一个中心信号量表，而是安排某个进程按需定位信号量表项的空间，并使用一个表项的地址作为信号量 ID。比较示例代码中中心表的方法。它们每个的优点和缺点各是什么？

7.6　wait、signal、semcreate 和 semdelete 使用信号量表在它们之间协作。使用一个信号量来保护信号量表的使用是否可行？试解释之。 126

7.7　为什么 semdelete 无需重新调度就可以调用 ready？

7.8　考虑一个可能的优化：在一个进程置于就绪链表前，semdelete 检测每个等待进程的优先级。如果这些进程中没有比当前进程高优先级更高的进程，则不调用 resched。这个优化的代价是什么？可能节省了什么？

7.9　构建一个新的系统调用，signaln（sem，n），它调用信号量 sem 的 signal 操作 n 次。你能找到一个比 n 次调用 signal 更加有效的实现吗？试解释之。

7.10　示例代码对信号量使用 FIFO 策略。也就是说，当信号量执行 signal 操作时，已经等待最长时间的队列变为就绪。考虑一个修改：等待信号量的进程保存在一个按进程优先级排列的优先级队列中（例如，当信号量执行 singal 操作时，最高优先级的等待进程变为就绪。）优先策略的主要缺点是什么？

7.11　语言总体若特别是写并发程序时，协作和同步常常直接包含在语言结构中。例如，有可能声明一组过程，使编译器自动插入代码来禁止多于一个进程来执行一个给定组。找到一个为并发程序设计的语言的例子，并且与 Xinu 代码中有信号量的进程进行比较。当程序员被明确要求操作信号量时，程序员会犯什么类型的错误？

7.12　当把一个等待进程变为就绪状态时，wait 设置进程表表项中的 prsem 字段为等待进程的信号量 ID。这个值会被使用吗？

7.13　如果程序员犯了个错误，更有可能的是这个错误会产生 0 或者 1，而不是任意整数。为了帮助避免错误，修改 newsem 从表的最大地址处分配信号量，0 和 1 不使用，一直到所有其他的表项已经用完。提出更好的方式来识别增加检测错误能力的信号量。

7.14　当删除一个非负整数值的信号量时，函数 semdelete 会表现异常。识别这个异常行为，并尝试重写代码以修正这个异常。

7.15　画一个从第 4 章～第 7 章所有操作系统函数的调用图，标注出一个给定函数会调用哪些函数。一个多层结构能否从图中推出？试解释之。 127

第8章

Operating System Design: The Xinu Approach, Linksys Version

消息传递

历史终究会传递的信息明确无误：过去在我们的后面。

——佚名

8.1 引言

前面的章节解释了进程管理器的基本构件，包括：调度器、上下文切换以及协调并发进程的计数信号量。这些章说明了进程是如何创建和终结的，并解释了操作系统如何在进程表中存放每个进程信息。

本章总结基本进程管理组件，介绍消息传递的概念，描述可能的方法，并且列举一个底层消息传递系统的例子。第11章解释高层消息传递组件是如何通过基本的进程管理机制来建立的。

8.2 两种类型的消息传递服务

我们使用消息传递（message passing）这个词来表示一种进程间交互的形式，其中一个线程可以把（通常是少量的）数据传送给另一个进程。在某些系统中，进程从一个称为收取点（pickup point）的地方存取和检索信息，它有时也称为邮箱（mailbox）。在其他系统中，消息可以直接定位到一个进程。

129 消息传递既方便又强大，有些操作系统使用它作为进程间所有交互和协作的基础。例如，通过计算机网络传送数据的操作就能够使用消息传递原语来实现。

有些消息传递组件提供了进程协作的能力，因为这个机制会阻塞接收者直到消息到达。因此，消息传递可以代替进程挂起和恢复。那么消息传递也能够代替同步原语（比如，信号量）吗？答案依赖于消息传递的实现方式。消息传递有两种方式：

- 同步　如果接收者试图在消息到达前接收，那么它就会阻塞；如果发送者试图在接收者就绪前发送消息，它也会阻塞。发送进程和接收进程应当同步或者一方为等待另一方而阻塞。
- 异步　消息可以在任意时刻到达，消息到达后通知接收者。接收者不需要知道有多少消息到达或者有多少发送者将发送消息。

尽管可能缺少灵活性和便利性，但同步消息传递组件能够在某些方面发挥信号量的作用。例如，考虑生产者－消费者问题。每次产生新的数据，生产者进程能够发送消息给消费者进程。同样，不用等待信号量，消费者就能够等待消息。使用消息传递来实现互斥更复杂，但常常是可行的。

因为同步消息传递系统适应传统的计算模型，所以它的主要优点就显露出来。为了在同步系统中接收消息，进程需要调用一次系统函数，这个调用会阻塞直到消息到达。相反，异步消息传递系统或者要求进程轮询（即过一段时间就检查消息），或者要求一个允许操作系统暂停进程的机制，允许进程来处理消息，然后恢复正常执行。尽管导致了额外的负载和复杂度，但是如果进程不知道有多少消息要到达，何时发送消息，或者哪些进程发送消息，那么使用异步消息传递比较方便。

8.3 消息使用资源的限制

Xinu支持两种消息传递：完全同步机制和部分异步机制。这两种方式也表明了直接和间接消息发送的区别：一个提供直接的进程间消息传递，另一个要求消息在会合点交互。本章的讨论从最简单一个

130 进程到另一个进程的直接通信的组件开始。第11章讨论第二种消息传递机制。将消息传递分成两个独立的部分有这样的优势：可以使得进程间底层的消息传递更高效。同时又允许程序员按需选择复杂的会合方法。

Xinu进程到进程的消息传递系统经过了仔细设计，以确保进程发送消息永不会阻塞，并且等待消息时不会消耗掉所有的处存。为了确保这些目标，消息传递组件遵循下条原则：

- 限制消息大小。系统限制每个消息为一个较小的固定尺寸。在我们的示例代码中，每个消息包括一个字（即一个整数或者一个指针）。
- 没有消息队列。系统允许一个给定进程为每个进程在任意时刻存储唯一一个未接收的消息。这里没有消息队列。
- 第一消息语义（first message semantic）。如果多个消息发送到一个给定的进程，而这个进程又不能接收它们，只将第一个消息存储和传递。随后的发送者不会阻塞。

当需要判断多个事件中的哪一个事件最先完成时，第一消息语义的概念就变得非常有用。等待事件的进程可以安排每个事件发送一个唯一的消息，然后进程等待消息的到来，由操作系统来保证进程会收到第一个发送的消息。

8.4 消息传递函数和状态转换

对消息进行操作的系统调用有3个：send，receive 和 recvclr。send 接收一个消息和一个进程 ID 作为参数，并且传递消息到指定进程。receive 不需要任何参数，它让当前进程等待直到消息到达，然后将消息返回给调用者。recvclr 提供一个非阻塞的 receive 版本。如果 recvclr 调用时当前进程已经接收了一个消息，那么这个调用就会像 receive 一样返回这个消息。但是如果没有消息在等待中，recvclr 立刻返回 OK 值，而不会延迟以等待一个消息的到达。正如名字暗示的，recvclr 在进行一轮消息传递前，能够用来删除一个旧消息。

问题来了：当进程等待消息时应处于哪种状态呢？因为等待消息不同于准备执行、等待信号量、等待 CPU、假死等已有的状态。因此，在我们的设计中应当使用另一个状态。这个新的状态接收（receiving）在示例软件中表示为符号常量 PR_RECV。增加这个状态产生了图 8-1 所示的状态转换图。

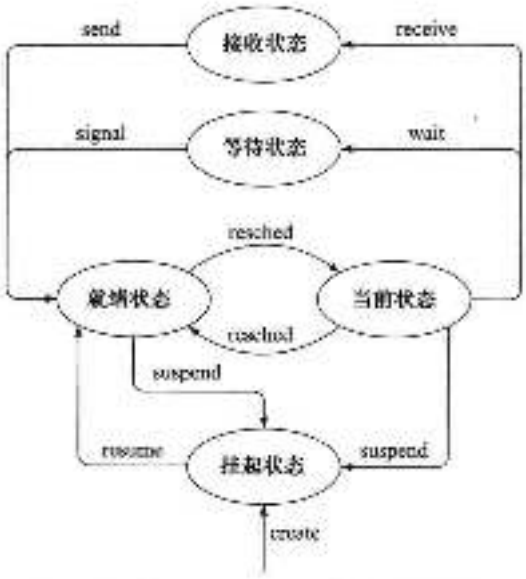


图 8-1 包括接收（receiving）状态的状态转换图 131

8.5 send 的实现

消息传递系统要求发送者和接收者之间达成共识，因为发送者应当在某处存储消息，接收者也能够从这个地方提取消息。消息不能在发送者内存中存储，因为在消息接收之前发送进程可能退出。大多数的操作系统不允许发送者把消息放到接收者的内存空间，因为允许一个进程写入另一个进程的内存空间会产生安全威胁。在我们的示例系统中，消息大小的限制消除了这个问题。我们的实现在接收者的进程表项的 pmsg 字段中预留了空间。

为了存储消息，send 函数首先检查指定的接收进程是否存在。然后检查以确保接收者没有未完成的消息。为了做到这点，send 在接收者的进程表项中检查 phasmsg 字段。如果这个接收者没有未完成的消息，则 send 在 pmsg 字段存储新的消息，并设置 phasmsg 字段为 TRUE 以表明有消息在等待。最后一步，如果接收者正在等待一个消息的到达（比如接收进程具有状态 PR_RECV 或者状态 PR_RECTIM），则 send 通过 RESCHED_YES 参数调用 ready 让进程就绪，并且重新建立调度不变式。在 PR_RECTIM（这个状态稍后介绍）情况下，send 应当首先调用 unsleep 从睡眠进程队列中移除这个进程。文件 send.c 包含以下代码。 132

```
/* send.c - send */

#include <xinu.h>
```

```
/*------------------------------------------------------------------------
 *  send  -  pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
        pid32       pid,            /* ID of recipient process      */
        umsg32      msg             /* contents of message          */
        )
{
        intmask mask;               /* saved interrupt mask         */
        struct  procent *prptr;     /* ptr to process' table entry  */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];
        if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
                restore(mask);
                return SYSERR;
        }
        prptr->prmsg = msg;             /* deliver message              */
        prptr->prhasmsg = TRUE;         /* indicate message is waiting  */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) {
                ready(pid, RESCHED_YES);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid, RESCHED_YES);
        }
        restore(mask);          /* restore interrupts */
        return OK;
}
```

133

8.6 receive 的实现

进程调用 receive（或者 recvclr）来获取一个传入的消息。receive 检测当前进程的进程表项，并使用 prhasmsg 字段来决定是否有消息在等待。如果没有消息到达，receive 就把进程状态改为 PR_RECV，并且调用 resched 来允许其他进程运行。当另一个进程发送消息给接收进程时，resched 的调用直接返回。一旦执行通过 if 语句，receive 将提取消息，设置 prhasmsg 为 FALSE，并且返回消息给它的调用者。文件 receive.c 包含以下代码：

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
        intmask mask;               /* saved interrupt mask         */
        struct  procent *prptr;     /* ptr to process' table entry  */
        umsg32  msg;                /* message to return            */
```

```
        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {
                prptr->prstate = PR_RECV;
                resched();              /* block until message arrives  */
        }
        msg = prptr->prmsg;             /* retrieve message             */
        prptr->prhasmsg = FALSE;        /* reset message flag           */
        restore(mask);
        return msg;
}
```

仔细看代码，注意 receive 从进程表项把消息复制到局部变量 msg 中，接着返回 msg 的值。有趣的是，receive 并没有修改进程表中的 prmsg 字段。因此，一种更有效的实现似乎应当避免把消息复制到局部变量，而只是从进程表返回信息：

```
return proctab[currpid].prmsg;
```

[134]

不幸的是，这种实现是不正确的。后面的一道练习会让读者考虑为什么这种实现可能导致不正确的结果。

8.7 非阻塞消息接收的实现

recvclr 操作除了总是立即返回外，非常类似于 receive。如果一个消息正在等待，则 recvclr 返回这个消息；否则，recvclr 返回 OK。

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        umsg32  msg;                    /* message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
                msg = prptr->prmsg;     /* retrieve message             */
                prptr->prhasmsg = FALSE;/* reset message flag           */
        } else {
                msg = OK;
        }
        restore(mask);
        return msg;
}
```

8.8 观点

类似于前面章节的计数信号量的部分，基本的信息传递组件的代码是极其紧凑而高效的。看看这些函数，寥寥数行就实现了每个操作。此外，在进程表中存储消息缓冲区很重要，因为这样能将消息传递从内存管理中独立出来，并且允许在底层的架构中使用信息传递。

[135]

8.9 总结

消息传递组件提供了允许一个进程发送消息给另一个进程的进程间通信机制。一个完全同步的消

息传递系统阻塞发送者也阻塞接收者，并依赖于有多少消息已经发送和接收。我们的示例系统包含了两个信息传递组件：一个低层的允许进程间直接通信的机制，一个高层的使用会合点的机制。

Xinu 底层的消息传递机制限制信息的大小为一个字，限制每个进程最多有一个未完成的消息，并使用第一消息语义。消息的存储和进程表相关——发送到进程 P 的消息存储在 P 的进程表项中。第一消息语义的使用允许进程决定先触发哪些事件。

底层的消息组件包括三个函数：send，receive 和 recvclr。这三个函数中，只有 receive 是阻塞的——它阻塞调用进程直到有消息到达。在开始使用消息传递来进行交互之前，进程使用 recvclr 移除一个旧的消息。

练习

8.1 写一个程序，输出一个提示，然后每 8 秒循环输出这个提示直到某个人输入一个字符。（提示：sleep(8) 延迟这个调用进程 8 秒。）

8.2 假设 send 和 receive 不存在，用 suspend 和 resume 写代码来实现消息传递。

8.3 示例的实现使用了第一消息语义。现在哪个组件处理了最后消息语义？

8.4 实现为每个进程记录 A 个消息的 send 和 receive 的版本（连续调用 send 阻塞）。

8.5 调查系统最内层使用了消息传递而不是用上下文切换的操作系统。优点是什么？主要可靠性呢？

8.6 考虑本章提到的直接从进程表项中返回消息的 receive 的修改：

```
return proctab[currpid].prmsg;
```

解释为什么这种实现是错误的。

8.7 实现定义了 32 个可能消息的 send 和 receive 的版本。不要使用整数来代表消息，用字的一位来代表每个消息。并允许一个进程来模拟所有的 32 个消息。

8.8 观察到，因为 receive 使用了 SYSERR 来表示一个错误，所以发送的消息和 SYSERR 有相同值时就会产生混淆。而且，如果没有消息在等待，recvclr 就返回 OK。修改 recvclr 使得如果没有消息在等待返回 SYSERR，并修改 send 使得它拒绝发送 SYSERR（即检查它的参数，如果值是 SYSERR 返回一个错误）。

136

基本内存管理

内存是经历的影子。

——佚名

9.1 引言

前面的章节解释了并发计算和操作系统管理并发进程的机制，讨论了进程的创建和终结、调度、上下文切换、协调和进程间通信。

本章开始讨论第二个关键主题：操作系统存储管理的机制。本章着重于基本的操作：栈和堆的动态分配。本章介绍一些用来分配和释放内存的方法，解释嵌入式系统如何为进程处理内存。第 10 章将继续通过描述地址空间、高层的内存管理机制和虚拟内存来讨论内存管理。

9.2 内存的类型

由于主存储器（主存）对于程序的执行和数据的存储至关重要，所以主存储器在操作系统管理的资源中等级很高。操作系统维护空闲内存块的大小和位置信息，并且在有请求的情况下分配给并发的程序。当进程结束的时候系统回收分配给它的内存，以便再次使用。 [139]

在大部分嵌入式系统中，主存包括一段从地址 $0 \sim N-1$ 连续的位置集合，代码和数据都存储在存储器（内存）中。小型的嵌入式系统可能使用两种存储器技术：

- 只读存储器（ROM）：包括闪存，用来存储程序代码和常量。
- 随机存储器（RAM）：用来存储运行时的变量。

在地址使用方面，两种存储器使用不同的位置。比如，地址 $0 \sim K-1$ 是 ROM，地址 $K \sim N-1$ 是 RAM[⊖]。

有些系统进一步区分存储器技术的特定类型。例如，RAM 可以分成两类：

- 静态 RAM（SRAM）：更快，但是更贵。
- 动态 RAM（DRAM）：价廉，但是更慢。

由于 SRAM 价格昂贵，所以系统通常使用少量的 SRAM 和大量的 DRAM。如果存储器的类型不同，程序员就需要仔细地将频繁使用的变量放在 SRAM，将很少使用的变量放在 DRAM。

9.3 重量级进程的定义

大型操作系统提供了一种保护机制，阻止程序访问分配给其他程序的内存区域。第 10 章讨论虚拟地址空间如何分配给不同的程序。重量级进程抽象（heavyweight process abstraction）所采取的方法是首先创建地址空间，然后创建一个运行在该地址空间上的进程。通常，重量级进程的代码采用动态加载——程序在被重量级进程使用之前必须编译并存储在硬盘上的文件中。因此，创建重量级进程的时候，程序员需要指定硬盘上包含编译代码的文件，操作系统将选定的程序装载到新的虚拟地址空间中，然后启动一个进程来执行该程序。

有趣的是，有些支持重量级进程抽象的操作系统使用一种混合的方法，其中包含轻量级进程抽象（lightweight process abstraction）（例如，线程）。操作系统允许用户创建多个执行在同一个地址空间内的线程，而不是只有一个。

在一个混合的系统中，线程与 Xinu 中的进程十分相似。每个线程拥有一个独立的运行时栈，用来存储局部变量和函数调用。重量级进程分配的所有线程共享全局变量，全局变量从重量级进程地址空 [140]

⊖ 通常 K 和 N 都是 2 的幂。

间的数据区域中分配。共享意味着协调——一个重量级进程的不同线程必须使用同步原语，比如使用信号量控制共享变量的访问。图 9-1 展示了一个混合系统。

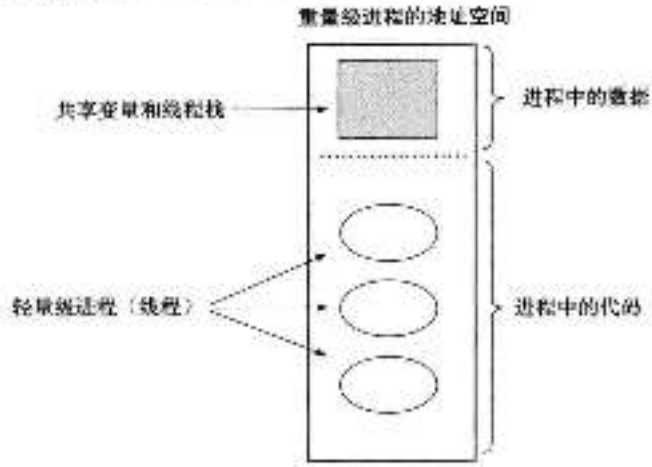


图 9-1 用多个轻量级进程（线程）共享地址空间来说明重量级进程抽象的概念

9.4 小型嵌入式系统的内存管理

大型嵌入式系统，例如应用在视频游戏控制中的系统，拥有支持虚拟地址空间的二级存储和内存管理硬件。然而，在小型嵌入式系统中，硬件不能支持多个地址空间，不能保护进程。因此，操作系统和所有的进程共享一个地址空间。

虽然在单个地址空间内运行多个进程缺少了保护，但是这种方法也有其优势。这些进程可以相互之间传递指针，共享大量数据而无需从一个地址空间复制到另一个地址空间。此外，操作系统可以取消指针引用，因为地址解析不依赖于进程上下文。最后，只有一个地址空间使得内存管理器比复杂系统中的简单很多。

[141]

9.5 程序段和内存区域

当 Xinu 为小型嵌入式系统进行编译时，内存映像被分为四个连续的区域：

- 文本段。
- 数据段。
- bss 段。
- 空闲空间。

文本段（text segment） 文本段从内存单元 0 开始，包含每个函数的编译代码在内存中的映像。文本段也可能包含常量（例如，字符串常量）。如果硬件有保护机制，那么文本段的地址就分类为只读（read only），意味着如果程序在运行时试图往文本段里写信息，那么将会出现错误。

数据段（data segment） 数据段紧接着文本段，存储有初始值的全局变量。数据段中的值可以被访问或者修改（例如，读、写）。

bss 段（bss segment） 以符号开始的块（block started by symbol，bss），取自设计 C 语言的汇编语言 PDP-11。bss 段紧接着数据段，存储没有被初始化的全局变量。和 C 的习惯一样，在程序开始前，Xinu 在每个 bss 段位置上写入 0。

空闲空间（free space） 超过 bss 段的内存在程序开始前被认为空闲的。下一节将介绍操作系统如何使用空闲空间。

正如本书第3章所描述的那样，C语言装载器定义了3个外部符号：etext、edata和end[1]，分别用来标记超过文本段的第一块内存单元，超过数据段的第一块内存单元和超过bss段的第一块内存单元。图9-2解释了Xinu开始执行时内存布局和3个外部符号。

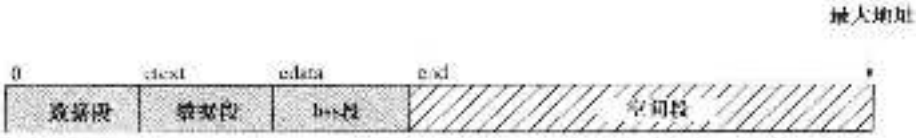


图9-2 Xinu开始执行时内存布局

图9-2中的外部符号不是变量，而是当映像装载时指向内存位置的名字。因此，程序只能使用某个外部符号来引用位置，而不应该装载一个值。例如，文本段占用内存单元0－etext－1。为了计算大小，程序定义etext为一个外部的整数，在表达式中的引用方式为&etext。 142

程序怎样确定空闲空间的大小？操作系统必须配置最大物理内存地址或者在启动时探寻内存[2]。在启动时，系统将最大有效内存放入全局变量maxheap中。因此，为了计算空闲空间的初始大小，内存管理代码取变量maxheap和变量&end的差值。

9.6 嵌入式系统中的动态内存分配

虽然在地址空间里分配有确定的位置，并且一直占用物理内存，程序段和全局变量却只占据一个执行进程所用内存的一部分。另外两种内存段的类型是：

- 栈。
- 堆。

栈（stack） 每个进程都需要栈的空间来存储与每个进程的函数调用相关的活动记录。除了参数外，这种活动记录还包括局部变量。

堆（heap） 一个进程或者一系列进程可能使用堆来存储动态分配的变量，这些变量与特定的函数调用无关。

Xinu使用这两种动态内存形式。第一，当创建一个新的进程时，Xinu为这个进程分配一个栈。栈从空闲空间的最高地址进行分配。第二，当一个进程请求堆存储时，Xinu从空闲空间的低地址分配必要大小的空间给该进程。图9-3显示了3个进程在执行时的内存布局，此时堆已经分配。

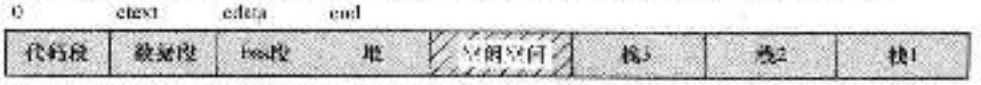


图9-3 3个进程创建后的内存 143

9.7 低层内存管理器的设计

一组函数和数据结构被用于管理空闲内存。低层内存管理器提供了4个这样的函数：

- getstk：当进程创建时分配栈空间。
- freestk：当进程终止时释放栈。
- getmem：按需分配堆存储。
- freemem：收到请求时释放堆存储。

我们的设计把空闲空间当成一块连续的，可用尽的资源——低层内存管理器负责分配可能满足请求的空间。此外，低层内存管理器不会把空闲内存分为提供给进程栈的内存和提供给堆的内存。请求一种类型的内存会占用剩余的空闲空间，这些空间不会留给另外一种类型。当然，这样一种分配只会在进程协作时有效。否则，进程会消耗所有空闲内存而不会留给其他进程。第10章介绍另外一种方

[1] 外部符号名会被装载器加上下划线，因此，etext变为_etext。
[2] 因为物理内存在E21001中被复制过，所以查看内存很困难。

式，一系列高层内存管理方法会阻止划分了系统时的内存容量。高层内存管理器还展示如何在分配内存前阻塞进程。

当创建一个新的进程时，create 调用 getstk 来分配栈。getstk 从空闲空间的高地址获取一块内存，返回指向该块的指针。create 在进程表项里记录栈空间的大小和位置，将 CONTEXT 区域内的栈地址放在栈顶。然后，当这个进程变成当前程序时，就进行上下文切换来访问 CONTEXT 区域，装载这个栈地址到栈指针寄存器。最后，当进程终止时，kill 函数调用 freestk 来释放进程的栈，将该块内存返回到空闲链表中。

函数 getmem 和 freemem 为堆存储执行相似的任务。与栈分配方法不同，getmem 和 freemem 在空闲空间的最低地址处分配内存块。

9.8 分配策略和内存持久性

因为只有 create 和 kill 分配和释放进程栈空间，所以系统可以保证分配给进程的栈空间会在进程退出时释放。然而，系统不会记录进程调用 getmem 所得到的堆块。因此，系统不会自动地释放堆空间。这样，回收堆空间的任务就留给了程序员：

> 堆空间与分配该堆的进程无关。在进程退出之前，进程必须显式地释放其得到的堆空间，否则该空间会一直处于分配的状态。

当然，释放堆空间并不能保证堆不会耗尽。一方面，需求可能超过可利用的空间；另外一方面，空闲空间可能变成很小的、不连续的碎片，以至于无法满足请求。第 11 章将继续讨论分配策略，展示出一种避免空闲空间碎片的方法。

9.9 追踪空闲内存

内存管理器必须知道所有的空闲内存块的信息。为了实现该目的，内存管理器维持一张链表，表中的每一项代表一个块开始的地址和块的长度。开始，这张表只有一项，代表程序结尾与内存中最高地址之间的内存块。当一个进程请求内存块时，内存管理器搜寻这张表，找到一块空闲区域，然后分配给其请求大小的块，更新这张表就会显示出空闲内存被分配的程度。类似地，当一个进程释放之前被分配的块时，内存管理器会将这个块加到表中。图 9-4 展示了有四块空闲内存块的例子。

块	地址	长度
1	0x84F800	4096
2	0x850F70	8192
3	0x8A03F8	8192
4	0x8C01D0	4096

图 9-4 四个空闲存储块的概念列表

内存管理器必须小心地检查每次存储操作，以避免产生不规则长度的表。当释放一块时，内存管理器扫描整个链表查看该块是否与已经存在的空闲块的末端相邻。如果是，则无需加入新的表项，只需要增加原有块的大小。类似地，如果该块与已经存在的空闲块开始处相邻，则更新这个表项。最后，如果释放的块恰好填充两个空闲块之间的空隙，内存管理器就将这两个表项合并成一个大的块，大的块涵盖了两个空闲块和刚释放的块。我们用联合（coalesce）表示合并表项。关键是，一旦将所有分配的块释放，这个表就回到初始状态，即只有一个表项代表程序结束处和内存最高处之间的内存块。

144
|
145

9.10 低层内存管理的实现

空闲内存块的链表存储在哪里？我们例子的实现遵循一种标准方法，即用空闲内存自身存储这张链表。毕竟，如果不使用空闲内存块，也就不需要这张表。因此，空闲内存块可以用指针连接起来形成链表。

在代码中，全局变量 memlist 拥有一个指向第一个空闲块的指针。理解这种实现的关键是一句话占不变的话：

> 所有的空闲内存块都存在链表中，空闲表中的块以地址递增顺序排序。

图 9-4 中的概念链表展示了与每个表项相联系的两个字段：地址和大小。在我们的链表实现中，

每个链表中的结点指向下一个结点（例如，通过存储地址的方式）。最后，我们必须存储每个块的大小。因此，每个空闲内存块包含两个字段：一个指向下一个空闲内存块的指针，一个给出当前块大小的整数。图 9-5 解释了这种概念。

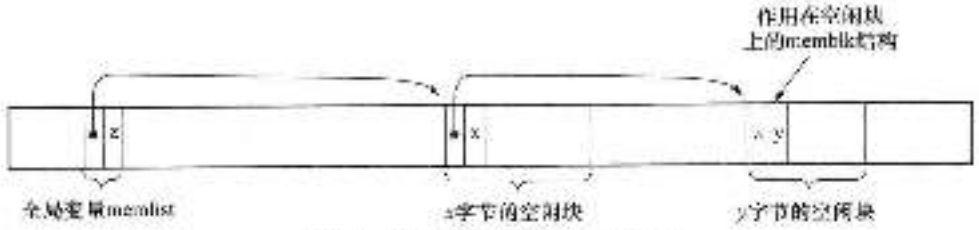


图 9-5　有两个内存块的空闲内存表

定义在 memory.h 中的结构 memblk 给出了可以实现每个空闲结点的数据结构的轮廓。在结构 memblk 中，mnext 字段指向链表中的下一个块，如果该块为最后一个块则赋值为 NULL。mlength 字段定义了当前块的长度（包含表头，用字节表示）。注意，长度是 unsigned long 变量，这样最大空间可以是整个物理地址空间。

变量 memlist 组成了这个链表的头部，该变量定义为 memblk 结构。因此，表头与表中其他的结点有完全相同的形式。然而，mlength 字段（用来存储块的大小）在 memlist 中却没有意义，因为 memlist 的大小可以直接表示为 sizeof（struct memblk）。所以，我们可以让长度字段有其他用途。Xinu 使用这个字段存储整个空闲内存的大小（例如，每个块中长度字段的总和）。知道空闲内存的大小可以在调试或者判断系统是否接近最大值时起到作用。

注意，每个空闲表中的块必须有完整的 memblk 结构（例如，8 字节）。这样的设计包含有缺点：内存管理器不能存储小于 8 字节的空闲块。我们如何保证不会有进程试图释放一个更小的内存呢？我们可以告诉程序员必须释放与请求相同的内存，并且让内存管理程序跟踪所有的请求必须至少为 8 字节。但是如果内存管理器选取一块空闲内存时，另外一个问题就可能产生：剩余的内存可能小于 8 字节。为了解决这个问题，内存管理器使所有的请求是 memblk 结构长度的倍数。使用两个内联函数，roundmb 和 truncmb，实现这种功能。roundmb 将请求取整为 8 字节的倍数，truncmb 用来将内存大小截所为 8 字节的倍数。我们只要写一次，空闲空间初始的大小必须截断而不是取整。关键点是：

> 所有的请求取整为 memblk 结构大小的倍数，确保每个请求满足约束条件，保证不会有空闲块太小而不能被返到空闲链表中。

文件 memory.h 包含与内存管理有关的声明，包括两个内联函数 roundmb 和 truncmb 的定义。为了高效地实现，这两个方法采用位数和布尔运算而不是 sizeof 函数和除法。使用布尔运算是可行的，因为内存块的大小是 2 的幂。

```
/* memory.h - roundmb, truncmb, freestk */

#define PAGE_SIZE       4096
#define MAXADDR         0x02000000      /* 16MB Xinu 32MB RAM           */

/*------------------------------------------------------------------------
 * roundmb, truncmb - round or truncate address to memory block size
 *------------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( 7 + (ulong)(x) & (~7) )
#define truncmb(x)      (char *)( (ulong)(x) & (~7) )

/*------------------------------------------------------------------------
 *  freestk  --  free stack memory allocated by getstk
 *------------------------------------------------------------------------
 */
```

```
#define freestk(p,len)  freemem((char *)((uint32)(p)                    \
                                  - ((uint32)roundmb(len))               \
                                  + (uint32)sizeof(uint32)),             \
                                  (uint32)roundmb(len) )

struct  memblk  {                       /* see roundmb & truncmb          */
        struct  memblk  *mnext;         /* ptr to next free memory blk    */
        uint32  mlength;                /* size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* head of free memory list       */
extern  void    *maxheap;               /* max free memory address        */
extern  void    *minheap;               /* address beyond loaded memory */

/* added by linker */

extern  int     end;                    /* end of program                 */
extern  int     edata;                  /* end of data segment            */
extern  int     etext;                  /* end of text segment            */
```

9.11 分配堆存储

函数 getmem 通过寻找满足请求的空闲块来分配堆存储。我们的实现使用最先适应（first-fit）分配策略，该策略分配空闲链表上第一个满足请求的块。getmem 从找到的空闲块中减去请求的内存大小并相应地调整空闲链表。文件 getmem.c 包含了这部分源代码。

148

```
/* getmem.c - getmem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char    *getmem(
          uint32        nbytes          /* size of memory requested       */
        )
{
        intmask mask;                   /* saved interrupt mask           */
        struct  memblk  *prev, *curr, *leftover;
                                        *
        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* use memblk multiples */

        prev = &memlist;
        curr = memlist.mnext;
        while (curr != NULL) {                  /* search free list      */

                if (curr->mlength == nbytes) {  /* block is exact match */
                        prev->mnext = curr->mnext;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);
```

```
        } else if (curr->mlength > nbytes) { /* split big block */
                leftover = (struct memblk *)((uint32) curr +
                                nbytes);
                prev->mnext = leftover;
                leftover->mnext = curr->mnext;
                leftover->mlength = curr->mlength - nbytes;
                memlist.mlength -= nbytes;
                restore(mask);
                return (char *)(curr);
        } else {                        /* move to next block   */
                prev = curr;
                curr = curr->mnext;
        }
    }
    restore(mask);
    return (char *)SYSERR;
}
```

验证了参数合法并且空闲链表不为空后，getmem 使用 roundmb 函数把请求的内存大小调整为 memblk 大小的倍数，然后搜索空闲链表寻找第一个大小满足请求的内存块。由于空闲链表是单向链表，所以 getmem 使用 prev 和 curr 两个指针来遍历链表。getmem 在搜索过程中维护如下不变式：当 curr 指向一个空闲块时，prev 指向该块在链表中的前驱块（可能为链表的头结点，memlist）。遍历链表的过程中，代码必须保证该不变式保持不变。因此，当找到一个大小满足请求的空闲块时，prev 指向块的前驱块。

在遍历链表的每一步，getmem 把当前块的大小和请求的大小 nbytes 进行比较。比较的结果有 3 种情况。如果当前块的大小小于请求的大小，则 getmem 移动到链表的下一块继续进行搜索；如果当前块的大小正好和请求的大小相等，则 getmem 从空闲链表中删除该块（通过把该块前驱块的 mnext 字段设置为该块的后继块来实现），然后返回指向该块的指针；如果当前块的大小大于请求的大小，则 getmem 把当前块分割为两块：一块大小为 nbytes，该块将返回给调用者；剩余部分将留在空闲链表上。进行块分割时，getmem 计算剩余部分的地址并保存在变量 leftover 中。计算剩余部分的地址在概念上很简单：剩余部分在该块开头之后的 nbytes 处。但是，把 curr 增加 nbytes 并不能产生期望的结果，因为 C 语言对指针进行的是指针运算。为了强制 C 语言使用整数运算，在与 nbytes 相加前，curr 指针需要先强制转换为一个无符号整数（(uint32) curr）。当计算出结果后，再使用强制转换把计算结果转换回指向内存块的指针。通过上述方法计算出 leftover 后，更新 prev 块的 mnext 字段，并且相应地对 leftover 块的 mnext 和 mlength 字段进行赋值。

这部分代码依赖一个基本的数学关系：减去两个 K 的倍数将产生另一个 K 的倍数。在上述的例子中，K 的大小为一个 memblk 结构的大小。因此，如果系统开始时使用 roundmb 对空闲内存的大小进行取整并使用 roundmb 调整请求的大小，则每个空闲块和每个剩余部分都将足够容纳一个 memblk 结构。 150

9.12 分配栈存储

函数 getstk 给进程动态分配一块内存。创建进程时都调用 getstk。这部分代码在文件 getstk.c 中。

由于空闲链表按照内存地址有序排列并且动态空间从地址最高的可用块分配，所以 getstk 必须搜索整个空闲链表。在搜索过程中，getstk 记录任何一个能满足请求的块地址[⊖]。这意味着在搜索完成后，最后记录的地址指向满足请求并且地址最高的空闲块。和 getmem 一样，getstk 在搜索过程中维护相同的不变式，变量 curr 和 prev 分别指向一个空闲块和该空闲块的前驱块。当找到一个大小满足请求的块时，getstk 把变量 fits 设置为该块的地址并把变量 fitprev 设置为该块前驱块的地址。因此，当搜索完成后，fits 指向可用的、内存地址最高的空闲块（如果没有满足请求的块，fits 等于 NULL）。

当搜索结束并找到一个块时，与 getmem 类似，这里有两种情况。如果这个块的大小正好和请求的大小相等，则 getstk 从空闲链表中移除该块并返回该块的地址给调用者；否则，getstk 把该块分为两

⊖ 分配具有最高地址并满足请求的块的策略称为最后适配（last-fit）策略。

块，一块分配 nbytes 大小，另一块留在空闲链表上。由于 getstk 需要返回选出块的最高地址部分，所
151 以地址计算和 getmem 中的方法稍微有些不同。

```
/* getstk.c - getstk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char	*getstk(
	  uint32	nbytes		/* size of memory requested	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct	memblk	*prev, *curr;	/* walk through memory list	*/
	struct	memblk	*fits, *fitsprev; /* record block that fits	*/

	mask = disable();
	if (nbytes == 0) {
		restore(mask);
		return (char *)SYSERR;
	}

	nbytes = (uint32) roundmb(nbytes);	/* use mblock multiples */

	prev = &memlist;
	curr = memlist.mnext;
	fits = NULL;

	while (curr != NULL) {			/* scan entire list	*/
		if (curr->mlength >= nbytes) {	/* record block address */
			fits = curr;		/*   when request fits  */
			fitsprev = prev;
		}
		prev = curr;
		curr = curr->mnext;
	}

	if (fits == NULL) {			/* no block was found	*/
		restore(mask);
		return (char *)SYSERR;
	}
	if (nbytes == fits->mlength) {		/* block is exact match */
		fitsprev->mnext = fits->mnext;
	} else {				/* remove top section	*/
		fits->mlength -= nbytes;
		fits = (struct memblk *)((uint32)fits + fits->mlength);
	}
	memlist.mlength -= nbytes;
	restore(mask);
	return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

9.13 释放堆和栈存储

当使用完一块堆存储后，进程调用 freemem 函数把该块返回到空闲链表中，使这块内存可以被后

块的内存分配使用。由于空闲链表中的块按照地址顺序保存，所以 freemem 根据该块的地址找到其在链表上的正确位置。另外，freemem 还负责合并（coalescing）该块与相邻的空闲块。合并存在 3 种情况：新的内存块与前驱块相邻；新的内存块与后继块相邻；或者与两块都相邻。当这 3 种情况中的任一种出现时，freemem 把新块和相邻块合并，在空闲链表上形成一个大的块。合并可以帮助避免产生内存碎片。

函数 freemem 的代码可以在 freemem.c 文件中找到。和 getmem 相似，prev 和 next 两个指针用来遍历空闲块链表。freemem 搜索空闲链表直到返回的块地址处于 prev 和 next 之间。一旦找到了正确的位置，代码就执行块合并。

合并的处理过程分为 3 步。代码首先检查能否与前驱块合并，即 freemem 将前驱块的长度和前驱块的地址相加计算出前驱块之后的内存地址。freemem 把存储在 top 变量中的计算结果与插入块的地址进行比较。如果插入块的地址与 top 变量的值相等，freemem 通过增加前驱块的大小把新块包含在前驱块中；否则，freemem 把新块插入到链表中。当然，如果 prev 指针指向 memlist 的头结点，则不需要执行合并。

当处理完与前驱块的合并后，freemem 检测能否与后继块合并。freemem 计算当前块之后的内存地址并测试该地址是否与后继块的地址相等。如果相等，说明当前块与后继块相邻，则 freemem 增加当前块的大小把后继块包含在当前块中，并把后继块从链表中移除。

重要的一点是，freemem 处理了 3 种特殊情况：

> 当向空闲链表中添加块时，内存管理器必须检查新的块是否与前驱块相邻、与后继块相邻，或者与两块都相邻。

152
|
153

```
/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
	  char		*blkaddr,	/* pointer to memory block	*/
	  uint32	nbytes		/* size of block in bytes	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct	memblk	*next, *prev, *block;
	uint32	top;

	mask = disable();
	if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
			  || ((uint32) blkaddr > (uint32) maxheap)) {
		restore(mask);
		return SYSERR;
	}

	nbytes = (uint32) roundmb(nbytes);	/* use memblk multiples */
	block = (struct memblk *)blkaddr;

	prev = &memlist;			/* walk along free list */
	next = memlist.mnext;
	while ((next != NULL) && (next < block)) {
		prev = next;
		next = next->mnext;
	}
```

```
if (prev == &memlist) {          /* compute top of previous block*/
        top = (uint32) NULL;
} else {
        top = (uint32) prev + prev->mlength;
}

/* Insure new block does not overlap previous or next blocks    */

if ((prev != &memlist) && (uint32) block < top)
    | ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
        restore(mask);
                return SYSERR;
        }

        memlist.mlength += nbytes;

        /* Either coalesce with previous block or add to free list */

        if (top == (uint32) block) {     /* coalesce with previous block */
                prev->mlength += nbytes;
                block = prev;
        } else {                         /* link into list as new node   */
                block->mnext = next;
                block->mlength = nbytes;
                prev->mnext = block;
        }

        /* Coalesce with next block if adjacent */

        if ((uint32) block + block->mlength) == (uint32) next) {
                block->mlength += next->mlength;
                block->mnext = next->mnext;
        }
        restore(mask);
        return OK;
}
```

由于空闲内存可以用做栈或堆存储的单一资源，所以释放栈内存调用与释放堆存储相同的算法。堆分配和栈分配唯一的不同点在于 setmem 返回分配块的最低地址，而 getstk 返回分配块的最高地址。在当前实现中，freestk 是一个调用 freemem 的内联函数。在调用 freemem 之前，freestk 必须把它的参数从块的最高地址转换到最低地址。这部分代码可以在 memory.h[⊖]中找到。虽然当前的实现使用一个底层单链表，但把 freestk 和 freemem 分开能保持概念区别，并且之后更容易对实现进行修改。它的要点是：

154
-
155

> 尽管与前的实现使用相同的底层函数来释放堆和栈存储，但是为 freestk 和 freemem 生成不同的系统调用保持概念区别并使系统以后更容易改变。

9.14 观点

虽然机制相对简单，但内存管理子系统的设计展现了操作系统中最令人惊讶的微妙之处。这个问题主要的冲突引起：一方面，操作系统设计为不间断运行，因此操作系统必须节约资源：当一个进程使用完一个资源后，系统必须回收该资源并使其对其他进程可用。另一方面，任何允许进程分配和释放任意大小内存块的内存管理机制都不是资源节约的，因为空闲内存可能要分割为小而不连续的块从而使内存碎片化。因此，设计者必须进行折中。允许分配任意大小内存使系统更容易实用，但也导

⊖ 文件 memory.h 可以在 9.9 节找到。

入了一些潜在的问题。

9.15 总结

大型操作系统提供了复杂的内存管理方案，允许对内存的请求超过实际内存的大小。这种系统把数据保存在二级存储器中，在引用数据时将数据移到主存中。高级内存管理系统支持多虚拟地址空间，允许每个应用程序从零开始对内存进行取址。我们使用术语重量级进程（heavyweight process）来指代运行在独立地址空间中的应用程序；轻量级进程则允许在一个虚拟空间中执行一个或多个进程。分页已经成为普遍使用的技术，用于提供虚拟地址空间和多路复用它们到物理内存上。

小型嵌入式系统通常把所有代码和数据保存在物理内存中。一个进程包含3段：一段包含已编译代码的文本段，一段包含已初始化数据值的数据段和一段包含未初始化变量的bss段。当系统启动后，没有分配给这三段的物理内存视为空闲内存，一个低层内存管理器被需分配这些空闲内存。

Xinu中的低层内存管理器维护一个空闲内存块链表，堆和栈存储段被需要从链表上进行分配。堆存储的分配通过寻找第一个满足请求的空闲内存块（即具有最低地址的空闲块）来完成。栈存储的分配则选取满足请求且具有最高地址的内存块。由于空闲内存块链表按照地址顺序单向连接，因此分配栈空间需要搜索整个空闲链表。

低层内存管理器把空闲空间视为可耗尽的资源并且栈存储和堆存储之间没有进行分离。由于这种内存管理器不包含防止一个进程分配完所有可用内存的机制，所以程序员必须设计规则来防止内存耗尽。第10章讨论的高级内存管理器将阐述内存如何分配给不同区域和一组进程如何配置等待内存变得可用。

156

练习

9.1 低级内存管理器的一个早期版本没有提供把内存块返回到空闲链表的功能。对嵌入式系统进行推测：freemem 和 freestk 是必要的吗？为什么是或为什么不是，请说明理由。

9.2 使用一组永久性分配堆和栈内存的函数（即不需要提供机制把存储空间返回到空闲链表）替换低级内存管理函数。新分配程序的大小与 getstk 和 getmem 的大小相比结果如何？

9.3 从空闲空间两端分配栈和堆存储的方法能帮助最小化内存碎片吗？为了找出答案，考虑一系列混合了分配和释放 1000 字节栈存储和 500 字节堆存储的请求。比较本章中描述的方法和一个从空闲空间同一端分配栈和堆请求的方法（即所有的内存分配都使用 getmem）。找出一个如果不从同端分配栈和堆请求则会导致内存碎片的请求序列。

9.4 这里描述的内存管理系统能分配任意小数量的内存吗？为什么能或为什么不能？

9.5 许多嵌入式系统经历一个原型阶段，在该阶段系统建立在一个运用的平台上；一个最终阶段，在该阶段为系统设计最小化的硬件。对内存管理来说，何需关注每个进程需要的栈的大小。修改程序代码使系统能测量一个进程使用的最大栈空间并在进程退出时报告最大栈空间大小。

157

高级内存管理和虚拟内存

是的，我会从我的记忆里擦去所有艰辛的记录。

——威廉·莎士比亚

10.1 引言

前面的章节考虑了操作系统管理计算和I/O的抽象。第9章描述了一个把内存看做可耗尽资源的低级内存管理工具，讨论了地址空间、程序段和管理全局空闲链表的函数。尽管它们是必要的，但低级的内存管理工具并不能满足所有的需求。

本章将通过介绍高级的工具来完成对内存管理的讨论。本章解释了把内存资源划分为独立子集的动机，介绍了一个允许把内存分成独立缓冲池的高级内存管理机制，并解释了如何分配和释放一个池中的内存而不影响其他池中内存的使用。本章还介绍了虚拟内存以及与虚拟内存相关的硬件是如何工作的。

[159]

10.2 分区空间分配

第9章描述的getmem和freemem函数组成了一个基本的内存管理器。它的设计没有设定一个进程可以分配的内存数量的上限，也没有尝试"公平地"分配空闲空间。相反，它的分配函数仅仅使用先来先服务的方式处理请求，直到没有空闲的内存为止。当空闲内存用尽后，函数拒绝后续的请求。它不是阻塞进程或等待内存释放。虽然它比较有效，但全局的分配策略迫使所有的进程争夺相同的内存。这会导致副作用——一个或多个进程由于所有内存无关而无法获得内存。因此，全局的内存分配方案并不适合操作系统的所有部分。

为了理解为什么系统不能依赖全局分配，我们考虑网络通信软件。网络通信中数据包随机到达。由于网络应用程序需要时间来处理数据包，所以在处理一个数据包时额外的数据包也可能会到达。如果每个传入的数据包都放在一个内存缓冲区中，这种耗尽的分配可能导致灾难。传入的报文堆积等待处理，而且每一个都占用内存。最坏的情况下，所有的可用空间都分配给数据包缓冲区，使其他的操作系统函数没有内存可以使用。特别地，如果磁盘I/O使用内存，则所有的磁盘I/O将中断直到内存变得可用。如果处理网络数据包的程序尝试写文件，则可能导致死锁：该进程阻塞等待磁盘缓冲区，但所有的内存都用于网络缓冲区并且在磁盘I/O完成前没有网络缓冲区能够被释放。

为了防止死锁，高级内存管理必须为把空闲内存划分为独立的子集，并确保分配和释放一个子集与分配和释放其他子集相互独立。通过限制特定函数可以使用的内存数量，系统可以确保定量的请求不会导致全局剥夺。此外，系统可以假设分配给特定函数的内存总是会被返回，因此它能安排挂起进程直到满足它们的内存请求，从而消除任等待产生的开销。分区不能保证不发生死锁，但它确实能限制由于一个子系统占有另一个子系统需要的内存而产生的无意识的争锁。

10.3 缓冲池

我们使用缓冲池（buffer pool）管理器来处理内存更分配函数，将内存划分到一组缓冲池中。每个缓冲池包含固定数量的内存块，缓冲池中的每一个内存块大小是一样的。缓冲区（buffer）这个术语用来反映I/O程序和通信软件对内存的频繁使用（例如，磁盘缓冲区、网络数据包缓冲区）。

[160]

当创建缓冲池时，内存空间分配到其一组缓冲区中。一旦已经分配了一个缓冲池，缓冲池中的缓

冲区数目就不能增加了，缓冲区大小也不能改变。

每个缓冲池由一个整数来标识，这个整数称为缓冲池标识符（pool identifier）或者缓冲池 ID（buffer pool ID）。与 Xinu 中的其他标识符一样，缓冲池标识符用来作为进入缓冲池表（buftab）的索引。一旦建立了一个缓冲池，进程就可以使用缓冲池标识符向该缓冲池请求分配缓冲区或者释放之前分配的缓冲区。分配和释放缓冲区的请求不需要指定缓冲区的大小，因为缓冲区的大小在缓冲池创建的时候已经确定了。

缓冲池机制与低级内存管理器的不同之处在于它采用了同步（synchronous）机制。也就是说，一个进程在请求分配缓冲区时会被阻塞，直到请求可以满足才继续执行。与之前的许多例子一样，同步机制使用信号量来实现对缓冲池的访问控制。每个缓冲池有一个信号量。分配缓冲区的代码调用 wait 等待信号量。如果缓冲池中的缓冲区还有剩余，则调用立即返回；如果没有，则调用会被阻塞。最后，当另一个进程把自己使用的缓冲区释放到缓冲池时，信号量会发信号通知等待的进程获得缓冲区并继续执行。

Xinu 系统使用一张表作为保存缓冲池信息的数据结构。表中的每个表项记录了缓冲区的大小、信号量 ID 和一个指向链表中下一个缓冲区的指针。相关的声明可以在 bufpool.h 文件中找到：

[161]

```
/* bufpool.h */

#ifndef NBPOOLS
#define NBPOOLS 20              /* Maximum number of buffer pools       */
#endif

#ifndef BP_MAXB
#define BP_MAXB 8192            /* Maximum buffer size in bytes         */
#endif

#define BP_MINB 8               /* Minimum buffer size in bytes         */
#ifndef BP_MAXN
#define BP_MAXN 2048            /* Maximum number of buffers in a pool  */
#endif

struct  bpentry {               /* Description of a single buffer pool  */
        struct  bpentry *bpnext;/* pointer to next free buffer          */
        sid32   bpsem;          /* semaphore that counts buffers        */
                                /*    currently available in the pool   */
        uint32  bpsize;         /* size of buffers in this pool         */
        };

extern  struct  bpentry buftab[];/* Buffer pool table                   */
extern  bpid32  nbpools;        /* current number of allocated pools    */
```

结构 bpentry 定义了缓冲池表 buftab 中表项的内容。缓冲池中的缓冲区由链表形式记录。bpnext 字段用于指向表中的第一个缓冲区。信号量 bpsem 控制缓冲区的分配。整数 bpsize 表示缓冲区的大小。

10.4　分配缓冲区

Xinu 有三个函数提供了和缓冲池有关的接口。其中一个函数是 mkpool，它用来创建缓冲池并获取缓冲池 ID。当缓冲池创建之后，就可以调用 getbuf 函数来获取缓冲区，也可以调用 freebuf 函数来释放缓冲区。

getbuf 函数就像名字所表述的那样工作，它等待信号量直到有一个缓冲区可用，然后从链表中将第一个缓冲区移去。相关代码在 getbuf.c 文件中可以找到：

[162]

```
/* getbuf.c - getbuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getbuf  --  get a buffer from a preestablished buffer pool
 *------------------------------------------------------------------------
 */
char	*getbuf(
	  bpid32	poolid		/* index of pool in buftab	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct	bpentry *bptr;		/* pointer to entry in buftab	*/
	struct	bpentry *bufptr;	/* pointer to a buffer		*/

	mask = disable();

	/* Check arguments */

	if ( (poolid < 0)  ||  (poolid >= nbpools) ) {
		restore(mask);
		return (char *)SYSERR;
	}
	bptr = &buftab[poolid];

	/* Wait for pool to have > 0 buffers and allocate a buffer */

	wait(bptr->bpsem);
	bufptr = bptr->bpnext;

	/* Unlink buffer from pool */

	bptr->bpnext = bufptr->bpnext;

	/* Record pool ID in first four bytes of buffer and skip */

	*(bpid32 *)bufptr = poolid;
	bufptr = (struct bpentry *)(sizeof(bpid32) + (char *)bufptr);
	restore(mask);
	return (char *)bufptr;
}
```

细心的读者已经注意到，getbuf 函数并不直接将缓冲区的地址返回给它的调用者，而是把缓冲池 ID 存储在分配空间的前 4 个字节，只返回该 ID 以外的地址。从调用者的角度看，调用 getbuf 返回缓冲区的地址，调用者不需要知道前面的字节保存了缓冲池 ID，这样对系统是透明的。当创建缓冲池的时候，缓冲池 ID 保存在每一个缓冲区中额外分配的空间中。当释放一个缓冲区的时候，freebuf 函数使用隐藏的缓冲池 ID 来确定该缓冲区是属于哪个缓冲池。当缓冲区由不是申请该缓冲区的进程返还给缓冲池时，这种利用隐藏信息来识别缓冲池的方法就会变得特别有用。

10.5 将缓冲区返回给缓冲池

函数 freebuf 的作用是将缓冲区返回给分配缓冲区给它的缓冲池。相关代码在 freebuf.c 文件中可以找到：

```
/* freebuf.c - freebuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freebuf  --  free a buffer that was allocated from a pool by getbuf
 *------------------------------------------------------------------------
 */
syscall freebuf(
	  char		*bufaddr	/* address of buffer to return	*/
	)
{
	intmask mask;			/* saved interrupt mask		*/
	struct	bpentry *bpptr;		/* pointer to entry in bftab	*/
	bpid32	poolid;			/* ID of buffer's pool		*/

	mask = disable();

	/* Extract pool ID from integer prior to buffer address */

	bufaddr -= sizeof(bpid32);
	poolid = *(bpid32 *)bufaddr;
	if (poolid < 0  ||  poolid >= nbpools) {
		restore(mask);
		return SYSERR;
	}
	/* Get address of correct pool entry in table */

	bpptr = &bftab[poolid];

	/* Insert buffer into list and signal semaphore */

	((struct bpentry *)bufaddr)->bpnext = bpptr->bpnext;
	bpptr->bpnext = (struct bpentry *)bufaddr;
	signal(bpptr->bpsem);
	restore(mask);
	return OK;
}
```

在分配缓冲区时，getbuf 函数在缓冲区地址的前四个字节记录了缓冲池 ID。freebuf 将缓冲区的前四个字节还原来获得缓冲池 ID，之后对缓冲池 ID 进行验证。若验证有效，则 getbuf[⊖] 用该 ID 来找到缓冲池表中的表项。然后，将返还的缓冲区链接到缓冲区的链表中。最后给 bpsem 缓冲池信号量发信号，允许其他进程使用缓冲区。

10.6　创建缓冲池

函数 mkbufpool 用来创建了一个新的缓冲池并返回其 ID。mkbufpool 有两个参数：缓冲区的大小和缓冲区的数量。

163 ~ 165

```
/* mkbufpool.c - mkbufpool */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mkbufpool  --  allocate memory for a buffer pool and link the buffers
 *------------------------------------------------------------------------
 */
```

⊖ 根据上下文理解应该是 freebuf——译者注。

```
bpid32  mkbufpool(
          int32         bufsiz,         /* size of a buffer in the pool */
          int32         numbufs         /* number of buffers in the pool*/
        )
{
        intmask mask;                   /* saved interrupt mask         */
        bpid32  poolid;                 /* ID of pool that is created   */
        struct  bpentry *bpptr;         /* pointer to entry in buftab   */
        char    *buf;                   /* pointer to memory for buffer */

        mask = disable();
        if (bufsiz<BP_MINB || bufsiz>BP_MAXB
            || numbufs<1 || numbufs>BP_MAXN
            || nbpools >= NBPOOLS) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        /* Round request to a multiple of 4 bytes */

        bufsiz = ( (bufsiz + 3) & (~3) );

        buf = (char *)getmem( numbufs * (bufsiz+sizeof(bpid32)) );
        if ((int32)buf == SYSERR) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        poolid = nbpools++;
        bpptr = &buftab[poolid];
        bpptr->bpnext = (struct bpentry *)buf;
        bpptr->bpsize = bufsiz;
        if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
                nbpools--;
                restore(mask);
                return (bpid32)SYSERR;
        }
        bufsiz+=sizeof(bpid32);
        for (numbufs-- ; numbufs>0 ; numbufs-- ) {
                bpptr = (struct bpentry *)buf;
                buf += bufsiz;
                bpptr->bpnext = (struct bpentry *)buf;
        }
        bpptr = (struct bpentry *)buf;
        bpptr->bpnext = (struct bpentry *)NULL;
        restore(mask);
        return poolid;
}
```

mkbufpool 首先检查函数的参数。如果缓冲区大小超出范围，或者请求的缓冲区数是负数，或者缓冲池表已经满了，那么 mkbufpool 就会报错。mkbufpool 计算持有这些缓冲区所需要的内存大小，调用 getmem 分配所需要的内存。如果内存分配成功，mkbufpool 在缓冲池表中分配并填充表项。mkbufpool 创建一个信号量，记录缓冲区大小，在链表指针 bpnext 中存储分配内存的地址。

在缓冲池表的表项初始化后，mkbufpool 开始依次访问已分配的内存，并把内存块分成一组组的缓冲区。mkbufpool 将每个缓冲区链接到空闲链表中。注意，当 mkbufpool 创建空闲链表时，每个内存块的大小包含了用户申请的缓冲区大小加上缓冲池 ID 的大小（4 字节）。因此，缓冲池 ID 存入内存块后，余下的充足空间是留给用户申请的缓冲区。mkbufpool 在创建空闲链表后，将缓冲池 ID 返回给调用者。

10.7　初始化缓冲池表

166
-
167

函数 bufinit 用来初始化缓冲池表。代码很简单，可以在 bufinit.c 文件中找到：

```
/* bufinit.c - bufinit */

#include <xinu.h>

struct  bpentry buftab[NBPOOLS];                /* buffer pool table    */
bpid32  nbpools;

/*------------------------------------------------------------------------
 *  bufinit  --  initialize the buffer pool data structure
 *------------------------------------------------------------------------
 */
status  bufinit(void)
{
        nbpools = 0;
        return OK;
}
```

函数 bufinit 所需要做的只是设置记录分配缓冲池数的全局计数器。在示例代码中，缓冲池可以被动态分配，但是一旦分配缓冲池就不能释放。通过扩展缓冲池机制来允许动态释放缓冲池将作为练习题留给读者。

10.8　虚拟内存和内存复用

大多数大型计算机系统都使用虚拟内存技术，并且向应用程序提供理想化的内存视图。每个应用程序就好像可以拥有超过物理内存大小的大内存。操作系统对所有进程需要使用的物理内存进行复用，在需要的时候把所有或者部分应用程序移到物理内存中。也就是说，进程的代码和数据保存在辅助（或称为二级）存储器（磁盘），然后在进程执行的时候暂时将其移入主存储器。尽管很少有嵌入式系统需要虚拟内存，但是许多处理器都带有虚拟内存硬件。

虚拟内存管理系统的主要设计问题是复用的形式。有几种方法已经在使用：

- 交换
- 分段
- 分页

交换（swapping）是指当调度器在执行当前的计算时将所有与计算相关的代码和数据都移到主存储器中。交换方法对长期运行的程序效率最好，例如，文字处理软件在用户输入文档时需要一直运行，168
程序移入主存储器并会驻留很长一段时间。

分段（segmentation）是指在需要时将计算相关的部分代码和数据移入主存储器。可以想象分段就是把每个函数和相关变量放入单独的段里。当调用一个函数的时候，操作系统把含有该函数的段装入主存储器中。较少使用的函数（例如，一个显示错误信息的函数）放在辅助存储器中。理论上，分段比交换使用更少的内存。因为分段允许在需要时只把程序的一部分装入内存中。虽然该方法直觉很好，但很少有操作系统使用动态分段。

分页（paging）是指将每个程序分成许多小且固定大小的页（page）。操作系统将最近引用的页放在主存储器中，把其他页的副本移到辅助存储器中。需要时到辅助存储器中提取页，即当一个运行的程序引用了内存单元 i 时，内存硬件检查包含单元 i 的页是否是驻存的（resident）（即该页当前在内存中）。如果该页不是驻存的，则操作系统将进程挂起（其他进程可以执行），然后向磁盘发送请求以获取所需页的副本。一旦该页被放入主存储器中，操作系统就让挂起的进程处于就绪状态。当进程重新尝试引用单元 i 时，引用就会成功执行。

10.9　实地址空间和虚地址空间

在许多操作系统中，内存管理器都为每个程序提供单独的地址空间（address space）。也就是说，

给应用程序分配从0～$M-1$的私有内存单元。操作系统需要底层硬件把每个地址空间与内存单元进行映射。因此，当一个应用程序引用了零地址时，引用映射的内存单元与进程中的零地址是一致的。当另一个应用程序引用了零地址时，引用映射的是另一个内存单元。因此，虽然多个应用程序可以引用零地址时，但是每个引用映射到单独的内存单元并且应用程序之间互不干扰。为了更准确地表达，我们使用术语物理地址空间（physical address space）或实地址空间（real address space）来表示内存硬件提供的地址空间，用术语虚地址空间（virtual address space）表示一个程序可以访问的地址空间。内存管理器将一个或多个虚地址空间映射到底层物理地址空间。例如，图10-1说明了3个K单元的虚地址空间是如何映射到$3K$的物理地址空间上的。

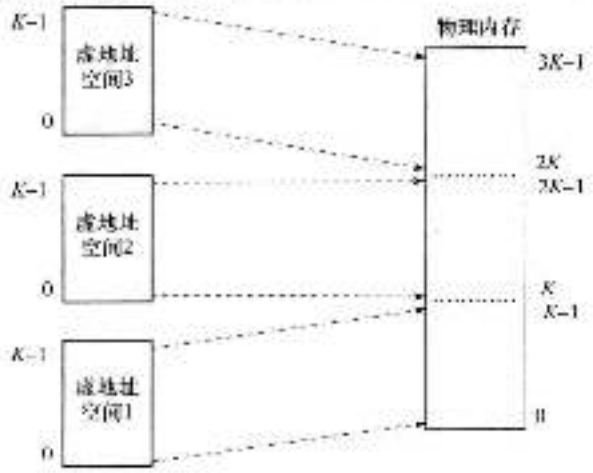


图10-1 3个虚拟地址空间映射到一个物理地址空间

[169] 从运行程序的角度看，只有在虚地址空间中的地址才可以引用。此外，因为操作系统负责将虚地址空间中的地址映射到某一块内存区域中，所以运行的程序不可能偶然地读取或覆盖分配给其他运行程序的内存。因此，提供虚地址空间服务的操作系统可以检测编写错误并预防错误的发生。可以归结如下：

把每个虚地址空间映射到单独内存块的内存管理系统，可以防止一个程序读取或者写入分配给其他程序的内存。

在图10-1中，每个虚拟地址空间都比底层物理地址空间小。但是，大多数内存管理系统都允许虚地址空间比机器上的内存空间大。比如，一个按需换页系统只把引用的页放在主存储器中，将其他页的[170] 副本放在磁盘上。

10.10 支持按需换页的硬件

操作系统进行虚地址与实地址之间映射的操作需要硬件支持。要理解其原因，首先要注意每个地址，包括运行时产生的地址，都需要映射。因此，如果一个程序计算一个值C然后跳转（jump）到单元C，那么内存系统必须把C映射到对应的实内存地址。只有硬件单元可以高效地进行这种映射。

支持按需换页的硬件包含一个页表和一个地址转换单元。页表常驻在核心内存中[⊖]，每个进程有一个页表。一般来说，硬件有两个寄存器，一个指向当前的页表，另一个标识长度。当操作系统在内存中创建一个页表后，操作系统将相关数值分配给寄存器并开启按需换页。类似地，当发生上下文切换时，操作系统改变页表寄存器并指向新进程的页表。图10-2解释了该机制。

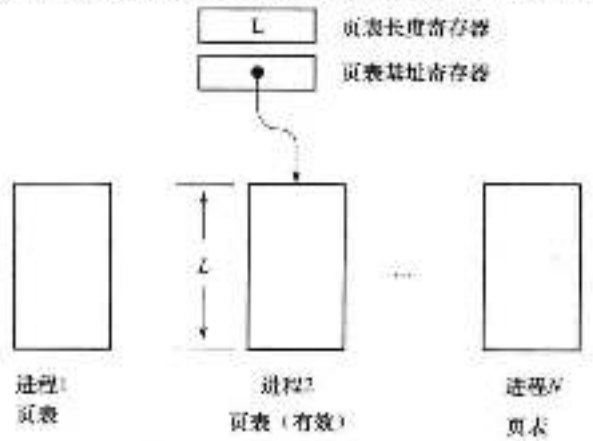


图10-2 内存中的页表和硬件寄存器。页表基寄存器指向了在给定的时间使用的页表

⊖ 核心内存不能被换出，从管理内存块。——译者注。

10.11　使用页表的地址翻译

从概念上说，页表包含了指向内存单元的指针数组。除了指针外，每个表项都包含 1 位元来确定表项是否有效（即，是否已初始化）。地址转换（address translation）硬件使用当前页表来转换内存地址。指令地址和用于存取数据的地址都需要地址转换。地址转换包括数组查找：硬件把地址的高位作为页号（page number），用页号作为页表的索引，根据对应的指针找到内存中页的单元。

实际上，一个页表项并不包含完整的内存指针。相反，页的起始内存单元限制在低位为 0 的内存单元上，页表项省略了地址的低位部分。例如，假设一台计算机的地址是 32 位的，使用 4096 字节大小的页（即，每个页有 2^{12} 字节）。如果内存按一个页框（frame）4096 字节大小来划分，那么每个页框的起始地址（页框第一个字节的地址）低位起的 12 位皆为 0。因此，指向内存中的页框，页表项只需要包含高位的 20 位即可。

为了转换地址 A，硬件首先把地址 A 的高位作为页表的索引，然后从页表指出的内存中抽取页框的地址，最后用地址 A 的低位作为页框的偏移值。图 10-3 说明了地址转换找到物理内存地址的过程。

我们的方法意味着每个地址转换都需要访问页表（即，一次内存访问）。但是，内存访问的开销是让我们不能容忍的。为使地址转换更加高效，处理器采用了一个专用硬件单元，称为快表或者转换后援缓冲器（Translation Look-aside Buffer，TLB）。TLB 缓存了最近访问的页表项，使得通过 TLB 查找页表项时的速度要远远快于一次普通内存访问[⊖]。有了 TLB，处理器的操作速度将不受地址转换开销的影响。

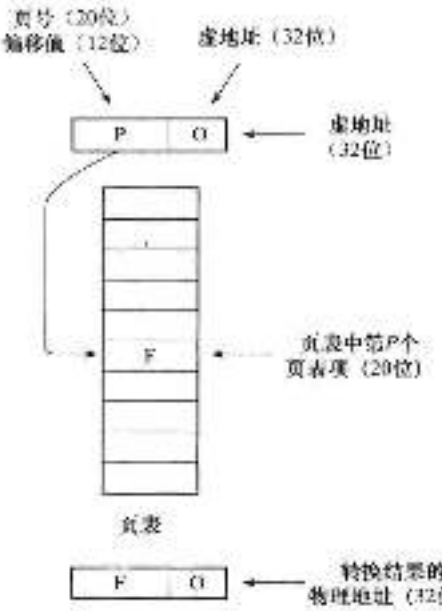


图 10-3　一个使用分页的虚地址转换的例子

10.12　页表项中的元数据

除了页框指针外，每个页表项包含了 3 位硬件和操作系统使用的元数据。图 10-4 列出了每 1 位的含义。

名字	含义
使用位	当页被引用（读取或存入）时，由硬件置 1
修改位	当存入页中的数据发生变化时，由硬件置 1
存在位	该位由操作系统设置，表示页是否在内存中

图 10-4　页表项中的 3 个元位及其含义

10.13　按需换页以及设计上的问题

按需换页（demand paging）是指这样一个系统，在这个系统中操作系统把所有进程的所有页面都放置在磁盘上，只有在需要某个页面时（即按需）将其读入内存。为了支持按需换页，需要特殊的处理器硬件：如果一个进程试图访问一个不在内存中的页面时，硬件必须暂停当前指令的执行，发出缺页（page fault）异常信号来通知操作系统。当缺页异常发生时，操作系统找到内存中尚未使用的页框，将进程需要的页面从磁盘读入，然后指示处理器从引起缺页异常的指令处恢复执行。

⊖　TLB 使用内容可寻址存储器（Content-Addressable Memory，CAM）来实现高速访问。

当计算机刚刚启动时，内存相对比较空闲，寻找空闲页框比较容易。然而，最终内存中的所有页框都会被填充，此时操作系统必须从中选出一个页框，将页面内容写回磁盘（如果页面已被修改），获取新的页面，并相应地修改页表。如何选择这个写回磁盘的页面，是操作系统设计者所面临的一个关键问题。

与分页设计相关的问题在于页面和进程之间的关系。当进程 X 产生一个缺页异常时，操作系统应该从进程 X 的页面中选择一个页面写回磁盘，还是从别的进程中选择？在从磁盘读取一个页面期间，
173 操作系统可以运行另一个进程，此时操作系统如何保证至少某个进程有足够多的页面来运行，而不会产生缺页异常[⊖]？应该锁定一些页面，让它们一直在内存中吗？如果需要，应该锁定哪些页面？页面选择策略如何与调度策略等其他策略交互？例如，操作系统是否应该保证每个高优先级进程的一个最小数目内存的页面数目？如果操作系统允许进程共享内存，应该对这些共享的页面应用什么策略？

在设计分页系统时有一个有趣的权衡。为了减小分页开销和进程感受到的延迟，可以在进程运行时让它独占最大数量的物理内存。然而，很多进程都是I/O密集型的，这意味着一个进程很可能会阻塞以等待I/O完成。当一个进程阻塞时，如果另一个进程处于就绪状态并且操作系统可以进行上下文切换，那么就能够极大限度地提高总体性能。也就是说，可以通过让很多进程处于就绪状态来增加CPU的使用率和总体吞吐量。因此，内存管理问题就产生了：应该给一个进程大量的内存页框，还是应该让很多进程分享内存？

10.14 页面替换和全局时钟算法

人们已经提出并尝试了很多种页面替换策略，包括：

- 最近最少使（LRU）。
- 最不频繁使（LFU）。
- 先进先出（FIFO）。

有趣的是，人们已经发现了一个可证明是最佳的替换策略，它称为贝莱迪最佳页面替换算法（Belady's optimal page replacement algorithm）。该策略选择一个在未来最长时间内不会被访问的页面进行替换。显然，要实现该算法是完全不现实的，因为操作系统无法预知页面在未来的使用情况。尽管如此，贝莱迪算法还是为研究人员提供了一个标杆，用以衡量替换策略的好坏。

就实际的系统而言，有一个算法已经成为页面替换算法事实上的标准，它称为全局时钟算法（global clock）或二次机会算法（second chance），该算法被设计为MULTICS操作系统的一部分，具有相对较小的开销。全局指的是所有进程相互竞争（换句话说，当进程 X 发生缺页异常时，操作系统可以从另一个进程 Y 选择替换页框）。该算法为一个名字源于全局时钟算法在回收一个页框之前，会给每个使用过的页框“第二次机会”。

全局时钟算法在发生缺页异常时执行。该算法维护一个指针，该指针对所有内存中的页框进行扫描，直到找到一个空闲的页框。算法下一次运行时，会从上一次结束位置的下一个页框继续。

为了确定是否选择某个页框，全局时钟算法检查页框的页表中的Use位和Modify位。如果Use/Modify位的值是（0，0），全局时钟算法就选择这个页框；如果是（1，0），全局时钟算法重设它们为
174 （0，0），并跳过这个页框；如果是（1，1），全局时钟将其修改为（1，0），跳过该页框，同时保存一份已修改位的副本，用来判断页面是否被修改过。在最坏情况下，全局时钟算法需要扫描所有的页框两次，才能回收一个页框。

实际上，大多数实现使用一个独立的进程来运行全局时钟算法（这样时钟进程可以进行磁盘I/O）。此外，全局时钟算法并不在找到一个页框后就立刻停止。相反，该算法继续运行，收集候选页面的集合。收集这个集合的目的在于使后续的缺页异常处理可以很快完成，避免频繁运行全局时钟算法的开销（即，避免频繁的上下文切换开销）。

⊖ 如果大存中的页框数不够，那么分页系统就会发生抖动现象，表示虚拟页异常发生的频率如此之高，以至于系统全部时间都花在换页上，每个步都需要花费很长的时间段来等待页面变得可用。

10.15　观点

虽然地址空间管理和虚拟内存子系统包含了操作系统大量的代码，但该问题最具智慧的方面在于分配策略的选择和随之而来的权衡。允许每个子系统分配任意数量的内存，最大限度地提高了灵活性，避免了系统在还有内存剩余的时候无法继续运行的问题。内存分区提供了最大程度的保护，避免一个子系统被另一个子系统抢占的问题。因此，内存管理存在灵活性和保护之间的权衡问题。

尽管经历了多年的研究，人们却没有提出一个适用的解决方案，权衡没有量化，也不存在适用的设计准则。类似地，尽管虚拟内存系统经过了多年的研究，但仍没有一个按需换页系统能在小内存上很好地运行。幸运的是，经济和科技的发展使得很多与内存管理相关的问题变得无关紧要；动态随机存储器（DRAM）芯片密度高速增长，使得大内存变得相当便宜。因此，计算机制造商为新产品配备了远多于过去产品的内存，使得操作系统不再需要使用按需换页，从而完全避免了内存管理的问题。

10.16　总结

低层内存分配机制把所有空闲内存看做一个单一的、可耗尽的资源。高层内存管理机制允许将内存分为独立的区域，保证单个子系统不会用尽所有的可用内存。

Xinu 的高层内存管理函数使用缓冲池范式，在这个范式中，给每个缓冲池分配一组固定的缓冲区。一旦建立缓冲池，一组进程便可以动态地分配和释放缓冲区。缓冲池接口只支持同步访问：一个进程将阻塞，直到缓冲池可用。 175

大型操作系统使用虚拟内存机制来为应用程序进程分配独立的地址空间。使用最广泛的虚拟内存机制——分页，将地址空间分为固定大小的页，并且按需加载页面。分页需要硬件的支持，因为每个内存引用都需要经过从虚拟地址到相应物理地址的映射。

练习

10.1　设计一个新的 getmem 函数，使其包含 getbuf。提示：允许用户从之前分配的内存块中进行子分配。

10.2　函数 mkbufpool 生成一个缓冲池中所有缓冲区的链表。试解释如何修改其代码，使该函数只分配内存，并仅在调用 getbuf 需要分配新缓冲区时才把缓冲区链接起来。

10.3　函数 freebuf 比 freemem 更高效吗？请解释你的答案。

10.4　调整缓冲池分配机制，使得能够对缓冲池进行回收。

10.5　缓冲池的现有实现在缓冲区前的内存中隐含了缓冲池 ID。重写函数 freebuf 使得不再需要这个缓冲池 ID。请保证你的 freebuf 能够检测无效的地址（换句话说，除非缓冲区是之前从该缓冲池分配的，否则不会将缓冲区释放回缓冲池）。

10.6　假定某个处理器支持分页。请描述能实现保护一个进程的栈不被其他进程访问的分页硬件，即使并未实现按需换页（即所有分页都驻留在内存中，不进行页面替换）。

10.7　实现 10.6 题所设计的方案，保护 Xinu 的栈。 176

第11章

Operating System Design: The Xinu Approach, Linksys Version

高层消息传递

消息总是使我不安。

Neil Tennant

11.1 引言

第8章描述了一个底层消息传递机制，允许一个进程直接向另一个进程传递消息。尽管底层消息传递系统提供了一个有用的功能，但它并不能指定多个接收方，也不能让一个进程参与而不干扰多个消息交换的过程。

本章通过引入一个高层消息传递机制来完成关于消息传递的讨论。该机制提供一个缓冲消息交换的同步接口，允许任意一个进程的子集传递消息，而不影响其他进程。本章还引入了独立于进程存在的命名会合点的概念。该机制的实现依赖于第10章所讲述的缓冲池机制。

11.2 进程间通信端口

Xinu使用进程间通信端口（inter-process communication port）来指定会合点，即进程可以进行消息
[179] 交换的地方。通过端口进行消息交换不同于第8章讲述的进程到进程的消息传递，因为端口允许暂存多个待发消息，并且访问它们的进程会被阻塞，直到请求可以满足。每个端口配置为可以暂存一定数量的消息，每个消息占用一个32位的字。当一个进程产生一条消息时，可以使用函数ptsend将消息发送到某个端口，消息以先进先出（FIFO）的顺序存储在端口中，当消息发送后，该进程便可继续运行。在任何时候，进程都可以使用函数ptrecv从某个端口接收下一条消息。

消息发送和接收是同步的。只要端口还有剩余空间，发送方就可以无延迟地暂存一条消息。然而，当端口存满消息时，每个发送消息的进程都将阻塞，直到有消息移除从而腾出空间。类似地，如果一个进程试图从一个空端口接收消息，它将被阻塞，直到有消息到达。进程的请求也是以先到先得的方式进行。例如，如果多个进程同时等待一个空端口，那么在消息到达时等待时间最长的进程将接收到该消息。类似地，如果多个进程试图发送消息时被阻塞，那么当端口空间可用时，等待时间最长的进程将被允许继续发送消息。

11.3 端口实现

每个端口由一个暂存消息的队列和两个信号量组成。其中一个信号量管理生产者，阻塞任何试图往被端口发送消息的进程；另一个信号量管理消费者，阻塞任何试图从空端口接收消息的进程。

由于端口可以动态创建，所以任何时候在所有端口上等待的消息总数是不确定的。虽然每条消息都很小（仅有一个字长），但必须限制所有端口消息队列总共需要的空间，以免端口函数用尽全部内存。为了提供限制总共使用的空间，端口函数分配固定数量的结点，用来存放待发消息，并让所有端口共享这些结点。开始，这些消息结点链接到由变量pfree指定的空闲链表；函数psend从空闲链表中取出一个结点，将消息存入其中，并将其添加到消息发往端口的队列中；函数ptrecv从指定的端口获取下一条消息，将包含该消息的结点释放回空闲链表中，然后将消息传递给函数调用者。

在文件ports.h中，结构ptnode定义了一个包含一条消息的结点的内容。结构ptnode中的两个字段是意料之中的：ptmsg保存32位的消息，ptnext指向下一个消息结点。

结构ptentry定义了端口表项的内容。ptssem和ptrsem字段分别包含了控制发送和接收的信号量ID。ptstate字段指示该表项是否正在使用，ptmaxcnt字段指定了任何时候允许暂存在该端口中的最大消息数目。pthead和pttail字段分别指向消息链表的第一个和最后一个结点。对于序列号字段ptseq，我们将
[180] 稍后讨论。

```
/* ports.h - isbadport */

#define NPORTS          20              /* maximum number of ports      */
#define PT_MSGS         100             /* total messages in system     */
#define PT_FREE         1               /* port is free                 */
#define PT_LIMBO        2               /* port is being deleted/reset  */
#define PT_ALLOC        3               /* port is allocated            */

struct  ptnode  {                       /* node on list of messages     */
        uint32  ptmsg;                  /* a one-word message           */
        struct  ptnode  *ptnext;        /* ptr to next node on list     */
};

struct  ptentry {                       /* entry in the port table      */
        sid32   ptssem;                 /* sender semaphore             */
        sid32   ptrsem;                 /* receiver semaphore           */
        uint16  ptstate;                /* port state (FREE/LIMBO/ALLOC)*/
        uint16  ptmaxcnt;               /* max messages to be queued    */
        int32   ptseq;                  /* sequence changed at creation */
        struct  ptnode  *pthead;        /* list of message pointers     */
        struct  ptnode  *pttail;        /* tail of message list         */
};

extern  struct  ptnode  *ptfree;        /* list of free nodes           */
extern  struct  ptentry porttab[];      /* port table                   */
extern  int32   ptnextid;               /* next port ID to try when     */
                                        /*   looking for a free slot    */

#define isbadport(portid)       ( (portid)<0 || (portid)>=NPORTS )
```

11.4　端口表初始化

由于之前初始化代码是在实现基本操作之后才设计的，所以我们总是在讨论其他函数之后才讨论初始化函数。然而，对于端口而言，我们要先讨论初始化，因为这样有助于理解其他函数。文件 ptinit.c 包含了初始化端口的代码和端口表的声明。全局变量 ptnextid 是数组 porttab 的一个索引，给出了在需要新端口时进行搜索的起始位置。初始化代码由以下三个步骤组成：标记所有端口是空闲的；构造空闲结点的链表；初始化 ptnextid。为了创建空闲链表，ptinit 使用函数 getmem 分配一块内存，然后遍历该内存，将单独的消息结点链接起来，组成空闲链表。 [181]

```
/* ptinit.c - ptinit */

#include <xinu.h>

struct  ptnode  *ptfree;                /* list of free message nodes   */
struct  ptentry porttab[NPORTS];        /* port table                   */
int32   ptnextid;                       /* next table entry to try      */

/*------------------------------------------------------------------------
 *  ptinit  -  initialize all ports
 *------------------------------------------------------------------------
 */
syscall ptinit(
          int32         maxmsgs         /* total messages in all ports  */
        )
{
        int32   i;                      /* runs through port table      */
        struct  ptnode  *next, *prev;   /* used to build free list      */
```

```
        ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
        if (ptfree == (struct ptnode *)SYSERR) {
                panic("pinit - insufficient memory");
        }

        /* Initialize all port table entries to free */

        for (i=0 ; i<NPORTS ; i++) {
                porttab[i].ptstate = PT_FREE;
                porttab[i].ptseq = 0;
        }
        ptnextid = 0;

        /* Create free list of message pointer nodes */

        for ( prev=next=ptfree ;  --maxmsgs > 0  ; prev=next )
                prev->ptnext = ++next;
        prev->ptnext = NULL;
        return(OK);
182 }
```

11.5 端口创建

端口创建过程就是分配端口表中空闲项的过程。函数 ptcreate 返回一个端口标识号（port Identifier, port ID）给函数调用者。函数 ptcreate 用一个参数来指定端口允许暂存的最大消息数目。因此，在一个端口创建后，调用它的进程可以确定在该端口上引起发送方阻塞之前能够暂存的消息数目。

```
/* ptcreate.c - ptcreate */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptcreate  --  create a port that allows 'count' outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
          int32         count
        )
{
        intmask mask;                   /* saved interrupt mask         */
        int32   i;                      /* counts all possible ports    */
        int32   ptnum;                  /* candidate port number to try */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if (count < 1) {
                restore(mask);
                return(SYSERR);
        }

        for (i=0 ; i<NPORTS ; i++) {    /* count all table entries      */
                ptnum = ptnextid;       /* get an entry to check        */
                if (++ptnextid >= NPORTS) {
                        ptnextid = 0;   /* reset for next iteration     */
                }

                /* Check table entry that corresponds to ID ptnum */
```

```
ptptr= &ports.tab[ptnum];
if (ptptr->ptstate == PT_FREE) {
        ptptr->ptstate = PT_ALLOC;
        ptptr->ptseen = screate(count);
                        ptptr->ptssem = screate(0);
                        ptptr->pthead = ptptr->pttail = NULL;
                        ptptr->ptseq++;
                        ptptr->ptmaxcnt = count;
                        restore(mask);
                        return(ptnum);
                }
        }
        restore(mask);
        return(SYSERR);
}
```

11.6　向端口发送消息

发送和接收消息是端口最基本的操作，它们分别由函数 ptsend 和 ptrecv 实现。这两个函数都需要调用者通过传递端口 ID 作为参数来指定要进行操作的端口。函数 ptsend 为等待在端口上的进程添加一条消息，它等待端口有空闲空间，将参数指定的消息入队列，给接收者发出信号量以指示另一条消息可用，然后返回。该函数的代码在文件 ptsend.c 中。

```
/* ptsend.c - ptsend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptsend  --  send a message to a port by adding it to the queue
 *------------------------------------------------------------------------
 */
syscall ptsend(
          int32         portid,         /* ID of port to use            */
          umsg32        msg             /* message to send              */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to table entry       */
        int32   seq;                    /* local copy of sequence num.  */
        struct  ptnode  *msgnode;       /* allocated message node       */
        struct  ptnode  *tailnode;      /* last node in port or NULL    */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return SYSERR;
}

/* Wait for space and verify port has not been reset */

seq = ptptr->ptseq;             /* record original sequence     */
if (wait(ptptr->ptssem) == SYSERR
    || ptptr->ptstate != PT_ALLOC
    || ptptr->ptseq != seq) {
        restore(mask);
        return SYSERR;
}
```

```
        if (ptfree == NULL) {
                panic("Port system ran out of message nodes");
        }

        /* Choose node from free list by unlinking */

        msgnode = ptfree;                   /* point to first free node */
        ptfree  = msgnode->ptnext;          /* unlink from the free list */
        msgnode->ptnext = NULL;             /* set fields in the node   */
        msgnode->ptmsg  = msg;

        /* Link into queue for the specified port */

        tailnode = ptptr->pttail;
        if (tailnode == NULL) {             /* queue for port was empty */
                ptptr->pttail = ptptr->pthead = msgnode;
        else {                              /* insert new node at tail  */
                tailnode->ptnext = msgnode;
                ptptr->pttail = msgnode;
        }
        signal(ptptr->ptrsem);
        restore(mask);
        return OK;
}
```

183 · 185

函数 ptsend 最开始的代码几乎没检查参数 portid 是否指定了一个有效的端口 ID。接下来发生的事情更有趣。函数 ptsend 保存一份序列号 ptseq 的临时副本，然后处理这个请求。它在发送方信号量上等待，然后确认端口仍然是被分配状态，并以序列号和保存的临时副本匹配。该函数此种确认端口 ID 的行为看上去有些古怪，然而，如果 ptsend 执行时，端口已满，则调用它的进程会被阻塞。而且，在进程阻塞等待发送消息期间，端口可能被删除（甚至重新创建）。为了理解序列号的作用，可以回想函数 ptcreate 的行为：它在创建端口时增加这个序列号。这里的核心思想是让等待的进程确认对函数 wait 的调用不是因为端口被删除而结束的。如果是因为端口被删除，那么这个端口要么保持未使用的状态，要么序列号已经增加了。因此，在调用 wait 之后的代码需要确认原来的端口仍然保持在已分配状态。

函数 ptsend 将消息以先进先出的顺序入队。队列非空时，该函数依赖 pttail 指向队列的最后一个结点。而且，ptsend 总是将 pttail 指向添加到该链表的新结点。最后，在将新消息添加到队列后，ptsend 给接收者发送信号量，以便接收者能够收到该消息。

与前面的代码一样，下面的不变式能帮助程序员理解它的实现：

> 当 n 条消息在端口中等待时，信号量 ptrsem 有非负的计数 n；当 n 个进程在等待消息时，信号量 ptrsem 有负的计数 $-n$。

对函数 panic 的调用更值得一说，因为它首次出现在代码中。在我们的设计中，消息结点耗尽是一个灾难性的错误，系统不能从中恢复。这表示在消息结点上为了防止端口用尽所有空闲内存而发置某个限制，这种做法并不能完全解决问题的。也许使用端口的程序没能正确运行，也许用户并没有出错，但系统却不能处理一个合理的请求。我们没有办法知道是哪种情况。在这种情况下，相对于尝试继续执行，报告出错并停止执行通常是比较好的做法。函数 panic 用来应对这种的情况，它输出指定的错误消息，然后停止处理。如果用户选择继续执行，对 panic 的调用可以返回，但更多情况下用户会重启系统或者更换程序。（本章练习中提出了处理这个问题的替代方案。）

11.7 从端口接收消息

函数 ptrecv 实现了一个基本的消费者操作：从指定端口移除一条消息，并将它返回给调用者。函数代码在 ptrecv.c 中。

```
/* precv.c  - ptrecv */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptrecv  --  receive a message from a port, blocking if port empty
 *------------------------------------------------------------------------
 */
uint32  ptrecv(
          int32         portid          /* ID of port to use            */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to table entry       */
        int32   seq;                    /* local copy of sequence num.  */
        umsg32  msg;                    /* message to return            */
        struct  ptnode  *msgnode;       /* first node on message list   */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Wait for message and verify that the port is still allocated */

        seq = ptptr->ptseq;             /* record original sequence     */
        if (ptptr->ptstate != PT_ALLOC  | wait(ptptr->ptrsem) == SYSERR
            || ptptr->ptseq != seq ) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Dequeue first message that is waiting in the port */

        msgnode = ptptr->pthead;
        msg = msgnode->ptmsg;
        if (ptptr->pthead == ptptr->pttail)     /* delete last item     */
                ptptr->pthead = ptptr->pttail = NULL;
        else
                ptptr->pthead = msgnode->ptnext;
        msgnode->ptnext = ptfree;               /* return to free list  */
        ptfree = msgnode;
        signal(ptptr->ptssem);
        restore(mask);
        return msg;
}
```

函数 ptrecv 首先检查参数，等待消息可用，确认端口没有被删除或重置，然后将消息结点出队。用局部变量 msg 记录要返回的消息，并将消息结点返回到空闲链表中，最后将消息返回给调用者。

11.8　端口的删除和重置

系统有时需要删除或重置一个端口。在这两种情况下，系统必须处理等待的消息，将消息结点返回给空闲链表，允许等待的进程继续执行。但端口系统怎么处理等待的消息？它可以选择将它们丢掉，或将它们返回给发送这些消息的进程。通常，应当允许用户指明处理的方式。函数 pdelete 和 ptreset 执行端口删除和重置操作。代码描述在文件 pdelete.c 和 ptreset.c 中。

```
/* ptdelete.c - ptdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptdelete  --  delete a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */
syscall ptdelete(
          int32         portid,         /* ID of port to delete         */
          int32         (*dispose)()    /* function to call to dispose  */
        )                               /*   of waiting messages        */
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return(SYSERR);
        }
        _ptclear(ptptr, PT_FREE, dispose);
        ptnext(0) = portid;
        restore(mask);
        return(OK);
}
/* ptreset.c - ptreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptreset  --  reset a port, freeing waiting processes and messages and
                 leaving the port ready for further use
 *------------------------------------------------------------------------
 */
syscall ptreset(
          int32         portid,         /* ID of port to reset          */
          int32         (*dispose)()    /* function to call to dispose  */
        )                               /*   of waiting messages        */
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return SYSERR;
        }
        _ptclear(ptptr, PT_ALLOC, dispose);
        restore(mask);
        return OK;
}
```

函数 ptdelete 和 preset 都验证它们的参数是否正确，然后调用_ptclear 函数来执行清除消息和等待进程的操作[⊖]。在执行清除消息的操作时，_ptclear 将端口号的状态置为“limbo”（PT_LIMBO），limbo

⊖ _ptclear 以下划线开头说明这个函数是系统内部函数，不会被用户直接调用。

状态可以保证其他的进程不能使用该端口——ptsend 和 ptrecv 将拒绝对端口进行操作。ptcreate 函数也拒绝分配该端口。

在声明一个端口可以再使用前，ptclear 会重复地调用 dispose，并将其传送给每个等待的消息。最后，当所有的消息都移除后，_ptclear 删除或用它的第二个参数指定的值重置信号量。在处理消息前，_ptclear 将端口的序号加 1，这样当等待的进程被唤醒后可以辨别端口已经发生了变化。这里的代码实现在 ptclear.c 文件中。

188 / 189

```
/* ptclear.c - _ptclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  _ptclear  --  used by ptdelete and ptreset to clear or reset a port
 *                  (internal function assumes interrupts disabled
 *                   and arguments have been checked for validity)
 *------------------------------------------------------------------------
 */
void    _ptclear(
          struct ptentry *ptptr,        /* table entry to clear         */
          uint16        newstate,       /* new state for port           */
          int32         (*dispose)(int32)/* disposal function to call   */
        )
{
        struct  ptnode  *walk;          /* pointer to walk message list */

        /* Place port in limbo state while waiting processes are freed */

        ptptr->ptstate = PT_LIMBO;

        ptptr->ptseq++;                 /* reset accession number       */
        walk = ptptr->pthead;           /* first item on msg list       */

        if ( walk != NULL ) {           /* if message list nonempty     */

                /* Walk message list and dispose of each message */

                for( ; walk!=NULL ; walk=walk->ptnext) {
                        (*dispose)( walk->ptmsg );
                }

                /* Link entire message list into the free list */

                (ptptr->pttail)->ptnext = ptfree;
                ptfree = ptptr->pthead;
        }

        if (newstate == PT_ALLOC) {
                ptptr->pttail = ptptr->pthead = NULL;
                sreset(ptptr->ptsasem, ptptr->ptmaxcnt);
                sreset(ptptr->ptrsem, 0);
        } else {
                sdelete(ptptr->ptsasem);
                sdelete(ptptr->ptrsem);
        }
        ptptr->ptstate = newstate;
        return;
}
```

11.9 观点

因为提供同步的接口，所以端口机制允许进程等待下一个消息的到来。同步接口可以很强大——聪明的程序员可以利用该机制来协调进程（例如，实现进程间的互斥）。有趣的是，在进程协调的同时也可能导致一个潜在的问题：死锁。也就是说，可能会有一系列端口被阻塞，所有的进程都在等待消息，没有进程发送消息。因此，当程序员使用端口时，应该小心确保不让死锁发生。

11.10 总结

本章介绍了一个高层消息传递机制，称为通信端口，该机制允许进程间通过在指定的端口交换信息。每个端口包含一个固定长度的消息队列。函数 ptsend 处理在队列末尾的消息，而 ptrecv 获取队列头部的消息。当进程试图从一个空端口接收消息时，该进程会被阻塞，直到有消息到达。当进程试图往一个队列已满的端口发送消息时，该进程会被阻塞，直到队列中有空闲的空间。

练习

11.1 考虑函数 send、receive 和 ptsend、ptrecv，是否可以设计一个简单的包含这两组函数的消息传递方案？解释应如何设计？

11.2 静态分配和动态分配资源间存在很大的区别。例如，尽管进程间的消息槽是静态分配的，但端口是动态分配的。在多进程的环境中，动态分配的关键问题是什么？

11.3 改变消息结点的分配方案，使一个信号量可以控制空闲字表中的结点。如果没有空闲结点，让 ptsend 等待一个空闲的结点。新引入的方案，如果存在潜在问题，请问潜在的问题是什么？

190 ⋮ 191

11.4 悉选用于内部不一致或潜在死锁的情况下，通常引起恐慌的条件是不能复制的，所以难以精确地找到原因。讨论在 ptsend 中，你可以采取何种措施来跟踪引起恐慌的原因？

11.5 因为在 ptsend 中对恐慌的调用是可选的，所以考虑分配更多的结点或重试该操作。每个操作的弊端是什么？

11.6 重写 ptsend 和 ptrecv，当它们等待的端口被删除时返回一个特殊的值。这个新机制的主要不利方面是什么？

11.7 修改前面章节中分配、使用和删除对象的程序，使得当通信端口函数执行删除操作时可以通过使用一个序列号来协调到。

11.8 ptsend 和 ptrecv 不能传递带有 SYSERR 值的消息，因为 ptrecv 不能区别是带有该值的消息和还是一个错误返回。重新设计该函数以便能传递任何值。

192

中断处理

音乐的乐趣从来不应该被广告所打断。

——Leonard Bernstein

12.1 引言

前面的章节主要讲述了处理器和内存管理。在讲述处理器管理的章节中，主要介绍了并发处理的概念，说明了进程是怎样创建和终止的，以及进程间是如何协调工作的。在内存管理的章节中描述了使用低级机制来管理动态分配和栈、堆存储器的释放。

本章首先讨论输入/输出（I/O）设备。本章回顾中断的概念，介绍操作系统用来处理中断的整体软件架构。本章将描述一个中断分配机制，当中断发生时，该机制将控制权传递给合适的中断处理程序。更重要的是，本章解释了中断间的复杂关系和并发进程的操作系统抽象，给出了当中断发生时，中断代码必须遵循的总的指导方针以便提供正确安全的并发进程实现。在后续章节中，我们将继续讨论如何对特定的设备进行中断处理，其中包括实时时钟设备。 195

12.2 中断的优点

中断机制主要用于第三代计算机系统，可以高有方地将输入/输出活动和计算处理分开。如果没有中断设备，操作系统提供的很多服务将不可能实现。

中断机制的动机就是并发。除了依赖CPU来完全控制输入/输出外，每一个设备都包含一个可以独立操作的硬件。只需要CPU启动或停止一个设备。一旦启动，设备将继续传送数据，而不需要进一步帮助。因为大多数的输入/输出进程比计算操作慢很多，所以CPU可以启动多个设备，允许它们并行进行。启动输入/输出后，CPU可以继续其他的计算（如执行其他的进程）直到设备终端表明处理已经完成。这里关键的思想是：

> 中断机制允许处理器和输入/输出设备并行。尽管实现细节可能有差别，但是硬件一般包括可以自动中断正常处理的机制，和当一个设备完成操作或需要CPU处理时通知操作系统的机制。

12.3 中断分配

当中断发生时，处理器中的硬件执行三个基本的步骤：

- 当中断正在处理时，立刻改变处理器的状态以阻止其他的中断发生。
- 当中断处理完后，保存足够的状态以允许处理器继续执行。
- 分配出预先定义好的内存单元，操作系统可以将处理中断的代码放置在这些单元中。

每个处理器都包含能够使中断处理变得复杂的细节。例如，当硬件保存状态时，大多数系统中的硬件并没有保存所有处理器寄存器的完整的状态信息。相反，这些硬件一般只记录了一些基本的值，比如指令指针[⊖]的副本，因此操作系统需要保存所有在中断处理期间用到的寄存器信息。当中断处理结束后，操作系统还要在正常处理前先恢复保存的所有值。 196

12.4 中断向量

当中断发生时，操作系统必须能识别哪个设备发出了中断请求。为了进行设备识别，人们提出了

⊖ 指令指针包含了下一个执行的指令的地址，在某些架构上使用术语程序计数器。

很多硬件机制。例如，在一些硬件上，为了识别设备，操作系统必须实用总线来询问设备以进行识别。在另一些设备上，CPU 中的硬件可以自动处理任务。接下来我们讨论一个例子。

一个设备怎样识别自己呢？最常用的机制是使用中断号。给每个设备分配一个唯一的整数（0，1，2等）。这个整数称为中断向量号或中断请求号（IRQ）。当中断发生时，设备说明自己的中断向量号。硬件或操作系统使用中断号作为索引访问内存中的中断向量表。操作系统通过指针来访问中断向量表的每个单元，该指针指向一个处理该中断的函数。因此，如果一个中断向量号为i的设备发出了中断，则控制分支去：

```
interrupt_vector[i]
```

图 12-1 说明了内存中的中断向量组。

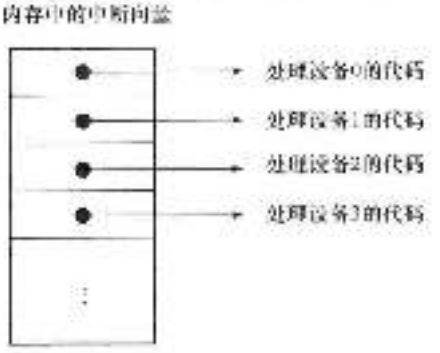


图 12-1 内存中的中断向量，每个表项包含一个指针，指向一种设备处理代码

12.5 中断向量号的分配

在计算机体系结构中，中断向量分配的细节差别很大。早期的系统要求在每个设备插入计算机前先手动给每个设备分配一个唯一的中断值（例如，在电路板上使用开关或跳线）。有些系统会给每个
197 设备分配两个中断向量号：一个用于设备完成输入操作，另一个用于设备完成输出操作。手动分配存在的问题是，这样的工作比较乏味且易于出错。如果计算机的使用者意外地将同一个中断向量号分给了两个不同的设备，那么设备将不能正确地执行。后来的系统采用了一种不太容易出错的方法：使用总接子（socket），每个设备可以通过其接子加入到系统中，这样中断向量号和每个插接字相关联而不是和设备相关联。无论分配是如何完成的，和设备相关联的中断处理必须和操作系统协调，因为操作系统初始化了内存中的中断向量。

在现代的系统中，中断向量的分配更加动态化。例如，有些设备允许中断号是可以协商的。当操作系统启动时，它使用总线来决定目前有哪些输入/输出设备。系统在可用的设备集中进行迭代，然后为每个设备选择一个唯一的中断向量号。系统使用总线来运算每个设备它的中断向量号，初始化内存中的中断向量以指向正确的中断处理程序。

最后一种方法允许设备在操作系统运行的过程中加入到系统中来。例如，USB 设备。操作系统给USB 控制器分配一个唯一的硬件中断，然后给控制器分配一个设备驱动器。驱动器可以动态地给每个设备加载额外的驱动器代码。因此，当一个新的设备加入时，控制器为设备加载驱动器并记录驱动器的位置信息。之后，当设备发生中断时，控制器接收中断，然后将中断转发给合适的驱动器代码。

12.6 硬件中断

中断向量位于内存中的什么位置呢？硬件负责处理中断的哪一部分呢？选择可以有很多，具体实现细节依赖于计算机系统的实现。许多大型计算机系统允许操作系统选择中断向量数组的位置。当操作系统启动时，首先选择位置信息，然后，通过将地址信息存储在内部硬件寄存器来通知硬件该位置信息。

在大型系统中，硬件可以在不使用 CPU 的情况下处理许多细节。例如，硬件可以使用总线来询问中断设备的中断向量号。使用向量号作为索引，然后直接将设备交给中断处理代码。然而，在小型系统中，处理器系统必须使用总线来决定中断向量号——处理器发出请求，中断设备则返回它的唯一中断向量号。

小型嵌入式系统经常采用一个简化的中断机制：每个中断单元是硬连接的，要么通过嵌入一个处理器芯片，要么通过主板。当中断发生时，操作系统必须决定是哪个设备发生了中断，然后使用中断
198 向量寻转入特定设备的中断函数。像第3章所描述的那样：硬件使用硬连接方法；当设备发生中断时，处理器转向 0x800000180 单元。幸运的是，示例系统使用了一个可以处理许多细节的协同处理器。当

中断发生时，协同处理器使用总线来识别设备。通过在 CAUSE 寄存器中放置一个值来识别中断。因此，当处理器处理中断时，处理器不需要直接与硬件或设备进行交互。处理器只需要从协同处理器的 CAUSE 寄存器中加载值就可以了，然后使用该值来判断发出中断请求的设备。

12.7 中断请求的局限性和中断多路复用

许多处理器对分配的唯一中断向量号有所限制（典型的上限是 8）。如果中断系统对向量的大小有限制，系统怎样给设备分配一个任意的中断向量号呢？答案取决于一种中断多路复用的技术：一个中断号可以分配给多个设备。当中断发生时，分配器必须确定分配相同中断号的哪个设备需要服务。

中断多路复用最适合于多个设备使用了一个硬件接口的情况。例如，USB 设备。从计算机的角度来看，USB 集线器就好像是连在计算机总线上的设备。事实上，USB 集线器只提供了一个额外的总线接口，使用该接口可以连接多个设备。无论什么时候当 USB 设备需要使用服务的时候，设备可以使用与 USB 关联的中断号来发送中断请求，然后操作系统就将控制权转给 USB 处理程序。USB 处理程序可以判断是哪个设备需要服务，然后将控制权转发给处理该设备的代码。

Linksys 硬件就是多重中断的一个例子。一个 MIPS 处理器只允许分配 8 个唯一的中断号。其他的保留用于特殊的目的。为了满足约束，E2100L 硬件将系统主板的所有硬件多路复用到一个中断向量上——硬件中断 4。当它收到中断 4 时，硬件轮询系统主板发出询问以判断是哪个设备引起了中断。例如，如果一个串行设备引起了中断，硬件将会识别中断是从主板的设备 3 发出的。

12.8 中断软件和分配

当操作系统识别了引起中断的设备后，它就调用处理该设备中断的函数。我们说操作系统将中断分配给相应的中断处理程序，我们使用分配器来代表执行分配函数的操作系统代码。即使计算机中的硬件可以自动接受中断向量，但许多操作系统用中断分配器的地址来填充所有的中断向量，这样分配器可以实现调度常量和其他的全局系统策略。图 12-2 展示了中断过程中分配器和处理程序集合的概念结构。 [199]

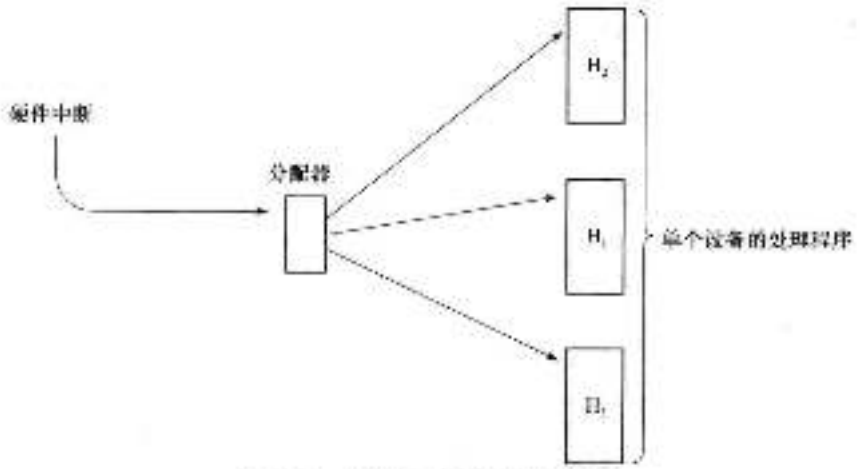


图 12-2 中断处理软件的概念结构

需要注意的是有些细节可能导致更加复杂的结构。例如，因为 MIPS 中断分配器需要询问协处理器的硬件寄存器，并使用特殊的指令从中断中返回，有些 MIPS 中断分配器必须用汇编语言来写。在这种情况下，扩展汇编语言代码以囊括所有的中断处理是合理的。然而，中断代码是操作系统的重要部分，而汇编语言代码难以理解和修改。因此，为了保持系统代码的可读性，大多数操作系统设计人员将分配器分为两个部分：系统将硬件中断分为更小的部分，用汇编语言来写的低层的代码。低层的代码处理保存和恢复寄存器等指令，与协处理器进行通信以识别中断设备，一旦中断完成后使用特殊的指令来从中断中返回。低层的代码很少——一旦寄存器保存完后，低层代码就调用一个用 C 语言编写的高层分配器函数。高层分配器检测中断向量数组或使用其他操作系统的数据结构来选择中断设备的处理

程序。一旦处理程序的地址计算完毕，高层中断分配器就调用该处理程序。尽管分为了两部分，但分配器的代码量还是很小的——所有与给定设备进行通信的代码都放在中断处理程序中，而不是分配器中。

200

我们的示例系统还实现了一个额外的细节。为了理解系统的结构，示例里的中断硬件总是指向0x80000180单元，操作系统位于0x80001000单元。因此，当操作系统启动时，它将代码放置在0x80001000单元。系统在指定位置存储了一个跳转指令，当中断发生时该指令可以使处理器跳转到低层中断分配器（代码中的intdispatch），这里并没有将整个低层中断处理程序的代码复制到指定的存储单元。图12-3显示了当操作系统完成初始化后的代码布局。

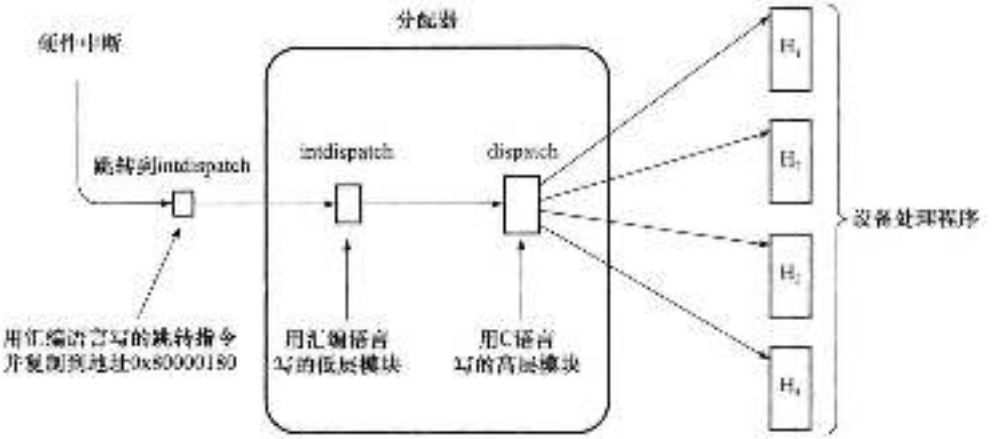


图12-3 示例系统中中断代码的组织结构

当intdispatch启动时，它在当前运行时栈中分配空间，保存每一个处理器寄存器，以便可以在中断返回前恢复。因为高层处理函数，dispatch，是用C语言写的，所以intdispatch必须用C语言来调用dispatch。因此，保存寄存器后，intdispatch从协处理器的CAUSE寄存器中抽取数值来判断中断源，然后将栈中的值作为dispatch的参数。intdispatch也将保存的栈页框的地址值作为第二个参数。

图12-3描述的四个中断处理程序分别和示例系统中的主要设备类型相匹配：

- 串行设备（控制台）。
- 有线网络设备（以太网）。
- 无线网络（Wi-Fi）。

201

- 实时时钟设备（定时器）。

12.9 中断分配器底层部分

示例代码可以解释清楚很多细节。文件intdispatch.S包含了中断分配器低层部分的代码。

```
/* intdispatch.S - intdispatch */

#include <mips.h>
#include <interrupt.h>

.text
        .align 4
        .globl intdispatch

/*------------------------------------------------------------------------
 * intdispatch - low-level piece of interrupt dispatcher
 *------------------------------------------------------------------------
 */
        .ent intdispatch
```

```
intdisptab:
        .set noreorder
        .set noat
        j       savestate               /* Jump to low-level handler    */
        nop
savestate:
        addiu   sp, sp, -IRQREC_SIZE    /* Allocate space on stack      */
        sw      AT, IRQREC_AT(sp)       /* Save assembler temp reg first*/
        mfc0    k0, CP0_CAUSE           /* Save interrupt CAUSE value   */
        mfc0    k1, CP0_EPC             /* Save interrupted PC value    */
        sw      k0, IRQREC_CAUSE(sp)
        mfc0    k0, CP0_STATUS          /* Save co-processor STATUS     */
        sw      k1, IRQREC_EPC(sp)
        sw      k0, IRQREC_STATUS(sp)
        .set at
        .set reorder
        sw      v0, IRQREC_V0(sp)       /* Save all general purpose regs*/
        sw      v1, IRQREC_V1(sp)
        sw      a0, IRQREC_A0(sp)
        sw      a1, IRQREC_A1(sp)
        sw      a2, IRQREC_A2(sp)
        sw      a3, IRQREC_A3(sp)
        sw      t0, IRQREC_T0(sp)
        sw      t1, IRQREC_T1(sp)
        sw      t2, IRQREC_T2(sp)
        sw      t3, IRQREC_T3(sp)
        sw      t4, IRQREC_T4(sp)
        sw      t5, IRQREC_T5(sp)
        sw      t6, IRQREC_T6(sp)
        sw      t7, IRQREC_T7(sp)
        sw      s0, IRQREC_S0(sp)
        sw      s1, IRQREC_S1(sp)
        sw      s2, IRQREC_S2(sp)
        sw      s3, IRQREC_S3(sp)
        sw      s4, IRQREC_S4(sp)
        sw      s5, IRQREC_S5(sp)
        sw      s6, IRQREC_S6(sp)
        sw      s7, IRQREC_S7(sp)
        sw      t8, IRQREC_T8(sp)
        sw      t9, IRQREC_T9(sp)
        sw      k0, IRQREC_K0(sp)
        sw      k1, IRQREC_K1(sp)
        sw      gp, IRQREC_S8(sp)
        sw      sp, IRQREC_SP(sp)
        sw      fp, IRQREC_S9(sp)
        sw      ra, IRQREC_RA(sp)
        sw      zero, IRQREC_ZER(sp)
        mfhi    t0                      /* Save hi and lo               */
        mflo    t1
        sw      t0, IRQREC_HI(sp)
        sw      t1, IRQREC_LO(sp)

        lw      a0, IRQREC_CAUSE(sp)    /* Pass cause and state into to */
        move    a1, sp                  /*  high-level dispatcher       */
        jal     dispatch

restorestate:                           /* Or return from dispatcher    */
        lw      t0, IRQREC_HI(sp)       /*  restore all state           */
        lw      t1, IRQREC_LO(sp)
        mthi    t0
```

```
nclo    t1
lw      ra, IRQREC_RA(sp)       /* Restore general purpose regs */
lw      fp, IRQREC_S8(sp)
lw      gp, IRQREC_S6(sp)
lw      t9, IRQREC_T9(sp)
lw      t8, IRQREC_T8(sp)
lw      s7, IRQREC_S7(sp)
lw      s6, IRQREC_S6(sp)
lw      s5, IRQREC_S5(sp)
lw      s4, IRQREC_S4(sp)
lw      s3, IRQREC_S3(sp)
lw      s2, IRQREC_S2(sp)
lw      s1, IRQREC_S1(sp)
lw      s0, IRQREC_S0(sp)
lw      t7, IRQREC_T7(sp)
lw      t6, IRQREC_T6(sp)
lw      t5, IRQREC_T5(sp)
lw      t4, IRQREC_T4(sp)
lw      t3, IRQREC_T3(sp)
lw      t2, IRQREC_T2(sp)
lw      t1, IRQREC_T1(sp)
lw      t0, IRQREC_T0(sp)
lw      a3, IRQREC_A3(sp)
lw      a2, IRQREC_A2(sp)
lw      a1, IRQREC_A1(sp)
lw      a0, IRQREC_A0(sp)
lw      v1, IRQREC_V1(sp)
lw      v0, IRQREC_V0(sp)

.set noreorder
.set noat
lw      k0, IRQREC_EPC(sp)      /* Restore interrupted PC value */
lw      AT, IRQREC_AT(sp)       /* Restore assembler temp reg   */
mtc0    k0, CP0_EPC
lw      k1, IRQREC_STATUS(sp)   /* Restore global status reg    */
addiu   sp, sp, IRQREC_SIZE     /* Restore stack pointer        */
mtc0    k1, CP0_STATUS
nop                             /* Delay for co-processor       */
eret                            /* Return from interrupt        */
nop
nop
.set at
.set reorder
.end intdispatch
```

12.10 中断分配器高层部分

中断分配器的低层部分保存了处理器寄存器的状态并通过询问协处理器确定了引起中断的设备。它调用 dispatch 函数，传递指定设备的参数和包含保存状态页主的指针。dispatch 使用 cause 参数确定合适的中断处理程序，并调用这个程序。一旦中断处理程序返回，dispatch 就将其返回给具有恢复处理器寄存器状态和从中断返回功能的中断分配器的低层部分。文件 dispatch.c 包含了该部分代码。

202 ~ 204

```
/* dispatch.c */

#include <xinu.h>
#include <ripe.h>
#include <sa9130.h>
```

```
/* Initialise list of interrupts */

char *interrupts[] = {
    "Software interrupt request 0",
    "Software interrupt request 1",
    "Hardware interrupt request 0, wmac",
    "Hardware interrupt request 1, usb",
    "Hardware interrupt request 2, ath0",
    "Hardware interrupt request 3, ath1",
    "Hardware interrupt request 4, misc",
    "Hardware interrupt request 5, timer",
    "Miscellaneous interrupt request 0, timer",
    "Miscellaneous interrupt request 1, error",
    "Miscellaneous interrupt request 2, gpio",
    "Miscellaneous interrupt request 3, uart",
    "Miscellaneous interrupt request 4, watchdog",
    "Miscellaneous interrupt request 5, perf",
    "Miscellaneous interrupt request 6, reserved",
    "Miscellaneous interrupt request 7, mbox",
};

/*-------------------------------------------------------------------
 * dispatch - high-level piece of interrupt dispatcher
 *-------------------------------------------------------------------
 */

void    dispatch(
          int32 cause,          /* identifies interrupt cause          */
          int32 *frame          /* pointer to interrupt frame that     */
                                /*  contains saved status              */
        )
{
        intmask mask;           /* saved interrupt status              */
        int32   irqcode = 0;    /* code for interrupt                  */
        int32   irqnum =  1;    /* interrupt number                    */
        void    (*handler)(void);/* address of handler function to call */

        if (cause & CAUSE_EXC) exception(cause, frame);
/* Obtain the IRQ code */

irqcode = (cause & CAUSE_IRQ) >> CAUSE_IRQ_SHIFT;

/* Calculate the interrupt number */

while (irqcode) {
        irqnum++;
        irqcode = irqcode >> 1;
}

if (IRQ_ATH_MISC == irqnum) {
        uint32 *miscStat = (uint32 *)RST_MISC_INTERRUPT_STATUS;
        irqcode = *miscStat & RST_MISC_IRQ_MASK;
        irqnum = 7;
        while (irqcode) {
                irqnum++;
                irqcode = irqcode >> 1;
        }
}
```

```
        /* Check for registered interrupt handler */

        if ((handler = interrupt_vector[irqnum]) == NULL) {
                kprintf("Xinu Interrupt %d uncaught, %s\r\n",
                        irqnum, interrupts[irqnum]);
                while (1) {
                        ;               /* forever */
                }
        }

        mask = disable();       /* Disable interrupts for duration */

        exlreset();             /* Reset system-wide exception bit */

        (*handler) ();          /* Invoke device-specific handler  */

        exlset();               /* Set system wide exception bit   */
        restore(mask);
}

/*------------------------------------------------------------------------
 * enable_irq - enable a specific IRQ
 *------------------------------------------------------------------------
 */
void enable_irq(
        irqmask irqnumber               /* specific IRQ to enable       */
        )
{
        int32   enable_cpuirq(int);
        int irqmisc;
        uint32 *miscMask = (uint32 *)RST_MISC_INTERRUPT_MASK;
        if (irqnumber >= 8) {
                irqmisc = irqnumber - 8;
                enable_cpuirq(IRQ_ATH_MISC);
                *miscMask |= (1 << irqmisc);
        } else {
                enable_cpuirq(irqnumber);
        }
}
```

函数enable_irq用以产生特殊的中断。注意硬件使用中断多路复用，允许主板上的设备发出硬件中断请求4。Xinu操作系统使用特殊的技术来存储主板设备发出的中断处理程序。尽管硬件只使用8个中断（分别对应于中断向量数组中0-7的单元），但我们的代码仍在内存中建立了较大的中断向量，为主板设备提供大于7的单元。也就是说，数组中单元8对应主板设备0，数组中9对应主板设备1，以此类推。但当设备发出中断请求时，硬件中断4标志着是主板设备发出的请求。系统查问主板设备并确定是主板设备3发出的请求。然后代码就给设备数字（3）加上8，从中断向量单元11处取出处理程序地址。尽管这种做法需要分配器再维护一张独立的主板设备表，但把信息放在单一的数据结构中能保持系统的统一，同时允许分配器中单独的一段代码使用任何处理程序，与是否使用中断复用独立开来。

205
|
207

中断复用允许设备共享一个中断向量。操作系统中的分配器能够隐藏分配过程，使用简单的机制来触发处理程序。

12.11 禁止中断

由于中断函数检测和改变全局数据结构（如I/O缓存），所以在执行中断时，操作系统必须阻止其他程序执行。如我们所见，当一个中断发生时，硬件禁止后续的中断，这说明中断处理不能被打断。

当中断分配器低层部分调用高层部分或者高层部分调用处理程序时，中断仍然处于禁止状态。当中断分配器高层部分返回时，控制权就回到了中断分配器的低层部分。低层部分存储有处理器的状态，并使用特殊的汇编语言结构返回到产生中断进程的确切位置。最后，低层中断分配器返回，中断开启。这个过程可以总结为：

> 在分配器调用高层中断处理程序前，中断禁止。高层中断处理程序在改变全局数据结构时保持中断禁止。

上述的中断策略会引起微妙的结果。当中断处于禁止状态时，硬件将执行对指令数量的限制。如果操作系统让中断禁止的时间任意长，那么设备的执行就会出现问题。例如，如果一个处理器在后续的字符到达前漏掉了一个字符，那么一个或者更多的字符将会丢失。因此，中断程序必须尽快地完成任务并执行开启中断的程序。更重要的是，中断是全局的——当一个设备的处理程序引起了中断后，所有的设备都会受到影响。因此，在写中断代码时，必须注意系统中所有设备的限制，同时给予设备最短，最合适的时间限制。

> 设备 D 的处理程序所能保持中断禁止的最长时间不是由检测设备 D 计算出来的。相反，该时间是通过选择所有设备的最短时间限制所得到的。

12.12 函数中中断代码引起的限制

为了满足中断代码给予设备最紧的时间限制，操作系统的设计者必须通过任意进程创建一个用以执行的中断代码。也就是说，当中断发生时，运行的进程都可以执行中断代码。 [208]

以下两个实际情况使得上述过程变得重要：

- 中断处理程序能调用操作系统函数。
- 由于调度器假设至少有一个进程保持就绪状态，所以空进程必须保持在当前或就绪状态。

空进程是一个不调用任何函数的无限循环。然而，中断被认为是发生在两个连续的指令之间。因此，当中断发生在空进程运行时，空进程仍然会在处理程序执行时运行。这里主要的结论是：

> 中断程序中只能调用操作系统中使进程保留在当前或就绪状态的函数。

因此，中断程序可以调用如 send 或 signal 这样的函数，但不能调用如 wait 这种使运行进程转为等待状态的函数。

12.13 中断过程中重新调度的必要性

考虑在中断过程中重新调度这个问题。为了了解重新调度的必要性，考虑以下情况：

- 调度不变原则始终在任何时候，最高优先权的合法进程将一定会执行。
- 当一个 I/O 操作完成时，一个高优先级的进程可能会变为合法状态以使用处理器。

例如，假设一个高优先级进程 P 试图从网络上读取数据包。即使这个进程的优先权很高，但 P 在等待数据包的时候也会阻塞。当 P 阻塞时，其他的进程，如 Q，会运行。而当数据包到达后，中断就会发生。如果中断处理程序仅仅是将 P 变为就绪状态就返回，则进程 Q 还是会继续运行。如果 Q 的优先权比 P 低，则调度不变原则将会打破。

考虑一个更加极端的例子，如果一个系统只有一个应用程序进程，这个进程等待 I/O 操作而阻塞。当中断发生时，空进程将运行。如果中断处理程序不能重新调度，则中断将返回给空进程，应用程序将永远不会执行。这里关键的思想就是： [209]

> 为了确保进程能在 I/O 操作结束时被唤醒，同时维护调度不变原则，当中断处理程序调到等待进程变为就绪状态的情况时，必须进行重新调度。

12.14 中断过程中的重新调度

中断策略和调度之间的交互导致了一个比较复杂的问题。我们说过在处理中断时，中断程序保持中断禁止。我们还说过，当一个进程变成就绪状态时，中断处理程序应该启动重新调度。然而，考虑重新调度时会发生的事情。假设被选定执行的进程执行时会使中断开启。一旦这个进程执行，它将开

启中断。这意味着中断处理程序似乎不应该进行重新调度，因为切换到一个能响应中断请求的进程可能会引起一系列连续的中断。我们必须确保只要全局数据结构是合法的，那么中断到何时重新调度也是安全的。

为了理解重新调度为什么是安全的，考虑从中断处理程序调用 resched 的一系列事件。假设在中断恢复的条件下，当中断发生时进程 U 在运行。低层中断分配器代码使用 U 的栈来保存状态，并在高层中断分配器执行且中断禁止时让进程 U 运行。当高层中断分配器调用中断处理程序时，中断仍然保持在禁止状态。假设在这个过程中，代码调用 resched，切换到另一个进程 T。如果 T 恰好能使中断恢复（例如，从一个系统调用中返回），则另一个中断就可能发生。那么，什么东西能够阻止那些未完成的中断不断累积直到栈溢出这样一个无限循环呢？假设每个进程都有自己的栈，当进程 U 被上下文切换停止时，在它的栈中会记录一个中断。新的中断会在使用 T 的栈时产生。在另一个中断在 U 的栈中累积之前，进程 U 必须完全争得到 CPU 的控制权同时恢复中断。然而，进程 U 调度器和上下文切换时中断是禁止的。当切换到进程 U 执行时，上下文切换代码将保存这个中断状态，U 仍然用中断禁止这一状态继续运行。

当 resched 返回到中断处理程序或者中断处理程序返回到分配器时，中断会保持在禁止状态。中断仅在分配器返回到中断产生的位置时才开启。所以当进程 U 执行中断指令时，其他中断不会发生（即使有也只会处进程 U 切换到别的进程，在另外的进程中发生）。也就是说，在任何时候，一个给定的进程只会有一个中断请求被响应。由于系统在一个给定的时间内有有限的进程存在，每个进程至多有
210 一个未完成的中断，所以未完成的中断数目是有限的。以上主要观点可以概括为：

重新调度是安全的，只要：1）中断程序让全局数据在重新调度前保持在合法的状态；

2）除了使用中断禁止外，没有函数能使中断恢复。

这个规则解释了为什么所有的操作系统使用的是禁止/恢复中断而不是关闭/开启中断。关闭中断的函数总是在返回到调用它的函数前恢复中断。没有任何一个子程序能够显式地开启中断，因为在进入到中断处理器的入口时中断就被关闭了，直到返回时才恢复。关于禁止和恢复中断的唯一例外就是在系统启动时开启中断的初始化方法。

12.15 观点

中断和进程之间的关系是操作系统最细致和复杂的一面。中断是一种低层机制——它们是基本硬件的一部分并以连续的操作来定义，如访问－执行循环；进程是高层抽象——它们是操作系统设计者想象出来的，并以一套系统函数来定义的。因此，理解中断时不必考虑并发进程，理解并发进程时不必考虑中断，这样思考是最简单的。

不幸的是，进程的抽象世界与中断的现实性界相结合形成了一个智力挑战。如果中断和进程间的交互不会想象中的那样复杂，人们就不会如此深入地考虑问题。如果读者觉得这种交互难以理解，请放心，很多人都遇到相同的问题。不过读者也要立刻鼓励自己去掌握它。

12.16 总结

为了处理中断，操作系统需要保存处理器状态、确定发出中断的设备并为相应的设备提供处理程序。由于高级语言（比如 C 语言）不提供直接操纵处理器或者协处理器寄存器的方法，所以一些中断进程代码并不是用 C 语言写的。当然示例系统也不是所有的中断代码都用汇编语言写的，它把中断分配器分成了两个部分：低层部分是用汇编写的，高层部分则是用 C 语言写的。

分配器抽获中断，保存寄存器状态，确定请求中断的设备并把控制权交给合适的高层中断处理程序。当高层中断处理程序返回时，控制权返回到重新加载寄存器的分配器并执行恢复状态的特殊指令，
211 最后返回给被中断的程序。

多个规则控制着中断过程。首先，中断代码不能让中断禁止任意长的时间，这样会使设备无法使用。中断时间的长短取决于连接到系统的设备，而不是运行的设备本身。其次，中断代码可能有空进程运行，它决不能调用会使执行程序离开当前或就绪状态的函数。再次，中断处理程序不能显式地开

后中断。

尽管开启中断是禁止的，但在一个等待的进程变成就绪状态时，中断处理程序必须进行重新调度。这样做就能保持调度不变原则，同时这也意味着一个进程等待I/O操作结束后会被唤醒。当然，系统必须保证在重新调度前，全局数据结构是合法的。重新调度不会引起连锁中断，因为每个进程在自己的栈中最多只有一个中断。

练习

12.1 假设一个中断处理程序包含了显式的中断开启功能，描述系统可能遇到的问题。

12.2 修改中断处理程序使其能够开启中断，观察系统经过多长时间会崩溃。感到惊讶吗？具体确认系统崩溃的原因（注意：本题中的关闭计时器设备将在第13章讨论）。

12.3 设想一个处理器，当中断发生时，其硬件能够自动地将上下文切换到一个特殊的中断进程中。这个中断进程的唯一目的是执行中断代码。试解释这样的操作系统是更容易设计还是更难设计。提示：中断进程是否允许重新调度？

12.4 本章曾提及一些系统会为每个设备分配一个独立的中断地址并将硬件的控制权直接交给处理程序。因此，分配软件就不需要了。这种设计的主要缺点是什么？

12.5 假设有8个设备，每个设备接收字符的速度是115 Kbaud（115千位/秒，或大约11 500字符/秒），计算每个中断要多少毫秒？

212

实时时钟管理

我们没有好的模型来。

—— Eugene Ionesco

13.1 引言

前面的章节介绍了操作系统的两个重要方面：一个是提供并发处理机制的处理器管理系统；一个是允许将内存块动态分配和释放的内存管理器。第12章介绍了中断处理。该章介绍了中断处理原则，讲述了当中断发生时，操作系统如何获取控制权，并解释控制如何由分配器传递给特定设备的中断处理程序。

本章将继续讲述中断处理，介绍计时器硬件和操作系统如何使用实时机制来提供具有控制定时事件能力的抽象。本章介绍两个基本概念：增量链表数据结构和选择性占。它们解释了操作系统如何利用时钟向一组相同优先级的进程提供轮转服务。后续的章节将通过介绍其他I/O设备类型入手介绍中断。

[215]

13.2 定时事件

许多应用程序都使用定时事件。比如，一个应用可能打开一个窗口来显示一条消息，让窗口在屏幕上显示5秒，然后关闭窗口。一个要求用户输入密码的应用程序如果没有得到用户的响应，可能需要在30秒内关闭。部分操作系统也使用定时事件：比如，许多网络协议都要求发送方在指定时间内没有收到响应时重发请求。类似地，如果一个外部设备，比如一台打印机，在一段时间内都处于无连接状态，则操作系统就应该通知用户。如果一个嵌入式系统没有独立硬件机制，则操作系统使用定时事件来记录当前的日历日期和时间。

因为时间是基础，所以大部分的操作系统都提供必需的机制。使应用程序创建和管理定时事件更容易。有些操作系统使用通用的异步事件范式。在其中，程序员定义一类事件处理程序，在事件发生时操作系统调用合适的处理程序。定时事件处理符合异步范式：运行中的进程要求特定事件在未来的某个时间单元内出现。有些系统则使用同步事件范式，在其中操作系统只提供延迟，程序员按需创建额外的进程来调度事件。我们的示例系统将使用异步的方法。

13.3 实时时钟和计时器硬件

Xinu有四种硬件设备与时间有关系：

- 处理器时钟。
- 实时时钟
- 日历时钟
- 间隔定时器。

处理器时钟　术语处理器时钟指的是产生高精度周期性脉冲（比如方波）的硬件设备。处理器时钟控制处理器执行指令的速率。为了最小化硬件投资，低端嵌入式系统经常使用处理器时钟作为计时信号源。不幸的是，处理器时钟频率使用起来不是很方便（因为时间脉冲快，并且频率不是10的幂次方）。

实时时钟　实时时钟与CPU相互独立，并且以1毫秒为时间间隔产生脉冲（即，1000次/秒）。每个脉冲产生一次中断。一般来说，实时时钟硬件并不对脉冲进行计数——如果操作系统需要计算流逝的时间，则当每个时钟中断发生时，系统必须对计数器加1。

日历时钟　技术上，日历时钟是一个精密计时器，它可以精确地计算出流逝的时间。其硬件包括一个内部实时时钟和一个调节脉冲的计数器。与普通的时钟一样，时间可以被修改。然而，一旦设定，

它的运行就与处理器独立。只要系统不断电它就能够持续运作（某些装置会配备一个小电池来确保在外部电源关闭时时钟依然能够正常工作）。不像其他时钟，日历时钟并不会产生中断，要使用它，处理器必须设置或者查询计数器。

间隔计时器　间隔计时器，有时又称为倒计时器或计时器，它由一个内部实时时钟和一个计数器构成。为了使用计时器，系统将计数器初始化为一个正值。每产生一个实时时钟脉冲，计数器减 1，当计数器的值为 0 时产生中断。正计时器要求操作系统将其初始化为 0 并且设置一个上限值。正如其名字，正计时器将计数器加 1，当计数器达到上限值时中断操作系统。

相对于实时时钟来说，计时器的主要优点在于更低的中断负载。实时时钟定期中断，即使下一个事件是在许多时间间隔之后才会发生。计时器只有在事件调度时才会产生中断。而且，计时器比实时时钟更灵活，因为它可以模拟实时时钟。例如，为了模拟一个每秒 R 次脉冲的实时时钟，可以将计时器设置为每 $1/R$ 秒产生一次中断。以上可以总结为：

> 可用于产生计时事件的硬件包括：实时时钟和间隔计时器。通过计算两次脉冲之间的时间 T，操作系统可以使用计时器模拟实时时钟，并将每个中断的计时器重新设置为 T。

E21001 硬件包含有一个间隔计时器。本章后面的示例代码说明了使用计时器模拟实时时钟是很简单的事情。

13.4　处理实时时钟中断

每个脉冲都会产生一次实时时钟中断，中断不计数，也不会积累。同样，如果使用计时器模拟实时时钟，计时器也不会积累中断。在任何一种情况下，如果处理器不能在下一次中断到来之前对上一次的中断做出反应，处理器将不会收到第二次中断。更重要的是，硬件不会检测和报告这样的错误。

> 如果处理器花费了太长的时间来处理实时时钟中断，或者处理了中断关闭时间大于一个时钟周期的中断，该中断就会被忽略，而且没有错误会被报告。 217

对于系统设计人员来说，实时时钟硬件的操作会有两个重要的后果。首先，因为系统必须能够在两次实时时钟中断之间执行许多条指令，所以处理器的处理速度要远高于实时时钟的频率。其次，实时时钟中断可能成为潜在的错误源。如果操作系统无法响应中断，那么时钟中断将会漏掉而对计时产生影响。这样的错误是不易察觉的。

因此，系统必须设计成可以快速响应时钟中断。有些硬件通过将实时时钟中断设置为最高优先级来解决这个问题。因此，如果 I/O 设备和时钟设备同时产生中断，处理器就会先处理时钟中断，当时钟中断响应完后再处理 I/O 中断。

13.5　延时与抢占

接下来，我们将关注操作系统使用时间的两种方式：

- 定时延迟。
- 抢占。

定时延迟　操作系统允许任意的进程请求一个定时延迟。当进程请求一个定时延迟时，操作系统把进程从当前状态移入到一个新的状态（休眠），并在特定时间调度要醒事件来重启该进程。当唤醒事件出现时，进程对 CPU 的使用变为合法，并根据调度策略来运行。后面的各节将介绍进程如何进入休眠状态以及如何在正确的时间被唤醒。

抢占　与第 5 章讲述的调度策略一样，操作系统中的进程管理器使用抢占机制来实现时间分片，进而保证在轮转机制下，各个进程有相同的优先级。系统定义了一个最大时间片 T，进程在时间片 T 内可以执行而不受其他进程的影响。当从一个进程切换到另一个进程时，调度器会在未来 T 时间后调度抢占事件。当抢占事件发生时，事件处理程序简单地调用 resched。

为了理解抢占的工作机制，我们假定系统包含许多具有相同优先级的进程。当一个进程执行时，其他有着相同优先级的进程都在就绪链表里。在这种情况下，调用 resched 把当前进程放在就绪链表的

最后，即其优具有相同优先级的进程后面，并切换到链表的第一个运行。因此，如果 n 个相同优先级的进程准备使用处理器，则在任何进程接受更多服务前，所有 n 个运程都将最多执行一次时间片。

时间片的长度应该为多少呢？时间片的长度决定了粒度程度。使用短的时间片使粒度更变小。小的粒度会使相同优先级的各个进程按几乎相同的速度执行。然而，小的粒度也会造成负载的增加因为系统经常进行上下文的切换。大的粒度可以减小上下文切换的负载，但也会让一个进程切换到下一个进程前占有处理器太长时间。

已证明，在大部分的系统中，为了抢占，进行理少使用处理器太长时间。相反，运程通常执行 I/O 或执行系统函数，比如 wait，导致重新调度。事实上，进程一般在时间片结束前运程自动放弃对 CPU 的控制。更丰富的是，因为输入和输出比处理慢，进程的大部分时间都用于等待 I/O 完成。

没有抢占，操作系统将无法从执行无限循环的进程那里夺回 CPU 的控制权。

13.6 使用计时器来模拟实时时钟

我们使用术语时钟滴答（clock tick）又指代实时时钟中断，用术语滴答频率（tick rate）来表明时钟滴答的频率。因为它是使用计时器硬件来模拟实时时钟，我们可以选择一个合适的滴答频率。Xinu 选择每 1ms 让时钟滴答一次。因此，当计时器中断发生时，系统设置定时器延时为 1ms（即 0.001s），从而使系统在 1ms 后再次产生一个中断。

不幸的是，F2100L 中的计时器硬件与很多小型嵌入式系统中使用的硬件相同：硬件仅让处理器时钟来对计数器加 1。就是说，间隔计时器对处理器周期进行度量。如果处理器时钟每秒包含 P 个周期，那么计时器每秒计算 P 次。为了获 1ms，我们必须将计时器设置为 1ms 内的处理器周期数。

幸运的是，在 F2100L 处理中处理器时钟的频率是知道的，这就使我们可以很容易地计算 1 毫秒内处理器周期的数值。在示例程序中，这个值在提前计算好的，保存在符号常量 CLKCYCS_PER_TICK 中。当计时器中断产生时，计时器被重新设置为之前的值加 CLKCYCS_PER_TICK，这样在 1ms 以后就会引起另一次的中断。

13.7 抢占的实现

示例程序实现了抢占和定时延迟。在检查这段代码之前，我们先讨论下这两种方式的实现。抢占是最容易理解的。预先定义的常量 QUANTUM 定义了一个时间片内时钟滴答的次数。当从一个进程切换到另外一个运程时，resched 将全局变量 preempt 设置为 QUANTUM。每次模拟时钟滴答时，时钟中断处理程序将 preempt 的值减 1。当它减为 0 时，时钟中断处理程序将 preempt 重新设置为 QUANTUM，并调用 resched。调用 resched 后，处理程序从中断返回。

对 resched 的调用会有两种可能的结果。第一，如果当前执行的进程会唯一拥有最高优先级的进程，则 resched 将立即返回，中断处理程序将返回，当前运程将在中断处继续执行。第二，如果另一个就绪进程与当前进程有相同的优先级，则 resched 将切换到新进程。最终，resched 将切换回被中断的进程。将 preempt 赋值为 QUANTUM 可以处理当前进程持续运行的问题。这样的赋值是必要的，因为 resched 只在运程切换时重置抢占计数器[⊖]。重置抢占计数器可以防止在一个进程无限执行时抢占计数器下溢。

13.8 使用增量链表对延迟进行有效管理

为了实现延迟，操作系统必须维护申请延时进程的集合。每个进程根据它申请的时间定义一个延迟，任何进程可以在任何时间提出申请。当一个进程的延迟过期时，系统将该运程的状态转换为就绪并调用 resched。

⊖ resched 的代码见 5.6 节。

每个进程都有特定的延迟请求，操作系统如何维护延迟进程集合呢？操作系统无法在每一个时钟滴答内搜索完任意长的睡眠进程链表。因此，需要设计高效的数据结构，这种结构只需要时钟中断处理程序在每一个时钟滴答内执行几条指令来检测请求特定延迟的进程集合。

解决方案是使用一种称为增量链表的数据结构。增量链表包含一个进程集合，并且链表是按每个进程唤醒的时间进行排序。使计算更高效的方法是使用相对时间而非绝对时间。也就是说，不是存储进程被唤醒的绝对时间，而是在增量链表中存储此进程相对于前一个进程必须延迟的多余的时间。

增量链表中第一个进程的键值指定了相对于当前时间，该进程需要等待的时钟滴答数。

其他每一个进程的键值指定了相对于各自的前一个进程，该进程需要等待的时钟滴答数。

例如，假设进程 A、B、C、D 分别请求延迟 6、12、27、50 个时钟滴答。而且假设这些的请求在几乎相同时间被做出（即，在一个时钟滴答内）。图 13-1 显示了增量链表的结果。 220

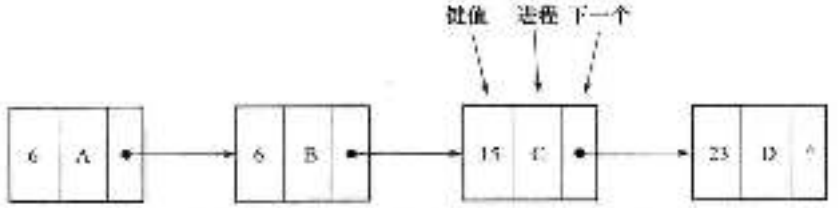


图 13-1　有 4 个进程的增量链表概念图，图中各进程的延迟分别为 6、12、27、50 个时钟滴答

给定一个增量链表，就可以通过计算部分键值的和获到每个进程被唤醒的时间。在图 13-1 中，进程 A 被唤醒的时间为 6，进程 B 被唤醒的时间为 6 + 6，进程 C 被唤醒的时间为 6 + 6 + 15，进程 D 被唤醒的时间为 6 + 6 + 15 + 23。

13.9　增量链表的实现

与其他进程链表一样，延迟进程的增量链表位于 queuetab 结构中。全局变量 sleepq 包含睡眠进程增量链表的队列 ID。在每一个时钟滴答，时钟中断处理程序递减 sleepq 中第一项的键值。如果该值变为 0，则时钟处理程序调用 wakeup 将第一个进程唤醒，因为它的延迟时间已到。

睡眠进程队列中的第一个进程是最先会被唤醒的进程，它的键值保存了唤醒之前会经历的时钟滴答。因为链表中所有后续进程的延迟都是与第一个延迟相关的，所以时钟中断只需递减第一个进程的值，并且不需要扫描整个链表。

使用增量链表的函数看起来好像很直接，但是实现起来可能需要一些技巧。因此，程序员必须关注每一个细节。就像下面将介绍的 insertd 函数，它需要 3 个参数：进程 ID（pid）、队列 ID（q）和延迟（key）。insertd 计算一个相对延迟，然后将指定的进程插入到 sleepq 队列中合适的位置。在该代码中，变量 next 扫描增量链表搜寻插入新进程的位置。

从观察可知，变量 key 的初始值定义了与当前时间相关的延迟。因此，变量 key 与增量链表中第一项的键值进行比较。然而，链表中的后续键值说明延迟与前驱的键值相关联。因此，链表中后续结点的键值不能直接与变量 key 的值进行比较。为了实现延迟的可比较性，当搜索进行时，insertd 函数将从 key 中减去相对延迟量，以保证不变性：

在搜索的任意时刻，key 和 queuetab[next].qkey 都指定了相对于“下一个进程”的前一个进程的相对延迟时间。

```
/* insertd.c - insertd */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insertd  -  insert a process in delta list using delay as the key
 *------------------------------------------------------------------------
 */
```

```
status  insertd(                        /* assumes interrupts disabled  */
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* delay from "now" (in ms.)    */
        )
{
        int     next;                   /* runs through the delta list  */
        int     prev;                   /* follows next through the list*/

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        prev = queuehead(q);
        next = queuetab[queuehead(q)].qnext;
        while (next != queuetail(q)) && (queuetab[next].qkey <= key)) {
                key  = queuetab[next].qkey;
                prev = next;
                next = queuetab[next].qnext;
        }

        /* Insert new node between prev and next nodes */

        queuetab[pid].qnext = next;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[next].qprev = pid;
        if (next != queuetail(q)) {
                queuetab[next].qkey -= key;
        }

        return OK;
}
```

尽管在搜索过程中，insertd 形式地检查链表的尾部，但在不影响运行的情况下，该测试可以被忽略。记住，链表尾部的键值比任何要插入的键值都长。只要这个断言成立，当到达尾部时循环就会终止。因为 insertd 不会检查参数的值，所以测试就可以安全进行。

insertd 遍历链表，当它发现待插入结点的相对延迟值小于链表中某一结点时，会在该位置插入新的结点进链表。同时 insertd 必须在链表的剩余结点中减去由于新结点插入所带来的延迟值的改变。为此，insertd 在下一个结点的键值中减去了待插入结点的键值。这里的减法操作必须保证产生一个非负数，这是因为循环终止的条件要保证待插入的键值小于链表上的键值。

222 / 223

13.10 将进程转入睡眠

应用程序不会调用 insertd，也不会直接访问睡眠队列。相反，应用程序调用系统调用 sleep 或 sleepms 来请求一个延迟。这两个函数的不同在于它们参数的粒度：sleepms 的参数以毫秒为单位定义延迟，sleep 的参数以秒为单位定义延迟。在 32 位的处理器上，以毫秒为单位测量延迟可以给延迟定义一个很好的范围：一个无符号 32 位整数可以表示的最大延迟为 1100 小时（49 天）。然而，在嵌入式系统中使用 16 位标数，毫秒级延迟意味着调用者只能表示 32 秒的延迟。因此，16 位处理器的操作系统通常使用较大粒度的测量标准（如 1/10 秒）。

为了避免代码的重复，sleep 函数将参数乘以 1000 并调用 sleepms。sleep 的一个有价值的方面是检查它的参数大小：避免整数溢出，sleep 将延迟设置为一个可以用一个 32 位无符号整数表示的值。如果调用者定义了一个更大的值，sleep 会返回 SYSERR。

sleepms 将调用进程插入到睡眠进程的增量链表中。当它做这延眠进程的增量链表中时，进程状态

将不再是就绪或当前。应该将进程转入什么状态呢？睡眠和挂起不同，睡眠是在等待一条消息，或等待一个信号量。因此，由于没有任何一个状态能够满足运行当前的状态，所以添加一个新的状态。我们把这种新的状态称为睡眠，并把它放在符号常量 PR_SLEEP 中。图 13-2 显示了包括睡眠状态的状态转换。

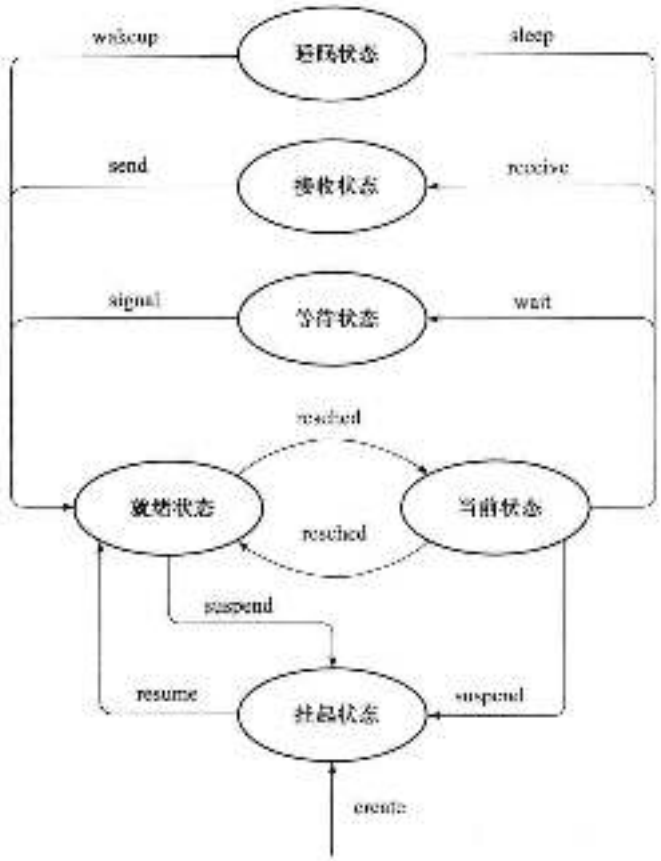


图 13-2　包含了睡眠状态的状态转换

下面的文件 sleep.c 展示了 sleepms 的执行。它包括一个特别的情况：如果一个进程将延迟定义为 0，sleepms 将会立即调用 resched。否则，sleepms 调用 insertd 将当前进程插入到延眠进程的增量链表中，并将进程的状态改为睡眠，调用 resched 允许其他进程执行。

```
/* sleep.c - sleep sleepms */

#include <xinu.h>

#define MAXSECONDS      4294967         /* max seconds per 32-bit msec  */

/*------------------------------------------------------------------------
 *  sleep  -  Delay the calling process n seconds
 *------------------------------------------------------------------------
 */
syscall sleep(
        uint32          delay           /* time to delay in seconds     */
        )
{
        if (delay > MAXSECONDS) {
                return(SYSERR);
        }
```

```
        sleepms(1000*delay);
        return OK;
}

/*------------------------------------------------------------------------
 *  sleepms  -  Delay the calling process n milliseconds
 *------------------------------------------------------------------------
 */
syscall sleepms(
          uint32        delay           /* time to delay in msec.       */
        )
{
        intmask mask;                   /* saved interrupt mask         */

        mask = disable();
        if (delay == 0) {
                yield();
                restore(mask);
                return OK;
        }

        /* Delay calling process */

        if (insertd(currpid, sleepq, delay) == SYSERR) {
                restore(mask);
                return SYSERR;
        }
        proctab[currpid].prstate = PR_SLEEP;
        resched();
        restore(mask);
        return OK;
}
```

[224]
[225]

13.11 定时消息接收

Xinu 拥有一个在计算机网络中非常有用的机制：定时消息接收。本质上该机制允许进程等待规定好的时间长度或消息到达为止，以先出现者为准。也就是说，这种机制执行起来就像一个含有等待最长时间上界的同步接收（receive）函数。

定时消息接收机制的基本原理是分离等待（disjunctive wait）：进程阻塞直到两个事的其中之一发生。很多网络协议采用分离等待以实现处理丢包的超时重传技术。当发送者发送一个消息时，它也同时启动计时器，然后等待回应到达或者计时器超时。如果回应消息先到，网络就删除定时器。当消息或者回应消息丢失时，计时器到期，协议软件就重传请求的副本。

在 Xinu 中，当一个进程请求定时接收时，该进程就进入睡眠进程队列中，与其他的睡眠进程一样。然而，与分配进程状态 PR_SLEEP 不同的是，系统将进程置于状态 PR_RECTIM 中以指明它已加入到具有计时模式的接收中。如果睡眠定时器到期，进程就像其他睡眠进程一样被唤醒。如果一个消息在延迟到期之前到达，函数 send 就调用函数 unsleep 将进程从睡眠进程队列中移除，然后继续传送消息。一旦它恢复执行，该进程就检查它的进程表项来查看是否有消息到达。如果没有消息，定时器就会到期。

[226] 图 13-3 显示了一个定时消息接收的状态图，该状态图拥有一个新状态：TIMED-RECV。

正如我们已经见到的，当一个进程处于定时接收状态时，第 8 章中的 send[⊖]函数将处理这种情况。因此，我们只需检查函数 recvtime 和函数 unsleep 的代码。除了之前要调用函数 resched 外，函数 recvtime 与函数 receive[⊖]几乎一样，它利用函数 insertd 以将调用进程插入睡眠进程队列中，分配状态 PR_RECTIM

⊖ send 可以在 8.5 节找到。
⊖ receive 可以在 8.6 节找到。

状态而不是状态 PR_RECV。文件 recvtime.c 中有该函数的代码。

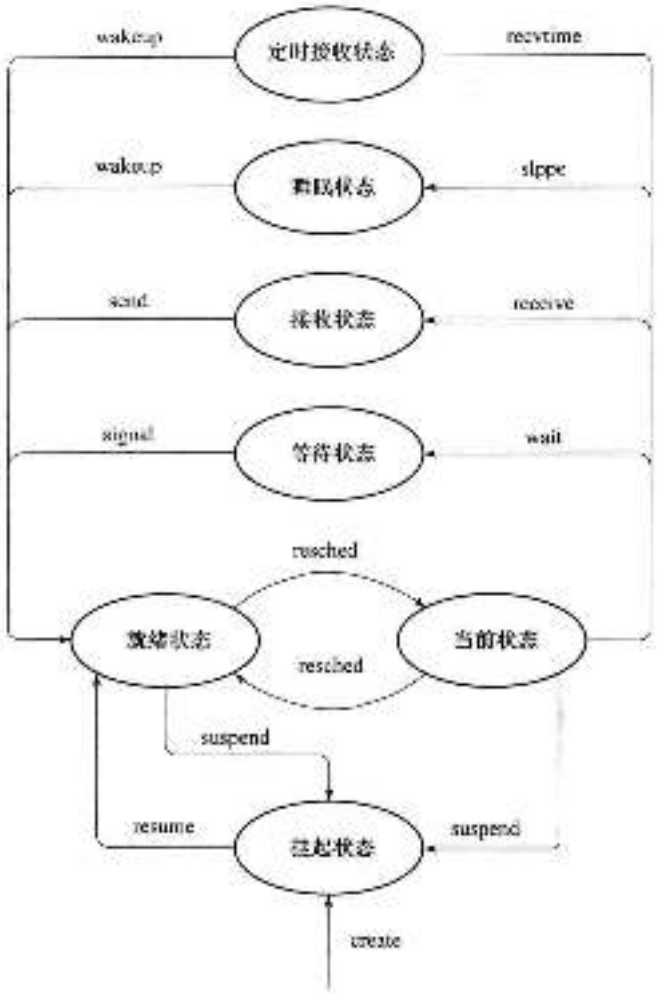


图 13-3　含定时接收状态的状态转换

```
/* recvtime.c - recvtime */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvtime  -  wait specified time to receive a message and return
 *------------------------------------------------------------------------
 */
umsg32  recvtime(
          int32         maxwait         /* ticks to wait before timeout */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* tbl entry of current process */
        umsg32  msg;                    /* message to return            */

        if (maxwait < 0) {
                return SYSERR;
        }
```

```
        mask = disable();

        /* Schedule wakeup and place process in timed-receive state */

        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) { /* if message waiting, no delay */
                if (insertd(currpid,sleepq,maxwait) == SYSERR) {
                        restore(mask);
                        return SYSERR;
                }
                prptr->prstate = PR_RECTIM;
                resched();
        }

        /* Either message arrived or timer expired */

        if (prptr->prhasmsg) {
                msg = prptr->prmsg;     /* retrieve message             */
                prptr->prhasmsg = FALSE;/* reset message indicator      */
        } else {
                msg = TIMEOUT;
        }
        restore(mask);
        return msg;
}
```

[228] 函数 unsleep 将一个进程从睡眠进程队列中移除。如果有其他进程位于队列中，函数 unsleep 必须调整随后进程的延迟值以抵消被移除的进程。unsleep.c 文件有该函数的代码。

```
/* unsleep.c - unsleep */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  unsleep  -  Remove a process from the sleep queue prematurely by
 *              adjusting the delay of successive processes
 *------------------------------------------------------------------------
 */
syscall unsleep(
        pid32           pid             /* ID of process to remove      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */

        pid32   pidnext;                /* ID of process on sleep queue */
                                        /* that follows the process that*/
                                        /* is being removed             */

        mask = disable();

        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        /* Verify that candidate process is on the sleep queue */

        prptr = &proctab[pid];
        if ((prptr->prstate!=PR_SLEEP) && (prptr->prstate!=PR_RECTIM)) {
```

```
        restore(mask);
        return SYSERR;
}

/* Increment delay of next process if such a process exists */

pidnext = queuetab[pid].qnext;
if (pidnext < NPROC) {
                queuetab[pidnext].qkey += queuetab[pid].qkey;
        }

        getitem(pid);                   /* unlink process from queue */
        restore(mask);
        return OK;
}
```

13.12　唤醒睡眠进程

时钟中断处理程序在每一个时钟滴答递减 sleepq 中的第一个键值的计数值，当延迟为零时调用 wakeup 唤醒进程。wakeup 必须能够处理同一时间多个进程被唤醒的情况。因此，wakeup 在所有键值为零的进程中迭代地从 sleepq 中移除进程并再次调用 ready 以确保进程符合 CPU 服务标准。因为 wakeup 是从一个中断分配器中调用的，所以它假定在入口处中断是禁止的。因此，在调用 ready 之前 wakeup 不能显式地禁止中断。一旦 wakeup 将进程移入就绪等表中，它就调用 resched 重建调度不变式并允许另一个进程执行。

```
/* wakeup.c - wakeup */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wakeup  -  Called by clock interrupt handler to awaken processes
 *------------------------------------------------------------------------
 */
void    wakeup(void)
{
        /* Awaken all processes that have no more time to sleep */

        while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
                ready(dequeue(sleepq), RESCHED_NO);
        }
        resched();
        return;
}
```

229
|
230

13.13　时钟中断处理

现在我们来准备测试时钟中断程序 clkhandler。每一次调用它，模拟时钟就发生中断。正如前面所描述的，时钟中断处理程序递减 sleepq（假定 sleepq 非空）中第一个进程的剩余时间。如果剩余延迟为零，clkint 调用 wakeup 从睡眠队列中移除所有延迟时间为零的进程并将它们置于就绪（ready）状态。最后，clkint 递减抢占计数器，如果抢占计数器为零，就调用 resched。

```
/* clkhandler.c - clkhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  clkhandler  -  handle clock interrupt and process preemption events
 *                      as well as awakening sleeping processes
 *------------------------------------------------------------------------
 */
```

```
interrupt clkhandler(void)
{
        clkupdate(CLKCYCS_PER_TICK);

        /* record clock ticks */

        clkticks++;

        /* update global counter for seconds */

        if (clkticks == CLKTICKS_PER_SEC) {
                clktime++;
                clkticks = 0;
        }

        /* If sleep queue is nonempty, decrement first key; when the    */
        /* key reaches zero, awaken a sleeping process                  */

        if (nonempty(sleepq) && (--firstkey(sleepq) <= 0)) {
                wakeup();
        }

        /* Check to see if this proc should be preempted */

        if (--preempt <= 0) {
                preempt = QUANTUM;
                resched();
        }
        return;
}
```

13.14 时钟初始化

时钟初始化例程 clkinit 执行 4 个主要功能。第一，它分配一个用以保存延眠进程增量链表的队列，在全局变量 sleepq 中存储队列 ID。第二，它开启一个每秒递增的计数器，并将其初始化为 0。第三，该代码将时钟中断处理程序的地址存储在名为 interruptVector 的数组中，允许中断分配器将计时器中断与 clkhandler 相关联。第四，clkinit 调用 clkupdate 更新间隔计时器。在文件 clkinit.c 中有该初始化程序代码。

```
/* clkinit.c */

#include <xinu.h>
#include <interrupt.h>
#include <clock.h>

uint32  clkticks = 0;           /* ticks per second                     */
uint32  clktime = 0;            /* current time in seconds              */
qid16   sleepq;                 /* queue of sleeping processes          */
uint32  preempt;                /* preemption counter                   */

/*------------------------------------------------------------------------
 * clkinit  -  initialize the clock and sleep queue at startup
 *------------------------------------------------------------------------
 */
void    clkinit(void)
{
        sleepq = newqueue();    /* allocate a queue to hold the delta   */
                                /* list of sleeping processes           */
```

```
clkticks = 0;           /* start counting one second                */

/* Add clock interrupt handler to interrupt vector array */

interruptVector[IRQ_TIMER] = &clkhandler;

/* Enable clock interrupts */

enable_irq(IRQ_TIMER);
        /* Start interval clock */

        clkupdate(CLOCKS_PER_SEC/HZ);
}
```

13.15 间隔计时器管理

在R2000上，只有通过协同处理器（co-processor）才能访问读取间隔定时器。因此，控制定时器的代码是用汇编语言写的。

前面提到过，计时器硬件使用处理器时钟，每微秒调度一次中断。为了更好地理解时钟管理程序，有必要指出软件使用递增计数的方法。定时器一直处于计时状态直到超过操作系统所设置的阈值N，此时计时器发生中断。

理论上，编写一个毫秒级的实时时钟是很简单的：当发生中断时，简单只需要增加N，N是一个常数，它的值等于1毫秒内计时器累加的时钟周期数。原因就在于硬件实现的计数器是一直计数的。在最后一次计时器中断之前将阈值设置为N可以使下一次中断被严格地调度为1毫秒。

然而，实际上，总是给前面的阈值增加N个单位的简单方法可能无法工作。原因在于当计时器到期时中断可能被禁止了（比如，因为处理器正在处理其他设备）。在当前中断结束后，在分配器调用时钟中断处理程序及处理程序更新计时器之前，又经过了少量的额外时间。因此，在计时器到期时与中断处理程序复位间隔计时器之间将有少量的时间。如果该时间为1毫秒，则计时器的值等于N与前一个阈值的和，在达到阈值之前计数器需要回滚。为了处理该情况，计时器管理程序就在前一个阈值增加N并将该值与时间计数值进行比较。如果计算出阈值已经过了，那么代码将阈值复位到当前计数值与N的和。文件clkupdate.S中含有该段代码。

231 / 233

```
/* clkupdate.S - clkupdate, clkcount */

#include <mips.h>

.text
        .align  4
        .globl  clkupdate
        .globl  clkcount

/*------------------------------------------------------------------------
 * clkupdate - update the timer by a specified number of cycles
 *------------------------------------------------------------------------
 */

/* Note: there are two cases
 *  Normal case: COMPARE is increased by N cycles and stored as the
 *               new threshold (N cycles beyond previous threshold)
 *  Abnormal case: the timer has already accumulated more than N cycles
 *               beyond the previous threshold.  Start over by making
 *               the threshold equal to the current count + N
 */
```

```
clkupdate:
        mfc0    v0, CP0_COMPARE     /* v0 = COMPARE                          */
        mtc0    v1, CP0_COUNT       /* v1 = COUNT                            */
        addu    v0, v0, a0          /* v0 = COMPARE + cycles                 */
        blez    v0, v1, compare_up  /* v0 <= COUNT, then goto compare_up*/
        mtc0    v0, CP0_COMPARE     /* Update COMPARE                        */
        jr      ra

/* Abnormal case: timer is beyond the next interrupt count, reset          */

compare_up:
        addu    a0, v1, a0          /* a0 = COUNT + cycles                   */
        mtc0    a0, CP0_COMPARE     /* COMPARE = a0                          */
        jr      ra

/*-------------------------------------------------------------------------
 * clkcount returns the count from the free-running clock
 *-------------------------------------------------------------------------
 */
clkcount:
        mfc0 v0, CP0_COUNT
234]    jr   ra
```

13.16 观点

时钟与计时器管理无论在技术上还是认识上都是富于挑战的。一方面，因为时钟或计时器中断经常发生且具有高的优先级，所以 CPU 处理时钟中断的总时间是很高的且在处理时屏蔽了其他中断。因此，必须对中断处理程序的代码进行优化以缩短它的执行时间。另一方面，当操作系统允许进程直接定义事件时，意味着它可以调度在同一时间同时发生的很多事件，也就意味着对于一个给定的中断，它的执行时间可以是任意长的。但对于处理器速度相对较慢的嵌入式系统以及其他要求实时服务的设备而言，这矛盾就显得尤为突出。

大多数操作系统允许调度任意事件，并对多个事件发生冲突时采用延迟处理方法。当手机启动一个程序时，这种方法就可能造成其界面不能实时显示，或者当运行多个程序时短信可能要花很长时间才能发送出去。核心问题是：操作系统如何在硬件资源有限的条件下提供最好的精确时钟服务，以及当请求无法响应时该如何告知用户？事件要不要分配优先级？如果要的话，事件优先级与调度优先级如何进行交互。这些问题到目前还没有一个很好的解决方案。

13.17 总结

实时时钟以固定的间隔向 CPU 发出中断请求。本书的设计使用计时器来模拟时钟中断。中断处理程序负责处理时钟中断和对下一个中断复位计时器。

操作系统使用时钟来处理抢占和进程延迟。当进程使用了 QUANTUM 个时间片的时间的处理器后，每次系统切换上下文时被调度的抢占事件就会强制调用调度器。抢占能够确保任何进程均不能永久占有 CPU，并且能够通过保证同等优先级进程间的轮转服务来实现就绪机制。

当进程请求计时延迟时，系统就会调度一个事件，该事件使得一个正在运行的进程将自己放入进程的有序非阻塞表中。中断程序通过将进程移入就绪状态并重新调度来唤醒一个计时器到期的睡眠进程。增量链表为管理睡眠进程提供了一个非常有效的方法。

练习

13.1 修改代码使时钟中断频率提高 10 倍，并让时钟中断处理程序在处理时忽略前 9 个中断。这个新添的
[235] 中断需要多少额外的代价？

13.2 设计一个实验以检测系统是否丢失时钟中断？如果是，丢失频率是多少。当中断发生时，从计时

器中读取累计的值并与“对照”值进行比较，看看与期望的值相比有多少的额外周期产生。

13.3　由时钟中断唤醒的两个睡眠进程，其中一个拥有比当前运行进程更高的优先级，跟踪这两个进程的情况。

13.4　当 QUANTUM 设置为 1 时，将会导致哪一部分失败。提示：考虑以下情况：当运行中断程序时，此时切换到一个由 resched 指定的进程的情况。

13.5　sleepms(3) 函数能否保证至少 3 毫秒的延迟？还是精确 3 毫秒？还是至多 3 毫秒？

13.6　仔细阅读 kill 函数代码，查找由 kill 函数将一个进程从睡眠队列中移除时导致的问题。重写 kill 函数代码修复该问题。

13.7　wakeup 调用 wait 时会出现什么问题？

13.8　要准确记录大量进程占有处理器的时间，操作系统还需处理以下的问题：当一个中断产生时，即使这个中断与当前进程看起来不相关，最普通的方法仍是让当前进程处理中断事件。对操作系统如何测量执行中断例程的代价进行分析，例如 wakeup 对受影响进程的执行代价分析。

13.9　如果将本章中的代码移植到一个同型号但有更高时钟频率的机器上，将出现什么现象？为什么？

13.10　设计一个实验验证抢占事件是否会导致系统重新调度。注意：一个通过调用 resched 函数用于测试重量或者检测 I/O 性能的单一进程会干扰实验。

13.11　假定一个系统有 3 个进程：一个处于睡眠状态的优先级比较低的进程 L 和两个具备执行条件的且优先级较高的进程 H_1 与 H_2。再假定当切换到进程 H_1 时立刻产生一个时钟中断，此间中断处理程序调用 resched 函数且进程 L 处于准备状态。虽然进程 L 不会运行，但 resched 也会将进程 H_1 切换到进程 H_2 同时不会将 quantum 值赋给 H_2。给出一个修改 resched 的方案，使得它能够保证一个进程不失去对处理器的控制权，除非优先级高的进程处于准备状态或者它的时间片到期了。 236

第 14 章

Operating System Design: The Xinu Approach, Linksys Version

设备无关的 I/O

我们总是被自己的意念所束缚，以至于一段时间之后，我们会喜欢被他人掌控的不确定性。

—— Tom Stoppard

14.1 引言

前几章解释了并发进程和内存管理机制。第 12 章对中断的重要概念进行了讨论，描述了中断处理流程，给出了中断代码架构并且解释了中断处理与并发进程之间的关系。第 13 章对第 12 章进行了扩充，描述了如何使用实时时钟中断来实现抢占和进程延迟。

本章将对操作系统如何实现 I/O 进行更广泛的概览，内容包括建立 I/O 抽象的理论基础和适用于通用目的 I/O 设备的架构。本章还描述进程如何在不必理解底层硬件的基础上与设备之间进行数据收发。本章还定义一个通用模型，并描述其如何合并设备无关的 I/O 函数。最后，本章对一个高效的 I/O 子系统进行了测试。

239

14.2 I/O 和设备驱动的概念结构

操作系统之所以要控制并管理输入和输出设备，其中有 3 个原因。第一，由于大部分设备硬件使用低层接口，所以软件接口较为复杂。第二，由于设备是共享资源，所以操作系统提供的设备访问方式应基于公平和安全考虑。第三，操作系统定义了高层接口，当接口与设备交互时，它隐藏了其中的细节，从而让程序只能够使用一组连贯且统一的操作来进行处理。

在概念上，I/O 子系统能够分为 3 个部分：抽象接口，它包括处理时执行 I/O 的高层 I/O 函数；一组物理设备；和连接前两者的设备驱动软件。图 14-1 描述了三者之间的结构关系。

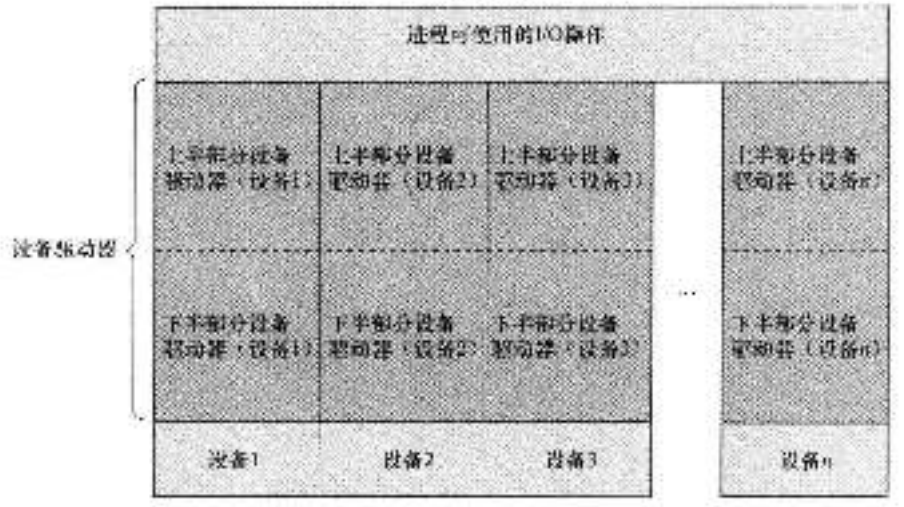


图 14-1 I/O 子系统概念结构，其中的设备驱动软件介于进程和底层设备之间

从图 14-1 中可以看出，设备驱动器软件在高层并发进程和低层硬件之间建立了桥梁。每个驱动在概念上分为两部分：上半部（upper half）和下半部（lower half）。当进程请求 I/O 时，调用上半部分的函数。这些函数通过进程间数据传送实现诸如读和写等操作。当设备中断时，中断分配器调用下半部分的处理程序函数。这些处理程序不仅处理中断，还与设备相互传送数据，并在可能触发额外

的 I/O 操作。

240

14.3　接口抽象和驱动抽象

操作系统设计者的最终目标是创建方便的编程抽象和找到高效的实现方法。关于 I/O，有两个编程抽象：

- 接口抽象。
- 驱动抽象。

接口抽象（interface abstraction）的问题是：操作系统应该给进程提供什么样的 I/O 接口？有多种可能的选择方案，这些选择简洁代表了在灵活性、简单性、高效性和通用性之间的权衡折中。为了理解这个问题涉及的范围，参见图 14-2，图中列出了一些示例设备和适合于该设备的操作类型。

设备	I/O范例
磁盘驱动器	寻到给定位置并传输一个数据块
键盘	接收单个输入的字符
打印机	传输单个将要输出的文本
音频输出	传输编码音频的连续流
无线网络	发送或接收单个网络数据包

图 14-2　示例设备和每个设备使用的 I/O 范例

早期的操作系统为每个单一硬件设备类提供了一组 I/O 操作。不幸的是，将设备特定的信息存放在软件里意味着当 I/O 设备被另一个供应商提供的同等设备取代时，就必须相应地修改软件。一个更加通用的方法是为每一类设备定义一组操作，并要求操作系统在给定设备上处理合适的低层操作。例如，操作系统提供了抽象函数 send_network_packet 和 receive_network_packet，这两个函数能够在任何类型的网络上传输网络数据包。第三种方法起源于 Multics 并由 UNIX 推广：选择一组数量小并能足够处理所有 I/O 的抽象 I/O 操作。

驱动抽象（driver abstraction）　我们可将第二类抽象看做是 I/O 语义。最著名的语义设计问题之一是同步：当进程正在等待 I/O 操作完成时是否会阻塞？同步接口，类似于前面描述的那个接口，提供了阻塞操作。例如，为了在同步系统中从键盘请求数据，进程调用上半部分的函数，这些函数能够 241
阻塞进程直到用户输入一个键。一旦用户按键，设备就产生中断，同时中断分配器调用下半部分的函数来处理中断。中断处理器对处于等待状态的进程解月阻塞，并对其重新调度使其运行。相反，异步 I/O 接口允许进程在产生一个 I/O 操作后继续执行。当 I/O 完成后，驱动必须通知请求 I/O 的进程（例如，可以调用与那个进程相关的事件处理程序函数）。

> 当使用同步 I/O 接口时，进程被阻塞直到操作完成。当使用异步 I/O 接口时，进程继续执行操作并在操作完成时收到通知。

每一种方法都有优点。当程序员需要对重叠的 I/O 和计算进行控制时，异步接口很有用。同步的方法在编程的简便性上有优势。

另一个设计问题是关于数据的格式和传输的尺寸。这里涉及两个问题。首先，数据以块还是字节进行传输？其次，一次单独的操作能传输多少数据？据观察，有些设备以单数据字节传输，有些设备以不同大小的数据块传输（比如网络数据包或文本行），另一些设备以固定大小的数据块进行传输。由于通用操作系统必须处理各种 I/O 设备，所以 I/O 接口可能同时需要单字节传输以及多字节传输。

最后一个设计问题是驱动提供的参数和它们释个别操作的方式。例如，进程需要在磁盘上指定位置并重复请求下一个磁盘块吗？或者进程需要在每次请求中指定一个块编号吗？Xinu 示例设备驱动程序描述了如何使用这些参数。

关键的思想是：

> 在现代操作系统中，I/O 接口和设备驱动程序用来隐藏设备的细节并为程序员提供便捷的、高层抽象。

14.4 I/O 接口的一个示例

我们的示例系统包含了一个有 9 个抽象 I/O 操作的 I/O 子系统，这些 I/O 操作适用于所有的输入和输出，这些操作是从 UNIX 操作系统中的 I/O 操作派生出来的。图 14-3 列出了这些操作及其用途。

[242]

操作	用途
close	终止使用某个设备
control	执行操作而不是传输数据
getc	输入单字节数据
init	在系统启动时初始化设备
open	使设备进入使用的准备状态
putc	输出单字节数据
read	输入多字节数据
seek	移到指定的数据位置（通常是磁盘）
write	输出多字节数据

图 14-3 Xinu 使用的一组抽象 I/O 接口操作

14.5 打开－读－写－关闭范式

类似于许多操作系统中的编程接口，Xinu 的示例 I/O 接口也遵循打开－读－写－关闭（open-read-write-close）范式。也就是说，在执行 I/O 前，进程必须打开（open）一个特定的设备。一旦设备已经打开，设备就允许进程调用读（read）来获取输入或者调用写（write）来发送输出。最后，一旦进程结束使用该设备，进程就调用关闭（close）来终止设备的使用。

> 打开－读－写－关闭范式要求进程在使用设备前打开（open）设备并在使用完成后关闭（close）设备。

open 和 close 允许操作系统管理需要独占使用的设备，在数据传输过程中准备设备，在传输结束后终止该设备的使用。例如，如果一个设备在没有使用的情况下需要切断电源或者进入待机状态，那么关闭该设备是非常有用的。read 和 write 两个操作负责处理数据传输，与主存中的缓存区之间进行数据字节的收发。getc 和 putc 传输单字节（通常是一字符）。control 允许程序控制某个设备或设备驱动器（比如，检查打印机的作业或者在无线频率中选择通道）。seek 是 control 的特例，它能够随机访问存储设备（如磁盘）的特定位置。最后，init 在系统启动时初始化设备及其驱动。

[243]

考虑操作如何应用到控制台窗口上：getc 从键盘读入下一个键入的字符，putc 在控制台窗口上显示字符，write 能够在一次调用中显示多个字符，read 能够读入特定数目的字符（或者所有输入的字符，取决于它的参数）。最后，control 允许程序改变驱动器的参数来控制诸如当密码输入时系统是否终止输出字符等这类情况。

14.6 绑定 I/O 操作和设备名

像 read 这样的抽象操作怎样在底层硬件设备上起作用呢？答案在于操作绑定。当进程调用一个高层操作时，操作系统必须将该调用映射到设备驱动器函数上。例如，如果进程在控制台设备上调用了 read，那么操作系统就将这个调用传递给实现 read 的控制台设备驱动器。为了这样做，操作系统对程序进程隐藏了硬件和设备驱动的细节并提供设备的一个抽象版本。通过对键盘和显示器上的窗口使用一个单独的抽象设备，操作系统可以对程序隐瞒底层硬件包含两个不同设备的事实。此外，操作系统能够通过为不同供应商提供的硬件提供相同的高层抽象来隐藏设备细节。

> 操作系统创建一个虚拟 I/O 环境——进程只能够通过接口和设备驱动器提供的抽象察觉到外围设备的存在。

除了将抽象的 I/O 操作映射到驱动程序外，操作系统还必须将设备名映射到具体设备上。这种映射有许多不同的方法。早期的系统要求程序员在源代码中嵌入设备名。后来的系统在程序中用小的整数来识别设备，并允许命令解释器在程序启动时将每个整数链接到具体的设备。许多现代系统将设备嵌入在文件名字空间的层次结构中，允许程序对设备使用符号名。

早期和后期的绑定方法都有它们的优点。一般来说，在后期绑定模式下，操作系统安装到运行时才将抽象设备名绑定到真实的物理设备，并将一组抽象操作绑定到设备驱动函数上。这种方式非常灵活。可是，这些后期绑定的系统会产生较多的计算开销，使它们无法小型的实入式系统中不可用。另外一种极端则是早期绑定：在编写应用程序时就要指定设备信息。因此，I/O 设备的本质就是在达到所要求性能的同时，使得绑定机制灵活性最大化。

我们的示例系统使用了适用于典型的小型嵌入式系统的方法：在操作系统编译重就指定设备信息。 244
对于每个设备，操作系统准确地知道每个抽象 I/O 操作对应于哪个驱动器函数。另外操作系统也知道每个抽象设备对应于哪个底层硬件设备。因此，无论何时安装一个新设备或卸载已有的设备，操作系统都必须重新进行编译。由于不包含具体的设备信息，所以程序代码能够很方便地从一个系统移植到另外一个系统。例如，一个仅仅在 CONSOLE 串口上处理 I/O 操作的应用程序，无能够在任何一个提供 CONSOLE 设备和相关驱动器的 Xinu 系统上运行，而且独立于物理设备硬件和中断结构。

14.7　Xinu 中的设备名

在 Xinu 中，系统的设计者必须在系统配置时指定一组抽象设备。配置程序为每个设备名指派一个唯一的整数值，这个整数值就是设备描述符（device descriptor）。例如，如果设计者命名一个设备为 CONSOLE，配置程序就分配描述符 0。配置程序则生成一个头文件，该头文件包含每个名字的#define 语句。因此，一旦包含这个头文件，程序员就能够在代码中引用 CONSOLE。例如，如果 CONSOLE 分配了描述符 0，那么调用：

```
read(CONSOLE,buf,100);
```

就等同于：

```
read(0,buf,100);
```

Xinu 为每个设备名仅仅于静态绑定。在操作系统编译前，在配置阶段，每个设备名与一个整数描述符绑定。

14.8　设备转换表概念

每当进程调用高层 I/O 操作（比如 read 和 write）时，操作系统必须将这次调用转发给适当的驱动器函数。Xinu 使用称为设备转换表（device switch table）的数组来提高实现效率。分配给设备的整数描述符是设备转换表的索引。为了便于理解，可以想象设备转换表是一个 2 维数组。从概念上来看，数组的每行对应于一个设备，每列对应于一次抽象操作。数组中的项指定了处理操作要调用的驱动函数。

例如，假设一个系统包含了如下 3 个设备：

- CONSOLE：发送和接收字符的串行设备。
- ETHER：以太网接口设备。
- DISK：硬盘驱动器。 245

图 14-4 描述了设备转换表的一部分。表中每一行表示一个设备，每一列表示一次 I/O 操作。表中的项表示处理操作的驱动器函数，该操作涉及的设备在行给出，操作在列给出。

	open	close	read	write	getc	
CONSOLE	conopen	conclose	conread	conwrite	congetc	
ETHER	ethopen	ethclose	ethread	ethwrite	ethgetc	...
DISK	dskopen	dskclose	dskread	dskwrite	dskkgetc	
			⋮			

图 14-4　设备转换表的概念结构，每一行表示一个设备，每一列表示一次抽象操作

例如，假设进程在 CONSOLE 设备上调用 write 操作。操作系统定入表中 CONSOLE 设备对应的行，找到对应于 write 操作的列，然后调用该位置的函数：conwrite。

本质上，设备转换表的每行定义了 I/O 操作如何应用到单个设备上。这就意味着 I/O 语义因设备

的不同而不同。例如，当设备是DISK时，一次read操作传输512字节的数据块，但是当设备是CONSOLE时，read传输用户输入的一行字符。

设备转换表的最重要方面是它在多个物理设备间定义了统一的抽象。例如，假设一台计算机有两块磁盘，一个扇区大小为1KB，另一个扇区大小为4KB，两块磁盘的驱动器能够为应用程序提供相同的接口，同时隐藏了底层硬件的不同之处。这就意味着，驱动器总是能够传输4KB的数据给用户，并且将每次传输操作转换成4次1KB的磁盘传输。

14.9 设备和共享驱动的多个副本

假设一台计算机有两个使用相同硬件的设备，那么操作系统需要两个不同的设备驱动副本吗？不需要。每个驱动程序只保持一个副本，且系统使用参数来区分两个设备。除了图14-4中的函数外，参数也可以保存在设备转换表的列中。例如，如果系统有2个以太网接口，每个以太网接口在
246 设备转换表中都有一行。那么两行中大多数的项都是相同的。然而，有一列将为每个设备指定唯一的控制和状态寄存器（Control and Status Register，CSR）地址。当系统调用驱动函数时，它将传入一个参数，该参数包含了指向该设备的设备转换表中的行的指针。因此，驱动函数就可以将操作应用到正确的设备上。

> 操作系统为每一类设备维护一个驱动副本，同时操作系统提供参数让驱动程序区分物理硬件的多个副本，而不是为每个物理设备创建一个设备驱动。

在示例代码中，设备转换表的名字为devtab，它能帮助我们了解其中的细节。结构体dentry定义了表中每项的格式，它的声明可以在conf.h[⊖]文件中找到。

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

/* Device switch table declarations */

/* Device table entry */
struct  dentry  {
        int32   dvnum;
        int32   dvminor;
        char    *dvname;
        devcall (*dvinit) (struct dentry *);
        devcall (*dvopen) (struct dentry *, char *, char *);
        devcall (*dvclose)(struct dentry *);
        devcall (*dvread) (struct dentry *, void *, uint32);
        devcall (*dvwrite)(struct dentry *, void *, uint32);
        devcall (*dvseek) (struct dentry *, int32);
        devcall (*dvgetc) (struct dentry *);
        devcall (*dvputc) (struct dentry *, char);
        devcall (*dvcntl) (struct dentry *, int32, int32, int32);
        void    *dvcsr;
        void    (*dvintr)(void);
        byte    dvirq;
};

extern  struct  dentry  devtab[]; /* one entry per device */

/* Device name definitions */

#define CONSOLE     0       /* type tty     */
#define NOTADEV     1       /* type null    */
#define ETH0        2       /* type eth     */
```

⊖ 文件conf.h还包含定义了整个系统中的全量的#define语句。第24章描述了Xinu配置并且解释了这些常量是如何出现的。

```
#define RFILESYS    3       /* type rfs      */
#define RFILE0      4       /* type rfl      */
#define RFILE1      5       /* type rfl      */
#define RFILE2      6       /* type rfl      */
#define RFILE3      7       /* type rfl      */
#define RFILE4      8       /* type rfl      */
#define RFILE5      9       /* type rfl      */
#define RDISK       10       /* type rds      */
#define LFILESYS    11       /* type lfs      */
#define LFILE0      12       /* type lfl      */
#define LFILE1      13       /* type lfl      */
#define LFILE2      14       /* type lfl      */
#define LFILE3      15       /* type lfl      */
#define LFILE4      16       /* type lfl      */
#define LFILE5      17       /* type lfl      */
#define TESTDISK    18       /* type ram      */
#define NAMESPACE   19       /* type nam      */

/* Control block sizes */

#define Nnull   1
#define Ntty    1
#define Neth    1
#define Nrts    1
#define Nrfl    6
#define Nrds    1
#define Nram    1
#define Nlfs    1
#define Nlfl    6
#define Nmem    1

#define DEVADDRNAME 24
#define NDEVS 20


/* Configuration and Size Constants */

#define NPROC           100         /* number of user processes             */
#define NSEM            100         /* number of semaphores                 */
#define IRQ_TIMER       IRQ_HW5     /* Timer IRQ is wired to hardware 5     */
#define IRQ_ATH_MISC    IRQ_HW4     /* Misc. IRQ is wired to hardware 4     */
#define MAXADDR         0x02000000 /* 32 MB of RAM                         */
#define CLKFRQ          200000000  /* 200 MHz clock                        */
#define FLASH_BASE      0xBD000000 /* Flash ROM device                     */

#define LF_DISK_DEV     TESTDISK
```

247
↓
248

devtab 中的每项对应于一个设备。表中的项为设备指定了构成驱动器函数的地址、设备 CSR 地址和驱动使用的其他信息。dvinit、dvopen、dvclose、dvread、dvwrite、dvseek、dvgetc、dvputc 和 dvcntl 字段保存了对应高层操作的驱动程序地址。dvminor 字段包含了设备控制块数组的整数索引。如果底层硬件包括了一组相同的硬件设备，那么次设备编号就是必需的——次编号意味着驱动可以为每个设备建立一个单独的控制块项。dvcsr 字段包含了设备的硬件 CSR 地址。控制块数组中的每项保存与特别的设备和驱动器实例相关的额外信息。控制块的内容取决于设备，但可能包含输入或输出缓存区、设备状态信息（例如，无线网络设备当前是否与另一个无线设备通信）、统计信息（例如，自系统启动后网络设备接收数据的总量）。

14.10 高层I/O操作的实现

由于设备转换表将高层I/O操作从底层细节中分离出来，所以它允许高层函数在任何设备驱动编写完之前创建。这种策略的一个主要优点是程序员可以不需要具体的硬件设备就能建立I/O系统。

我们的示例系统为每个高层的操作设计了一个函数。该系统包含了open、close、read、write、getc、putc等函数。然而，这些高层read函数并不执行I/O操作。相反，每一个高层I/O函数间接地（indirectly）执行I/O操作：函数使用设备转换表找到并调用适当的低层设备驱动程序来执行请求的任务。

read和write等高层函数为具体的设备间接调用低层驱动器函数，而不是直接执行I/O操作。

249 研究代码将有助于理解概念。下面为read.c文件中的read函数：

```
/* read.c - read */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  read  -  read one or more bytes from a device
 *------------------------------------------------------------------------
 */
syscall read(
          did32         descrp,         /* descriptor for device        */
          char          *buffer,        /* address of buffer            */
          uint32        count           /* length of buffer             */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvread) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

函数read的参数由设备描述符、缓存区地址和要读取的最大字节数组成。read使用descrp作为设备描述符，它是devtab表的索引，通过它从并分配设备转换表获得地址指针devptr。return语句调用了底层设备驱动函数并将结果返回给调用read的函数。代码：

```
(*devptr->dvread)(devptr,buffer,count)
```

执行间接函数调用。也就是说，通过传入3个参数：devptr（devtab表项的地址），buffer（缓存区地址）和count（要读取的字符数）该代码调用设备转换表项中dvread字段给出的驱动器
250 函数。

14.11 其他高层I/O函数

其他高层传输和控制函数与read函数的操作方式一致：它们利用设备转换表来选择并调用合适的低层驱动函数，并将结果返回给调用者。下面是各个函数的实现代码。

```
/* control.c - control */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  control  -  control a device or a driver (e.g., set the driver mode)
 *------------------------------------------------------------------------
 */
syscall control(
	  did32		descrp,		/* descriptor for device	*/
	  int32		func,		/* specific control function	*/
	  int32		arg1,		/* specific argument for func	*/
	  int32		arg2		/* specific argument for func	*/
	)
{
	intmask		mask;		/* saved interrupt mask		*/
	struct dentry	*devptr;	/* entry in device switch table	*/
	int32		retval;		/* value to return to caller	*/

	mask = disable();
	if (isbaddev(descrp)) {
		restore(mask);
		return SYSERR;
	}
	devptr = (struct dentry *) &devtab[descrp];
	retval = (*devptr->dvcntl) (devptr, func, arg1, arg2);
	restore(mask);
	return retval;
}
/* getc.c - getc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getc  -  obtain one byte from a device
 *------------------------------------------------------------------------
 */
syscall getc(
	  did32		descrp		/* descriptor for device	*/
	)
{
	intmask		mask;		/* saved interrupt mask		*/
	struct dentry	*devptr;	/* entry in device switch table	*/
	int32		retval;		/* value to return to caller	*/

	mask = disable();
	if (isbaddev(descrp)) {
		restore(mask);
		return SYSERR;
	}
	devptr = (struct dentry *) &devtab[descrp];
	retval = (*devptr->dvgetc) (devptr);
	restore(mask);
	return retval;
}

/* putc.c - putc */
```

```
#include <xinu.h>

/*------------------------------------------------------------------------
 *  putc  -  send one character of data (byte) to a device
 *------------------------------------------------------------------------
 */
syscall putc(
	  did32		descrp,		/* descriptor for device	*/
	  char		ch		/* character to send		*/
	)
{
	intmask		mask;		/* saved interrupt mask		*/
	struct dentry	*devptr;	/* entry in device switch table */
	int32		retval;		/* value to return to caller	*/

	mask = disable();
	if (isbaddev(descrp)) {
		restore(mask);
		return SYSERR;
	}
	devptr = (struct dentry *) &devtab[descrp];
	retval = (*devptr->dvputc) (devptr, ch);
	restore(mask);
	return retval;
}

/* seek.c - seek */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  seek  -  position a random access device
 *------------------------------------------------------------------------
 */
syscall seek(
	  did32		descrp,		/* descriptor for device	*/
	  uint32	pos		/* position			*/
	)
{
	intmask		mask;		/* saved interrupt mask		*/
	struct dentry	*devptr;	/* entry in device switch table */
	int32		retval;		/* value to return to caller	*/

	mask = disable();
	if (isbaddev(descrp)) {
		restore(mask);
		return SYSERR;
	}
	devptr = (struct dentry *) &devtab[descrp];
	retval = (*devptr->dvseek) (devptr, pos);
	restore(mask);
	return retval;
}
/* write.c - write */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  write  -  write one or more bytes to a device
```

```
 *------------------------------------------------------------------------
 */
syscall write(
          did32         descrp,         /* descriptor for device        */
          char          *buffer,        /* address of buffer            */
          uint32        count           /* length of buffer             */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvwrite) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

用户进程可以使用上述函数来访问 I/O 设备。另外，系统还提供一个高层 I/O 函数 init，这个函数只供操作系统使用。每次系统启动时，操作系统就调用各个设备的 init 函数。与其他 I/O 函数一样，init 函数也使用设备转换表来调用合适的低层驱动函数。每个驱动内部的初始化函数能够初始化硬件设备，如果有必要的话，也能够初始化驱动所使用的数据结构（如缓冲区和信号量）。后面我们将看到一些驱动初始化的例子。目前，理解 init 函数与其他 I/O 函数具有相同的实现方式就足够了：

```
/* init.c - init */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  init  -  initialize a device and its driver
 *------------------------------------------------------------------------
 */
syscall init(
          did32         descrp          /* descriptor for device        */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvinit) (devptr);
        restore(mask);
        return retval;
}
```

14.12　打开、关闭和引用计数

open 和 close 函数与其他 I/O 函数的操作方式非常相似，也使用设备转换表来调用合适的驱动函

数。之所以使用 open 和 close 函数，原因在于可以用它们建立设备的所属关系或者为一个将要使用的设备提供准备工作。例如，如果一个设备需要互斥访问，那么 open 函数将阻塞后续用户直到设备变得空闲为止。另外，操作系统在设备处于不使用状态时通过使设备保持空闲来节约资源。虽然设计者可以通过使用 control 函数来启动或者关闭磁盘，但是 open 和 close 函数显得更为方便。因此，当进程调用 open 函数时，磁盘就会启用，而当进程调用 close 函数时，磁盘就会关闭。

虽然一个小型嵌入式系统可能会选择在进程调用一个设备的 close 函数时让磁盘处于休眠状态，但是由于在较大型系统中多个进程能够同时使用一台设备，所以需要一套更为复杂的机制。大部分设备驱动器都引入了一类称为引用计数的技术，即驱动维护一个整型变量来记录占用当前设备的进程数。在初始化过程中，这个引用计数设置为 0。一旦进程调用 open 函数，驱动器就将引用计数加 1；当进程调用 close 函数时，驱动器就将引用计数减 1。当这个计数为 0 时，驱动器就关闭设备。

251
~
255

open 和 close 函数的实现方法和其他上层 I/O 函数的实现方法相同：

```
/* open.c - open */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  open  -  open a device (some devices ignore name and mode parameters)
 *------------------------------------------------------------------------
 */
syscall open(
        did32           descrp,         /* descriptor for device        */
        char            *name,          /* name to use, if any          */
        char            *mode           /* mode for device, if any      */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvopen) (devptr, name, mode);
        restore(mask);
        return retval;
}

/* close.c - close */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  close  -  close a device
 *------------------------------------------------------------------------
 */
syscall close(
        did32           descrp          /* descriptor for device        */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct  dentry  *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */
```

```
        mask = disable();
        if (isbaddev(devptr)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[devnum];
        retval = (*devptr->dvclose) (devptr);
        restore(mask);
        return retval;
}
```

14.13　devtab 中的空条目和错误条目

I/O 函数的操作方式带来一个有趣的问题。一方面，上层函数，比如 read 和 write，不检查 devtab 中的条目是否有效而直接使用。因此，对于每个设备的每个 I/O 操作都需要提供一个函数。另一方面，一个操作可能并不对所有的设备都有意义。例如，seek 就不能在串行设备上使用，而 gete 对于以数据包方式传输的网络设备也没有意义。而且，设计者可以选择忽略特定设备的一些操作（比如，选择让 CONSOLE 设备一直处于打开状态，这样 close 操作就不能启到作用）。

对于那些没有意义的操作应该在 devtab 中赋予什么值呢？可以使用如下两种程序来描述那些在 devtab 中没有的驱动器函数的条目：

- ionull——不执行任何动作返回 OK。
- ioerr——不执行任何动作返回 SYSERR。

按照惯例，那些值为 ioerr 的条目不应该被调用，因为它们意味着非法操作。对于那些没有必要但是又无害的操作（比如，打开终端设备），应该使用 ionull 函数。这两个函数的代码实现是不重要的：

256
·
257

```
/* ionull.c - ionull */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ionull  -  do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ionull(void)
{
        return OK;
}

/* ioerr.c - ioerr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ioerr  -  return an error status (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ioerr(void)
{
        return SYSERR;
}
```

14.14　I/O 系统的初始化

如何初始化设备转换表？如何安装驱动器函数？在大型复杂操作系统中，设备驱动是动态管理的。因此，当用户插入一个新设备时，操作系统不需要重新启动就可以识别这个设备并为它寻找和安装一个合适的驱动。

一个小型嵌入式系统在二级存储器内没有可用的驱动集合，也可能没有足够的计算资源在运行时安装驱动。因此，大部分的嵌入式系统使用静态设备配置文件，在这个文件中，设备集合和设备驱动集合在系统编译时就确定了。Xinu使用的就是静态方法，它要求系统的设计者指定设备集合和构成每一个驱动的底层驱动函数的集合。不需要程序员以式地声明静态设备转换表。然而，它使用一个单独的应用程序来读取配置文件并且生成一个C文件，这个C文件包含了一个所有字段都有初始值的
[258] devtab的声明。

小型嵌入式系统使用静态设备定义，在这个定义中，设计者指定设备集合和每一个设备的驱动函数集合。配置文件程序能够生成代码，这段代码用来为设备转接表的每个字段赋值。

conf.c文件包含了一个由配置文件程序生成的C代码的例子。目前，它已经足以检测devtab中的每一个条目并观察每个字段是如何初始化的。

```
/* conf.c (GENERATED FILE; DO NOT EDIT) */

#include <xinu.h>


extern  devcall ioerr(void);
extern  devcall ionull(void);

/* Device independent I/O switch */

struct  dentry  devtab[NDEVS] =
{
/**
 * Format of entries is:
 * dev-number, minor-number, dev-name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, control,
 * dev-csr-address, intr-handler, irq
 */

/* CONSOLE is tty */
        { 0, 0, "CONSOLE",
          (void *)ttyInit, (void *)ionull, (void *)ionull,
          (void *)ttyRead, (void *)ttyWrite, (void *)ioerr,
          (void *)ttyGetc, (void *)ttyPutc, (void *)ttyControl,
          (void *)0xb8020000, (void *)ttyInterrupt, 11 },

/* NOTADEV is null */
        { 1, 0, "NOTADEV",
          (void *)ionull, (void *)ionull, (void *)ionull,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 },
/* ETHERO is eth */
        { 2, 0, "ETHERO",
          (void *)ethInit, (void *)ethOpen, (void *)ioerr,
          (void *)ethRead, (void *)ethWrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ethControl,
          (void *)0xb9000000, (void *)ethInterrupt, 4 },

/* RFILESYS is rfs */
        { 3, 0, "RFILESYS",
          (void *)rfsInit, (void *)rfsOpen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
```

```
        (void *)ioerr, (void *)ioerr, (void *)rtsControl,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE0 is rfl */
        { 4, 0, "RFILE0",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE1 is rfl */
        { 5, 1, "RFILE1",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE2 is rfl */
        { 6, 2, "RFILE2",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE3 is rfl */
        { 7, 3, "RFILE3",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE4 is rfl */
        { 8, 4, "RFILE4",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RFILE5 is rfl */
        { 9, 5, "RFILE5",
        (void *)rflInit, (void *)ioerr, (void *)rflClose,
        (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
        (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },

/* RDISK is rds */
        { 13, 0, "RDISK",
        (void *)rdsInit, (void *)rdsOpen, (void *)rdsClose,
        (void *)rdsRead, (void *)rdsWrite, (void *)ioerr,
        (void *)ioerr, (void *)ioerr, (void *)rdsControl,
        (void *)0x0, (void *)ionull, 0 },

/* LFILESYS is lfs */
        { 11, 0, "LFILESYS",
        (void *)lfsInit, (void *)lfsOpen, (void *)ioerr,
        (void *)ioerr, (void *)ioerr, (void *)ioerr,
        (void *)ioerr, (void *)ioerr, (void *)ioerr,
        (void *)0x0, (void *)ionull, 0 },
```

```
/* LFILE0 is lfl */
        { 12, 0, "LFILE0",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE1 is lfl */
        { 13, 1, "LFILE1",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE2 is lfl */
        { 14, 2, "LFILE2",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },
/* LFILE3 is lfl */
        { 15, 3, "LFILE3",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE4 is lfl */
        { 16, 4, "LFILE4",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE5 is lfl */
        { 17, 5, "LFILE5",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* TESTDISK is ram */
        { 18, 0, "TESTDISK",
          (void *)ramInit, (void *)ramOpen, (void *)ramClose,
          (void *)ramRead, (void *)ramWrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* NAMESPACE is nam */
        { 19, 0, "NAMESPACE",
          (void *)namInit, (void *)namOpen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 }
};
```

14.15 观点

设备独立的I/O是现在主流计算不可分割的一部分，它的优势非常明显。然而，仅在使用设备独

立的 I/O 上达成共识并独立初步的原语就花费了计算机界十年的时间。由于每个程序语言定义的 I/O 抽象集合不同也导致了很多的争议。例如，FORTRAN 语言使用设备号，并要求一个机制来把每个设备号绑定到 I/O 设备或者文件。因为每一种语言都实现了大量的代码，所以操作系统设计者想要兼容所有语言。这里的问题是：我们已经选择了最好的设备独立的 I/O 函数集，还是我们只是习惯于使用它们而没有寻找更好的替代品？ 259 ↓ 262

14.16　总结

操作系统隐藏了外部设备的细节，提供了抽象集合和用于执行 I/O 操作的设备独立的函数。本书示例系统使用九个 I/O 抽象函数：open、close、control、gete、pute、read、write、seek 和一个初始化函数，init。在我们的设计中，每个 I/O 原语都是同步操作的，直到请求得到满足（例如，read 函数延迟调用过程直到数据已经到达）。

Xinu 系统为每一个设备定义了抽象设备名（比如，CONSOLE），并给设备分配一个唯一的整型设备描述符。在运行时系统使用设备转换表把描述符绑定到特定的设备上。从概念上来讲，设备转换表中一行对应一个设备，一列对应一个抽象 I/O 操作，附加的列指向设备的控制块，次级设备号用来区分一台物理设备的多份副本。有些上层 I/O 函数，比如 read 或者 write，使用设备转换表来调用设备驱动器函数来完成指定设备要求的操作。单独的驱动器可以中断一个特定设备上的调用。如果一个操作在一个设备上是无意义的话，那么设备转换表应该为其配置 ioncll 或 ioerr 函数。

练习

14.1　识别 Linux 系统上所有抽象 I/O 操作。

14.2　对于一个使用异步 I/O 的系统，通过观察一个 I/O 操作完成时是否会唤醒一个运行的程序来识别这种异步机制。阐述这两种机制哪一种更容易实现，同步还是异步？

14.3　本章讨论了两种独立的绑定：通过设备名（比如 CONSOLE）绑定描述符（比如，0）和通过设备描述符来绑定硬件设备。解释 Linux 系统如何进行这两种绑定。

14.4　考察示例代码中有关设备名的实现，解释是否有可能编写一个程序允许用户输入设备名（比如 CONSOLE）来打开设备，为什么？

14.5　假设在调试过程中，你怀疑进程正在错误地调用某些上层 I/O 函数（比如，在 seek 操作无意义的设备上调用 seek），应该如何快速修改你的代码来中断这样的错误并显示错误进程的进程标识符？（这种修改不需要重新编译源代码。） 263

14.6　解释在本章中提到的抽象 I/O 函数是否已经可以满足所有的 I/O 操作？（提示：考虑 UNIX 系统中的 socket 函数。）

14.7　Xinu 系统把设备子系统定义为最基本的 I/O 抽象并把文件也并入了设备系统。UNIX 系统定义文件系统是最基本的抽象而把设备也归为文件系统。请比较这两种方式的差异并列举各自的优点。 264

设备驱动示例

现在要找到一个品格和风度皆佳的司机（驱动）是非常困难的。

——佚名

15.1 引言

本章探索 I/O 系统中的通用结构，其中包括中断处理及实时时钟管理。前面的第 14 章介绍了 I/O 子系统的组成结构、I/O 抽象操作集以及应用设备转换表的高效实现。

本章继续探索 I/O 系统。在本章中，我们将阐述驱动是如何独立于底层硬件来定义高层抽象的 I/O 服务。此外，本章还详细描述在概念上设备驱动上半部及下半部的划分，并解释这两半部分如何共享数据结构（如缓冲区），以及如何交互。最后，本章将给出一个异步字符串行设备驱动的案例。

15.2 tty 抽象

Xinu 使用 tty 这个术语来指代字符串行设备的接口抽象，这些字符串行设备包括串行接口、键盘以及文本窗体[⊖]。概括来说，tty 设备支持双向通信：进程可以向输出端发送字符，并从输入端接收字符。
267 尽管底层串行硬件机制单纯地处理输入输出，但是 tty 抽象允许将两者连接起来。举例来说，我们的 tty 设备支持字符回显操作，即驱动的输入端可以配置为将每一个输入字符的拷贝传输到输出端。当用户在键盘上输入时期望能在屏幕上同时看到所输入的字符时，回显操作就非常重要。

tty 抽象阐明许多设备驱动的重要特征：运行时可以选择多种模式。我们的 tty 驱动提供三种不同的模式，说明驱动在将输入字符传送到应用前如何处理。图 15-1 给出这三种模式的总结及其特性。

模　式	意　义
raw	驱动在接收到字符后，不进行回显、缓存、转义、控制输出等操作，直接传输字符串
cooked	驱动缓存输入，以可读的形式回显字符，处理退格和行删除，允许提前键入，处理流控制，以及发送整行文本
cbreak	驱动处理字符转换，字符回显和流控制，但并不缓存整行文本，在字符到达时直接将其传送出去

图 15-1　tty 抽象支持的三种模式

cooked 模式旨在处理交互式的键盘输入。每当接收一个字符时，驱动回显该字符（即将一份该字符的拷贝传输到输出端）。允许用户在输入的同时看到所输入的字符。回显并不是强制性的。相反，驱动中有一个参数控制字符回显，这意味着应用可以在请求用户输入密码的时候将回显关闭。cooked 模式支持行缓存，即驱动在收集一行的所有字符之后才将其传输给读进程。因为 tty 驱动在中断时执行字符回显和其他功能，所以即便没有应用程序来读取字符，用户仍然可以提前键入（即在当前指令运行期间，用户可以输入下一条指令）。行缓存的主要优势在于可以对行进行编辑，用户可以退格或者键入一个删除整行的特殊字符，以便重新输入该行。

此外，cooked 模式还提供额外的两个功能。首先，它处理输出流控制，允许用户临时终止输出，并在之后重启输出。当流控制启用时，键入 control-s 终止输出；键入 control-q 重启输出。其次，cooked
268 模式也支持输入映射。尤其是，有些计算机或应用使用一个包含回车（cr）和换行（lf）的双字符串来表示一行的结束。另一些仅使用一个字符。cooked 模式包含一个 crlf[⊖] 参数用来控制驱动如何处理行

⊖ tty 这个名字是从早期 UNIX 操作系统继承下来的，早期 UNIX 操作系统使用 ASCII 码电传设备，包括键盘及相关的计算机[1]。

⊖ 发音为 curl-if。

的结束。当用户输入名为 ENTER 或者 RETURN 的键后，驱动就会查询该参数并决定是否给应用传送一个换行（也称为 NEWLINE）或者映射为回车和换行组成的双字符序列。

raw 模式旨在向应用提供没有预处理的输入字符。在 raw 模式中，tty 驱动仅传送字符，而不解析或者变更字符。驱动既不回显字符也不处理流控制。raw 模式在处理非交互通信时非常有用，例如，通过串行线路下载二进制文件，或者用串行设备控制传感器。

cbreak 模式提供一种介于 cooked 模式和 raw 模式之间的折中方案。在 cbreak 模式中，接收的字符会立即传送给应用，不需要累积至一行文本。因此，驱动并不缓存输入，也不支持退格、行删除操作。尽管如此，驱动仍然处理回显字符和流控制。

15.3　tty 设备驱动的组成

与大多数设备驱动一样，示例 tty 驱动也被划分为上半部和下半部。上半部包含可被应用进程调用的函数（直接从设备转换表调用）；下半部包含设备中断时调用的函数。上、下部分共享包含设备的信息、驱动的当前模式、输入/输出数据缓冲区的数据结构。通常，上半部函数从共享数据结构读取数据或者向其写入数据。它与设备硬件的交互极少。举例来说，上半部函数将输出数据放置在共享数据结构中，下半部函数访问该结构内的数据并将这些数据发送到设备。同样，下半部函数将输入数据放置在共享数据结构中，上半部函数可以从中抽取这些数据。

驱动划分的目的在一开始并不好理解。然而，我们可以看出将驱动划分为上、下两部分是功能性需求，因为这样的划分允许系统设计者将正常处理与硬件终端处理的耦合解除，并正确理解每个函数是如何调用的。这里的关键点在于：

> 当创建一个设备驱动时，程序员需要非常小心的保持上下半部的划分，因为上半部函数是由应用进程调用的而下半部函数是由中断调用的。 [269]

15.4　请求队列和缓冲区

驱动中的共享数据结构通常包含两个关键元素：

- 请求队列。
- 输入/输出缓冲区。

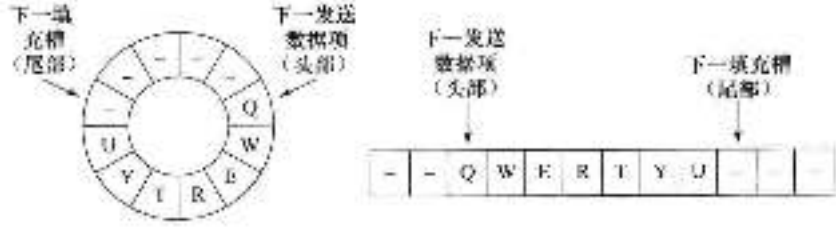
请求队列　原则上，在较上、下半部共享的数据结构中最重要的元素是上半部存放请求的队列。从概念上来讲，请求队列连接由应用指定的高层操作和运行在设备上的低层行为。每个驱动有它自己的请求集合，请求队列中元素的内容依赖于具体的设备以及所执行的操作。举例来说，发送给磁盘设备的请求指明传输的方向（读或写）、硬盘的地址和需要传输的数据量。发送给网络设备的请求指明将要通过网络传输的数据包集。我们的示例驱动由于仅包含发送输出字符和接收输入字符这两个操作，所以无需单独的请求队列。因此，输出字符队列中的字符可以看成发送请求；输入字符队列中的空格可以看成接收请求。

缓冲区　驱动使用输出缓冲区来存放将要发送给设备的数据。输出的数据项在应用将其发送出去至设备准备好接收它期间一直保留在缓冲区中。输入缓冲区则存放从设备接收的数据。输入的数据项在设备存储该项至某一适当请求期间一直保留在缓冲区之中。

缓冲区的重要性体现在如下几点：第一，驱动可以在用户适时读取数据前接收输入数据并放置在输入缓冲区中。输入缓冲区在诸如网络接口或者键盘等设备驱动中特别重要，其原因在于数据包可以在任意时刻到达，用户也可以在任意时刻输入某个键。第二，对于一个以块为单位传输数据的设备（例如硬盘）来说，即使应用只需读取块中的某个字符，操作系统也必须获取整个块。第三，缓冲区允许驱动支持并发地处理输入/输出。当进程写数据时，驱动将数据复制到输出缓冲区中，启动输出操作，并让进程继续执行。

对于每个串行设备，示例 tty 驱动使用三个环形字符缓冲区：输入缓冲区、输出缓冲区，以及回显缓冲区。回显字符保存在与正常输出不同的缓冲区中，因为回显字符具备更高的优先级。我们认为每个缓冲区都是一个概念上的队列，字符从尾部输入、从头部读出。图 15-2 阐释了环形输出缓存的概

[270] 念，并展示用内存中字节数组的一种实现。



a）用做队列的环路输出缓冲区　　b）用字节数组的实现方式

图 15-2

输出函数把即将发送的字符存储在输出缓冲区中，并返回给它的调用者。当它将字符放置在输出缓冲区后，上半部函数还必须启动设备上的输出中断。一旦设备生成输出中断，下半部函数就从输出缓冲区中提取至多16个字符，然后将字符存储在设备的先进先出输出队列上。当所有字节都传输完后，设备再次中断。因此输出会继续直至清空输出缓冲区，这时驱动停止输出，设备恢复空闲状态。

输入与输出相反。一旦接收字符，设备就产生中断，中断分配器调用下半部函数（即，ttyInterrupt）。中断处理程序将字符从设备的先进先出队列中读取出来，并存储在环路输入缓冲区中。当进程调用上半部函数读取输入时，上半部函数从输入缓冲区中读取字符。

从概念上来讲，驱动的上、下半部只通过共享的缓冲区来通信。上半部函数将输出数据放在缓冲区中，或者从缓冲区中提取输入数据。下半部函数从缓冲区提取输出数据并发送给设备，或者将设备的输入数据放在缓冲区中。总之：

上半部函数在进程和缓冲区之间传输数据，下半部函数在硬件设备和缓冲区之间传输数据

15.5 上半部和下半部的同步

在实践中，驱动的上、下半部需要处理数个共享数据结构。举例来说，如果设备空闲，上半部函数就可能需要启动输出传输。更重要的是，两个部分需要在请求队列和缓冲区中协调操作。例如，如果输出缓冲区没有空间，那么当进程试图写数据时，它将被阻塞。之后当缓冲区中的字符被传送到设备中，缓冲区又变成可用时，必须允许阻塞的进程继续执行。同样，当进程尝试从设备读数据时，如
[271] 果输入缓冲区空闲，那么进程被阻塞。之后当接收到输入数据并将数据放置在缓冲区中时，必须允许等待输入的进程继续执行。

乍一看，驱动上半部和下半部之间的同步包括两个生产者－消费者协调问题，它们可以用信号量来解决。在输出端，上半部函数生产数据，下半部函数消费数据；在输入端，下半部函数生产输入数据，下半部函数消费数据。输入并不会为生产者－消费者模型带来任何问题，可以创建信号量解决协调问题。当进程调用上半部输入函数时，进程等待输入信号量直至下半部生产输入数据项并通知该信号量。

输出则有一个难题需要解决。为了理解这个问题，回想我们对于中断处理的限制：由于中断例程可以被null进程执行，所以它并不能调用一个会将进程状态变更为除了就绪或者当前之外的函数。特别地，下半部函数不能调用wait。因此，驱动不能通过信号量实现上半部函数生产数据，下半部函数消费数据。

如何协调上半部和下半部函数的输出呢？令人惊讶的是，信号量可以很容易地解决这个问题。方法是通过变更输出信号量的目的，将调用转向wait函数。与让下半部等待上半部生产数据相反，让上半部等待缓冲区中的空间。因此，下半部不再看成消费者，相反，它是生产缓冲区中产生空间的生产者，并通知每个槽对应的信号量。总之：

信号量可以用于协调驱动的上、下半部。为了防止下半部函数被阻塞，将输出设计为由上半部函数等待缓冲区中的空间。

15.6　硬件缓冲区和驱动设计

硬件的设计可能使驱动的设计复杂化。以 E2100L 中的通用异步发送器和接收器为例，该设备包含两个板载的缓冲区，称为先进先出（FIFO）。一个处理输入字符，另一个处理输出字符。每个 FIFO 可以缓存 16 个字符。设备不会在每次字符到达时发出中断。相反，它在第一个字符到达时发出中断，但在处理中断前持续地向 FIFO 中写入字符。因此，当系统收到一个输入中断时，驱动必须不断地从 FIFO 中提取字符直至 FIFO 为空。 [272]

多个输入字符如何影响驱动的设计呢？考虑如下情况，当进程阻塞在输入信号量时，它等待输入字符的到来。理论上，一旦驱动从设备提取一个字符并将它放在输入缓冲区中，它应该通知信号量重新调度，以表明输入缓冲区中有字符可用。然而，这样做会立即导致上下文切换，让 FIFO 中的字符没有被处理。为了防止这一问题，我们的驱动使用 sched_cntl[⊖] 恰时推迟重新调度。当所有在 FIFO 中的字符都被提取出来并处理完后，驱动再次调用 sched_cntl，允许其他进程执行。

15.7　tty 控制块和数据声明

如果系统包含某个硬件设备的多个拷贝，那么操作系统只保存一份设备驱动代码，但对每个设备都创建一个独立的共享数据结构。有些系统使用术语控制块来描述共享数据结构，并声明给每个物理设备分配一个控制块。当它运行时，设备驱动函数将接收一个参数，该参数表明需要使用哪一个控制块。因此，如果一个特定的系统有三个使用同一个 tty 抽象的串行设备，那么这个操作系统只拥有一份读、写该 tty 设备的函数，但包含三份 tty 控制块的独立拷贝。

控制块存储设备信息、驱动以及请求队列。它包含缓冲区或者指向内存中缓冲区的指针[⊜]。控制块还存储上、下半部用以协调的信息。举例来说，由于示例 tty 驱动使用一个信号量来协调对输出缓冲区的访问，使用另一个信号量来协调对输入缓冲区的访问，所以 tty 控制块存储这两个信号量 ID。 [273]

文件 tty.h 中的代码包含对 tty 控制块数据结构的定义 ttyblk。

```
/* tty.h */

#define TTY_CBMINSP             20      /* min space in buffer before   */
                                        /* processes awakened to write  */
#define TTY_ECHOQSIZ            20      /* size of echo queue           */

/* Size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef TTY_CBUFLEN
#define TTY_CBUFLEN     128             /* num. chars in input queue    */
#endif
#ifndef TTY_OBUFLEN
#define TTY_OBUFLEN     64              /* num. chars in output queue   */
#endif

/* Mode constants for input and output modes */

#define TTY_INRAW       'R'             /* raw mode => nothing done     */
#define TTY_INCOOKED    'C'             /* cooked mode -> line editing  */
#define TTY_INCBREAK    'K'             /* honor echo, ecc, no line edit*/
#define TTY_OMRAW       'R'             /* raw mode => normal processing*/

struct  ttyblk {                        /* tty line control block       */
```

⊖ sched_cntl 的代码可以在 5.13 节找到。

⊜ 在某些系统中，输入/输出缓冲区必须放在内存的特定区域，使设备能够直接访问内存。

```
    char    *tyihead;               /* next input char to read      */
    char    *tyitail;               /* next slot for arriving char  */
    char    tyibuff[TY_IBUFLEN];    /* input buffer (holds one line)*/
    sid32   tyisem;                 /* input semaphore              */
    char    *tyohead;               /* next output char to xmit     */
    char    *tyotail;               /* next slot for outgoing char  */
    char    tyobuff[TY_OBUFLEN];    /* output buffer                */
    sid32   tyosem;                 /* output semaphore             */
    char    *tyehead;               /* next echo char to xmit       */
    char    *tyetail;               /* next slot to deposit echo ch */
    char    tyebuff[TY_EBUFLEN];    /* echo buffer                  */
    char    tyimode;                /* input mode raw/cbreak/cooked */
    bool8   tyiecho;                /* is input echoed?             */
    bool8   tyieback;               /* do erasing backspace on echo?*/
    bool8   tyevis;                 /* echo control chars as ^X ?   */
    bool8   tyecrlf;                /* echo CR-LF for newline?      */
    bool8   tyicrlf;                /* map '\r' to '\n' on input?   */
    bool8   tyierase;               /* honor erase character?       */
    char    tyierasec;              /* erase character (backspace)  */
    bool8   tyeof;                  /* honor EOF character?         */
    char    tyeofch;                /* EOF character (usually ^D)   */
    bool8   tyikill;                /* honor line kill character?   */
    char    tyikillc;               /* line kill character          */
    int32   tyicursor;              /* current cursor position      */
    bool8   tyoflow;                /* honor ostop/ostart?          */
    bool8   tyohold;                /* output currently being held? */
    char    tyostop;                /* character that stops output  */
    char    tyostart;               /* character that starts output */
    bool8   tyocrlf;                /* output CR/LF for LF ?        */
    char    tyifullc;               /* char to send when input full */
};
extern  struct  ttyblk ttytab[];

/* Characters with meaning to the tty driver */

#define TY_BACKSP       '\b'        /* Backspace character          */
#define TY_BELL         '\07'       /* Character for audible beep   */
#define TY_BOFCH        '\04'       /* Control-D is EOF on input    */
#define TY_BLANK        ' '         /* Blank                        */
#define TY_NEWLINE      '\n'        /* Newline -- line feed         */
#define TY_RETURN       '\r'        /* Carriage return character    */
#define TY_STOPCH       '\023'      /* Control-S stops output       */
#define TY_STRTCH       '\021'      /* Control-Q restarts output    */
#define TY_KILLCH       '\025'      /* Control-U is line kill       */
#define TY_UPARROW      '^'         /* Used for control chars (^X)  */
#define TY_FULLCH       TY_BELL     /* char to echo when buffer full*/

/* Tty control function codes */

#define TC_NEXTC        3           /* look ahead 1 character       */
#define TC_MODER        4           /* set input mode to raw        */
#define TC_MODEC        5           /* set input mode to cooked     */
#define TC_MODEK        6           /* set input mode to cbreak     */
#define TC_ICHARS       8           /* return number of input chars */
#define TC_ECHO         9           /* turn on echo                 */
#define TC_NOECHO       10          /* turn off echo                */
```

结构 ttyblk 的关键组件包括输入缓冲区 tyibuff、输出缓冲区 tyobuff 和单独的回显缓冲区 tyebuff。tty 驱动中的每个缓冲区都用字符数组实现。驱动将每个缓冲区看成一个循环链表，数组中的单元 0 好像是连接着最后一个单元。头部和尾部指针分别给出数组中下一填充地址和下一个清空地址。因此，

274 ~ 275

程序员可以通过下面这条简单的规则来记忆：

无论是输入缓冲区还是输出缓冲区，字符总是输入到尾部并从头部提取出来。

开始，头部和尾部均指向单元 0 号，但是不会存在输入/输出缓冲区是空的还是满的这种误解，因为每个缓冲区均有一个信号量表示当前缓冲区中的字符数目。信号量 tyisem 控制输入缓冲区时，非负数字 n 表示缓冲区中有 n 个字符。信号量 tyosem 控制输出缓冲区时，非负数字 n 表示缓冲区中有 n 个未填充的槽。回显缓冲区则是一个例外。我们的设计假设回显仅用在输入字符时，这时候仅有少量的字符会占据回显队列。因此，我们假设不会溢出，这说明不需要信号量来控制回显队列。

15.8　次设备号

前面提到，配置程序为系统中的每个设备分配一个唯一的设备 ID。需要注意的是，尽管系统包含使用给定抽象的多个物理设备，但给这些设备分配的设备 ID 可能并不连续。因此，如果系统包含 3 个 tty 设备，那么配置程序可能给这些设备分配的设备 ID 为 2、7、8。

我们还看到过操作系统必须为每个设备分配一个控制块。举例来说，如果系统包含 3 个 tty 设备，那么系统必须分配 3 个 tty 控制块拷贝。许多系统使用一种对给定设备控制块高效访问的技术。这些系统给每一个设备分配一个次设备号，这些次设备号是从 0 开始的整数。因此，如果系统包含 3 个 tty 设备，那么它们就会分配 0、1、2 这三个次设备号。

分配连续的次设备号如何使访问更加高效呢？次设备号可以用于控制块数组的索引。举例来说，考虑 tty 控制块是如何分配的。与 tty.h 文件说明一样，控制块放置在 ttytab 数组中。系统配置程序将 tty 设备数定义为常量 Ntty，该常量用于声明 ttytab 数组的大小。配置程序为这些 tty 设备分配从 0 ~ Ntty - 1 的次设备号。次设备号存储在设备转换表中。下半部的中断驱动例程和上半部的驱动例程均可以访问该次设备号并用它来索引 ttytab 数组。 276

15.9　上半部 tty 字符输入（ttyGetc）

ttyGetc、ttyPutc、ttyRead 和 ttyWrite 这 4 个函数是 tty 驱动上半部的主要组成部分。这些函数与 14 章中描述的高层操作 getc、putc、read 和 write 一一对应。最简单的驱动例程是 ttyGetc。其代码可以在 ttyGetc.c 中找到。

```
/* ttyGetc.c - ttyGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyGetc - read one character from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyGetc(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        char    ch;
        struct  ttyblk *typtr;          /* pointer to ttytab entry      */

        typtr = &ttytab[devptr->dvminor];

        /* Wait for a character in the buffer */

        wait(typtr->tyisem);
        ch = *typtr->tyihead++;         /* extract one character        */

        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
```

```
            typtr->tyihead = typtr->tyibuff;
        }

        if ( (typtr->tyimode == TY_INCOOKED) && (typtr->tyeof) &&
             (ch == typtr->tyeofch) ) {
            return (devcall)EOF;
        }

        return (devcall)ch;
277 }
```

当调用ttyGetc时，它首先从设备转换表中获取次设备号，并用它来索引ttytab数组以获取正确的控制块。然后它执行wait程序等待输入信号量(isem)，并阻塞直到下半部存储一个字符到缓冲区中。当wait返回时，ttyGetc从输入缓冲区中获取下一个字符并更新头指针，准备下面的读取操作。通常，ttyGetc将该字符返回给调用者。然而，在特殊情况下：如果驱动启用文件结束符并且字符与文件结束符（控制块中的tyeofch字段）匹配，ttyGetc就返回常量EOF。

15.10 通用上半部tty输入（ttyRead）

使用read操作可以在一次操作中获取多个字符。实现read操作的驱动函数ttyRead，在下面的文件ttyRead.c中。ttyRead在原理上很简单：它通过反复调用ttyGetc来获取字符。当驱动以cooked模式执行时，ttyRead返回单行输入，在NEWLINE或RETURN字符后终止；当以其他模式执行时，ttyRead读取字符但不检查行结束符。

```
/* ttyRead.c - ttyRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyRead  -  read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyRead(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to read   */
        )
{
        struct  ttyblk *typtr;          /* pointer to tty control block */
        int32   avail;                  /* characters available in buff.*/
        int32   nread;                  /* number of characters read    */
        int32   firstch;                /* first input character on line*/
        char    ch;                     /* next input character         */

        if (count < 0) {
                return SYSERR;
        }
        typtr= &ttytab[devptr->dvminor];

        if (typtr->tyimode != TY_IMCOOKED) {

                /* For count of zero, return all available characters */

                if (count == 0) {
                        avail = semcount(typtr->tyisem);
                        if (avail == 0) {
                                return 0;
                        } else {
                                count = avail;
```

```
                ;
        }
        for (nread = 0; nread < count; nread++) {
                *buff++ = (char) ttyGetc(devptr);
        }
        return nread;
}

/* Block until input arrives */

firstch = ttyGetc(devptr);

/* Check for end-of-file */

if (firstch == EOF) {
        return (EOF);
}

/* read up to a line */

ch = (char) firstch;
*buff++ = ch;
nread = 1;
while ( (nread < count) && (ch != TY_NEWLINE) &&
                (ch != TY_RETURN) ) {
        ch = ttyGetc(devptr);
        *buff++ = ch;
        nread++;
}
return nread;
}
```

终端如何执行 read 操作的语义说明了 I/O 原语如何运用到各种不同的设备和模式上。例如，使用 raw 模式的应用程序可能需要非阻塞地从输入缓冲区中读取所有可用的字符。ttyRead 不能简单地反复调用 ttyGetc，因为一旦缓冲区为空，ttyGetc 将被阻塞。为了满足非阻塞的要求，我们的驱动允许一种通常认为是非法操作的行为：它将读取零个字符的请求解释为“读取所有等待中的字符”。

ttyRead 中的代码说明了在 raw 模式中长度为 0 的请求是如何处理的：驱动使用 semcount 来获取输入信号量 tyisem 的当前计数，然后就清楚地知道可以调用 ttyGetc 多少次而不引起阻塞。

对于 cooked 模式，驱动一直阻塞，直到至少有一个字符到达。它处理文件结束字符的特殊情况，然后反复地调用 ttyGetc 来读取该行的剩余部分。

15.11　上半部 tty 字符输出（ttyPutc）

上半部的输出例程几乎与输入例程一样简单。ttyPutc 等待输出缓冲区有空间，然后在输出队列 tyobuff 存储特殊的字符中，递增队列的尾指针 tystail。文件 ttyPutc.c 包含了相关代码。

```
/* ttyPutc.c - ttyPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyPutc - write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyPutc(
        struct  dentry  *devptr,        /* entry in device switch table */
        char    ch                      /* character to write           */
        )
```

```
{
        struct  ttyblk *typtr;          /* pointer to tty control block */

        typtr = &ttytab[devptr->dvminor];

        /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyPutc(devptr, TY_RETURN);
        }

        wait(typtr->tyosem);            /* wait for space in queue */
        *typtr->tyotail++ = ch;

        /* Wrap around to beginning of buffer, if needed */
        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Start output in case device is idle */

        ttyKickOut(typtr, (struct uart_csreg *)devptr->dvcsr);

        return OK;
}
```

除了上述处理外，ttyPutc 还接受一个 tty 参数 tyocrlf，然后开始输出。当 tyocrlf 为 TRUE 时，每个 NEWLINE 就对应于 RETURN 和 NEWLINE 的组合。ttyPutc 通过递归地调用自身来输出 RETURN 字符。

15.12 开始输出（ttyKickOut）

在函数 ttyPutc 返回之前，它调用 ttyKickOut 开始输出。事实上，ttyKickOut 并没有对设备执行任何输出，因为当输出中断产生时，所有的输出已经被下半部的函数处理了。为了理解 ttyKickOut 是如何工作的，必须知道操作系统如何与硬件设备进行交互。看起来当一个字符准备输出的时候，ttyPutc 会执行以下步骤：

```
与设备交互来决定设备是否正忙；
if（设备不忙）{
送字符到设备；
} else {
当输出结束时告诉设备进行中断；
}
```

不幸的是，设备是与处理器并行运行的。所以在处理器获取了设备状态与指示设备进行中断的这段时间内，设备能够完成处理。

为了避免竞争条件，设备硬件允许操作系统不检查设备状态而请求中断。发起请求十分简单：驱动只需要对设备控制寄存器中的某一位进行置位。关键在于没有竞争条件发生，因为无论设备正在发送字符还是空闲，置位操作都将引起中断。如果设备忙，硬件将等待直到输出完成且板或缓冲区为空时才产生中断；如果设备空闲，设备立即中断。

280
281

对设备中断位进行置位仅需要一条赋值语句，相关代码可在如文件 ttyKickOut.c 中找到：

```
/* ttyKickOut.c - ttyKickOut */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyKickOut  -  'kick' the hardware for a tty device, causing it to
```

```
 *              generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyKickOut(
         struct ttyblk *typtr,          /* ptr to ttytab entry          */
         struct uart_csreg *uptr        /* address of UART's CSRs       */
        )
{
        /* Set output interrupts on the UART, which causes */
        /*   the device to generate an output interrupt    */

        uptr->ier = UART_IER_ERBFI | UART_IER_ETBEI | UART_IER_ELSI;

        return;
}
```

15.13　上半部 tty 多字符输出（ttyWrite）

tty 驱动包支持多字符输入/输出（读写操作）。文件 ttyWrite.c 中的驱动函数 ttyWrite 处理了一个字节或多个字节的输出。ttyWrite 首先检查参数 count，它表示将要写的字节数。负值是无效的，但是 0 是有效值，意味着没有字符要写。

一旦 ttyWrite 完成了对参数 count 的检查，它就会进入一个循环。在循环的每次迭代中，ttyWrite 首先从用户缓冲区中提取下一个字符，然后调用 ttyPute 将该字符发送到输出缓冲区。正如我们看到的，ttyPute 会连续执行直到输出缓冲区满，此时调用 ttyPute 将会阻塞直到有新的可用空间出现。 282

```
/* ttyWrite.c - ttyWrite, writecopy */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyWrite  -  write character(s) to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyWrite(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to write  */
        )
{

        if (count < 0) {
                return SYSERR;
        } else if (count == 0) {
                return OK;
        }

        for (; count>0 ; count--) {
                ttyPute(devptr, *buff++);
        }
        return OK;
}
```

15.14　下半部 tty 驱动函数（ttyInterrupt）

tty 驱动函数的下半部在中断发生时被调用。它包括函数 ttyInterrupt，如文件 ttyInterrupt.c 中所示。 283

```
/* ttyInterrupt.c - ttyInterrupt */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInterrupt - handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
interrupt ttyInterrupt(void)
{
        struct  dentry  *devptr;        /* pointer to devtab entry      */
        struct  ttyblk *typtr;          /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */
        int32   iir = 0;                /* interrupt identification     */
        int32   lsr = 0;                /* line status                  */

        /* For now, the CONSOLE is the only serial device */

        devptr = (struct dentry *)&devtab[CONSOLE];

        /* Obtain the CSR address for the UART */

        uptr = (struct uart_csreg *)devptr->dvcsr;

        /* Obtain a pointer to the tty control block */

        typtr = &ttytab[ devptr->dvminor ];

        /* Decode hardware interrupt request from UART device */

        /* Check interrupt identification register */

        iir = uptr->iir;
        if (iir & UART_IIR_IRQ) {
                return;
        }

        /* Decode the interrupt cause based upon the value extracted    */
        /* from the UART interrupt identification register.  Clear      */
        /* the interrupt source and perform the appropriate handling    */
        /* to coordinate with the upper half of the driver              */

        iir &= UART_IIR_IDMASK;         /* Mask off the interrupt ID */
        switch (iir) {
            /* Receiver line status interrupt (error) */

            case UART_IIR_RLSI:
                lsr = uptr->lsr;
                return;

            /* Receiver data available or timed out */

            case UART_IIR_RDA:
            case UART_IIR_RTO:

                sched_cntl(DEFER_START);

                /* For each char in UART buffer, call ttyInter_in */
```

```
        while (uptr->lsr & UART_LSR_DR) {  /* while chars avail */
                ttyInter_in(typtr, uptr);
        }

        sched_call(DEFER_SIDE);

        return;

    /* Transmitter output FIFO is empty (i.e., ready for more)  */

    case UART_IIR_THRE:
        lsr = uptr->lsr;  /* Read from LSR to clear interrupt */
        ttyInter_out(typtr, uptr);
        return;

    /* Modem status change (simply ignore) */

    case UART_IIR_MSC:
        return;
    }
}
```

记得处理程序是被间接调用的——中断分配器在设备中断时就会调用处理程序。我们后面将看到 tty 初始化例程负责安排分配器和 ttyInterrupt 之间的连接。现在只需要知道在以下的任意时刻处理函数都会被调用：设备接收到（一个或多个）字符，或者设备已经发送完所有的字符到其 FIFO 输出队列中，并准备好处理更多的字符。

从设备交换表中获取设备 CSR 基址后，ttyInterrupt 将设备 CSR 地址载入变量 uptr，随后使用 uptr 来访问该设备。关键步骤为读取中断识别寄存器，并使用该值来确定准确的中断原因。驱动感兴趣的两个原因是输入中断（有数据到达）和输出中断（如传输 FIFO 队列为空并且驱动能够发送额外的字符）。

284 / 285

15.15　输出中断处理（ttyInter_out）

输出中断处理程序容易理解：当输出中断发生时，设备已经传送完所有来自接收 FIFO 的字符并准备好处理更多的字符，ttyInterrupt 清除中断后，调用 ttyInter_out 重新开始输出。ttyInter_out 的代码可以在文件 ttyInter_out.c 中找到：

```
/* ttyInter_out.c - ttyInter_out */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInter_out  -  Handle an output on a tty device by sending more
 *                   characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_out(
          struct ttydblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{

        int32   ocount;                 /* number of output chars sent  */
                                        /*   to the UART                */
        int32   avail;                  /* available chars in output buf*/
        int32   uspace;                 /* space left in onboard UART   */
                                        /*   output FIFO                */

        /* If output is currently held, turn off output interrupts */
```

```
    if (typtr->tyshield) {
            uptr->ier = UART_IER_ERBFI | UART_IER_ELSI;
            return;
    }

    /* If echo and output queues empty, turn off output interrupts */
    if ( (typtr->tyehead == typtr->tyetail) &&
         (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
            uptr->ier = UART_IER_ERBFI | UART_IER_ELSI;
            return;
    }

    /* Initialize uspace to the size of the transmit FIFO */

    uspace = UART_FIFO_SIZE;

    /* While onboard FIFO is not full and the echo queue is */
    /* nonempty, xmit chars from the echo queue             */

    while ( (uspace>0) &&  (typtr->tyehead != typtr->tyetail) ) {
            uptr->buffer = *typtr->tyehead++;
            if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
                    typtr->tyehead = typtr->tyebuff;
            }
            uspace--;
    }

    /* While onboard FIFO is not full and the output queue  */
    /* is nonempty, xmit chars from the output queue        */

    ochars = 0;
    avail = TY_OBUFLEN - semcount(typtr->tyosem);
    while ( (uspace>0) &&  (avail > 0) ) {
            uptr->buffer = *typtr->tyohead++;
            if (typtr->tyohead >= &typtr->tybuff[TY_OBUFLEN]) {
                    typtr->tyohead = typtr->tybuff;
            }
            avail--;
            uspace--;
            ochars++;
    }
    if (ochars > 0) {
            signaln(typtr->tyosem, ochars);
    }
    return;
}
```

在开始输出之前，ttyInter_out 进行了一系列的测试。例如，如果用户输入 Ctrl+S 键，则不应该开始输出。类似地，如果回显队列和输出队列都为空，则也没有必要开始输出。为了理解 ttyInter_out 如何开始输出的，记得底层硬件上有一个能够保存多个字符的板载 FIFO。一旦确定要开始输出，ttyInter_out 就能向设备发送最多 UART_FIFO_SIZE（16）个字符。字符持续地发送直到 FIFO 满或者缓冲区为空，无论哪个事件先出现。回显队列具有最高的优先级。所以 ttyInter_out 首先从回显队列中发送字符。如果还有多余的空间，ttyInter_out 就从输出队列中发送字符。

286 ~ 287

理论上，每当将 ttyInter_out 从输出队列中删除了一个字符并将该字符发送给设备时，它都需要向输出信号量发送信号，指示缓冲区中有一个新的可用空间。但是，因为调用 signal 可能引起重新调度，所以 ttyInter_out 不能立刻调用 signal。相反，它仅仅递增变量 ochars 来对输出队列中新创建出的空间进行计数。一旦加满 FIFO（或者已经清空输出队列），ttyInter_out 就调用 signaln 来表示缓冲区中有可用

空间。

15.16　tty 输入处理（tty Inter-in）

由于板载输入 FIFO 可以包含多个字符，所以输入中断处理比输出处理更复杂。因此，ttyInterrupt[1] 使用了循环来处理输入中断：当板载 FIFO 队列不为空时，ttyInterrupt 调用 ttyInter_in 函数，ttyInter_in 函数的作用是每次从 UART 的输入 FIFO 队列中取出一个字符并进行处理。为了避免在循环终止时所有字符都从设备读出后才进行重调度，ttyInterrupt 使用 sched_cntl 函数。因此，尽管 ttyInter_in 调用了 signal 使得每个字符都可用，但是只有在所有的字符都从设备上读出后才会进行重调度。

处理单个输入字符是 tty 设备驱动最复杂的部分，因为它包含了一些很细节化的代码，比如字符回显和行编辑。ttyInter_in 函数处理过程有三种模式：raw、cbreak 和 cooked。文件 ttyInter_in.c 包含了这些代码。

```
/* ttyInter_in.c ttyInter_in, erasec, eputc, echoch */

#include <xinu.h>

local   void    erasec(struct ttycblk *, struct uart_csreg *);
local   void    echoch(char, struct ttydblk *, struct uart_csreg *);
local   void    eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
 *  ttyInter_in  --  handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_in(
          struct ttydblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        char    ch;                     /* next char from device        */
        int32   avail;                  /* chars available in buffer    */

        ch = uptr->buffer;              /* extract char. from device    */

        /* Compute chars available */

        avail = semcount(typtr->tyisem);
        if (avail < 0) {                /* one or more processes waiting*/
                avail = 0;
        }

        /* Handle raw mode */

        if (typtr->tyimode == TY_IMRAW) {
                if (avail >= TY_IBUFLEN) { /* no space => ignore input  */
                        return;
                }

                /* Place char in buffer with no editing */

                *typtr->tyitail++ = ch;

                /* Wrap buffer pointer  */
```

[1] ttyInterrupt 的代码在 15.14 节。

```
        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Signal input semaphore and return */

        signal(typtr->tyisem);
        return;
}

/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
        ch = TY_NEWLINE;
}

/* If flow control is in effect, handle ^S and ^Q */

if (typtr->tyoflow) {
        if (ch == typtr->tyostart) {        /* ^Q starts output */
                typtr->tyohold = FALSE;
                ttyKickOut(typtr, uptr);
                return;
        } else if (ch == typtr->tyostop) {  /* ^S stops output  */
                typtr->tyohold = TRUE;
                return;
        }
}

typtr->tyohold = FALSE;          /* Any other char starts output */

if (typtr->tymode == TY_DMCBREAK) {       /* Just cbreak mode  */

        /* If input buffer is full, send bell to user */

        if (avail >= TY_IBUFLEN) {
                cputs(typtr->tyifullc, typtr, uptr);
        } else {        /* Input buffer has space for this char */
                *typtr->tyitail++ = ch;

                /* Wrap around buffer */

                if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                if (typtr->tyiecho) {   /* are we echoing chars?*/
                        echoch(ch, typtr, uptr);
                }
        }
        return;

} else {        /* Just cooked mode (see common code above)     */

        /* Line kill character arrives - kill entire line */

        if (ch == typtr->tyikillc && typtr->tyikill) {
                typtr->tyitail -= typtr->tycursor;
                if (typtr->tyitail < typtr->tyibuff) {
```

```
        typtr->tyihead += TY_IBUFLEN;
}
typtr->tyicursor = 0;
eputc(TY_RETURN, typtr, uptr);
eputc(TY_NEWLINE, typtr, uptr);
return;
}

/* Erase (backspace) character */

if ( (ch == typtr->tyierasec) && typtr->tyierase) {
        if (typtr->tyicursor > 0) {
                typtr->tyicursor--;
                erasel(typtr, uptr);
        }
        return;
}

/* End of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, uptr);
        }
        *typtr->tyitail++ = ch;
        if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }

        /*  Make entire line (plus \n or \r) available */

        signaln(typtr->tyisem, typtr->tyicursor + 1);
        typtr->tyicursor = 0;   /* Reset for next line  */
        return;
}

/* Character to be placed in buffer - send bell if      */
/*      buffer has overflowed                           */

avail = semcount(typtr->tyisem);
if (avail < 0) {
        avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
        eputc(typtr->tyifullc, typtr, uptr);
        return;
}

/* NON characters recognize at beginning of line, but   */
/*      print and ignore otherwise.                     */
                if (ch == typtr->tyeofch && typtr->tyeof) {
                        if (typtr->tyiecho) {
                                echoch(ch, typtr, uptr);
                        }
                        if (typtr->tyicursor != 0) {
                                return;
                        }
                        *typtr->tyitail++ = ch;
```

```
                signal(typtr->tyisem);
                return;
        }


        /* Echo the character */

        if (typtr->tyiecho) {
                echoch(ch, typtr, uptr);
        }

        /* Insert character in the input buffer */

        typtr->tyicursor++;
        *typtr->tyitail++ = ch;

        /* Wrap around if needed */

        if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }
        return;
    }
}

/*------------------------------------------------------------------------
 *  erasel  --  erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local   void    erasel(
          struct ttycblk        *typtr, /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        char    ch;                     /* character to erase           */

        if ( (--typtr->tyitail) < typtr->tyibuff) {
                typtr->tyitail += TY_IBUFLEN;
        }

        /* Pick up char to erase */

        ch = *typtr->tyitail;
        if (typtr->tyiecho) {                       /* Are we echoing?   */
                if (ch < TY_BLANK || ch == 0177) { /* Nonprintable      */
                        if (typtr->tyevis) {        /* Visual cntl chars */
                                eputc(TY_BACKSP, typtr, uptr);
                                if (typtr->tyiebback) {  /* erase char   */
                                        eputc(TY_BLANK, typtr, uptr);
                                        eputc(TY_BACKSP, typtr, uptr);
                                }
                        }
                        eputc(TY_BACKSP, typtr, uptr);/* bypass up arrow*/
                        if (typtr->tyiebback) {
                                eputc(TY_BLANK, typtr, uptr);
                                eputc(TY_BACKSP, typtr, uptr);
                        }
                } else {  /* A normal character that is printable       */
```

```
                        eputc(TY_BACKSP, typtr, uptr);
                        if (typtr->tyeback) {  /* erase the character  */
                                eputc(TY_BLANK, typtr, uptr);
                                eputc(TY_BACKSP, typtr, uptr);
                        }
                }
        }
        return;
}

/*-------------------------------------------------------------------------
 *  echoch  --  echo a character with visual and output crlf options
 *-------------------------------------------------------------------------
 */
local   void    echoch(
          char  ch,                     /* character to echo            */
          struct ttyblk *typtr,         /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
                eputc(TY_RETURN, typtr, uptr);
                eputc(TY_NEWLINE, typtr, uptr);
        } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
                eputc(TY_UPARROW, typtr, uptr); /* print ^x             */
                eputc(ch+0100, typtr, uptr);    /* make it printable    */
        } else {
                eputc(ch, typtr, uptr);
        }
}

/*-------------------------------------------------------------------------
 *  eputc - put one character in the echo queue
 *-------------------------------------------------------------------------
 */
local   void    eputc(
          char  ch,                     /* character to echo            */
          struct ttyblk *typtr,         /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        *typtr->tyetail++ = ch;

        /* Wrap around buffer, if needed */

        if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyetail = typtr->tyebuff;
        }
        ttyKickOut(typtr, uptr);
        return;
}
```

15.16.1　raw 模式处理

raw 模式最容易理解，只有几行代码。在 raw 模式中，ttyInter_in 检查输入缓冲区是否还有空闲空间。它通过比较输入信号量的数量（即缓冲区中可用的字符数量）和缓冲区大小来完成检测。如果没有空闲空间，ttyInter_in 仅仅只是返回（即丢弃字符）。如果有空闲空间，ttyInter_in 将字符放在输入缓冲区尾部，并移动到下一个缓冲区的位置，通知输入信号量并返回。

15.16.2 cbreak模式处理

cooked和cbreak模式共享一段代码，这段代码将RETURN映射到NEWLINE，并进行输出流控制。
288 tty控制块中的字段tyoflow决定驱动是否进行流控制。如果进行流控制，那么当驱动收到字符tyostop
294 时，通过设置tyoheld为TRUE中断输出；当收到tyostart时重新开始输出。tyostart和tyostop被认为是"控制"字符，驱动不会将它们放入缓冲区。

cbreak模式检测输入缓冲区，并在缓冲区满时发送字符tyifullc。一般来说，tyifullc会像"警声"一样使控制台发出响亮的警报声。这个想法的目的是为了让打字的人能够听到警报声并停止打字，直到字符都读入，有更多的缓冲区空间被释放出来。如果缓冲区有空闲空间，程序就将字符放入其中，并在有必要时回绕指针。最后，cbreak模式调用echoch进行字符回显。

15.16.3 cooked模式处理

cooked模式与cbreak模式非常相似，唯一的区别在于它还进行行编辑。驱动不断地读入字符行到数据缓冲区中，通过变量tyicursor来标明当前行中的字符数。当接收到清除字符tyierase时，ttyInter_in将tyicursor减1，并回退一个字符，同时调用函数erasel清除显示的字符。当收到抹行符tyikillc时，ttyInter_in通过设置tyicursor为0来清除当前行，并将尾指针退回到行开始处。最后，当收到NEWLINE或者RETURN符号时，ttyInter_in调用signaln将整个键入行变为可用。它为下一行把tyicursor重置为0。这里需要注意的是，检测缓冲区是否为满的测试，会在缓冲区中给行终止符（即NEWLINE）保留一个额外的空间。

15.17 tty控制块初始化（ttyInit）

在下文所示的文件ttyInit.c中，函数ttyInit初始化tty控制块中的字段。TtyInit使用dvnq作为中断向量表的索引，将中断函数的地址赋予向量，然后将控制块初始化为cooked模式，创建输入信号量和输出信号量，并分别设置缓冲区的头指针和尾指针。在完成了驱动参数、缓冲区和中断向量的初始化后，ttyInit清空硬件上的接收缓冲区，允许接收中断，并禁止发送中断。

ttyInit将tty初始化为cooked模式，并假设它连接到了人可以使用的键盘和显示器上。选取的参数能够使视频设备而不是纸质设备工作在最好状态，这些视频设备能够在显示器上回退字符。特别地，当ttyInter_in收到清除字符tyierase时，参数tyicback使函数回显3个字符：退格－空格－退格（backspace-space-backspace）。在显示屏幕上，发送这样一个3字符序列的效果就好像用户覆盖了它们。回头看看函数ttyInter_in[⊖]，你会发现它很小心地填充了合适数目的空格符，即使用户要清除的是一个占
295 两个字符位的控制字符。

```
/* ttyInit.c - ttyInit */

#include <xinu.h>

struct  ttyblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyInit - initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  ttyblk *typtr;          /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */
```

⊖ ttyInter_in的代码可以在16.11节找到。

```
typtr = &tytab[ devptr->dvminor ];

/* Initialize values in the tty control block */

typtr->tyihead = typtr->tyitail =       /* set up input queue   */
        &typtr->tyibuff[0];             /*    as empty          */
typtr->tyisem = semcreate(0);           /* input semaphore      */
typtr->tyohead = typtr->tyotail =       /* set up output queue  */
        &typtr->tyobuff[0];             /*    as empty          */
typtr->tyosem = semcreate(TY_OBUFLEN);  /* output semaphore     */
typtr->tyehead = typtr->tyetail =       /* set up echo queue    */
        &typtr->tyebuff[0];             /*    as empty          */
typtr->tyimode = TY_IMCOOKED;           /* start in cooked mode */
typtr->tyiecho = TRUE;                  /* echo console input   */
typtr->tyieback = TRUE;                 /* honor erasing bksp   */
typtr->tyevis = TRUE;                   /* visual control chars */
typtr->tyecrlf = TRUE;                  /* echo CRLF for NEWLINE*/
typtr->tyicrlf = FALSE;                 /* map CR to NEWLINE    */
typtr->tyierase = TRUE;                 /* do erasing backspace */
typtr->tyierasec = TY_BACKSP;           /* erase char is ^H     */
typtr->tyicot = TRUE;                   /* honor cot on input   */
typtr->tyeofch = TY_EOFCH;              /* end-of-file character*/
typtr->tyikill = TRUE;                  /* allow line kill      */
typtr->tyikillc = TY_KILLCH;            /* set line kill to ^U  */
typtr->tyicursor = 0;                   /* start of input line  */
typtr->tycflow = TRUE;                  /* handle flow control  */
typtr->tycheld = FALSE;                 /* output not held      */
typtr->tystopc = TY_STOPCH;             /* stop char is ^S      */
typtr->tystart = TY_STRTCH;             /* start char is ^Q     */
typtr->tyocrlf = TRUE;                  /* send CRLF for NEWLINE*/
typtr->tyifullc = TY_FULLCH;            /* send ^G when buffer  */
                                        /*   is full            */

/* Initialize the UART */

uptr = (struct uart_csreg *)devtab[CONSOLE].dvcsr;

/* Set baud rate */

uptr->lcr = UART_LCR_8N1;           /* 8 bit char, No Parity, 1 Stop*/
uptr->fcr = 0xC0;                   /* Disable FIFO for now         */
/* OUT2 value is used to control the onboard interrupt tri-state*/
/* buffer. It should be set high to generate interrupts         */
uptr->mcr = UART_MCR_OUT2;          /* Turn on user-defined OUT2    */

/* Enable interrupts */

/* Enable UART FIFOs, clear and set interrupt trigger level     */
uptr->fcr = UART_FCR_EFIFO | UART_FCR_FRESET
          | UART_FCR_TRIGGP | UART_FCR_TRIG2;

/* Register the interrupt handler for the dispatcher */

interruptVector[devptr->dvirq] = (void *)devptr->dvintr;

/* Ready to enable interrupts on the UART hardware */

enable_irq(devptr->dvirq);
```

```
        ttyKickOut(typtr, uptr);

        return OK;
}
```

15.18 设备驱动控制

到目前为止，我们已经讨论过了驱动中的函数，如处理上层数据传输操作的函数（如 read 和 write）、处理下层输入中断和输出中断的函数以及在系统启动时的初始化函数。在第 14 章定义的 I/O 接口提供了另一种类发送操作的函数：control。control 允许应用程控制设备驱动或者控制基础的设备。在我们的驱动例子中，ttyControl 函数提供了基本的控制功能：

296
↓
297

```
/* ttyControl.c - ttyControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyControl  -  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttyControl(
         struct dentry *devptr,        /* entry in device switch table */
         int32  func,                  /* function to perform          */
         int32  arg1,                  /* argument 1 for request       */
         int32  arg2                   /* argument 2 for request       */
        )
{
        struct  ttyblk *typtr;         /* pointer to tty control block */
        char    ch;                    /* character for lookahead      */

        typtr = &ttytab[devptr->dvminor];

        /* Process the request */

        switch ( func ) {

        case TC_NEXTC:
                wait(typtr->tyisem);
                ch = *typtr->tyitail;
                signal(typtr->tyisem);
                return (devcall)ch;

        case TC_MODER:
                typtr->tyimode = TY_IMRAW;
                return (devcall)OK;

        case TC_MODEC:
                typtr->tyimode = TY_IMCOOKED;
                return (devcall)OK;

        case TC_MODEK:
                typtr->tyimode = TY_IMKBREAK;
                return (devcall)OK;

        case TC_ICHARS:
                return semcount(typtr->tyisem);

        case TC_ECHO:
                typtr->tyiecho = TRUE;
                return (devcall)OK;
```

```
        case TC_NOECHO:
                typtr->tyieche = FALSE;
                return (devcall)OK;

        default:
                return (devcall)SYSERR;
        }
}
```

TC_NEXTC 允许应用程序“向前看”（lookahead）（也就是，发现等待读取的下一个字符是什么，但是没有真正读取下一个字符）。用户可以通过 3 个控制功能（TC_MODER，TC_MODERC 和 TC_MODERK）设置 tty 驱动的模式。TC_ECHO 和 TC_NOECHO 控制字符回显，允许调用者关闭回显、接收输入，然后再打开回显。TC_ICHARS 使用户可以通过查询驱动来决定多少个字符在输入队列中等待。

善于观察的读者也许已经注意到了，在函数 ttyControl 中没有使用参数 arg1 和 arg2。然而声明它们是因为设备无关的高级 I/O 控制总是在调用 ttyControl 时提供 4 个参数。即使编译器不能在间接调用中进行类型检查，去掉参数声明也会使代码的移植性变差，且变得难懂。

15.19　观点

代码的长度揭示了设备驱动一个很重要的方面。要理解这一点，可以比较简单串行设备驱动的代码量与用于消息传送和处理同步的代码量。尽管消息传送和信号量分别提供了强有力的抽象，但是代码量仍然很小。

为什么一个很小的设备驱动包含这么多代码？毕竟，驱动只需要读和写字符串。答案在于，硬件提供的抽象层和驱动提供的抽象层是不同的。底层硬件仅仅传送字符，输出端和输入端是相互独立的。因此，硬件不会进行流控制或字符回显。而且，硬件不知道任何关于行结束符号的信息。所以，驱动需要处理更多细节问题。

虽然看上去可能会很复杂，但是本章中的示例驱动还是很小的。一个作为产品的设备驱动可能包含数万行代码，其中可能有数百个函数。可以在运行时插入的设备驱动（如 USB 设备）比静态设备驱动更复杂。所以，作为一个整体，驱动的代码兼具大规模性和复杂性。

298
~
299

15.20　总结

设备驱动是函数的集合，这些函数控制外围硬件设备。这些驱动程序分为两部分：上半部分包含由应用程序调用的函数组成，下半部分由中断发生时系统调用的函数组成。这两部分函数通过一个称为设备控制块的共享数据结构进行通信。

本章中的示例设备驱动用于控制 tty 设备。它管理硬件之上的输入和输出功能，如链接到键盘。上半部分函数实现了 read、write、getc、putc、control 操作。每一个上半部分函数通过设备转换表被间接调用。下半部分函数处理中断。在输出中断过程中，下半部分函数把回显或输出队列字符填入板载 FIFO 队列中。在输入中断过程中，下半部分函数从输入 FIFO 中抽取字符并对其进行处理。

练习

15.1　预测当两个进程同时执行 ttyRead 并同时要求处理大数量的字符时，会发生什么？做个实验测试一下，并检查结果。

15.2　Kprintf 使用能前 I/O：禁止中断，等待直到设备空闲，显示消息，然后恢复中断。如果输出缓冲区是满的并且反复调用 kprintf 显示 NEWLINE 字符时，会发生什么？并解释。

15.3　有些系统将异步设备驱动分为 3 个等级：中断级别（只传送字符给设备和从设备接收字符）、高级级别（传送字符给用户并从用户接收字符）和中等级别（实现处理字符回显、流控制、特别处理和带外信号量等或者规程）。将 Xinu tty 驱动转换为三级调度，并让进程执行中等

级层的代码。

15.4 假设两个进程同时尝试使用 CONSOLE 设备上的 write() 函数，输出会是什么？为什么？

15.5 实现一个控制函数，允许一个进程获得 CONSOLE 设备的独占使用权，以及另一个控制函数，使得进程可以释放它的使用权。

15.6 ttyControl 很低效地进行模式转换，因为它不会重置光标和缓冲区指针。重写代码改进这一点。

300 当部分地输入一行，并同时从 cooked 模式转换到 raw 模式时，会发生什么？

15.7 当连接两台计算机时，在双向上进行流控制是很有用的。修改 tty 驱动，使它包含“tandem”模式，在这个模式中，可以在输入缓冲区快要满时发送 Control-S，并在缓冲区半空时发送 Control-Q。

15.8 当用户切换 tty 设备的模式时，对于正在输入队列上的字符（在模式转换前已经被接收了），应该怎么处理？一种可能性是将队列丢弃。修改代码实现这一可能的功能。

301

DMA 设备和驱动（以太网）

与现代硬件行交道是极其困难的事情。

——詹姆斯·布坎南

16.1 引言

前面几章考虑了通用的 I/O 模式，并解释设备驱动如何协调工作。第 15 章给出了一个 tty 驱动例子来说明上层部分和底层部分是如何相互作用的。

本章将扩展对 I/O 设备驱动的讨论，并设计一种硬件设备驱动，可以对内存数据进行传输。本章使用一个以太网接口作为例子，展示 CPU 如何通知设备当前可用的缓冲区，以及设备如何在不需要处理器的情况下访问缓冲区并且在总线上传输数据。

16.2 直接内存访问和缓冲区

虽然总线一次只能传输一个字的数据，但是一个块设备，如磁盘或网络接口，需要多个字的数据传输以满足一个给定的请求。直接内存访问（DMA）的目的是实现并行性：通过增加 I/O 设备的智能，使设备在不需要 CPU 中断的情况下可以执行多次总线数据传输。因此，具有 DMA 功能的磁盘可以在 CPU 中断之前，完成整个磁盘块在内存和设备之间的传输。对于网络接口，可以在 CPU 中断之前传输整个数据包。

DMA 输出是很容易理解的。例如，考虑 DMA 输出到磁盘设备。操作系统将数据写入内存缓冲区中，通过一个写请求将缓冲区的地址传送给磁盘设备，允许 CPU 继续执行原来的作业，同时 DMA 通过总线将数据从内存缓冲区写入磁盘。一旦整个块已经从内存中读出，并写入磁盘中时，DMA 向 CPU 发送一个中断。如果还有一个的磁盘块准备输出，那么操作系统就可以启动另一次 DMA 操作传输该块。

DMA 输入以另一种方式工作。为了读取一个磁盘块，操作系统首先分配一个内存缓冲区，然后将缓冲区地址连同读操作请求发送给设备。DMA 传输开始后，操作系统将继续执行原来的作业。在 CPU 执行的同时，DMA 使用总线将数据块从磁盘传输到内存缓冲区。一旦整个块已经写入内存缓冲区，DMA 就中断 CPU。因此，每个块传输只有一次 DMA 中断。

16.3 多缓冲区和环

实际的 DMA 设备比上文提到的更复杂。在实际中，操作系统分配多个缓冲区，通过链表把它们链接在一起，传输给设备的不是缓冲区的地址，而是链表的地址。设备硬件根据链表来执行，无需等待 CPU 重新启动操作。例如，考虑一个使用 DMA 硬件作为输入的网络接口。为了从网络接收数据包，操作系统分配一个缓冲区链表，每一个缓冲区可以容纳一个网络数据包，通过链表中的地址访问网络接口设备。当一个包到达时，网络设备移动到链表上的下一个缓冲区，使用 DMA 将数据包复制到缓冲区，然后产生中断。只要链表中还有缓冲区，设备就将继续接收传入的数据包，并将数据包放置在缓冲区中。

如果一个 DMA 设备达到缓冲区链表的末尾，那么将会发生什么？有趣的是，大多数 DMA 设备从未到达链表的末尾，因为硬件使用循环链表，称为一个缓冲环。也就是说，链表上的最后一个结点指向第一个结点。链表中的每个结点包含两个值：指向缓冲区的指针和状态位，状态位表示缓冲区是否可用。在输入时，操作系统初始化链表每个结点所指向的缓冲区，并设置状态位为 EMPTY。当缓冲区满以后，DMA 硬件将状态位设置为 FULL，并产生中断。设备驱动处理中断，从所有满的缓冲区中提取数据，清除状态位并标记缓冲区为 EMPTY。一方面，如果操作系统速度足够快，那么它能够及时处

理传入的每个数据包并在下一个包到达之前将缓冲区标记为EMPTY。这样，DMA硬件将继续围绕环移动而不会遇到一个标记为FULL的缓冲区。另一方面，如果操作系统无法快速处理到达的数据包，那么设备所有的缓冲区最终将被填满，并会遇到一个标记为FULL的缓冲区。如果遍历整个环，缓冲区全被填满，DMA将设置一个错误标示（通常是溢出位），并产生中断来通知操作系统。

304

大多数DMA的输出缓冲区也采用循环链表。操作系统创建一个环，其中每个缓冲区标志位为EMPTY。当有数据包要发送时，操作系统把数据包放在下一个可用的输出缓冲区中，然后标志该缓冲区为FULL，如果设备当前没有运行就启动设备。设备移动到下一个缓冲区，提取数据包，并发送数据包。一旦启动DMA，环指针将持续移动，直到它到达环上的一个空缓冲区。因此，如果应用程序生成数据的速度足够快，那么DMA硬件将不断地传输数据包而不会出现空的缓冲区。

16.4 使用DMA的以太网驱动例子

接下来用例子来阐明上述的讨论。我们的驱动例子是为Atheros AG71xx以太网接口[⊖]编写的。虽然很多细节是针对Atheros的设备，但是处理器和设备之间的相互作用对于大多数DMA设备是相同的。

AG71xx可以执行输入和输出。处理芯片为输入和输出提供了不同的DMA引擎。也就是说，一个驱动必须创建两个环——一个环用来指向接收数据包的缓冲区，另一个环指向用于发送数据包的缓冲区。设备具有独立的寄存器用于向环传递指针，这样设备可以同时处理输入和输出。尽管操作是独立的，但是输入和输出中断使用一个中断向量。因此，当中断发生时，设备驱动软件必须与设备进行交互，以确定中断对应的是输入还是输出操作。

16.5 设备的硬件定义和常量

文件ag71xx.h为AG71xx硬件定义常量和结构。该文件包含了许多细节，有些部分可能有点难以理解。这里的定义是直接由供应商的设备手册提供的。

305

```
/* ag71xx.h  -  Definitions for an Atheros ag71xx Ethernet device        */

/* Ring buffer sizes */

#define ETH_RX_RING_ENTRIES 64  /* Number of buffers on Rx Ring */
#define ETH_TX_RING_ENTRIES 128 /* Number of buffers on Tx Ring */

#define ETH_PKT_RESERVE     64

/* Control and Status register Layout for the ag71xx  */

struct ag71xx {
    volatile uint32 macConfig1;          /* 0x000 MAC configuration 1    */

#define MAC_CFG1_TX         (1 << 0)     /* Enable Transmitter           */
#define MAC_CFG1_SYNC_TX    (1 << 1)     /* Syncronize Transmitter       */
#define MAC_CFG1_RX         (1 << 2)     /* Enable Receiver              */
#define MAC_CFG1_SYNC_RX    (1 << 3)     /* Syncronize Receiver          */
#define MAC_CFG1_LOOPBACK   (1 << 8)     /* Enable Loopback              */
#define MAC_CFG1_SOFTRESET  (1 << 31)    /* Software Reset               */

    volatile uint32 macConfig2;          /* 0x004 MAC configuration 2    */
#define MAC_CFG2_FDX        (1 << 0)     /* Enable Full Duplex           */
#define MAC_CFG2_CRC        (1 << 1)     /* Enable CRC appending         */
#define MAC_CFG2_PAD        (1 << 2)     /* Enable padding of short pkts */
#define MAC_CFG2_LEN_CHECK  (1 << 4)     /* Enable length field checking */
#define MAC_CFG2_HUGE       (1 << 5)     /* Enable frames longer than max*/
```

⊖ 指处的xx指的是拥有API的一系列产品。

```
#define MAC_CFG2_INNIBBLE  (1 << 0)    /* "nibble mode" interface type */
#define MAC_CFG2_INBYTE    (2 << 0)    /* "byte mode" interface type   */


    volatile uint32 pad00[2];
    volatile uint32 pad01[4];
    volatile uint32 pad02[4];
    volatile uint32 pad03[4];

    volatile uint32 macAddr1;          /* 0x040 MAC Address part 1     */
    volatile uint32 macAddr2;          /* 0x044 MAC Address part 2     */

    volatile uint32 fifoConfig0;       /* 0x048 MAC configuration 0    */

#define FIFO_CFG0_WTMENREQ (1 << 9)    /* Enable FIFO watermark module */
#define FIFO_CFG0_SYSRXENREQ (1 << 9)  /* Enable FIFO system Rx module */
#define FIFO_CFG0_FRFENREQ (1 << 10)   /* Enable FIFO fabric Rx module */
#define FIFO_CFG0_SYSTXENREQ (1 << 11) /* Enable FIFO system Tx module */
#define FIFO_CFG0_FTFENREQ (1 << 12)   /* Enable FIFO fabric Tx module */

    volatile uint32 fifoConfig1;       /* 0x04C MAC configuration 1    */
    volatile uint32 fifoConfig2;       /* 0x050 MAC configuration 2    */
    volatile uint32 fifoConfig3;       /* 0x054 MAC configuration 3    */
    volatile uint32 fifoConfig4;       /* 0x058 MAC configuration 4    */
    volatile uint32 fifoConfig5;       /* 0x05C MAC configuration 5    */

    volatile uint32 pad060[72];

    volatile uint32 txControl;         /* 0x180 Tx Control             */

#define TX_CTRL_ENABLE     (1 << 0)    /* Enable Tx                    */
    volatile uint32 txDMA;             /* 0x184 Tx DMA Descriptor      */
    volatile uint32 txStatus;          /* 0x188 Tx Status              */

#define TX_STAT_SENT       (1 << 0)    /* Packet Sent                  */
#define TX_STAT_UNDER      (1 << 1)    /* Tx Underrun                  */

    volatile uint32 rxControl;         /* 0x18C Rx Control             */

#define RX_CTRL_RXE        (1 << 0)    /* Enable receiver              */

    volatile uint32 rxDMA;             /* 0x190 Rx DMA Descriptor      */
    volatile uint32 rxStatus;          /* 0x194 Rx Status              */

#define RX_STAT_RECVD      (1 << 0)    /* Packet Received              */
#define RX_STAT_OVERFLOW   (1 << 2)    /* DMA Rx overflow              */
#define RX_STAT_COUNT      (0xFF << 16)/* Count of packets received    */

    volatile uint32 interruptMask;     /* 0x198 Interrupt Mask         */

#define IRQ_TX_PKTSENT     (1 << 0)    /* Packet Sent                  */
#define IRQ_TX_UNDERFLOW   (1 << 1)    /* Tx packet underflow          */
#define IRQ_TX_BUSERR      (1 << 3)    /* Tx Bus Error                 */
#define IRQ_RX_PKTRCVD     (1 << 4)    /* Rx Packet received           */
#define IRQ_RX_OVERFLOW    (1 << 6)    /* Rx Overflow                  */
#define IRQ_RX_BUSERR      (1 << 7)    /* Rx Bus Error                 */

    volatile uint32 interruptStatus;   /* 0x19C Interrupt Status       */
};
```

```
/* Receiver header struct and constants */

#define ETH_RX_FLAG_OFIFO   0x0001      /* FIFO Overflow                */
#define ETH_RX_FLAG_CRCERR  0x0002      /* CRC Error                    */
#define ETH_RX_FLAG_SERR    0x0004      /* Receive Symbol Error         */
#define ETH_RX_FLAG_OLC     0x0008      /* Frame has odd number nibbles */
#define ETH_RX_FLAG_LARGE   0x0010      /* Frame is > RX MAX Length     */
#define ETH_RX_FLAG_MCAST   0x0020      /* Dest is Multicast Address    */
#define ETH_RX_FLAG_BCAST   0x0040      /* Dest is Broadcast Address    */
#define ETH_RX_FLAG_MISS    0x0080      /* Received due to promisc mode */
#define ETH_RX_FLAG_LAST    0x0800      /* Last buffer in frame         */
#define ETH_RX_FLAG_ERRORS  ( ETH_RX_FLAG_OLC | ETH_RX_FLAG_SERR | \
                              ETH_RX_FLAG_CRCERR | ETH_RX_FLAG_OFIFO )

/* Header on a received packet */

struct rxHeader {
    uint16 length;                      /* Length of packet data        */
    uint16 flags;                       /* Receive flags                */
    uint16 pad[12];                     /* Padding                      */
};

/* Ethernet DMA descriptor */

#define ETH_DESC_CTRL_LEN   0x00001fff  /* Mask for length field        */
#define ETH_DESC_CTRL_MORE  0x10000000  /* More fragments               */
#define ETH_DESC_CTRL_EMPTY 0x00000000  /* Empty descriptor             */

/* Descriptor for the DMA engine to determine where to  */
/*      find a packet buffer.                           */

struct dmaDescriptor {
    uint32 address;                     /* Stored as physical address   */
    uint32 control;                     /* DMA control bits             */
    uint32 next;                        /* Next descriptor in the ring  */
};

#define RESET_CORE 0xB806001C           /* Atheros bus core reset reg   */
#define RESET_E0_MAC (1 << 9)           /* Reset Ethernet zero MAC      */
#define RESET_E1_MAC (1 << 13)          /* Reset Ethernet one  MAC      */
```

306 ↑ 308

结构 ag71xx 规定 AG71xx 硬件的控制和状态寄存器的格式。代码中的有些变量标记为 volatile 同性，该属性是用来告知编译器使用这些引用时需要逐一进行取地址操作。（即，编译器不能进行下面这样的优化：取地址，将其存入寄存器，下次需要使用的时候不进行取地址操作，而直接从寄存器中将该值取出。

16.6 环和内存缓冲区

从设备的角度看，输入或输出环由内存中的链表组成。链表上的每个结点使用结构 dmaDescriptor 定义，该结构包含 3 项：1）一个指向内存缓冲区的指针；2）一个状态字，用来告诉缓冲区是否包含数据；3）链表上指向下一个结点的指针。图 16-1 说明了发送和接收环是如何组织的，以及环上的每个结点如何包含缓冲区的指针。

正如图 16-1 所示，每个环由一个循环链表组成，指针从一个结点指向它的后继结点（最后一个结点指向第一个结点）。阅读代码时，有一点要记住，以太网设备视环为一个链表（即遵循指针设备在一个结点可以得到指向下一个结点的指针）。驱动将环结点存放在连续的存储空间里（例如，链表的结点分配在数组中）。因此，驱动可以使用数组索引访问结点。图 16-2 显示了结点的结构，并说明环

的结点如何存储在数组中。

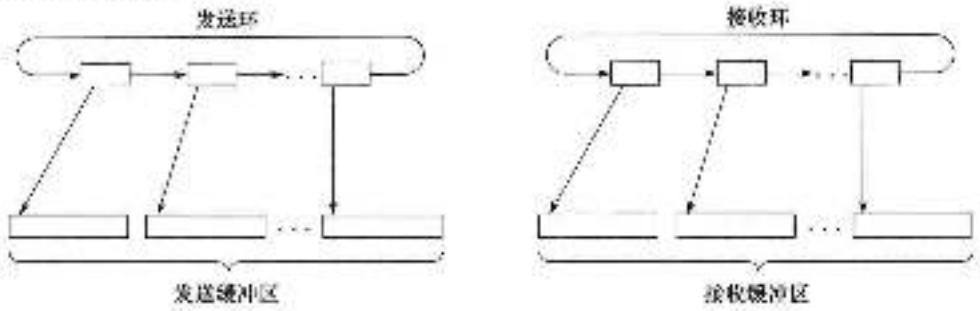
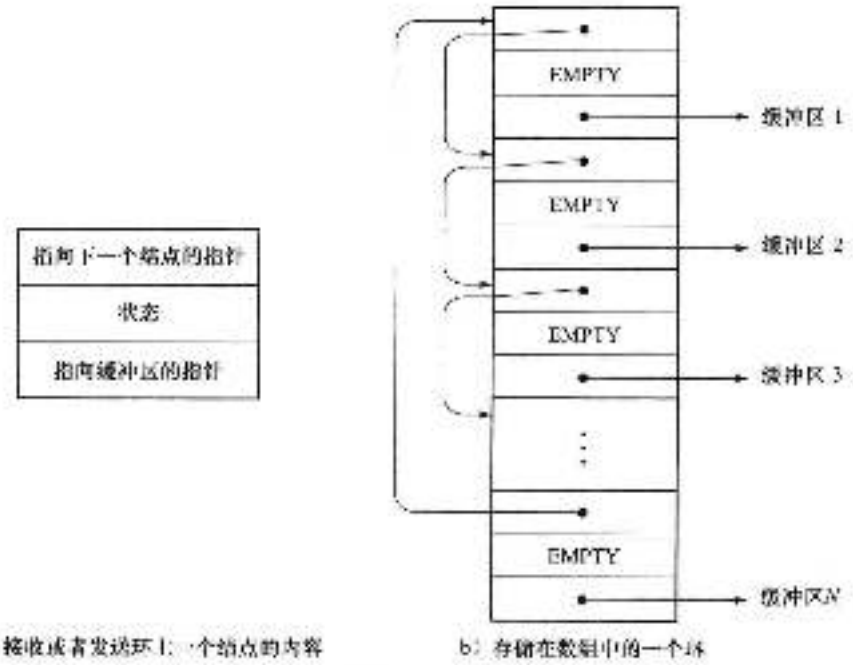


图 16-1　示例 DMA 设备的发送/接收环说明



a）接收或者发送环上一个结点的内容　　b）存储在数组中的一个环

图　16-2

16.7　以太网控制块的定义

文件 ether.h 定义了以太网驱动使用的常量和数据结构，包括以太网数据包头格式、数据包缓冲区在内存中的布局，以太网控制块的内容。

```
/* ether.h */

/* Ethernet packet format:

 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Dest. MAC (6)     Src. MAC (6)    |Type (2)|      Data (46-1500)...    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*/

#define ETH_ADDR_LEN    6       /* Length of Ethernet (MAC) address     */

/* Ethernet packet header */
```

```
struct  etherPkt {
        byte    dst[ETH_ADDR_LEN];      /* Destination Mac address      */
        byte    src[ETH_ADDR_LEN];      /* Source Mac address           */
        uint16  type;                   /* Ether type field             */
        byte    data[1];                /* Packet payload               */
};

#define ETH_HDR_LEN     14      /* Length of Ethernet packet header     */

/* Ethernet buffer lengths */

#define ETH_INBUFSIZ    1024    /* input buffer size                    */

/* Ethernet DMA buffer sizes */

#define ETH_MTU         1500    /* Maximum transmission unit            */
#define ETH_VLAN_LEN    4       /* Length of Ethernet vlan tag          */
#define ETH_CRC_LEN     4       /* Length of CRC on Ethernet frame      */

#define ETH_MAX_PKT_LEN ( ETH_HDR_LEN + ETH_VLAN_LEN + ETH_MTU )

#define ETH_RX_BUF_SIZE ( ETH_MAX_PKT_LEN + ETH_CRC_LEN \
                                + sizeof(struct rxHeader) )

#define ETH_TX_BUF_SIZE ( ETH_MAX_PKT_LEN )

/* State of the Ethernet interface */

#define ETH_STATE_FREE  0       /* control block is unused */
#define ETH_STATE_DOWN  1       /* interface is currently inactive */
#define ETH_STATE_UP    2       /* interface is currently active */

/* Ethernet device control functions */

#define ETH_CTRL_CLEAR_STATS 1  /* Reset Ethernet Statistics            */
#define ETH_CTRL_SET_MAC     2  /* Set the MAC for this device          */
#define ETH_CTRL_GET_MAC     3  /* Get the MAC for this device          */
#define ETH_CTRL_SET_LOOPBK  4  /* Set Loopback Mode                    */
#define ETH_CTRL_RESET       5  /* Reset the Ethernet device            */
#define ETH_CTRL_DISABLE     6  /* Disable the Ethernet device          */

/* Ethernet packet buffer */

struct  ethPktBuffer {
        byte    *buf;           /* Pointer to a packet buffer           */
        byte    *data;          /* Start of data within the buffer      */
        int32   length;         /* Length of data in the packet buffer  */
};

/* Ethernet control block */

#define ETH_INVALID     (-1)    /* Invalid data (virtual devices)       */

struct  ether   {
        byte    state;          /* ETH_STATE_... as defined above       */
        struct  dentry  *phy;   /* physical ethernet device for Tx DMA  */

        /* Pointers to associated structures */
```

```
    struct  dentry  *dev;   /* address in device switch table       */
    void    *csr;           /* addr of control and status regs.     */

    uint32  interruptMask;  /* interrupt mask                       */
    uint32  interruptStatus;/* interrupt status                     */

    struct  dmaDescriptor *rxRing;/* array of receive ring descrip. */
    struct  ethPktBuffer **rxBufs;/* Rx ring array                  */
    uint32  rxHead;         /* Index of current head of Rx ring     */
    uint32  rxTail;         /* Index of current tail of Rx ring     */
    uint32  rxRingSize;     /* size of Rx ring descriptor array     */
    uint32  rxirq;          /* Count of Rx interrupt requests       */
    uint32  rxOffset;       /* Size in bytes of rxHeader            */
    uint32  rxErrors;       /* Count of Rx errors                   */

    struct  dmaDescriptor *txRing;/* array of transmit ring descrip.*/
    struct  ethPktBuffer **txBufs;/* Tx ring array                  */
    uint32  txHead;         /* Index of current head of Tx ring     */
    uint32  txTail;         /* Index of current tail of Tx ring     */
    uint32  txRingSize;     /* size of Tx ring descriptor array     */
    uint32  txirq;          /* Count of Tx interrupt requests       */

    byte    devAddress[ETH_ADDR_LEN]; /* MAC address */

    byte    addressLength;  /* Hardware address length              */
    uint16  mtu;            /* Maximum transmission unit (payload)  */

    uint32  errors;         /* Number of Ethernet errors            */
    uint16  ovrrun;         /* Buffer overruns                      */
    sid32   isem;           /* I/O semaphore for Ethernet input     */
    uint16  istart;         /* Index of packet in the input buffer  */
    uint16  icount;         /* Count of packets in the input buffer */

    struct  ethPktBuffer *in[ETH_IBUFSIZ]; /* Input buffer          */

    int     inPool;         /* Buffer pool ID for input buffers     */
    int     outPool;        /* Buffer pool ID for output buffers    */
};
extern  struct  ether   ethertab[];     /* array of control blocks */

int32   colon2mac(char *, byte *);
int32   allocRxBuffer(struct ether *, int32);
int32   waitOnBit(volatile uint32 *, uint32, const int32, int32);
```

16.8　设备和驱动初始化

306 - 313

当系统启动时，操作系统会首先调用 ethInit 函数来初始化以太网设备以及相应设备驱动的数据结构。文件 ethInit.c 包含如下代码：

```
/* ethInit.c - ethInit */

#include <xinu.h>

struct  ether   ethertab[Neth];         /* Ethernet control blocks */

/*------------------------------------------------------------------------
 * udelay - microsecond delay loop (CPU loop)
 *------------------------------------------------------------------------
 */
```

```
void    udelay(uint32 n) {

        uint32  delay;                  /* amount to delay measured in  */
                                        /*   clock cycles               */
        uint32  start = 0;              /* clock at start of delay      */
        uint32  target = 0;             /* computed clk at end of delay */
        uint32  count = 0;              /* current clock during loop    */

        delay = 200 * n;                /* 200 CPU cycles per usec */

        start = clkcount();             /* Get current clock */
        target = start + delay;         /* Compute finish time */

        if (target >= start) {
                while ((count = clkcount()) < target) &&
                                (count >= start)) {
                        ;  /* spin doing nothing */
                }
        } else {

                /* need to wrap around counter */

                while ((count = clkcount()) > start) {
                        ;  /* spin doing nothing */
                }
                while ((count = clkcount()) < target) {
                        ;  /* spin doing nothing */
                }
        }
}

/*------------------------------------------------------------------------
 * mdelay - millisecond delay loop (CPU loop)
 *------------------------------------------------------------------------
 */
void    mdelay(uint32 n) {
        int i;

        for (i = 0; i < n; i++) {
                udelay(1000);
        }
}

/*------------------------------------------------------------------------
 * ethInit - Initialize Ethernet device structures
 *------------------------------------------------------------------------
 */
devcall ethInit (
         struct dentry *devptr
        )
{
        struct  ether   *ethptr;
        struct  eg7lxx  *nicptr;
        uint32  *rstptr;
        uint32  rstbit;

        /* Initialize structure pointers */

        ethptr = &ethertab[devptr->dvminor];
```

```
        memset(ethptr, '\0', sizeof(struct ether));
        ethptr->dev = devptr;
        ethptr->csr = devptr->dvcsr;

        /* Get device CSR address */

        nicptr = (struct ag7bxx *)devptr->dvcsr;
        rstptr = (uint32 *)RESET_CORE;
        if (devptr->dvminor == 0) {     /* use E0 on first device only */
                rstbit = RESET_E0_MAC;
        } else {
                rstbit = RESET_E1_MAC;
        }

        ethptr->state = ETH_STATE_DOWN;
        ethptr->rxRingSize = ETH_RX_RING_ENTRIES;
        ethptr->txRingSize = ETH_TX_RING_ENTRIES;
        ethptr->mtu = ETH_MTU;
        ethptr->interruptMask = IRQ_TX_PKTSENT | IRQ_TX_BUSERR
                | IRQ_RX_PKTRCV | IRQ_RX_OVERFLOW | IRQ_RX_BUSERR;
ethptr->errors = 0;
ethptr->isem = semcreate(0);
ethptr->istart = 0;
ethptr->icount = 0;
ethptr->ovrrun = 0;
ethptr->rxOffset = ETH_PKT_ALIGNED;

colon2mac(nvramGet("et0macaddr"), ethptr->devAddress);
ethptr->addressLength = ETH_ADDR_LEN;

/* Reset the device */

nicptr->macConfig1 |= MAC_CFG1_SOFTRESET;
udelay(20);
*rstptr |= rstbit;
mdelay(100);
*rstptr &= ~rstbit;
mdelay(100);

/* Enable transmit and receive */

nicptr->macConfig1 = MAC_CFG1_TX | MAC_CFG1_SYNC_TX |
        MAC_CFG1_RX | MAC_CFG1_SYNC_RX;

/* Configure full duplex, auto padding CRC, */
/*      and interface mode                  */

nicptr->macConfig2 |= MAC_CFG2_FDX | MAC_CFG2_PAD |
                MAC_CFG2_LEN_CHECK | MAC_CFG2_IMNTBBLR;

/* Enable FIFO modules */

nicptr->fifoConfig0 = FIFO_CFG0_WTMENREQ | FIFO_CFG0_SRFENREQ |
  FIFO_CFG0_FRSTENREQ | FIFO_CFG0_STFENREQ | FIFO_CFG0_DTFENREQ;

nicptr->fifoConfig1 = 0x0FFF0000;

/* Max out number of words to store in Receiver RAM */
```

```
    nicptr->fifoConfig2 = 0x00001FFF;

    /* Drop any incoming packet with errors in the Rx status vector  */

    nicptr->fifoConfig4 = 0x0003FFFF;

    /* Drop short packets (set "don't care" on Rx stats vector bits */

    nicptr->fifoConfig5 = 0x0003FFFF;

    /* Buffers should be page-aligned and cache-aligned */

    ethptr->rxBufs = (struct ethPktBuffer *)getstk(PAGE_SIZE);
    ethptr->txBufs = (struct ethPktBuffer **)getstk(PAGE_SIZE);
    ethptr->rxRing = (struct dmaDescriptor *)getstk(PAGE_SIZE);
    ethptr->txRing = (struct dmaDescriptor *)getstk(PAGE_SIZE);

    if (   ( (int32)ethptr->rxBufs == SYSERR )
        || ( (int32)ethptr->txBufs == SYSERR )
        || ( (int32)ethptr->rxRing == SYSERR )
        || ( (int32)ethptr->txRing == SYSERR ) ) {
            return SYSERR;
    }

    /* Translate buffer and ring pointers to KSEG1 */

    ethptr->rxBufs = (struct ethPktBuffer **)
            ((uint32)ethptr->rxBufs - PAGE_SIZE +
              sizeof(int32)) | KSEG1_BASE);
    ethptr->txBufs = (struct ethPktBuffer **)
            ((uint32)ethptr->txBufs - PAGE_SIZE +
              sizeof(int32)) | KSEG1_BASE);
    ethptr->rxRing = (struct dmaDescriptor *)
            ((uint32)ethptr->rxRing - PAGE_SIZE +
              sizeof(int32)) | KSEG1_BASE);
    ethptr->txRing = (struct dmaDescriptor *)
            ((uint32)ethptr->txRing - PAGE_SIZE +
              sizeof(int32)) | KSEG1_BASE);

    /* Set buffer pointers and rings to zero */

    memset(ethptr->rxBufs, '\0', PAGE_SIZE);
    memset(ethptr->txBufs, '\0', PAGE_SIZE);
    memset(ethptr->rxRing, '\0', PAGE_SIZE);
    memset(ethptr->txRing, '\0', PAGE_SIZE);

    /* Initialize the interrupt vector and enable the device */

    interruptVector[devptr->dvirq] = devptr->dvintr;
    enable_irq(devptr->dvirq);
            return OK;
    }
```

硬件在各个初始化步骤中需要特定的延迟（使设备中的硬件有充足的时间进行初始化操作）。但是，当执行这些初始化代码时，操作系统并没有在运行，一些如 sleep 的延迟函数还不能让用户调用。因此，代码中自定义了两个延迟函数 udelay 和 mdelay，分别延迟指定的微秒数和毫秒数。在每种情况下，代码由一个使 CPU 被占指定时间的循环组成。由于因为循环迭代的次数取决于 cpu 的时钟速度，所以这两个函数都是平台相关的。更重要的是，一个相同类型硬件的时钟周期可能与另一个的时钟周期稍微不一致。为了协调这种差异，通过使用实时性时钟来实现延迟同步，即实现一个 delay 函数读取

当前时钟，估计循环需要执行多少次并运行该循环。然后它重新读取时钟来决定是否还需要额外的延迟。

当被调用时，ethInit 函数初始化设备控制块中的属性，并初始化硬件。这里的许多细节依赖于具体的以太网网络设备，但有一个选项对于 DMA 硬件设备来说总是一致的：寻址方式。对于 i21001 来说，DMA 硬件设备直接使用底层的总线，即这个硬件设备使用的是物理地址而不是操作系统使用的虚地址。因此，当操作系统把一个地址传给设备时，它必须先把所有的地址转化成物理地址，尤其是，物理地址必须存放在缓冲环中并且传给物理设备的缓冲环地址必须是物理地址。在代码中，虚地址到物理地址的转化是由一个内联函数来完成的，它用常量 KSEG1_BASE 与一个地址进行逻辑或运算来获得虚然转换的细节随着具体的平台有所不同（可能需要操作系统使用 MMU 硬件设备），但基本原理是一样的：当在 DMA 中使用链表时，每个地址都必须转换为一种硬件可以理解的形式。

最后一步，ethInit 启动设备中断（即，允许设备开始接收和存储数据包，并产生中断）。启动使设备中断并不意味着中断会出现：在初始化过程中，操作系统运行在所有 CPU 中断都关闭的情形下。因此，设备能请求一个中断，但 CPU 并不处理中断。一旦操作系统启动 CPU 中断，设备就能够中断 CPU 并把以前一些积累的数据包传给 CPU。

16.9　分配输入缓冲区

在读取数据包之前，驱动必须分配一个缓冲区来存储数据包，并把缓冲区链接到环中以供 DMA 输入。我们的驱动使用一个工具函数 allocRxBuffer 从缓冲区池中分配一个缓冲区，并把缓冲区地址链接到 DMA 环中。文件 allocRxBuffer.c 如下所示。

314 ~ 318

```
/* allocRxBuffer.c - allocRxBuffer */

#include <xinu.h>

/*------------------------------------------------------------------------
 * allocRxBuffer - allocate an Ethernet packet buffer structure
 *------------------------------------------------------------------------
 */
int32	allocRxBuffer (
	 struct ether	*ethptr,	/* ptr to device control block	*/
	 int32	destIndex		/* index in receive ring	*/
	)
{
	struct	ethPktBuffer	*pkt;
	struct	dmaDescriptor	*dmaptr;

	/* Compute next ring location modulo the ring size */

	destIndex %= ethptr->rxRingSize;

	/* Allocate a packet buffer */

	pkt = (struct ethPktBuffer *)queue[ethptr->rxPool];

	if ((int32)pkt == SYSERR) {
		kprintf("eth0 allocRxBuffer() error\r\n");
		return SYSERR;
	}
	pkt->length = ETH_RX_BUF_SIZE;
	pkt->buf = (byte *)(pkt + 1);

	/* Data region offset by size of rx header */

	pkt->data = pkt->buf + ethptr->rxOffset;
```

```
        ethptr->rxBufs[destIndex] = pkt;

        /* Fill in DMA descriptor fields */

        dmaptr = ethptr->rxRing + destIndex;
        dmaptr->control = ETH_DESC_CTRL_EMPTY;
        dmaptr->address = (uint32)(pkt->buf) & PMEM_MASK;

        return OK;
219     }
```

在我们的DMA描述中，环中的每个结点都包括一个位，以告诉我们缓冲区是满的还是空的。AC71xx硬件使用DMA描述结构体（struct dmaDescriptor）中的control字段中的一位作为标示位，其常量名为ETH_DESC_CTRL_EMPTY。当缓冲区为空时，值为1，当缓冲区包含数据包时，该位为0。

因此，当系统分配一个缓冲区并把缓冲区链接到环中之后，allocRxBuffer必须将ETH_DESC_CTRL_EMPTY位设置为1来指明缓冲区是空的；当有数据包到达并放入缓冲区之后，设备硬件就把该位设置为0。AllocRxBuffer接收两个参数：指向以太网控制块的指针和缓冲环中目前使用的缓冲区的位置。当系统分配一个缓冲区之后，allocRxBuffer计算该缓冲区在缓冲环中的位置，把缓冲区的地址放入数据包的头部，并设置DMA描述结构体中的control字段。

16.10 从以太网设备中读取数据包

因为DMA引擎使用输入环将到来的数据包存储到连续的缓冲区内，所以读取数据包并不需要与设备硬件进行交互。相反，驱动使用信号量机制来调节读操作：一个读数据包的应用程序进程等在信号量上，当有数据包到达时，中断代码释放信号量。因此，如果当前没有数据包需要处理，调用者会由于得不到信号量而被阻塞。当缓冲区中有数据包时，中断处理程序会发信号给调用者，从而使调用者能继续执行。数据包也会被放入缓冲环中的下一个缓冲区中。处理读取操作的驱动只需要把数据包从缓冲环复制到调用者的缓冲区，然后返回即可。文件ethRead.c代码如下：

```
/* ethRead.c - ethRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethRead - read a packet from an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethRead (
        struct dentry  *devptr,         /* entry in device switch table */
        void   *buf,                    /* buffer to hold packet        */
        uint32 len                      /* length of buffer             */
        )
{
        struct  ether   *ethptr;        /* ptr to entry in ethertab     */
        struct  ethPktBuffer    *pkt;   /* ptr to a packet              */
        uint32  length;                 /* packet length                */

        ethptr = &ethertab[devptr->dvminor];
        if (ETH_STATE_UP != ethptr->state) {
                return SYSERR; /* interface is down */
        }

        /* Make sure user's buffer is large enough to store at least    */
        /*   the header of a packet                                     */

        if (len < ETH_HDR_LEN) {
```

```
        return SYSERR;
    }

    /* Wait for a packet to arrive */

    wait(ethptr->isema);

    /* Pick up packet */

    pkt = ethptr->in[ethptr->istart];
    ethptr->in[ethptr->istart] = NULL;
    ethptr->istart = (ethptr->istart + 1) % ETH_INVSIZ;
    ethptr->icount--;

    if (pkt == NULL) {
        return 0;
    }

    length = pkt->length;
    memcpy(buf, (byte *)((uint32)pkt->but) | KSEG1_BASE), length);
    freebut((char *)pkt);

    return length;
}
```

在验证以太网设备已经就绪并检查了它的参数之后，ethRead 等待输入信号量，调用函数 wait，等待阻塞直到以太网设备中至少有一个数据包。一旦 ethRead 函数通过 wait 函数，即从下一个可用的缓冲环中把数据包复制到调用函数的缓冲区中，然后返回。设备驱动控制块中的 istart 字段指定了要使用的缓冲环指针（注意：虽然设备硬件的缓冲环是用链表来实现的，但驱动仍然使用数组方式来进行索引）。ethRead 通过加 1 并模除缓冲环大小把 istart 指向下一个要使用的环中的缓冲区。

320
|
321

16.11　向以太网设备中写入数据包

使用 DMA 来进行，把输出像输入一样简单。应用程序调用 write 来发送数据包，其中 write 函数又调用了 ethWrite 函数。与输入一样，输出端只与缓冲环交互；ethWrite 函数将调用者的缓冲区内容复制到下一个可用的输出缓冲区中。文件 ethWrite.c 代码如下：

```
/* ethWrite.c -  etherWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethWrite - write a packet to an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethWrite (
        struct dentry  *devptr,         /* entry in device switch table */
        void   *buf,                    /* buffer to hold packet        */
        uint32 len                      /* length of buffer             */
        )
{
        struct  ether   *ethptr;
        struct  ag7lxx  *nicptr;
        struct  ethPktBuffer *pkt;
        struct  dmaDescriptor *dmaptr;
        uint32  tail = 0;
        byte    *buffer;
```

```
    buffer = buf;

    ethptr = &ethertab[devptr->dvminor];
    nicptr = ethptr->csr;

    if ((ETH_STATE_UP != ethptr->state)
            || (len < ETH_HDR_LEN)
            || (len > (ETH_TX_BUF_SIZE - ETH_VLAN_LEN))) {
            return SYSERR;
    }

    tail = ethptr->txTail % ETH_TX_RING_ENTRIES;
    dmaptr = &ethptr->txRing[tail];

    if (!(dmaptr->control & ETH_DESC_CTRL_EMPTY)) {
            ethptr->errors++;
            return SYSERR;
    }

    pkt = (struct ethPktBuffer *)getbuf(ethptr->outPool);
    if ((uint32)pkt == SYSERR) {
            ethptr->errors++;
            return SYSERR;
    }

    /* Translate pkt pointer into uncached memory space */

    pkt = (struct ethPktBuffer *)((int)pkt | KSEG1_BASE);
    pkt->buf = (byte *)(pkt + 1);
    pkt->data = pkt->buf;
    memcpy(pkt->data, buffer, len);

    /* Place filled buffer in outgoing queue */
    ethptr->txBufs[tail] = pkt;

    /* Add the buffer to the transmit ring.  Note that the address  */
    /*  must be physical (MIPS) because the DMA engine will used it */

    ethptr->txRing[tail].address = (uint32)pkt->data & PMEM_MASK;

    /* Clear empty flag and write the length */

    ethptr->txRing[tail].control = len & ETH_DESC_CTRL_LEN;

    /* move to next position */

    ethptr->txTail++;

    if (!(nicptr->txStatus & TX_STAT_UNDER)) {
            nicptr->txDMA = ((uint32)(ethptr->txRing + tail))
                    & PMEM_MASK;
            nicptr->txStatus = TX_STAT_UNDER;
    }

    /* Enable transmit interrupts */

    nicptr->txControl = TX_CTRL_ENABLE;
    return len;
}
```

在验证参数之后，ethWrite 函数等待输出缓冲环中一个空的位置，并从调用者缓冲区复制一个数据包到该环缓冲区上。如果设备目前处于空闲状态，ethWrite 就必须启动该设备。启动 DMA 设备非常简单，只需将常量 TX_CTRL_ENABLE 赋给设备的传输控制寄存器。如果设备已经在运行，那么这个赋值将不产生任何效果；否则这个赋值就会在环的下一个位置启动该设备（驱动将把下一个数据包传送到空的位置）。

16.12　以太网设备的中断处理

利用 DMA 设备启动的优势之一在于 DMA 设备上的引擎能处理很多细节。因此，中断处理没有涉及很多与设备的互动。中断会出现在输入/输出操作成功完成或者 DMA 引擎出错时。中断处理程序询问设备来确认中断的原因。对于成功的输入中断，处理程序调用函数 rxPackets；对于成功的输出中断，处理程序调用函数 txPackets。对于出错情况，处理程序都是直接解决。文件 ethInterrupt.c 的代码如下：

```
/* ethInterrupt.c - ethInterrupt */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rxPackets  -  handler for receiver interrupts
 *------------------------------------------------------------------------
 */
void    rxPackets (
          struct ether *ethptr,         /* ptr to control block         */
          struct ag7lxx *nicptr         /* ptr to device CSRs           */
        )
{
        struct  dmaDescriptor *dmaptr;  /* ptr to DMA descriptor        */
        struct  ethPktBuffer  *pkt;     /* ptr to one packet buffer     */
        int32   head;


        /* Move to next packet, wrapping around if needed */

        head = ethptr->rxhead % ETH_RX_RING_ENTRIES;
        dmaptr = &ethptr->rxRing[head];
        if (dmaptr->control & ETH_DESC_CTRL_EMPTY) {
                nicptr->rxStatus = RX_STAT_RECVD;
                return;
        }

        pkt = ethptr->rxBufs[head];
        pkt->length = dmaptr->control & ETH_DESC_CTRL_LEN;

        if (ethptr->icount < ETH_IBUFSIZ) {
                allocRxBuffer(ethptr, head);
                ethptr->in[(ethptr->istart + ethptr->icount) %
                           ETH_IBUFSIZ] = pkt;
                ethptr->icount++;
                signal(ethptr->isema);
        } else {
                ethptr->ovrrun++;
                memset(pkt->buf, '\0', pkt->length);
        }

        ethptr->rxHead++;

        /* Clear the Rx interrupt */
```

```
        nicptr->rxStatus = RX_STAT_RECVD;
        return;
}

/*-------------------------------------------------------------------------
 * txPackets - handler for transmitter interrupts
 *-------------------------------------------------------------------------
 */
void    txPackets (
         struct ether *ethptr,          /* ptr to control block         */
         struct ag7lxx *nicptr          /* ptr to device CSRs           */
        )
{
        struct  dmaDescriptor *dmaptr;
        struct  ethPktBuffer **epb = NULL;
        struct  ethPktBuffer *pkt = NULL;
        uint32  head;

        if (ethptr->txHead == ethptr->txTail) {
                nicptr->txStatus = TX_STAT_SENT;
                return;
        }

        /* While packets remain to be transmitted */
        while (ethptr->txHead != ethptr->txTail) {
                head = ethptr->txHead % ETH_TX_RING_ENTRIES;
                dmaptr = &ethptr->txRing[head];
                if (!(dmaptr->control & ETH_DESC_CTRL_EMPTY)) {
                        break;
                }

                epb = &ethptr->txBufs[head];

                /* Clear the Tx interrupt */

                nicptr->txStatus = TX_STAT_SENT;

                ethptr->txHead++;
                pkt = *epb;
                if (NULL == pkt) {
                        continue;
                }
                freebuf((void *)((hpid32)pkt & (PHYS_MASK | RSRVD_BASE)));
                *epb = NULL;
        }
        return;
}

/*-------------------------------------------------------------------------
 * ethInterrupt - decode and handle interrupt from an Ethernet device
 *-------------------------------------------------------------------------
 */
interrupt ethInterrupt(void)
{
        struct  ether   *ethptr;        /* ptr to control block         */
        struct  ag7lxx  *nicptr;        /* ptr to device CSRs           */
        uint32  status;
        uint32  mask;
```

```
    /* Initialize structure pointers */

    ethptr = &ethertab[0];          /* default physical Ethernet    */
    if (!ethptr) {
            return;
    }
    nicptr = ethptr->csr;
    if (!nicptr) {
            return;
    }

    /* Obtain status bits from device */

    mask = nicptr->interruptMask;
    status = nicptr->interruptStatus & mask;

    /* Record status in ether struct */

    ethptr->interruptStatus = status;

    if (status == 0) {
            return;
    }

    sched_cntl(DEFER_START);

    if (status & IRQ_TX_PKTSENT) {  /* handle transmitter interrupt */
            ethptr->txirq++;
            txPackets(ethptr, nicptr);
    }

    if (status & IRQ_RX_PKTRECV) {  /* handle receiver interrupt    */
            ethptr->rxirq++;
            rxPackets(ethptr, nicptr);
    }

    /* Handle errors (transmit or receive overflow) */

    if (status & IRQ_RX_OVERFLOW) {
            /* Clear interrupt and restart processing */
            nicptr->rxStatus = RX_STAT_OVERFLOW;
            nicptr->rxControl = RX_CTRL_RXE;
            ethptr->errors++;
    }

    if (!(status & IRQ_TX_UNDERFLOW) ||
            (status & IRQ_TX_BUSERR) || (status & IRQ_RX_BUSERR)) {
            panic("Catastrophic Ethernet error");
    }
    sched_cntl(DEFER_STOP);
    return;
}
```

注意，除非硬件失灵，否则不会出现发送下溢或总线错误。

16.13　以太网控制函数

以太网驱动支持 3 个控制函数：调用者可以从设备中获取 MAC 地址，设定 MAC 地址和设定用于测试的回环模式（loopback mode）。我们通常认为以太网的 MAC 地址是硬连线到设备的。然而，在最

低水平下，制造商不想在硬件被测试前分配永久地址。Xinu 系统是按照典型的小型嵌入式系统设计的：MAC 地址不是烧入到以太网接口芯片中，而是载入不易丢失的 RAM 中。当系统启动时，系统从中提取地址并把它载入到设备中。

因为基础系统使用 32 位数据传输，所以设备会将 48 位 MAC 地址分为两部分。为了设定 MAC 地址，ethControl 一定要设置两个值。ethControl 先提取用户 MAC 地址的前 4 个字节，将每个字节转换到 32 位整数内的对应位置上，再将结果存储到设备上。然后提取 MAC 地址的最后两个字节，转换到对应位置，并将结果存储到设备中。

从设备上获得 MAC 地址与存储 MAC 地址正好相反。ethControl 先从设备上获得一个整数值，然后对每个字节进行移位并计算其掩码，并存储在到用者特有的数组中。一旦前 4 个字节存储后，ethControl 就读取第二个整数并提取最后两个字节。文件 ethControl.c 的代码如下：

```
/* ethControl.c - ethControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethControl - implement control function for an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethControl (
         struct dentry  *devptr,        /* entry in device switch table */
         int32  func,                   /* control function             */
         int32  arg1,                   /* argument 1, if needed        */
         int32  arg2                    /* argument 2, if needed        */
        )
{
        struct  ether   *ethptr;        /* ptr to control block         */
        struct  ag7lex  *nicptr;        /* ptr to device CSRs           */
byte    *macptr;                /* ptr to MAC address           */
uint32  temp;                   /* temporary                    */

ethptr = &ethertab[devptr->dvminor];
if (ethptr->csr == NULL) {
        return SYSERR;
}
nicptr = ethptr->csr;

switch (func) {

/* Program MAC address into card. */

case ETH_CTRL_SET_MAC:
        macptr = (byte *)arg1;

        temp  = ((uint32)macptr[0]) << 24;
        temp |= ((uint32)macptr[1]) << 16;
        temp |= ((uint32)macptr[2]) <<  8;
        temp |= ((uint32)macptr[3]) <<  0;
        nicptr->macAddr1 = temp;

        temp  = 0;
        temp  = ((uint32)macptr[4]) << 24;
        temp |= ((uint32)macptr[5]) << 16;
        nicptr->macAddr2 = temp;
        break;
```

```
    /* Get MAC address from card */

    case END_GET_ADDR:
            macptr = (byte *)arg1;

            temp = nicptr->macAddr1;
            macptr[0] = (temp >> 24) & 0xff;
            macptr[1] = (temp >> 16) & 0xff;
            macptr[2] = (temp >>  8) & 0xff;
            macptr[3] = (temp >>  0) & 0xff;

            temp = nicptr->macAddr2;
            macptr[4] = (temp >> 24) & 0xff;
            macptr[5] = (temp >> 16) & 0xff;
            break;

    /* Set receiver mode */

            case END_SET_RECV_MODE:
                    if (TRUE == (uint32)arg1) {
                            nicptr->macConfig1 |= MAC_CFG1_LOOPBACK;
                    } else {
                            nicptr->macConfig1 &= ~MAC_CFG1_LOOPBACK;
                    }
                    break;
            default:
                    return SYSERR;
        }
        return OK;
    }
```

16.14　观点

DMA 设备对必须编写设备驱动的程序员来说是喜忧参半。一方面，DMA 硬件极为复杂，数据单（data sheet）往往理解，以至于程序员会觉得它晦涩难懂。DMA 设备与含几个简单控制器和状态寄存器的设备不同，它需要程序员在存储器中创建复杂的数据结构，并把它们的地址传送给设备。此外，程序员必须了解硬件何时且如何在数据结构中发出同复位，以及如何解释操作系统发出的请求。另一方面，一旦程序员领悟了参考资料，随之产生的驱动代码比非 DMA 设备的代码更加精巧。

16.15　总结

使用 DMA 设备可以在设备和存储器之间移动任意数据块，而不需要使用 CPU 来获得数据中的每个字。DMA 设备一般会在存储器中使用缓冲环，缓冲环内的每个结点都指向同一个缓冲区。一旦驱动将硬件指向环内的一个结点，DMA 设备引擎就可以执行操作，并自动移至环内的下一个结点。

DMA 设备的主要优势在于较低的开销：设备只需要每块中断一次，而不是每字节或每字一次。DMA 设备的驱动代码比传统设备的代码更为简单，因为不需要执行低级操作。

328 ~ 330

练习

16.1　驱动代码使用数组指针从一个结点移到下一个结点。如果将代码改为使用链接而不是数组指针，那么这个驱动更高效还是低效？

16.2　读取以太网数据包并接到最小数据包大小。在 100Mbps 下，每秒钟有多少数据包到达？

16.3　创建一个尽可能快发送以太网数据包的测试程序。每秒可以发送多少大数据包？多少小数据包？

16.4　更正的驱动很复杂并且代码很难理解。重写代码，使用数组描述发送和接收环。静态分配数据包缓冲区。

311

第 17 章

Operating System Design: The Xinu Approach, Linksys Version

最小互联网协议栈

遥远的诱惑都是骗人的。最大的机会就在你眼前。

——约翰·巴勒斯

17.1 引言

由于很多嵌入式系统使用网络进行通信，网络协议软件已经成为小型嵌入式操作系统的标准部分。前面的章节描述了发送和接收数据包的基本以太网设备驱动。尽管以太网设备可以传输数据包，但仍需要其他通信软件以允许应用程序在网络中进行通信。一般来说，大部分系统使用 TCP/IP 协议簇（TCP/Internet Protocol Suite）。这些协议组成了协议栈（protocol stack）。

完整的 TCP/IP 栈包含很多协议，远不止一章就能描述清楚。因此，本章描述的是其最小实现，能够支持远程磁盘和远程文件系统（这些内容将在后面的章节介绍）。这里仅简单地描述，没有深入探讨协议的细节。读者可以参考作者编写的其他书籍，以了解协议簇及其完整实现。

[333]

17.2 所需的功能

互联网协议的实现允许 Xinu 系统上运行的进程与远程计算机上运行的应用程序进行通信（包括 PC、Mac 或 UNIX 系统，如 Linux 或 Solaris）。它可以识别远程计算机并与计算机交换报文。

Xinu 实现的协议包括：

- IP　互联网协议
- UDP　用户数据报文协议
- ARP　地址解析协议
- DHCP　动态主机配置协议
- ICMP　互联网控制报文协议

IP　互联网协议（Internet Protocol）定义网络数据包的格式，数据包称为数据报。每个数据报放在以太网帧的数据区域内。互联网协议还定义了地址格式。Xinu 的实现不支持 IP 选项，如存储分片。数据包转发采用大多数终端系统所使用的模式：IP 软件必须知道计算机的 IP 地址，局域网的地址掩码和一个默认路由器地址。如果目的地不在局域网中，那么数据包将会被送到默认路由器上。

UDP　用户数据报协议（User Datagram Protocol）定义了一组 16 位的端口号（port number），操作系统利用这些端口号来识别特殊的应用程序。通信程序必须支持它们将要使用的端口号。端口号允许没有干扰的同步通信：一个应用程序在与远程服务器交互的同时，另一个应用程序可以与其他服务器进行交互。

ARP　地址解析协议（Address Resolution Protocol）提供两种功能。在其他计算机发送 IP 数据包到本系统之前，该计算机必须发送请求以太网地址的 ARP 数据包，而我们的系统必须用 ARP 做出回应。类似地，在我们的系统发送 IP 数据包给其他计算机之前，也先发送 ARP 请求，获得计算机的以太网地址后使用以太网地址发送 IP 数据包。

DHCP　动态主机配置协议（Dynamic Host Configuration Protocol）提供了获取 IP 地址、网络地址掩码和默认路由器（default router）IP 地址的机制。计算机广播一个请求，运行在网络上的 DHCP 服务器发送回应。在网络通信建立前，必须获取 IP 地址等相关的信息。我们的实现不是在启动时立即调用 DHCP，而是等到尝试获取本地 IP 地址时再调用。

[334]

ICMP　互联网控制报文协议（Internet Control Message Protocol）提供了支持 IP 协议的错误和消息报文。我们的实现只处理 ping 程序中使用的两种 ICMP 报文：回显请求（Echo Request）和回显应答（Echo Reply）。由于 ICMP 的代码很大，所以我们在描述协议软件结构时并不展示所有细节，其完整代码可以通过本书中给出的网址找到[1]。

17.3　同步对话、超时和进程

协议软件是如何组织的？需要多少进程？一个完整栈包含很多协议，为了同步使用这些协议，很多实现都使用多进程，给每个进程分配其中的一个协议。其他实现使用软件中断来选择进程。如后面将会看到的，我们的最小规模软件只使用两个额外的进程来描述上述的协议集。本节将描述进程结构，后面的各节将会解释代码。

为什么需要进程呢？因为其是网络协议设计的基础。超时重发（timeout-and-retransmission）技术负责处理数据包丢失的情况（如服务器中队列溢出）。为了使用超时重发，必须设计一个协议，使接收者对每个报文产生应答。当发送者发送报文时，它同时启动一个定时器。如果应答到达之前定时器到期，那么发送者将认为报文丢失并重发第二个副本。

我们的软件设计优雅，拥有一个网络输入进程，名为 netin[2]。设计中使用函数 recvtime 来处理超时。也就是说，当报文传送结束后，发送者调用 recvtime 函数来等待响应。当收到响应时，网络输入进程 netin 发送报文给等待进程，同时 recvtime 函数将报文作为返回值返回。如果定时器到期，recvtime 返回 TIMEOUT。为了使系统正常工作，发送者必须与 netin 进程协调。即在发送者发送报文前，它必须在 netin 知道的位置存储进程 ID。对于 ARP 报文，进程 ID 存储在 ARP 表以用于解析地址。对于 UDP 报文，进程 ID 存储在 UDP 表项的数据包队列中，供端口使用。图 17-1 说明 netin 进程如何存储 UDP 队列中的传入数据包或者从传入的 ARP 数据包中提取信息并将信息放入 ARP 表项中。另一种情况是，如果进程在等待，那么 netin 就发送报文给等待的进程。

335

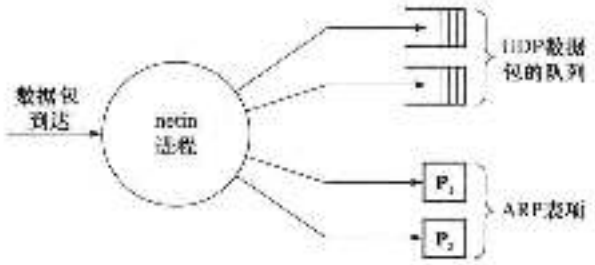


图 17-1　netin 进程的概念性功能

在输出端，我们的设计允许应用程序调用输出函数。唯一的例外来自 ICMP 回显：我们的设计用一个单独的 ICMP 输出进程用来处理 ping 应答。这个例外是不可或缺的，因为我们必须将 ICMP 输入和输出去耦，同时允许在 ICMP 发送应答时，网络输入进程能够继续执行。去耦是必不可少的，因为 ICMP 应答以 IP 数据包的形式传输，而发送 IP 数据包需要 ARP 交换。为了使 ARP 工作，网络输入进程需要继续执行（即必须读取和处理传入的 ARP 应答）。因此，如果网络输入进程正在等待 ARP 的应答，那么系统将会死锁。图 17-2 阐述的是解耦合的过程：netin 进程将输出的 ICMP 数据包放入队列中来供 ICMP 输出进程使用。

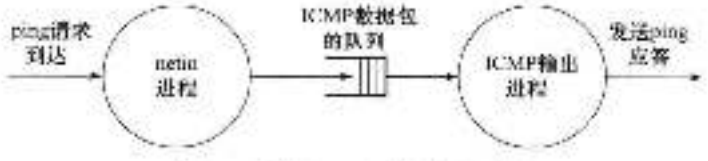


图 17-2　处理 ping 应答的进程结构

[1] URL：sinu.cs.purdue.edu.

[2] netin 的代码见 17.6 节。

17.4 ARP 函数

在以太网上的两台计算机能够使用 IP 协议通信之前，它们必须知道对方的以太网地址。这个协议交换两个报文：计算机 A 广播一个包括 IP 地址的 ARP 请求。任何一个在这个网络上的计算机，如果有请求中的 IP 地址，那么它将发送一个 ARP 应答说明自己的以太网地址。当有应答到达时计算机 A 后，就在自己的 ARP 缓存中增加一项。表项包括远程计算机的 IP 地址和自己的以太网地址。在后续与
336 相同的目的块交互时，便从 ARP 缓存中提取信息而不需要发送 ARP 请求。

Xinu 实现将 ARP 信息存储在数组 arpcache 中。结构 arpentry 定义了数组中每项的内容，包括：一个状态字段（指定了该项是否未使用、正在被填充或者已经被填充）、一个 IP 地址、对应的以太网地址和一个进程 ID。如果进程处于挂起状态，那么这个进程 ID 字段包括的是等待信息到达的进程 ID。文件 arp.h 中定义了 ARP 数据包的格式（当在以太网中使用时）和 ARP 缓存项的格式：

```
/* arp.h */

/* Items related to ARP - definition of cache and the packet format */

#define ARP_HALEN       6               /* size of Ethernet MAC address */
#define ARP_PALEN       4               /* size of IP address           */

#define ARP_HTYPE       1               /* Ethernet hardware type       */
#define ARP_PTYPE       0x0800          /* IP protocol type             */

#define ARP_OP_REQ      1               /* Request op code              */
#define ARP_OP_REPLY    2               /* Reply op code                */

#define ARP_SIZ         16              /* number of entries in a cache */

#define ARP_RETRY       3               /* num. retries for ARP request */

#define ARP_TIMEOUT     250             /* retry timer in milliseconds  */

/* State of an ARP cache entry */

#define AR_FREE         0               /* slot is unused               */
#define AR_PENDING      1               /* resolution in progress       */
#define AR_RESOLVED     2               /* entry is valid               */

#pragma pack(2)
struct  arppacket {                     /* ARP packet for IP & Ethernet */
        byte    arp_tharaddr[ETH_ADDR_LEN];/* Ethernet dest. MAC addr   */
        byte    arp_ethsaddr[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  arp_ethtype;            /* Ethernet type field          */
        uint16  arp_htype;              /* ARP hardware type            */
        uint16  arp_ptype;              /* ARP protocol type            */
        byte    arp_hlen;               /* ARP hardware address length  */
        byte    arp_plen;               /* ARP protocol address length  */
        uint16  arp_op;                 /* ARP operation                */
        byte    arp_sndha[ARP_HALEN];   /* ARP sender's Ethernet addr.  */
        uint32  arp_sndpa;              /* ARP sender's IP address      */
        byte    arp_tarha[ARP_HALEN];   /* ARP target's Ethernet addr.  */
        uint32  arp_tarpa;              /* ARP target's IP address      */
};
#pragma pack()

struct  arpentry {                      /* entry in the ARP cache       */
        int32   arstate;                /* state of the entry           */
```

```
        uint32  arpaddr;                /* IP address of the entry      */
        pid32   arpid;                  /* waiting process or -1        */
        byte    arhaddr[ARP_HALEN];     /* Ethernet address of the entry*/
};

extern  struct  arpentry arpcache[];
```

ARP 在请求和响应中使用相同的数据包格式，在头部字段指定请求或者响应的类型。在任何情况下，数据包里都包括发送者和接受者的 IP 地址及以太网地址。在请求中，目标的以太网地址是不知道的，所以这个字段的值为 0。

我们的 ARP 软件包括 4 个函数，arp_init、arp_resolve、arp_in 和 arp_alloc。所有这 4 个函数都包括在一个单独的源文件 arp.c 中：

```
/* arp.c - arp_init, arp_resolve, arp_in, arp_alloc */

#include <xinu.h>

struct  arpentry  arpcache[ARP_SIZ];    /* ARP cache                    */
sid32   arpmutex;                       /* Mutual exclusion semaphore   */

/*------------------------------------------------------------------------
 * arp_init - initialize ARP mutex and cache
 *------------------------------------------------------------------------
 */
void    arp_init(void) {

        int32   i;                      /* ARP cache index              */

        arpmutex = semcreate(1);
        for (i=0; i<ARP_SIZ; i++) {     /* Initialize cache to empty    */
                arpcache[i].arstate = AR_FREE;
        }
}

/*------------------------------------------------------------------------
 * arp_resolve - use ARP to resolve an IP address into an Ethernet address
 *------------------------------------------------------------------------
 */
status  arp_resolve(
         uint32 ipaddr,                 /* IP address to resolve        */
         byte   mac[ETH_ADDR_LEN]       /* array into which Ethernet    */
        )                               /*  address should be placed    */
{
        struct  arppacket apkt;         /* local packet buffer          */
        int32   i;                      /* index into arpcache          */
        int32   slot;                   /* ARP table slot to use        */
        struct  arpentry  *arptr;       /* ptr to ARP cache entry       */
        int32   msg;                    /* message returned by recvtime */
        byte    ethbcast[] = {0xff,0xff,0xff,0xff,0xff,0xff};

        if (ipaddr == IP_BCAST) {       /* set mac address to b-cast    */
                memcpy(mac, ethbcast, ETH_ADDR_LEN);
                return OK;
        }

        /* Ensure only one process uses ARP at a time */

        wait(arpmutex);
        for (i=0; i<ARP_SIZ; i++) {
```

```
            arptr = &arpcache[i];
            if (arptr->arstate == AR_FREE) {
                continue;
            }
            if (arptr->arpaddr == ipaddr) { /* address is in cache */
                break;
            }
        }

        if (i < ARP_SIZ) {      /* entry was found */

            /* Only one request can be pending for an address */

            if (arptr->arstate == AR_PENDING) {
                signal(arpmutex);
                return SYSERR;
            }

            /* Entry is resolved - handle and return */

            memcpy(mac, arptr->arhaddr, ARP_HALEN);

            signal(arpmutex);
            return OK;
        }

        /* Must allocate a new cache entry for the request */

        slot = arp_alloc();
        if (slot == SYSERR) {
            signal(arpmutex);
            return SYSERR;
        }
        arptr = &arpcache[slot];
        arptr->arstate = AR_PENDING;
        arptr->arpaddr = ipaddr;
        arptr->arpid = currpid;

        /* Release ARP cache for others */

        signal(arpmutex);

        /* Send draft an ARP Request packet */

        memcpy(apkt.arp_ethdst, ethbcast, ETH_ADDR_LEN);
        memcpy(apkt.arp_ethsrc, NetData.ethaddr, ETH_ADDR_LEN);
        apkt.arp_ethtype = ETH_ARP;         /* Packet type is ARP          */
        apkt.arp_htype = ARP_HTYPE;         /* Hardware type is Ethernet   */
        apkt.arp_ptype = ARP_PTYPE;         /* Protocol type is IP         */
        apkt.arp_hlen = 0xff & ARP_HALEN;   /* Ethernet MAC size in bytes  */
        apkt.arp_plen = 0xff & ARP_PALEN;   /* IP address size in bytes    */
        apkt.arp_op = 0xffff & ARP_OP_REQ;  /* ARP type is Request         */
        memcpy(apkt.arp_sndha, NetData.ethaddr, ARP_HALEN);
        apkt.arp_sndpa = NetData.ipaddr;    /* Local IP address            */
        memset(apkt.arp_tarha, '\0', ARP_HALEN); /* Target HA is unknown*/
        apkt.arp_tarpa = ipaddr;            /* Target protocol address     */

        /* Send the packet ARP_RETRY times and await response*/
```

```
msg = recvclr();
for (i=0; i<ARP_SRETRY; i++) {
        write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
        msg = recvtime(ARP_TIMEOUT);
        if (msg == TIMEOUT) {
                continue;
        } else if (msg == SYSERR) {
                return SYSERR;
                } else {        /* entry is resolved */
                        break;
                }
        }

        /* Verify that entry has not changed */

        if (arptr->arpaddr != ipaddr) {
                return SYSERR;
        }

        /* Either return hardware address or TIMEOUT indicator */

        if (i < ARP_RETRY) {
                memcpy(mac, arptr->arhwaddr, ARP_HALEN);
                return OK;
        } else {
                arptr->arstate = AR_FREE;    /* invalidate cache entry */
                return TIMEOUT;
        }
}


/*------------------------------------------------------------------------
 * arp_in - handle an incoming ARP packet
 *------------------------------------------------------------------------
 */
void    arp_in (void) {                 /* currpkt points to the packet */

        struct  arppacket *pktptr;      /* ptr to incoming packet       */
        struct  arppacket apkt;         /* local packet buffer          */
        int32   slot;                   /* slot in cache                */
        struct  arpentry  *arptr;       /* ptr to ARP cache entry       */
        bool8   found;                  /* is the sender's address in   */
                                        /*   the cache?                 */

        /* Insure only one process uses ARP at a time */

        wait(arpmutex);

        pktptr = (struct arppacket *)currpkt;

        /* Search cache for sender's IP address */

        found = FALSE;

for (slot=0; slot < ARP_SIZ; slot++) {
        arptr = &arpcache[slot];

        /* Ignore unless entry valid and address matches */
```

```
        if ( (arptr->arstate != AR_FREE) &&
             (arptr->arpaddr == pktptr->arp_tpapa) ) {
                found = TRUE;
                break;
        }
}

if (found) {    /* Update sender's hardware address */

        memcpy(arptr->arhaddr, pktptr->arp_srdha, ARP_HALEN);

        /* Handle entry that was pending */

        if (arptr->arstate == AR_PENDING) {
                arptr->arstate = AR_RESOLVED;

                /* Notify waiting process */

                send(arptr->arpid, OK);
        }
}

/* For an ARP reply, processing is complete */

if (pktptr->arp_op == ARP_OP_RPLY) {
        signal(arpmutex);
        return;
}

/* ARP request packet: if local machine is not  the target,     */
/*      processing is complete                                  */

if ( (! NetData.ipvalid) || (pktptr->arp_tpapa!=NetData.ipaddr) ) {
        signal(arpmutex);
        return;
}

/* Request has been sent to local machine: add sender's info    */
/*      to cache, if not already present                        */
        if (! found) {
                slot = arp_alloc();
                if (slot == SYSERR) {   /* cache overflow */
                        signal(arpmutex);
                        return;
                }
                arptr = &arpcache[slot];
                arptr->arstate = AR_RESOLVED;
                arptr->arpaddr = pktptr->arp_spapa;
                memcpy(arptr->arhaddr, pktptr->arp_srdha, ARP_HALEN);
        }

        /* Hand-craft an ARP reply packet and send */

        memcpy(epkt.arp_ardhst, pktptr->arp_srdha, ARP_HALEN);
        memcpy(epkt.arp_ethsrc, NetData.ethaddr, ARP_HALEN);
        epkt.arp_ethtype= ETH_ARP;      /* Frame carries ARP    */
        epkt.arp_htype  = ARP_HTYPE;    /* Hardware is Ethernet */
        epkt.arp_ptype  = ARP_PTYPE;    /* Protocol is IP       */
```

```
        apkt.arp_hlen   = ARP_HALEN;    /* Ethernet address size*/
        apkt.arp_plen   = ARP_PALEN;    /* IP address size      */
        apkt.arp_op     = ARP_OP_RPLY;  /* Type is Reply        */

        /* Insert local Ethernet and IP address in sender fields */

        memcpy(apkt.arp_srcha, NetData.ethaddr, ARP_HALEN);
        apkt.arp_srcpa = NetData.ipaddr;

        /* Copy target Ethernet and IP addresses from request packet */

        memcpy(apkt.arp_tarha, pktptr->arp_srcha, ARP_HALEN);
        apkt.arp_tarpa = pktptr->arp_srcpa;

        /* Send the reply */

        write(ETHER0, (char *)&apkt, sizeof(struct  arppacket));
        signal(arpmutex);
        return;
}

/*------------------------------------------------------------------------
 * arp_alloc - find a free slot or kick out an entry to create one
 *------------------------------------------------------------------------
 */
int32   arp_alloc (void) {

        static  int32   nextslot = 0;   /* next slot to try             */
        int32   i;                      /* counts slots in the table    */
        int32   slot;                   /* slot that is selected        */

        /* Search for free slot starting at nextslot */

        for (i=0; i < ARP_SIZ; i++) {
                slot = nextslot++;
                if (nextslot >= ARP_SIZ) {
                        nextslot = 0;
                }
                if (arpcache[slot].arstate == AR_FREE) {
                        return slot;
                }
        }

        /* Search for resolved entry */

        slot = nextslot + 1;
        for (i=0; i < ARP_SIZ; i++) {
                if (slot >= ARP_SIZ) {
                        slot = 0;
                }
                if (arpcache[slot].arstate == AR_RESOLVED) {
                        return slot;
                }
        }

        /* All slots are pending */

        kprintf("ARP cache size exceeded\n\r");
```

```
        return SYSERR;
    }
```

arp_init 当系统启动时，调用函数 arp_init。它确保 ARP 缓存中的所有项为空并创建一个互斥信号量来确保在任何时刻只有一个进程尝试修改 ARP 缓存（如插入一个新项）。函数 arp_resolve 和 arp_in 分别用于处理地址查找和即将到来的 ARP 包。当一个新条目加入到表中时，会调用函数 arp_alloc 为其分配一项。

337 ↑ 344

arp_resolve 当准备发送 IP 数据包时，发送进程调用函数 arp_resolve。arp_resolve 有两个参数：第一个指定请求以太网地址的计算机的 IP 地址，第二个是存储以太网地址的数组指针。

尽管代码看上去比较复杂，但却只有 3 种情况：IP 地址是个广播地址、信息已经存储在 ARP 缓存中、信息还是未知的。对于 IP 广播地址，arp_resolve 将以太网广播地址复制到第二个参数所指定的数组中。如果信息已经存在于缓存中，arp_resolve 将寻找到正确的条目，从条目中将以太网地址复制到调用者的数组里不需要在网络上发送任何数据包就返回到调用者。

当请求的信息并不在缓存中时，arp_resolve 必须在网络中发送数据包以获得信息。这个交换包括发送请求和等待应答。arp_resolve 先在表中创建一个条目，标记这个条目为 AR_PENDING，形成一个 ARP 请求包，在局域网上广播这个数据包，然后等待应答。正如上面所讨论的，arp_resolve 使用 recvtime 来启动等待超时。调用 recvtime 会在应答到达或者定时器超时时返回，而不管哪个先发生。在下一节我们将描述如何处理传入的数据包和如何给等待进程发送消息。

arp_in 这是第二个主要的 ARP 函数。netin 进程检查每个传入的以太网数据包的类型字段。如果发现 ARP 数据包的类型为 0x0806，那么 netin 调用函数 arp_in 去处理这个数据包。arp_in 必须处理两种情况：数据包是另一台计算机发起的请求或者是我们发送请求后的应答。

ARP 协议规定无论何种类型的数据包到达，ARP 必须检测发送者的信息（IP 地址和以太网地址），并且更新本地缓存。如果有进程正在等待响应，那么 arp_in 会发送消息来通知该进程。

因为 ARP 请求是广播的，所有在网络上的计算机都会收到每个请求。因此在更新发送者信息之前，arp_in 会检查请求中的目标 IP 地址以便决定请求是针对本地系统还是针对网络上的其他计算机。如果请求是针对其他计算机，那么 arp_in 不做任何事情就返回。如果传入请求中的目标 IP 地址与本地系统的 IP 地址匹配，那么 arp_in 发送一个 ARP 应答。arp_in 在变量 apkt 中构造一个应答。当数据包中所有字段填满时，就调用在以太网设备上的 write 将应答传送给请求者。

345

17.5 网络数据包的定义

我们网络协议的最小实现结合了 IP、UDP、ICMP 和以太网。也就是说，我们用一个单独的名为 netpacket 的数据结构去描述包含 IP 数据报的以太网数据包——这个数据包可能包含 ICMP 报文或者 UDP 报文。文件 net.h 定义了 netpacket 以及其他常量和数据结构。

```
/* net.h */

/* Constants used in the networking code */

#define ETH_ARP     0x0806              /* Ethernet type for ARP        */
#define ETH_IP      0x0800              /* Ethernet type for IP         */

#define IP_BCADDR   0xffffffff          /* IP local broadcast address   */
#define IP_THIS     0xffffffff          /* "this host" src IP address   */

#define IP_ICMP     1                   /* ICMP protocol type for IP    */
#define IP_UDP      17                  /* UDP protocol type for IP     */

#define IP_ASIZE    4                   /* bytes in an IP address       */
```

```
#define IP_HDR_LEN  20                  /* bytes in an IP header        */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket       {
        byte    net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
        byte    net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  net_ethtype;            /* Ethernet type field          */
        byte    net_ipvh;               /* IP version and hdr length    */
        byte    net_iptos;              /* IP type of service           */
        uint16  net_iplen;              /* IP total packet length       */
        uint16  net_ipid;               /* IP datagram ID               */
        uint16  net_ipfrag;             /* IP flags & fragment offset   */
        byte    net_ipttl;              /* IP time-to-live              */
        byte    net_ippproto;           /* IP protocol (actually type)  */
        uint16  net_ipcksum;            /* IP checksum                  */
        uint32  net_ipsrc;              /* IP source address            */
        uint32  net_ipdst;              /* IP destination address       */
        union {
         struct {
          uint16        net_udpsport;   /* UDP source protocol port     */
          uint16        net_udpdport;   /* UDP destination protocol port*/
          uint16        net_udplen;     /* UDP total length             */
          uint16        net_udpcksum;   /* UDP checksum                 */
          byte  net_udpdata[1500-42];/* UDP payload (1500-above)*/
         };
         struct  {
          byte          net_ictype;     /* ICMP message type            */
          byte          net_iccode;     /* ICMP code field (0 for ping) */
          uint16        net_iccksum;    /* ICMP message checksum        */
          uint16        net_icident;    /* ICMP identifier              */
          uint16        net_icseq;      /* ICMP sequence number         */
          byte          net_icdata[1500-42];/* ICMP payload (1500-above)*/
         };
        };
};
#pragma pack()

extern  struct  netpacket *currpkt;     /* ptr to current input packet  */
extern  bpid32  netbufpool;             /* ID of net packet buffer pool */

struct  network {
        uint32  ipaddr;                 /* IP address                   */
        uint32  addrmask;               /* Subnet mask                  */
        uint32  routeraddr;             /* Address of default router    */
        bool8   ipvalid;                /* Is IP address valid yet?     */
        byte    ethaddr[ETH_ADDR_LEN];  /* Ethernet address             */
};

extern  struct  network NetData;        /* Local network interface      */
```

17.6　网络输入进程

在系统启动时，Xinu 创建网络输入进程 netin。因此，netin 在其他应用进程之前就开始运行。在创建了一个缓冲池并初始化全局变量之后，netin 调用初始化函数 arp_init、udp_init 和 icmp_init。然后分配一个初始网络缓冲区并进入一个重复读取和处理数据包的无限循环中。当从以太网中读取一

个数据包时，netin 用在数据包中的以太网类型字段来决定以太网数据包是否装载了 ARP 报文或者 IP 数据报。如果是 ARP 报文，netin 就调用 arp_in 处理这个数据包。如果是 IP 数据报，netin 就检查这个 IP 头部的校验和是否有效，验证目的 IP 地址是否与广播地址或者本地地址匹配，然后用在 IP 头部的类型字段验证数据报装载的是 UDP 还是 ICMP。如果所有的测试都失败，netin 就转入下一个数据包。如果测试表明信息是有效的，netin 就调用 icmp_in 或者 udp_in 来处理数据包。文件 netin.c 包含了这些代码。

346
↓
347

```
/* netin.c - netin */

#include <xinu.h>

bpid32  netbufpool;                     /* ID of network buffer pool    */

struct  netpacket *currpkt;             /* packet buffer being used now */

struct  network NetData;                /* local network interface      */

/*------------------------------------------------------------------------
 * netin - continuously read the next incoming packet and handle it
 *------------------------------------------------------------------------
 */

process netin(void) {

        status  retval;                 /* return value from function   */

        netbufpool = mkbufpool(PACKLEN, UDP_SLOTS * UDP_QSIZ +
                               ICMP_SLOTS * ICMP_QSIZ + ICMP_OQSIZ + 1);
        if (netbufpool == SYSERR) {
                kprintf("Cannot allocate network buffer pool");
                kill(getpid());
        }

        /* Copy Ethernet address to global variable */

        control(ETHER0, ETH_CTRL_GET_MAC, (int32)NetData.ethaddr, 0);

        /* Indicate that IP address, mask, and router are not yet valid */

        NetData.ipvalid = FALSE;

        NetData.ipaddr = 0;
        NetData.addrmask = 0;
        NetData.routeraddr = 0;

        /* Initialize ARP cache */

        arp_init();

        /* Initialize UDP table */

        udp_init();

/* Initialize ICMP table */

icmp_init();

currpkt = (struct netpacket *)getbuf(netbufpool);
```

```
/* Do forever: read packets from the network and process */

while(1) {
    retval = read(ETHER0, (char *)currpkt, PACKLEN);
    if (retval == SYSERR) {
        panic("Ethernet read error");
    }

    /* Demultiplex on Ethernet type */

    switch (currpkt->net_ethtype) {

        case ETH_ARP:
                arp_in();               /* Handle an ARP packet */
                continue;

        case ETH_IP:
                if (ipcksum(currpkt) != 0) {
                        kprintf("checksum failed\n\r");
                        continue;
                }

                if (currpkt->net_ipvh != 0x45) {
                        kprintf("version failed\n\r");
                        continue;
                }

                if ( (currpkt->net_ipdst != IP_BCAST) &&
                     (NetData.ipvalid) &&
                     (currpkt->net_ipdst != NetData.ipaddr) ) {
                        continue;
                }

                /* Demultiplex ICMP or UDP and ignore others */

                if (currpkt->net_ipproto == IP_ICMP) {
                        icmp_in();      /* Handle an ICMP packet*/
                        } else if (currpkt->net_ipproto == IP_UDP) {
                                udp_in();       /* Handle a UDP packet  */
                        }
                        continue;

                default:        /* Ignore all other Ethernet types */
                        continue;
                }
        }
}


/*------------------------------------------------------------------------
 * ipcksum - compute the IP checksum for a packet
 *------------------------------------------------------------------------
 */

uint16  ipcksum(
         struct  netpacket *pkt         /* ptr to a packet              */
        )
{
```

```
        uint16  *hptr;                  /* ptr to 16-bit header values  */
        int32   i;                      /* counts 16-bit values in hdr  */
        uint32  cksum;                  /* computed value of checksum   */

        hptr = (uint16 *) &pkt->net_ipvh;
        cksum = 0;
        for (i=0; i<10; i++) {
                cksum += (uint32) *hptr++;
        }
        cksum += (cksum >> 16);
        cksum = 0xffff & ~cksum;
        if (cksum == 0xffff) {
                cksum = 0;
        }
        return (uint16) (0xffff & cksum);
}
```

我们的 netin 实现依赖一个全局变量 currpkt，它总是指向正在处理的数据包。也就是说，currpkt 指向当前数据包所使用的缓冲区。在循环开始前，netin 调用 getbuf 获得一个缓冲区，然后将缓冲区的地址分配给 currpkt。在每次迭代中，netin 会将数据包读到当前的缓冲区中。因此，当 netin 调用 arp_in 或 udp_in 时，currpkt 指向应处理的数据包。arp_in 从数据包中抽取信息，并使 currpkt 指向可以重用的缓冲区。如果调用的是 udp_in，那么到达的数据包可能需要入队——进入一个 UDP 表项里。如果当前的数据包入队，那么在返回到 netin 前，udp_in 会分配一个新的缓冲区并将缓冲区的地址分配给 currpkt。

348 / 350

17.7 UDP 表的定义

UDP 维护一张表来记录当前正在使用的 UDP 终端的集合。每一个终端由一个 IP 地址和一个 UDP 端口号组成。UDP 表项用 4 个字段来指明两个终端对，一个是远程计算机，另一个是本地计算机。

为了允许从任意远程计算机上接收数据包的服务器，UDP 进程分配一个 UDP 表项，填写本地的终端信息，但不指明远程终端的信息。为了允许可以和特定的远程计算机通信的客户端，则分配一个表项，填写本地和远程终端信息。

除了终端信息外，每个 UDP 表项都包含一个从远程系统到达的数据包队列（数据包中指定的终端必须与表项中的相匹配）。UDP 表的每个表项都用结构 udpentry 来描述。udp.h 文件定义了该结构和一些关联的符号常量。

351

```
/* udp.h - declarations pertaining to User Datagram Protocol (UDP) */

#define UDP_SLOTS       6               /* num. of open UDP endpoints   */
#define UDP_QSIZ        8               /* packets enqueued per endpoint*/

#define UDP_DHCP_CPORT  68              /* port number for DHCP client  */
#define UDP_DHCP_SPORT  67              /* port number for DHCP server  */


/* Constants for the state of an entry */

#define UDP_FREE        0               /* entry is unused              */
#define UDP_USED        1               /* entry is being used          */
#define UDP_RECV        2               /* entry has a process waiting  */

#define UDP_HDR_LEN     8               /* bytes in a UDP header        */

struct  udpentry {                      /* entry in the UDP endpoint tbl*/
        int32   udstate;                /* state of entry: free/used    */
        uint32  udremip;                /* remote IP address (zero      */
```

```
                                        /*   means "don't care"             */
        uint32  udlocip;                /* local IP address                 */
        uint16  udremport;              /* remote protocol port number      */
        uint16  udlocport;              /* local protocol port number       */
        int32   udhead;                 /* index of next packet to read     */
        int32   udtail;                 /* index of next slot to insert     */
        int32   udcount;                /* count of packets enqueued        */
        pid32   udpid;                  /* ID of waiting process            */
        struct  netpacket *udqueue[UDP_QSIZ];/* circular packet queue       */
};

extern  struct  udpentry udptab[];      /* table of UDP endpoints           */
```

17.8　UDP 函数

在我们的系统中，应用程序使用 UDP 进行所有的通信。因此，UDP 接口允许应用程序发送和接收 UDP 消息，并且应用程序可以充当客户端或服务器的角色。我们的 UDP 软件包括 7 个函数：udp_init、udp_in、udp_register、udp_recv、udp_recvaddr、udp_send 和 udp_release。udp.c 文件中包含这 7 个函数。下面的代码描述了每个 UDP 函数的功能。

```
/* udp.c - udp_init udp_in udp_register udp_recv udp_recvaddr udp_send  */
/*                      udp_release                                     */

#include <xinu.h>

struct  udpentry udptab[UDP_SLOTS];             /* table of UDP endpts  */

/*------------------------------------------------------------------------
 * udp_init - initialize UDP endpoint table
 *------------------------------------------------------------------------
 */
void    udp_init(void) {

        int32   i;                      /* table index */

        for(i=0; i<UDP_SLOTS; i++) {
                udptab[i].udstate = UDP_FREE;
        }

        return;
}

/*------------------------------------------------------------------------
 * udp_in - handle an incoming UDP packet
 *------------------------------------------------------------------------
 */
void    udp_in(void) {                  /* currpkt points to the packet */

        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */

        for (i=0; i<UDP_SLOTS; i++) {
            udptr = &udptab[i];
            if ( (udptr->udstate != UDP_FREE) &&
                 (currpkt->net_udpdport == udptr->udlocport) &&
                   ((udptr->udremport == 0) ||
                       (currpkt->net_udpsport == udptr->udremport)) &&
                   ((udptr->udremip==0)     ||
                       (currpkt->net_ipsrc == udptr->udremip)) ) {
```

```
                /* Entry matches incoming packet */

                if (udptr->udcount < UDP_QSIZ) {
                        udptr->udcount++;
                        udptr->udqueue[udptr->udtail++] = currpkt;
                        if (udptr->udtail >= UDP_QSIZ) {
                                udptr->udtail = 0;
                        }
                        currpkt = (struct netpacket *)getmut(netbufpool);
                        if (udptr->udstate == UDP_RECV) {
                                udptr->udstate = UDP_USED;
                                send (udptr->udpid, OK);
                        }
                        return;
                }
        }
        }

        /* no match - simply discard packet */

        return;
}

/*------------------------------------------------------------------------
 * udp_register - register a remote (IP,port) and local port to receive
 *                incoming UDP messages from the specified remote site
 *                sent to a specific local port
 *------------------------------------------------------------------------
 */
status  udp_register (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport                 /* local UDP protocol port      */
        )
{
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */

        /* See if request already registered */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if (udptr->udstate == UDP_FREE) {
                        continue;
                }
                if ((remport == udptr->udremport) &&
                        (locport == udptr->udlocport) &&
                        (remip   == udptr->udremip  ) ) {

                        /* Entry in table matches request */
                        return SYSERR;
                }
        }

        /* Find a free slot and allocate it */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
```

```
                if (udptr->udstate == UDP_FREE) {
                        udptr->udstate = UDP_USED;
                        udptr->udlocport = locport;
                        udptr->udremport = remport;
                        udptr->udremip = remip;
                        udptr->udcount = 0;
                        udptr->udhead = udptr->udtail = 0;
                        udptr->udpid = -1;
                        return OK;
                }
        }

        return SYSERR;
}

/*------------------------------------------------------------------------
 * udp_recv - receive a UDP packet
 *------------------------------------------------------------------------
 */
int32   udp_recv (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport,                /* local UDP protocol port      */
         char   *buff,                  /* buffer to hold UDP data      */
         int32  len,                    /* length of buffer             */
         uint32 timeout                 /* read timeout in msec         */
        )
{
        intmask mask;                   /* interrupt mask               */
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        umsg32  msg;                    /* message from recvtime()      */
        struct  netpacket *pkt;         /* ptr to packet being read     */
        int32   msglen;                 /* length of UDP data in packet */
        char    *udataptr;              /* pointer to UDP data          */
mask = disable();
for (i=0; i<UDP_SLOTS; i++) {
        udptr = &udptab[i];
        if ((remport == udptr->udremport) &&
                (locport == udptr->udlocport) &&
                (remip   == udptr->udremip  ) ) {

                /* Entry in table matches request */

                break;
        }
}

if (i >= UDP_SLOTS) {
        restore(mask);
        return SYSERR;
}

if (udptr->udcount == 0) {              /* No packet is waiting */
        udptr->udstate = UDP_RECV;
        udptr->udpid = currpid;
        msg = recvclr();
        msg = recvtime(timeout);        /* Wait for a packet    */
        udptr->udstate = UDP_USED;
```

```
        if (msg == TIMEOUT) {
                restore(mask);
                return TIMEOUT;
        } else if (msg != OK) {
                restore(mask);
                return SYSERR;
        }
}

/* Packet has arrived -- dequeue it */

pkt = udptr->udqueue[udptr->udhead++];
if (udptr->udhead >= UDP_SLOTS) {
        udptr->udhead = 0;
}
udptr->udcount--;

/* Copy UDP data from packet into caller's buffer */

msglen = pkt->net_udplen - UDP_HDR_LEN;
udataptr = (char *)pkt->net_udpdata;
        for (i=0; i<msglen; i++) {
                if (i >= len) {
                        break;
                }
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return i;
}

/*------------------------------------------------------------------------
 * udp_recvaddr - receive a UDP packet and record the sender's address
 *------------------------------------------------------------------------
 */
int32   udp_recvaddr (
         uint32 *remip,                 /* loc to record remote IP addr.*/
         uint16 *remport,               /* loc to record remote port    */
         uint16 locport,                /* local UDP protocol port      */
         char   *buff,                  /* buffer to hold UDP data      */
         int32  len,                    /* length of buffer             */
         uint32 timeout                 /* read timeout in msec         */
        )
{
        intmask mask;                   /* interrupt mask               */
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        umsg32  msg;                    /* message from receive()       */
        struct  netpacket *pkt;         /* ptr to packet being read     */
        int32   msglen;                 /* length of UDP data in packet */
        char    *udataptr;              /* pointer to UDP data          */

        mask = disable();
        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if ( (udptr->udremip == 0 ) &&
                     (locport == udptr->udlocport) ) {
```

```
                /* Entry in table matches request */
                break;
        }
}

if (i >= UDP_SLOTS) {
        restore(mask);
                return SYSERR;
        }

        if (udptr->udcount == 0) {      /* no packet is waiting */
                udptr->udstate = UDP_RECV;
                udptr->udpid = currpid;
                msg = recvclr();
                msg = recvtime(timeout);        /* wait for packet */
                udptr->udstate = UDP_USED;
                if (msg == TIMEOUT) {
                        restore(mask);
                        return TIMEOUT;
                } else if (msg != OK) {
                        restore(mask);
                        return SYSERR;
                }
        }

        /* Packet has arrived -- dequeue it */

        pkt = udptr->udqueue[udptr->udhead++];
        if (udptr->udhead >= UDP_SLOTS) {
                udptr->udhead = 0;
        }
        udptr->udcount--;

        /* Record sender's IP address and UDP port number */

        *remip = pkt->net_ipsrc;
        *remport = pkt->net_udpsport;

        /* Copy UDP data from packet into caller's buffer */

        msglen = pkt->net_udplen - UDP_HDR_LEN;
        udataptr = (char *)pkt->net_udpdata;
        for (i=0; i<msglen; i++) {
                if (i >= len) {
                        break;
                }
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return i;
}
/*------------------------------------------------------------------------
 * udp_send - send a UDP packet
 *------------------------------------------------------------------------
 */
status  udp_send (
        uint32 remip,                   /* remote IP address or IP_BCAST*/
                                        /*  for a local broadcast       */
```

```
    uint16 remport,                 /* remote UDP protocol port     */
    uint32 locip,                   /* local IP address             */
    uint16 locport,                 /* local UDP protocol port      */
    char   *buff,                   /* buffer of UDP data           */
    int32  len                      /* length of data in buffer     */
)
{
    struct  netpacket pkt;          /* local packet buffer          */
    int32   pktlen;                 /* total packet length          */
    static  uint16 ident = 1;       /* datagram IDENT field         */
    char    *udataptr;              /* pointer to UDP data          */
    byte    ethbcast[] = {0xff,0xff,0xff,0xff,0xff,0xff};

    /* Compute packet length as UDP data size + fixed header size   */

    pktlen = (char *)pkt.net_udpdata - (char *)&pkt) + len;

    /* Create UDP packet in pkt */

    memcpy(pkt.net_ethsrc, NetData.ethaddr, ETH_ADDR_LEN);
    pkt.net_ethtype = 0x800;        /* Type is IP */
    pkt.net_ipvh = 0x45;            /* IP version and hdr length    */
    pkt.net_iptos = 0x00;           /* Type of service              */
    pkt.net_iplen= pktlen - ETH_HDR_LEN;/* total IP datagram length */
    pkt.net_ipid = ident++;         /* datagram gets next IDENT     */
    pkt.net_ipfrag = 0x0000;        /* IP flags & fragment offset   */
    pkt.net_ipttl = 0xff;           /* IP time-to-live              */
    pkt.net_ipproto = IP_UDP;       /* datagram carries UDP         */
    pkt.net_ipcksum = 0x0000;       /* initial checksum             */
    pkt.net_ipsrc = locip;          /* IP source address            */
    pkt.net_ipdst = remip;          /* IP destination address       */

    /* compute IP header checksum */
    pkt.net_ipcksum = 0xffff & ipcksum(&pkt);

    pkt.net_udpsport = locport;     /* local UDP protocol port      */
    pkt.net_udpdport = remport;     /* remote UDP protocol port     */
    pkt.net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length       */
    pkt.net_udpcksum = 0x0000;      /* ignore UDP checksum          */
    udataptr = (char *) pkt.net_udpdata;
    for (; len>0; len--) {
        *udataptr++ = *buff++;
    }

    /* Set MAC address in packet, using ARP if needed */

    if (remip == IP_BCAST) {        /* set mac address to b-cast    */
        memcpy(pkt.net_ethdst, ethbcast, ETH_ADDR_LEN);

    /* If destination isn't on the same subnet, send to router      */

    } else if ((locip & NetData.addrmask)
                != (remip & NetData.addrmask)) {
        if (arp_resolve(NetData.routeraddr, pkt.net_ethdst)!=OK) {
            kprintf("udp_send: cannot resolve router %08x\n\r",
                    NetData.routeraddr);
            return SYSERR;
        }
```

```
        } else {
                /* Destination is on local subnet    get MAC address */

                if (arp_resolve(remip, pxt.net_ethdst) != OK) {
                        kprintf("udp_send: cannot resolve %08x\n\r",remip);
                        return SYSERR;
                }
        }

        write(ETHER0, (char *)&pkt, pktlen);
        return OK;
}

/*------------------------------------------------------------------------
 * udp_release - release a previously-registered remote IP, remote
 *                      port, and local port (exact match required)
 *------------------------------------------------------------------------
 */
status  udp_release (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport                 /* local UDP protocol port      */
        )
{
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        struct  netpacket *pkt;         /* ptr to packet being read     */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if (udptr->udstate != UDP_USED) {
                        continue;
                }
                if ((remport == udptr->udremport) &&
                        (locport == udptr->udlocport) &&
                        (remip   == udptr->udremip  ) ) {

                        /* Entry in table matches */

                        sched_cntl(DEFER_START);
                        while (udptr->udcount > 0) {
                                pkt = udptr->udqueue[udptr->udhead++];
                                if (udptr->udhead >= UDP_SLOTS) {
                                        udptr->udhead = 0;
                                }
                                freebuf((char *)pkt);
                                udptr->udcount--;
                        }
                        udptr->udstate = UDP_FREE;
                        sched_cntl(DEFER_STOP);
                        return OK;
                }
        }
        return SYSERR;
}
```

udp_init　初始化函数比较容易理解，启动时系统调用 udp_init，udp_init 设置每个 UDP 表项的状态以表明该表项是未被使用的。

udp_in　当携带 UDP 报文的数据包到达时，netin 进程调用函数 udp_in。全局指针 currpkt 指向到达的

数据包。udp_in 搜索 UDP 表来判断表项与当前数据包的 IP 地址和端口号是否匹配。如果不匹配，直接将数据包丢掉——udp_in 返回，netin 将使用同一个缓冲区来读取下一个数据包。如果匹配，udp_in 就将到达的数据包插入与表项相关联的队列中。如果队列已满，udp_in 就返回，数据包将被丢弃，缓冲区将用来读取下一个数据包。当 udp_in 将数据包插入队列中时，它检查是否有进程正在等待到达的数据包（查看 UDP_RECV 的状态），如果有进程正在等待数据包，就给等待进程发送一个报文。注意，只能有一个进程在等待表项。如果有多个进程需要使用表项来进行通信，它们之间必须协调。

353 ~ 361

udp_register 应用程序在使用 UDP 进行通信前，必须调用 udp_register 来指明一个特定的端口，应用程序利用该端口传出接收的数据包。通过指明一个远程 IP 地址，应用程序可以充当客户端。应用程序还可以充当从任意发送者那里接收数据包的服务器。udp_register 负责分配 UDP 表项，在表项中记录远程和本地协议端口号和 IP 地址信息，并创建一个保存到达的数据包的队列。

udp_recv 一旦注册了一个本地端口号后，应用程序可以调用 udp_recv 从表项中提取数据包。调用的参数指明远程端口号和 IP 地址，以及本地端口号。在 udp_recv 的调用中指明的这 3 项信息必须与表项中的这些信息匹配（本地和远程端口号以及 IP 地址必须是以前注册过的）。udp_recv 使用与 ARP 相类似的范式。如果没有数据包在等待（表项的队列是空的），那么 udp_recv 就阻塞，等待一段时间，该时间由上次的调用参数指定。当 UDP 数据包到达时，netin 调用 udp_in。udp_in 中的代码可以找到 UDP 表中合适的表项，如果有一个应用程序处于等待状态，udp_in 就给等待进程发送一个报文。如果数据包在指定的时间内到达，udp_recv 就将 UDP 数据复制到调用者的缓冲区并返回 UDP 数据的长度。如果数据包到达之前定时器已经超时，udp_recv 就返回 TIMEOUT。

udp_recvaddr 当进程充当服务器的角色时，它必须知道与它通信的客户端的地址。服务器进程调用 udp_recvaddr，udp_recvaddr 和 udp_recv 除了返回值不同外，其他功能类似，udp_recvaddr 的返回值不仅包含传入的数据包还包含发送者的地址。服务器可以使用该地址来发送应答消息。

udp_send 进程调用 udp_send 发送 UDP 报文。输入参数指明发送的数据包的远程和本地协议端口号，以及本地和远程 IP 地址、报文的内存地址、报文的长度。udp_send 创建一个以太网数据包，该数据包包含一个携带指定 UDP 报文的 IP 数据报。注意，必须使用有效的地址和端口号，因为 udp_send 只是将信息复制到数据包而不去校验信息是否有效。

udp_release 当进程使用完 UDP 终端后，该进程调用 udp_release 释放表项。如果数据包在表项队列中，那么 udp_release 在释放表项前会将每个数据包返回给缓冲池。

17.9 互联网控制报文协议

Xinu 只处理 ping 程序中的两种报文类型来实现 ICMP：ICMP 回显请求和 ICMP 回显应答。尽管只有两种消息类型，但代码还是有 7 个主要函数：icmp_init、icmp_in、icmp_out、icmp_register、icmp_send、icmp_recv 和 icmp_release。

362

与其他协议栈相比，网络输入函数调用 icmp_init 对 ICMP 初始化。当 ICMP 数据包到达时，网络输入进程调用 icmp_in 来进行处理，应用进程调用 icmp_register 来注册它使用的远程 IP 地址，然后调用 icmp_send 发送 ping 请求，调用 icmp_recv 接收应答。最后，在完成任务后，应用程序调用 icmp_release 释放远程 IP 地址，并允许其他进程使用它。

虽然在 ICMP 的代码中没有表现出来，[⊖]但这些函数遵循了与 UDP 函数相同的适用结构。发出的 ping 数据包中的标识字段是 ping 表的索引。当应答到达时，应答将包含相同的标识，icmp_in 使用它作为数组的索引。因此，与 UDP 不同，ICMP 代码不需要搜索表；当然，仅用标识字段是不够的：在标识了表项后，icmp_in 就验证应答中的 IP 源地址是否与表项中的 IP 地址相匹配。

icmp_out 作为一个单独的进程运行。icmp_out 使用 ARP 解析目的 IP 地址，并通过以太网发送数据包。回忆前面的讨论可知，即使在 ICMP 输出请求的 icmp_out 进程阻塞等待 ARP 应答的情况下，也需要单独的进程来确保 netin 可以继续运行。

⊖ 代码可在网站 xinu.cs.purdue.edu 中获得。

17.10　动态主机配置协议

在启动时，计算机必须获得自己的 IP 地址和默认路由器的 IP 地址。这种用于在启动时获得信息的协议称为动态主机配置协议（DHCP，Dynamic Host Configuration Protocol）。虽然 DHCP 数据包包含了许多字段，但基本的数据包交换是直接的。一台称为主机（host）的计算机广播 DHCP Discover 报文。在本地网络中的 DHCP 服务器通过发送 DHCP Offer 报文来应答，该报文包含主机的 IP 地址、本地网络的 32 位子网掩码和默认路由器的地址。

当系统启动时，应用程序调用 getlocalip 来获取本地 IP 地址。如果 IP 地址在之前已经获得，就仅仅返回该 IP 值；如果主机的 IP 地址是未知的，getlocalip 就使用 DHCP 来获得地址。程序从创建并发送 DHCP Discover 报文开始，然后使用 udp_recv 来等待应答。

文件 dhcp.h 定义 DHCP 报文的结构。整个 DHCP 报文装载在 UDP 报文的有效载荷中，然后装载在 IP 数据报中，再装载在以太网数据包中。 [363]

```
/* dhcp.h - Definitions related to DHCP */

#define DHCP

#pragma pack(2)
struct  dhcpmsg {
        byte    dc_bop;                 /* DHCP bootp op 1=req 2=reply  */
        byte    dc_htype;               /* DHCP hardware type           */
        byte    dc_hlen;                /* DHCP hardware address length */
        byte    dc_hops;                /* DHCP hop count               */
        uint32  dc_xid;                 /* DHCP xid                     */
        uint16  dc_secs;                /* DHCP seconds                 */
        uint16  dc_flags;               /* DHCP flags                   */
        uint32  dc_cip;                 /* DHCP client IP address       */
        uint32  dc_yip;                 /* DHCP your IP address         */
        uint32  dc_sip;                 /* DHCP server IP address       */
        uint32  dc_gip;                 /* DHCP gateway IP address      */
        byte    dc_chaddr[16];          /* DHCP client hardware address */
        byte    dc_bootp[192];          /* DHCP bootp area (zero)       */
        uint32  dc_cookie;              /* DHCP cookie                  */
        byte    dc_opt[1024];           /* DHCP options area (large     */
                                        /*  enough to hold more than    */
                                        /*  reasonable options          */
};
#pragma pack()

extern  struct  netpacket *currpkt;     /* ptr to current input packet  */
extern  bpid32  netbufpool;             /* ID of net packet buffer pool */

#define PACKLEN sizeof(struct netpacket)

extern  uint32  myipaddr;               /* IP address of computer       */
```

如果本地 IP 地址尚未初始化，函数 getlocalip 就创建并发送 DHCP Discover 消息，等待接收回应，并从应答中获取 IP 地址、子网掩码和默认路由器地址，存储到 Netdata，并返回 IP 地址。此段代码在 dhcp.c 中：

```
/* dhcp.c - getlocalip */

#include <xinu.h>

/*------------------------------------------------------------------------
 * getlocalip - use DHCP to obtain an IP address
 *------------------------------------------------------------------------
 */
```

```
uint32  getlocalip(void)
{
        struct  dhcpmsg dmsg;           /* holds outgoing DHCP discover */
                                        /*      message                 */
        struct  dhcpmsg dmsg2;          /* holds incoming DHCP offer    */
                                        /* and outgoing request message */
        uint32  xid;                    /* xid used for the exchange    */
        int32   i;                      /* retry counter                */
        int32   len;                    /* length of data read          */
        char    *optptr;                /* pointer to options area      */
        char    *eop;                   /* address of end of packet     */
        int32   msgtype;                /* type of DCHP message         */
        uint32  addrmask;               /* address mask for network     */
        uint32  routeraddr;             /* default router address       */

        if (NetData.ipvalid) {          /* already have an IP address   */
                return NetData.ipaddr;
        }
        udp_register(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
        memcpy(&xid, NetData.ethaddr, 4);
                                        /* use 4 bytes from MAC as XID  */

        // handcraft a DHCP Discover message in dmsg */

        dmsg.dc_bop = 0x01;             /* Outgoing request             */
        dmsg.dc_htype = 0x01;           /* hardware type is Ethernet    */
        dmsg.dc_hlen = 0x06;            /* hardware address length      */
        dmsg.dc_hops = 0x00;            /* Hop count                    */
        dmsg.dc_xid = xid;              /* xid (unique ID)              */
        dmsg.dc_secs = 0x0000;          /* seconds                      */
        dmsg.dc_flags = 0x0000;         /* flags                        */
        dmsg.dc_cip = 0x00000000;       /* Client IP address            */
        dmsg.dc_yip = 0x00000000;       /* Your IP address              */
        dmsg.dc_sip = 0x00000000;       /* Server IP address            */
        dmsg.dc_gip = 0x00000000;       /* Gateway IP address           */
        memset(&dmsg.dc_chaddr,'\0',16);/* Client hardware address      */
        memcpy(&dmsg.dc_chaddr, NetData.ethaddr, ETH_ADDR_LEN);
        memset(&dmsg.dc_bootp,'\0',192);/* zero the bootp area          */
        dmsg.dc_cookie = 0x63825363;    /* Magic cookie for DHCP        */

        dmsg.dc_opt[0] = 0xff & 53;     /* DHCP message type option     */
        dmsg.dc_opt[1] = 0xff &  1;     /* option length                */
        dmsg.dc_opt[2] = 0xff &  1;     /* DHCP Discover message        */
        dmsg.dc_opt[3] = 0xff &  0;     /* Options padding              */
        dmsg.dc_opt[4] = 0xff & 55;     /* DHCP parameter request list  */
        dmsg.dc_opt[5] = 0xff &  2;     /* option length                */
        dmsg.dc_opt[6] = 0xff &  1;     /* request subnet mask          */
        dmsg.dc_opt[7] = 0xff &  3;     /* request default router addr. */

        dmsg.dc_opt[8] = 0xff &  0;     /* options padding              */
        dmsg.dc_opt[9] = 0xff &  0;     /* options padding              */
        dmsg.dc_opt[10]= 0xff &  0;     /* options padding              */
        dmsg.dc_opt[11]= 0xff &  0;     /* options padding              */

        len = (char *)&dmsg.dc_opt[11] - (char *)&dmsg + 1;

        udp_send(IP_BCAST, UDP_DHCP_SPORT, IP_THIS, UDP_DHCP_CPORT,
                                (char *)&dmsg, len);
```

```
/* Read 3 incoming DHCP messages and check for an offer or      */
/*    wait for three timeout periods if no message arrives.     */

for (i=0; i<3; i++) {
        if ((len=udp_recv(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT,
                        (char *)&dmsg2, sizeof(struct dhcpmsg),
                                3000)) == TIMEOUT) {
                continue;
        }

        /* Check that incoming message is a valid response (ID  */
        /* matches our request)                                 */

        if ( dmsg2.dc_xid != xid) {
                continue;
        }

        eop = (char *)&dmsg2 + len - 1;
        optptr = (char *)&dmsg2.dc_opt;
        msgtype = addrmask = routeraddr = 0;
        while (optptr < eop) {

            switch (*optptr) {
                case 53:        /* message type */
                        msgtype = 0xff & *(optptr+2);
                        break;

                case 1:         /* subnet mask */
                        memcpy(&addrmask, optptr+2, 4);
                        break;
                        case 3:         /* router address */
                                memcpy(&routeraddr, optptr+2, 4);
                                break;
                    }
                    optptr++;   /* move to length octet */
                    optptr += (0xff & *optptr) + 1;
                }

                if (msgtype == 0x02) {  /* offer - send request */
                        dmsg2.dc_opt[0] = 0xff & 53;
                        dmsg2.dc_opt[1] = 0xff &  1;
                        dmsg2.dc_opt[2] = 0xff &  3;
                        dmsg2.dc_hop = 0x01;
                        udp_send(IP_ADDR, UDP_DHCP_SPORT, IP_THIS,
                                UDP_DHCP_CPORT, (char *)&dmsg2,
                                        sizeof(struct dhcpmsg) - 4);

                } else if (dmsg2.dc_opt[2] != 0x05) {   /* if not an ack*/
                        continue;                       /* skip it      */
                }
                if (addrmask != 0) {
                        NetData.addrmask = addrmask;
                }
                if (routeraddr != 0) {
                        NetData.routeraddr = routeraddr;
                }
                NetData.ipaddr = dmsg2.dc_yip;
                NetData.ipvalid = TRUE;
```

```
            udp_release(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
            return htonl(data.ipaddr);
        }
        kprintf("DHCP failed to get response\r\n");
        udp_release(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
        return (uint32)SYSERR;
    }
```

DHCP服务器通过发送请求信息来响应最初的Discover报文。当它接收到应答时，getlocalip检查报文的选项区域。DHCP因为携带信息的选项而不同寻常。特别地，DHCP报文的头包含及某些计算机系统用于初始化网络参数的信息都存储在选项区域内。其中的3个选项是我们的实现关键：选项53定义DHCP报文的类型、选项1指定本地网络使用的子网掩码、选项3指定默认路由器的地址。如果有选项存在，getlocalip就可以从应答中选择需要的信息，为后续的函数调用存储信息，并将IP地址返回给函数调用者。

关于DHCP的细节超出本书的讨论范围。然而要明白，DHCP使用UDP接口的方式与其他应用程序是一样的。即在运行开始前，getlocalip必须调用udp_register记录DHCP将要使用的端口。一旦记录了端口，getlocalip就创建一个DHCP Discover报文并调用udp_send广播这个报文。DHCP Discover报文引发一个DHCP服务器的应答，系统据此从应答中获得它的IP地址。

17.11 观点

本章描述的是互联网协议最简陋的实现。很多细节都省略了，代码也缩减了很多。比如，在我们的实现里，用来定义报文格式的结构将多层协议模块结合起来，且假设底层网络总是以太网。所以，不能把这里的代码看成是一个典型协议的实现，也不能假设相同的结构对一个完整的协议栈会有效。

不过，除了它的局限性外，本章代码阐述了计时操作的重要性。特别是，定时接收函数的运用使整个代码结构变得简单，使整个操作变得更容易理解。如果这个系统没有提供定时接收，那么就将需要更多的进程——一个进程实现计时器功能。另一个进程处理重发。

17.12 总结

即使小型嵌入式系统也使用互联网协议进行通信。因此，大多数操作系统都包括称为协议栈的软件。

本章讨论了支持IP、UDP、ICMP、ARP和DHCP在以太网上运行的有限版本的最小协议栈。以上的协议都是紧密联系的，ICMP和UDP报文都在IP数据报中，DHCP报文在UDP的数据包中。

为了适应异步数据包传输，我们的协议使用网络输入进程netin。netin进程重复地接收以太网数据包，验证数据包头，使用数据包头信息来决定怎样处理数据包。当ARP数据包到达时，netin调用arp_in处理数据包；当UDP数据包到达时，netin调用udp_in处理数据包；当ICMP数据包到达时，netin调用icmp_in处理数据包。对于其他数据包，netin则会忽略。在接收数据包时，我们的实现允许进程指
368 定等待数据包到达的最长时间。超时机制可以用来实现重发：如果应答超时到达，程序就要求重发。

练习

17.1 重写代码以消除对单独ICMP模块进程的需要。提示：当ARP应答到达时，维护一个队列用来缓存IP数据包以及与之对应的ARP表项并且安排要发送的数据包。

17.2 UDP函数要求其用者指定终端信息，如IP地址和协议端口号。重写代码以改变这个范例：让udp_register返回表中条目的索引，加入一些其他函数，如udp_recv将索引当做参数。

17.3 使用计时器进程替代recvtime来重新设计UDP协议。需要多少个进程？解释之。

17.4 Xinu在抽象设备和硬件设备中使用设备范式。重写UDP代码以使用设备范式，其中还包括调用UDP主设备上的open来指定协议端口和IP地址信息，并接收用于通信的伪设备描述符。

17.5 练习17.4中的设备范式能够处理所有的ICMP吗？如果问题限定在ICMP回显（如ping），那么答案
369 会不会改变？解释之。

远程磁盘驱动

我的目标坚定不移：奋斗，搜寻，发现，绝不妥协。

——Alfred, Lord Tennyson

18.1 引言

前面的章节解释了I/O设备和设备驱动的结构。第16章介绍了面向块的设备如何使用DMA，并给出了一个以太网驱动的例子。

本章讨论辅助存储设备（如磁盘或者软盘）的设备驱动的设计，侧重于基本的数据传输操作，第19章介绍系统的高层如何使用磁盘硬件来提供文件和目录。

18.2 磁盘抽象

磁盘硬件提供了一种存储机制的基本抽象模型，该模型具有以下的特点。

- 非易失性（nonvolatile）：即使失去电源，数据仍然存在。
- 面向块（block-oriented）：接口提供了读、写固定大小数据块的能力。
- 多次使用（multi-use）：块可以被读、写多次。
- 随机存取（random-access）：块可以以任何顺序访问。

371

与第16章所描述的以太网硬件一样，磁盘硬件通常使用直接内存存取（DMA）机制来实现数据传输同时不引起CPU中断。与以太网驱动一样，磁盘驱动不需要了解和检查数据块里的内容。相反，驱动只将整个磁盘视为由一个数据块构成的数组。

18.3 磁盘操作驱动支持

在磁盘设备驱动中，磁盘由固定大小的数据块组成，可以通过以下3种基本操作来随机存取数据块：

- 提取（fetch）：将磁盘指定位置的数据块复制到指定内存中。
- 存储（store）：将内存数据复制到磁盘上指定的数据块上。
- 搜索（seek）：将磁头移动到磁盘指定的块。搜索操作对于机电设备（如磁盘）是非常重要的，将磁头移动到未来可能需要的位置是磁盘优化的一个重要机制。不过，随着固态磁盘变得更加广泛，搜索操作的重要性也许正在下降。

磁盘的块大小由磁盘上的扇区决定。工业上已经把512字节作为块的非实标准，在本章中，我们假定块的大小为512字节[⊖]。

18.4 块传输和高层I/O函数

由于硬件提供了块传输，所以就可以定义读（read）和写（write）操作的接口来传输整个数据块。这里的问题是，如何在现有高层的I/O操作中提供块地址。我们可以使用搜索：这要求在调用读、写接口来访问数据块之前调用一次搜索操作将磁头移动到某个块上。不幸的是，要求用户在数据传输之前调用搜索是笨拙和易出错的。因此，为了保持接口的简单易用，我们将扩展读和写操作参数的语义：假设缓冲区足够大，可以放下一个磁盘块的数据，并用第三个参数指定块号。例如，函数调用：

```
read (DISK0, buffer, 5)
```

请求驱动从磁盘上的第5个数据块开始将数据块读到内存中。

372

⊖ 尽管现代的磁盘通常使用大小为4KB的基本块，但硬件提供了一个使用512块的接口。

本章中的驱动将提供基本的磁盘操作：读（read）从磁盘将单个块复制到内存；写（write）把内存中的数据复制到指定的磁盘块。此外，本驱动还具有控制（control）功能，包括格式化磁盘（即销毁所有存储的数据）和同步写请求（如确保所有缓存的数据已经写入磁盘）。

18.5 远程磁盘范式

因为 E2100L 硬件不包括本地磁盘，所以本章介绍的是遵循远程磁盘范式的软件。远程磁盘系统提供与本地磁盘相同的抽象，允许进程读、写磁盘块。不同的是，远程磁盘系统通过网络将请求发送到另一台正在运行的远程磁盘服务器上，而不是使用本地的磁盘硬件。

远程磁盘驱动与传统的磁盘一样，驱动函数分为上半部分和下半部分，这两部分通过一个共享数据结构进行通信。远程磁盘驱动通过网络服务器发送请求和接收响应，并使用高优先级进程进行通信，而不是使用 DMA 硬件来实现下半部分功能。图 18-1 说明了这种结构。

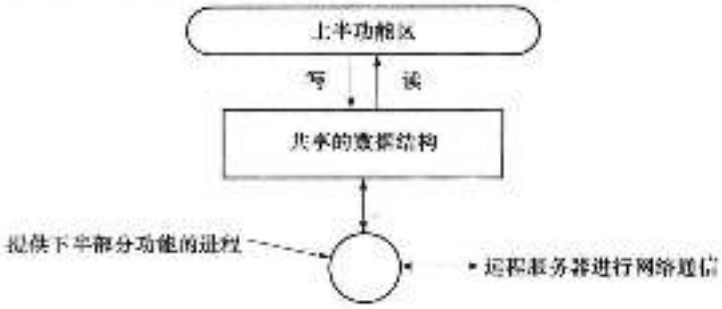


图 18-1 远程磁盘驱动的组织结构

与传统的磁盘驱动一样，在我们例子中的共享数据结构包含了两个关键项：

- 最近存取块的缓存。
- 挂起的请求链表。

375

最近存取块的缓存 因为磁盘操作比处理器慢很多，所以磁盘驱动必须进行优化以避免不必要的数据传输。磁盘的主要优化技术是在共享数据区使用缓存（cache）。缓存存储最近访问的数据块，这样同一块内的后续操作就可以在缓存的副本中进行。例如，如果用户编辑一个文档，该文件所在的磁盘块很可能就在缓存中，这意味着文档编辑应用程序的启动速度远远超过了从硬盘读取块的速度。

挂起的请求链表 与传统的驱动一样，远程磁盘系统允许多个进程访问磁盘，实现同步读操作和异步写操作。也就是说，当驱动读数据块时，进程必须进入等待状态直到需要的数据被读取。当它写一个块时，进程并不阻塞——驱动将需要的数据块副本放到请求链表中，并允许进程继续执行。下半部分进程处理与远程服务器的通信，不断处理请求队列中的任务，并执行指定的操作。因此，数据将在之后的某个时候写入到磁盘中。

18.6 磁盘操作的语义

虽然磁盘驱动可以使用缓存和延迟操作等优化方法，但是磁盘驱动必须始终保证同步接口的正确性。也就是说，驱动必须总是返回系统最后写入的数据。

如果给定块上的读操作和写操作可以表示为以下序列：

$$op_1, op_2, op_3, \cdots, op_k$$

如果 op_i 是对存储块 i 的读操作，则磁盘驱动返回的必须是该计时块在 op_j 操作下所写入的数据，其中 k 是写操作顺序号中最大的值，并且小于 i（即在 op_j 和 op_i 之间的所有操作都是读操作）。为完善这个定义，我们假定在系统启动盘的时刻有一个隐含的写操作。这样，如果系统在调用写块操作前企图读块，那么驱动将返回磁盘上的任何数据。

我们将以上概念称为最后写入语义（last-write semantics）：

磁盘驱动可以使用缓存等技术优化性能，但需保证驱动符合最后写入语义。

示例驱动使用排队技术来确保最后写入语义：请求链表是 FIFO 队列。也就是：　374

> 将新的项添加到请求队列的末尾，前半部分进程不断地选择并执行队列头的项。

因为项总是添加到队尾，所以磁盘驱动处理请求的顺序和请求产生的顺序相同。这样，如果进程 A 对 5 号存储块的数据进行读、而进程 B 在之后对 5 号存储块进行写，那么这两个请求的执行将按照队列中的顺序正确执行。读请求先执行，之后执行写请求。

我们会发现这样的队列规则可以扩展到缓存中：在搜索缓存时，驱动函数总是从缓存队列的头进行。我们的代码将依靠于这个规则以保证进程接收的数据符合最后写入语义。

18.7　驱动数据结构的定义

文件 rdisksys.h 定义远程磁盘系统中使用的常量和数据结构。该文件还定义了磁盘缓冲区的格式。每个缓冲区包含一个指定缓冲区中存储磁盘块大小的头部和用来连接请求链表、缓存或空闲链表的字段。此外，文件中还定义了设备控制块的内容和发到远程服务器的信息格式。　375

```
/* rdisksys.h - definitions for remote disk system pseudo-devices */

#ifndef Nrds
#define Nrds            1
#endif

/* Remote disk block size */

#define RD_BLKSIZ       512

/* Global data for the remote disk server */

#ifndef RD_SERVER_IP
#define RD_SERVER_IP    "255.255.255.255"
#endif

#ifndef RD_SERVER_PORT
#define RD_SERVER_PORT  33124
#endif

#ifndef RD_LOC_PORT
#define RD_LOC_PORT     33124           /* base port number - minor dev */
                                        /*   number is added to insure  */
                                        /*   that each device is unique */
#endif

/* Control block for remote disk device */

#define RD_IDLEN        64              /* Size of a remote disk ID     */
#define RD_BUFFS        64              /* Number of disk buffers       */
#define RD_STACK        8192            /* Stack size for comm. process */
#define RD_PRIO         200             /* Priority of comm. process    */

/* Constants for state of the device */

#define RD_FREE          0              /* device is available          */
#define RD_OPEN          1              /* device is open (in use)      */
#define RD_PEND          2              /* open is pending              */

/* Operations for request queue */

#define RD_OP_READ      1               /* Read operation on req. list  */
#define RD_OP_WRITE     2               /* Write operation on req. list */
#define RD_OP_SYNC      3               /* Sync operation on req. list  */
```

```
/* Status values for a buffer */

#define RD_VALID        0               /* Buffer contains valid data   */
#define RD_INVALID      1               /* Buffer does not contain data */

/* Definition of a buffer with a header that allows the same node to be */
/*  used as a request on the request queue, an item in the cache, or a  */
/*  node on the free list of buffers                                    */

struct  rdbuff  {                       /* Request list node            */
        struct  rdbuff  *rd_next;       /* ptr to next node on a list   */
        struct  rdbuff  *rd_prev;       /* ptr to prev node on a list   */
        int32   rd_op;                  /* operation - read/write/sync  */
        int32   rd_refcnt;              /* reference count of processes */
                                        /*   reading the block          */
        uint32  rd_blknum;              /* block number of this block   */
        int32   rd_status;              /* is buffer currently valid?   */
        pid32   rd_pid;                 /* process that initiated a     */
                                        /*   read request for the block */
        char    rd_block[RD_BLKSIZ];    /* space to hold one disk block */
};

struct  rdblk {
        int32   rd_state;               /* state of device              */
        char    rd_id[RD_IDLEN];        /* Disk ID currently being used */
        int32   rd_seq;                 /* next sequence number to use  */

        /* Request queue head and tail */

        struct  rdbuff  *rd_rnext;      /* head of request queue: next  */
        struct  rdbuff  *rd_rprev;      /*  and previous                */
        struct  rdbuff  *rd_rtnext;     /* tail of request queue: next  */
        struct  rdbuff  *rd_rtprev;     /*  (null) and previous         */

        /* Cache head and tail */

        struct  rdbuff  *rd_chnext;     /* head of cache: next and      */
        struct  rdbuff  *rd_chprev;     /*  previous                    */
        struct  rdbuff  *rd_ctnext;     /* tail of cache: next (null)   */
        struct  rdbuff  *rd_ctprev;     /*  and previous                */

        /* Free list head (singly-linked) */

        struct  rdbuff  *rd_free;       /* ptr to free list             */

        pid32   rd_comproc;             /* process ID of comm. process  */
        sid32   rd_availsem;            /* semaphore ID for avail butts */
        sid32   rd_regsem;              /* semaphore ID for requests    */
        uint32  rd_ser_ip;              /* server IP address            */
        uint16  rd_ser_port;            /* server UDP port              */
        uint16  rd_lo_port;             /* local (client) UDP port      */
        bool8   rd_registered;          /* has UDP port been registered?*/
};


extern  struct  rdblk rdstab[];         /* remote disk control block    */

/* Definitions of parameters used during server access */
```

```
#define RD_RETRIES      3               /* times to retry sending a msg */
#define RD_TIMEOUT      2000            /* wait two seconds for reply   */

/* Control functions for a remote file pseudo device */

#define RDS_CTL_DEL     1               /* Delete (erase) an entire disk*/
#define RDS_CTL_SYNC    2               /* Write all pending blocks     */

/*****************************************************************/
/*      Definition of messages exchanged with the remote disk server    */
/*****************************************************************/
/* Values for the type field in messages */

#define RD_MSG_RESPONSE 0x0100          /* bit that indicates response  */

#define RD_MSG_RREQ     0x0010          /* Read request and response    */
#define RD_MSG_RRES     (RD_MSG_RREQ  RD_MSG_RESPONSE)

#define RD_MSG_WREQ     0x0020          /* Write request and response   */
#define RD_MSG_WRES     (RD_MSG_WREQ  | RD_MSG_RESPONSE)

#define RD_MSG_OREQ     0x0030          /* Open request and response    */
#define RD_MSG_ORES     (RD_MSG_OREQ  | RD_MSG_RESPONSE)

#define RD_MSG_CREQ     0x0040          /* Close request and response   */
#define RD_MSG_CRES     (RD_MSG_CREQ  | RD_MSG_RESPONSE)

#define RD_MSG_DREQ     0x0050          /* Delete request and response  */
#define RD_MSG_DRES     (RD_MSG_DREQ  | RD_MSG_RESPONSE)

#define RD_MIN_RQ       RD_MSG_RREQ     /* Minimum request type         */
#define RD_MAX_RQ       RD_MSG_DREQ     /* Maximum request type         */
/* Message header fields present in each message */

#define RD_MSG_HDR                      /* Common message fields        */\
        uint16  rd_type;                /* message type                 */\
        uint16  rd_status;              /* 0 in req, status in response */\
        uint32  rd_seq;                 /* message sequence number      */\
        char    rd_id[RD_IDLEN];        /* null-terminated disk ID      */

/*****************************************************************/
/*                              Header                           */
/*****************************************************************/
/* The standard header present in all messages with no extra fields */
#pragma pack(2)
struct  rd_msg_hdr {                    /* header fields present in each*/
        RD_MSG_HDR                      /*   remote file system message */
};
#pragma pack()

/*****************************************************************/
/*                              Read                             */
/*****************************************************************/
#pragma pack(2)
struct  rd_msg_rreq     {               /* remote file read request     */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number to read         */
```

```
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_rres     {               /* remote file read reply       */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number that was read   */
        char    rd_data[RD_BLKSIZ];     /* array containing one block   */
};
#pragma pack()

/*****************************************************************/
/*                              Write                            */
/*****************************************************************/
#pragma pack(2)
struct  rd_msg_wreq     {               /* remote file write request    */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number to write        */
        char    rd_data[RD_BLKSIZ];     /* array containing one block   */
};
#pragma pack()
#pragma pack(2)
struct  rd_msg_wres     {               /* remote file write response   */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number that was written*/
};
#pragma pack()

/*****************************************************************/
/*                              Open                             */
/*****************************************************************/
#pragma pack(2)
struct  rd_msg_oreq     {               /* remote file open request     */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_ores     {               /* remote file open response    */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*****************************************************************/
/*                              Close                            */
/*****************************************************************/
#pragma pack(2)
struct  rd_msg_creq     {               /* remote file close request    */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_cres     {               /* remote file close response   */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*****************************************************************/
```

```
/*                              Delete                                  */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_dreq     {               /* remote file delete request   */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_dresp    {               /* remote file delete response  */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()
```

18.8　驱动初始化（rdsInit）

必需初始化的设计应该在驱动的其他部分设计完成后再进行，但我们现在就对初始化函数进行分析，因为这样有助于我们了解共享数据结构。文件 rdsInit.c 包含了驱动初始化代码：

```
/* rdsInit.c  -  rdsInit */

#include <xinu.h>

struct  rdecblk rdetab[Ndsk];

/*------------------------------------------------------------------------
 *  rdsInit  -  initialize the remote disk system device
 *------------------------------------------------------------------------
 */
devcall rdsInit (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rdecblk *rdptr;         /* ptr to device control block  */
        struct  rdbuff  *bptr;          /* ptr to buffer in memory      */
                                        /*    used to form linked list  */
        struct  rdBuff  *pptr;          /* ptr to previous buff on list */
        struct  rdbuff  *buffend;       /* last address in buffer memory*/
        uint32  size;                   /* total size of memory needed  */
                                        /*    buffers                   */

        /* Obtain address of control block */

        rdptr = &rdstab[devptr->dvminor];

        /* Set control block to unused */
rdptr->rd_state = RD_FREE;
rdptr->rd_id[3] = NETLOCH;

/* Set initial message sequence number */

rdptr->rd_seq = 1;

/* Initialize request queue and cache to empty */

rqptr->rd_rhnext = (struct rdbuff *) &rdptr->rd_rtnext;
rdptr->rd_rhprev = (struct rdbuff *)NULL;

rqptr->rd_rtnext = (struct rdBuff *)NULL;
rdptr->rd_rtprev = (struct rdbuff *) &rdptr->rd_rhnext;
```

```
rdptr->rd_ctnext = (struct rdbuff *) &rdptr->rd_ctnext;
rdptr->rd_ctprev = (struct rdbuff *)NULL;

rdptr->rd_ctnext = (struct rdbuff *)NULL;
rdptr->rd_ctprev = (struct rdbuff *) &rdptr->rd_ctnext;

/* Allocate memory for a set of buffers (actually request       */
/*    blocks and link them to form the initial free list        */

size = sizeof(struct rdbuff) * RD_BUFFS;

bptr = (struct rdbuff *)getmem(size);
rdptr->rd_free = bptr;

if ((int32)bptr == SYSERR) {
        panic("Cannot allocate memory for remote disk buffers");
}

buffend = (struct rdbuff *) ((char *)bptr + size);
while (bptr < buffend) {        /* walk through memory */
        pptr = bptr;
        bptr = (struct rdbuff *)
                        (sizeof(struct rdbuff) + (char *)bptr);
        pptr->rd_status = RD_INVALID;   /* buffer is empty      */
        pptr->rd_next = bptr;           /* point to next buffer */
}
pptr->rd_next = (struct rdbuff *) NULL; /* last buffer on list  */

/* Create the request list and available buffer semaphores */
rdptr->rd_availsem = semcreate(RD_BUFFS);
rdptr->rd_reqsem   = semcreate(0);

/* Set the server IP address, server port, and local port */

if ( dot2ip(RD_SERVER_IP, &rdptr->rd_ser_ip) == SYSERR ) {
        panic("invalid IP address for remote disk server");
}

/* Set the port numbers */

rdptr->rd_ser_port = RD_SERVER_PORT;
rdptr->rd_loc_port = RD_LOC_PORT + devptr->dvminor;

/* Specify that the server port is not yet registered */

rdptr->rd_registered = FALSE;

/* Create a communication process */

rdptr->rd_comproc = create(rdsprocess, RD_STACK, RD_PRIO,
                                        "rdsproc", 1, rdptr);

if (rdptr->rd_comproc == SYSERR) {
        panic("Cannot create remote disk process");
}
resume(rdptr->rd_comproc);

return OK;
}
```

除了初始化数据结构外，rdsInit 还实现了另外 3 个重要任务。首先，它分配一组磁盘缓冲区并将其链接到空闲链表；其次，它创建与服务器通信的高优先级进程；最后，它创建两个用来控制进程的信号量。信号量 rd_reqsem 用于保护请求链表。信号量从 0 开始，每当有一个新的请求添加到请求链表时就加 1。通信进程在从链表中取出数据之前会先等待 rd_reqsem 信号量，这意味着如果链表为空，进程将被阻塞。

另一个信号量为 rd_availsem，用来记录可用的缓冲区的数量（例如，空闲的或者在缓存中的）。最初始，RD_BUFFS 缓冲区都在空闲链表中，rd_availsem 的值等于 RD_BUFFS。当需要缓存区时，调用者等待信号量。不在缓存中的所有缓冲区都可用。由于进程正在等待数据，缓冲区必须驻留在缓存中。后面我们将看到缓存和信号量将如何使用。

376 ‹ 383

18.9　上半部打开函数(rdsOpen)

远程磁盘服务器允许多客户端同时访问服务器。每个客户端有一个唯一的身份标识字符串以使服务器区分各个客户端。示例代码允许用户通过在磁盘设备上调用打开（open）函数来指定其身份标识字符串，而不是用某个硬件值（如以太网地址）作为其唯一的字符串。将身份标识符（ID）与硬件区分开的最大好处是可移植性——远程磁盘 ID 可以绑定到操作系统的镜像，这意味着把镜像从一台物理计算机移动到另外一台计算机不会发现系统正在使用的磁盘。

当进程对某个远程磁盘设备调用打开（open）函数时，第二个参数为 ID 字符串，该字符串将被复制到设备控制块中，只要设备处于打开状态，此 ID 就可以一直使用。当然，我们可以关闭远程磁盘设备并用一个新的 ID 重新打开它（如连接到另一个远程磁盘上）。然而，对于多数系统而言，我们期望一旦打开一个远程磁盘设备就永远不要关闭它。文件 rdsOpen.c 包含如下代码：

```
/* rdsOpen.c  -  rdsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsOpen - open a remote disk device and specify an ID to use
 *------------------------------------------------------------------------
 */

devcall rdsOpen (
        struct dentry  *devptr,         /* entry in device switch table */
        char   *diskid,                 /* disk ID to use               */
        char   *mode                    /* unused for a remote disk     */
        )
{
        struct  rdscblk *rdptr;         /* ptr to control block entry   */
        struct  rd_msg_crcq msg;        /* message to be sent           */
        struct  rd_msg_crcs resp;       /* buffer to hold response      */
        int32   retval;                 /* return value from rdscomm    */
        int32   len;                    /* counts chars in diskid       */
        char    *idto;                  /* ptr to ID string copy        */
        char    *idfrom;                /* pointer into ID string       */

        rdptr = &rdstab[devptr->dvminor];

        /* Reject if device is already open */
        if (rdptr->rd_state != RD_FREE) {
                return SYSERR;
        }
        rdptr->rd_state = RD_PEND;

        /* Copy disk ID into free table slot */
```

```
        idto = rdptr->rd_id;
        idfrom = diskid;
        len = 0;
        while ( (*idto++ = *idfrom++) != NULLCH) {
                len++;
                if (len >= RD_IDLEN) {  /* ID string is too long */
                        return SYSERR;
                }
        }

        /* Verify that name is non-null */

        if (len == 0) {
                return SYSERR;
        }

        /* Hand-craft an open request message to be sent to the server */

        msg.rd_type = htons(RD_MSG_OREQ);/* Request an open              */
        msg.rd_status = htons(0);
        msg.rd_seq = 0;                  /* rdscomm fills in an entry    */
        idto = msg.rd_id;
        memset(idto, NULLCH, RD_IDLEN);/* Initialize ID to zero bytes    */

        idfrom = diskid;
        while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID to req.  */
                ;
        }

        /* Send message and receive response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                                sizeof(struct rd_msg_oreq),
                        (struct rd_msg_hdr *)&resp,
                                sizeof(struct rd_msg_oresp),
                                rdptr );

        /* Check response */
        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                return SYSERR;
        } else if (ntohs(resp.rd_status) != 0) {
                return SYSERR;
        }

        /* Change state of device to indicate currently open */

        rdptr->rd_state = RD_OPEN;

        /* Return device descriptor */

        return devptr->dvnum;
}
```

18.10 远程通信函数(rdscomm)

rdsOpen 为打开本地远程磁盘设备（local remote disk device）的一步，用于与远程服务器交换报文。

它申请开请求报文放在本地变量 msg 中，并调用 rdscomm 将该报文转发给服务器。rdscomm 的参数包括一个传出的报文、存储应答的缓冲区以及这两者各自的长度。rdscomm 将传出的报文发送给服务器，并等待其应答。若应答有效，rdscomm 就返回应答的长度给调用者；否则，它返回 SYSERR 以指示发生错误，或者返回 TIMEOUT 以表明未收到应答。文件 rdscomm.c 包含如下代码：

```
/* rdscomm.c  -  rdscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdscomm - handle communication with a remote disk server (send a
 *             request and receive a reply, including sequencing and
 *             retries)
 *------------------------------------------------------------------------
 */
status  rdscomm (
          struct rd_msg_hdr *msg,       /* message to send              */
          int32             mlen,       /* message length               */
          struct rd_msg_hdr *reply,     /* buffer for reply             */
          int32             rlen,       /* size of reply buffer         */
          struct rdeublk    *rdptr      /* ptr to device control block  */
        )
{
        int32   i;                      /* counts retries               */
        int32   retval;                 /* return value                 */
        int32   seq;                    /* sequence for this exchange   */
        uint32  localip;                /* local IP address             */
        int16   rtype;                  /* reply type in host byte order*/
        bool8   xmit;                   /* Should we transmit again?    */

        /* For the first time after reboot, register the server port */

        if ( ! rdptr->rd_registered ) {
                retval = udp_register(0, rdptr->rd_ser_port,
                                rdptr->rd_loc_port);
                rdptr->rd_registered = TRUE;
        }

        if ( (int32)(localip = getlocalip()) == SYSERR ) {
                return SYSERR;
        }

        /* Assign message next sequence number */

        seq = rdptr->rd_seq++;
        msg->rd_seq = htons((seq));

        /* Repeat RD_RETRIES times: send message and receive reply */

        xmit = TRUE;
        for (i=0; i<RD_RETRIES; i++) {
            if (xmit) {

                /* Send a copy of the message */

                retval = udp_send(rdptr->rd_ser_ip, rdptr->rd_ser_port,
                        localip, rdptr->rd_loc_port, (char *)msg, mlen);
```

```
            if (retval == SYSERR) {
                kprintf("Cannot send to remote disk server\n\r");
                return SYSERR;
            }
        } else {
            xmit = TRUE;
        }
        /* Receive a reply */

        retval = udp_recv(0, rdptr->rd_ser_port,
            rdptr->rd_loc_port, (char *)reply, rlen,
            RD_TIMEOUT);

        if (retval == TIMEOUT) {
            continue;
        } else if (retval == SYSERR) {
            kprintf("Error reading remote disk reply\n\r");
            return SYSERR;
        }

        /* Verify that sequence in reply matches request */

        if (ntohl(reply->rd_seq) < seq) {
            xmit = FALSE;
        } else if (ntohl(reply->rd_seq) != seq) {
                continue;
        }

        /* Verify the type in the reply matches the request */

        rtype = ntohs(reply->rd_type);
        if (rtype != ( ntohs(msg->rd_type) | RD_MSG_RESPONSE) ) {
            continue;
        }

        /* Check the status */

        if (ntohs(reply->rd_status) != 0) {
            return SYSERR;
        }

        return OK;
    }

    /* Retries exhausted without success */

    kprintf("Timeout on exchange with remote disk server\n\r");
    return TIMEOUT;
}
```

384 ~ 388

rdscomm 包含本地 IP 地址（使用 UDP 协议），给下一个报文分配序列号，然后进入一个迭代 RD_RETRIES 次的循环。每次迭代，rdscomm 调用 udp_send 将报文的副本传送给服务器，并调用 udp_recv 来接收应答。如果应答到达，rdscomm 就验证应答的参数是否与请求的类型相匹配，应答的序列号是否与已发送的序列号相匹配，以及状态值是否标识成功（比如，为 0）。如果应答有效，rdscomm 就返回 OK 给调用者；否则，返回一个错误指示。

18.11 上半部写函数(rdsWrite)

由于远程磁盘系统提供异步写操作，所以上半部写函数很容易理解。文件 rdsWrite.c 包含如下代码：

```
/* rdsWrite.c  -  rdsWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsWrite - Write a block to a remote disk
 *------------------------------------------------------------------------
 */
devcall rdsWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer that holds a disk blk */
          int32 blk                     /* block number to write        */
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *bptr;          /* ptr to buffer on a list      */
        struct  rdbuff  *pptr;          /* ptr to previous buff on list */
        struct  rdbuff  *nptr;          /* ptr to next buffer on list   */
        bool8   found;                  /* was buff found during search?*/

        /* If device not currently in use, report an error */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* If request queue already contains a write request */
        /*    for the block, replace the contents            */
        bptr = rdptr->rd_chnext;
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                if ( (bptr->rd_blknum == blk) &&
                     (bptr->rd_op == RD_OP_WRITE) ) {
                        memcpy(bptr->rd_block, buff, RD_BLKSIZ);
                        return OK;
                }
                bptr = bptr->rd_next;
        }


        /* Search cache for cached copy of block */

        bptr = rdptr->rd_chnext;
        found = FALSE;
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                if (bptr->rd_blknum == blk) {
                        if (bptr->rd_refcnt <= 0) {
                                pptr = bptr->rd_prev;
                                nptr = bptr->rd_next;

                                /* Unlink node from cache list and reset*/
                                /*  the available semaphore accordingly */

                                pptr->rd_next = bptr->rd_next;
                                nptr->rd_prev = bptr->rd_prev;
                                semreset(rdptr->rd_availsem,
                                        semcount(rdptr->rd_availsem) - 1);
                                found = TRUE;
                        }
```

```
                break;
            }
            bptr = bptr->rd_next;
        }

        if ( !found ) {
            bptr = rdsbufalloc(rdptr);
        }

        /* Create a write request */

        memcpy(bptr->rd_block, buff, RD_BLKSIZ);
        bptr->rd_op = RD_OP_WRITE;
        bptr->rd_refcnt = 0;
        bptr->rd_blknum = blk;
        bptr->rd_status = RD_VALID;
        bptr->rd_pid = getpid();

        /* Insert new request into list just before tail */

        pptr = rdptr->rd_rtprev;
        rdptr->rd_rtprev = bptr;
        bptr->rd_next = pptr->rd_next;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;

        /* Signal semaphore to start communication process */

        signal(rdptr->rd_regsem);
        return OK;
}
```

这段代码首先考虑如下情况：请求队列已经包含对同一个块的挂起写请求。注意：任意时刻对请求队列中的某个给定块只能有一个写请求。这段代码从头到尾对队列进行遍历搜索。如果发现对块的写请求，rdsWrite 就用一个新的数据替换所请求的内容，然后返回。

在搜索请求队列后，rdsWrite 会首先检查缓存。如果指定的块在缓存中，则该缓存的副本必须置为无效。这段代码顺序地对缓存进行遍历。如果发现一个匹配，rdsWrite 就从缓存中删除该缓冲区。不过，rdsWrite 并不是将该缓冲区移至空闲链表，而是使用该缓冲区生成一个请求。如果没有匹配，rdsWrite就调用 rdsbufalloc 为请求分配一个新的缓冲区。

rdsWrite 的最后一部分生成一个写请求（write request），并将其插入请求队列的末尾。为了方便调试，代码填写了请求的各个字段。例如，进程 ID 字段尽管未被用到，但仍被设置。

18.12 上半部读函数(rdsRead)

第二个主要的上半部函数与读操作有关。读比写更复杂，因为输入是同步的：一个试图从磁盘中读取数据的进程必须等待直到该数据为可用的。一个等待进程的同步操作要用到发送（send）和接收（receive）函数。请求队列中的每个结点包含有进程 ID 字段。当进程调用读函数时，驱动代码创建一个读请求。该请求包含调用者的进程 ID。然后，进程将请求插入请求队列，并调用接收函数来等待响应。当请求到达队列头时，远程磁盘通信进程将报文发送到服务器并接收包含指定块的响应。通信进程将该块复制到包含原始请求的缓冲区，把该缓冲区移至缓存中，再使用发送函数将报文发送给带有缓冲区地址的等待进程。等待进程接收报文，提取数据的副本，并返回给调用读的函数。

389 – 391

上述方案的效率不高，因为缓冲区是动态使用的。为了理解这个问题，假设一个低优先级的进程因为要读数据块 5 而被阻塞。最后，通信进程从服务器获得数据块 5，将其存储在缓存中，并向这个等

待进程发送报文。然而，假设请求在请求队列中，同时高优先级应用程序开始执行，这就意味着低优先级的进程不会被执行。不幸的是，如果高优先级进程一直使用磁盘缓冲区，那么保存数据块 5 的缓冲区也将被分配出去。

因为远程磁盘系统允许并发访问，所以这个问题将变得更加严重：当一个进程等待读取一个数据块时，另一进程也可以试图读取同一个块。这样，当通信进程最终从服务器检索某个块的副本时，需要通知与之相关的多个进程。

本书示例程序使用了引用计数（reference count）来解决对同一个块的多次请求：每个缓冲区的头包含一个整型值来记录读取某个块的进程数。当一个进程结束复制数据时，该进程递减引用计数。文件 rdsRead.c 中的代码显示了进程如何产生请求，将其插入请求链表的末尾，等待请求满足，并从请求将数据复制到每个调用者的缓冲区。在本章的后面部分，我们将看到如何管理引用计数。

```
/* rdsRead.c  -  rdsRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsRead - Read a block from a remote disk
 *------------------------------------------------------------------------
 */
devcall rdsRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer to hold disk block    */
          int32 blk                     /* block number of block to read*/
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *bptr;          /* ptr to buffer possibly on    */
                                        /*  the request list            */
        struct  rdbuff  *nptr;          /* ptr to "next" node on a      */
                                        /*  list                        */
        struct  rdbuff  *pptr;          /* ptr to "previous" node on    */
                                        /*  a list                      */
        struct  rdbuff  *cptr;          /* ptr used to walk the cache   */

        /* If device not currently in use, report an error */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* Search the cache for specified block */

        bptr = rdptr->rd_chhead;
        while (bptr != (struct rdbuff *)&rdptr->rd_chtail) {
                if (bptr->rd_blknum == blk) {
                        if (bptr->rd_status == RD_INVALID) {
                                return SYSERR;
                        }
                        memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                        return OK;
                }
                bptr = bptr->rd_next;
        }

        /* Search the request list for most recent occurrence of block */

        bptr = rdptr->rd_rtprev;        /* start at tail of list */
```

```
while (bptr != (struct rdbuff *)&rdptr->rd_rhnext) {
    if (bptr->rd_blknum == blk)  {

        /* If most recent request for block is write, copy data */

        if (bptr->rd_op == RD_OP_WRITE) {
                memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                return OK;
        }
        break;
    }
    bptr = bptr->rd_prev;
}

/* Allocate a buffer and add read request to tail of req. queue */

bptr = rdbufalloc(rdptr);
bptr->rd_op = RD_OP_READ;
bptr->rd_refcnt = 1;
bptr->rd_blknum = blk;
bptr->rd_status = RD_INVALID;
bptr->rd_pid = getpid();

/* Insert new request into list just before tail */

pptr = rdptr->rd_rtprev;
rdptr->rd_rtprev = bptr;
bptr->rd_next = pptr->rd_next;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;

/* Prepare to receive message when read completes */

recvclr();

/* Signal semephore to start communication process */

signal(rdptr->rd_reqsem);

/* Block to wait for message */

bptr = (struct rdbuff *)receive();
if (bptr == (struct rdbuff *)SYSERR) {
        return SYSERR;
}
memcpy(buff, bptr->rd_block, RD_BLKSIZ);
bptr->rd_refcnt--;
if (bptr->rd_refcnt <= 0) {

        /* Look for previous item in cache with the same block  */
        /*    number to see if this item was only being kept    */
        /*    until pending read completed                      */

        cptr = rdptr->rd_chnext;
        while (cptr != bptr) {
                if (cptr->rd_blknum == blk) {

                        /* Unlink from cache */

                        pptr = bptr->rd_prev;
                        nptr = bptr->rd_next;
                        pptr->rd_next = nptr;
```

```
                        nptr->rd_prev = pptr;

                        /* Add to the free list */

                        kptr->rd_next = rdptr->rd_free;
                        rdptr->rd_free = kptr;
                    }
                }
        }
        return OK;
}
```

rdsRead 开始先处理了两种特殊情形。第一，如果被请求的块在缓存中，rdsRead 就提取数据的副本并返回。第二，如果请求链表包含某个写指定数据块的请求，rdsRead 就从缓冲区提取数据的副本并返回。最后，rdsRead 产生一个读请求，将其插入请求链表的末尾，并等待从通信进程发来的报文。

这段代码还处理了一个更细节的情形：引用计数为 0 并已对同一个数据块的下一个读处于缓存的某个较新（分配）的缓冲区中。如果这种情况发生，较新版本将被用于后面的读操作。因此，rdsRead 必须从缓存中提取较早（分配）的缓冲区并把它移至空闲链表中。

18.13　刷新挂起的请求

因为写操作并不需要等待数据传送，所以当写操作完成的时候，驱动不需要通知进程写操作结束。但是，对于软件来说确保数据安全存储是很重要的事情。比如，操作系统通常在关机之前必须确保写操作已经完成。

为了确保所有的磁盘传输已经完成，驱动包含了一个原语，该原语的作用是：在所有的请求完成之前，阻塞其他的进程对其进行调用。由于磁盘"同步"并非单纯的数据传送操作，所以我们使用了一个名为控制（control）的高层操作。进程调用：

```
control(disk_device,RD_SYNC)
```

在指定的设备满足当前请求之前，驱动将暂停调用进程。一旦挂起操作完成，调用就返回。

392 ~ 395

18.14　上半部控制函数(rdsControl)

正如上面讨论的那样，例子中的驱动程序类供了两个控制（control）函数：一个用于清除磁盘，一个用于将数据同步到磁盘上（即强制完成所有的写操作）。文件 rdsControl.c 包含如下代码：

```
/* rdsControl.c - rdsControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsControl - Provide control functions for the remote disk
 *------------------------------------------------------------------------
 */
devcall rdsControl (
        struct dentry  *devptr,         /* entry in device switch table */
        int32  func,                    /* a control function           */
        int32  arg1,                    /* argument #1                  */
        int32  arg2                     /* argument #2                  */
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *kptr;          /* ptr to buffer that will be   */
                                        /*  placed on the req. queue    */
        struct  rdbuff  *pptr;          /* ptr to *previous* node on    */
                                        /*  a list                      */
```

```
struct  rd_msg_dreq msg;        /* buffer for delete request    */
struct  rd_msg_dres resp;       /* buffer for delete response   */
char    *to, *from;             /* used during name copy        */
int32   retval;                 /* return value                 */

/* Verify that device is currently open */

rdptr = &rdstab[devptr->dvminor];
if (rdptr->rd_state != RD_OPEN) {
        return SYSERR;
}

switch (func) {

/* Synchronize writes */

case RDS_CTL_SYNC:
        /* Allocate a buffer to use for the request list */

        bptr = rdsbufalloc(rdptr);
        if (bptr == (struct rdbuff *)SYSERR) {
                return SYSERR;
        }

        /* Form a sync request */

        bptr->rd_op = RD_OP_SYNC;
        bptr->rd_refcnt = 1;
        bptr->rd_blknum = 0;            /* unused */
        bptr->rd_status = RD_INVALID;
        bptr->rd_pid = getpid();

        /* Insert new request into list just before tail */

        pptr = rdptr->rd_rtprev;
        rdptr->rd_rtprev = bptr;
        bptr->rd_next = pptr->rd_next;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;

        /* Prepare to wait until item is processed */

        recvclr();
        resume(rdptr->rd_comproc);

        /* Block to wait for message */

        bptr = (struct rdbuff *)receive();
        break;

/* Delete the remote disk (entirely remove it) */

case RDS_CTL_DEL:

        /* Handcraft a message for the server that requests     */
        /*      deleting the disk with the specified ID         */

        msg.rd_type = htons(RD_MSG_DREQ);/* Request deletion    */
        msg.rd_status = htons(0);
        msg.rd_seq = 0; /* rdscomm will insert sequence # later */
        to = msg.rd_id;
        memset(to, NULLCH, RD_IDLEN);   /* initialize to zeroes */
```

```
        from = rdptr->rd_id;
        while ( (*to++ = *from++) != NULLCH ) { /* copy ID      */
                ;
        }

        /* Send message and receive response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                         sizeof(struct rd_msg_dreq),
                 (struct rd_msg_hdr *)&rcmp,
                         sizeof(struct rd_msg_dres),
                         rdptr);

        /* Check response */

        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file delete\n\r");
                return SYSERR;
        } else if (ntohs(resp.rd_status) != 0) {
                return SYSERR;
        }

        /* Close local device */

        return rdsClose(devptr);

    default:
        kprintf("rtsControl: function %d not valid\n\r", func);
        return SYSERR;
    }

    return OK;
}
```

每个函数中的代码看起来都很相似。清除磁盘的代码与函数 rdsOpen 中的代码很相似——函数 rdsOpen 为服务器创建消息，并使用函数 rdscomm 来传送消息。同步磁盘写操作的代码与函数 rdsRead 中的代码很相似——rdsRead 创建请求，并将其加入请求队列，调用函数 receive 来等待应答。一旦应答到达，函数 rdsControl 就唤醒 rdsClose 来关闭本地设备，并返回到它的调用者。

396 ; 398

18.15　分配磁盘缓冲区(rdsbufalloc)

正如我们所见，当需要分配缓冲区的时候，驱动函数调用函数 rdsbufalloc。rdsbufalloc.c 文件包含如下代码：

```
/* rdsbufalloc.c - rdsbufalloc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsbufalloc - allocate a buffer from the free list or the cache
 *------------------------------------------------------------------------
 */
struct rdbuff *rdsbufalloc (
          struct rdscblk *rdptr         /* ptr to device control block  */
        )
{
        struct  rdbuff  *bptr;          /* ptr to a buffer              */
        struct  rdbuff  *pptr;          /* ptr to previous buffer       */
```

```
	struct	rdbuff	*nptr;		/* ptr to next buffer		*/

	/* Wait for an available buffer */

	wait(rdptr->rd_availsem);

	/* If free list contains a buffer, extract it */

	bptr = rdptr->rd_free;

	if ( bptr != (struct rdbuff *)NULL ) {
		rdptr->rd_free = bptr->rd_next;
		return bptr;
	}

	/* Extract oldest item in cache that has ref count zero (at	*/
	/*   least one such entry must exist because the semaphore	*/
	/*   had a nonzero count)				*/

	bptr = rdptr->rd_ctprev;
	while (bptr != (struct rdbuff *) &rdptr->rd_chnext) {
		if (bptr->rd_refcnt <= 0) {

			/* Remove from cache and return to caller */
			pptr = bptr->rd_prev;
			nptr = bptr->rd_next;
			pptr->rd_next = nptr;
			nptr->rd_prev = pptr;
			return bptr;
		}
		bptr = bptr->rd_prev;
	}
	panic("Remote disk cannot find an available buffer");
	return (struct rdbuff *)SYSERR;
}
```

记住，信号量计数器的可用缓冲区要么在空闲链表中，要么在缓存中。在等待信号量后，rdsbufalloc 函数首先对空闲链表进行检查。如果空闲链表不为空，rdsbufalloc 就提取第一个缓冲区，并且将它返回。如果空闲链表为空，rdsbufalloc 在缓存中搜索一个可用的缓冲区，禁收该缓冲区，并且将它返回给 rdsbufalloc 函数的调用者。如果搜索完成没找到可用的缓冲区，那么系统认为信号量计数器出错，rdsbufalloc 调用 panic 来终止系统。

18.16 上半部关闭函数(rdsClose)

进程调用关闭（close）函数来关闭远程磁盘设备，停止所有的通信。文件 rdsClose.c 包含如下代码：

```
/* rdsClose.c  -  rdsClose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsClose - Close a remote disk device
 *------------------------------------------------------------------------
 */
devcall rdsClose (
	  struct dentry *devptr		/* entry in device switch table */
	)
{
	struct	rdscblk *rdptr;		/* ptr to control block entry	*/
	struct	rdbuff	*bptr;		/* ptr to buffer on a list	*/
	struct	rdbuff	*nptr;		/* ptr to next buff on the list */
	int32	nmoved;			/* number of buffers moved	*/

	/* Device must be open */
```

```
        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* Request queue must be empty */

        if (rdptr->rd_rhnext != (struct rdbuff *)&rdptr->rd_rtnext) {
                return SYSERR;
        }

        /* Move all buffers from the cache to the free list */

        bptr = rdptr->rd_chnext;
        moved = 0;
        while (bptr != (struct rdbuff *)&rdptr->rd_rtnext) {
                moved++;

                /* Unlink buffer from cache */

                nptr = bptr->rd_next;
                (bptr->rd_prev)->rd_next = nptr;
                nptr->rd_prev = bptr->rd_prev;

                /* Insert buffer into free list */

                bptr->rd_next = rdptr->rd_free;

                rdptr->rd_free = bptr;
                bptr->rd_status = RD_INVALID;

                /* Move to next buffer in the cache */

                bptr = nptr;
        }

        /* Set the state to indicate the device is closed */

        rdptr->rd_state = RD_FREE;
        return OK;
}
```

399 / 401

为了关闭远程磁盘设备，必须把所有的缓冲区都移回到空闲链表中并将控制块中的状态字段设置为 RD_FREE。我们的系统虽然实现了从缓存中移除缓冲区，但没有处理请求链表。相反我们要求用户等待直到所有的请求完成且请求链表为空后才可以调用 rdsClose。同步函数 RDS_CTL_SYNC[⊖] 提供了一种等待请求链表为空的方法。

18.17　下半部通信进程(rdsprocess)

如例子中实现的那样，每个远程磁盘设备都有它自己的控制块、磁盘缓冲区集合和远程通信进程。因此，假设远程磁盘进程只需要处理单独队列中的请求。尽管下面的代码可能看起来冗长繁复，但是算法非常浅显易懂：持续等待请求信号量，检查队列头的请求类型，执行读、写或同步操作。文件 rdsprocess.c 包含如下代码：

```
/* rdsprocess.c  -  rdsprocess */

#include <xinu.h>
```

⊖　文件 rdsControl.c 中的同步代码可以在 18.14 节找到。

```
/*------------------------------------------------------------------------
 * rdsprocess - high-priority background process that repeatedly extracts
 *              an item from the request queue and sends the request to
 *              the remote disk server
 *------------------------------------------------------------------------
 */
void    rdsprocess (
          struct rdscblk    *rdptr      /* ptr to device control block  */
        )
{
        struct  rd_msg_wreq msg;        /* message to be sent           */
                                        /*  (includes data area)        */
        struct  rd_msg_rres resp;       /* buffer to hold response      */
                                        /*  (includes data area)        */
        int32   retval;                 /* return value from rdscomm    */
        char    *idto;                  /* ptr to ID string copy        */
        char    *idfrom;                /* ptr into ID string           */
        struct  rdbuff  *bptr;          /* ptr to buffer at the head of */
                                        /*   the request queue          */
        struct  rdbuff  *nptr;          /* ptr to next buffer on the    */
                                        /*   request queue              */
        struct  rdbuff  *pptr;          /* ptr to previous buffer       */
        struct  rdbuff  *qptr;          /* ptr that runs along the      */
                                        /*   request queue              */
        int32   i;                      /* loop index                   */

        while (TRUE) {                  /* do forever */

            /* Wait until the request queue contains a node */
            wait(rdptr->rd_reqsem);
            bptr = rdptr->rd_rhnext;

            /* Use operation in request to determine action */

            switch (bptr->rd_op) {

            case RD_OP_READ:

                /* Build a read request message for the server */

                msg.rd_type = htons(RD_MSG_RREQ);       /* read request */
                msg.rd_status = htons(0);
                msg.rd_seq = 0;         /* rdscomm fills in an entry    */
                idto = msg.rd_id;
                memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero */
                idfrom = rdptr->rd_id;
                while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID  */
                        ;
                }

                /* Send the message and receive a response */

                retval = rdscomm((struct rd_msg_hdr *)&msg,
                                 sizeof(struct rd_msg_rreq),
                                (struct rd_msg_hdr *)&resp,
                                 sizeof(struct rd_msg_rres),
                                rdptr );

                /* Check response */
```

```
if ( (retval == SYSERR) |  (retval == TIMEOUT) ||
                (ntohs(resp.rd_status) != 0) ) {
        panic("Failed to contact remote disk server");
}

/* Copy data from the reply into the buffer */
for (i=0; i<RD_BLKSIZ; i++) {
        bptr->rd_block[i] = resp.rd_data[i];
}

/* Unlink buffer from the request queue */

nptr = bptr->rd_next;
pptr = bptr->rd_prev;
nptr->rd_prev = bptr->rd_prev;
pptr->rd_next = bptr->rd_next;

/* Insert buffer in the cache */

pptr = (struct rdbuff *) &rdptr->rd_chnext;
nptr = pptr->rd_next;
bptr->rd_next = nptr;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;
nptr->rd_prev = bptr;

/* Initialize reference count */

bptr->rd_refcnt = 1;

/* Signal the available semaphore */

signal(rdptr->rd_availsem);

/* Send a message to waiting process */

send(bptr->rd_pid, (uint32)bptr);

/* If other processes are waiting to read the  */
/*   block, notify them and remove the request */

qptr = rdptr->rd_rhnext;
while (qptr != (struct rdbuff *)&rdptr->rd_rhnext) {
        if (qptr->rd_blknum == bptr->rd_blknum) {
                bptr->rd_refcnt++;
                send(qptr->rd_pid, (uint32)bptr);

                /* Unlink request from queue    */

                pptr = qptr->rd_prev;
                nptr = qptr->rd_next;
                pptr->rd_next = bptr->rd_next;
                nptr->rd_prev = bptr->rd_prev;

                /* Move buffer to the free list */

                qptr->rd_next = rdptr->rd_free;
                rdptr->rd_free = qptr;
```

```
                signal(&rdptr->rd_available);
                break;
            }
            qptr = qptr->rd_next;
        }
        break;

    case RD_OP_WRITE:

        /* Build a write request message for the server */

        msg.rd_type = htons(RD_MSG_WREQ);       /* write request*/
        msg.rd_blk = bptr->rd_blknum;
        msg.rd_status = htons(0);
        msg.rd_seq = 0;         /* rdscomm fills in an entry    */
        idto = msg.rd_id;
        memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero */
        idfrom = rdptr->rd_id;
        while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID  */
            ;
        }
        for (i=0; i<RD_BLKSIZ; i++) {
            msg.rd_data[i] = bptr->rd_block[i];
        }

        /* Unlink buffer from request queue */

        nptr = bptr->rd_next;
        pptr = bptr->rd_prev;
        pptr->rd_next = nptr;
        nptr->rd_prev = pptr;

        /* Insert buffer in the cache */

        pptr = (struct rdbuff *) &rdptr->rd_chnext;
        nptr = pptr->rd_next;
        bptr->rd_next = nptr;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;
        nptr->rd_prev = bptr;
        /* Declare that buffer is eligible for reuse */

        bptr->rd_refcnt = 0;
        signal(rdptr->rd_available);

        /* Send the message and receive a response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                         sizeof(struct rd_msg_wreq),
                        (struct rd_msg_hdr *)&resp,
                         sizeof(struct rd_msg_wres),
                         rdptr );

        /* Check response */

        if ( (retval == SYSERR) || (retval == TIMEOUT) ||
                        (ntohs(resp.rd_status) != 0) ) {
            panic("failed to contact remote disk server");
        }
        break;
```

```
        case RD_OP_SYNC:

            /* Send a message to the waiting process */

            send(bptr->rd_pid, OK);

            /* Unlink buffer from the request queue */

            nptr = bptr->rd_next;
            pptr = bptr->rd_prev;
            nptr->rd_prev = bptr->rd_prev;
            pptr->rd_next = bptr->rd_next;

            /* Insert buffer into the free list */

            bptr->rd_next = rdptr->rd_free;
            rdptr->rd_free = bptr;
            signal(rdptr->rd_availsem);
            break;
        }
    }
}
```

402 ~ 406

在运行代码检查的时候，要牢记远程磁盘进程比其他任何应用进程拥有更高的优先级。因此，在访问请求队列、缓存或空闲链表的时候，不需要关中断或者使用互斥信号量。但是，在使用rnscommm去与服务器交换报文的时候，rdsprocess必须保持所有的数据结构处于有效的状态，因为消息接收块在调用进程的同时也可能有其他的进程在运行。在执行读操作的时候，rdsprocess将缓冲区放在请求队列中直到请求满足。在执行写操作的时候，rdsprocess提取数据的副本，在调用rdscomm前将缓冲区移动到缓存中。

18.18　观点

理论上，远程磁盘系统只需要提供两个基本操作：读一个块和写一个块。但是，在实际应用中，同步、缓存、共享都是要考虑的问题。在本书给出的例子中，Xinu系统仅仅是作为一个客户端系统，所以尽可能地简化了设计需求。客户端可以在不与其他Xinu系统协调的情况下管理它的本地缓存。类似地，缺少共享简化了有关的同步问题：客户端只需要本地信息来强制执行最后写入语义（last-write semantics）。

如果系统扩展后允许多个Xinu系统共享一个磁盘，那么必须改变全部的设计。一个给定的客户端在与服务器进行协调之后才能进行缓存块操作。此外，最后写入语义必须横贯所有系统。这就意味着读操作需要一个集中分配机制来保证它们按照顺序发生。这里的重点是：

> 扩展远程磁盘系统以便包含多个Xinu系统共享机制，将导致系统结构的重大变化。

18.19　总结

我们认为远程磁盘系统是一个可以对磁盘块进行读和写操作，并使用网络与远程服务器进行通信和基本操作的应用程序。驱动把磁盘当做一个可以进行随机块访问的数组，不提供文件、目录或者其他的用来加速搜索的索引。读操作将磁盘中的数据块复制到内存中，写操作将内存中的数据块复制到指定的磁盘块中。

驱动代码可以分为应用程序调用的上半部函数和作为单独进程执行的下半部分函数。输入是同步的，进行读操作的进程块直到满足要求。输出是异步的，驱动接受一个输出的数据块，加入到队列中，在不阻塞进程的情况下立即返回给调用者。进程可以使用控制（control）函数刷新以前写在磁盘上的数据。 407

驱动主要使用3个数据结构：请求队列、最近使用块的缓存和空闲链表。尽管它依赖于缓存，但

是驱动程序负责确保最后写入语义。

练习

18.1 重新设计程序实现请求链表结点的缓冲区与缓存中的缓冲区分开（即为每个链表定义一个结点，为每个结点定义一个指向一个缓冲区的指针）。这样做的优点和缺点分别是什么。

18.2 重新设计远程磁盘系统，通过使用“缓冲区交换”范式来允许应用程序和驱动共享一个缓冲池。为了读磁盘块，当调用写操作时安排应用程序分配缓冲区，填充缓冲区，传递缓冲区。该操作必须返回一个缓冲区指针以使在缓冲区中的数据被提取后，应用程序可以释放申请的空间。

18.3 配置一个拥有多个远程磁盘设备的系统是可能的。修改 rdsOpen 中的代码来检查每个打开的远程磁盘设备确保每个磁盘 ID 是唯一的。

18.4 创建一个不使用缓存的远程磁盘系统，对两个版本的性能进行比较，找出不同。

18.5 高优先级进程的请求应该在低优先级进程的请求之前得到满足吗？解释其原因。

18.6 调查其他的算法，比如电梯（elevator）算法，这是一种可以用来安排磁盘请求的算法。

408 18.7 验证在到所有的挂起请求满足后才能返回对“同步”的访求。该时间延迟是否有一个边界？

文件系统

存储关心的是过去，你需要查看的事物间关系未来。

——Katharine Whitehorn

第18章讨论了磁盘设备并描述了允许系统读写磁盘块的硬件接口。尽管磁盘在存储持久、非易失数据上有优势，但是系统提供的磁盘块操作接口非常不方便使用。

本章介绍文件系统抽象。文件系统可以反映操作系统如何管理一组动态变化的文件对象，如何将文件映射到底层磁盘硬件上。

19.1 文件系统是什么

文件系统是管理持久化数据的软件，这些持久化数据的生存周期比创建并使用它们的进程的生存周期更长。持久化的数据保存在辅助存储设备上的文件中，辅助存储设备主要是磁盘设备。从概念上讲，每个文件由数据对象序列组成（例如，一个整数序列）。文件系统提供与文件相关的下述操作：创建（create）和删除（delete）文件，打开（open）指定的文件，从打开的文件中读（read）下一个对象，向打开的文件中写（write）对象，或者关闭（close）文件。如果文件系统允许随机访问，那么文件接口也给进程提供一个在文件中搜寻（seek）特定位置的操作。

许多文件系统不仅仅对辅助存储设备上的文件进行操作——它们还提供了抽象名字空间和在这个空间中对对象进行处理的高级（high-level）操作。文件名字空间由一组符合命名规则的文件名组成。名字空间可以像“由1~9个字符组成的字符串集合”那样简单，也可以像“网络、机器、用户、子目录和文件标识符按照特定的语法合理编码的字符串集合”那样复杂。在某些系统中，抽象名字空间中的名字语法表现了文件的类型信息（比如，文本文件以“.txt”结尾）。在其他的一些文件系统中，名字反映了文件系统的组织结构信息（比如，以字符串“M1 d0:”为前缀的文件名可能存在于1号机器（machine 1）的0号磁盘（disk 0）上。我们把有关文件命名的讨论放在第21章，本章中只关注文件的访问。

19.2 文件操作的示例集合

为了使文件系统尽可能小，我们在设计这个系统时使用了一种更直观的方法设计设备与文件之间的统一接口。Xinu文件的语义取自UNIX，并遵循如下的原则：

> 文件系统认为每个文件都是0个或多个字节的序列，文件上任何其他结构都是由使用文件的应用程序来解释。

把文件当做字节流有几个优点。第一，文件系统认为文件没有类型，因此不需要区分不同的文件类型。第二，因为文件系统函数集足够处理所有的文件，所以文件系统的代码很精简。第三，文件语义不仅可以适于传统文件，也可以用来解释设备和服务。第四，应用程序可以选择任意的结构来存储文件中的数据而不会影响文件系统本身。最后，文件内容与处理器或内存是独立的（比如，应用程序可能需要区分文件中的32位与64位整数，但是文件系统不需要）。

我们的系统对待文件和设备使用同样的高级操作。这样，文件系统需要支持open、close、read、write、putc、getc、seek、init和control操作。这些操作应用在传统文件上产生如下的结果：init启动时初始化与文件相关的数据结构。open打开一个命名的文件，把正在执行的进程与磁盘上的数据关联起来，并且建立一个指向首字节的指针。getc和read从文件中获取数据并移动指针。getc读取一个字节的数据，read可以读取多个字节。putc和write修改文件中的字节数据，并移动文件指针，如果写入的新数据超出了文件的末尾，文件的长度就会增加。类似地，putc修改一个字节，write修改多个字节。seek操作把指针移动到文件中指定字节的位置，文件的首字节在0字节位置。最后，close断开进程与

412 文件的关联，文件中的数据留在永久存储设备中。

19.3 本地文件系统的设计

如果文件所在的磁盘连接在计算机上，那么对这个文件对这台计算机是本地（local）的。设计一个管理本地文件的系统也不是一件容易的事，关于这个主题有大量的研究。尽管文件操作看上去很直接，但是当文件变得动态（dynamic）可变时，复杂性就表现了出来。也就是说，一个磁盘可以存储很多文件，同时一个文件的尺寸可以无限地大（直到磁盘空间用尽）。为了使文件的动态增长成为可能，文件系统不能为文件预分配（pre-allocate）磁盘块。因此，这就需要使用动态数据结构。

第二种复杂性来自并发。系统需要提供什么程度的并发文件操作？大型系统通常允许任意数量的进程并发地读、写任意数量的文件。多点访问（multiple access）的难点在于需要明确指出多进程同时读、写同一文件意味着什么。数据何时变得可读？如果两个进程试图修改文件中的同一字节，系统应该接受哪个写操作的结果？进程能不能对文件中的数据加锁以避免进程间的干扰？

小型嵌入式系统通常不需要多进程读、写文件的通用性。因此，为了限制软件复杂性和更好地利用磁盘空间，小型系统可以限制文件的访问方式。它们可以限制一个进程同时访问的文件数量，或者限制同时访问同一个文件的进程数量。

我们的目标是设计一个高效、紧凑的，可以允许进程在不引入不必要开销的前提下动态地创建文件和扩展文件的文件系统软件。作为通用性和效率的折中，我们允许进程打开任意数量的文件直到系统资源耗尽。面对一个文件，系统只允许一个打开（open）操作是活跃的（active）。也就是说，如果文件已经被某个进程打开，那么后续的打开（open）请求都会失败，直到打开这个文件的进程关闭文件。每个文件都持有一个互斥信号量（mutual exclusion semaphore）以确保一次只有一个进程可以尝试向文件中写入数据，从文件中读取数据或者修改当前文件位置。同时，目录也持有一个互斥信号量来确保每次只有进程可以试图创建文件或者修改目录项。尽管并发处理需要集中更多注意力，但我们的设计更多地关注对文件动态增长的支持：数据结构需要动态地分配磁盘空间。19.4 节将分析所需要的数据结构。

19.4 Xinu 文件系统的数据结构

为了支持动态增长和随机存取，Xinu 文件系统动态分配磁盘块并使用索引机制（index mechanism）来快速定位给定文件中的数据。Xinu 将磁盘划分为 3 个独立的区域（如图 19-1 所示）：目录（directory）、索引区（index area）和数据区（data area）。

413



图 19-1 Xinu 文件系统的磁盘划分为 3 个区域的示意图

磁盘的第一个扇区存储目录。该目录包含了一个文件名链表和指向给定文件的索引块链表的指针。目录还包含另外两个指针：一个指向未使用的索引块链表，另一个指向未使用的数据块链表。文件的目录项还包含一个表示文件当前字节大小的整数值。

索引区在磁盘上紧随目录，包含了索引块（index block）集合，简称为 i-blocks。每个文件都有自己的索引，该索引由单链表连起的索引块组成。初始时，所有的索引块都链接在一个空闲链表上，系统按需从中分配索引块。当文件缩减或删除时，将释放的索引块返回给空闲链表。

数据区占据了磁盘上剩下的所有空间。数据区中的每个块称为数据块（data block），简称为 d-blocks，因为数据块中包含了存储在文件中的数据。当数据块中没有指向其他数据块的指针时，它也不包含把数据块关联到文件某一部分的信息，所有这些信息都保存在文件的索引中。

与索引块类似，当磁盘初始化时，数据块被组织成一个空闲链表。文件系统按需从空闲链表中分配数据块。当文件缩减或删除时，将无用的数据块返回给空闲链表。

图 19-2 是 Xinu 文件系统的数据结构示意图。该图与实际比例不符：实际上，数据块比索引块大得多，它占据了一个完整的磁盘块。

414

需要注意的一点是图 19-2 中的数据结构存储在磁盘上。对于指定的某一时刻，只有一部分结构存

放在内存中——文件系统必须在不将结构读入内存的前提下创建和维护索引。

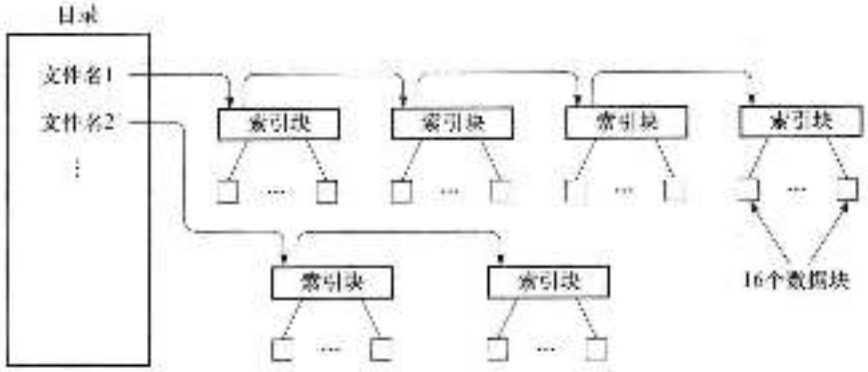


图 19-2 Xinu 文件系统示意图，图中每个文件由包含数据块指针的索引块链表组成

19.5 索引管理器的实现

从概念上，索引块构成了一个映射到磁盘连续区域的随机访问数组，即索引块从 0～K 编号，软件利用索引号来引用给定的索引块。因为索引块比物理磁盘块小，所以系统在每个物理块中存储了 7 个索引块，读、写索引块的细节由软件来处理。

因为底层硬件每次传输一个整磁盘块，所以文件系统不可能只传输一个索引块而不传输与该块处在同一磁盘块上的其他索引块。因此，为了写索引块，软件必须读出该索引块所在的整个磁盘块，将新的索引块复制到正确位置，再将结果物理块写回磁盘。类似地，为了读一个索引块，软件需要读整个物理磁盘块，然后从中提取索引块。

在分析处理索引块的代码之前，我们需要理解一些基本概念。文件 lfilesys.h 定义整个本地文件系统使用的常量和数据结构，包括索引块内容的 lfblk 结构。 415

```
/* lfilesys.h - ib2eoct, ib2disp */

/*------------------------------------------------------------------------*/
/*                                                                        */
/*              Local File System Data Structures                         */
/*                                                                        */
/*   A local file system uses a random-access disk composed of 512-byte   */
/* sectors numbered 0 through N-1.  We assume disk hardware can read or   */
/* write any sector at random, but must transfer an entire sector.        */
/* Thus, to write a few bytes, the file system must read the sector,      */
/* replace the bytes, and then write the sector back to disk.  Xinu's     */
/* local file system divides the disk as follows: sector 0 is a           */
/* directory, the next K sectors constitute an index area, and the        */
/* remaining sectors comprise a data area.  The data area is easiest to   */
/* understand:  each sector holds one data block (d-block) that stores    */
/* contents from one of the files (or is on a free list of unused data    */
/* blocks).  We think of the index area as holding an array of index      */
/* blocks (i-blocks) numbered 0 through I-1.  A given sector in the       */
/* index area holds 7 of the index blocks, which are each 72 bytes        */
/* long.  Given an i-block number, the file system must calculate the     */
/* disk sector in which the i-block is located and the byte offset        */
/* within the sector at which the i-block resides.  Internally, a file    */
/* is known by the i-block index of the first i-block for the file.       */
/* The directory contains a list of file names and the  i-block number    */
/* of the first i-block for the file.  The directory also holds the       */
/* i-block number for a list of free i-blocks and a data block number     */
/* of the first data block on a list of free data blocks.                 */
/*                                                                        */
/*------------------------------------------------------------------------*/
```

```
#ifndef NLFL
#define NLFL    1
#endif

/* Use the remote disk device if no disk is defined (file system  */
/*  *assumes* the underlying disk has a block size of 512 bytes)  */

#ifndef LF_DISK_DEV
#define LF_DISK_DEV     SYSERR
#endif

#define LF_MODE_R       F_MODE_R        /* mode bit for "read"          */
#define LF_MODE_W       F_MODE_W        /* mode bit for "write"         */
#define LF_MODE_RW      F_MODE_RW       /* mode bits for "read or write"*/
#define LF_MODE_O       F_MODE_O        /* mode bit for "old"           */
#define LF_MODE_N       F_MODE_N        /* mode bit for "new"           */

#define LF_BLKSIZ       512             /* assume 512 byte disk blocks */
#define LF_NAME_LEN     16              /* length of name plus null     */
#define LF_NUM_DIR_ENT  20              /* num. of files in a directory */

#define LF_FREE         0               /* slave device is available    */
#define LF_USED         1               /* slave device is in use       */

#define LF_INULL        (ibid32) -1     /* index block null pointer     */
#define LF_DNULL        (dbid32) -1     /* data block null pointer      */
#define LF_IBLEN        16              /* data block ptrs per i-block  */
#define LF_IDATA        8192            /* bytes of data indexed by a   */
                                        /*  single index block          */
#define LF_IBASE        0xFFFFE000      /* mask for the data indexed by */
                                        /*  a single index block (i.e., */
                                        /*  bytes 0 through 8191).      */
#define LF_IMASK        0x00001FFF      /* mask for the data in a data  */
                                        /*  block  (0 through 511)      */

#define LF_AREA_IB      1               /* first sector of i-blocks     */
#define LF_AREA_DIR     3               /* first sector of directory    */

/* Structure of an index block on disk */

struct  lfibik          {               /* format of index block        */
        lbid32          ib_next;        /* address of next index block  */
        uint32          ib_offset;      /* first data byte of the file  */
                                        /*  indexed by this i-block     */
        dbid32          ib_dba[LF_IBLEN];/* ptrs to data blocks indexed */
};

/* Conversion functions below assume 7 index blocks per disk block */

/* Conversion between index block number and disk sector number */

#define ib2sect(ib)     (((ib)/7)+LF_AREA_IB)

/* Conversion between index block number and the relative offset within */
/*      a disk sector                                                   */

#define ib2ioff(ib)     (((ib)%7)*sizeof(struct lfibik))

/* Structure used in each directory entry for the local file system */
```

```
struct  ldentry {                       /* description of entry for one */
                                        /*  file in the directory       */
        uint32  ld_size;                /* curr. size of file in bytes  */
        ibid32  ld_ilist;               /* ID of first i-block for file */
                                        /*   or IB_NULL for empty file  */
        char    ld_name[LF_NAME_LEN];   /* null-terminated file name    */
};

/* Structure of a data block when on the free list on disk */

struct  lfdbfree {
        dbid32  lf_nextdb;              /* next data block on the list  */
        char    lf_unused[LF_BLKSIZ - sizeof(dbid32)];
};

/* Format of the file system directory, either on disk or in memory */

#pragma pack(2)
struct  lfdir   {                       /* entire directory on disk     */
        dbid32  lfd_dfree;              /* list of free d-blocks on disk*/
        ibid32  lfd_ifree;              /* list of free i-blocks on disk*/
        int32   lfd_nfiles;             /* current number of files      */
        struct  ldentry lfd_files[LF_NUM_DIR_ENT]; /* set of files      */
        char    lf_pad[20];             /* unused chars in directory blk*/
};
#pragma pack()

/* Global data used by local file system */

struct  lfdata  {                       /* local file system data       */
        did32   lf_dskdev;              /* device ID of disk to use     */
        sid32   lf_mutex;               /* mutex for the directory and  */
                                        /*  index/data free lists       */
        struct  lfdir   lf_dir;         /* In-memory copy of directory  */
        bool8   lf_dirpresent;          /* True when directory is in    */
                                        /*  memory (first file is open) */
        bool8   lf_dirdirty;            /* Has the directory changed?   */
};

/* Control block for local file pseudo-device */

struct  lflcblk {                       /* Local file control block     */
                                        /*  (one for each open file)    */
        byte    lfstate;                /* Is entry free or used        */
        did32   lfdev;                  /* device ID of this device     */
        sid32   lfmutex;                /* Mutex for this file          */
        struct  ldentry *lfdirptr;      /* Ptr to file's entry in the   */
                                        /*  in memory directory         */
        int32   lfmode;                 /* mode (read/write/both)       */
        uint32  lfpos;                  /* Byte position of next byte   */
                                        /*   to read or write           */
        char    lfname[LF_NAME_LEN];    /* Name of the file             */
        ibid32  lfinum;                 /* ID of current index block in */
                                        /*   lfiblock or LF_INULL       */
        struct  lfiblk  lfiblock;       /* In-mem copy of current index */
                                        /*   block                      */
        dbid32  lfdnum;                 /* Number of current data block */
                                        /*   in lfdblock or LF_DNULL    */
        char    lfdblock[LF_BLKSIZ];    /* In-mem copy of current data  */
```

```
                                        /*   block                          */
        char    *lfbyte;                /* Ptr to byte in lfdblock or      */
                                        /*  address one beyond lfdblock     */
                                        /*   if current file pos lies       */
                                        /*   outside lfdblock               */
        bool8   lfibdirty;              /* Has lfiblock changed?            */
        bool8   lfdbdirty;              /* Has lfdblock changed?            */
};

extern  struct  lfdata  LF_data;
extern  struct  lfiblk lfltab[];

/* Control functions */

#define LF_CTL_DEL      F_CTL_DEL       /* Delete a file                    */
#define LF_CTL_TRUNC    F_CTL_TRUNC     /* Truncate a file                  */
#define LF_CTL_SIZE     F_CTL_SIZE      /* Obtain the size of a file        */
```

如上所示，每个索引块包含指向下一索引块的指针。偏移量（offset）标识文件中被索引的最低位置，同时索引块包含16个指向数据块的指针的数组。也就是说，数组中每个项给出了数据块在物理磁盘上的扇区号。因为一个扇区512字节长，所以一个索引块可索引16个512字节的数据块，共8192字节的数据。

给出一个索引块的地址，软件如何知道到哪里寻找索引块？索引块是连续的，占用了从LF_AREA_IB扇区开始的连续磁盘扇区。在我们的设计中，目录占用磁盘块0，意味着索引区从扇区1开始。因此，索引块0～7存储在扇区1，8～15存储在扇区2，以此类推。内联函数ib2sect把一个索引块号转换为正确的扇区号，内联函数ib2disp把一个索引块号转换为到物理磁盘块上的字节偏移。两个函数都可以在上述lfilesys.h文件中找到。

19.6 清空索引块(lfibclear)

当从空闲块中分配一个索引块时，文件系统必须把索引块读入内存并且清空索引块以删除旧数据。尤其是，所有的数据块指针必须设置为空值（null value），从而保证它们不会与合法指针混淆。而且，索引块中的偏移量必须赋予合适的文件偏移量。函数lfibclear清空一个索引块，该段代码在文件lfibclear.c中。

```
/* lfibclear.c - lfibclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibclear  --  clear an in core copy of an index block
 *------------------------------------------------------------------------
 */
void    lfibclear(
          struct lfiblk *ibptr,         /* address of i-block in memory */
          int32         offset          /* file offset for this i-block */
        )
{
        int32   i;                      /* indexes through array        */

        ibptr->ib_offset = offset;      /* assign specified file offset */
        for (i=0 ; i<LF_IBLEN ; i++) {  /* clear each data block pointer*/
                ibptr->ib_dba[i] = LF_DNULL;
        }
        ibptr->ib_next = LF_INULL;      /* set next ptr to null         */
}
```

19.7 获取索引块(lfibget)

为了把一个索引块读入内存，系统必须把索引块号映射到物理磁盘块地址，读取物理磁盘块，从物理块中的合适区域复制到指定内存。文件 lfibget.c 包含了实现代码，它使用内联函数 ib2sect 将索引块号转换为磁盘扇区号，函数 ib2disp 计算索引块在磁盘扇区上的位置。 [420]

```
/* lfibget.c - lfibget */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibget  --  get an index block from disk given its number (assumes
 *                   mutex is held)
 *------------------------------------------------------------------------
 */
void    lfibget(
          did32         diskdev,        /* device ID of disk to use     */
          ibid32        inum,           /* ID of index block to fetch   */
          struct lfiblk *ibuff          /* buffer to hold index block   */
        )
{
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* dbuff to hold disk block     */

        /* Read disk block that contains the specified index block */

        read(diskdev, dbuff, ib2sect(inum));

        /* Copy specified index block to caller's ibuff */

        from = dbuff + ib2disp(inum);
        to = (char *)ibuff;
        for (i=0 ; i<sizeof(struct lfiblk) ; i++)
                *to++ = *from++;
        return;
}
```

19.8 存储索引块(lfibput)

与获取索引块相比，存储索引块的操作更加复杂，因为要存储一个索引块，必须首先读取相应的磁盘扇区，将要存储的索引块复制到该磁盘扇区中的合适位置，然后将此磁盘扇区写回到磁盘中。该部分的实现代码在文件 lfibput.c 中，其中包含了一个与 lfibget 函数相类似的 lfibput 函数。 [421]

```
/* lfibput.c - lfibput */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibput  --  write an index block to disk given its ID (assumes
 *                   mutex is held)
 *------------------------------------------------------------------------
 */
status  lfibput(
          did32         diskdev,        /* ID of disk device            */
          ibid32        inum,           /* ID of index block to write   */
          struct lfiblk *ibuff          /* buffer holding the index blk */
        )
```

```
{
        dbid32  diskblock;              /* ID of disk sector (block)    */
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* temp. buffer to hold d-block */

        /* Compute disk block number and offset of index block */

        diskblock = ib2sect(inum);
        to = dbuff + ib2disp(inum);
        from = (char *)ibuff;

        /* Read disk block */

        if (read(diskdev, dbuff, diskblock) == SYSERR) {
                return SYSERR;
        }

        /* Copy index block into place */

        for (i=0 ; i<sizeof(struct lfiblk) ; i++) {
                *to++ = *from++;
        }

        /* Write the block back to disk */

        write(diskdev, dbuff, diskblock);
        return OK;
}
```

422

19.9 从空闲链表中分配索引块（lfiballoc）

当文件需要索引时，文件系统从空闲链表中为此文件分配一个索引块。函数lfiballoc负责获取下一个空闲的索引块并返回它的ID。该函数在文件lfiballoc.c中，假设文件系统目录的副本已经读入到内存中，通过Lf_data.lf_dir全局变量来表示该副本。

```
/* lfiballoc.c  -  lfiballoc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfiballoc  -  allocate a new index block from free list on disk
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
ibid32  lfiballoc(void)
{
        ibid32  ibnum;          /* ID of next block on the free list    */
        struct  lfiblk  iblock; /* buffer to hold index block           */

        /* Get ID of first index block on free list */

        ibnum = Lf_data.lf_dir.lfd_ifree;
        if (ibnum == LF_INULL) {        /* ran out of free index blocks */
                panic("out of index blocks");
        }
        lfibget(Lf_data.lf_dskdev, ibnum, &iblock);

        /* Unlink index block from the directory free list */
```

```
        Lf_data.lf_dir.lfd_ifree = iblock.ib_next;

        /* Write a copy of the directory to disk after the change */

        write(Lf_data.lf_dskdev, (char *) &Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        return ibnum;
}
```

423

19.10 从空闲链表中分配数据块（lfdballoc）

因为索引块包含了一个名为“next”的指针字段，所以将索引块加入空闲链表的操作比较简单。但是，因为数据块通常无法包含指针字段，所以对于数据块而言，空闲链表通常并不是显式存在的。Xinu系统在设计时使用了单向的空闲链表，这意味着仅需要使用一个指针。如果数据块位于空闲链表中，那么系统使用数据块的前4个字节存储指向空闲链表中下一个数据块的指针。文件lflesys.h中定义了一个名为lfdbfree的结构，它描述空闲链表中数据块的具体格式。当需要从空闲链表中获取数据块的时候，文件系统就使用该结构体的定义。而一旦将数据块分配到文件中并从空闲链表中删除，该数据块就又可以被看做是一个字节数组。

函数lfdballoc负责从空闲链表中分配数据块并返回该数据块号，从中我们可以看到系统是如何使用结构lfdbfree的，该代码在文件lfdballoc.c中。

```
/* lfdballoc.c - lfdballoc */

#include <xinu.h>

#define  DFILL  '+'                     /* char. to fill a disk block   */

/*------------------------------------------------------------------------
 * lfdballoc  -  allocate a new data block from free list on disk
 *                   (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
dbid32  lfdballoc (
          struct lfdbfree *dbuff /* addr. of buffer to hold data block  */
        )
{
        dbid32  dnum;           /* ID of next d-block on the free list  */
        int32   retval;         /* return value                         */

        /* Get the ID of first data block on the free list */

        dnum = Lf_data.lf_dir.lfd_dfree;
        if (dnum == LF_DNULL) { /* ran out of free data blocks */
                panic("out of data blocks");
        }
        retval = readlf(Lf_data.lf_dskdev, (char *)dbuff, dnum);
        if (retval == SYSERR) {
                panic("lfdballoc cannot read disk block\n\r");
        }
        /* Unlink d-block from in-memory directory */

        Lf_data.lf_dir.lfd_dfree = dbuff->lf_nextdb;
        write(Lf_data.lf_dskdev, (char *)&Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        /* Fill data block to erase old data */

        memset((char *)dbuff, DFILL, LF_BLKSIZ);
```

```
        return dnum;
}
```

另一个相关的函数是 lfdbfree.c 文件中的 lfdbfree，该函数的作用是向空闲链表添加一个数据块。

```
/* lfdbfree.c - lfdbfree */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfdbfree  --  free a data block given its block number (assumes
 *                    directory mutex is held)
 *------------------------------------------------------------------------
 */
status  lfdbfree(
          did32         diskdev,        /* ID of disk device to use     */
          dbid32        dnum            /* ID of data block to free     */
        )
{
        struct  lfdir   *dirptr;        /* pointer to directory         */
        struct  lfdbfree buf;           /* buffer to hold data block    */

        dirptr = &bf_data.lf_dir;
        buf.lf_nextdb = dirptr->lfd_dfree;
        dirptr->lfd_dfree = dnum;
        write(diskdev, (char *)&buf,   dnum);
        write(diskdev, (char *)dirptr, LF_ARRA_DIR);

        return OK;
}
```

424
425

为了将数据块放入空闲链表中，函数 lfdbfree 首先会将该数据块指向当前的空闲链表，然后将当前空闲链表指向该数据块。由于插入了一个指针，所以该数据块必须写入磁盘中。又由于目录的空闲链表被修改了，所以该目录的副本也必须写入到磁盘中。

19.11 使用设备无关的 I/O 函数的文件操作

文件系统必须在执行进程与磁盘文件之间建立联系，这样才能保证相关的文件操作（例如，read 和 write）能够映射到正确的文件上。而文件系统如何正确实现这种映射操作依赖于文件系统大小和实际的需要。为了保证文件系统尽可能地小，我们使用系统中已经存在的设备转换机制（device switch mechanism），而不是引入新的函数。

假设设备转换表（device switch table）中已经包含了许多文件伪设备（pseudo-device），并且每个伪设备都能用来控制一个打开的文件。与常规的设备相类似，伪设备也有一组驱动函数，例如 read、write、gete、pute、seek 和 close 操作。当进程需要打开一个磁盘文件时，文件系统寻找一个当前未使用的伪设备，设置此伪设备的控制块（control block），然后给调用者返回此伪设备的 ID。文件打开之后，进程使用该伪设备 ID 来执行 gete、read、pute、write 和 seek 操作。我们知道设备转换机制将高级操作映射到物理设备相应的驱动函数，与此类似，设备转换机制也会将高级操作映射到文件伪设备相应的驱动函数。最后，在文件使用结束后，进程会调用 close 来断开连接，这样该伪设备就可以被其他文件使用。稍后，我们通过代码来说明具体的细节。

设计一个伪设备的驱动与设计一个常规硬件设备的驱动基本相同。与其他设备类似，伪设备的驱动为每一个伪设备创建一个控制块。文件伪设备控制块定义可在文件 lfilesys.h 的结构 lfdblk 中。从概念上，该控制块包含了两类字段：存储伪设备信息的字段和存储来自磁盘信息的字段。字段 lfstate 和 lfmode 是前一种类型；lfstate 字段表示该设备是否正在被使用，lfmode 字段表示该文件是否因为需要读、写操作而已经被打开。字段 lfiblock 和 lfdblock 用于存储来自磁盘的信息。字段 lfiblock 和 lfdblock 是后一种类型：当一个文件正被读、写时，需要包含一份索引块的副本和数据块的副本，同时还应该

包含一个表示文件当前位置（以字节表示）的副本（使用lfpos字段来表示）。

当文件被打开时，位置（控制块中的lfpos字段）被赋予值0。该位置会随着进程读数据或写数据而不断增加。通过调用seek函数，进程可以将位置移动到文件中任意的地方，并同时会更新lfpos的值。

19.12 文件系统的设备设置和函数名称

打开一个文件并为之分配一个伪设备进行读、写操作，我们需要使用什么样的接口呢？因为Xinu系统需要将所有的函数都映射到设备空间中，所以本地文件系统会定义一个名为LFILESYS的本地文件主设备（master local file device）。调用LFILESYS设备的open函数就意味着系统会分配一个伪设备，然后返回此伪设备的ID。文件伪设备以LFILE0、LFILE1、…、来命名，但需要注意的是，这些名字只会在配置文件中使用。图19-3显示了主设备和伪设备是如何进行设置的。 426

```
/* 本地文件系统主设备类型 */

lfs: on disk
                -i lfsInit      -o lfsOpen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n riaControl
                -intr NULL

/* 本地文件伪设备类型 */

lfl: on lfs
                -i lflInit      -o ioerr        -c lflClose
                -r lflRead      -g lflGetc      -p lflPutc
                -w lflWrite     -c lflSeek      -n ioerr
                -intr NULL
```

图19-3 对本地文件系统主设备类型的设置和对本地文件伪设备类型的设置

如图19-3所示，文件系统主设备驱动函数的名字都以lfs开头，而文件伪设备驱动函数的名字都以lfl开头。之后我们还将会看到，这两种类型设备的驱动函数所使用的辅助函数的名字都以lf开头。

19.13 本地文件系统打开函数（lfsOpen）

图19-4显示了对本地文件主设备的设置以及对多个本地文件伪设备的设置。因为同打开一个文件都要使用一个伪设备，所以本地文件伪设备的总量限定了可以同时打开的文件的个数。 427

```
/* 本地文件系统主设备（系统中只有一个） */
        LFILESYS is lfs on disk
/* 本地文件伪设备（系统中有多个） */
        LFILE0 is lfl on lfs
        LFILE1 is lfl on lfs
        LFILE2 is lfl on lfs
        LFILE3 is lfl on lfs
        LFILE4 is lfl on lfs
        LFILE5 is lfl on lfs
        LFILE6 is lfl on lfs
```

图19-4 对本地文件系统主设备的设置和多个本地文件伪设备的设置

以上设置表明进程可以使用主设备来打开本地文件，然后使用伪设备存取文件。例如，打开一个名为myfile的文件进行读、写操作，有如下的代码：

```
fd = open(LFILESYS, "myfile", "rw");
```

假设成功打开了该文件，那么就可以使用描述符fd向文件中写数据，如下所示：

```
char buffer[1500];
... code to fill buffer ...
fd = write(fd, buffer, 1500);
```

设备LFILESYS只用于打开文件。因此，文件系统主设备的驱动只需要实现open和init，所有其他的I/O操作都映射为ioerr。函数lfsOpen实现open操作，该函数在文件lfsOpen.c中：

```
/* lfsOpen.c  -  lfsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsOpen - open a file and allocate a local file pseudo-device
 *------------------------------------------------------------------------
 */
```

```
devcall lfsOpen (
        struct dentry  *devptr,         /* entry in device switch table */
        char   *name,                   /* name of file to open         */
        char   *mode                    /* mode chars: 'r' 'w' 'o' 'n'  */
        )
{
        struct  lfdir   *dirptr;        /* ptr to in-memory directory   */
        char            *from, *to;     /* ptrs used during copy        */
        char            *nam, *cmp;     /* ptrs used during comparison  */
        int32           i;              /* general loop index           */
        did32           lfnext;         /* minor number of an unused    */
                                        /*    file pseudo-device        */
        struct  ldentry *lfptr;         /* ptr to an entry in directory */
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
        bool8           found;          /* was the name found?          */
        int32   retval;                 /* value returned from function */
        int32   mbits;                  /* mode bits                    */

        /* Check length of name file (leaving space for NULLCH */

        from = name;
        for (i=0; i< LF_NAME_LEN; i++) {
                if (*from++ == NULLCH) {
                        break;
                }
        }
        if (i >= LF_NAME_LEN) {         /* name is too long */
                return SYSERR;
        }

        /* Parse mode argument and convert to binary */

        mbits = lfgetmode(mode);
        if (mbits == SYSERR) {
                return SYSERR;
        }

        /* If named file is already open, return SYSERR */

        lfnext = SYSERR;
        for (i=0; i<NLFL; i++) {        /* search file pseudo-devices   */
                lfptr = &lflitab[i];
                if (lfptr->lfstate == LF_FREE) {
                        if (lfnext == SYSERR) {
                                lfnext = i; /* record index */
                        }
                        continue;
                }

                /* Compare requested name to name of open file */
                nam = name;
                cmp = lfptr->lfname;
                while(*nam != NULLCH) {
                        if (*nam != *cmp) {
                                break;
                        }
                        nam++;
                        cmp++;
                }
```

```
                    /* See if comparison succeeded */

                    if ( (*nam==NULLCH) && (*cmp == NULLCH) ) {
                            return SYSERR;
                    }
            }
            if (lfnext == SYSERR) { /* no slave file devices are available  */
                    return SYSERR;
            }

            /* Obtain copy of directory if not already present in memory    */

            dirptr = &lf_data.lf_dir;
            wait(lf_data.lf_mutex);
            if !! Lf_data.lf_dirpresent) {
                retval = read(lf_data.lf_dskdev,(char *)dirptr,LF_AREA_DIR);
                if (retval == SYSERR ) {
                    signal(lf_data.lf_mutex);
                    return SYSERR;
                }
                lf_data.lf_dirpresent = TRUE;
            }

            /* Search directory to see if file exists */

            found = FALSE;
            for (i=0; i<dirptr->lfd_nfiles; i++) {
                    lfptr = &dirptr->lfd_files[i];
                    nam = name;
                    cmp = lfptr->lfd_name;
                    while(*nam != NULLCH) {
                            if (*nam != *cmp) {
                                    break;
                            }
                    nam++;
                    cmp++;
            }
            if ( (*nam==NULLCH) && (*cmp==NULLCH) ) { /* name found */
                    found = TRUE;
                    break;
            }
    }
    /* Case #1 - file is not in directory (i.e., does not exist)    */

    if (! found) {
            if (mbits & LF_MODE_O) {        /* file 'must' exist    */
                    signal(lf_data.lf_mutex);
                    return SYSERR;
            }

            /* Take steps to create new file and add to directory   */

            /* Verify that space remains in the directory */

            if (dirptr->lfd_nfiles >= LF_NUM_DIR_ENT) {
                    signal(lf_data.lf_mutex);
                    return SYSERR;
            }

            /* Allocate next dir. entry & initialize to empty file  */
```

```
                ldptr = &dirptr->lfd_files[dirptr->lfd_nfiles++];
                ldptr->ld_size = 0;
                from = name;
                to = ldptr->ld_name;
                while ( (*to++ = *from++) != NULLCH ) {
                        ;
                }
                ldptr->ld_ilist = LF_INULL;

        /* Case #2 - file is in directory (i.e., already exists)        */

        } else if (mbits & LF_MODE_N) {         /* file must not exist  */
                        signal(&LF_data.lf_mutex);
                        return SYSERR;
        }

        /* Initialize the local file pseudo-device */
        lfptr = &lflltab[lfnext];
        lfptr->lfstate = LF_USED;
        lfptr->lfdirptr = ldptr;        /* point to directory entry     */
        lfptr->lfmode = mbits & LF_MODE_RW;

        /* File starts at position 0 */

        lfptr->lfpos    = 0;

        to = lfptr->lfname;
        from = name;
        while ( (*to = *from++) != NULLCH ) {
                ;
        }

        /* Neither index block nor data block are initially valid       */

        lfptr->lfinum   = LF_INULL;
        lfptr->lfdnum   = LF_DNULL;

        /* Initialize byte pointer to address beyond the end of the     */
        /*      buffer (i.e., invalid pointer triggers setup)           */

        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];
        lfptr->lfibdirty = FALSE;
        lfptr->lfdbdirty = FALSE;

        signal(LF_data.lf_mutex);

        return lfptr->lfdev;
}
```

在验证了文件名字的长度有效后，lfsOpen 函数调用 lfgetmode 函数来解析模式（mode）参数并将其转换为一组二进制位。模式参数由字符号结尾的字符串组成，其中包含了 0 个或多个如图 19-5 所示的字符。

428 | 432

字符	含义
r	打开文件进行读操作
w	打开文件进行写操作
o	文件必须为“old”（即必须存在）
n	文件必须为“new”（即必须不存在）

图 19-5 模式参数中允许使用的字符串及其含义

模式参数字符串中的字符不能重复，同时使用字符“o”和“n”也是不合法的。另外，如果既没有使用字符“r”也没有使用字符“w”，lfgetmode 将默认同时允许读操作和写操作。该部分代码在文件 lfgetmode.c 中。

```
/* lfgetmode.c  -  lfgetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfgetmode  -  parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */
int32	lfgetmode (
	  char	*mode			/* string of mode characters	*/
	)
{
	int32	mbits;			/* mode bits to return		*/
	char	ch;			/* next char in mode string	*/

	mbits = 0;
	while ( (ch = *mode++) != NULLCH) {
		switch (ch) {

		    case 'r':	if (mbits&LF_MODE_R) {
					return SYSERR;
				}
				mbits |= LF_MODE_R;
				continue;

		    case 'w':	if (mbits&LF_MODE_W) {
					return SYSERR;
				}
				mbits |= LF_MODE_W;
				continue;

		    case 'o':	if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
					return SYSERR;
				}
				mbits |= LF_MODE_O;
				break;

		    case 'n':	if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
					return SYSERR;
				}
				mbits |= LF_MODE_N;
				break;

		    default:	return SYSERR;
		}
	}

	/* If neither read nor write specified, allow both */

	if ( (mbits&LF_MODE_RW) == 0 ) {
		mbits |= LF_MODE_RW;
	}
	return mbits;
}
```

模式参数解析完成后，lfsOpen 进行如下检验：检验文件是否还未打开，检验是否可以获取到一个文件伪设备，同时检验文件系统目录中该文件是否已经存在。如果该文件已经存在（同时模式参数允许打开已存在的文件），lfsOpen 将设置文件伪设备的控制块。如果该文件不存在（同时模式参数允许创建新文件），lfsOpen 在文件系统目录中为之分配一个条目，然后设置文件伪设备的控制块。最初的文件位置设置为 0。控制块中的 lfinum 字段和 lfdnum 字段设置为 null 以表示索引块和数据块当前还未被使用。重要的是，将 lfbyte 字段设置为一个超出数据块缓冲区末端的值。我们会发现，设置 lfbyte

是非常重要的，因为代码在获取数据时将会使用到这个值。

433
-
434

当 lfbyte 在 ldblock 中包含地址时，它指向的字节表示了 lfpos 给定位置上文件中的数据。当 lfbyte 包含着对应于 ldblock 的地址时，ldblock 中的值不能使用。

以上过程会数据传输函数中有具体说明，例如，lfClose 函数和 lfPute 函数。

19.14 关闭文件伪设备（lfiClose）

当应用程序结束使用文件时，它调用 close 来终止文件的使用，并且使这个文件伪设备对其他用户可用。理论上，关闭一个伪设备很简单：只要改变设备的状态，表明它没有被使用。但是，实际上，因为控制块可能包含没有被写入文件的数据，所以缓冲使关闭操作变得复杂。因此，函数 lfClose 必须检查控制块的各位来确定索引块或数据块在被写入磁盘之后它们的内容是否已经改变了。如果发生了改变，lfClose 需要在改变控制块状态之前，调用函数 lfflush 将改变写入磁盘。文件 lfClose.c 包含以下代码：

```
/* lfiClose.c      lfiClose.c */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfiClose  --  close a file by flushing output and freeing device entry
 *------------------------------------------------------------------------
 */
devcall lfiClose (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  lflblk *lfptr;          /* ptr to open file table entry */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Write inode or data blocks to disk if they have changed */
        if (lfptr->lfdbdirty || lfptr->lfibdirty) {
                lfflush(lfptr);
        }

        /* Set device state to FREE and return to caller */

        lfptr->lfstate = LF_FREE;
        signal (lfptr->lfmutex);
        return OK;
}
```

19.15 刷新磁盘中的数据（lfflush）

函数 lfflush 会像预期的那样操作。它接收一个指向伪设备控制块的指针作为参数，并使用这个指针来检查控制块中的“脏”位[⊖]。如果索引块改变了，lfflush 使用 lfbput 函数将副本写入磁盘；如果数据块改变了，lfflush 使用 write 函数将副本写入磁盘。字段 lfinum 和 lfdnum 包含使用的索引块号和数据块号。这段代码包含在 lfflush.c 文件中。

⊖ 术语脏位（dirty bit）：有一个被设置为 TRUE 的布尔值（即1个位），表明最近被修改过。

```
/* lfflush.c  -  lfflush */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfflush  -  flush data block and index blocks for an open file
 *                      (assumes file mutex is held)
 *------------------------------------------------------------------------
 */
status  lfflush (
          struct lflcblk  *lfptr        /* ptr to file pseudo device    */
        )
{
        if (lfptr->lfstate == LF_FREE) {
                return SYSERR;
        }
        /* Write data block if it has changed */

        if (lfptr->lfdbdirty) {
                write(Lf_data.lf_dskdev, lfptr->lfdblock, lfptr->lfdbnum);
                lfptr->lfdbdirty = FALSE;
        }
        /* Write i-block if it has changed */

        if (lfptr->lfibdirty) {
                lfibput(Lf_data.lf_dskdev, lfptr->lfinum, &lfptr->lfiblock);
                lfptr->lfibdirty = FALSE;
        }

        return OK;
}
```

19.16 文件的批量传输函数（lflWrite，lflRead）

Xinu采用直接方法来写和读取文件：一个循环反复使用适应的字符传输函数。例如，实现了写操作的函数lflWrite会重复调用lflPutc。文件lflWrite.c包含这些代码：

```
/* lflWrite.c  -  lfWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflWrite  --  write data to a previously opened local disk file
 *------------------------------------------------------------------------
 */
devcall lflWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer holding data to write */
          int32 count                   /* number of bytes to write     */
        )
{
        int32           i;              /* number of bytes written      */

        if (count < 0) {
                return SYSERR;
        }
        for (i=0; i<count; i++) {
                if (lflPutc(devptr, *buff++) == SYSERR) {
                        return SYSERR;
                }
        }
        return count;
}
```

函数lflRead实现了读操作。为了满足一个请求，lflRead重复调用lflGetc，每次调用接收一个字

节，并将这个字节放在调用者缓冲区的下一个位置。当它到达文件的末尾（end-of-file）时，lflGetc返回常量EOF。如果当lflRead收到lflGetc传来的EOF时，lflRead已经读取了一个或多个字节的数据，那么lflRead就停止循环，返回已经读取的字节数。如果当到达文件末尾时，没有发现数据，那么lflRead给调用者返回EOF。文件lflRead.c包含这些代码。

```
/* lflRead.c - lflRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflRead  --  read from a previously opened local file
 *------------------------------------------------------------------------
 */
devcall lflRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer to hold bytes         */
          int32 count                   /* max bytes to read            */
        )
{
        uint32  numread;                /* number of bytes read         */
        int32   nxtbyte;                /* character or SYSERR/EOF      */

        if (count < 0) {
                return SYSERR;
        }

        for (numread=0 ; numread < count ; numread++) {
                nxtbyte = lflGetc(devptr);
                if (nxtbyte == SYSERR) {
                        return SYSERR;
                } else if (nxtbyte == EOF) {    /* EOF before finished */
                    if (numread == 0) {
                        return EOF;
                    } else {
                        return numread;
                    }
                } else {
                        *buff++ = (char) (0xff & nxtbyte);
                }
        }
        return numread;
}
```

19.17 在文件中查找一个新位置（lflSeek）

进程可以调用seek来改变文件中下一个读取的位置。我们的系统使用函数lflSeek来实现seek，并且限制其只能访问文件的有效位置（它与UNIX系统不同，UNIX系统允许应用程序寻找超过文件结尾的位置）。

查找一个新的位置包括改变文件控制块中的lfpos字段，将字段lfbyte设置为一个超过lfdblock的地址（根据上面的不变量，这意味着首到索引块和数据块在位置上时，指针才能被用来提取数据）。文件lflSeek.c包含这些代码。

```
/* lflSeek.c - lflSeek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfseek  -  seek to a specified position in a file
 *------------------------------------------------------------------------
 */
devcall lflSeek (
          struct dentry *devptr,        /* entry in device switch table */
          uint32        offset          /* byte position in the file    */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
```

```
        /* If file is not open, return an error */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);
        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Verify offset is within current file size */

        if (offset > lfptr->lfdirptr->ld_size) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }
        /* Record new offset and invalidate byte pointer (i.e., */
        /*   force the index and data blocks to be replaced if  */
        /*   an attempt is made to read or write)               */

        lfptr->lfpos = offset;
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

        signal(lfptr->lfmutex);
        return OK;
}
```

19.18 从文件中提取一个字节（lflGetc）

一旦一个文件打开，索引块和数据块正确地载入内存，从文件中读取一个字节就很简单：它包括将 lfbyte 作为指向字节的指针，提取字节，并将指针移到下一个字节。函数 lflGetc 完成这个操作，相关代码在文件 lflGetc.c 中：

```
/* lflGetc.c - lflGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflGetc  --  Read the next byte from an open local file
 *------------------------------------------------------------------------
 */
devcall lflGetc (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  lflblk *lfptr;          /* ptr to open file table entry */
        struct  ldentry *ldptr;         /* ptr to file's entry in the   */
                                        /*  in-memory directory         */
        int32   onebyte;                /* next data byte in the file   */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);
        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Return EOF for any attempt to read beyond the end-of-file */

        ldptr = lfptr->lfdirptr;
```

```
        if (lfptr->lfpos >= ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return EOF;
        }
        /* If byte pointer is beyond the current data block, */
        /*      set up a new data block                      */
        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {
                lfsetup(lfptr);
        }
        /* Extract the next byte from block, update file position, and  */
        /*      return the byte to the caller                           */
        onebyte = 0xff & *lfptr->lfbyte++;
        lfptr->lfpos++;
        signal(lfptr->lfmutex);
        return onebyte;
}
```

如果文件没有打开，lfGetc 就返回 SYSERR。如果当前文件位置超过了文件大小，那么 lfGetc 返回 EOF。在其他情况下，lfGetc 检查指针 lfbyte，看它是否超出了 lfdblock 中数据块的范围。若超出了，lfGetc 就调用函数 lfsetup 将正确的索引块和数据块读入内存。

438 ~ 441

一旦数据块在内存中，lfGetc 就可以提取一个字节。为了完成这个操作，lfGetc 对 lfbyte 进行解引用，获得一个字节并将它放到变量 onebyte 中。在把这个字节返回给调用者之前，lfGetc 递增字节指针和文件位置。

19.19 改变文件中的一个字节（lfPutc）

函数 lfPutc 将一个字节存储在文件当前位置。与 lfGetc 一样，实现数据传输很简单，仅需要几行代码。指针 lfbyte 给出 lfdblock 中的一个位置，字节必须存储在这个位置上。代码长用指针来存储指定的字节，递增指针，并设置 lfddirty 来表明这个数据块被改变了。注意 lfPutc 只是把字符加到内存的缓冲区中。每次改变发生时，它并不把缓冲区的内容写回磁盘。只有在当前位置移动到下一个磁盘块时，缓冲区才复制到磁盘中。

与 lfGetc 一样，lfPutc 在每次调用时都要检查 lfbyte。如果 lfbyte 位于数据块 lfdblock 的外面，lfPutc 就返回 SYSERR。然而，lfPutc 和 lfGetc 在处理非法文件位置方式上有细微的不同。如果文件位置超过了文件的最后一个字节，lfGetc 总是返回 EOF。当文件位置超过文件结尾多个字节时，lfPutc 返回 SYSERR，但是如果位置正好超过结尾一个字节，它允许操作继续进行。就是说，它允许对文件进行扩展。当扩展一个文件时，在相应的目录项中其文件大小必须增加。文件 lfPutc.c 包含以下代码。

```
/* lflPutc.c  -  lfPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflPutc  -  write a single byte to an open local file
 *------------------------------------------------------------------------
 */
devcall lflPutc (
          struct dentry *devptr,        /* entry in device switch table */
          char          ch              /* character (byte) to write    */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
        struct  ldentry *ldptr;         /* ptr to file's entry in the   */
                                        /*  in-memory directory         */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
```

```
            signal(lfptr->lfmutex);
            return SYSERR;
    }

    /* Return SYSERR for an attempt to skip bytes beyond the */
    /*      current end of the file                          */

    ldptr = lfptr->lfdirptr;
    if (lfptr->lfpos > ldptr->ld_size) {
            signal(lfptr->lfmutex);
            return SYSERR;
    }

    /* If pointer is outside current block, set up new block */

    if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {

            /* Set up block for current file position */
            lfsetup(lfptr);
    }

    /* If appending a byte to the file, increment the file size.   */
    /* Note: comparison might be equal, but should not be greater. */

    if (lfptr->lfpos >= ldptr->ld_size) {
            ldptr->ld_size++;
    }

    /* Place byte in buffer and mark buffer "dirty" */

    *lfptr->lfbyte++ = ch;
    lfptr->lfpos++;
    lfptr->lfdbdirty = TRUE;

    signal(lfptr->lfmutex);
    return OK;
}
```

19.20 载入索引块和数据块（lfsetup）

一旦将一个文件位置分配给字段 lfpos，函数 lfsetup 就从磁盘中载入与这个位置相关的索引块和数据块的副本。lfsetup 从获得指向数据结构的指针开始。如果现有的索引块或数据块已经改变，lfsetup 就调用 lfflush 把它们写回磁盘，然后检查文件控制块中的索引块。

442, 443

载入当前文件位置数据的第一步是载入一个索引块。这个索引块在当前位置之前或者恰好与当前位置一致。这里有两种情况。如果没有索引块被载入（即文件刚刚被打开），lfsetup 就获取一个索引块。对于一个新文件，lfsetup 必须从空闲列表中分配一个初始索引块；对于一个已经存在的文件，它载入这个文件的第一个索引块。如果索引块已经被载入，lfsetup 就必须处理下面这种情况：索引块对应于文件中在当前文件位置之后的一部分（例如，进程已经给更早的位置发出了 seek）。为此，lfsetup 用文件最初的索引块来替换这个索引块。

一旦 lfsetup 载入了索引块，它就会进入一个循环，沿着索引块的链表前移直到到达覆盖当前文件位置的索引块。每次迭代中，lfsetup 使用 lb_next 字段找到链表下一个索引块的位置，然后调用 lfbget 将索引块读到内存中。

一旦正确的索引块被载入，lfsetup 必须决定需要载入的数据块。为了确定这些数据块，它使用文件位置来计算数据块数组的索引（从 0～15）。因为每一个索引块仅覆盖 8KB（即 2^{13} 字节）的数据，并且数组中的每项对应于一个 512 字节的块（2^9），可以使用二进制算术来进行计算：lfsetup 计算 LF

IMASK（低13位）的逻辑与，然后将结果向右移9位。

lfsetup 使用上述计算的结果作为对数组 ib_dba 的索引来获得数据块的 ID。此时会有两种情况需要 lfsetup 载入一个新的数据块。第一种情况是，数组中的指针为空，意味着 lfPute 准备在文件末尾写一个新的字节，没有数据块被分配到这个位置。lfsetup 调用 lfdballoc 从空闲链表中分配一个新的数据块，记录数组 ib_dba 中实项的 ID。第二种情况是，数组 ib_dba 中的项指定数据块，而不是当前被载入的数据块。lfsetup 调用 read 从磁盘中获取正确的数据块。

作为返回前的最后一步，lfsetup 使用文件位置来计算数据块中的位置，并且分配这个地址给 lfbyte 字段。数据块的大小为 2 的幂，这样任何的安排意味着从 0 ~ 511 之间的索引可以通过选择文件位置的低 9 位来计算。这段代码使用和 LF_DMASK 的逻辑与。文件 lfsetup.c 包含这段代码。

```
/* lfsetup.c  -  lfsetup */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsetup  -  set a file's index block and data block for the current
 *               file position (assumes file mutex held)
 *------------------------------------------------------------------------
 */
status  lfsetup (
          struct lflcblk  *lfptr        /* ptr to slave file device     */
        )
{
        dbold32  dnum;                  /* data block to fetch          */
        ibid32   ibnum;                 /* i-block number during search */
        struct  ldentry *ldptr;         /* ptr to file entry in dir.    */
        struct  lfiblk  *ibptr;         /* ptr to in-memory index block */
        uint32  newoffset;              /* computed data offset for     */
                                        /*  next index block            */
        int32   dindex;                 /* index into array in an index */
                                        /*  block                       */


        /* Obtain exclusive access to the directory */

        wait(lf_data.lf_mutex);

        /* Get pointers to in-memory directory, file's entry in the     */
        /*      directory, and the in-memory index block                */

        ldptr = lfpcb->lfdirptr;
        ibptr = &lfptr->lfiblock;

        /* If existing index block or data block changed, write to disk */

        if (lfptr->lfibdirty || lfpcb->lfdbdirty) {
                lfflush(lfptr);
        }
        ibnum = lfptr->lfinum;          /* get ID of curr. index block  */

        /* If there is no index block in memory (e.g., because the file */
        /*      was just opened), either load the first index block of  */
        /*      the file or allocate a new first index block            */

        if (ibnum == LF_INULL) {

                /* Check directory entry to see if index block exists   */
```

```
ibnum = ldptr->ld_ilist;
if (ibnum == LF_INULL) { /* empty file - get new i-block*/
        ibnum = ifiballoc();
        lfibclear(ibptr, 0);
        ldptr->ld_ilist = ibnum;
        lfptr->lfibdirty = TRUE;
        } else {                /* nonempty - read first i-block*/
                lfibget(lf_data.lf_dskdev, ibnum, ibptr);
        }
        lfptr->lfinum = ibnum;

/* Otherwise, if current file position has been moved to an     */
/*   offset before the current index block, start at the        */
/*   beginning of the index list for the file                   */

} else if (lfptr->lfpos < ibptr->ib_offset) {

        /* Load initial index block for the file (we know that  */
        /*      at least one index block exists)                */

        ibnum = ldptr->ld_ilist;
        lfibget(lf_data.lf_dskdev, ibnum, ibptr);
        lfptr->lfinum = ibnum;
}

/* At this point, an index block is in memory, but may cover    */
/*  an offset less than the current file position.  Loop until  */
/*  the index block covers the current file position.           */

while ((lfptr->lfpos & ~LF_IMASK) > ibptr->ib_offset ) {
        ibnum = ibptr->ib_next;
        if (ibnum == LF_INULL) {
                /* allocate new index block to extend file */
                ibnum = ifiballoc();
                ibptr->ib_next = ibnum;
                lfibput(lf_data.lf_dskdev, lfptr->lfinum, ibptr);
                lfptr->lfinum = ibnum;
                newoffset = ibptr->ib_offset + LF_IDATA;
                lfibclear(ibptr, newoffset);
                lfptr->lfibdirty = TRUE;
        } else {
                lfibget(lf_data.lf_dskdev, ibnum, ibptr);
                lfptr->lfinum = ibnum;
        }
        lfptr->lfdnum = LF_DNULL; /* Invalidate old data block */
}
/* At this point, the index block in lfiblock covers the        */
/*  current file position (i.e., position lfptr->lfpos).  The   */
/*  next step consists of loading the correct data block.       */

dindex = (lfptr->lfpos & LF_IMASK) >> 9;

/* If data block index does not match current data block, read  */
/*   the correct data block from disk                           */

dnum = lfptr->lfiblock.ib_dba[dindex];
if (dnum == LF_DNULL) {         /* allocate new data block */
        dnum = lfdballoc((struct lfdbfree *)&lfptr->lfdblock);
```

```
                lfptr->lfiblock.ib_dba[dirBlk] = dnum;
                lfptr->lfiDirty = TRUE;
        } else if ( dnum != lfptr->lfdnum) {
                readDL( lf_data.lf_dskdev, (char *)lfptr->lfdblock, dnum);
                lfptr->lfDirty = FALSE;
        }
        lfptr->lfdnum = dnum;

        /* Use current file offset to set the pointer to the next byte  */
        /*   within the data block                                      */

        lfptr->lfbyte = &lfptr->lfdblock[lfptr->lfpos & LF_BMASK];
        signal(Lf_data.lf_mutex);
        return OK;
}
```

19.21 主文件系统设备的初始化（lfsInit）

主文件系统设备的初始化很明确，由函数 lfsInit 完成。具体任务包括记录磁盘设备的 ID、创建一个提供目录互斥的信号量、清除内存目录（仅帮助调试）和设置一个布尔值来表明目录还未读入内存。主文件系统设备的数据保存在全局结构 Lf_data 中。文件 lfsInit.c 包含以下代码。

444 – 447

```
/* lfsInit.c  -  lfsInit */

#include <xinu.h>

struct  lfdata  Lf_data;

/*------------------------------------------------------------------------
 * lfsInit  --  initialize the local file system master device
 *------------------------------------------------------------------------
 */
devcall lfsInit (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        /* Assign ID of disk device that will be used */

        Lf_data.lf_dskdev = LF_DISK_DEV;

        /* Create a mutual exclusion semaphore */

        Lf_data.lf_mutex = semcreate(1);

        /* Zero directory area (for debugging) */

        memset((char *)&Lf_data.lf_dir, NULLCH, sizeof(struct lfdir));

        /* Initialize directory to "not present" in memory */

        lf_data.lf_dirpresent = Lf_data.lf_dirdirty = FALSE;

        return OK;
}
```

19.22 伪设备的初始化（lflInit）

当打开一个文件时，lfsOpen 将文件伪设备控制块里的许多表项初始化。然而，有些初始化在系统启动的时候就已经实现了。为了标记设备未被使用，需要将状态的值设置为 LF_FREE。为了确保在一段给定时间里在某个文件上最多只能有一个操作，需要创建一个互斥信号量。控制块里的其他字段都赋值为 0（这些字段只有在文件打开的时候会被用到，而初始化为 0 便于调试）。文

件 lflinit.c 包含了相应代码。 448

```
/* lflInit.c  -  lflInit */

#include <xinu.h>

struct  lflblk lftab[Nlfl];             /* control blocks */

/*------------------------------------------------------------------------
 * lflInit  -  initialize control blocks for local file pseudo-devices
 *------------------------------------------------------------------------
 */
devcall lflInit (
         struct dentry *devptr          /* Entry in device switch table */
        )
{
        struct  lflblk *lfptr;          /* Ptr. to control block entry  */
        int32   i;                      /* Walks through name array     */

        lfptr = &lftab[ devptr->dvminor ];

        /* Initialize control block entry */

        lfptr->lfstate = LF_FREE;       /* Device is currently unused   */
        lfptr->lfdev = devptr->dvnum;   /* Set device ID                */
        lfptr->lfmutex = semcreate(1);
        lfptr->lfdirptr = (struct  ldentry *) NULL;
        lfptr->lfpos = 0;
        for (i=0; i<LF_NAME_LEN; i++) {
                lfptr->lfname[i] = NULLCH;
        }
        lfptr->lfinum = LF_INULL;
        memset((char *) &lfptr->lfiblock, NULLCH, sizeof(struct lfiblk));
        lfptr->lfdnum = 0;
        memset((char *) &lfptr->lfdblock, NULLCH, LF_BLKSIZ);
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ]; /* beyond lfdblock */
        lfptr->lfibdirty = lfptr->lfdbdirty = FALSE;
        return OK;
}
```

19.23 文件截断（lftruncate）

Xinu 用文件截断的方式来释放文件的数据结构。为了使文件长度缩减为零，该文件的所有索引块都必须置于索引块空闲链表上，而这些索引块只有在其所指向的所有数据块都置于数据块空闲链表上的时候才能被释放。lftruncate 函数实现了文件截断，文件 lftruncate.c 包含了相应代码。 449

```
/* lftruncate.c  -  lftruncate */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lftruncate  -  truncate a file by freeing its index and data blocks
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
status  lftruncate (
          struct lflblk *lfptr          /* ptr to file's ctl blk entry */
        )
{
        struct  ldentry *ldptr;         /* pointer to file's dir. entry */
```

```
struct  lfiblk  iblock;         /* buffer for one index block   */
ibid32  ifree;                  /* start of index blk free list */
ibid32  firstib;                /* first index blk of the file  */
ibid32  nextib;                 /* walks down list of the       */
                                /*   file's index blocks        */
dbid32  nextdb;                 /* next data block to free      */
int32   i;                      /* moves through data blocks in */
                                /*   a given index block        */

ldptr = lfptr->lfdirptr;        /* Get pointer to dir. entry    */
if (ldptr->ld_size == 0) {      /* file is already empty */
        return OK;
}

/* Clean up the open local file first */

if ( (lfptr->lfibdirty) || (lfptr->lfdbdirty) ) {
        lfflush(lfptr);
}
lfptr->lfpos = 0;
lfptr->lfinum = LF_INULL;
lfptr->lfdnum = LF_INULL;
lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

/* Obtain ID of first index block on free list */

ifree = lf_data.lf_dir.lfd_ifree;
/* Record file's first i-block and clear directory entry */

firstib = ldptr->ld_ilist;
ldptr->ld_ilist = LF_INULL;
ldptr->ld_size = 0;
lf_data.lf_dirdirty = TRUE;

/* Walk along index block list, disposing of each data block    */
/*  and clearing the corresponding pointer.  A note on loop     */
/*  termination: last pointer is set to ifree below.            */

for (nextib=firstib; nextib!=ifree; nextib=iblock.ib_next) {

        /* Obtain a copy of current index block from disk       */

        lfibget(lf_data.lf_dskdev, nextib, &iblock);

        /* Free each data block in the index block              */

        for (i=0; i<LF_INDEN; i++) {    /* for each d-block     */

                /* Free the data block */

                nextdb = iblock.ib_dba[i];
                if (nextdb != LF_DNULL) {
                        lfdbfree(lf_data.lf_dskdev, nextdb);
                }

                /* Clear entry in i-block for this d-block      */

                        iblock.ib_dba[i] = LF_DNULL;
        }
```

```
                /* Clear offset (just to make debugging easier)         */

                iblock.ib_offset = 0;

                /* For the last index block on the list, make it point  */
                /*      to the current free list                        */

                if (iblock.ib_next == LF_INULL) {
                        iblock.ib_next = ifree;
                }

                /* Write cleared i-block back to disk */
                lfibput(Lf_data.lf_dskdev, nextib, &iblock);
        }

        /* Last index block on the file list now points to first node   */
        /*   on the current free list.  Once we make the free list      */
        /*   point to the first index block on the file list, the       */
        /*   entire set of index blocks will be on the free list        */

        Lf_data.lf_dir.lfd_ifree = firstib;

        /* Indicate that directory has changed and return */

        Lf_data.lf_dirdirty = TRUE;

        return OK;
}
```

以上所使用的方法很简单：如果文件长度为零，则直接返回给调用函数；否则遍历文件的索引块链表，依次将每个索引块读入内存，调用 fdbfree 释放该索引块所指向的每个数据块。

当遍历到最后一个索引块时，将文件的所有索引块加入到空闲链表中。注意，由于文件的所有索引块都已经链接起来，要完成此步，只需要改变两个指针：首先，把文件最后一个索引块的 next 指针指向当前的空闲链表；其次，把空闲链表指向文件的第一个索引块。

19.24 初始文件系统的创建（lfscreate）

最后一个初始化函数将完整个文件系统的细节加以完善。函数 lfscreate 在磁盘上创建一个初始的空文件系统，即生成一个索引块的空闲链表、一个数据块的空闲链表和一个没有文件的目录。文件 lfscreate.c 包含了相应代码。

```
/* lfscreate.c  -  lfscreate */

#include <xinu.h>
#include <ramdisk.h>

/*------------------------------------------------------------------------
 * lfscreate  --  Create an initially-empty file system on a disk
 *------------------------------------------------------------------------
 */
status  lfscreate (
          did32         disk,           /* ID of an open disk device    */
          ibid32        lfiblks,        /* num. of index blocks on disk */
          uint32        dsiz            /* total size of disk in bytes  */
        )
{
        uint32  sectors;                /* number of sectors to use     */
        uint32  ibsectors;              /* number of sectors of i-blocks*/
```

```
uint32  ibpersector;            /* number of i-blocks per sector*/
struct  lfdir   dir;            /* buffer to hold the directory */
uint32  dblks;                  /* total free data blocks       */
struct  lfiblk  iblock;         /* space for one i-block        */
struct  lfdblock dblock;        /* data block on the free list  */
dbid32  dbindex;                /* index for data blocks        */
int32   retval;                 /* return value from func call  */
int32   i;                      /* loop index                   */

/* Compute total sectors on disk */

sectors = dsiz  / LF_BLKSIZ;    /* truncate to full sector */

/* Compute number of sectors comprising i-blocks */

ibpersector = LF_BLKSIZ / sizeof(struct lfiblk);
ibsectors = (lfiblks+(ibpersector-1)) / ibpersector; /* round up*/
lfiblks = ibsectors * ibpersector;
if (ibsectors > sectors/2) {    /* invalid arguments */
        return SYSERR;
}

/* Create an initial directory */

memset((char *)&dir, NULLCH, sizeof(struct lfdir));
dir.lfd_nfiles = 0;
dbindex= (dbid32)(ibsectors + 1);
dir.lfd_dfree = dbindex;
dblks = sectors - ibsectors - 1;
retval = write(disk,(char *)&dir, LF_AREA_DIR);
if (retval == SYSERR) {
        return SYSERR;
}

/* Create list of free i-blocks on disk */

lfiblear(&iblock, 0);
for (i=0; i<lfiblks-1; i++) {
        iblock.ib_next = (ibid32)(i + 1);
        lfibput(disk, i, &iblock);
}
iblock.ib_next = LF_INULL;
lfibput(disk, i, &iblock);

/* Create list of free data blocks on disk */

memset((char*)&dblock, NULLCH, LF_BLKSIZ);
for (i=0; i<dblks-1; i++) {
        dblock.lf_nextdb = dbindex + 1;
        write(disk, (char *)&dblock, dbindex);
        dbindex++;
}
dblock.lf_nextdb = LF_DNULL;
write(disk, (char *)&dblock, dbindex);
close(disk);
return OK;
}
```

19.25 观点

文件系统是操作系统中最复杂的部分之一。实现时面临的一个问题是文件共享。本章的实现通过添加一个约束来规避这个问题，即规定在给定时间内一个文件只允许被打开一次。一旦我们放宽这个约束，文件系统就必须处理多个指向同一个文件的文件描述符。文件共享会带来很多语义问题：怎样解释覆盖写操作？当一个进程尝试对一个文件的0～N字节进行写操作时，另一个进程同时尝试写入同一个文件的2～N－1字节，此时应该怎么做？文件系统如何保证两个操作中的某一个一定是先执行的？文件系统是否应该允许字节混合编址？文件系统如何管理共享缓存以使操作更高效？

第二个复杂的方面来自其实现。所有对文件的操作必须转换为对磁盘块的操作，因此诸如链表之类的基本数据结构都难以处理。有趣的是，许多复杂性都源于磁盘块的共享。例如，由于一个磁盘块里可以存放多个文件的索引块，两个进程有可能同时访问同一个磁盘块。大部分的文件系统都设置了缓存磁盘块，从而使这种访问更加高效。

第三个复杂的方面来自于对数据安全和恢复的需要。用户认为一旦数据被写入文件，即使断电，这个数据仍然是“安全”的。然而，文件系统并不能在每次应用程序向文件中写入字节的时候都对磁盘进行写操作。因此，设计文件系统的一个大问题就是保持效率与数据安全性的平衡——设计者总是试图把数据丢失的危险降低到最小，同时试图把文件系统的效率提升到最大。 450 ; 454

19.26 总结

文件系统是在非易失性存储上进行操作的。为了保证文件接口与设备接口的一致性，本书把示例系统组织为一个主文件系统设备和一系列的文件伪设备。为了访问文件，进程打开主设备，程序调用返回文件的一个伪设备描述符。文件打开后，就可以对它进行 read、write、getc、putc、seek 和 close 操作了。

本书的设计允许文件动态增长。文件的数据结构包含了目录表项和一个索引块的链表，其中每个索引块都指向一系列的数据块。当使用文件时，驱动软件将索引块和数据块读入内存。对文件的后续访问和读、写作用于内存中的数据块。而当文件的位置超出当前数据块时，文件系统把该块写回磁盘并给内存分配另一个数据块。类似地，当一个文件的位置超出当前索引块所能覆盖的位置时，系统把当前索引块写回磁盘并给内存分配一个新的索引块。

练习

19.1 重新设计 lfRead 和 lfWrite 程序，使其能进行高速复制（即在向当前数据块或者从当前数据块复制数据时，不必重复调用 lfGetc 或者 lfPutc）。重新设计系统，使其允许多个进程同时打开同一个文件。处理所有的写操作以保证文件中的一个给定字节总能包含最后一次写操作。

19.2 空闲数据块是用一个单链表连接起来的。重新设计系统，将它们放在一个文件中（即把0索引块存储为一个未命名文件，文件中的索引块都指向空闲数据块）。对比该设计与原设计的性能。

19.3 在题19.2中，原设计与新设计分配和释放一个数据块所需要的最大磁盘访问次数分别是多少？

19.4 索引块的数目非常重要，因为如果使用太多的索引块，就会浪费本来可以分配给数据块的空间；而如果使用太少的索引块，那么由于索引块的不足会导致数据块的浪费。假设一个索引块中能存放16个数据块指针。一个磁盘块中能存放7个索引块，那么在一个有 n 个磁盘块的磁盘中，如果目录中可以放 k 个文件，那么磁盘需要有多少个索引块？

19.5 当前索引块ID是32位长。重新设计系统，让其使用16位的索引块ID。两者的优、劣分别是什么？ 455

19.6 重新设计系统，使得当一个进程结束时，该进程打开的所有文件都能被关闭。

19.7 改变系统，使文件转换表从设备转换表中分离出来。这两种方法的优、劣分别是什么？

19.8 当改变空闲图表后，函数 lfballoc 会向磁盘中写入目录的一个副本。对于该操作，lfballoc 也可以选择把目录标记为“脏”（dirty）从而推迟写操作。讨论这两种方法的优、劣。

19.9 考虑两个尝试向同一个文件进行写操作的进程。假设其中一个重复地写20个字节的字符A，而另一个则重复地写20个字节的字符B。描述该文件中字符出现的顺序。

19.10 为文件伪设备的驱动创建一个 control 函数，从而允许一个调用程序调用 lftruncate。

19.11 为主文件系统创建一个 control 函数，从而允许一个调用程序调用 lfcreate。 456

远程文件机制

网络使天涯变咫尺。

——佚名

20.1 引言

第16章讨论了使用硬件接口来发送和接收数据包的网络接口和设备驱动器。第18章解释了磁盘硬件和块传输范式。第19章解释了文件系统如何建立包括动态文件在内的高层抽象，并说明文件如何映射到磁盘。

本章通过解释另一种选择——使用远程文件服务器（remote file server），来进一步讨论文件系统。操作系统利用称为服务器（server）的独立计算机来实现文件抽象，而不是本地硬件上。当应用程序请求文件操作时，操作系统就会向服务器发送请求（request）并接收响应（response）。第21章则通过说明如何整合远程和本地文件系统来进一步讨论这个话题。

20.2 远程文件访问

远程文件访问（remote file access）机制有四个概念性的组成部分。第一，操作系统必须包含网络设备的设备驱动（比如，以太网）。第二，操作系统必须包含协议软件（比如，UDP和IP）来处理寻址（addressing），这样数据包才能到达远程服务器，应答才能返回。第三，操作系统必须有远程文件访问软件做为客户端（客户端的功能是生成请求，通过网络发送请求到服务器并接收响应，然后解释响应）。当进程调用远程文件的输入输出操作（比如，读或写）时，远程文件访问软件就会生成一条消息来详细说明这个操作，发送请求到远程文件服务器，并处理响应。第四，网络上的计算机必须运行能够响应每一个请求的远程文件服务器应用程序。

实际上，关于远程文件访问机制的设计会牵到很多问题。远程文件服务器应该提供哪些服务？服务器应该允许客户端创建分层目录（hierarchical directory），还是应该只允许客户端创建数据文件（data file）？这个机制应该允许客户端删除文件吗？如果两个或者两个以上客户端向某一个指定的服务器发送请求，文件应该共享还是每个客户端拥有一份自己的文件？文件应该缓存在客户端机器的内存里吗？例如，当进程从远程文件读取1字节时，客户端软件是否应该请求1000字节，并把剩余的字节保存在缓存中，以避免再次发送请求到远程服务器来获取后续的字节呢？

20.3 远程文件语义

远程文件系统的主要设计考虑是异构性（heterogeneity）：客户端和服务端机器上的操作系统可能不同。因此，远程服务器上的有效文件操作可能与客户端机器上使用的文件操作不同。比如，因为远程文件服务器使用的Xinu运行在UNIX系统上（例如Linux或者Solaris），所以服务器提供的便是来自UNIX文件系统的功能。

大部分Xinu文件操作可以直接映射到UNIX文件操作上。例如，Xinu和UNIX在读（read）操作上使用相同的语义——读请求指定缓冲区大小，读操作指出放进缓冲区的数据字节数。类似地，Xinu写（write）操作也与UNIX写操作有相同的语义。

然而，Xinu语义在很多地方与UNIX语义是有区别的。每一个UNIX文件都有一个由UNIX的userid标识的属主，而Xinu一般没有userid，即使它有，也不会与服务器使用的userid一致。两者甚至在一些小的细节上也有不同。例如，Xinu的打开（open）操作中使用的模式（mode）参数允许调用者

指明文件必须是存在（也就是说，必须不存在）或者是可创建的（也就是说，必须存在）。UNIX 允许创建一个文件，但并不会检测这个文件是否已经存在。与 Xinu 不同的是，如果这个文件存在，UNIX 会将这个文件截断为 0 字节。所以，要实现 Xinu 下的新模式，运行在 UNIX 系统上的远程服务器必须先检测文件是否存在，如果确实存在，则返回错误标志。

20.4　远程文件设计和消息

我们的示例远程文件系统提供了如下的基本功能：一个 Xinu 进程能够创建文件，向文件写数据，搜索文件中的任意位置，从文件中读取数据，截断文件以及删除文件。除此之外，远程文件系统允许 Xinu 进程创建或删除目录。对于每一个操作，系统都定义一个请求消息（从 Xinu 客户端发送到远程文件服务器）和一个响应消息（从服务器返回到 Xinu 客户端）。每个消息都包含一个用来标识操作类型的通用头（common header），一个状态值（用于报错的响应），一个序列号（sequence number），以及文件名。给每个发出的请求都分配一个唯一的序列号，远程文件系统通过检查应答来确保收来的应答与发出的请求相匹配。我们的实现为每个消息类型定义一个结构（structure）。为了避免嵌套结构声明，代码使用一个预处理器定义，RF_MSG_HDR，来表示消息头字段，然后把消息头包含在每一个结构体中。文件 rflssys.h 包含如下代码。

```
/* rflssys.h - definitions for remote file system pseudo-devices */

#ifndef NRF
#define NRF     10
#endif

/* Control block for a remote file pseudo device */

#define RF_MAXNLEN      128             /* Maximum length of file name  */
#define RF_DATALEN      1024            /* Maximum data in read or write*/
#define RF_MODE_R       F_MODE_R        /* Bit to grant read access     */
#define RF_MODE_W       F_MODE_W        /* Bit to grant write access    */
#define RF_MODE_RW      F_MODE_RW       /* Mask for read and write bits */
#define RF_MODE_N       F_MODE_N        /* Bit for "new" mode           */
#define RF_MODE_O       F_MODE_O        /* Bit for "old" mode           */
#define RF_MODE_NO      F_MODE_NO       /* Mask for "n" and "o" bits    */

/* Global data for the remote server */

#ifndef RF_SERVER_IP
#define RF_SERVER_IP    "255.255.255.255"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT  33123
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT     33123
#endif

struct  rfdata  {
        int32   rf_seq;                 /* next sequence number to use  */
        uint32  rf_ser_ip;              /* server IP address            */
        uint16  rf_ser_port;            /* server UDP port              */
        uint16  rf_loc_port;            /* local (client) UDP port      */
        sid32   rf_mutex;               /* mutual exclusion for access  */
        bool8   rf_registered;          /* has UDP port been registered?*/
};
```

```
extern  struct  rfdata  Rf_data;

/* Definition of the control block for a remote file pseudo-device      */

#define RF_FREE 0                       /* Entry is currently unused    */
#define RF_USED 1                       /* Entry is currently in use    */

struct  rflcblk {
        int32   rfstate;                /* entry is free or used        */
        int32   rfdev;                  /* device number of this dev.   */
        char    rfname[RF_NAMELEN];     /* Name of the file             */
        uint32  rfpos;                  /* current file position        */
        uint32  rfmode;                 /* mode: read access, write     */
                                        /*       access or both         */
};

extern  struct  rflcblk rfTab[];        /* remote file control blocks   */

/* Definitions of parameters used when accessing a remote server        */

#define RF_RETRIES      3               /* time to retry sending a msg  */
#define RF_TIMEOUT      1000            /* wait one second for a reply  */

/* Control functions for a remote file pseudo device */

#define RFS_CTL_DEL     F_CTL_DEL       /* Delete a file                */
#define RFS_CTL_TRUNC   F_CTL_TRUNC     /* Truncate a file              */
#define RFS_CTL_MKDIR   F_CTL_MKDIR     /* make a directory             */
#define RFS_CTL_RMDIR   F_CTL_RMDIR     /* remove a directory           */
#define RFS_CTL_SIZE    F_CTL_SIZE      /* Obtain the size of a file    */

/*----------------------------------------------------------------------*/
/*                                                                      */
/*      Definition of messages exchanged with the remote server         */
/*                                                                      */
/*----------------------------------------------------------------------*/

/* Values for the type field in messages */
#define RF_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RF_MSG_RREQ     0x0001          /* Read Request and response    */
#define RF_MSG_RRES     (RF_MSG_RREQ | RF_MSG_RESPONSE)

#define RF_MSG_WREQ     0x0002          /* Write Request and response   */
#define RF_MSG_WRES     (RF_MSG_WREQ | RF_MSG_RESPONSE)

#define RF_MSG_OREQ     0x0003          /* Open request and response    */
#define RF_MSG_ORES     (RF_MSG_OREQ | RF_MSG_RESPONSE)

#define RF_MSG_DREQ     0x0004          /* Delete request and response  */
#define RF_MSG_DRES     (RF_MSG_DREQ | RF_MSG_RESPONSE)

#define RF_MSG_TREQ     0x0005          /* Truncate request & response  */
#define RF_MSG_TRES     (RF_MSG_TREQ | RF_MSG_RESPONSE)

#define RF_MSG_SREQ     0x0006          /* Size request and response    */
#define RF_MSG_SRES     (RF_MSG_SREQ | RF_MSG_RESPONSE)

#define RF_MSG_MREQ     0x0007          /* Mkdir request and response   */
```

```
#define RF_MSG_MRES      (RF_MSG_MREQ | RF_MSG_RESPONSE)

#define RF_MSG_XREQ      0x0008          /* Rmdir request and response   */
#define RF_MSG_XRES      (RF_MSG_XREQ | RF_MSG_RESPONSE)

#define RF_MIN_REQ       RF_MSG_RREQ     /* Minimum request type         */
#define RF_MAX_REQ       RF_MSG_XREQ     /* Maximum request type         */

/* Message header fields present in each message */

#define RF_MSG_HDR                       /* Common message fields        */\
        uint16  rf_type;                 /* message type                 */\
        uint16  rf_status;               /* 0 in req. status in response */\
        uint32  rf_seq;                  /* message sequence number      */\
        char    rf_name[RF_NAMLEN];      /* null-terminated file name    */

/* The standard header present in all messages with no extra fields     */

/*********************************************************************/
/*                                                                   */
/*                              Header                               */
/*                                                                   */
/*********************************************************************/
#pragma pack(2)
struct  rf_msg_hdr {                     /* header fields present in each*/
        RF_MSG_HDR                       /*   remote file system message */
};
#pragma pack()

/*********************************************************************/
/*                                                                   */
/*                              Read                                 */
/*                                                                   */
/*********************************************************************/

#pragma pack(2)
struct  rf_msg_rreq      {               /* remote file read request     */
        RF_MSG_HDR                       /* header fields                */
        uint32  rf_pos;                  /* position in file to read     */
        uint32  rf_len;                  /* number of bytes to read      */
                                         /*   (between 1 and 1024)       */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_rres      {               /* remote file read reply       */
        RF_MSG_HDR                       /* header fields                */
        uint32  rf_pos;                  /* position in file             */
        uint32  rf_len;                  /* number of bytes that follow  */
                                         /*   (0 for EOF)                */
        char    rf_data[RF_DATALEN];     /* array containing data from   */
                                         /*   the file                   */
};
#pragma pack()

/*********************************************************************/
/*                                                                   */
/*                              Write                                */
/*                                                                   */
/*********************************************************************/
```

```
#pragma pack(2)
struct  rf_msg_wreq     {               /* remote file write request    */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* position in file             */
        uint32  rf_len;                 /* number of valid bytes in     */
                                        /*   array that follows         */
        char    rf_data[RF_DATALEN];    /* array containing data to be  */
                                        /*   written to the file        */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_wres     {               /* remote file write response   */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* original position in file    */
        uint32  rf_len;                 /* number of bytes written      */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Open                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_oreq     {               /* remote file open request     */
        RF_MSG_HDR                      /* header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_ores     {               /* remote file open response    */
        RF_MSG_HDR                      /* header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Size                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_sreq     {               /* remote file size request     */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_sres     {               /* remote file status response  */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_size;                /* size of file in bytes        */
};
#pragma pack()
```

```
/*****************************************************************************/
/*                                                                           */
/*                                Delete                                     */
/*                                                                           */
/*****************************************************************************/

#pragma pack(2)
struct  rf_msg_dreq     {               /* remote file delete request    */
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_dres     {               /* remote file delete response   */
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

/*****************************************************************************/
/*                                                                           */
/*                               Truncate                                    */
/*                                                                           */
/*****************************************************************************/

#pragma pack(2)
struct  rf_msg_treq     {               /* remote file truncate request  */
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_tres     {               /* remote file truncate response*/
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

/*****************************************************************************/
/*                                                                           */
/*                                Mkdir                                      */
/*                                                                           */
/*****************************************************************************/

#pragma pack(2)
struct  rf_msg_mreq     {               /* remote file mkdir request     */
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_mres     {               /* remote file mkdir response    */
        RF_MSG_HDR                      /* header fields                 */
};
#pragma pack()

/*****************************************************************************/
/*                                                                           */
/*                                Rmdir                                      */
/*                                                                           */
/*****************************************************************************/
```

```
#pragma pack(2)
struct  rf_msg_xreq     {               /* remote file rmdir request    */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_xres     {               /* remote file rmdir response   */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()
```

在文件中，以 RF_MSG_ 开头的常量为每条消息定义了一个唯一的类型值。例如，RF_MSG_RREQ 定义了用于读请求消息的类型值，RF_MSG_RRES 定义了用于读响应消息的类型值。我们在实现中使用了一个小窍门来改善效率：响应消息的类型值是由请求消息的类型值和常量 RF_MSG_RESPONSE（值被定义为 0x0100）进行逻辑或操作生成。也就是说，除了第二个字节的低位设置成 1 外，响应和请求有相同的类型值。

消息的大小取决于类型。许多消息只需要适用消息头中的字段。例如，文件删除请求只需要一个类型（指出这是一个删除请求）、一个文件名，以及一个序列号。因此，定义删除请求的结构体 rf_msg_dreq 只包含消息头字段。然而，一个写请求消息必须包含文件偏移量、请求中的数据字节数，以及将要写入的数据。因此，定义写请求的消息的结构体 rf_msg_wreq 除了包含通用消息头外，还有另外的三个字段。

461 ~ 467

20.5 远程文件服务器通信

本章设计的远程文件系统软件分为两层。低层功能负责与远程服务器通信——发送消息、等待响应和在某些情况下的重新发送。高层功能负责处理消息语义——生成消息、向低层传递消息、接收响应和解析响应。这里的关键思想是：低层只处理消息的传输和接收，不需要理解或解析消息的内容。因此只需要一个单独的函数就提供了所有的低层功能。

以下代码清晰地阐述了这个思想。rfscomm 函数负责向远程文件服务器发送消息并接收响应。文件 rfscomm.c 包含以下的代码。

```
/* rfscomm.c - rfscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscomm - handle communication with RFS server (send request and
 *              receive a reply, including sequencing and retries)
 *------------------------------------------------------------------------
 */
int32   rfscomm (
          struct rf_msg_hdr *msg,       /* message to send              */
          int32  mlen,                  /* message length               */
          struct rf_msg_hdr *reply,     /* buffer for reply             */
          int32  rlen                   /* size of reply buffer         */
        )
{
        int32   i;                      /* counts retries               */
        int32   retval;                 /* return value                 */
        int32   seq;                    /* sequence for this exchange   */
        int16   rtype;                  /* reply type in host byte order*/

        /* For the first time after reboot, register the server port */

        if ( ! Rf_data.rf_registered ) {
```

```
        retval = udp_register(0, Rf_data.rf_ser_port,
                        Rf_data.rf_loc_port);
        Rf_data.rf_registered = TRUE;
}

/* Assign message next sequence number */

seq = Rf_data.rf_seq++;
msg->rf_seq = htonl(seq);

/* Repeat RF_RETRIES times: send message and receive reply */

for (i=0; i<RF_RETRIES; i++) {

        /* Send a copy of the message */

        retval = udp_send(Rf_data.rf_ser_ip, Rf_data.rf_ser_port,
                NetData.ipaddr, Rf_data.rf_loc_port, (char *)msg,
                mlen);
        if (retval == SYSERR) {
                kprintf("Cannot send to remote file server\n\r");
                return SYSERR;
        }

        /* Receive a reply */

        retval = udp_recv(0, Rf_data.rf_ser_port,
                Rf_data.rf_loc_port, (char *)reply, rlen,
                RF_TIMEOUT);

        if (retval == TIMEOUT) {
                continue;
        } else if (retval == SYSERR) {
                kprintf("Error reading remote file reply\n\r");
                return SYSERR;
        }

        /* Verify that sequence in reply matches request */

        if (ntohl(reply->rt_seq) != seq) {
                continue;
        }

        /* Verify the type in the reply matches the request */

                rtype = ntohs(reply->rf_type);
                if (rtype != ( ntohs(msg->rf_type) | RF_MSG_RESPONSE) ) {
                        continue;
                }

                return retval;          /* return length to caller */
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote file server\n\r");
        return TIMEOUT;
}
```

函数的4个参数分别指定了应该发送到服务器的消息的地址、消息的长度、服务器响应消息的缓

冲区地址、缓冲区的长度。当给要发送的消息设定了一个单独的序列号之后，rfscomm 函数进入到一个迭代 RF_RETRIES 次的循环中。每次迭代时，rfscomm 都会尝试调用 udp_send 通过网络发送一份请求消息的副本[⊖]，并且调用 udp_recv 接收响应，直至成功。

udp_recv 允许调用者定义等待响应的最长时间，即 rfscomm 指定的 RF_TIMEOUT[⊖]。如果在指定时间内没有消息到达，则 udp_recv 返回 TIMEOUT，然后通过发送请求的另一个副本循环继续。如果 RF_RETRIES 次尝试都没有响应到达，则 rfscomm 返回 TIMEOUT 给调用者。

如果响应消息在指定时间内到达了，rfscomm 验证它的序列号、消息类型与已发送的请求消息的序列号、消息类型是否匹配。如果任一验证失败，说明服务器生成了错误的消息或者这个消息被发送给了错误的客户端。不管是哪种情况，rfscomm 都会继续循环，发送请求消息的另一份副本并等待响应消息的到达。如果两个验证都成功了，说明到达的消息是合法的响应，然后 rfscomm 返回响应消息的长度给调用者。

20.6 发送一个基本消息

为了理解 rfscomm 如何工作，可以想象一个只需要适用头内各个字段的消息，比如，truncate 操作的请求、响应消息。因为有很多类型的消息只需要适用头内的各个字段，所以 rfsndmsg 函数被用来专门发送这样的消息。文件 rfsndmsg.c 包含以下的代码。

468 - 470

```
/* rfsndmsg.c - rfsndmsg */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsndmsg  -  Create and send a message that only has header fields
 *------------------------------------------------------------------------
 */
status  rfsndmsg (
        uint16 type,                    /* message type                 */
        char   *name                    /* null-terminated file name    */
        )
{
        struct  rf_msg_hdr  req;        /* request message to send      */
        struct  rf_msg_hdr resp;        /* buffer for response          */
        int32   retval;                 /* return value                 */
        char    *to;                    /* used during name copy        */

        /* Form a request */

        req.rf_type = htons(type);
        req.rf_status = htons(0);
        req.rf_seq = 0;                 /* rfscomm will set sequence    */
        to = req.rf_name;
        while ( (*to++ = *name++) != 0 ) {  /* copy name to request     */
                ;
        }

        /* Send message and receive response */

        retval = rfscomm(&req,  sizeof(struct rf_msg_hdr),
                         &resp, sizeof(struct rf_msg_hdr) );

        /* Check response */
```

⊖ 我们说“发送一份请求消息的副本”是因为该消息不会被修改。

⊖ RF_TIMEOUT 定义成 1000 毫秒（即 1 秒），这对于客户端与服务器的一次消息往来是足够的。

```
        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file server access\n\r");
                return SYSERR;
        } else if (ntohl(rcvp.rf_status) != 0) {
                return SYSERR;
        }

        return OK;
}
```

[471]

rfsndmsg 函数需要两个参数，一个用来指定消息的类型，另外一个用来指定文件名。为了创建一条请求消息，代码给 req 变量的每个字段分配一个值。然后调用 rfscomm 发送消息并接收响应。如果 rfscomm返回错误、超时或者响应的状态指出一个错误，rfsndmsg 就返回 SYSERR；否则，rfsndmsg 返回 OK。

20.7　网络字节序

远程文件访问有一个很重要的问题：计算机中整型数的格式（比如，字节序）和其体系结构有关。如果我们把一台计算机内存中的整型数直接复制到另外一台计算机的内存中，那么这个整型数的值可能会发生变化。为了避免这个问题，跨计算机网络传输数据的软件都要遵从一个规范：发送之前要把整型数从本地字节序（local byte order）转换成标准的网络字节序（network byte order），接收之后再把整型数从网络字节序转换回本地字节序。

为了避免不同的字节序对数据解释的不同，网络上传输的整型数必须在发送之前转换成网络字节序，并且在接收之后转换为本地字节序。在我们的设计中，协议软件负责这个转换。

Xinu 遵从 UNIX 命名规范来命名字节序转换函数。htonl（htons）把一个整型（短整型）数从本地字节序转换成网络字节序，ntohl（ntohs）函数把一个整型（短整型）数从网络字节序转换成本低字节序。比如，rfsndmsg 函数调用 htons 把用来标识消息类型和状态的整型数从本地字节序转换成网络字节序。

20.8　使用设备范式的远程文件系统

我们已经看到了，Xinu 使用设备范式（device paradigm）来表示设备和文件。它的远程文件系统同样遵从此模式。图 20-1 显示了定义远程文件系统主设备（master device）类型和一些远程文件伪设备（pseudo device）类型的 Xinu 配置文件。

[472]

```
/* 远程文件系统主设备类型 */

rfs:
        on rfs
                -i rfsInit      -o rfsOpen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n rfsControl
                -intr NULL

/* 远程文件伪设备类型 */

rfl:
        on rfs
                -i rflInit      -o ioerr        -c rflClose
                -r rflRead      -g rflGetc      -p rflPutc
                -w rflWrite     -s rflSeek      -n ioerr
                -intr NULL
```

图 20-1　摘录自 Xinu 配置文件，定义了远程文件系统使用的两个设备类型

图 20-2 摘录的 Xinu 配置文件定义了一个远程文件系统主设备（RFILESYS）和 6 个远程文件伪设备（RFILE0 - RFILE5）。

```
/* 远程文件系统主设备（每个系统一个）*/

RFILESYS is rfs on udp

/* 远程文件伪设备（每个系统多个实例）*/

RFILE0 is rfl on rfs
RFILE1 is rfl on rfs
RFILE2 is rfl on rfs
RFILE3 is rfl on rfs
RFILE4 is rfl on rfs
RFILE5 is rfl on rfs
```

473

图 20-2 摘录自 Xinu 配置文件，定义了远程文件系统使用的设备

当应用程序对远程文件系统的主设备调用打开（open）命令时，这个命令自动分配一个远程文件伪设备并返回它的设备 ID（device ID），之后程序就可以使用这个设备 ID 来对其进行读（read）、写（write）以及最终的关闭（close）操作。20.9 节将定义远程文件系统主设备和远程文件伪设备共同使用的设备驱动函数（device driver functions）。

20.9 打开远程文件

为了打开一个远程文件，程序需要提供文件名和模式参数来对 RFILESYS 设备调用打开（open）命令。打开（open）命令调用 rfsOpen 函数，rfsOpen 函数生成一个请求并调用 rfscomm 与远程文件服务器进行通信。如果成功，打开（open）命令返回与远程文件相关联的远程文件伪设备的描述符（descriptor）。（使用这个描述符就可以对远程文件进行读、写等操作了）。rfsOpen.c 文件包含以下的代码。

```
/* rfsOpen.c - rfsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsOpen - allocate a remote file pseudo-device for a specific file
 *------------------------------------------------------------------------
 */

devcall rfsOpen (
        struct dentry  *devptr,         /* entry in device switch table */
        char   *name,                   /* file name to use             */
        char   *mode                    /* mode chars: 'r' 'w' 'o' 'n'  */
      )
{
        struct  rflblk *rfptr;          /* ptr to control block entry   */
        struct  rf_msg_oreq msg;        /* message to be sent           */
        struct  rf_msg_ores resp;       /* buffer to hold response      */
        int32   retval;                 /* return value from rfscomm    */
        int32   len;                    /* counts chars in name         */
        char    *nptr;                  /* pointer into name string     */
        char    *fptr;                  /* pointer into file name       */
        int32   i;                      /* general loop index           */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);
```

```
/* Search control block array to find a free entry */

for(i=0; i<Nrfl; i++) {
        rfptr = &rflrab[i];
        if (rfptr->rrstate == RF_FREE) {
                break;
        }
}
if (i >= Nrfl) {                /* No free table slots remain   */
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Copy name into free table slot */

nptr = name;
fptr = rfptr->rfname;
len = 0;
while ( (*fptr++ = *nptr++) != NULLCH) {
        len++;
        if (len >= RF_MAXLEN) { /* File name is too long        */
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}

/* Verify that name is non-null */

if (len==0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Parse mode string */

if ( (rfptr->rfmode = rfsetmode(mode)) == SYSERR ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form an open request to create a new file or open an old one */

msg.rf_type = htons(RF_MSG_OREQ);/* Request a file open         */
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* rfscomm fills in seq. number */
nptr = msg.rf_name;
memset(nptr, NULLCH, RF_MAXLEN);/* initialize name to zero bytes*/
while ( (*nptr++ = *name++) != NULLCH ) { /* copy name to req.  */
        ;
}
msg.rf_mode = htonl(rfptr->rfmode); /* Set mode in request      */

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_oreq),
                (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_oresp) );
```

```
        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Set initial file position */

        rfptr->rfpos = 0;

        /* Mark state as currently used */

        rfptr->rfstate = RF_USED;

        /* Return device descriptor of newly created pseudo-device */

        signal(Rf_data.rf_mutex);
        return rfptr->rfdev;
}
```

在检查参数之前，rfOpen首先检查是否有空闲的远程设备可用。然后检查文件名是否小于长度限制，模式字符串是否有效。

在分配远程文件伪设备之前，rfOpen必须与远程服务器通信来确认文件可以打开。它首先生成一个请求消息，然后调用rfscomm把消息传递到服务器。如果得到肯定的响应，则设置远程文件设备的控制块表项（control block entry）状态为“使用”，rfOpen设置初始文件位置为零，然后返回描述符给到用者。

20.10 检查文件模式

当需要检查文件模式参数时，rfOpen以模式字符串作为参数调用rfsetmode函数。代码可以在rfsetmode.c中找到。

```
/* rfsetmode.c  - rfsetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsetmode - parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */

int32   rfsetmode (
          char    *mode                 /* string of mode characters    */
        )
{
        int32   mbits;                  /* mode bits to return (in host */
                                        /*    byte order)               */
        char    ch;                     /* next character in mode string*/

        mbits = 0;
```

```
	while ( (ch = *mode++) != NULLCH ) {
		switch (ch) {

		    case 'r':	if (mbits&RF_MODE_R) {
					return SYSERR;
				}
				mbits |= RF_MODE_R;
				continue;

		    case 'w':	if (mbits&RF_MODE_W) {
					return SYSERR;
				}
				mbits |= RF_MODE_W;
				continue;

		    case 'o':	if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
					return SYSERR;
				}
				mbits |= RF_MODE_O;
				break;

		    case 'n':	if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
					return SYSERR;
				}
				mbits |= RF_MODE_N;
				break;

		    default:	return SYSERR;
		}
	}

	/* if neither read nor write specified, allow both */

	if ( (mbits&RF_MODE_RW) == 0 ) {
		mbits |= RF_MODE_RW;
	}
	return mbits;
}
```

rfsetmode 从模式字符串中逐个抽取出字符，假设每个都是有效的，同时检查非法的组合（比如，模式字符串不能同时包含 new 和 old 模式）。在扫描模式字符串的同时，rfsetmode 设置 mbits 变量的各个位。当扫描完整个字符串并检查完非法组合后，rfsetmode 返回整型的 mbits 变量给调用者。

20.11　关闭远程文件

当进程使用完了一个文件后，它可以调用关闭（close）命令来释放相应的远程文件设备，系统将远程文件设备给其他文件使用。对于一个远程文件伪设备，关闭（close）命令调用 rfClose 函数。在我们的实现中，关闭一个远程文件非常简单。rfClose.c 文件包含以下的代码。

```
/* rfClose.c - rfClose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfClose  -  Close a remote file device
 *------------------------------------------------------------------------
 */
devcall rfClose (
	  struct dentry *devptr		/* entry in device switch table */
	)
```

```
{
        struct  rflblk *rfptr;          /* pointer to control block     */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify remote file device is open */

        rfptr = &rftab[devptr->dvminor];
        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Mark device closed */

        rfptr->rfstate = RF_FREE;
        signal(Rf_data.rf_mutex);
        return OK;
}
```

当确认设备当前处于打开状态后，rfClose 设置控制块表项状态为 RF_FREE。注意，这个版本的 rfClose 并不通知文件服务器文件已经关闭。之后的练习会建议你重新设计系统并加入通知远程服务器文件已经关闭的功能。

20.12 读远程文件

一旦一个远程文件被打开，进程将可以从该文件中读数据。驱动器函数 rfRead 将执行读（read）
479 操作。rfRead 的代码可在文件 rfRead.c 中找到。

```
/* rflRead.c - rflRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflRead - Read data from a remote file
 *------------------------------------------------------------------------
 */
devcall rflRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of bytes              */
          int32 count                   /* count of bytes to read       */
        )
{
        struct  rflblk *rfptr;          /* pointer to control block     */
        int32   retval;                 /* return value                 */
        struct  rf_msg_rreq  msg;       /* request message to send      */
        struct  rf_msg_rrep *rrep;      /* buffer for response          */
        int32   i;                      /* counts bytes copied          */
        char    *from, *to;             /* used during name copy        */
        int32   len;                    /* length of name               */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify count is legitimate */
```

```
if ( (count <= 0) || (count > RF_DATALEN) ) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device is in use */

rfptr = &rftab[devptr->dvminor];

/* If device not currently in use, report an error */

if (rfptr->rfstate == RF_FREE) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device allows reading */
if ((rfptr->rfmode & RF_MODE_R) == 0) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}

/* Form read request */

msg.rf_type = htons(RF_MSG_RREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMELEN);  /* start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) != 0 ) {   /* copy name to request        */
        if (++len >= RF_NAMELEN) {
                signal(&rf_data.rf_mutex);
                return SYSERR;
        }
}
msg.rf_pos = htonl(rfptr->rfpos);/* set file position           */
msg.rf_len = htonl(count);       /* set count of bytes to read  */

// Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_rreq),
                 (struct rf_msg_hdr *)&rmsg,
                                sizeof(struct rf_msg_rres) );

/* Check response */

if (retval == SYSERR) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
} else if (retval == TIMEOUT) {
        kprintf("Timeout during remote file read\n\r");
        signal(&rf_data.rf_mutex);
        return SYSERR;
} else if (ntohs(resp.rf_status) != 0) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}
```

```
        /* Copy data to application buffer and update file position */

        for (i=0; i<ntohl(resp.rf_len); i++) {
                *buff++ = resp.rf_data[i];
        }
        rfptr->rfpos += ntohl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return ntohl(resp.rf_len);
}
```

rfRead 函数先检查参数 count，验证请求消息没有超界。然后它验证该设备已打开并且当前模式允许读操作。一旦检查完成，rfRead 函数执行读（read）操作：它构造读请求消息，使用 rfscomm 函数将消息的副本发送给服务器，并接收响应，解析响应消息。

如果 rfscomm 函数返回一个有效的响应，则此消息将包含读取的数据。rfRead 函数将数据从响应消息复制到调用者的缓冲区，更新文件位置，并将读取数据的字节数返回给调用者。

20.13 写远程文件

向一个远程文件写数据的流程与从一个远程文件读数据的流程相同。驱动器函数 rfWrite 执行写（write）操作。它的代码可在文件 rfWrite.c 中找到。

```
/* rfWrite.c - rfWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfWrite - Write data to a remote file
 *------------------------------------------------------------------------
 */
devcall rfWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of bytes              */
          int32 count                   /* count of bytes to write      */
        )
{
        struct  rfcblk *rfptr;          /* pointer to control block     */
        int32   retval;                 /* return value                 */
        struct  rf_msg_wreq  msg;       /* request message to send      */
        struct  rf_msg_wres resp;       /* buffer for response          */
        char    *from, *to;             /* used to copy name            */
        int     i;                      /* counts bytes copied into req */
        int32   len;                    /* length of name               */

        /* Wait for exclusive access */

        wait(RF_data.rf_mutex);

        /* Verify count is legitimate */

        if ( (count <= 0) || (count > RF_DATALEN) ) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Verify pseudo-device is in use and mode allows writing */

        rfptr = &rftab[devptr->dvminor];
```

```
if ( (rfptr->rfstate == RF_FREE) ||
     ( (rfptr->rfmode & RF_MODE_W) == 0) ) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}

/* Form write request */

msg.rf_type = htons(RF_MSG_WREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMELEN);  /* start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) != 0 ) {   /* copy name to request        */
        if (++len >= RF_NAMELEN) {
                signal(&rf_data.rf_mutex);
                return SYSERR;
        }
}
while ( (*to++ = *from++) != 0 ) {   /* copy name into request      */
        ;
}
msg.rf_pos = htonl(rfptr->rfpos);/* set file position           */
msg.rf_len = htonl(count);       /* set count of bytes to write */
for (i=0; i<count; i++) {        /* copy data into message      */
        msg.rf_data[i] = *buff++;
}
while (i < RF_DATALEN) {
        msg.rf_data[i++] = NULLCH;
}

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_wreq),
                (struct rf_msg_hdr *)&resp,
                        sizeof(struct rf_msg_wres) );

/* Check response */

if (retval == SYSERR) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
} else if (retval == TIMEOUT) {
        kprintf("Timeout during remote file read\n\r");
        signal(&rf_data.rf_mutex);
        return SYSERR;
} else if (ntohs(resp.rf_status) != 0) {
        signal(&rf_data.rf_mutex);
        return SYSERR;
}

/* Report results to caller */

rfptr->rfpos += ntohl(resp.rf_len);

signal(&rf_data.rf_mutex);
return ntohl(resp.rf_len);
}
```

与读（read）操作一样，rfWrite 函数先检查参数 count，验证伪设备已打开并且当前模式允许写。然后 rfWrite 函数构造写请求消息，使用 rfscomm 函数将消息发送给服务器。

480 ~ 484

与读（read）请求不一样的是，写（write）请求包含数据。因此，当构造写请求消息时，rfWrite 函数将数据从用户的缓冲区复制到请求消息中。当一个响应到达时，响应消息并不包含已写入数据的副本。因此，rfWrite 函数在消息中使用状态字段来决定向调用者报告成功还是失败。

20.14 远程文件的定位

远程文件系统应当如何实现定位（seek）操作？有两种可能的选择。在一种设计中，系统发送消息给远程文件服务器，远程文件服务器定位到文件的指定位置。在另一种设计中，所有的位置数据都保存在本地计算机中，每一个向服务器发送的请求都包含一个显式的文件位置。

本章的实现使用了后者：当前文件位置存储在远程文件设备的控制块入口中。当读（read）操作被调用时，rfRead 向服务器请求数据并相应地更新控制块入口中的文件位置数据。因为每一个请求都包含显式的位置信息，所以远程服务器无需记录位置信息。

因为所有的文件位置信息都存储在客户端，所以定位（seek）操作可以在本地进行。这意味着在下一次读（read）或写（write）操作中，存储在控制块入口中的文件位置信息仍可以使用。rfSeek 函数在远程文件设备上执行定位操作。函数的代码可在文件 rfSeek.c 中找到。

```
/* rfSeek.c - rfSeek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfSeek  -  change the current position in an open file
 *------------------------------------------------------------------------
 */
devcall	rfSeek (
	  struct dentry *devptr,	/* entry in device switch table */
	  uint32 pos			/* new file position		*/
	)
{
	struct	rfcblk *rfptr;		/* pointer to control block	*/

	/* Wait for exclusive access */

	wait(Rf_data.rf_mutex);

	/* Verify remote file device is open */

	rfptr = &rfltab[devptr->dvminor];
	if (rfptr->rfstate == RF_FREE) {
		signal(Rf_data.rf_mutex);
		return SYSERR;
	}
	/* Set the new position */

	rfptr->rfpos = pos;
	signal(Rf_data.rf_mutex);
	return OK;
}
```

上面的代码很简单。在得到独占访问权限后，rfSeek 函数验证设备是否打开。接着函数在控制块的 rfpos 字段存储文件位置参数，发出互斥信号量，然后返回。

20.15 远程文件单字符 I/O

使用远程文件服务器读、写单个字节数据的代价是昂贵的，因为每一个字符都必须有一个对

应的消息发送到服务器上。但是 Xinu 实现并没有禁止单个字符的 I/O，有关 getc 和 putc 的实现仅仅只是分别调用了远程文件函数 rflRead 和 rflWrite。函数的代码可在文件 rflGetc.c 和 rflPut.c 中找到。

```
/* rflGetc.c - rflGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflGetc - read one character from a remote file (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall rflGetc(
        struct  dentry  *devptr         /* entry in device switch table */
        )
{
        char    ch;                     /* character to read            */
        int32   retval;                 /* return value                 */

        retval = rflRead(devptr, &ch, 1);

        if (retval != 1) {
                return SYSERR;
        }

        return (devcall)ch;
}
/* rflPutc.c - rflPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflPutc - write one character to a remote file (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall rflPutc(
        struct  dentry  *devptr,        /* entry in device switch table */
        char    ch                      /* character to write           */
        )
{
        struct  rflblk *rfptr;          /* pointer to rfl control block */

        rfptr = &rfltab[devptr->dvminor];

        if (rflWrite(devptr, &ch, 1) != 1) {
                return SYSERR;
        }

        return OK;
}
```

20.16　远程文件系统控制函数

很多的文件操作需要打开（open）、读（read）、写（write）和关闭（close）函数的底层支持。例如，这些函数对删除一个文件而言是必需的。Xinu 远程文件系统使用控制（control）函数来实现这些功能。图 20-3 列出了控制函数所使用的符号常量及其意义。

一个控制（control）函数是在设备 RFILESYS（远程文件系统的主设备）上执行，而不是在独立的远程文件设备上执行。驱动器函数 rfsControl 实现了控制（control）操作，它的代码可在文件 rfsCon-

tral.c 中找到。

常量	意义
RFS_CTL_DEL	删除给定文件
RFS_CTL_TRUNC	截断给定文件为0字节
RFS_CTL_MKDIR	创建一个目录
RFS_CTL_RMDIR	删除一个目录
RFS_CTL_SIZE	返回当前文件的字节数

485
|
487

图 20-3 在远程文件系统中使用的控制函数

```
/* rfsControl.c - rfsControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsControl - Provide control functions for the remote file system
 *------------------------------------------------------------------------
 */
devcall rfsControl (
        struct dentry  *devptr,         /* entry in device switch table */
        int32  func,                    /* a control function           */
        int32  arg1,                    /* argument #1                  */
        int32  arg2                     /* argument #2                  */
        )
{
        int32   len;                    /* length of name               */
        struct  rf_msg_sreq msg;        /* buffer for size request      */
        struct  rf_msg_sres resp;       /* buffer for size response     */
        struct  rfltblk *rfptr;         /* pointer to entry in rfltab   */
        char    *to, *from;             /* used during name copy        */
        int32   retval;                 /* return value                 */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Check length and copy (needed for size) */

        rfptr = &rfltab[devptr->dvminor];
        from = rfptr->rfname;
        to = msg.rf_name;
        len = 0;
        memset(to, NULLCH, RF_NAMELEN);  /* start name as all zeroes    */
        while ( (*to++ = *from++) ) {    /* copy name to message        */
                len++;
                if (len >= (RF_NAMELEN - 1) ) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
        }
        switch (func) {

        /* Delete a file */

        case RFS_CTL_DEL:
                if (rfsendmsg(RF_MSG_DREQ, (char *)&msg) == SYSERR) {
```

```
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Truncate a file */

case RFS_CTL_TRUNC:
        if (rfsndmsg(RF_MSG_TRBQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;


/* Make a directory */

case RFS_CTL_MKDIR:
        if (rfsndmsg(RF_MSG_MRBQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Remove a directory */

case RFS_CTL_RMDIR:
        if (rfsndmsg(RF_MSG_XRBQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Obtain current file size (non-standard message size) */

case RFS_CTL_SIZE:

        /* Hand-craft a size request message */
        msg.rf_type = htons(RF_MSG_SRBQ);
        msg.rf_status = htons(0);
        msg.rf_seq = 0;         /* rfscomm will set the seq num */

        /* Send the request to server and obtain a response     */

        retval = rfscomm( (struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_sreq),
                        (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_sresp) );
        if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else {
                signal(Rf_data.rf_mutex);
                return ntohl(resp.rf_size);
        }

default:
        kprintf("rfsControl: function %d not valid\n\r", func);
```

```
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        signal(Rf_data.rf_mutex);
        return OK;
}
```

对所有的控制函数而言，参数 arg1 包含了一个指向以空字符终止的函数名称的指针。在得到对文件的独占访问并检查文件名长度之后，rfsControl 函数使用函数参数在多个操作中做出选择。这些操作包括文件删除、文件截断、创建目录、删除目录和文件大小请求。在每个操作中，rfsControl 函数必须向远程文件服务器发送消息并接收响应。

除了文件的大小请求外，所有发送给服务器的消息只包含适用头字段。因此，除了文件大小请求函数外，rfsControl 使用 rfsndmsg 生成请求并向服务器发送该请求。对于文件大小请求，rfsControl 在变量 msg 中创建一个消息，并使用 rfscomm 函数来发送消息，接收响应。为了避免扫描文件名两次，rfsControl 函数在检查文件名长度时，将文件名复制到变量 msg 的名称字段中。因此，当 rfsControl 函数创建文件大小请求时，就不需要额外的文件名复制操作。如果一个文件大小请求的有效响应到达，rfsControl 函数就从响应中抽取文件大小，将它转换为本地字节序，然后将其值返回给调用者。在其他操作中，rfsControl 函数既可能返回状态 OK，也可能返回状态 SYSERR。

488 > 490

20.17 初始化远程文件数据结构

因为本章程序设计包括一个远程文件系统主设备和一系列远程文件伪设备，所以系统需要两个初始化函数。第一个是 rfsInit 函数，它初始化与主设备相关的控制块。rfsInit.c 文件中包含以下代码。

```
/* rfsInit.c - rfsInit */

#include <xinu.h>

struct  rfdata  Rf_data;

/*------------------------------------------------------------------------
 *  rfsInit - initialize the remote file system master device
 *------------------------------------------------------------------------
 */
devcall rfsInit(
          struct dentry *devptr          /* entry in device switch table */
        )
{

        /* Choose an initial message sequence number */

        Rf_data.rf_seq = 1;

        /* Set the server IP address, server port, and local port */

        if ( dot2ip(RF_SERVER_IP, &Rf_data.rf_ser_ip) == SYSERR ) {
                panic("invalid IP address for remote file server");
        }
        Rf_data.rf_ser_port = RF_SERVER_PORT;
        Rf_data.rf_loc_port = RF_LOC_PORT;

        /* Create a mutual exclusion semaphore */

        if ( (Rf_data.rf_mutex = semcreate(1)) == SYSERR ) {
                panic("Cannot create remote file system semaphore");
        }

        /* Specify that the server port is not yet registered */

        Rf_data.rf_registered = FALSE;
```

```
        return OK;
}
```

主设备的数据保存在全局变量 RF_data 中。rfsInit 函数在 RF_data 结构的字段中设置远程服务器的 IP 地址和 UDP 端口号，还分配一个互斥信号量并把该信号量 ID 保存在结构中。rfsInit 函数把 rf_registered 字段设置为 FALSE，表示在与服务器通信之前，必须用网络代码计算服务器的 UDP 端口。

rfsInit 函数初始化各个远程文件设备。rfsInit.c 文件中的代码如下：

```
/* rfsInit.c - rfsInit */

#include <xinu.h>

struct  rfcblk rf_tab[Nrf];             /* rf device control blocks     */

/*------------------------------------------------------------------------
 *  rfsInit - initialize a remote file device
 *------------------------------------------------------------------------
 */
devcall rfsInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rfcblk *rfptr;          /* ptr. to control block entry  */
        int32   i;                      /* walks through name array     */

        rfptr = &rftab[ devptr->dvminor ];

        /* Initialize entry to unused */

        rfptr->rfstate = RF_FREE;
        rfptr->rfdev = devptr->dvnum;
        for (i=0; i<RF_NAMLEN; i++) {
                rfptr->rfname[i] = NULLCH;
        }
        rfptr->rfpos = rfptr->rfmode = 0;
        return OK;
}
```

rfsInit 把表项的状态设置为 RF_FREE，表明该表项目前未被使用。该函数也将 rfptr 变量的名字和模式字段置零。如果 rfptr 的状态标记为 RF_FREE，那么该表项的其他字段本待被引用。将该表项的字段置零有助于程序调试。

191
|
192

20.18　观点

与本地文件系统一样，设计远程文件系统过程中的最复杂决策在于如何在效率和文件共享之间寻找平衡。为了理解这个选择，想象运行在多个计算机上的共享一个文件的多个应用程序。在极端情况下，为了保证共享文件最后写语义的正确性，文件的各个操作应按它们出现的顺序发送到远程服务器。这种可以将请求序列化并应用到文件上。在另一个极端情况下，计算机可以缓存文件（或者部分文件）并且可以从本地缓存中读取文件信息，效率可以达到最大化。本书设计远程文件系统的目标是，在没有文件共享时，性能最大化；在需要共享文件时，保证正确性，并能够在这两个极端之间自动、优雅地切换。

20.19　总结

远程文件访问机制允许客户端计算机访问存储在远程服务器上的文件。示例中使用一种通过调用远程文件系统上设备的打开（open）函数来获得远程文件伪设备 ID 的设备范式。然后应用程序可以在伪设备上调用读（read）和写（write）函数。

当应用程序访问远程文件时，远程文件程序创建一条消息，发送消息到远程文件服务器，等待响应并解析响应。程序传送各个请求多次以防网络丢包或服务器过于忙碌而难以回应。

文件删除、截断、创建和删除目录，文件大小查询等操作由 control 函数完成。与数据传输操作一样，调用 control 函数将引起请求消息的传送和来自服务器的响应。

练习

20.1 修改远程文件服务器和 rfClose 函数，使得 rfClose 函数在每次文件关闭时发送消息到服务器并让服务器返回响应。

20.2 底层协议限制读请求的数据大小为 RF_DATALEN 字节，rfRead 拒绝任何申请更大尺寸的请求。修改 rfRead 函数使得用户可以申请任意大小的数据，但是仍限制请求消息的大小为 RF_DATALEN（也就是说，不拒绝更大的请求，但限制返回的数据为 RF_DATALEN 字节）。

493 20.3 类似上述练习，请设计一个系统，使 rfRead 函数能够申请任意大小的数据并通过发送多个请求来完成。

20.4 rfCete 中的代码直接调用 rfRead 函数，这样的设计会产生什么潜在的问题？修改代码使得在调用该函数时使用设备转换表。

20.5 考虑另一个能够提高效率的远程文件系统方案。修改 rfRead 函数使得它每次均请求 RF_DATALEN 字节，以便调用者访求更小的尺寸。将额外的字节放在缓存中，并让它可用于接下来的调用。

20.6 在前面的练习中，将数据缓存用于接下来的读调用的主要缺陷是什么？（提示：考虑服务器的文件共享。）

20.7 考虑当两个客户端同时企图使用远程文件服务器时会发生什么。当每个客户端启动时，它们均将起始包顺序号设置为 1，这使得出现冲突的概率极高。修改系统，改用随机起始包顺序号

494 （同时修改服务器使它能够接受任意起始顺序号）。

句法名字空间

> 玫瑰的另一个名字。
>
> ——威廉·莎士比亚

21.1 引言

本书第14章简述了一系列设备无关的输入/输出操作（包括读和写），描述了设备转换表如何为每个设备提供高层操作与驱动器函数之间的有效映射。后面的章节将详细描述如何构建设备驱动器，并提供相应示例。第20章解释了文件系统如何嵌入设备范式，并解释了伪设备的概念。

本章将讲解设备名字，解释如何从语法角度理解名字，在统一名字空间中如何表示设备与文件。

21.2 透明与名字空间的抽象

对上层的透明是操作系统设计的基本原则之一：

> 只要可能，应用程序就不应该知道实现的细节，如对象的位置或者它的表示形式。 497

例如，当应用程序创建一个新进程时，它不需要知道栈空间在何处分配。类似地，当应用程序打开了一个本地文件时，它也不需要知道该文件所在的磁盘块。

对于文件访问，Xinu范式似乎违反了透明原则，因为它在应用程序打开文件时需要用户提供文件系统名称。例如，本地文件系统的主设备名为LFILESYS。当Xinu系统包含远程文件系统时，就更加严重地违背了透明原则：程序员必须知道远程文件系统的主设备名称（RFILESYS），必须在本地文件和远程文件之间做出选择。而且，文件名也必须符合特定系统的命名风格。

如何才能在文件与设备命名上保持透明呢？答案就是提供一个高层抽象——名字空间。从概念上讲，名字空间提供一系列统一的名字，这些名字将不同的文件命名方式整合为一个整体，允许用户在不对文件位置的情况下打开文件或设备。UNIX系统通过文件系统接口名字空间抽象：本地文件、远程文件和设备在分层文件名字空间中命名。例如：名字/dev/console通常对应于系统控制台设备，而名字/dev/usb对应于USB设备。

Xinu采用一个新颖的方法将名字空间机制与底层文件系统进行分离。而且，Xinu使用句法的方法，也就是说，名字空间检查名字时不需要理解它们的具体含义。简单和能力的结合是名字空间有吸引力的原因。通过将名字理解为一个字符串，我们可以理解它们之间的相似性。通过使用前缀字符串与树之间的关系，能够轻易地操纵名字。通过遵循透明原则，系统效率能够得到大幅地提升。通过在现有机制中添加一个中间层的办法，能够实现统一命名的目的。

在介绍名字空间机制之前，本书先通过一些已有的文件命名示例来了解遇到的问题。讨论文件命名后，读者将了解通用句法命名方案，然后验证一个简单的、专用的方案。最后，将验证一个简化方案的实现。

21.3 多种命名方案

设计名字空间时，设计者面临的问题很简单：他们必须将众多不相关的命名方案整合在一起，而各个命名方案又各自演进为一个自包含的系统。在某些系统中，文件名指明了该文件所在的存储设备。而另一些系统中，文件用后缀来表明文件的类型（老的系统使用后缀来指明文件的版本）。其他系统将所有的文件映射到平面名字空间中，这里文件名仅仅是一个由字母和数字组成的字符串。本章后面 498
将给出一些系统中文件名的例子，希望能够帮助读者理解名字空间必须适应的名字类型与格式。

21.3.1 MS-DOS

MS-DOS中的名字包含两部分：设备说明和文件名。从句法上讲，MS-DOS名字的格式为X:file，

其中X为单一的字母，指定保存该文件的磁盘设备；file为文件的名字。特别地，字母C代表系统硬盘，C:abc表示的是在硬盘上的文件abc。

21.3.2 UNIX

UNIX将文件组织成一种分层的树形结构目录系统。文件名是针对当前目录的相对路径或者从根目录开始直到文件的完整路径名。

从句法上看，完整路径名由一些被斜杠划分的组件组成，中间的组件表示目录，最后一个组件表示文件。因此，UNIX文件名/homes/xinu/x指的是在子目录xinu下的文件x，而目录xinu是homes的子目录，homes是根目录。根目录本身可以用单独的斜杠（/）表示。注意，前缀/home/xinu/指的是一个目录，而该目录下所有的文件都包含这个前缀。

以后前缀为性的重要性将会变得很明显。现在，只需要记住树形结构与名字前缀相关：

> 当文件名中的组件指定的是树形结构目录的路径时，保存在同一个目录下的所有文件的名字都共享同一个表示该目录的前缀。

21.3.3 V系统

V系统是一种用于研究的操作系统，它允许用户指定上下文和一个名字来命名，系统通过上下文来解析名字。语法上它用括号将上下文括起来。因此，[xix]abc表示在xix上下文中的名为abc的文件。通常，可以把各个上下文当做某个远程服务器上的一系列文件的集合。

21.3.4 IBIS

另一种用于研究的操作系统IBIS为多个机器连接提供另一种语法。在IBIS中，名字的格式为：machine:path。其中machine代表某个特定计算机系统，而path则是该机器上的文件名（例如，UNIX上的完整路径）。

499

21.4 命名系统设计的其他方案

我们需要的是一个能够提供与文件位置、文件所在操作系统等无关的统一命名系统。一般来说，设计者解决这个问题有两个基本方向：设计一个新的文件命名方案或者改进现有的某个命名方案。令人惊奇的是，Xinu名字空间没有使用这两种方案！它通过添加一个语法命名机制来整合众多底层的命名方案，为用户提供命名软件的统一接口。该命名软件将用户提供的名字映射到底层系统中。

通过上面的命名机制来整合众多底层命名方案有以下优点：第一，它允许设计者将现有文件系统和设备整合到一个统一的名字空间中，即使它们是由一系列异构系统中的远程服务器所实现；第二，它允许设计者在不重新编译应用程序的情况下添加新设备或文件系统。从一个极端角度看，选择最简单的命名方案保证所有的文件系统都能处理那些名字，但这也意味着用户不能利用某些服务器提供的复杂服务；从另一个极端看，选择一个包含最复杂情况的命名方案，虽然可以充分利用某些服务器的复杂性，但相对简单的文件系统又可能不能很好地支持它。

21.5 基于句法的名字空间

可以通过考虑名字的句法来理解如何处理这些名字。一个名字仅仅是一串字符。名字空间可以用来对字符串进行转换。对于名字空间来说，既不需要提供文件和路径，也不需要理解每一个底层文件系统的语义。相反，名字空间将用户选用的统一表示的字符串映射到每一个特定的子系统上。例如，名字空间能够将字符串alf转换成字符串C:a_long_file_name。

什么使基于句法的名字空间有如此强大的功能呢？基于句法的方法既自然又灵活，既容易使用，也容易理解，并且能够很好地兼容许多底层的命名方案。用户可以使用一组一致的命名方案，然后使用该命名软件将现有的名字格式转换成底层文件系统所要求的格式。例如，假设本地文件系统使用MS-DOS命名系统，而远程文件系统使用UNIX个路径名字系统。用户可以将所有的名字格式都改成远

种 UNIX 系统所要求的全路径名字语法，将本地磁盘上的名字以/local 开头。在这种方案中，名字/local/abc 指的是本地硬盘系统中的 abc 文件，而名字/etc/passwd 指的是远程的文件。名字空间必须将/local/abc 转换成 C:abc，这样本地 MS-DOS 文件系统才能够识别它，但是该名字空间能够将文件/etc/passwd 传送给远程 UNIX 文件系统而不需要对它进行任何变化。 [503]

21.6 模式和替换

基于句法的名字空间是如何精确运行的呢？一种简便的方法是使用模式（pattern）字符串去指定名字句法，并使用替换（replacement）字符串去指定相应的映射。例如，考虑如下的模式替换对：

```
"/local"  "C:"
```

意思是"将所有出现的/local 字符串都转换成字符串 C:"。

如何构成这种模式呢？由文字串组成的模式无法明确地指定相应的替换字符串。在上面的例子中，模式对于形如/local/x 这样的字符串表现出非常良好的性能，但是对于形如/homes/local/bin 这样的字符串却表现得很差，因为/local 不应该转换为内部子字符串。为此，必须使用更加强有力的模式。UNIX 模式匹配工具引入了说明如何进行匹配的元字符。例如，caret（有时候也称为上箭头）被用来对字符串的开头进行匹配。因此，UNIX 模式为：

```
"^/local"  "C:"
```

指出/local 只能对字符串的开头进行匹配。不幸的是，如果允许任意模式出现，则会使得替换变得非常麻烦，并且模式会变得难以理解。因此，还需要一种更加有效的解决方法。

21.7 前缀模式

现在急需解决的问题是在不增加不必要复杂性的前提下，找到一种允许用户定义子系统名字匹配方法的模式替换策略。在考虑复杂模式前，先考虑使用包含文字串的模式能够做些什么。这个设计的关键是将文件想象成分层组织的，并使用前缀属性来理解为什么模式与前缀相对应。

在每一层中，名字的前缀将文件分成子目录，这就使定义名字和底层文件系统或者设备之间的关系变得更加容易。另外，每个前缀都可以用一个文字串来表示。这里的关键点是：

> 对于前缀的严格名字替换策略意味着使用文字串将底层文件系统分成不同层次的名字空间成为可能。 [501]

21.8 名字空间的实现

下面的一个具体例子将会详细地解释基于句法的名字空间是如何使用模式－替换范式的，并说明名字空间是如何隐藏子系统的细节。在这个例子中，模式包含固定长度的字符串，只有前缀被匹配。下面的小节则讨论其他的实现和普遍应用。

该示例中要实现的名字空间包含了一个叫做 NAMESPACE 的伪设备，程序使用该伪设备打开一个已命名的对象。应用程序调用 NAMESPACE 设备上的打开（open），将名字和模式作为参数传递过去。NAMESPACE 伪设备使用一组前缀模式将现有的名字转换为新的名字，然后通过调用打开命令将新的名字传递给合适的底层设备。我们可以看到所有的文件和设备都可以成为名字空间的一部分，这就意味着一个应用除了需要打开 NAMESPACE 伪设备外，不需要打开其他任何设备。

下面的各节将具体介绍名字空间软件，从介绍基本数据格式声明开始，以介绍 NAMESPACE 伪设备的定义结束。在数据格式声明的后面，介绍了根据前缀模式转换名字的两个函数。这些函数构成该名字空间软件最重要部分的基础：这些函数实现对 NAMESPACE 伪设备的打开。

21.9 名字空间的数据结构和常量

文件 name.h 包含了 Xinu 名字空间中使用的数据结构的声明和常量的定义。

```
/* name.h */

/* Constants that define the namespace mapping table sizes */

#define NM_PRELEN       64              /* max size of a prefix string  */
#define NM_REPLLEN      65              /* maximum size of a replacement*/
#define NM_MAXLEN       256             /* maximum size of a file name  */
#define NNAMES          40              /* number of prefix definitions */

/* Definition of the name prefix table that defines all name mappings */

struct  nmentry {                       /* definition of prefix table   */
        char    nprefix[NM_PRELEN];     /* null terminated prefix       */
        char    nreplace[NM_REPLLEN];   /* null-terminated replacement  */
        dev32   ndevice;                /* device descriptor for prefix */
};
extern  struct  nmentry nametab[];      /* table of name mappings       */
extern  int32   nnames;                 /* num. of entries allocated    */
```

其主要的数据结构是数组 nametab，最大能够容纳 NNAEMES 项。每项包含一个前缀模式字符串、一个替换字符串和一个设备 ID。外部的整型变量 nnames 定义了 nametab 数组中有效表项的个数。

21.10 增加名字空间前缀表的映射

函数 mount 用来添加前缀表的映射。下列所展望的一样，mount 有 3 个参数：一个前缀字符串、一个替换字符串和一个设备 ID。相应的代码在文件 mount.c 中。

```
/* mount.c - mount, namlen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mount  -  add a prefix mapping to the name space
 *------------------------------------------------------------------------
 */
syscall mount(
          char          *prefix,        /* prefix to add                */
          char          *replace,       /* replacement string           */
          dev32         device          /* device ID to use             */
)
{
        intmask mask;                   /* saved interrupt mask         */
        struct  nmentry *namptr;        /* pointer to unused table entry*/
        int32   psiz, rsiz;             /* sizes of prefix & replacement*/
        int32   i;                      /* counter for copy loop        */

        mask = disable();

        psiz = namlen(prefix, NM_PRELEN);
        rsiz = namlen(replace, NM_REPLLEN);
        if ((psiz == SYSERR) || (rsiz == SYSERR) || isbaddev(device)) {
                restore(mask);
                return SYSERR;
        }
        if (nnames >= NNAMES) {         /* If table full return error */
                restore(mask);
                return SYSERR;
        }
```

```
        /* Allocate a slot in the table */

        nameptr = &namecache[nnames];       /* next unused entry in table   */

        /* Copy prefix and replacement strings and record device ID */

        for (i=0; i<pfxlen; i++) {          /* copy prefix into table entry */
                nameptr->nprefix[i] = *prefix++;
        }

        for (i=0; i<rsiz; i++) {            /* copy replacement into entry  */
                nameptr->nrepl[i] = *replace++;
        }

        nameptr->ndevice = device;          /* record the device ID         */

        nnames++;                           /* increment number of names    */

        restore(mask);
        return OK;
}


/*------------------------------------------------------------------------
 *  nwrlen  -  compute the length of a string stopping at maxlen
 *------------------------------------------------------------------------
 */
int32   nwrlen(
          char          *name,              /* name to use                  */
          int32         maxlen              /* maximum length (including a  */
                                            /*   NULL byte)                 */
)
{
        int32   i;                          /* counter */

        /* Search until a null terminator or length reaches max */

        for (i=0; i < maxlen; i++) {
                if (*name++ == NULLCH) {
                        return i+1;
                }
        }
        return SYSERR;
}
```

如果任何一个参数是无效的或名表已经满了，则 mount 返回 SYSERR；否则，它将增加 nnames 的数量，以便在表中分配一个新表项，并将相应的值填入其中。

21.11　使用前缀表进行名字映射

一旦创建了前缀表，就可以进行名字转换了。映射包括找到一个前缀匹配并将合适的替换字符串代入。函数 nammap 负责执行转换步骤。文件 nammap.c 包含了相应的代码。

```
/* nammap.c - nammap, namrepl, namcpy */

#include <xinu.h>

status  namcpy(char *, char *, int32);
did32   namrepl(char *, char[]);
```

```
/*------------------------------------------------------------------------
 *  nammap  -  using namespace, map name to new name and new device
 *------------------------------------------------------------------------
 */
devcall nammap(
        char    *name,                  /* a name to map                */
        char    newname[NM_MAXLEN],     /* buffer for mapped name       */
        did32   namdev                  /* ID of the namespace device   */
        )
{
        did32   newdev;                 /* device descriptor to return  */
        char    tmpname[NM_MAXLEN];     /* temporary buffer for name    */
        int32   iter;                   /* number of iterations         */

        /* Place original name in temporary buffer and null terminate */

        if (nampy(tmpname, name, NM_MAXLEN) == SYSERR) {
                return SYSERR;
        }
        /* Repeatedly substitute the name prefix until a non-namespace  */
        /* device is reached or an iteration limit is exceeded          */

        for (iter=0; iter<nnames ; iter++) {
                newdev = namrepl(tmpname, newname);
                if (newdev != namdev) {
                        nampy(tmpname, newname, NM_MAXLEN);
                        return newdev;  /* either valid ID or SYSERR    */
                }
        }
        return SYSERR;
}

/*------------------------------------------------------------------------
 *  namrepl  -  use the name table to perform prefix substitution
 *------------------------------------------------------------------------
 */
did32   namrepl(
        char    *name,                  /* original name                */
        char    newname[NM_MAXLEN]      /* buffer for mapped name       */
        )
{
        int32   i;                      /* iterate through name table   */
        char    *pptr;                  /* walks through a prefix       */
        char    *rptr;                  /* walks through a replacement  */
        char    *optr;                  /* walks through original name  */
        char    *nptr;                  /* walks through new name       */
        char    olen;                   /* length of original name      */
                                        /*  including the NULL byte     */
        int32   plen;                   /* length of a prefix string    */
                                        /*  "not" including NULL byte   */
        int32   rlen;                   /* length of replacement string */
        int32   remain;                 /* bytes in name beyond prefix  */
        struct  nmentry *namptr;        /* pointer to a table entry     */

        /* Search name table for first prefix that matches */
```

```
        for (i=0; i<nnames; i++) {
                namptr = &nameTab[i];
                cptr = name;            /* start at beginning of name   */
                pptr = namptr->nprefix; /* start at beginning of prefix */
            /* Compare prefix to string and count prefix size */

            for (plen=0; *pptr != NULLCH ; plen++) {
                    if (*pptr != *cptr) {
                            break;
                    }
                    pptr++;
                    cptr++;
            }
            if (*pptr != NULLCH) {  /* prefix does not match */
                    continue;
            }

            /* found a match - check that replacement string plus   */
            /* bytes remaining at the end of the original name will */
            /* fit into new name buffer.  Ignore null on replacement*/
            /* string, but keep null on remainder of name.          */

            clen = nmlen(oname ,NM_MAXLEN);
            rlen = nmlen(namptr->nreplace,NM_MAXLEN) - 1;
            remain = clen - plen;
            if ( (rlen + remain) > NM_MAXLEN)  {
                    return (did32)SYSERR;
            }

            /* Place replacement string followed by remainder of */
            /* original name (and null) into the new name buffer */


            cptr = newname;
            rptr = namptr->nreplace;
            for (; rlen>0 ; rlen--) {
                    *cptr++ = *rptr++;
            }
            for (; remain>0 ; remain--) {
                    *cptr++ = *cptr++;
            }
            return namptr->ndev(0);
        }
        return (did32)SYSERR;
}
/*------------------------------------------------------------------------
 *  nmcopy  -  copy a name from one buffer to another, checking length
 *------------------------------------------------------------------------
 */
status  nmcopy(
         char           *newname,       /* buffer to hold copy          */
         char           *oldname,       /* buffer containing name       */
         int32          buflen          /* size of buffer for copy      */
        )
{
        char    *nptr;                  /* point to new name            */
        char    *optr;                  /* point to old name            */
        int32   cnt;                    /* count of characters copied   */
```

```
        nptr = newname;
        optr = oldname;

        for (cnt=0; cnt<buflen; cnt++) {
                if ( (*nptr++ = *optr++) == NULL) {
                        return OK;
                }
        }
        return SYSERR;          /* buffer filled before copy completed  */
}
```

nammap 最有趣的地方是它支持多重映射。特别是，因为名字空间是一个伪设备，所以对于用户来说，可以指定一个映射重新映射到 NAMESPACE 设备。比如，考虑下面两个在 nametab 中的项：

```
                        "/local/"       ""              LFILESYS
                        "LFS:"          "/local/"       NAMESPACE
```

第一个项说明，如果名字以/local/开头，那么就去除前缀并将名字传递到本地文件系统。第二个项说明，LFS:是/local/的缩写。也就是说，前缀 LFS: 被/local/所替换，同时将结果字符串传递回 NAMESPACE 设备以进行下一轮的映射。

当然，递归映射可能会有危险。考虑如果用户将下面的内容添加到名字空间会发生什么：

```
                        "/x"    "/x"            NAMESPACE
```

当出现名字/xyz，一次简单的操作就会找到前缀/x，进行替换，并在 NAMESPACE 设备上调用打开（open）函数，这样就会引起递归的死循环。为了避免这个问题，通过 NAMESPACE 进行的替换迭代限制整个迭代次数。特别地，只允许 nametab 中的每一个前缀迭代一次（也就是，每一个前缀只能被替换一次）。当然，nammap 也限制了名字的长度：如果替换将名字扩展到超过 NM_MAXLEN 字符串的长度，nammap 将停止并返回 SYSERR。

502 - 508

nammap 一开始将原始名字复制到本地数组 unpname 中。然后，它进行迭代，直到名字被映射到除了 NAMESPACE 以外的一个设备，或者到达迭代的限制。在每次迭代的期间，nammap 调用函数 namrepl 来查看目前的名字，并形成一个替换。

函数 namrepl 实现了一个基本的替换策略（replacement policy）。示例中的替换策略经过了简化：namrepl 线性地搜索表。每次搜索总是始于表中第一项，一旦表中的一个前缀与参数 name 表示的字符串相匹配时，搜索就停止。一旦搜索停止，nammap 将原始名字中的未匹配部分添加到替换字符串后面，从而形成一个映射名字，并将它赋给参数 newname。然后返回这个表项的设备 ID。后面的一节将解释这个设计对用户的影响。

21.12 打开命名文件

一旦 nammap 可用，为名字空间伪设备构造上半部的打开（open）操作将变得极为简单。打开操作的基本目标是定义一个名字空间伪设备，NAMESPACE，使得打开这个设备的操作让系统打开合适的底层设备。一旦名字被映射，并且新的设备被定义，则 namopen 只要调用打开（open）函数即可。这段代码包含在文件 namopen.c 中。

509

```
/* namopen.c - namopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  namopen  -  open a file or device based on the name
 *------------------------------------------------------------------------
 */
devcall namopen(
        struct dentry *devptr,          /* entry in device switch table */
        char  *name,                    /* name to open                 */
        char  *mode                     /* mode argument                */
        )
```

```
{
        char    newname[NM_MAXLEN];     /* name with prefix replaced    */
        did32   newdev;                 /* device ID after mapping      */

        /* Use namespace to map name to a new name and new descriptor */

        newdev = nammap(name, newname, devptr->dvnum);

        if (newdev == SYSERR) {
                return SYSERR;
        }

        /* Open underlying device and return status */

        return  open(newdev, newname, mode);
}
```

21.13 名字空间初始化

前缀表如何初始化呢？有两个可能的方法：一是要求用户填写名字空间表的表项，二是为这个表提供初始化的项。因为名字空间已经被设计为一个伪设备，系统启动时调用函数 init，即可完成初始化。

决定如何初始化前缀表可能有点儿困难。因此，我们检查初始化函数看它如何构造前缀表，并且将实际前缀的讨论推迟到 21.14 节。文件 naminit.c 包含 naminit 函数的代码。 [510]

```
/* naminit.c - naminit */

#include <xinu.h>

#ifndef RFILESYS
#define RFILESYS        SYSERR
#endif

#ifndef FILESYS
#define FILESYS         SYSERR
#endif

#ifndef LFILESYS
#define LFILESYS        SYSERR
#endif

struct  nmentry nametab[NNAMES];        /* table of name mappings       */
int32   nnames;                         /* num. of entries allocated    */

/*------------------------------------------------------------------------
 *  naminit  -  initialize the syntactic namespace
 *------------------------------------------------------------------------
 */
status  naminit(void)
{
        did32   i;                      /* index into devtab            */
        struct  dentry *devptr;         /* ptr to device table entry    */
        char    tmpstr[NM_MAXLEN];      /* string to hold a name        */
        status  retval;                 /* return value                 */
        char    *tptr;                  /* ptr into tempstring          */
        char    *rptr;                  /* ptr to device name           */
        char    devprefix[] = "/dev/";  /* prefix to use for devices    */
        int32   len;                    /* length of created name       */
        char    ch;                     /* storage for a character      */

        /* Set prefix table to empty */
```

```
        nnames = 0;

        for (i=0; i<NDEVS ; i++) {
                tptr = tmpstr;
                nptr = devprefix;
        /* Copy prefix into tmpstr*/

        len = 0;
        while ((*tptr++ = *nptr++) != NULLCH) {
                len++;
        }
        tptr--; /* move pointer to position before NULLCH */
        devptr = &devtab[i];
        nptr = devptr->dvname;  /* move to device name */

        /* Map device name to lower case and append */

        while(++len < NM_MAXLEN) {
                ch = *nptr++;
                if ( (ch >= 'A') && (ch <= 'Z') ) {
                        ch += 'a' - 'A';
                }
                if ( (*tptr++ = ch) == NULLCH) {
                        break;
                }
        }

        if (len > NM_MAXLEN) {
                kprintf("namespace: device name %s too long\r\n",
                                devptr->dvname);
                continue;
        }

        retval = mount(tmpstr, NULLSTR, devptr->dvnum);
        if (retval == SYSERR) {
                kprintf("namespace: cannot mount device %d\r\n",
                                devptr->dvnum);
                continue;
        }
}

/* Add other prefixes (longest prefix first) */

mount("/dev/null",      "",         CONSOLE);
mount("/remote/",       "remote:",  RFILESYS);
mount("/local/",        NULLSTR,    LFILESYS);
mount("/dev/",          NULLSTR,    SYSERR);
mount("~/",             NULLSTR,    LFILESYS);
mount("/",              "root:",    RFILESYS);
mount("",               "",         LFILESYS);
        return OK;
}
```

这里忽略指定的前缀和替换名字，仅关注直接的初始化是如何工作的。在将有效表项的数目设置为0之后，naminit调用mount向前缀表中增加表项，其中每个表项包含一个前缀模式、替换字符串和设备标识符。for循环遍历设备转换表。对每个设备，它创建一个形式为/dev/xxx的名字，其中xxx是映射到小写字母的设备名。因此，它为/dev/console创建一个表项，将其映射到CONSOLE设备。因此，如果运程调用：

```
d = open(NAMESPACE,"/dev/console","rw");
```

名字空间将会在 CONSOLE 设备上调用 open 函数，并返回结果。

21.14 对前缀表中的项进行排序

Xinu 名字替换策略会影响用户。要理解如何影响用户，首先要指出的是 namrepl 函数使用顺序查询。因此，用户加载名字的方式必须使顺序查询产生预期的结果。特别地，我们的实现不禁止重叠的前缀，当出现重叠的前缀时也不会对用户发出警告。因此，如果重叠的前缀出现了，用户就必须确保在表中最长前缀出现在较短前缀的前面。例如，如果表中包含两项，如表 21-1 所示，考虑会发生什么情况。

前缀	替换	设备
"x"	""（空字符串）	LFILESYS
"xyz"	"/"（斜线符）	RFILESYS

表 21-1 前缀表中的两项，它们的顺序必须交换，否则第二项永远不会被使用

第一项映射前缀 x 到本地文件系统，第二项映射前缀 xyz 到远程文件系统。不幸的是，因为 namrepl 顺序搜索表，所以任何以 x 开头的文件名将匹配第一项，并被映射到本地文件系统。第二项将永远不会被使用。然而，如果交换这两项的位置，以 xyz 开头的文件名就会被映射到远程文件系统，而其他以 x 开头的文件名被映射到本地文件系统。于是可以得出结论：

511 ~ 513

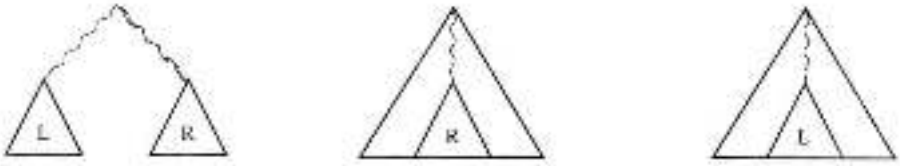
> 因为程序顺序搜索前缀表，并且不检测重叠的前缀，所以用户必须按照长度反向地装入前缀，以确保系统能够首先匹配最长的前缀。

21.15 选择一个逻辑名字空间

将名字空间仅仅看做一个用来缩写长名字的机制的想法很诱人。然而，仅仅专注于机制可能会产生误导。选择有意义前缀名的关键在于构建一个文件存放的层次结构。事实上，名字空间设计定义了层次结构的组织方式。

> 我们认为所有的名字需要组织在层次结构中，而不是将名字空间仅仅看做缩写名字的机制。名字空间中的表项引发实现需要的层次。

请读者花一些时间想象一个系统，它能够获取本地磁盘和远程服务器的文件。不要思考如何缩写特定的文件名，而要思考如何组织文件。图 21-2 显示了二种可能的结构。



a）本地文件和远程文件在同一层　b）远程文件在本地文件的子目录中　c）本地文件作为远程文件的子目录

图 21-2 三种可能的本地文件和远程文件的层次结构

如图 21-2 所示，本地文件和远程文件可以相同地存放，但是在不同的层次结构中：或者，本地文件系统可以形成层次的主要部分，远程文件在子层次；或者远程文件构成主层次，而本地文件作为子层次。对于这三种选择，两种文件系统的大小和存取的频率可能有助于决定优先采用哪种结构。例如，如果远程文件系统有数千个文件，而本地文件系统仅有 10 个文件，那么自然想到使用远程文件作为主层次，将本地文件移到子层次。

514

21.16 默认层次和空前缀

Xinu 名字空间的软件设计可以容易地支持图 21-2 中的任何层次。特别地，挂载（mount）操作允

许用户选择一个子系统作为默认，系统将根据这个层次来组织剩余文件。

一个子系统如何成为默认？首先，子系统的前缀必须能够匹配所有不能被其他表项匹配的名字。而空前缀将为示例名字空间提供有保证的匹配。其次，带有空前缀的默认项必须在所有其他前缀被检测后再被检查。因为 rammap 顺序搜索前缀表，所以默认项必须放在表的最后。如果任何其他项匹配，那么 nameval 就按照这个匹配进行替换。

请再次查看 naminit，看看本地文件系统如何成为默认的。对 mount 的最终调用将向默认映射中插入一个空前缀。因此，任何不匹配其他前缀的名字将适用于本地文件系统。

21.17 额外的对象操作函数

尽管看起来是将所有名字组织在一个唯一的、统一的层次结构中，但前面介绍的名字空间并没有提供需要的所有功能。为了理解其中的原因，可以看一段仅处理打开的已命名的对象的代码。事实上其他可能的对已命名对象的操作还包括：

- 检测对象的存在性
- 改变对象的名字
- 删除对象

检测对象的存在性　通常，软件需要检测一个对象的存在性，而不影响对象。看起来下面的代码可以用来检测一个对象是否存在。

```
dev = open(NAMESPACE, "object", "r")
if (dev == SYSERR) {
        ...object does not exist
} else {
        close(dev);
        ...object exists
}
```

515

但不幸的是，对 open 的调用会产生副作用。例如，打开一个网络接口设备可能引起系统声明这个接口可用于数据包传输。因此，打开和关闭设备会引起数据包传输，即使进程已经明确地使这个接口不可用。为了避免副作用，需要使用额外的功能。

改变对象的名字　大部分文件系统允许用户重命名文件。然而，当使用名字空间时，这将出现两个问题。第一，因为用户通过名字空间查看所有的文件，所以对重命名文件的要求是含糊不清的：应该改变底层文件的名字，还是应该改变名字空间中的映射？尽管系统可以使用一个逃避机制来让用户区别抽象名字和底层系统使用的名字，但这样做会有危险，因为下层名字的改变可能会导致文件不再能够通过名字空间的映射。第二，如果用户将名字 α 改变为 β，可能的结果是字符串 β 映射到一个本地文件系统，α 映射到一个远程文件系统。因此，尽管用户看到的是一个统一的层次，但重命名操作可能不被允许（或者仅涉及文件复制）。

删除对象　上面给出的理由同样适用于对象删除。也就是说，因为用户通过名字空间看到所有的名字，所以删除对象的要求必须通过名字空间对名字进行映射以决定合适的底层文件系统。

删除对象、重命名对象和检测对象的存在性应该如何实现？有三种可能的方法：为每个操作创建单独的函数、扩展设备转换表，或者为函数 control 增加额外的操作。第一种方法（单独的函数），将名字作为参数，使用名字空间来将名字映射到底层设备，然后在该设备上调用合适的操作。第二种方法扩展设备转换表来增加额外的高层函数，例如删除、重命名和存在性检测函数。也就是说，除了打开（open）、读（read）、写（write）和关闭（close）操作外，增加新的操作来实现额外的功能。第三种方法将功能增加到 control 函数中。例如可以指定如果子系统实现对象删除，那么实现 control 的驱动函数必须响应一个 DELETE 请求。Xinu 使用第一种和第三种方法的混合。本章练习要求读者考虑扩展设备转换表的优缺点。

21.18 名字空间方法的优点和限制

句法名字空间将程序与底层设备和文件系统隔离开，允许对命名的层次结构进行设计或修改，而

不改变底层系统。为了显示名字空间的能力，考虑一个系统，它需要在本地磁盘保存临时文件，并使用前缀/tmp/与其他文件区分开来。将临时文件移动至远程文件系统包括改变指明如何操作前缀/tmp/ 516
的名字空间项。因为当程序引用文件时，它们总是要使用名字空间，所以所有程序不必改变源代码就可以继续正确地操作。

名字空间允许重新组织命名层次结构，而不需要重新编译使用它的程序。

仅使用前缀模式的名字空间软件不能处理所有的层次结构或者文件映射。例如，在某些 UNIX 系统中，名字/dev/tty 指的是进程的控制终端，服务器不需要使用它。名字空间可以通过将前缀/dev/tty 映射到设备标识符 SYSEDB 来阻止意外的访问。不幸的是，这样的映射也阻止了客户访问其他需要共享相同前缀的其他项（例如，/dev/tty1）。

当分隔符出现在名字的中间时，使用固定字符串作为前缀模式也可以防止名字空间改变分隔符。例如，假设一台计算机有两个底层文件系统，一个遵循 UNIX 规范，使用斜杠来分隔路径的各部分，而另外一个文件系统使用反斜杠来分隔各部分。因为名字空间仅仅处理前缀模式，所以它就不能将斜杠映射为反斜杠，反之亦然，除非所有可能的前缀都存储在名字空间中。

21.19　广义模式

名字空间的很多限制可以通过使用本章开头描述的更广义的模式来克服。例如，如果可以指定一个完全字符串匹配，而不仅仅是前缀匹配，那么区分名字/dev/tty 和/dev/tty1 的问题就可以解决。完全匹配和前缀匹配可以组合使用；可以给 mount 指定一个额外的参数，指明匹配的类型和可以存储在表项中的值。

广义模式不仅仅允许采用固定字符串来解决额外问题，还可以在模式本身中保存所有的匹配信息。例如，假设下列字符在模式中有如图 21-3 所示的特殊意义。[⊖]

字符	意义
↑	匹配字符串的开头
$	匹配字符串的结尾
.	匹配单个字符
*	模式重复0到多次
\	将模式中的下一个字符按照字面上的意义进行解释
其他	作为固定字符串自匹配

图 21-3　广义模式的一个示例定义

517

因此，像↑/dev/tty$ 这样的模式指定字符串/dev/tty 的全匹配，而像\$ 这样的模式匹配可以嵌入在字符串中的美元符号。

为了使广义模式匹配在名字空间更有用，需要两个额外的规则。第一，假定使用最左边的可能匹配。第二，假定在所有最左边的匹配中，选择最长的匹配。本章练习建议如何使用这些广义模式来匹配那些固定前缀所不能处理的名字。

21.20　观点

名字的语法已经获得了广泛的研究和讨论。从前，每个操作系统有自己的命名方案，涌现出各种命名方案。然而，在层次目录系统变得流行后，大部分操作系统都采用了层次命名方案，仅仅在一些

⊖ 这里给出的匹配模式对应于 UNIX sed 命令使用的模式。

小细节上有所不同。例如不同组成之间的分隔是采用斜杠还是反斜杠。

正如前文的设计所指出的那样，命名从概念上与底层文件和I/O系统分离开，并允许设计者将一个统一的名字空间加到所有的底层设备上。然而，句法方法有优点也有缺点。主要的问题出在语义上：尽管它提供一致的外表，但名字空间引入了模糊和混淆的语义。例如，如果一个对象被重命名，那么应该修改名字空间还是改变底层对象的名字？如果名字空间将两个前缀映射到同样的底层文件系统，那么使用不同前缀的应用程序可能因为疏漏而访问了相同的对象。如果将两个名字映射到不同的底层文件系统，那么涉及名字的操作（例如 move 操作）可能不会像预期那样地工作。即使像 delete 这样的操作也可能有预料之外的语义（例如，删除一个本地对象可能把它移动到回收站，而删除一个远程对象则会永远移除这个对象）。

21.21 总结

处理文件名是困难的，尤其是当操作系统支持多种底层命名方案时。解决命名问题的一个方法是在应用程序和底层文件系统之间增加一层名字空间软件。名字空间本身不实现文件本身，而仅仅把名字当做字符串，根据映射表中的信息，将名字映射为适合底层系统的形式。

本章查看了一个句法名字空间的实现，它使用模式替换方案，其中模式是表示名字前缀的固定字符串。这个软件包括函数 mount 和 unmount[⊖]来处理映射表，函数 nammap 将名字映射为目标形式。当
518 打开一个文件时，我们的示例名字空间包含一个由用户指定的 NAMESPACE 伪设备。NAMESPACE 伪设备映射指定的文件名，然后打开指定的文件。

名字空间软件是优雅且强大的。只需要几个函数和简化的前缀匹配概念就可以容纳很多命名方案。特别地，它适应于远程文件系统、本地文件系统和一组设备。然而，简化版本不能处理所有可能的映射。为了提供更复杂的命名系统，模式的概念必须广义化。一个可能的广义化是给模式中的某些字符赋予特殊的意义。

练习

21.1 用户应该同时拥有对 nammap 和 namrepl 的访问权限吗？为什么？

21.2 修改 mount 使它拒绝设置可能引起无限循环的前缀替换对吗？为什么？

21.3 可能引起 nammap 超过最大字符串长度的前缀替换对的最小数目是多少？

21.4 用一个包含两个处理调用的语句替换 namopen 函数的主体，使 namopen 的代码最最小。

21.5 为 NAMESPACE 伪设备实现一个上半部分 control 函数，并使 nammap 成为一个控制函数。

21.6 实现广义化的模式匹配。参考 UNIX sed 命令，设计额外的方法来定义模式匹配字符。

21.7 建立一个具有前缀匹配又有全字符串匹配的名字空间。

21.8 假设一个名字空间，它使用固定字符串模式，除了目前的前缀匹配外，也允许全字符串匹配。有没有这种情况存在，拥有一个与前缀模式相同的全字符串模式是有意义的？解释原因。

519 21.9 除了重命名、删除和存在性检测外，还需要哪些文件操作原语？

⊖ 函数 unmount 从名字空间移除一个前缀。本章没有说明函数 unmount。

系统初始化

只有避免事情的开端，才能逃避事情的结局。

——Cyril Connolly

22.1 引言

初始化是设计过程的最后一环。设计人员总是从运行状态的视角来进行系统的设计工作，并将如何启动的设计放在后面进行。过早地考虑系统初始化与过早地考虑系统优化一样，都有一些不良影响：这往往会对设计施加不必要的限制，并且会将设计者的注意力从重要问题转移到一些细枝末节的问题上。

本章介绍初始化系统所需要的步骤，并描述初始化代码如何将顺序运行的程序转换为支持并发处理。我们将看到其中并没有特殊硬件的参与，并发只是操作系统创建的一种抽象。

22.2 引导程序：从头开始

初始化的讨论从系统终止开始。每个使用过计算机的人都知道，软件或硬件上的错误都会造成灾难性的失效，俗称为"死机"（crashes）。当计算机硬件因为错误的代码和数据进行非法操作的时候，死机就会发生。当死机发生时，所有内存中的数据将丢失，操作系统必须重启，这通常会花费相当多的时间。

怎么才能让一台没有操作系统代码的计算机启动并开始运行呢？答案是不行的。代码是计算机启动的必要条件。在老式计算机中，重启是一个折磨人的过程，需要操作员通过面板上的开关输入初始化程序。后来开关换成了键盘，然后是磁带、磁盘这些 I/O 设备，最后是现在的只读内存（ROM）。

有些嵌入式设备将整个操作系统而不仅仅是初始化代码存储在 ROM 中。这就意味着这些设备可以在通电之后立即开始工作（更换了电池或者打开电源之后）。然而，大部分的计算机需要多个步骤来启动。通电之后，硬件运行存储在 ROM 中的初始化启动程序。虽然可能包括调试硬件的机制，但初始化启动程序通常还是很小的——其主要功能是装载和运行大型程序。在标准的个人计算机中，启动程序初始化设备（如显示器、键盘、磁盘等），将操作系统映像复制到内存中，然后跳转到操作系统的入口。

将计算机系统链接到网络可能还需要其他的步骤，初始化启动程序可能会加载一个中间程序来初始化网络接口，然后通过网络从远程服务器下载操作系统映像。

装载更大程序的顺序程序通常被称为引导程序，而整个系统称为引导系统[⊖]。更传统的名字为初始化程序装载（IPL）或者冷启动（cold start）。

22.3 操作系统初始化

当然，当 CPU 开始执行操作系统时，初始化的工作并没有结束。操作系统还必须完以下的任务：

- 初始化内存管理硬件和空闲内存列表。
- 初始化各个操作系统模块。
- 装载（如果没有）并初始化设备驱动程序。
- 启动（或者重置）I/O 设备。
- 从顺序执行转换成并行执行。

⊖ bootstrap 一词来源于短语 "pulling one's self up by one's bootstraps"（通过拽住靴子把人提起来），这个短语描述为是一个不可能的任务。

- 创建一个空进程。
- 创建一个执行用户代码的进程（如桌面）。

基础初始化结束之后，最重要的步骤是：操作系统必须将自己从一个顺序执行的程序转变为一个支持并行操作的程序。在22.4节中，我们将了解E2100L启动的情况，理解Xinu是如何装载到硬件上的，回顾Xinu启动之后依次执行的步骤，以及转换是如何发生的。

522

22.4 在E2100L上启动一个可选的映像

E2100L作为无线路由器使用的过程在前面的章节中已经描述过了。在通电之后，路由器执行存储在ROM中的初始化程序。在有了初始化的设备后，启动程序运行嵌入式Linux系统的一个版本（Linux代码也存储在ROM中）。启动之后，Linux运行将系统转换为无线路由器的应用程序。例如，一个应用程序通过IP、UDP或者DHCP等标准网络协议与网络服务提供商（ISP）协商网络连接，另一个应用程序接收通过无线或者有线网络传来的本地计算机的路由请求。一旦Linux开始运行，所有的输入和输出都由内置的Linux进程控制，该设备只能作为路由器使用。没有任何办法替换或者修改软件。

如果E2100L用于运行内置的操作系统和应用程序，那它怎么才能启动Xinu呢？幸运的是，设计者提供了中断初始启动流程并在Linux启动前获得控制权限的办法。用户在启动过程中键入特定字符，可以使初始化启动程序在载入Linux之前停止，并且开始从控制台接受命令。系统会显示如下提示：

```
ar7100>
```

然后等待用户输入指令。在指令执行完后，系统又会继续出现提示并等待下一条指令。

启动程序提供了很多下载映像的命令。例如，loadb命令可以用来使用kermit协议从控制台串口线上下载二进制映像。而且，E2100L还支持bootp命令，它能通过BOOTP和TFTP协议从网络中下载映像。bootp命令接受一个参数来指定映像的具体位置。Xinu系统将通过如下命令下载到位于0x81000000的内核空间中：

```
bootp 0x81000000
```

对E2100L命令发送一个BOOTP请求数据包。网络中必须运行一个已配置的BOOTP服务进程用于响应这些请求。在响应的数据中，必须包括下载文件名和存储这个文件的服务器地址。E2100L发送一个TFTP请求数据包序列来获取文件。网络中必须也运行一个已配置的TFTP服务进程来响应每个请求包。

一旦将文件下载到了内存中，启动程序在命令行上再次发出提示，并等待用户输入命令。如果用户输入命令：

523

```
bootm
```

启动程序就跳转到地址为0x81000000的位置（即开始运行下载到内存中的Xinu代码）。

22.5 Xinu初始化

如果C语言的运行时环境已经建立，在完成了操作系统各模块的初始化并启动了系统中断之后，Xinu才能正式成为操作系统。E2100L上的初始化程序只进行了低级别的硬件初始化，如系统总线的初始化。启动程序还需要对控制台串行设备进行初始化（如设置波特率），使Xinu代码可以通过控制台线缆发送与接收字符。

尽管在Xinu引导之前某些低级别的初始化工作已经完成，但仍然需要汇编语言函数来执行其他相关的初始化任务。在我们的代码中，执行从标签_start开始，代码可以在start.S中找到。

```
/*  start.S  _start, nanzero */

/*************************************************************************/
/*                                                                       */
/*   External symbol start (_start in assembly language) gives the       */
/* location where execution begins after the bootstrap loader has        */
/* placed a Xinu image in memory and is ready to execute the image.      */
/*                                                                       */
/*   After initializing the hardware and establishing a run time         */
/* environment suitable for C (including a valid stack pointer), the     */
/* code jumps to the C function nulluser.                                */
```

```
/*****************************************************************************/

#include <interrupt.h>
#include <mips.h>

#define NULLSTK      8192       /* Safe size for NULLSTK */

.extern flash_size

.text
        .align  4
        .globl  _minheap
        .globl  _start
/*-----------------------------------------------------------------------------
 *
 * _start   - set up interrupts, initialize the stack pointer, clear the
 *            null process stack, zero the BSS (uninitialized data)
 *            segment, and invoke nulluser
 *-----------------------------------------------------------------------------
 */
        .ent _start
 start:

        /* Pick up flash size from a3 (where the boot loader leaves it) */

        sw      a3, flash_size

        /* Clear Xinu-defined trap and interrupt vectors */

        la      a0, IRQ_ADDR
        la      a1, IRQVEC_END
        jal     memzero

        /* Copy low-level interrupt dispatcher to reserved location.    */

        la      a0, IRQ_ADDR            /* Reserved vector location     */
        la      a1, intdispatch         /* Start of dispatch code       */
        lw      v0, 0(a1)
        sw      v0, 0(a0)               /* Store jump opcode            */

        /* Clear interrupt related registers in the coprocessor */

        mtc0    zero, CP0_STATUS        /* Clear interrupt masks        */
        mtc0    zero, CP0_CAUSE         /* Clear interrupt cause reg.   */

        /* Clear and invalidate the L1 instruction and data caches */

        jal     flushcache

        /* Set up Stack segment (see function summary) */

        li      s0, NULLSTK             /* Stack is NULLSTK bytes       */
        la      a0, _end
        addu    s0, s0, a0              /* Top of stack = _end+NULLSTK  */

        /* Word align the top of the stack */
        subu    s1, s0, 1
        srl     s1, 4
        sll     s1, 4
```

```
        /* Initialize the stack and frame pointers */

        move    sp, s1
        move    fp, s1

        /* Zero out 64KB space below new stack pointer */

        move    a1, s0          /* note: s0 still points to _end */
        jal     memzero

        /* Clear the BSS segment */

        la      s0, _bss
        la      a1, _end
        jal     memzero

        /* Store bottom of the heap */

        la      t0, minheap
        sw      s0, 0(t0)

        j       nulluser        /* jump to the null process code */
        .end    _start

/*------------------------------------------------------------------------
 * memzero - clear a specified area of memory
 *
 *      args are: starting address and ending address
 *------------------------------------------------------------------------
 */
        .ent    memzero
memzero:
        sw      zero, 0(s0)
        addiu   s0, s0, 4
        bltu    s0, a1, memzero
        jr      ra
        .end    memzero
```

22.6 系统启动

当处理器跳转到C语言函数nulluser之后，运行的是程序而不是操作系统。该程序初始化所有重要的操作系统数据结构、设备、信号量和进程。读者可以在initialize.c中找到相应的代码。所有的精华都在这里。前文所提到的从程序到系统的重大转变也发生在这里。

```
/* initialize.c - nulluser, sysinit */

/* Handle system initialization and become the null process */

#include <xinu.h>
#include <string.h>

extern  void    _start(void);   /* start of Xinu code */
extern  void    *_end;          /* end of Xinu code */

/* Function prototypes */

extern  void main(void);        /* main is the first process created    */
extern  void xdone(void);       /* system "shutdown" procedure          */
```

```
static  void sysinit(void);     /* initializes system structures        */

/* Declarations of major kernel variables */

struct  procent proctab[NPROC]; /* Process table                        */
struct  sentry  semtab[NSEM];   /* Semaphore table                      */
struct  memblk  memlist;        /* List of free memory blocks           */

/* Active system status */

int     prcount;                /* Total number of live processes       */
pid32   currpid;                /* ID of currently executing process    */

/* Memory bounds set by startup.S */

void    *minheap;               /* start of heap                        */
void    *maxheap;               /* highest valid memory address         */

/*------------------------------------------------------------------------
 * nulluser - initialize the system and become the null process
 *
 * Note: execution begins here after the C run-time environment has been
 * established.  Interrupts are initially DISABLED, and must eventually
 * be enabled explicitly.  The code turns itself into the null process
 * after initialization.  Because it must always remain ready to execute,
 * the null process cannot execute code that might cause it to be
 * suspended, wait for a semaphore, put to sleep, or exit.  In
 * particular, the code must not perform I/O except for polled versions
 * such as kprintf.
 *------------------------------------------------------------------------
 */

void    nulluser(void)
{
        kprintf("\n\r%s\n\n\r", VERSION);

        sysinit();

        /* Output Xinu memory layout */

        kprintf("%10d bytes physical memory.\r\n",
                (uint32)maxheap - (uint32)addressp2k(0));
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)addressp2k(0), (uint32)maxheap - 1);
        kprintf("%10d bytes reserved system area.\r\n",
                (uint32)_start - (uint32)addressp2k(0));
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)addressp2k(0), (uint32)_start - 1);
        kprintf("%10d bytes Xinu code.\r\n",
                (uint32)&_end - (uint32)_start);
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)_start, (uint32)&_end - 1);
        kprintf("%10d bytes stack space.\r\n",
                (uint32)minheap - (uint32)&_end);
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)&_end, (uint32)minheap - 1);
        kprintf("%10d bytes heap space.\r\n",
                (uint32)maxheap - (uint32)minheap);
        kprintf("           [0x%08X to 0x%08X]\r\n\r\n",
```

```
                (uint32)minheap, (uint32)maxheap - 1);

        /* Enable interrupts */

        enable();

        /* Create a process to execute function main() */
        resume(create
          ((void *)main, INITSTK, INITPRIO, "Main process", 0, NULL));

        /* Become the Null process (i.e., guarantee that the CPU has    */
        /*  something to run when no other process is ready to execute) */

        while (TRUE) {
                ;                /* do nothing */
        }
}

/*------------------------------------------------------------------------
 *
 * sysinit - intialize all Xinu data structures and devices
 *
 *------------------------------------------------------------------------
 */

static  void    sysinit(void)
{
        int32   i;
        struct  procent *prptr;         /* ptr to process table entry   */
        struct  dentry  *devptr;        /* ptr to device table entry    */
        struct  sentry  *semptr;        /* ptr to semaphore table entry */
        struct  memblk  *memptr;        /* ptr to memory block          */

        /* Initialize system variables */

        /* Count the Null process as the first process in the system */

        prcount = 1;

        /* Scheduling is not currently blocked */

        Defer.ndefers = 0;

        /* Initialize the free memory list */

        maxheap = (void *)addressp2k(MAXADDR);

        memlist.mnext = (struct memblk *)minheap;

        /* Overlay memblk structure on free memory and set fields */

        memptr = (struct memblk *)minheap;
        memptr->mnext = NULL;
        memptr->mlength = memlist.mlength = (uint32)(maxheap - minheap);

        /* Initialize process table entries free */

        for (i = 0; i < NPROC; i++) {
                prptr = &proctab[i];
```

```
                prptr->prstate = PR_FREE;
                prptr->prname[0] = NULLCH;
                prptr->prstkbase = NULL;
                prptr->prprio = 0;
        }

        /* Initialize the Null process entry */

        prptr = &proctab[NULLPROC];
        prptr->prstate = PR_CURR;
        prptr->prprio = 0;
        strncpy(prptr->prname, "prnull", 7);
        prptr->prstkbase = minheap;
        prptr->prstklen = NULLSTK;
        prptr->prstkptr = 0;
        currpid = NULLPROC;

        /* Initialize semaphores */

        for (i = 0; i < NSEM; i++) {
                semptr = &semtab[i];
                semptr->sstate = S_FREE;
                semptr->scount = 0;
                semptr->squeue = newqueue();
        }

        /* Initialize buffer pools */

        bufinit();

        /* Create a ready list for processes */

        readylist = newqueue();

        /* Initialize real time clock */

        clkinit();
        /* Initialize non-volative RAM storage */

        nvramInit();

        for (i = 0; i < NDEVS; i++) {
                if (!isbaddev(i)) {
                        devptr = (struct dentry *) &devtab[i];
                        (devptr->dvinit) (devptr);
                }
        }
        return;
}
```

nulluser 函数本身相对易懂，它调用 sysinit 来初始化操作系统数据结构。当 sysinit 返回时，运行程序变为空进程（进程 0），但中断仍然是禁止的，也没有其他进程存在。在输出了一些简介信息后，nulluser 开启中断，并调用 create 启动进程来执行用户的主程序。

由于执行 nulluser 的程序已经成为了空进程，所以它本身不能退出、睡眠、等待信号量或者自行暂停。幸运的是，初始化函数不会对调用者进行改变当前状态或准备状态的操作。如果需要执行这种操作，sysinit 会创建另一个进程来完成这种操作。当初始化完成并成功创建了一个用于执行用户主程序的进程后，0 号进程进入了一个无限循环，当没有用户进程处于准备并且可运行的状态的时候，resched 进程会安排 0 号进程运行。

22.7 从程序转化为进程

函数 sysinit 负责系统初始化工作。它初始化系统数据结构，如信号量表、进程表和空闲内存链表。它还调用 clkinit 以初始化实时时钟。最后，sysinit 迭代遍历所有已经配置的设备，并调用每个设备的初始化函数。为了实现这个目的，它从设备转换表项中提取 dvinit 字段，将该字段的值作为地址，调用位于该地址上的函数。

初始化代码中最有趣的部分出现在 sysinit 中间的位置。此段代码用于在初始化时填写 0 号进程的进程表项。进程表的许多字段还没有初始化，如进程名字段——该字段的初始化仅仅是为了方便调试。初始化空进程时有两行关键代码：把当前进程 ID 变量 currpid 设置为空进程的 ID；把 PR_CURR 赋值给进程表中的进程状态字段。只有在为 currpid 和进程表状态字段赋值以后，系统才能进行进程调度。一旦它们被赋值，初始化程序就成为了一个正在运行的进程，resched 就能够把它识别为 ID 为 0 的进程。

527 : 531

总结以上内容：

> 函数 sysinit 在进程表中填写进程 0 的表项之后，就把变量 currpid 设置为 0，于是就转化为一个进程。

在完成空进程的创建之后，sysinit 会对系统中剩余的其他进程进行初始化。这样在 nulluser 函数开始进程执行用户主程序的时候，所有服务都是可用的。

22.8 观点

操作系统设计的精妙之处就在于为底层的硬件创造了新的抽象。对于系统初始化来说，它呈现了程序到进程转化这一概念，这一概念远比它的实现细节更重要：处理器从“取指－执行”为周期的串行执行指令开始，初始化代码将自身转化为一个并行处理系统。这里的关键之处在于初始化代码并没有创建一个独立的、并发系统，然后跳转到新的系统。抽象建立的前后不存在真正的跨越，原来的串行执行程序也并没有被终止。相反，运行中的串行系统声明自己为一个进程，填充进程需要的系统数据结构，最后允许其他进程执行。与此同时，处理器仍然继续着“取指－执行”周期，而新的抽象可以在没有任何干扰的情况下出现。

22.9 总结

初始化是系统设计的最后一步，一定不能为简化初始化过程而牺牲系统的设计。尽管初始化过程涉及诸多细节，但从概念上看，最有趣的部分是将串行程序转化为支持并发处理的系统。为了把自己设置成空进程，初始化程序填写进程表中进程 0 的表项，并将 currpid 设置为 0。

练习

22.1 如果你正在设计一个引导加载程序，你会添加哪些额外的功能？为什么？

532 22.2 进程表、信号量表、内存空闲链表、设备和就绪表的初始化顺序是否重要？请解释。

22.3 在许多系统中，可以实现 sizemem 函数以找到最高有效内存地址。方法是不断探查内存直到有异常发生。请问该函数能否在 E2100L 上实现？为什么？

22.4 根据本章所列出的函数，请解释如果在调用 sysinit 之前，nulluser 就允许中断，会有哪些错误产生？

22.5 如果网络代码、远程磁盘驱动和远程文件系统驱动各自创建了一个进程，请问这些进程是否应该在 sysinit 中创建？为什么？

22.6 大多数操作系统会为网络代码的运行做准备，并在任何用户进程开始之前就获取 IP 地址。请设计一种方法，使 Xinu 可以创建网络进程，并等待网络进程获取 IP 地址，然后创建一个进程运行主程序

533 （注意：空进程不能够阻塞）。

异常处理

我从不允许例外，因为例外看完规则。

——阿瑟·内尼·迪尔爵士

23.1 引言

本章的主题是异常处理。由于底层硬件决定了异常的报告方式，所以操作系统处理异常时所使用的技术完全取决于硬件。我们将以E2100L为例，描述在该系统中的异常处理方式，而其他系统和架构中的异常处理方式将给读者自己探索。

通常，异常处理涉及的细节要多于概念。因此与前面的章节不同，本章不会介绍太多的概念。读者应该把本章所讨论的内容作为一个实例，并牢记一旦使用其他的操作系统，处理异常的细节和技术都可能改变。

23.2 异常、陷阱和恶意中断

将操作系统中的设备地址、中断向量地址和中断调度器正确地配置和连接在一起，对大多数的实现者来说是一项艰巨的任务。硬件和操作系统配置之间的差异，会导致设备进入中断向量表的位置与操作系统的预期位置不同。向嵌入式系统中添加新的设备时，操作系统的软件必须重新配置，以适应新设备的要求。

[535]

程序中的错误与致程序产生异常（exception）时，有时候也叫做陷入（trap），相关的问题就出现了。如果程序试图向一个不是4的整数倍的内存地址读、写一个字，或者试图执行除以0的操作，异常就会发生。记得当程序试图执行非法指令时，硬件处理问题的方式与处理设备中断是相同的。即硬件暂时停止"取指-执行"周期，并使用从异常向量（exception vector）所获得的地址将控制权交给处理异常的操作系统函数。

如果发生未知的中断，则问题出现在系统配置。——操作系统必须被重新设置以应对新的设备。异常非常复杂，其处理取决于系统的大小和目的。如果异常是操作系统代码引起的，那么系统的实现者必须处理该异常；如果异常是第三方的应用程序引起的，那么该异常必须由应用程序的提供者处理。

在传统的操作系统中，当应用程序产生异常时，操作系统可以终止运行该程序的进程，并告知用户有问题发生。然而，对于嵌入式系统来说，异常恢复是困难的，甚至是不可能的。在这种情况下，即使允许用户与系统交互，用户也无能为力修正问题。因此，大部分的嵌入式系统不是选择重启就是关机。

本章的实例代码沿袭UNIX的传统，把处理异常的函数命名为panic。这个版本的panic函数以一个字符串作为参数。在控制台显示该字符串，并调用halt函数来停止处理器。panic函数的代码也极为简单：它既不试图恢复，也不试图识别引起异常的进程。

由于涉及诸多硬件相关的细节，所以显示寄存器或处理器状态的panic函数的版本可能需要用汇编语言编写。例如，因为异常可能由无效的栈指针引起，所以在panic函数中就要避免调用栈。为了应对所有的情况，panic函数不能简单地把变量入栈或执行函数调用。类似地，因为设备转换表的表项可能是不正确的，所以panic函数从设备转换表中获取的控制台（CONSOLE）信息也可能是不可行的。幸运的是，这些情况是十分罕见的。所以，许多操作系统设计者都从panic函数的基本版本开始，只要操作系统和运行环境的大部分运行正常，那该版本的panic函数就仍然可用。

23.3 panic的实现

本章实现的panic函数版本十分简单：在控制台（CONSOLE）上显示一条消息，关闭"运行"指示灯（即面板上的LED灯），并调用halt函数。因为MIPS架构没有实际停止处理器的指令，我们实现

536 的 halt 函数简单地进入死循环。文件 panic.c 包含以下代码：

```
/* panic.c - panic */

#include <xinu.h>

/*------------------------------------------------------------------------
 * panic - display a message and stop all processing
 *------------------------------------------------------------------------
 */
void    panic (
          char  *msg                    /* message to display           */
        )
{
        intmask mask;                   /* saved interrupt mask         */

        mask = disable();
        kprintf("\n\n\rpanic: %s\n\n\r", msg);
        ledctl(GPIO_LED_CLEAR(0));      /* turn off LED "run" light     */
        halt();                         /* halt the processor           */
}
```

23.4 观点

处理异常比普上大要更加复杂。考虑一个应用程序正运行系统调用：尽管应用程序的进程仍在运行，但它执行的是操作系统的代码。如果此时异常发生，该异常应当被认为是操作系统的问题，而不应该调用进程的异常处理函数。类似地，当应用程序执行共享库中的代码时，引起的异常不应当与应用程序本身所产生的异常同等对待。异常处理中的这些差别要求操作系统能够准确地跟踪应用程序当前的运行状态。

进程交互所引起的异常也使异常处理更加复杂。例如，如果一个 Xinu 进程不小心写入了另一个进程的地址空间，那么第一个进程的行为将引发第二个进程的异常。因此，即使系统提供了捕获异常的
537 机制，第二个进程的异常处理函数也可能无法侦测该问题，从而没有办法从异常中恢复。

23.5 总结

捕捉和识别异常和未预期的中断是十分重要的，因为它们有助于隔离实现中操作系统的错误。因此，有必要及早地创建错误捕捉函数，即使它们的实现是粗糙和简陋的。

在嵌入式系统中，异常往往致使系统重启或关机。本章给出了 panic 函数的实例，它假定操作系统的大部分是运行正常的。在此种情况下，panic 要做的事情是禁止中断，在控制台上输出一条消息，调用 halt 函数停止处理器。

练习

23.1 改写 Xinu 使系统代码 n 连续调用，并修改 panic 函数使其等待 15 秒，然后跳转到起始位置（即重启）。

23.2 在 panic 函数中，需要多少个其门栈中的位置来处理一个异常？

23.3 设计一个允许执行中的进程捕获异常的机制。

23.4 一台旧的 LSI-11 计算机能够捕获电源故障的异常，而作者也曾看到过 Xinu 输出电源故障的信息。你能否找到一个处理器，它能够检测到因发生的电源故障？从电源故障到关机这段时间内，有多少条指令可以执行？

538 23.5 列出 panic 函数的运行需求（提示：是否需要调用栈？）

系统配置

没有变化的快乐不能持久。

——Publilius Syrus

24.1 引言

本章通过解决一个实际问题来讨论操作系统的基本设计：如何对前面章节中的代码进行转换使其能够在有特定设备的计算机上工作。

本章介绍配置的目的，静态配置和动态配置之间的权衡，并提供基本的配置程序，该程序可以根据对系统的描述来生成与描述相匹配的源文件。

24.2 多重配置的需求

早期的计算机是作为单片系统来设计的，硬件和软件是一起设计的。设计者选择CPU、内存和I/O设备的细节，并设计一个操作系统来控制硬件。后来的计算机增加了一些选项，允许用户选择大的内存还是小的内存和大的磁盘还是小的磁盘。随着产业的成熟，第三方供应商开始出售能够连接到计算机上的外围设备。现在的计算机有很多的选择——购买者能够在供应商提供的众多外围设备中进行选择。因此，一个给定的计算机应该是多种硬件设备的一种组合，而且这种组合可能不同于其他的计算机。 [541]

设计者使用两种方法来识别设备的组合：

- 静态系统配置
- 动态设备驱动

静态系统配置 静态配置用于自成一体的小型嵌入式系统中。对于一个典型的嵌入式系统来说，操作系统仅仅支持可用的硬件设备，并不包含任何额外的模块。为了设计这样的系统，设计者编写规范来说明这个系统上的正确的外围设备。规范是控制操作系统源代码的配置程序的输入。配置程序使用规范来选择目标设备所需要的模块，同时去除其他硬件模块。当生成代码进行编译时，就可以认为硬件已经配置好了。

动态设备驱动 动态设备驱动用于拥有大量资源的大型系统中。基本的操作系统在不知道具体硬件的条件下开始运行。系统负责探测硬件设备，确定哪些设备已经存在，并自动地载入设备驱动。当然，驱动软件需要在本地磁盘上可用，或者能够由设备提供，或者能够在网络上下载。动态配置需要花费额外的时间（例如，引导程序需要额外的时间）。

静态配置是早期绑定的形式。它最主要的优点在于内存映像中只包含存在的硬件模块。另一个优点表现在系统在启动期间不需要花时间来识别硬件，信息是绑定在代码中的。早期配置方法最主要的缺点是，为一台机器所配置的系统不能用于另一台机器，除非这两台机器是完全一样的，包括内存大小在所有设备的细节。

推迟配置的时间到系统开始启动能够使设计者编写出更短的代码，因为一个单独的系统能够在多种硬件配置上执行。在启动阶段，系统能够使自己与硬件相适应。更重要的是，动态配置能够使系统在不停止运行的条件下适应硬件的变化（例如，当使用者插上或拔除一个USB设备时）。

24.3 Xinu系统配置

因为它作为嵌入式系统运行，Xinu允许静态配置方法，配置工作主要发生在系统编译和链接期间。当然，即使在某些嵌入式系统中，一部分配置工作也能够推迟到系统启动之后。例如，有些版本的Xinu在Xinu开始运行之后计算内存的大小并检测实时时钟的存在。在这些系统中，中断向量的初始化也发生在系统运行之后，系统调用驱动初始化程序来进行中断向量的初始化。 [542]

Xinu 使用配置程序来自动地筛选设备驱动模块。不过，config 程序并不是操作系统的一部分，我们不需要检查其源代码。相反，我们要察看 config 如何运行：它有一个包含规范说明的输入文件，并生成能够成为操作系统代码一部分的输出文件。24.4 节将解释配置程序并给出例子。

24.4 Xinu 配置文件的内容

config 程序将一个命名为 Configuration 的文本文件作为输入。它解释输入文件并产生两个输出文件 conf.h 和 conf.c。我们已经看到过输出文件，它包含对设备定义的常量和设备转换表的定义[⊖]。

文件 Configuration 被分隔符 "%%" 划分为 3 部分：

- 第一部分：设备类型的类型声明。
- 第二部分：特定设备的设备规范。
- 第三部分：自动产生的符号常量。

24.4.1 第一部分：类型声明

类型声明是为了应对系统可能包含特定硬件设备的多个副本的情况。例如，一个系统可能有两个使用 tty 这个抽象概念的 UART 设备，而包含 tty 设备的函数集合必须对每一个 UART 设备是特定的。手动多次输入规范说明容易导致错误，且可能产生不一致。因此，类型声明允许只输入规范说明一次，并分配一个在设备规范部分两个设备都能使用的名字。

每一个类型声明为一种类型的设备定义一个名字，并为这个类型列出一个默认的设备驱动函数集合。这个声明同时允许指明这种类型与哪个设备硬件绑定。例如，类型声明：

```
tty:
        on uart
                -i ttyInit      -o ionull       -c ionull
                -r ttyRead      -g ttyGetc      -p ttyPutc
                -w ttyWrite     -s ioctl        -n ttyControl
                -intr ttyInterrupt              -irq 11
```

定义一种名为 tty 的、使用在 UART 设备上的类型。tty 和 uart 既不是关键词也没有任何意义。它们仅仅是设计者所选择的名字。剩下的条目指出类型 tty 的默认设备函数。每一个驱动函数之前都有一个以负号开头的关键字。图 24-1 列出了可能的关键字并给出了它们的含义。注意：一个给定的规范并不一定需要使用所有的关键字。

[543]

关键字	含义
-i	执行 init
-o	执行 open
-c	执行 close
-r	执行 read
-w	执行 write
-s	执行 seek
-g	执行 getc
-p	执行 putc
-n	执行 control
-intr	执行 interrupt
-csr	控制和状态寄存器地址
-irq	中断向量号

图 24-1 Xinu 配置文件中使用的关键字和它们的含义

⊖ 文件 conf.h 可以在 14.9 节找到，conf.h 可以在 14.14 节找到。

24.4.2 第二部分：设备规范

Configuration 文件的第二部分包含了系统中每个设备的声明。声明需要给出设备的名称（例如，CONSOLE），并指明构成驱动的函数集合。在 Xinu 中，设备是一个抽象的概念，不需要与物理硬件设备相连。例如，除了像 CONSOLE 和 ETHERNET 与指定的硬件设备相一致外，设备部分还能列出伪设备，如用于 I/O 的 FILE 设备。

声明一个设备有两个目的。首先，它在设备转换表里给设备分配位置，允许使用高级别的 I/O 原语来指明设备，而不需要程序来调用特定的驱动函数。其次，它允许 config 程序给每个设备分配一个数小的设备号。所有同样类型的设备都分配了从 0 开始序列的最小可用设备号。

当一个设备被声明后，其特定值可以按需提供，其驱动函数也可以被重写。例如，声明：

```
CONSOLE is tty  on uart -csr 0x88020000
```

声明 CONSOLE 作为运行在 UART 硬件上的类型 tty 的设备。此外，这个声明指出其控制和状态寄存器 CSR 的地址为 0x88020000。 544

如果程序员想要测试新版本的 ttyGetc，只需要改变说明书为：

```
CONSOLE is tty  on uart -csr 0x88020000  -g myttyGetc
```

上面给出的 tty 声明中使用的是默认的驱动函数，但参数 myttyGetc 重写了 getc 函数。需要注意的是，使用配置参数可以很方便地在不改变或者替换原文件的情况下改变函数。

24.4.3 自动产生符号常量

除了定义设备转换表的结构外，conf.h 中还包含设备总数和每个类型号的常量。而 config 程序生成用于反映 Configuration 文件中设备规范的常量。例如，常量 NDEVS 是一个整数，用于指明已经配置的设备总数，而设备转换表则包含了 NDEVS 个设备，与设备无关的 I/O 通常使用 NDEVS 来检测设备 ID 的合法性。

config 还生成一些其他变量的集合，这些变量指明每个类型的设备号。驱动函数可以使用适当的常量来声明控制块数组。每个常量的形式都为 Nxxx，这里的 xxx 为类型名。例如，如果文件 Configuration 定义了两个 tty 类型的设备，conf.h 将包含下面这行代码：

```
#define Ntty  2
```

24.5 计算次设备号

下面我们来看看配置（config）相关的文件。conf.h 包含设备转换表的声明，conf.c 包含初始化表的代码。对于给定的设备，其 devtab 表项包含了指向设备驱动程序指针集合，这个程序与 open，close，read 和 write 等高级 I/O 操作相对应。表项还包含了中断向量地址和设备的 CSR 地址。设备转换表中的所有信息直接从文件 Configuration 得到。 545

如前所述，设备转换表中的每个表项都包含一个次设备号。次设备号是用于区别多个相同类型设备的一个整数。记得设备驱动函数用次设备号作为控制块数组的索引，与每个设备里一个特殊表项相关联。本质上，config 程序对每种类型的设备进行计算，当发现一个设备时，config 就根据设备类型分配下一个次设备号（数字从 0 开始）。例如，图 24-2 说明了在一个有 3 台 tty 设备和 2 台 eth 设备的系统中，如何分配设备标识符和次设备号。

设备名	设备标识符	设备类型	次设备号
CONSOLE	0	tty	0
ETHERNET	1	eth	0
COM2	2	tty	1
ETHER2	3	eth	1
PRINTER	4	tty	2

图 24-2 设备配置的示例

注意，虽然三个 tty 设备的编号为 0，2 和 4，但是它们的次设备号为 0，1 和 2。

24.6 配置 Xinu 系统的步骤

为了配置 Xinu 系统，程序员需要编辑 Configuration 文件，依照要求增加或者修改设备信息和符号常量。在运行过程中，config 程序首先读取和解析该文件，收集每种设备类型的信息。然后读取设备规范，分配次设备号并生成输出文件 conf.c 和 conf.h。最后，config 程序将规范第三部分中的符号常量加入到 conf.h 中，使它们对于操作系统可用。

[546] 在 config 生成了 conf.c 和 conf.h 的新版本后，conf.c 必须重新编译。

24.7 观点

操作系统的历史是从静态配置发展到动态配置的过程。有趣的问题是动态配置的益处是否大于它的成本。例如，比较 Xinu 和大型的商业操作系统，比如 Windows，即使底层的计算机硬件没有经常改变，但商业操作系统总是通过轮询总线来找到现有驱动，加载驱动，然后与每个设备相互作用。如果操作系统只有在硬件改变的时候才进行重新配置，那么系统就可以很快启动，接近 Xinu 所用的时间。

24.8 总结

设计者寻求方法使操作系统能够配置，而不是为特定的硬件建立完整的操作系统。静态配置在系统编译和链接的时候选择模块，而动态配置在运行时才加载设备驱动模块。

因为 Xinu 是为嵌入式系统设计的，所以它使用静态配置。config 程序读取 Configuration 文件并生成 conf.c 和 conf.h 文件，它们定义和初始化设备转换表。将设备类型从设备声明中分离使得配置程序能够计算设备的次设备号。

练习

24.1 编写函数 myttyRead，它错误调用 ttyGete 来满足需求。为了测试你的代码，修改 Configuration 文件来代替 ttyRead 部分的代码。

24.2 查找其他系统是如何进行配置？例如，当 Windows 启动时，发生了什么？

24.3 如果每个操作系统的函数都包括了 conf.h，那么任何对 Configuration 文件的修改都意味着生成一个新版本的 conf.h，而且整个系统必须重新编译。重新编写 config 程序使其将不同导入文件中的常量分类，以消除不必要的编译。

24.4 讨论配置程序是否值得，包括一些使系统更容易配置所需要的额外代价。记住程序员在系统第一次配置的时候很有可能没有经验。

24.5 理论上，当将系统从一台计算机移植到另一台计算机的时候，系统的很多部分是需要改变的。除了设备外，例如，考虑处理器（不只是基本的指令集，也可能是某些模块的高加指令），协处理器的可用性（包括浮点数）、实时时钟或时间源所，以及整型的字节顺序。证明如果系统有以上的参数，

[547] 那么该系统是不可测试的。

一个用户接口例子：Xinu 壳

一个人需要明白，他不能控制一切……

——James Allen

25.1 引言

前面的章节以函数集合来阐述操作系统，应用程序可以调用这些函数来获得服务。但是，一个普通用户从来不会接触系统函数，而是调用应用程序，通过应用程序来访问底层系统函数。

本章将研究基础的用户接口——壳（shell）[⊖]，用户可以通过它来启动应用程序并控制这些应用程序的输入/输出。壳的设计将遵循系统其他部分的模式，强调简单、优雅而不是复杂的功能。本章将关注一些基础思想，使得完本需要大量代码就能足够强大。本章还将介绍一些例子，包括解释用户命令的软件和用户可调用的应用程序。本章的解释器例子虽然只提供了基本的功能，但是它能够说明几个重要的概念。 549

25.2 用户接口

用户接口包括硬件和软件，用户可以与之交互执行计算任务并观察结果。因此，用户接口软件处于用户和计算机系统之间，用户指定需要做什么，而计算机系统则执行指定的任务。

用户接口设计的目标是创建一个工作环境，使得用户在其中执行计算任务方便而高效。比如，大多数现代用户接口都利用了图形表示，它包含一系列图标，用户可以选择这些图标来启动应用。图像的使用使得应用程序的选择变得快捷，用户也不再需要记住一大堆的应用程序名。

典型的小型嵌入式系统提供两层用户接口：一层面向终端用户，另一层面向系统构建者。比如 WRT54GL 路由器，它提供了两层接口：一层面向用户的 Web 接口，一层面向程序员的控制台接口。当用户使用无线路由器时，用户启动路由器，然后利用一个常用的浏览器与该设备进行交互。Web 接口允许用户设置密码，控制无线网络硬件以及设置路由。可是，Web 接口不能用来下载软件或者修改系统。要执行该类任务，程序员必须使用控制台接口并通过串行线与之交互。

25.3 命令和设计原则

业界一般使用命令行接口（Command Line Interface，CLI）来描述一个允许用户输入一系列文本命令的用户接口。许多嵌入式系统产品都提供了命令行接口。通常，每一行输入对应一条命令，系统执行完一行命令再读取下一行命令。术语命令（command）的出现是因为大多数命令行接口都遵循相同的语法格式，每一行以一个名字开头指定行为，而后接着的参数用来指定该行为的详细内容以及该行为的对象。例如，想象一个系统使用命令 config 来控制与网络接口相关的设置，那么设置接口 0 的 MTU 参数的命令可能是：

```
config    0    MTU=1500
```

所有可用命令的集合决定了用户可用的功能（即定义了计算系统的能力）。可是，好的设计不只是收集随机命令，它需要遵循以下原则：

- 功能性：充分满足所有需要。
- 正交性：只有一种方式执行指定任务。
- 一致性：命令遵循一致模式。
- 最小意外性：用户应该能预测结果。 550

⊖ 术语“壳”和 Xinu 系统中其他许多思想都来自 UNIX。

25.4 一个简化壳的设计决策

当程序员设计壳时，他必须从诸多决策中进行选择。下面的段落将展示程序员需要面对的方方面面，并描述一个简化的 Xinu 壳需要做哪些选择。

输入处理 在处理回退、字符回显、行删除的细节时，接口设计终端设备驱动处理还是自己处理？由于该选择决定了完能够控制输入的程度，所以这是一个重要的决定。例如，现代 UNIX 壳在进行行输入编辑时允许使用 Control-B 和 Control-F 来移动光标的位置[⊖]。而 Xinu 的 tty 驱动并没有提供这样的编辑功能。

对台或者后台执行 壳在开始一条命令前需要等待前一个命令完成吗？我们的壳遵循 UNIX 传统，允许用户来决定是等待还是后台执行命令。

输入和输出控制 也与 UNIX 一样，我们的壳允许用户指定输入源地址和输出目的地址。这种技术称为输入/输出重定向（I/O redirection），它允许每条命令像通用工具一样可以应用于各种文件和输入/输出设备。提供重定向的壳也意味着输入/输出格式是统一的——一个单一的重定向机制适用于所有命令。

类型化或非类型化参数 该问题是指壳是否理解一个指定命令的参数数量和类型。按照 UNIX 传统，我们的壳不明白参数，也不会解释它们。相反，壳将每个参数视为文本字符串，将参数集合再传递给命令。因此，每个命令必须检查它的参数是否合法。

25.5 壳的组织和操作

壳被组织成一个循环，反复读取输入行并执行命令。一旦一行被读取，壳必须提取命令名、参数以及其他东西，如输入/输出重定向或者后台执行指示。按照语法分析的标准惯例，我们将这些代码分为两个函数：一个进行词法分析，给字符分组生成符号（token）；另一个检查这些符号集合是否形成一个合法的命令。

使用单独的词法分析函数对我们简单的壳范例语法或许不太必要。然而，由于它方便以后的扩展，
[551] 我们还是选择了该组织方式。

25.6 词法符号的定义

在词法层，我们的壳扫描输入行并将字符分组为具有语义的符号。图 25-1 列举了扫描器在识别词法符号以及分类符号时所用到的 4 种词法类型。

令牌类型（数字，值）	字符	描述
SH_TOK_AMPER (0)	&	"与"符号
SH_TOK_LESS (1)	<	"小于"符号
SH_TOK_GREATER (2)	>	"大于"符号
SH_TOK_OTHER (3)	'...'	被引号括起来的字符串（单引号）
SH_TOK_OTHER (3)	"..."	被引号括起来的字符串（双引号）
SH_TOK_OTHER (3)	其他	非空格字符序列

图 25-1 Xinu 壳使用的词法符号

使用带有引号的字符串允许用户指定包含任意字符串的参数（或者文件名），包括壳识别的特殊字符。每一个被引用的字符串以单引号或者双引号开始，可以包含所有字符，包括空格和制表符，直到碰到相应的结尾引号。因此，字符串

```
'a string'
```

⊖ 对 Control-B 和 Control-F 的使用由 Emacs 编辑器派生而来。

有 8 个字符，包括一个空格符，而字符串

"don't blink"

包含 11 个字符，包括一个单引号字符。词法扫描器移除两侧的引号，将剩下的字符串序列分类为 SH_TOK_OTHER。

词法扫描器将仅包含空格符或者制表符的字符定义为空白字符。两个 SH_TOK_OTHER 类型的符号之间至少有一个空字符，否则空白符将被忽略。

25.7　命令行语法的定义

当一行输入被扫描并被分割成一系列词法符号时，壳解析符号并验证它们是不是有效的序列。壳所用的语法是：

```
command_name arg* [redirection][background]
```

552

方括号[ ]表示参数可选，星号 * 表示某项有 0 个或多个。字符串 command_name 表示命令的名称，arg * 表示参数可以有 0 个或多个，可选项 redirection 表示：输入重定向、输出重定向，或者两者都有，background 表示是否后台执行。图 25-2 包含了利用符号来定义有效输入的语法。

命令	⟶	命令名［参数］［重定向命令］［后台命令］
命令名	⟶	SH_TOK_OTHER
参数	⟶	SH_TOK_OTHER［参数］
重定向命令	⟶	输入重定向命令［输出重定向命令］
重定向命令	⟶	输出重定向命令［输入重定向命令］
输入重定向命令	⟶	SH_TOK_LESS SH_TOK_OTHER
输出重定向命令	⟶	SH_TOK_GREATER SH_TOK_OTHER
后台命令	⟶	SH_TOK_AMPER

图 25-2　有效符号序列的语法

本质上，一个命令包含一个或者多个“其他的”可选令牌，它们表示重定向输入/输出或者后台执行的符号。一行中第一个符号必须是命令名。

25.8　Xinu 壳的实现

我们从壳中使用的常量和变量的定义开始来检查它的实现。文件 shell.h 包含这些声明。 553

```
/* shell.h - declarations and constants used by the Xinu shell */

/* Size constants */

#define SHELL_BUFLEN    TY_IBUFLEN+1    /* length of input buffer       */
#define SHELL_MAXTOK    32              /* maximum tokens per line      */
#define SHELL_CMDSTK    8192            /* size of stack for process    */
                                        /*    that executes command     */
#define SHELL_ARGLEN    (SHELL_BUFLEN+SHELL_MAXTOK) /* argument area    */
#define SHELL_CMDPRIO   20              /* process priority for command */

/* Message constants */

/* Shell banner (assumes VT100) */

#define SHELL_BAN0      "\033[1;31m"
#define SHELL_BAN1      "     ------------------------------------"
#define SHELL_BAN2      "   __    __   ____   _   _   _   _   "
#define SHELL_BAN3      "   \\ \\  / /  |__ __|  | \\ | |  | |  | |   "
#define SHELL_BAN4      "    \\ \\/ /     | |    |  \\| |  | |  | |   "
#define SHELL_BAN5      "    / /\\ \\     | |_   | \\   |  | |  | |   "
```

```
#define SHELL_BAN6      "   / /  \\ \\  |     |  | | \\ |  \\  --  /   "
#define SHELL_BAN7      "   --   --   -----   -   -    ----     "
#define SHELL_BAN8      "-----------------------------------------"
#define SHELL_BAN9      "\033[0:39m\n"

/* Messages shell displays for user */

#define SHELL_PROMPT    "xsh $ "                /* prompt                  */
#define SHELL_STRTMSG   "Welcome to Xinu!\n"    /* Welcome message         */
#define SHELL_EXITMSG   "Shell closed\n"        /* shell exit message      */
#define SHELL_SYNERRMSG "Syntax error\n"        /* syntax error message    */
#define SHELL_CREATMSG  "Cannot create process\n"/* command error          */
#define SHELL_INERRMSG  "Cannot open file %s for input\n" /* input err     */
#define SHELL_OUTERRMSG "Cannot open file %s for output\n"/* output err    */
#define SHELL_BCERRMSG  "Cannot redirect I/O or background a builtin\n"
                                                /* builtin cmd err         */
/* Constants used for lexical analysis */

#define SH_NEWLINE      '\n'            /* New line character              */
#define SH_EOF          '\04'           /* Control-D is EOF                */
#define SH_AMPER        '&'             /* ampersand character             */
#define SH_BLANK        ' '             /* blank character                 */
#define SH_TAB          '\t'            /* tab character                   */
#define SH_SQUOTE       '\''            /* single quote character          */
#define SH_DQUOTE       '"'             /* double quote character          */
#define SH_LESS         '<'             /* less-than character  */
#define SH_GREATER      '>'             /* greater-than character          */

/* Token types */

#define SH_TOK_AMPER    0               /* ampersand token                 */
#define SH_TOK_LESS     1               /* less-than token                 */
#define SH_TOK_GREATER  2               /* greater-than token              */
#define SH_TOK_OTHER    3               /* token other than those          */
                                        /*   listed above (e.g., an        */
                                        /*   alphanumeric string)          */

/* Shell return constants */

#define SHELL_OK        0
#define SHELL_ERROR     1
#define SHELL_EXIT      -3

/* Structure of an entry in the table of shell commands */

struct  cmdent  {                       /* entry in command table          */
        char    *cname;                 /* name of command                 */
        bool8   cbuiltin;               /* is this a builtin command?      */
        int32   (*cfunc)(int32,char*[]);/* function for command            */
};

extern  uint32  ncmd;
extern  const   struct  cmdent  cmdtab[];
```

文件 shell.h 最后部分定义的 cmdtab 表负责保存壳命令信息。表中的每一个表项都是一个 cmdent 结构，它包含 3 项：命令名、指定命令是否严格要求内置执行的布尔值，和指向实现命令函数的指针。后面的章节将讨论命令表如何初始化以及如何使用。

25.9　符号的存储

Xinu 壳使用的数据结构有点令人意外：它用一个叫做 toktyp 的整数数组来记录每个符号的类型，并将这些符号存储为以空值终止的字符串，打包存储在一个字符串数组 tokbuf 组成的连续区域内；用一个整数数组 tok 来保存每个符号的起始索引。Xinu 壳依赖于两个计数器：ntok，统计目前找到的符号数；tlen，统计 tokbuf 数组中保存的字符数。为了理解该数据结构，不妨考虑如下输入行的例子：

554 ~ 555

```
date > file &
```

该行包含 4 个符号。图 25-3 展示了词法分析器是如何利用填充数据结构来保存输入行中所提取的符号

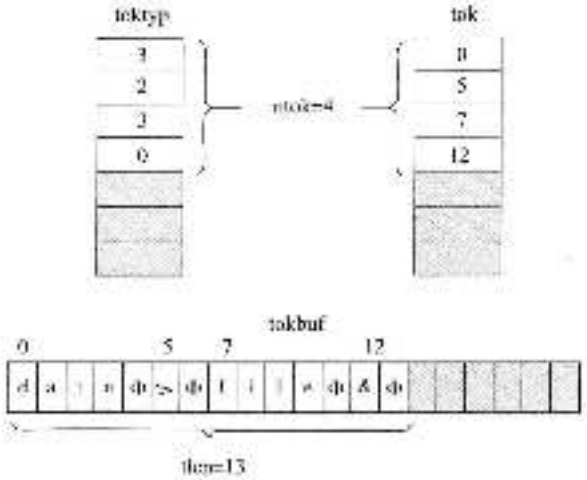


图 25-3　输入行“date > file &”的变量 tokbuf、toktyp、ntok 和 tlen 的内容

如图 25-3 所示，符号自身保存在数组 tokbuf 中，空白字符被移除。每个符号以一个空字符结尾。数组 tok 中保存的每个整数都是指向 tokbuf 的一个索引值——tok 的第 i 个元素给出了第 i 个符号在 tokbuf 中的位置。最后，数组 toktyp 的第 i 个元素表明了第 i 个符号的类型。例如，输入行的第三个符号 file 的类型是 3（SH_TOK_OTHER[⊖]），它的起始值为数组 tokbuf 的第 i 个元素。

25.10　词法分析器代码

由于 Xinu 壳语法很简单，所以其词法分析也十分容易。文件 lexan.c 包含这些代码。 556

```
/* lexan.c - lexan */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lexan - ad hoc lexical analyzer to divide command line into tokens
 *------------------------------------------------------------------------
 */

int32   lexan {
        char        *line,          /* input line terminated with   */
                                    /*  NEWLINE or NULLCH           */
        int32       len,            /* length of the input line,    */
                                    /*   including NEWLINE          */
        char        *tokbuf,        /* buffer into which tokens are */
                                    /*   stored with a null         */
                                    /*   following each token       */
        int32       *tlen,          /* place to store number of     */
```

⊖ 图 25-1 列出了每个令牌类型的数字值。

```
                                        /*   chars in tokbuf                */
    int32       tok[],                  /* array of pointers to the         */
                                        /*   start of each token            */
    int32       toktyp[]                /* array that gives the type        */
                                        /*   of each token                  */
)
{
    char    quote;                      /* character for quoted string      */
    uint32  ntok;                       /* number of tokens found           */
    char    *p;                         /* pointer that walks along the     */
                                        /*   input line                     */
    int32   tbindex;                    /* index into tokbuf                */
    char    ch;                         /* next char from input line        */

    /* Start at the beginning of the line with no tokens */

    ntok = 0;
    p = line;
    tbindex = 0;

    /* While not yet at end of line, get next token */

    while ( (*p != NULLCH) && (*p != SH_NEWLINE) ) {

        /* If too many tokens, return error */

        if (ntok >= SHELL_MAXTOK) {
            return SYSERR;
    }

    /* Skip whitespace before token */

    while ( (*p == SH_BLANK) || (*p == SH_TAB) ) {
            p++;
    }

    /* Stop parsing at end of line (or end of string) */

    ch = *p;
    if ( (ch==SH_NEWLINE) || (ch==NULLCH) ) {
            *tlen = tbindex;
            return ntok;
    }

    /* Set next entry in tok array to be an index to the    */
    /*   current location in the token buffer               */

    tok[ntok] = tbindex;    /* the start of the token       */

    /* Set the token type */

    switch (ch) {

        case SH_AMPER:      toktyp[ntok] = SH_TOK_AMPER;
                            tokbuf[tbindex++] = ch;
                            tokbuf[tbindex++] = NULLCH;
                            ntok++;
                            p++;
                            continue;

        case SH_LESS:       toktyp[ntok] = SH_TOK_LESS;
```

```
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;

    case SH_GREATER:    toktyp[ntok] = SH_TOK_GREATER;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;

    default:            toktyp[ntok] = SH_TOK_OTHER;
};

/* Handle quoted string (single or double quote) */

if ( (ch==SH_SQUOTE)  | (ch==SH_DQUOTE) ) {
        quote = ch;     /* remember opening quote */

        /* Copy quoted string to arg area */

        p++;    /* Move past starting quote */

        while ( ((ch = *p++) != quote) && (ch != SH_NEWLINE)
                        && (ch != NULLCH) ) {
                tokbuf[tbindex++] = ch;
        }
        if (ch != quote) {  /* string missing end quote */
                return SYSERR;
        }

        /* finished string - count token and go on      */

        tokbuf[tbindex++] = NULLCH; /* terminate token  */
        ntok++;         /* count string as one token    */
        continue;       /* go to next token             */
}

/* Handle a token other than a quoted string            */

tokbuf[tbindex++] = ch; /* put first character in buffer*/
p++;

while ( ((ch = *p) != SH_NEWLINE) && (ch != NULLCH)
        && (ch != SH_LESS)  && (ch != SH_GREATER)
        && (ch != SH_BLANK) && (ch != SH_TAB)
        && (ch != SH_AMPER) && (ch != SH_SQUOTE)
        && (ch != SH_DQUOTE) ) {
                tokbuf[tbindex++] = ch;
                p++;
}

/* Report error if other token is appended */
if (        (ch == SH_SQUOTE)   | (ch == SH_DQUOTE)
        || (ch == SH_LESS)      | (ch == SH_GREATER) ) {
        return SYSERR;
}
```

```
            tokbuf[tbindex++] = NULLCH;     /* terminate the token  */

            ntok++;                         /* count valid token    */

        }
        *ntok = tbindex;
        return ntok;
    }
```

函数 lexan 的前两个参数给出了输入行的地址和它的长度。后面的参数给出了图 25-3 中数据结构的指针。lexan 首先初始化找到的符号数量、输入行的指针以及数组 tokbuf 中的索引，然后进入 while 循环运行直到指针 p 到达文件的结尾。

在处理符号时，lexan 跳过空白字符（即空格和制表符），将 tokbuf 的当前索引保存在 tok 中，用 switch 语句选择适用于下一个输入字符的行为。对于 3 个单字符符号（即 &、> 和 <），lexan 将符号类型记录在数组 toktyp 中，将符号以空值结尾放在 tokbuf 数组中，递增 ntok，移动到字符串的下一个字符，然后继续执行 while 循环来处理下一个输入字符。

当字符不是这 3 个单字符符号的时候，lexan 记录这些符号类型为 SH_TOK_OTHER，然后继续进入 switch 语句中。此时有两种情况。符号是引号引起的字符串；或者以特殊字符或空白字符结尾的连续字符串。lexan 能够识别单引号和双引号字符，字符串以第一个匹配的引号或者文件结束符结束。如果碰到文件结束符，lexan 就返回 SYSERR；否则，它毫无修改地将字符串复制到 tokbuf 数组中，也就是说，它能包含任意字符，包括空白或者其他引号标记的字符。当复制操作结束时，词法分析器添加一个 null 字符来定义符号的结尾。然后继续在 while 循环中寻找下一个符号。

557 ~ 560

代码的最后一部分处理包含连续字符的符号。代码循环直到找到一个特殊字符或者空白字符，然后将字符放在数组 tokbuf 接下来的位置中。在继续处理下一个符号前，代码检查两个符号间是否有空白。

当 lexan 碰到行的结束符时，它就返回找到的符号数量。如果在执行期间检测到一个错误，lexan 就返回 SYSERR 给调用者，并不尝试修复或复原问题。练习题中讨论了错误处理和继续的替代方案。

25.11 命令解释器的核心

尽管命令解释器必须处理许多细节，但总的基本的算法还是不难理解的。本质上，其代码包含一个重复读取输入行的循环，然后用 lexan 抽取符号，检查语法，传递参数，如果有必要还要进行 I/O 重定向，并且根据说明在后台或者前台运行命令。如果用户输入文件结束符（control-d）或者命令返回特殊的退出码，循环就终止。

与词法分析器一样，命令解释器使用了一种特殊的实现。它的代码既不像传统的编译器，也不包括独立的代码来检查符号序列是否正确。相反，在处理过程的每一步中它都进行错误检查。例如，在处理完后台参数和 I/O 重定向参数之后，Xinu 壳就认为下的符号是否都是 SH_TOK_OTHER 类型。

下面的文件 shell.c 包含这部分的代码。需要注意的是，代码中还声明了 cmdtab 数组，该数组中指定了命令集和处理这些命令相应的函数。此外代码还包含外部变量 ncmd，它指明表中命令的数量。

在概念上，命令集独立于处理用户输入的代码。因此，将 shell.c 分为两个文件是合理的：一个文件定义命令，另一个包括处理代码。然而，实际上这两个文件被合并为一个，因为在这个例子中命令集很少，增加一个文件没有必要。

561

```
/* shell.c  -  shell */

#include <xinu.h>
#include <stdio.h>
#include "shprototypes.h"

/************************************************************************/
/* Xinu shell commands and the function associated with each            */
/************************************************************************/
```

```
const	struct	cmdent	cmdtab[] = {
	{"argecho",	TRUE,	xsh_argecho},
	{"arp",		FALSE,	xsh_arp},
	{"cat",		FALSE,	xsh_cat},
	{"clear",	TRUE,	xsh_clear},
	{"date",	FALSE,	xsh_date},
	{"devdump",	FALSE,	xsh_devdump},
	{"echo",	FALSE,	xsh_echo},
	{"ethstat",	FALSE,	xsh_ethstat},
	{"exit",	TRUE,	xsh_exit},
	{"help",	FALSE,	xsh_help},
	{"ipaddr",	FALSE,	xsh_ipaddr},
	{"kill",	TRUE,	xsh_kill},
	{"led",		FALSE,	xsh_led},
	{"memdump",	FALSE,	xsh_memdump},
	{"memstat",	FALSE,	xsh_memstat},
	{"nvram",	FALSE,	xsh_nvram},
	{"ping",	FALSE,	xsh_ping},
	{"ps",		FALSE,	xsh_ps},
	{"sleep",	FALSE,	xsh_sleep},
	{"udpdump",	FALSE,	xsh_udpdump},
	{"udpecho",	FALSE,	xsh_udpecho},
	{"udpeserver",	FALSE,	xsh_udpeserver},
	{"uptime",	FALSE,	xsh_uptime},
	{"?",		FALSE,	xsh_help}

};

uint32	ncmd = sizeof(cmdtab) / sizeof(struct cmdent);

/*------------------------------------------------------------------------*/
/* Xinu shell	provide an interactive user interface that executes	  */
/*		commands.  Each command begins with a command name, has	  */
/*		a set of optional arguments, has optional input or	  */
/*		output redirection, and an optional specification for	  */
/*		background execution (ampersand).  The syntax is:	  */
/*									  */
/*		   command_name [args*] [redirection] [&]		  */
/*									  */
/*		Redirection is either or both of:			  */
/*									  */
/*				< input_file				  */
/*		      or						  */
/*				> output_file				  */
/*									  */
/*------------------------------------------------------------------------*/

process shell (
		did32	dev		/* ID of tty device from which	*/
	)				/*  to accept commands		*/
{
	char	buf[SHELL_BUFLEN];	/* input line (large enough for	*/
					/*  one line from a tty device	*/
	int32	len;			/* length of line read		*/
	char	tokbuf[SHELL_BUFLEN +	/* buffer to hold a set of	*/
			SHELL_MAXTOK];	/* contiguous null-terminated	*/
					/* strings of tokens		*/
	int32	tlen;			/* current length of all data	*/
					/*   in array tokbuf		*/
```

```
int32   tok[SHELL_MAXTOK];      /* index of each token in       */
                                /*   tokbuf                     */
int32   toktyp[SHELL_MAXTOK];   /* type of each token in tokbuf */
int32   ntok;                   /* number of tokens on line     */
pid32   child;                  /* process ID of spawned child  */
bool8   backgnd;                /* run command in background?   */
char    *outname, *inname;      /* ptrs to strings for file     */
                                /*   names that follow > and <  */
did32   stdinput, stdoutput;    /* descriptors for redirected   */
                                /*   input and output           */
int32   i;                      /* index into array of tokens   */
int32   j;                      /* index into array of commands */
int32   msg;                    /* message from receive() for   */
                                /*   child termination          */
int32   tmpary;                 /* address of this var is used  */
                                /*   when first creating child  */
                                /*   process, but is replaced   */
char    *src, *cmp;             /* ptrs using during name       */
                                /*   comparison                 */
bool8   diff;                   /* was difference found during  */
                                /*   comparison                 */
char    *args[SHELL_MAXTOK];    /* argument vector passed to    */
                                /*   builtin commands           */

/* Print shell banner and startup message */

fprintf(dev, "\n\n%s%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
        SHELL_BAN0,SHELL_BAN1,SHELL_BAN2,SHELL_BAN3,SHELL_BAN4,
        SHELL_BAN5,SHELL_BAN6,SHELL_BAN7,SHELL_BAN8,SHELL_BAN9);

fprintf(dev, "%s\n\n", SHELL_STRTMSG);

/* Continually prompt the user, read input, and execute command */

while (TRUE) {

        /* Display prompt */

        fprintf(dev, SHELL_PROMPT);

        /* Read a command (for tty, 0 means entire line) */

        len = read(dev, buf, sizeof(buf));

        /* Exit gracefully on end-of-file */

        if (len == EOF) {
                break;
        }

        /* If line contains only NEWLINE, go to next line */

        if (len <= 1) {
                fprintf(dev,"\n");
                continue;
        }

        buf[len] = SH_NEWLINE;  /* terminate line */

        /* Parse input line and divide into tokens */
```

```
ntok = lexan(buf, len, tokbuf, &tlen, tok, toktyp);

/* Handle parsing error */

if (ntok == SYSERR) {
        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
        continue;
}
        /* If line is empty, go to next input line */

        if (ntok == 0) {
                fprintf(dev, "\n");
                continue;
        }

        /* If last token is '&', set background */

        if (toktyp[ntok-1] == SH_TOK_AMPER) {
                ntok-- ;
                tlen-= 2;
                backgnd = TRUE;
        } else {
                backgnd = FALSE;
        }


        /* Check for input/output redirection (default is none) */

        outname = innane = NULL;
        if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                        ||(toktyp[ntok-2] == SH_TOK_GREATER)) ) {
                if (toktyp[ntok-1] != SH_TOK_OTHER) {
                        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                        continue;
                }
                if (toktyp[ntok-2] == SH_TOK_LESS) {
                        innane =  &tokbuf[tok[ntok-1]];
                } else {
                        outname = &tokbuf[tok[ntok-1]];
                }
                ntok -= 2;
                tlen = tok[ntok] - 1;
        }


        if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                        ||(toktyp[ntok-2] == SH_TOK_GREATER)) ) {
                if (toktyp[ntok-1] != SH_TOK_OTHER) {
                        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                        continue;
                }
                if (toktyp[ntok-2] == SH_TOK_LESS) {
                        if (innane != NULL) {
                            fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                            continue;
                        }
                        innane = &tokbuf[tok[ntok-1]];
                } else {
```

```
                        if (outname != NULL) {
                            fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                            continue;
                        }
                        outname = &tokbuf[tok[ntok-1]];
                    }
                    ntok -= 2;
                    tlen = tok[ntok] - 1;
                }

                /* Verify remaining tokens are type "other" */

                for (i=0; i<ntok; i++) {
                    if (toktyp[i] != SH_TOK_OTHER) {
                        break;
                    }
                }
                if ((ntok == 0) || (i < ntok)) {
                    fprintf(dev, SHELL_SYNERRMSG);
                    continue;
                }

                stdinput = stdoutput = dev;

                /* Lookup first token in the command table */

                for (j = 0; j < ncmd; j++) {
                    src = cmdtab[j].cname;
                    cmp = tokbuf;
                    diff = FALSE;
                    while (*src != NULLCH) {
                        if (*cmp != *src) {
                            diff = TRUE;
                            break;
                        }
                        src++;
                        cmp++;
                    }
                    if (diff) {
                        continue;
                    } else {
                        break;
                    }
                }

                /* Handle command not found */

                if (j >= ncmd) {
                    fprintf(dev, "command %s not found\n", tokbuf);
                    continue;
                }

                /* Handle built-in command */

                if (cmdtab[j].cbuiltin) { /* no background or redirection */
                    if (inname != NULL || outname != NULL || backgnd) {
                        fprintf(dev, SHELL_BGERRMSG);
                        continue;
                    } else {
```

```
                /* Set up arg vector for call */

                for (i=0; i<ntok; i++) {
                        args[i] = &tokbuf[tok[i]];
                }

                /* Call built-in shell function */

                if ((*cmdtab[j].cfunc)(ntok, args)
                                        == SHELL_EXIT) {
                        break;
                }
        }
        continue;
}

/* Open files and redirect I/O if specified */

if (inname != NULL) {
        stdinput = open(NAMESPACE,inname,"ro");
        if (stdinput == SYSERR) {
                fprintf(dev, SHELL_INERRMSG, inname);
                continue;
        }
}
if (outname != NULL) {
        stdoutput = open(NAMESPACE,outname,"w");
        if (stdoutput == SYSERR) {
                fprintf(dev, SHELL_OUTERRMSG, outname);
                continue;
        } else {
                control(stdoutput, F_CTL_TRUNC, 0, 0);
        }
}

/* Spawn child thread for non-built-in commands */

child = create(cmdtab[j].cfunc,
        SHELL_CMDSTK, SHELL_CMDPRIO,
        cmdtab[j].cname, 2, ntok, &tmparg);

/* If creation or argument copy fails, report error */

if ((child == SYSERR) ||
    (addargs(child, ntok, tok, tlen, tokbuf, &tmparg)
                                        == SYSERR) ) {
        fprintf(dev, SHELL_CREATMSG);
        continue;
}

/* Set stdinput and stdoutput in child to redirect I/O */

proctab[child].prdesc[0] = stdinput;
proctab[child].prdesc[1] = stdoutput;

msg = recvclr();
resume(child);
if (! backgnd) {
        msg = receive();
```

```
                        while (msg != child) {
                                msg = receive();
                        }
                }
        }

        /* Close shell */

562     fprintf(dev,SHELL_EXITMSG);
 :      return OK;
568 }
```

主循环调用 lexan 将输入行分割为一个个符号，并开始处理具体的命令。首先，代码检查用户是否在最后添加了 & 符号。如果是，它设置布尔型变量 backgnd 为 TRUE；否则 backgnd 设置为 FALSE。该变量用来决定命令是否在后台运行。

在后台符号删除之后，壳检查 I/O 重定向。命令中可以指定输入和输出重定向，指定的顺序不限，但是必须在剩下符号的最后。因此，壳会检查重定向次数。如果指定了两个重定向，壳会检查这两个重定向是否都是输入或者都是输出。处理到此处时，壳只保留文件名字指针，而不去试图打开文件（打开文件在之后进行）。

删除指定的 I/O 重定向的符号后，剩下的就是命令名和命令参数。因此，在继续处理命令之前，壳会迭代确认剩余的符号是否是“其他”类型（SH_TOK_OTHER）。如果有不是的，代码输出错误信息并移到下一输入行。检查无误后，壳执行相应的函数来运行该命令。

25.12 命令名查询和内部处理

每一行的第一个符号是命令的名字。记得命令的信息是存储在 cmdtab 数组中，因此可以直接进行顺序查询，查看是否与当前的命令名相匹配。如果没有找到匹配的命令名，代码输出错误信息，并继续处理下一个命令。

我们的壳支持两种命令：内部命令和外部命令。这两种命令的区别在于它们的运行方式不同：壳用传统的函数调用方式运行内部命令，而运行外部命令则创建一个线程来执行。这样的区别也就意味着对于内部命令，用户无法设定其为后台运行并且也无法设定 I/O 重定向[⊖]。

为了检查一个命令是否是内部命令，壳检查数组 cmdtab 中每一项的 cbuiltin 字段。对于内部命令，不允许重定向和后台运行。因此，壳确认用户没有进行这两项设置，然后用 args 创建一个参数表，并
569 调用命令函数。25.13 节介绍如何组织命令参数。

25.13 传给命令的参数

我们例子中的壳参数传递策略与 UNIX 壳采用的策略一样。在调用命令时，壳把从命令行中取得的符号作为未解析的以 null 结尾的字符串传递。壳不知道一个命令需要多少个参数，也不知道传递的参数是否有意义；壳只是负责传递这些参数，然后让命令自己去检查和解析它们。

理论上，壳可以传递任意数量的字符串参数，而参数的数量仅仅受限于输入行的长度。为了让编程简单统一，壳创建了一个指针数组，并且在调用命令时只传递两个参数：一个是参数的个数，一个是参数指针数组。UNIX 将这两个参数命名为 argc 和 argv，Xinu 则叫做 nargs 和 args。这些名字只是一种约定——程序员在编写实现命令的函数时，可以任意给它们起名字。

例子中的壳采用了 UNIX 中的另一个约定：args 数组中的第一项是指向命令名的指针。通过下面的例子可以看到其中的细节。考虑这样一个命令行：

```
date -f illegal
```

尽管参数 illegal 在命令 date 中并不合法，但是壳只是简单地传递这些参数，然后让执行 date 命令的函

⊖ 有一道练习题提示了一种方法，使得内部命令和外部命令之间的区别变得不那么明显。

数对这些参数的合法性进行检查。图 25-4 说明了壳向 date 函数传递的两个参数。

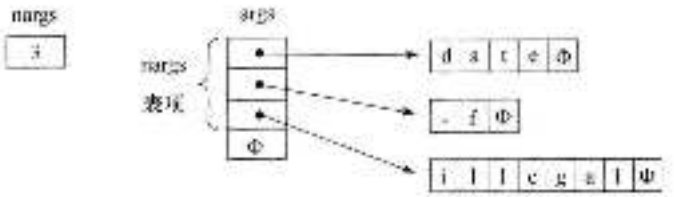


图 25-4　对输入行：date -f illegal，壳传递给 date 命令的两个参数（nargs 和 args）

虽然向命令传递一个整数（如 nargs）是很容易的，但是传递 args 数组则要复杂得多。本质上，壳必须先创建数组 args，然后将它的地址传递给命令。这里有两种情况：内部命令和外部命令。我们先考虑内部命令。

570

先解析完命令行并移除了 I/O 重定向和后台符号之后，变量 ntok 将包含剩下符号的个数，也就是 nargs。而且，数组 tok 中包含每个符号在 tokbuf 中的索引。因此，壳通过计算每个符号的位置可以创建 args 数组。

为了形成 args 数组，代码迭代处理 ntok 个符号，对于第 i 个符号，计算下面的表达式：

```
&tokbuf[tok[i]]
```

也就是说，让 args[i] 等于 tokbuf 中的第 i 个符号的地址。一旦 args 数组初始化后，壳就调用相应的函数来完成内部命令。

25.14　向外部命令传递参数

第二种情况（外部命令）更加复杂。对于此种情况，壳创建一个单独的进程来运行命令，该进程可以运行在后台（比如，壳可以在后台运行命令，同时继续读取和处理键入行）。但问题来了：壳应该用什么样的机制向进程传递参数？壳不能采用内部命令那种方式来传递参数，因为运行在后台的命令需要一份独立的参数副本，这样就可以保证壳继续处理其他命令而不出错了。

有两种方式可以解决外部命令的参数传递问题：壳可以为参数申请独立的内存来存储参数，或者也可以把参数直接存储在进程中已经申请的一块内存中。因为 Xinu 在进程结束时，不会自动释放堆内存，所以第一种方法要求壳记录为每个命令所申请的内存，这样就能够在进程结束时释放这些内存。为此，我们选择第二种方法：

> 在创建了一个运行命令的进程后，壳将参数的副本放在进程的栈区域内，然后让进程运行。

参数应该放在进程栈的什么位置呢？尽管可以重写 create 函数，使栈顶有空间可用，但是这样做也是很复杂的。因此，我们选择使用栈底的空间。壳在栈中存储一份 args 数组的副本，紧跟着是 tokbuf 中字符串的一份副本。当然，tokbuf 中字符串的地址必须赋予 args 副本中的指针。图 25-5 说明了图 25-4 中的数据如何分配到连续的内存区域中。

571

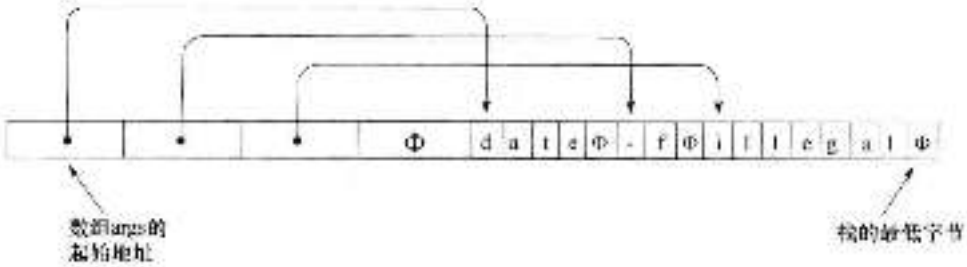


图 25-5　在进程的栈中，args 数组和参数字符串的副本

将参数项复制到进程的栈中的代码并没有加入壳中的，我们用了一个独立的函数 addargs 实现该功

能。文件 addargs.c 包含下述代码。

```
/* addargs.c - addargs */

#include <xinu.h>
#include "shprototypes.h"

/*------------------------------------------------------------------------
 *  addargs  -  add local copy of argv-style arguments to the stack of
 *                a command process that has been created by the shell
 *------------------------------------------------------------------------
 */
status  addargs(
          pid32         pid,            /* ID of process to use         */
          int32         ntok,           /* count of arguments           */
          int32         tok[],          /* index of tokens in tokbuf    */
          int32         tlen,           /* length of data in tokbuf     */
          char          *tokbuf,        /* array of null-term. tokens   */
          void          *dummy          /* dummy argument that was      */
                                        /*   used at creation and must  */
                                        /*   be replaced by a pointer   */
                                        /*   to an argument vector      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        uint32  aloc;                   /* argument location in process */
                                        /*   stack as an integer        */
        uint32  *argloc;                /* location in process's stack  */
                                        /*   to place args vector       */
        char    *argstr;                /* location in process's stack  */
                                        /*   to place arg strings       */
        uint32  *aptr2;                 /* pointer that searches for    */
                                        /*   dummy argument on stack    */
        uint32  *aptr;                  /* walks through args array     */
        int32   i;                      /* index into tok array         */

        mask = disable();

        /* Check argument count and data length */

        if ( (ntok <= 0) || (tlen < 0) ) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];

        /* Compute lowest location in the process stack where the       */
        /*      args array will be stored followed by the argument      */
        /*      strings                                                 */

        aloc = (uint32) (prptr->prstkbase
                - prptr->prstklen + sizeof(uint32));
        argloc = (uint32*) ((aloc + 3) & -0x3); /* round multiple of 4  */

        /* Compute the first location beyond args array for the strings */

        argstr = (char *) (argloc + (ntok+1));  /* +1 for a null ptr    */
```

```
        /* Set each location in the args vector to be the address of    */
        /*      string area plus the offset of this argument            */

        for (aptr=argloc, i=0; i < ntok; i++) {
                *aptr++ = (uint32) (argstr + tok[i]);
        }

        /* Add a null pointer to the args array */

        *aptr++ = (uint32)NULL;

        /* Copy the argument strings from tokbuf into process's stack   */
        /*      just beyond the args vector                             */

        memcpy(aptr, tokbuf, tlen);
        /* Find the second argument in process's stack */

        for (search = (uint32 *)prptr->prstkptr;
             search < (uint32 *)prptr->prstkbase; search++) {

                /* If found, replace with the address of the args vector*/

                if (*search == (uint32)dummy) {
                        *search = (uint32)argloc;
                        restore(mask);
                        return OK;
                }
        }

        /* Argument value not found on the stack - report an error */

        restore(mask);
        return SYSERR;
}
```

一旦建立了进程，进程表项中就包括栈顶地址和栈大小。因为在内存中，栈向下增长，所以 addargs 通过栈顶地址减去栈大小就可以计算出栈地址的最低内存地址。然而，有些细节使得代码变得复杂。例如，因为指针必须字节对齐，所以 addargs 计算的栈的起始地址必须是 4 的倍数。因此，最后一个参数字符串的最后一个字节可能比栈的最低地址的字节超出 3 个字节。而且，在图 25-5 中还可以看到，代码在 args 数组的最后加了一个空指针。

addargs 中的大部分代码还是如预期那样：计算栈的地址，以栈地址为开头复制 args 数组和参数字符串到栈中。然而，最后一个 for 循环似乎看起来有些不正常：查找传递到进程中的第二个参数，然后用 args 数组中的一个指针来替换它。在创建进程后，壳用一个哑值作为参数，将这个值作为 dummy 参数传递到 addargs 中。因此，addargs 查找栈直到找到这个值，然后对其进行替换。

为什么用一个哑参数，然后再查找该参数呢？这样选择是为了让 addargs 计算第二个参数的位置。尽管直接计算看起来更清晰，但是直接计算要求 addargs 了解初始进程栈的结构。用查找的方式意味着只要 create 函数知道进程的细节和栈的结构就可以了。当然，使用查找的方式也有一个缺点：壳必须选择一个哑参数，并且保证该参数值不会在栈中很早出现。我们的壳使用变量 tmparg 的地址，而不是任意一个整数来作为哑值。

572
↑
574

25.15　I/O 重定向

一旦创建了进程并执行命令，解释器就会调用 addargs 来将其参数复制到运行栈中，所有这些都是为了处理输入/输出重定向并开始进程的执行。为了重定向 I/O，解释器将设备描述器分配到进程表项中的进程描述（prdesc）队列中。其中两个关键值是 prdesc［0］和 prdesc［1］，分别被解释器用来设

标准输入（stdin）和标准输出（stdout）。

如何设置变量 stdin 和 stdout 的值？解释器将它们初始化为 dev，当调用完时，需要将设备描述器作为参数传入。通常，设备 CONSOLE 调用完。因此，如果用户没有重定向 I/O，那么执行命令的进程将“继承”控制台设备来进行输入和输出。如果用户重定向了 I/O，那么完将变量 inname 或 outname 按照命令行指定的方式进行设置。否则将 inname 和 outname 设置为 NULL。在为命令进程分配 stdin 和 stdout 之前，完检查 inname 和 outname。如果 inname 是非空值，完调用 open 打开 inname 用于读，并给描述器设置 stdin。类似地，如果 outname 是非空值，完调用 open 打开 outname 用于写入，并给描述器设置 stdout。

描述器应该在什么时候关闭？我们的示例代码假设命令会在其退出前关闭它的标准输入和标准输出描述器。完在命令执行完后不会清空描述器。强制所有命令在退出前关闭它们的标准 I/O 设备有一些缺点，这会使命令难以理解、难以正确编程，因为命令需要记得关闭设备，即使代码中并没有打开它们。

完代码的最后部分是运行命令进程。这里有两种情况。为了在前台中运行进程，完调用 resume 开始这个进程，然后调用 receive 等待进程结束消息（当进程退出时，kill 给完发送一条消息）。对于在后台运行的情况，完启动命令进程，但并不等待。相反，主完继续循环，读取下一条命令。练习题中建议对代码进行一定的修改来提高正确性。

25.16 示例命令函数（sleep）

525 为了理解命令进程参数，考虑函数 xsh_sleep，该函数实现 sleep 命令[⊖]。sleep 会根据自身参数的设定产生几秒的延迟。因此，通过调用 sleep 系统函数，一行独立的代码就可以实现延迟。下面展示的代码仅为说明参数是如何传递，以及命令函数如何输出一条帮助消息。文件 xsh_sleep.c 包含了如下代码。

```
/* xsh_sleep.c - xsh_sleep */

#include <xinu.h>
#include <stdio.h>
#include <string.h>

/*------------------------------------------------------------------------
 * xsh_sleep - shell command to delay for a specified number of seconds
 *------------------------------------------------------------------------
 */
shellcmd xsh_sleep(int nargs, char *args[])
{
        int32   delay;                  /* delay in seconds             */
        char    *chptr;                 /* walks through argument       */
        char    ch;                     /* next character of argument   */

        /* For argument '--help', emit help about the 'sleep' command   */

        if (nargs == 2 && strncmp(args[1], "--help", 7) == 0) {
                printf("Use: %s\n\n", args[0]);
                printf("Description:\n");
                printf("\tDelay for a specified number of seconds\n");
                printf("Options:\n");
                printf("\t--help\t display this help and exit\n");
                return 0;
        }
```

⊖ 根据习惯，实现命令 X 的函数的文件被命名为 xsh_X。

```
/* Check for valid number of arguments */

if (nargs > 2) {
        fprintf(stderr, "%s: too many arguments\n", args[0]);
        fprintf(stderr, "Try '%s --help' for more information\n",
                        args[0]);
        return 1;
}

if (nargs != 2) {
        fprintf(stderr, "%s: argument is error\n", args[0]);
        fprintf(stderr, "Try '%s --help' for more information\n",
                        args[0]);
        return 1;
        }

        chptr = args[1];
        ch = *chptr++;
        delay = 0;
        while (ch != NULLCH) {
                if ( (ch < '0') || (ch > '9') ) {
                        fprintf(stderr, "%s: nondigit in argument\n",
                                args[0]);
                        return 1;
                }
                delay = 10*delay + (ch - '0');
                ch = *chptr++;
        }
        sleep(delay);
        return 0;
}
```

25.17 观点

壳的这种设计实际上很多选择。设计者几乎有完全的自由，因为完成一个位于系统其他部分之外的应用程序一样运行，只有命令语法依赖于壳。因此，与我们示例一样，壳使用的参数传递语式与系统其他部分有着显著的不同。类似地，设计者可以在不影响系统其他部分的情况下，选择输入命令行的语法和语义解释方式。

壳设计最有挑战的地方来自对命令理解能力的决策。一方面，如果壳知道所有命令和它们的参数，壳可以自动填充命令名并检查它们的参数，使实现该命令的代码变得更简单。另一方面，允许延迟绑定意味着更高程度的灵活性，因为在有新命令产生的时候，壳不需要进行改变，但作为权衡，每条命令都必须检查它自己的参数。此外，在 UNIX 系统中，设计者可以选择将每个命令方法编译到壳中或者将每个命令作为一个单独的文件。

在我们的示例中，壳演示了其设计过程中最重要的原则之一：以相对少的代码为用户提供强大的抽象。例如，考虑为实现输入和输出的重定向所需要的最小代码量来识别行尾符号以便在后台执行命令和请求所需要的最小代码量。相比每条命令与用户交互时都提示输入和输出信息或者询问其是否需要在后台运行而言，I/O 重定向和后台处理使壳变得更加强大和用户友好。 [577]

25.18 总结

本章介绍了壳（shell）——一个基本的命令行解释器。尽管示例代码很小，但它支持并发命令执行、输入和输出重定向，以及任意字符串参数传递。其实现在概念上分为两个部分：词法分析，用来读取一行文本并将字符组合成词；壳函数，用来检查词的顺序并执行命令。

这段示例代码演示了用户接口和底层系统所提供设备之间的关系。例如，尽管底层系统支持并发

进程，但壳使用户可以并发执行进程。类似地，尽管底层系统提供打开设备或文件的能力，但壳使用户可以进行 I/O 重定向。

练习

25.1 重写一个壳，使用 cbreak 模式并处理所有的键盘输入。输入序列如 Control-P 会移动到“前一条”指令，Control-B 和 Control-F 分别解释为向后和向前移动一行，可类比 UNIX 中 k-h 或 bash 的实现方式。

25.2 重写图 25-2 中的程序，移除可选符号 [ ]。

25.3 修改壳使其允许内部命令中的 I/O 重定向。需要做哪些必要的修改？

25.4 设计一个改进版的 create，可以使壳处理字符串参数，在创建进程时自动执行相同的方法，如 add-args。

25.5 改进壳，使其可以作为一条命令使用。即允许用户执行命令 shell，从而开启一个新的壳进程。当子壳退出时，将控制权转移到原壳中。请仔细思考其中可能遇到的问题。

25.6 改进壳，使其可以从文件中输入（例如，允许用户建立一个含有命令的文件，之后启动一个壳解释它们）。

25.7 改进壳，使其允许用户像重定向标准输出一样重定向标准错误。

25.8 阅读 UNIX 壳中的壳变量，并在 Xinu 壳中实现一个类似变量机制。

25.9 找出在 UNIX 壳中如何将环境变量传递到命令进程中，并在 Xinu 壳中实现一个类似机制。

25.10 实现内联输入重定向，允许用户输入

```
command << stop
```

随后的输入行被以 stop 字符序列开始的输入终止。令壳将输入保存到临时文件中，使用临时文件作为标准输入来执行命令。

578

25.11 扩展命令表使其包含每条指令所需要的参数的数量和类型，使壳在传递参数到命令前检查它们是否是可行的。列出让壳检查参数的优、缺点，至少各两条。

25.12 假设设计者决定在壳中添加 for 声明，这样用户可以重复执行如下指令：

```
for 1 2 3 4 5 6 7 8 9; command-line
```

这里 for 是关键词，command-line 是与当前壳可以接受命令一样的命令。设计者需要修改壳的语法和解析器吗？或者需要使 for 成为一个内部命令吗？请解释理由。

25.13 为壳添加命令扩展，使用户可以通过 ESC 在命令中输入特定的前缀，使壳输出完整的命令，并等待用户添加参数并按 Enter 键。

25.14 UNIX 允许如下形式的命令行：

```
command_1 | command_2 | command_3
```

这里符号 | ，称为管道（pipe），用于指定一个命令的标准输出与下一个命令的标准输入相连接。为 Xinu 实现一个管道设备，并改进壳使其可以支持指令管道行。

25.15 改进 tty 设备驱动和壳使其可以通过传入 CONTORL_c 杀死当前的执行进程。

25.16 改进 tty 设备驱动和壳使其可以通过传入 CONTORL_z 停止当前的后台执行进程。

25.17 改进设计使得内部命令可以处理 I/O 重定向和后台处理。如果需要重定向或者后台处理，像一般命令一样对待该命令，并创建一个独立的进程。

25.18 本章介绍了在命令退出时关闭设备描述器的问题。改进系统使得 kill 在进程退出时自动关闭进程描述器。

25.19 示例中的壳调用 receive 来等待前台进程结束，但并未检查接收的消息。说明何种事件序列会触发壳在前台进程结束前继续执行。

579

25.20 改进壳代码，修复之前练习中存在的问题。

操作系统移植

进步，绝不是仅由故事组成，而是不重复过去的错误。

——George Santayana

A1.1 引言

在前面的章节中，我们介绍了操作系统的内部实现。我们主要描述了系统的抽象，讨论了设计上的折中方案，说明了怎样将代码组合到层级组织结构中，并且介绍了实现的细节问题。第 24 章讲述了如何配置系统以允许代码可以在有多个外设的系统中运行。

本附录部分主要讨论两个大问题。第一，如何将一个已经存在的操作系统移植到一个新的机器或者一个完全不同的硬件平台上。第二，一个操作系统可以以一种易于移植的方式来编写吗？为了回答第一个问题，附录部分讨论了跨平台开发问题，并提供了一些很实用的意见。为了回答第二个问题，附录将讨论一些可以提升操作系统可移植性的技术。

580

A1.2 动机：硬件的演化

尽管设计操作系统需要抓住高层抽象、设计有效的机制、理解小的细节，但是对操作系统的设计者来说，他们面临的最大挑战不是源于某些知识上的困难。挑战主要源于技术的频繁更新，以及随之而来的供应商为了生产新的产品，或者在已有的产品中添加新的特性而产生的经济压力。例如，在开始修订本书后的 14 个月时间里，硬件供应商就两次改变了模型，而且这种改变很引人注目——处理器芯片、指令集、存储器结构和 I/O 设备全都发生了变化。

因为操作系统是与底层硬件直接交互的，所以即使硬件只有很小的变化，也会对系统产生很大的影响。例如，如果硬件供应商改变硬件以便给闪存（Flash ROM）预留一片内存地址空间，那么操作系统中的内存管理软件必须做相应的修改。尽管这些修改可能不是直接的，但是可能涉及页表、与 MMU 硬件进行交互的代码和按需分配内存的代码。如果内存地址空间的大片地址都被用于预留，则操作系统就需要改变它的地址分配策略。这里的要点是：

> 因为技术和经济上的某些因素引发硬件不断更新，所以操作系统的设计人员必须做好将系统移植到新的平台上的准备。

A1.3 操作系统移植的步骤

无论硬件如何变化，移植一个已有的操作系统到一个新的平台上比从头开始设计和构造一个新的系统简单。特别是，在操作系统是用高级语言实现的情况下，移植这个系统到一个新的平台是非常简单的，因为编译器能做大部分的工作。

以 Xinu 做参考，它的大部分代码都是用 C 实现的。如果在新的平台上能够运行 C 编译器，那么它的源代码中的许多函数不需要修改就能编译。对于那些处理基本数据结构的函数，如整型、字符、数组和结构体，在不需要修改代码的情况下，编译器就能编译，并且生成的二进制程序可以正确运行。即使在需要更改的情况下，源代码也只需要修改很少的部分（比如，编译器之间的不同所造成的调整）。

> 用高级编程语言实现的操作系统（如 C），比用汇编语言实现的系统更容易移植到新的平台上。

581

假设一个操作系统是使用 C 语言实现的，我们来考虑将其移植到新平台上的步骤。特别地，图 A1-1 列出了移植 Xinu 的步骤。

步骤	描述
1	学习新硬件的知识
2	创建交叉开发工具
3	学习编译器的调用规则
4	创建引导机制
5	设计一个基本的结构输出函数
6	加载和运行一个平行程序
7	移植和测试基本内存管理器
8	重写上下文切换和进程创建函数
9	移植和测试剩余进程管理器函数
10	创建一个中断测度器
11	移植和测试实时时钟函数
12	移植和测试 ny 驱动
13	为其他设备要么移植要么创建驱动
14	只要磁盘可用，移植文件系统
15	只要网络驱动可以工作，移植协议软件
16	移植完成者其他应用程序

图 A1-1 移植 Xinu 到一个新平台所需要的步骤

注意，表 A1-1 所列出来的步骤和 Xinu 操作系统层次结构之间的关系。本质上，移植和设计操作系统在模式上是一样的：低层次的结构先移植，然后是高层次的结构。A1.3.1 节将重点介绍每一步。

582

A1.3.1 学习新硬件的知识和编译

步骤 1 可能看起来很直接、很简单。不幸的是，有些供应商不愿意暴露他们商业硬件和软件的细节。即使对于通用信息（比如，处理器指令集），供应商仍会选择将某些细节保密（比如，总线地址空间的映射、硬件初始化顺序，或者设备的细节）。尽管如此，在此附录接下来的部分，我们仍然假设这些需要的信息是可以得到的。

A1.3.2 交叉开发

如果需要移植操作系统的硬件平台已经有了一个可以运行的、具有完整功能的操作系统，那么步骤 1－6 都是可以跳过的，我们可以利用当前已经存在的机制编译和启动一个新的系统。但是，在大部分的情况下，目标平台都是新的，而且可能缺少一个生产系统所需要的能力，这使得操作系统设计者经常能在目标平台上开发。在这种情况下，设计者使用一种交叉开发的方法，利用编译器和链接器来生成目标平台上的代码，而开发工具则运行在一个常规的计算机上。

有一种广泛应用的交叉开发环境是由 GNUC 编译器（GNUCCompiler）gcc 组成。gcc 可以从以下网址免费下载和使用：

http://gcc.gnu.org

下载了 gcc 的源代码之后，程序员必须选择某些配置选项来指定所需的细节，如目标处理器类型和目标机器的字节顺序。程序员运行 UNIX 工具程序 make，创建编译器、汇编器和链接器，进而为目标机器生成代码。

A1.3.3 调用规则

函数调用是操作系统移植的一个重要方面。例如，为了建立上下文切换机制，程序员必须精确地

理解相关函数调用的所有细节。尽管硬件设计者的工作已经包含了子函数的调用机制，但是对于程序员来说，理解硬件并不够，他们还必须要应付编译器带来的附加需求。

使用开源编译器时调用规则可能是显而易见的。但是，操作系统的设计者需要了解各种特例的信息，而这在实际代码中是很难找到的。幸运的是，这些信息可以在网络上找到。 583

A1.3.4 引导机制

在编译和链接之后，程序映像必须下载到目标机器上。早期的嵌入式硬件要求将映像刻录在光盘上，再把光盘插在插槽上。幸运的是，现代系统使用了一些不那么费力的可选机制。在一般情况下，硬件包含的引导功能包括从光盘中读取映像，从控制台串行线路接收映像和通过网络下载映像。不过，引导的步骤不被一般人知道。

以 Linksys E2100L 无线路由器为例，访问引导装载程序不大可能，除非你打开包装盒，连接上一个串行线路，并在启动序列的过程中通过串行线路发送字符。只要正常的引导序列被中断，引导装载程序就会显示一个提示，提供许多可以下载映像的途径（比如，通过控制台串行线路可以下载多种格式的映像，还有通过以太网接口下载网络引导映像等）。无论选择哪种方法，找到一种方法将映像的备份放到目标机器的内存中都是必要的。

A1.3.5 轮询输出

移植操作系统的下一步需要程序员设计一种方法用来运行输出字符的程序。在某些基本输入/输出设备可用之前，程序员都必须在没有提示的情况下工作，只能寄希望于映像成功下载并开始。所以，基本的输入/输出是非常宝贵的：一旦基本输入/输出可用，程序员就可以很快地判断程序运行到哪里，而且能很快地找到问题所在。

因为早期的测试程序并没有包含中断处理，所以基本输入/输出必须使用轮询机制。即程序员创建一种类似于 kputc 的方法，用来等待输入/输出设备准备好的时候传输字符。两边的终端都必须在某些细节上（如波特率和每个字符的位数）保持一致。如果不这样做的话，就会使程序员的调试过程变得冗长而低效。为了简化代码，kputc 的第一个版本可以通过汇编语言来实现，并且可以将设备的某些信息（比如，控制和状态寄存器的地址和波特率）直接写在程序中。

A1.3.6 串行程序的执行

一旦一个映像可以下载并且运行，那么下一步就是建立可以运行串行程序的环境。特别地，一个 C 程序的成功运行需要正确的内存访问机制设置（程序的文本可以读，而且数据的位置可以被读出和写入）和一个运行时的栈（这是函数调用所需要的）。

初始化这个环境可能看起来很简单，但是它需要知道许多硬件的细节。例如，在 E2100L 上，物理内存是在地址空间中复制的。为了给一个高的物理内存地址分配栈指针，仅仅确定某个特定地址是可用的是不够的，你还得明白每个地址是怎样与物理地址相对应的。 584

A1.3.7 基本的内存管理

一旦内存的分布知道了，并且串行程序可以下载和运行在目标硬件上，那么程序员就可以移植和测试 4 个基本的内存管理函数：getmem、freemem、getstk 和 freestk。另外，对于基本的分配内存和释放内存程序，程序员需要专注在地址空间中的对齐上。有些硬件平台要求所有的内存访问是字对齐的，而另一些则不需要。在需要对齐的机器上，程序员应该保证内存释放链表以一种能使对齐正常工作的方式被初始化（例如，所有分配的块都从适当的边界上开始）。

A1.3.8 进程创建和上下文切换

一旦基本的内存管理工作以后，程序员就可以开始移植进程管理函数了。特别地，程序员可以开始移植上下文切换、调度和进程创建函数。这三个基本进程管理函数的移植成功，是整个移植过程的

重大步骤：与串行程序不同，这时已经形成了操作系统的雏形，它能够支持并发执行。

这一步骤有两个困难的部分。创建一个进程的保存信息需要了解关于机器状态和上下文切换操作的错综复杂的信息。而建立上下文切换机制是一个难点，因为它需要寻找一种能保存当前进程的所有状态信息，并且重新加载另一个进程的所有状态信息的方法。另外在保存备份的时候，忽略掉细节或者不经意间破坏状态信息很容易发生（比如，更改一个寄存器）。这让调试变得极度地困难，因为错误可能隐藏到在系统尝试重新加载存储的状态信息时。

A1.3.9 同步和其他的进程管理函数

一旦进程创建、调度和上下文切换正常工作，那么其他的进程管理函数就可以很简单地添加进来。信号量函数和消息传递函数都可以移植和测试了。除了上下文切换外，大部分的进程管理函数都不依赖于硬件。当然，各个数据结构都可能会发生改变，而这依赖于最下层的硬件。例如，当将操作系统从32位移植到64位的机器时，msg32这个数据类型可能会改变为msg64。无论如何，移植信号量函数
585 和消息传递函数都是一个相对比较简单的任务。

A1.3.10 中断调度和实时时钟

移植过程中，最大的硬件阻拦来自中断。创建一个中断调度需要对硬件有一个细致的了解。处理器、协处理器和总线是怎样交互的？当一断发生的时候硬件保存的是什么状态信息？什么状态信息是操作系统要求保存的？调度器怎么样判定哪个设备发生了中断？在中断结束的时候，调度器又怎样可到原来运行的程序？什么地址用于总线和设备？

中断细节的实现非常微妙。在许多机器上，输入/输出都是内存映射的，输入/输出设备（也许还有总线硬件）被映射到特定的地址。而为了访问到输入/输出设备，操作系统可能需要禁用或者避免使用内存缓存，因为输入/输出必须访问底层的硬件而不是缓存。

在任何情况下，一旦中断调度移植成功，就需要一个例子来测试这个机制。首先测试实时时钟合乎逻辑。在某些系统上，首先建立实时时钟处理程序是必需的，因为时钟不能停止。如果系统启用了中断，那么时钟中断将会出现。时钟中断意味着进程可以调用sleep()来延迟一段特定的时间和使时间切片有效。

A1.3.11 tty设备驱动

时钟中断和其他设备不同，因为其没有输出。串行线路可能是同时拥有输入和输出功能的最简单的设备（有些硬件将输入和输出中断处理分开）。因此，tty驱动需要同时测试输入和输出，保证所有的基本中断处理正常运行。

幸运的是，大部分系统都包含了串行线路，并且许多都使用相同的UART硬件。因此，tty驱动的许多代码（包括下半部）都可以简单地重新编译和使用。而步骤5中的基本设备参数都可以适当地添加到设备转换表或者下半部分中。

A1.3.12 其他的输入输出软件和设备驱动

一旦输入和输出通过了测试，更复杂的设备驱动就可以移植进来，如使用DMA的设备（比如，磁
586 盘或网络接口）。

A1.3.13 文件系统

对于可用的磁盘驱动，移植一个基本的文件系统是直接简单的。建议第一步移植和测试那些能读和写索引块的函数；第二步移植和测试那些建立索引块和数据块的代码。一旦基本的操作就位，就可以将目录添加进来，而整个文件系统也可以开始测试了。

A1.3.14 协议软件

对于基本的网络设备驱动，我们可以通过传输和接收未加工的数据包来进行测试（比如，在以太

网上）。但请注意，测试网络连通性和加载时的性能经常需要在高层次的协议软件上进行，至少在互联网协议和用户数据报文协议上。在用户数据报协议就位后，程序员就可以在通用系统上运行一个发送和接收数据包的应用程序。

A1.3.15　壳和应用程序

一旦系统可以运行了，最后的一步就是移植一个壳和其他通用的程序。

A1.4　适应变化的编程

操作系统的设计者怎么样才能和不断的变化做斗争？他们能够预知未来的硬件吗？一个系统能否设计和实现得适应未来的变化？设计者在这些问题上花费了数十年。大部分的早期操作系统都是针对特定硬件来设计，并且是用汇编语言实现的，且每个系统都从头开始设计和生成。当输入/输出设备（例如，磁盘）和操作系统抽象（例如，文件）实现了标准化后，从头开始设计一个新的系统变得比修改现有操作系统更加耗时费力。现在，先进的操作系统采用两种技术来适应改变：

- 编译时（compile time）：编写能生成不同版本的源代码。
- 运行时（run-time）：允许操作系统动态地改变。

编译时　使系统具有适应性的一种方法是，使用条件编译指令编写源代码，这样能够使给定的源程序在多个系统上执行。举一个简单的例子，考虑实现一个既能在拥有实时时钟的硬件上运行，又能在没 587
有实时时钟的硬件上运行的操作系统。程序员可以根据硬件利用 C 预处理器来条件性地编译源代码。例如，如果预处理器变量 RT_CLOCK 已经定义，那么使用了实时时钟的函数就按照通用情况进行编译。其他情况下，依赖于时钟的函数就会被报告错误的版本替换掉。在第 13 章中的 sleep 函数可以用来阐述这种概念。为了适应这两种情况，sleep 代码可以写成下面的形式：

```
syscall sleep(
        cint32          delay           /* time to delay in seconds     */
        )
{
#ifdef RT_CLOCK
        if (delay > MAXSECONDS) {
                return(SYSERR);
        }
        slnap(ms(1000*delay);
        return OK;
#else
        return SYSERR;
#endif
}
```

如果变量 RT_CLOCK 已经定义，那么 C 预处理器就按照第 13 章所述生成源代码。如果 RT_CLOCK 没有定义，那么 C 预处理器将会去掉 sleep 函数的大部分，只生成一行源代码：

```
return SYSERR;
```

条件编译的主要优势在于高效性：与运行时进行测试判断不同，生成的源代码实际上是为指定的硬件量身定做的，这样的系统并没有包含不会被使用的代码（这一点在嵌入式系统中是非常重要的）。

运行时　最简单的增加运行时可移植性的方法就是使用条件执行。当操作系统启动的时候，它先收集硬件相关的信息，并将其封装在一个全局数据结构中，而这个全局数据结构中的某个布尔变量可能指定这个硬件是否拥有实时时钟。每个操作系统函数都先询问这个全局数据结构，然后按照得到的信息采取相应行动。这种方法的最大优势就在于通用性——映像可以在不经过重新编译的情况下在新硬件上运行。

现在，比种动态适应方法一般是将操作系统分成两个部分：一个包含基本进程管理函数的微核（microkernel）和一系列扩展功能的动态加载模块。在理论上，移植一个微核到一个新的环境下比直接移植整个系统简单，因为这个移植过程可以一块一块地完成。即先移植微核，然后移植所需要的模块。 588

A1.5 总结

由于硬件一直在发生变化，所以系统的移植性是非常重要的。移植一个操作系统到一个新环境下的步骤与初始系统设计的模式是一样的：先移植低层次的系统，然后移植系统中的高层次部分。

有一些办法可以提高操作系统代码的移植性。运用条件性编译的编译时方法具有最高的效率。而
589 运用条件性执行的运行时方法允许映像运行在多种版本的平台上。

Xinu 设计注解

A2.1 引言

本附件包含一系列非正规的关于 Xinu 的特点和基本设计的注解。这些注解并不是教程，也不是关于这个系统的完整描述。相反，它们只是关于这些特性和功能的简明总结。

A2.2 概括

嵌入式的范式 由于嵌入式系统的需要，Xinu 遵循交叉开发范式。程序员使用常规的计算机（一般是使用一种 UNIX 操作系统，例如 Linux）来编写、编辑、交叉编译和交叉链接 Xinu 软件。交叉开发软件的输出结果就是一个内存中的映像。一旦这种映像被创建，程序员需要将它下载到目标系统上（通常是通过计算机网络）。最后，程序员启动运行在目标系统上的这个映像。

源代码组成 Xinu 软件源代码由少量的，遵循 UNIX 系统组织风格的目录构成。与将每个模块的所有文件放在一个单独目录相反，所有文件只是被分组到少量的几个目录中。例如，所有的 include 文件放在一个目录下，所有构成 kernel 的源代码放在另一个目录中。不过设备驱动代码例外，每个设备驱动代码文件放在以这个设备类型命名的子目录中。这些目录组织如下：

compile	包含编译和链接一个映像所需指令的 Makefile。
include	所有的 include 文件。
config	配置程序的代码以及生成和安装 conf.h 和 conf.c 的 Makefile。
system	Xinu kernel 函数的源代码。
devices	设备驱动的源代码，每个设备类型分到一个子目录中。
tty	tty 驱动的源代码。
rfs	远程文件访问系统的源代码，包括主远程文件系统设备和远程文件伪设备。
ether	以太网驱动的源代码。

A2.3 Xinu 设计注解

Xinu 特性 下面是 Xinu 实现中需要的注意事项：

- 系统支持多个并发进程。
- 每个进程通过它的进程 ID 识别。
- 进程 ID 是进程表中的索引。
- 系统包含信号量。
- 每个信号量通过它的 ID 识别，这个 ID 是信号量表中的索引。
- 系统支持实时时钟，用于同等优先级进程之间的循环调度和计时延迟。
- 给每个进程分配一个优先级，用在进程调度中。进程优先级是可以动态修改的。
- 系统支持多个输入/输出设备，也支持多种输入/输出设备。
- 系统包含一系列不依赖于设备的输入/输出原语。
- 控制台设备使用 tty 抽象，在这个抽象中，字符在输入和输出队列中。
- tty 驱动支持多种模式；cooked 模式包含字符回显、删除、回退等。
- 系统包含一个可以发送和接收以太网数据包的以太网驱动，这个驱动使用 DMA 技术。
- Xinu 包含一个本地文件系统，它支持在没有预分配空间的情况下的并发增长，该本地文件系统只拥有一个单一层次的目录结构。

- Xinu 还包含了一种允许通过远程服务器访问服务器上文件的机制。
- 系统包含一种用于进程间通信的消息传递机制；每个消息只有一个字长。
- 进程是动态的。进程可以被创建、挂起、重启或终止。
- Xinu 包含一个低层次的用于分配和释放堆区域和进程栈的内存管理器，和一个高层次的用于创建缓冲区池的内存管理器。这个缓冲区池包含了一系列固定大小的缓冲区。
- Xinu 包含一个配置程序，能够根据指定的信息生成 Xinu 系统。这个配置程序允许用户选择一系列的设备并设置系统参数。

A2.4 Xinu 实现

函数和模块 系统的源代码由一系列函数组成。一般情况下，每个文件对应于一个系统调用（例如，文件 resume.c 包含系统调用 resume）。除了系统调用函数外，文件还可能包含系统同步所需要的实用函数。其他的文件列在下面：

Configuration	一个包含了设备信息和描述系统和硬件常量的文本文件。config 程序使用文件 Configuration 作为输入，生成 conf.c 和 conf.h。
conf.h	由 config 程序生成，它包含声明和定义输入/输出设备的名字常量，例如 CONSOLE。
conf.c	由 config 程序生成，它包含设备转换表的初始化信息。
kernel.h	贯穿整个 kernel 核使用的通用符号常量和类型声明。
prototypes.h	所有系统函数的原型声明。
xinu.h	主要的 include 文件，以正确的顺序包含所有的头文件。大部分的 Xinu 函数只需要包含 xinu.h。
process.h	进程表表项结构声明，状态常量。
semaphore.h	信号量表表项结构声明，信号量常量。
tty.h	tty 线路规程控制块、缓冲区。
bufpool.h	缓冲区池常量和格式。
memory.h	低层次内存管理使用的常量和结构。
ports.h	高层次进程间通信机制的定义。
sleep.h	实时时钟延迟函数的定义。
queue.h	适用的进程队列处理函数的声明和常量。
resched.c	选择下一个进程运行的 Xinu 调度器；resched 需要调用上下文切换程序 ctxsw。
ctxsw.S	从一个执行中的进程切换到另一个进程的上下文切换功能。它包含一小段汇编代码，这段汇编代码用了一点儿技巧：当进程的状态信息处于保存状态时，进程重新启动的执行地址就是紧跟着 ctxsw 的指令地址。
initialize.c	通用的初始化代码和空进程（进程0）的代码。
userret.c	当进程退出时进程返回到的函数。userret 从来不会返回。

592

A2.5 主要的概念和实现

进程状态 每个进程在它的进程表表项中都有一个 prstate 字段用于存储其状态。定义进程状态信息的常量拥有 PR_xxxx 这样一种形式。其中，PR_FREE 意味着进程表项并没有被利用。PR_READY 意味着进程在准备链表中，可以被 CPU 调用。PR_WAIT 意味着进程正在等待着信号量（由 preem 提供）。PR_SUSP 意味着进程处在挂起状态，这时它不在任何一个链表中。PR_SLEEP 意味着进程在睡眠进程的队列中，在超时后将会被唤醒。PR_CURR 意味着进程正在运行。正在运行的进程不在准备链表中。PR_RECV 意味着进程阻塞正在等待接收信息的状态。PR_RECTIM 是定时阻塞等待，当计时器超时或者信息到达的时候，无论哪个先发生，它都会醒来。

593

信号量 信号量位于数组 semtab 中。数组中的每个元素对应一个信号量，每个信号量拥有一个计数值（scount）和状态信息（sstate）。如果信号量没有被赋值，那么其状态是 S_FREE；否则，就是

S_USED。如果这个计数值是负 P，那么信号量表中表项的头和尾指向等待该信号量的 P 个进程组成先进先出队列的头和尾。如果这个计数值是非负 P，那么没有进程正在等待。

阻塞的进程　无论哪种原因造成阻塞的进程都不会有使用 CPU 的资格。任何阻塞当前运行进程的操作都会迫使它放弃 CPU 的拥有权，并让其他的进程运行。阻塞在信号量上的进程处在与这个信号量有关的队列中。由于时间延迟造成的进程阻塞是在睡眠进程队列中。其他阻塞的进程则不在任何队列中。ready 函数将阻塞进程移到准备链表中。

睡眠进程　进程调用 sleep 函数来延迟一段指定的时间。进程会被添加到睡眠进程链表中。进程只能睡眠自己。

进程队列和有序链表　Xinu 系统有一个单独的数据结构用于所有的进程链表。这个结构包含每个链表的头和尾的表项。其中第一个 NPROC 表项（0 ~ NPROC − 1）对应系统中的 NPROC 个进程，接下来的表项是成对分配的，每对意味着一个链表的头和尾。

将所有链表的头和尾保存在一个数据结构中的优点就在于，入队、出队、测试是否为空，以及从中间删除（例如，当进程被终止时）等功能将只被一小部分函数控制（文件 queue.c 和 queue.h 中）。空队列的头和尾是指向对方的。因此测试一个链表是否为空非常简单。链表可以是顺序的，也可以是先进先出。如果链表是先进先出的，那么每个表项的键是可以忽略的。

空进程　进程 0 是一个随时准备运行或者正在运行的空进程。注意，进程 0 从来不会运行导致它阻塞的代码（例如，它不能等待信号量）。因为空进程在中断的时候可能运行，所以中断代码也不能等待信号量。当系统启动的时候，初始化代码创建一个进程来执行 main 函数，然后就变成一个空进程（执行无限循环）。因为它的优先级比其他进程低，所以空进程只在没有其他进程处于准备状态的时候才会运行。

[594]

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

activation record（活动记录），8
addargs.c，572
addition in exb.c（在 exb.c 中的添加项目），25
address（地址）
　space（空间），169
　translation hardware（转换硬件），171
Address Resolution Protocol（地址解析协议），334
AG71xx Ethernet（AG71xx 以太网），305
ag71xx.h，306
allocRxBuffer.c，319
API（应用程序编程接口），7
Application Program Interface（应用程序编程接口），7，9
ARP（地址解析协议），334
　cache（缓存），336
　request（请求），336
　response（响应），336
arp.c，338
arp.h，337
arp_alloc in arp.c，338
arp_in in arp.c，338
arp_init in arp.c，338
arp_resolve in arp.c，338
asynchronous（异步），242
　event handler（事件处理程序），12
　event paradigm（事件范式），216
Atheros AG71xx Ethernet，305

B

backplane（主板），32
backspace（回退），295
Basic Input Output System（基本输入输出系统），6
BIOS，6
block（块），372
block-oriented device（面向数据块的设备），371
booting E2100L，523
bootstrap（引导程序），522，523
bss segment（bss 段），44，142
buffer（缓冲区），270
　allocation（分配），162
　pool（池），160
　pool creation（池创建），165
　pool initialization（池初始化），167
　release（释放），164
　ring（环），304
bufinit in bufinit.c（bufinit.c 中的 bufinit），168
bufpool.h，162
busy（忙）
　wait（等待），22
　waiting（等待），113
byte（字节），35

C

carriage return（回车），269
chreak mode（chreak 模式），268，269，288
character（字符），35
character echo（字符回显），268
chprio，106
chprio in chprio.c（chprio.c 中的 chprio），107
CLI（命令行界面），6
clkcount in clkupdate.S（clkupdate.S 中的 clkcount），234
clkhandler in clkhandler.c（clkhandler.c 中的 clkhandler），231
clkinit in clkinit.c（clkinit.c 中的 clkinit），232
clkupdate in clkupdate.S（clkupdate.S 中的 clkupdate），234
clock（时钟）
　hardware（硬件），216
　initialization（初始化），531
　interrupt（中断），231
　tick（滴答），219
close（关闭），243
close in close.c（close.c 中的 close），256
cold start（冷启动），522
Command Line Interface（命令行接口），6
command-line syntax（命令行语法），552
compare-and-swap（比较和交换），125
concurrent（并发）
　execution（执行），13，15
　processing（处理），13
conf.c，259
conf.h，247
config program（config 程序），543
configuration（配置），245，541
Configuration file（配置文件），543

configuration static（静态配置），542
cons2 in ex5.c（ex5.c 中的 cons2），23
CONSOLE，7
consume in ex4.c（ex4.c 中的 consume），21
consumer（消费者），113
context（上下文），499
context switch（上下文切换），67，78
control（控制），243
control in control.c（control.c 中的 control），251
  block（块），273，545
  tty driver（tty 驱动），297
control.c，251
cooked mode（cooked 模式），268，288
coordination（协作），111
copyhandleer in start.S（start.S 中的 copyhandleer），524
count-up timer（自增计时器），217
counter（计数器），42
counting semaphore（计数信号量），112
cr，269
create（创建），17
create in create.c（create.c 中的 create），102
creation process（创建进程），101
critical section（修界区），24
crlf，269
ctxsw in ctxsw.S（ctxsw.S 中的 ctxsw），78
current state（当前状态），72
currpid，74

D

d-block（d-模块），414
data（数据）
  area（区域），413
  block（模块），414，415
  segment（段），44，142
  sheet（表），330
date（日期），216
deadlock（死锁），160
default router（默认路由器），334
deferred rescheduling（推迟重新调度），83
deferred rescheduling in sched_cntl.c（sched_cntl.c 中的推迟重新调度），84
delay（延迟），218
delta list（增量链表），220，221
dequeue in queue.c（queue.c 中的 dequeue），58
descriptor（描述符），245
design notes（设计注解），590
device（设备）
  descriptor（描述符），245，250
  driver（驱动），240，267
  initialization（初始化），531
  management（管理），240
  switch table（转换表），245，426，545
destab in conf.c（conf.c 中的 destab），259
destab in conf.h（conf.h 中的 destab），247
DHCP，334
dhcp.c，364
Direct Memory Access（直接内存访问），34，303，372
directory（目录），411，413
disable（禁止），41，95
disjunctive wait（分离等待），226
disk（磁盘），371
  driver（驱动），371
  initialization（初始化），381
  partition（分区），413
dispatch in dispatch.c（dispatch.c 中的 dispatch），205
dispatcher（分配器），199
DMA（直接内存访问），34，303，372
driver（驱动），267，303
dynamic（动态的）
  device drivers（设备驱动），542
  file（文件），413
  semaphore allocation（信号量分配），119
Dynamic Host Configuration Protocol（动态主机配置协议），334

E

E2100L，31
Echo（响应）
  Reply（ICMP）（应答（ICMP）），335
  Request（ICMP）（请求（ICMP）），335
echo（回显），268
  reply（应答），362
  request（请求），362
EMPTY，54
end-of-file（文件结束），438
enqueue in queue.c（queue.c 中的 enqueue），58
ethControl in ethControl.c（ethControl.c 中的 ethControl），328
ether.h，310
Ethernet（以太网），303
ethInit in ethInit.c（ethInit.c 中的 ethInit），314
ethInterrupt in ethInterrupt.c（ethInterrupt.c 中的 ethInterrupt），324
ethRead in ethRead.c（ethRead.c 中的 ethRead），320
ethWrite in ethWrite.c（ethWrite.c 中的 ethWrite），322
event paradigm（事件范式），216
ex1.c，7
ex2.c，16
ex3.c，18
ex4.c，21
ex5.c，23

exit.c, 25
ex7.c, 588
exception (异常), 40, 535
exception handling (异常处理), 535
exit (退出), 98
exit of a process (进程退出), 19

F

fetch (获取), 372
fetch-store paradigm (获取－存储范式), 34
FIFO (先进先出), 57
file (文件), 411
  data block (数据块), 414
  index block (索引块), 414
  local (本地的), 413
  name (名称), 412, 497, 499
  pseudo-device (伪设备), 426
  system (系统), 411
  system organization (系统组织), 415
file name (文件名)
  UNIX, 499
firmware (固件), 6
first-fit memory allocation (最先适配内存分配), 148
fmtid in queue.h (queue.h 中的 fmtid), 53
fmtkey in queue.h (queue.h 中的 fmtkey), 53
flow control (流控制), 268
folder (文件夹), 411
fragmentation of memory (内存碎片), 145
frame (页帧), 172
free memory (空闲内存), 146
freebuf in freebuf.c (freebuf.c 中的 freebuf), 164
freemem in freemem.c (freemem.c 中的 freemem), 154
freestk in memory.h (memory.h 中的 freestk), 148

G

gcc, 583
general-purpose register (通用寄存器), 33
getbuf in getbuf.c (getbuf.c 中的 getbuf), 163
getfirst in getitem.c (getitem.c 中的 getfirst), 56
getitem in getitem.c (getitem.c 中的 getitem), 56
getlast in getitem.c (getitem.c 中的 getlast), 56
gethostaddr in dhcp.c (dhcp.c 中的 gethostaddr), 364
getmem in getmem.c (getmem.c 中的 getmem), 149
getpid, 20, 106
getpid in getpid.c (getpid.c 中的 getpid), 106
getprio, 106
getprio in getprio.c (getprio.c 中的 getprio), 106
getstk in getstk.c (getstk.c 中的 getstk), 152
global clock (全局时钟), 174
Gnu C Compiler (Gnu C 编译器), 583
granularity of preemption (抢占粒度), 238

H

heap (堆), 143
heavyweight process (重量级进程), 140
hierarchy (层次结构), 4

I

i-blocks (i 块), 414
I/O (输入/输出), 195, 240
  interface (接口), 241
  redirection (重定向), 551, 575
ih2disp in ifilesys.h (ifilesys.h 中的 ih2disp), 416
ih2sect in ifilesys.h (ifilesys.h 中的 ih2sect), 416
IDES, 499
ICMP, 335, 362
icmp_in, 362
icmp_init, 362
icmp_out, 362
icmp_recv, 362
icmp_register, 362
icmp_release, 362
icmp_send, 362
ID (标识符)
  buffer pool (缓冲区池), 161
  process (进程), 68
identifier (标识符), 183
include file (include 文件), 7
index (索引)
  area (区域), 413
  block (块), 414, 415, 419
  block operations (块操作), 420
  mechanism (机制), 413
init, 243
init in init.c (init.c 中的 init), 254
Initial Program Load (Initial 程序加载), 522
initialization (初始化), 521
initialize.c, 527
input (输入), 195, 239
insert in insert.c (insert.c 中的 insert), 61
instruction pointer (指令指针), 196
intdispatch in intdispatch.S (intdispatch.S 中的 intdispatch), 203
inter-process communication (进程间通信), 179
interface abstraction (接口抽象), 241
Internet (互联网)
  Control Message Protocol (控制报文协议), 335, 362
  Protocol (协议), 334
  Service Provider (服务提供者), 525
  protocols (协议), 333

interrupt（中断），40，93，195
dispatcher（分配器），199
handler（处理程序），199，200
mask（屏蔽），40
multiplexing（多路复用），199
request（请求），197
sp，283
vector（向量），197
interrupt-driven I/O（中断驱动的 I/O），43
interval timer（间隔计时器），217
ioctl，257
ioctl in ioctl.c（ioctl.c 中的 ioctl），258
ionull，257
ionull in ionull.c（ionull.c 中的 ionull），258
IP（互联网协议），334
ipcksum in netin.c（netin.c 中的 ipcksum），348
IPL（初始化装载），522
IRQ（中断请求号），197
isbadpid in process.h（process.h 中的 isbadpid），70
isbadport in ports.h（ports.h 中的 isbadport），181
isbadsem in semaphore.h（semaphore.h 中的 isbadsem），116
isempty in queue.h（queue.h 中的 isempty），53
ISP（网络服务提供者），523

K

kernel（内核），1，2
mode（模式），45
space（空间），35
kill，20
kill in kill.c（kill.c 中的 kill），99
killing a process（杀死一个进程），98
kprintf，43
kputc，43，584

L

last-fit memory allocation（最后匹配的内存分配），151
last-write semantics（最后写入语义），374
lastkey in queue.h（queue.h 中的 lastkey），53
lcasex.c，557
lf，269
lfdballoc in lfdballoc.c（lfdballoc.c 中的 lfdballoc），424
lfdbfree in lfdbfree.c（lfdbfree.c 中的 lfdbfree），425
lflflush in lflflush.c（lflflush.c 中的 lflflush），436
lfgetmode in lfgetmode.c（lfgetmode.c 中的 lfgetmode），433
lfiballoc in lfiballoc.c（lfiballoc.c 中的 lfiballoc），423
lfibclear in lfibclear.c（lfibclear.c 中的 lfibclear），420
lfibget in lfibget.c（lfibget.c 中的 lfibget），421
lfilk，415
lfibput in lfibput.c（lfibput.c 中的 lfibput），422
lfilesys.h，416
lfiClose in lfiClose.c（lfiClose.c 中的 lfiClose），435
lfiGetc in lfiGetc.c（lfiGetc.c 中的 lfiGetc），440
lfiInit in lfiInit.c（lfiInit.c 中的 lfiInit），449
lfiPutc in lfiPutc.c（lfiPutc.c 中的 lfiPutc），442
lfiRead in lfiRead.c（lfiRead.c 中的 lfiRead），438
lfiSeek in lfiSeek.c（lfiSeek.c 中的 lfiSeek），439
lfiWrite in lfiWrite.c（lfiWrite.c 中的 lfiWrite），437
lfscreate in lfscreate.c（lfscreate.c 中的 lfscreate），452
lfsetup in lfsetup.c（lfsetup.c 中的 lfsetup），444
lfslnit in lfslnit.c（lfslnit.c 中的 lfslnit），448
lfsOpen in lfsOpen.c（lfsOpen.c 中的 lfsOpen），428
lftruncate in lftruncate.c（lftruncate.c 中的 lftruncate），450
lightweight process abstraction（轻量级的进程抽象），140
limit（限制），42
linedisc（换行），269
linked list（链表），52
Linksys，31
loadb，523
local file（本地文件），413
lower half（下半部），240，269

M

main in ex1.c（ex1.c 中的 main），7
main in ex2.c（ex2.c 中的 main），16
main in ex3.c（ex3.c 中的 main），18
main in ex4.c（ex4.c 中的 main），21
main in ex5.c（ex5.c 中的 main），23
main program（主程序），15
mask（屏蔽），40
mdelay in ethInit.c，314
memlist，146
memory（内存）
allocation（分配），160
fragmentation（分片），145
management（管理），139，159
partitioning（分区），160
segment（段），44
memory-mapped I/O（内存映射的 I/O），34
memory.h，148
memzero in start.S，524
message（报文）
passing（传递），129，131，179
sending（发送），132
sending to a port（发送到一个端口），184
microkernel（微内核），2，588
minor device number（次设备号），546
mkbufpool in mkbufpool.c（mkbufpool.c 中的 mkbufpool），546
mode（模式），45
mode of tty（tty 的模式），268
monitor（监视器），1

mount in mount.c（mount.c 中的 mount），503
MS_DOS，499
multi-level hierarchy（多层次结构），4
MULTICS（符号常量），174
multiprogramming（多道程序设计），14
mutual exclusion（互斥），24，112

N

name mapping（名字映射），500
name.h，502
namespace（名字空间），497，498
  abstraction（抽象），497
  initialization（初始化），510
NAMESPACE device（NAMESPACE 设备），502
naminit in naminit.c（naminit.c 中的 naminit），511
nammap in nammap.c（nammap.c 中的 nammap），505
namopen in namopen.c（namopen.c 中的 namopen），510
NDEVS，545
net.h，346
netin in netin.c（netin.c 中的 netin），348
network byte order（网络字节序），472
NEWLINE，269
newpid in create.c（create.c 中的 newpid），102
newqueue in newqueue.c（newqueue.c 中的 newqueue），63
newsem in semcreate.c（semcreate.c 中的 newsem），120
non-selfreferential（非自引用），600
nonempty in queue.h（queue.h 中的 nonempty），53
nonvolatile storage（非易失存储器），371
NPROC，51，68
NQENT，54
null process（空进程），82
nulluser in initialize.c（initialize.c 中的 nulluser），527

O

open（打开），243，509
open in open.c（open.c 中的 open），256
open-read-write-close（打开－读－写－关闭），243
open.c，256
operating system（操作系统），1
output（输出），195，239

P

paging（分页），169
panic，186
panic.c，537
paradox（悖论）
  参见 non-selfreferential
partitioning memory（分区内存），160
path name（路径名），499
pattern string（模式字符串），501
physical address space（物理地址空间），169
ping，362
pipe（管道），579
polled I/O（轮询 I/O），27，43，584
polling（轮询），130
pool identifier（池 ID），161
port（端口），183
  creation（创建），183
  deletion（删除），188
  number（数量），334
  reception（接收），186
  reset（重置），188
  table initialization（表初始化），181
ports（端口），179
ports.h，181
preemption（抢占），218
preemption granularity（抢占赠粒度），218
prefix（前缀）
  pattern（模式），501
  property（属性），499
priority（优先级），60
  inversion（倒置），114
  queue（队列），60
process（进程），14，50，531
  ID（ID），17，50，68，106
  参见 thread
  blocking（阻塞），23
  coordination（协调），111
  creation（创建），17，101
  execution（运行），17
  exit（退出），19，98，101
  heavyweight（重量级），140
  management（管理），5
  priority（优先级），60，106
  ready state（准备状态），72
  state（状态），71，72，223
  synchronization（同步），111
  table（表），68
  termination（终止），20，98
  waiting（等待），115
Process（Linux）（进程），15
process.h，70
processor clock（处理器时钟），216
proctab，68
prod2 in ex5.c（ex5.c 中的 prod2），23
produce in ex4.c（ex4.c 中的 produce），21
producer（生产者），113
producer-consumer（生产者－消费者模型），22，113，272
program counter（程序计数器），196

programmable interrupt address（可编程中断地址），198
protocol（协议），333
protocol stack（协议栈），333
pseudo call（伪调用），101
pseudo-device（伪设备），426
ptclear in ptclear.c（ptclear.c 中的 ptclear），190
ptcreate in ptcreate.c（ptcreate.c 中的 ptcreate），183
ptdelete in ptdelete.c（ptdelete.c 中的 ptdelete），188
ptinit in ptinit.c（ptinit.c 中的 ptinit），182
ptrecv in ptrecv.c（ptrecv.c 中的 ptrecv），186
ptreset in ptreset.c（ptreset.c 中的 ptreset），189
ptsend in ptsend.c（ptsend.c 中的 ptsend），184
pute，7，243
pute in pute.c（pute.c 中的 pute），252

Q

queue table（队列表），52
queue.c，58
queue.h，53

R

RAM（随机存取存储器），140
Random Access Memory（随机存取存储器），140
random semaphore policy（随机信号量策略），114
random-access device（随机存取设备），371
raw mode（raw 模式），158，169，188
rdsksys.h，376
rdshufalloc in rdshufalloc.c（rdshufalloc.c 中的 rdshufalloc），399
rdsClose in rdsClose.c（rdsClose.c 中的 rdsClose），400
rdscomm in rdscomm.c（rdscomm.c 中的 rdscomm），386
rdsControl in rdsControl.c（rdsControl.c 中的 rdsControl），396
rdsInit in rdsInit.c（rdsInit.c 中的 rdsInit），381
rdsOpen in rdsOpen.c（rdsOpen.c 中的 rdsOpen），384
rdsprocess in rdsprocess.c（rdsprocess.c 中的 rdsprocess），402
rdsRead in rdsRead.c（rdsRead.c 中的 rdsRead），392
rdsWrite in rdsWrite.c（rdsWrite.c 中的 rdsWrite），389
read，243
read in read.c（read.c 中的 read），250
read-only address（只读地址），42
Read-Only Memory（只读内存），140
read.c，250
ready in ready.c（ready.c 中的 ready），83
  list（链表），73
  state（状态），72
ready.c，83
real address space（实地址空间），169
real-time（实时），215
  clock（时钟），216
  system（系统），14
recmsg，131
receive in receive.c（receive.c 中的 receive），134
receiving（接收）
  from a port（从一个端口），186
  state（状态），131
recvclr，131
recvclr in recvclr.c（recvclr.c 中的 recvclr），135
recvtime in recvtime.c（recvtime.c 中的 recvtime），228
redirection（重定向），575
redirection of I/O（I/O 重定向），551
Reduced Instruction Set Computer（精简指令集），33
reference count（引用计数），255
register（寄存器），39
relative pointer（相对指针），51
remote files（远程文件），459
replacement string（替换字符串），501
request queue（请求队列），270
resched in resched.c（resched.c 中的 resched），74
RESCHED_YES and RESCHED_NO（RESCHED_YES 和 RESCHED_NO），82
resident page（常驻页面），168
restore（恢复），41，93
resume（恢复），17
resume in resume.c（resume.c 中的 resume），92
rflersys.h，461
rflClose in rflClose.c（rflClose.c 中的 rflClose），479
rflGete in rflGete.c（rflGete.c 中的 rflGete），486
rflInit in rflInit.c（rflInit.c 中的 rflInit），492
rflPute in rflPute.c（rflPute.c 中的 rflPute），487
rflRead in rflRead.c（rflRead.c 中的 rflRead），480
rflSeek in rflSeek.c（rflSeek.c 中的 rflSeek），485
rflWrite in rflWrite.c（rflWrite.c 中的 rflWrite），482
rfscomm in rfscomm.c（rfscomm.c 中的 rfscomm），468
rfsControl in rfsControl.c（rfsControl.c 中的 rfsControl），488
rfsgetmode in rfsgetmode.c（rfsgetmode.c 中的 rfsgetmode），477
rfsInit in rfsInit.c（rfsInit.c 中的 rfsInit），491
rfsrdmsg in rfsrdmsg.c（rfsrdmsg.c 中的 rfsrdmsg），471
rfsOpen in rfsOpen.c（rfsOpen.c 中的 rfsOpen），474
ring of buffers（缓冲区环），304
RISC（精简指令集），33
ROM（只读内存），140
round-robin（轮询），75
roundmb in memory.h，148
run-time stack（运行时栈），38

S

sched.h，84

schedule, 72, 74
sched.exit.c, 84
second chance (二次机会), 174
sector (扇区), 372
seek (查找), 243, 372
seek in seek.c (seek.c 中的 seek), 253
segment (段), 35, 142
segmentation (分段), 169
semantics of disk operations (磁盘操作语义), 374
semaphore (信号量), 22, 112, 272
ID (标识符), 116
count (计数), 118
creation (创建), 120
data structure (数据结构), 116
deletion (删除), 121
implementation (实现), 113
invariant (不变量), 118
priority policy (优先级策略), 114
reset (重置), 123
scheduling policy (调度策略), 112
semaphores.h, 116
semaphores static and dynamic (静态和动态信号量), 119
semcreate, 22
semcreate in semcreate.c (semcreate.c 中的 semcreate), 120
semdelete in semdelete.c (semdelete.c 中的 semdelete), 122
semreset in semreset.c (semreset.c 中的 semreset), 123
send, 113, 131
send in send.c (send.c 中的 send), 133
sequential program (串行程序), 13
shared memory (共享内存), 20
shell (壳, 命令解释器), 549
shell in shell.c (shell.c 中的 shell), 562
shell.h, 554
signal (信号), 22, 118
signal in signal.c (signal.c 中的 signal), 118
sleep (睡眠), 223
sleep in sleep.c (sleep.c 中的 sleep), 224
sleep command in xsh_sleep.c (xsh_sleep.c 中的 sleep command), 576
sleep.c, 224
sleeping state (睡眠状态), 223
sleepms, 223
sndA in ex2.c (ex2.c 中的 sndA), 16
sndB in ex2.c (ex2.c 中的 sndB), 16
sndch in ex2.c (ex2.c 中的 sndch), 16
SoC (片上系统), 32
spin lock (自旋锁), 124
stack (栈), 38, 68, 143, 333
(protocol) (协议), 333
frame (帧架), 38
start in start.S (start.S 中的 start), 524
starvation (饥饿), 113
state (状态), 72
state suspended (状态挂起), 89
static (静态的)
configuration (配置), 542
semaphore allocation (信号量分配), 119
storage layout (存储器布局), 44
store (存储), 372
suite of protocols (协议簇), 333
suspend in suspend.c (suspend.c 中的 suspend), 96
suspended state (挂起状态), 89
swapping (交换), 168
symbolic constant (符号常量), 54
synchronization (同步), 20, 22, 111
synchronous (同步), 161, 241
event loop (事件循环), 12
event paradigm (事件模式), 216
sysinit in initialize.c (initialize.c 中的 sysinit), 527
system (系统)
backplane (底板), 32, 34
boot (引导), 522
call (调用), 7, 91
call interface (调用接口), 9
System on a Chip (片上系统), 32

T

task (任务), 14
TCP/IP (传输控制协议/互联网协议), 333
terminal application (终端应用程序), 7
terminating a process (终止进程), 98
test-and-set (测试和设置), 124
text segment (文本段), 44, 142
thrash (抖动), 174
thread (线程), 14
threads of execution (执行的线程), 140
tick rate (节拍率), 219
time (时间)
of day (一天的时间), 216
slice (切片), 218
time-of-day clock (时间时钟), 217
timed (定时的)
event (事件), 215, 216
message reception (消息接收), 226
timeout and retransmission (超时重发), 335
timer (定时器), 217
timer device (定时器设备), 42
timesharing system (定时系统), 14
TLB (转换后援缓冲器), 172
tokens (符号), 551

Translation Look-aside Buffer（转换后援缓冲器），172
transparency（透明度），497
trap（陷阱），535
tty（tty 设备），267
  control block（控制块），273
  initialization（初始化），295
  input（输入），277
  input modes（输入模式），288
  interrupt handler（中断处理程序），283
  lower half（下半部），283
  mode（模式），268
  output（输出），280
tty.h，274
ttyControl in ttyControl.c（ttyControl.c 中的 ttyControl），298
ttyGetc in ttyGetc.c（ttyGetc.c 中的 ttyGetc），277
ttyInit in ttyInit.c（ttyInit.c 中的 ttyInit），296
ttyInterrupt in ttyInterrupt.c（ttyInterrupt.c 中的 ttyInterrupt），284
ttyInter_in in ttyInter_in.c（ttyInter_in.c 中的 ttyInter_in），288
ttyInter_out in ttyInter_out.c（ttyInter_out.c 中的 ttyInter_out），286
ttyKickOut in ttyKickOut.c（ttyKickOut.c 中的 ttyKickOut），282
ttyPutc in ttyPutc.c（ttyPutc.c 中的 ttyPutc），280
ttyRead in ttyRead.c（ttyRead.c 中的 ttyRead），278
ttyWrite in ttyWrite.c（ttyWrite.c 中的 ttyWrite），283

U

UART（通用异步收发传输器），272
udelay in ethInit.c（ethInit.c 中的 udelay），314
UDP（用户数据报协议），334
udp.c，353
udp.h，352
udp_in，352
udp_in in udp.c（udp.c 中的 udp_in），353
udp_init，352
udp_init in udp.c（udp.c 中的 udp_init），353
udp_recv，352
udp_recv in udp.c（udp.c 中的 udp_recv），353
udp_recvaddr，352
udp_recvaddr in udp.c（udp.c 中的 udp_recvaddr），353
udp_register，352
udp_register in udp.c（udp.c 中的 udp_register），353
udp_release，352
udp_release in udp.c（udp.c 中的 udp_release），353
udp_send，352
Universal Asynchronous Transmitter and Receiver（通用异步发送器和接收器），272
unsleep.c，229
upper half（上半），240，269
USB（通用串行总线），199
user（用户）
  interface（接口），549
  mode（模式），45
  space（空间），35
User Datagram Protocol（用户数据报协议），334
usernet in usernet.c（usernet.c 中的 usernet），105

V

V system（V 系统），499
vectored interrupt（矢量中断），197
virtual（虚拟的）
  address space（地址空间），169
  memory（内存），159

W

wait，22，113
wait in wait.c（wait.c 中的 wait），17
waiting process state（等待进程状态），115
wakeup in wakeup.c（wakeup.c 中的 wakeup），230
word（词），35
write（写），263
write in write.c（write.c 中的 write），254

X

xdone in xdone.c（xdone.c 中的 xdone），100
Xinu，5
Xinu hierarchy（Xinu 层次结构），4
xinu.h，59
xsh_sleep.c，376

推荐阅读



大规模并行处理器程序设计
（英文版·第2版）

作者：David B. Kirk Wen-mei W. Hwu
ISBN：978-7-111-41629-6 定价：79.00元



计算机组成与设计：硬件/软件接口
（英文版·第4版）

作者：David A. Patterson John L. Hennessy
ISBN：978-7-111-41237-3 定价：139.00元



嵌入式计算系统设计原理
（英文版·第3版）

作者：Marilyn Wolf
ISBN：978-7-111-41228-1 定价：79.00元



多处理器编程的艺术
（英文版·修订版）

作者：Maurice Herlihy Nir Shavit
ISBN：978-7-111-41233-5 定价：79.00元



现代嵌入式计算（英文版）

作者：Peter Barry Patrick Crowley
ISBN：978-7-111-41235-9 定价：79.00元



Python语言程序设计（英文版）

作者：Y. Daniel Liang
ISBN：978-7-111-41234-2 定价：79.00元