计 算 机 科 学 丛 书

PEARSON

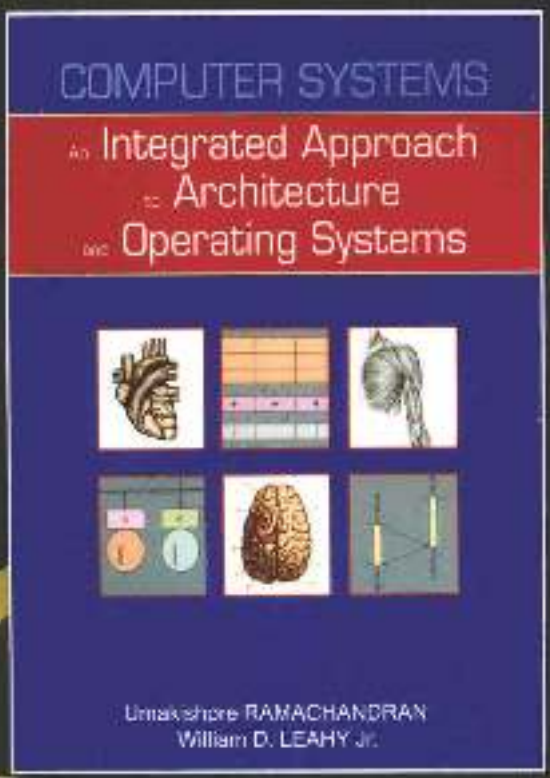
计算机系统

系统架构与操作系统的高度集成

[美] 阿麦肯尚尔·拉姆阿坦德兰（Umakishore Ramachandran） 小威廉 D. 莱希（William D. Leahy Jr.） 著
佐治亚理工学院
陈文光 清华大学 等译

Computer Systems

An Integrated Approach to Architecture and Operating Systems



机械工业出版社
China Machine Press

计算机系统 系统架构与操作系统的高度集成

Computer Systems An Integrated Approach to Architecture and Operating Systems

传统的计算机课程体系人为地割裂了解决问题时所需技能的综合性。例如汇编语言、计算机原理、计算机系统结构、操作系统和编译原理分别从不同角度介绍了计算机的硬件和软件系统，但是随着多核系统日渐成为主流，这种软硬件分离的教学方法变得不切实际。

国内外大学都在这方面展开了探索，即如何用一种综合的方法来介绍计算机系统的相关内容。卡内基·梅隆大学的《深入理解计算机系统》是目前比较成功的探索，它主要从程序员视角来讲解计算机系统，内容偏向系统软件（特别是操作系统），国内外很多大学已采用该教材作为课程的基础。而佐治亚理工学院的这本教材则是另一个有益的尝试，书中计算机系统结构和操作系统的内容基本平衡，旨在让学生了解计算机体系结构和系统软件之间的关系，为进一步深入学习计算机体系结构、操作系统和网络的高级课程或研究生课程，在这些领域进一步深造奠定良好的基础。

本书采用软硬件集成的方法，系统地讲解了计算机系统的软件和硬件知识及其相互关系。全书分为5个模块：处理器、内存系统、存储系统、并行系统和网络，分别讨论了处理器及其相关的软件问题、内存系统和分级存储体系、I/O和文件系统、操作系统问题及支持并行编程的多处理器中指应体系结构的特点、网络硬件的发展和处理各种网络行为的网络协议栈的特点等。



计算机系统：系统架构与操作系统的高度集成（英文版）
书号：978-7-111-31955-9
定价：69.00元

PEARSON

PEARSON
www.pearson.com

购书热线：(010) 88379804
编辑热线：(010) 88379991 88361066
印刷热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

ISBN 978-7-111-50636-2

9 787111 506362 >

定价：99.00元

计 算 书

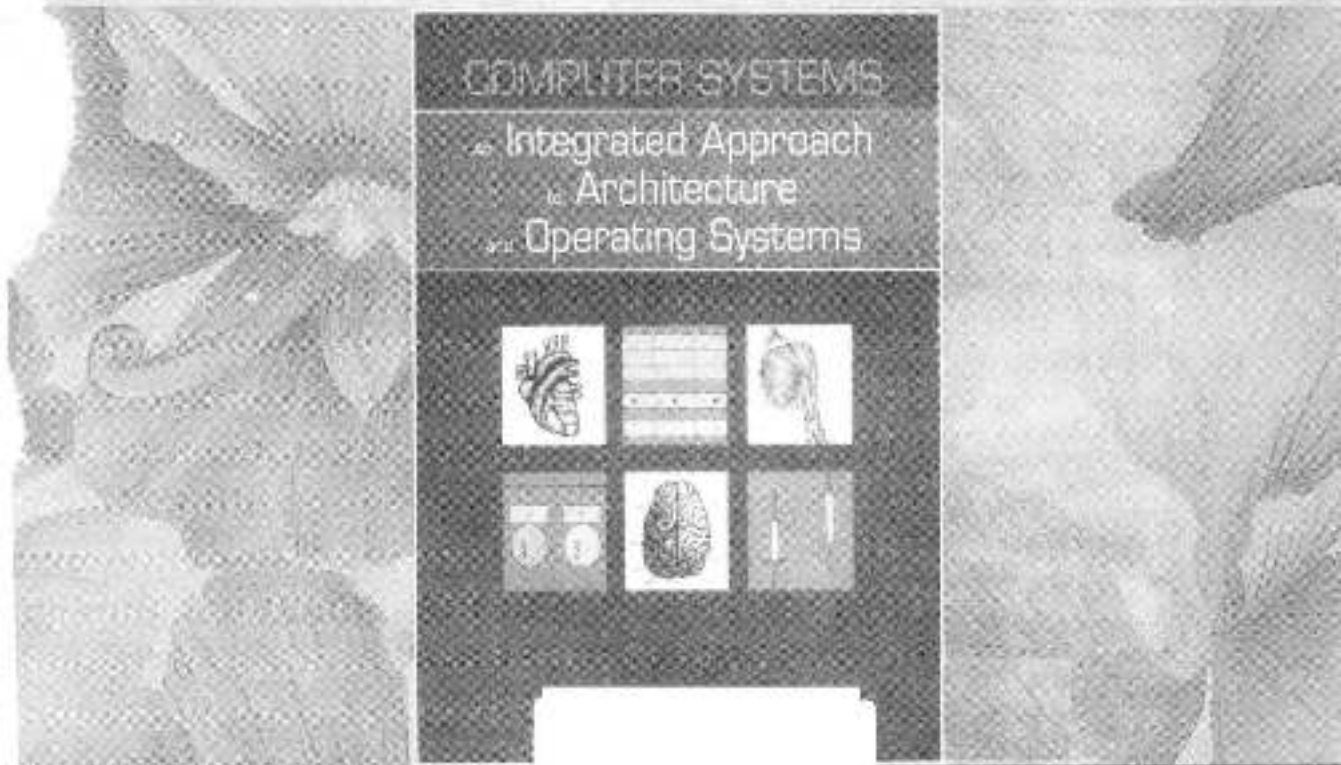
计算机系统

系统架构与操作系统的高度集成

[美] 阿麦肯尚尔·拉姆阿堪德兰（Umakishore Ramachandran） 小威廉 D. 莱希（William D. Leahy Jr.） 著
佐治亚理工学院
陈文光 等译
清华大学

Computer Systems

An Integrated Approach to Architecture and Operating Systems



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

计算机系统：系统架构与操作系统的高度集成 / （美）拉斯阿塔德兰（Ramachandran, U.），（美）莱希（Leahy, W. D.）著；陈文光等译．—北京：机械工业出版社，2015.7
（计算机科学丛书）
书名原文：Computer Systems: An Integrated Approach to Architecture and Operating Systems

ISBN 978-7-111-50636-2

Ⅰ. 计… Ⅱ. ①拉… ②莱… ③陈… Ⅲ. 计算机系统 Ⅳ. TP30

中国版本图书馆 CIP 数据核字（2015）第 128116 号

本书版权登记号：图字：01-2010-5389

Authorized translation from the English language edition, entitled *Computer Systems: An Integrated Approach to Architecture and Operating Systems*, 9780321486134 by Umaxishore Ramachandran, William D. Leahy, Jr., published by Pearson Education, Inc. Copyright © 2011.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2015.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或与原本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

本书采用集成方法，系统地讲解了计算机系统的软件和硬件知识。全书分为 5 个模块：处理器、内存系统、存储系统、并行系统和网络。分别讨论了处理器及其相关的软件问题、内存系统和分级存储体系、I/O 和文件系统、操作系统问题及支持并行编程的多处理器中相应体系结构特点、网络硬件的发展和处理各种网络行为的网络协议栈的特点等。

本书适合作为本科生“计算机系统”课程的教材，同时也适合相关专业研究人员阅读。

出版发行：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：殷虹琳　　关　敏　　责任校对：董纪丽

印　　刷：北京城信伟业印刷有限公司　　版　　次：2015 年 7 月第 1 版第 1 次印刷

开　　本：185mm × 260mm　1/16　　印　　张：34.25

书　　号：ISBN 978-7-111-50636-2　　定　　价：99.00 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

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

文艺复兴以来，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；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，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，不仅反映计算机学科的发展与进步，更揭示了学术的源变，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退。

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相助，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；而原书的作者也非常关注其作品在中国的传播，有的还专门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

Computer Systems: An Integrated Approach to Architecture and Operating Systems

美国未来学家阿尔温·托夫勒在1980年3月出版了他的经典著作《第三次浪潮》。该书在全世界引起了巨大的反响。在这本书中，作者将人类社会发展分为农业阶段、工业阶段和以信息时代为主要特征的后工业化社会阶段。从历史观角来看，信息化具有与工业革命等同的重要性，将重塑我们的社会结构和日常生活。

计算机是信息化的核心，随着信息化与社会生活的深度融合，人们对计算机专业毕业生的要求也越来越高。他们不仅需要掌握计算机本身的知识，还需要了解相关行业的知识。这就给计算机专业教学带来了很大的挑战。如何在有限的课时里面，能够覆盖计算机专业的核心内容，提供给学生足够的基础使其能够在所选的方向上继续深造呢？

传统的计算机课程体系有一个主要问题，就是课程人为地割裂了解决问题时所需技能的综合性。例如汇编语言，计算机原理，计算机系统结构，操作系统和编译原理分别从不同角度介绍了计算机的硬件和软件系统，但人们在面临一个具体问题的时候，比如优化一个数据分析程序时需要的技能是综合性的，需要知道高级语言程序变成了什么样的汇编语言，这些汇编语言在操作系统的调度下如何加载和运行，运行时的指令如何在处理器的流水线里乱序执行，其访存是缓存命中还是缓存缺失，并发访问是如何相互隔离的，等等。

因此，国内外大学都在这方面展开了探索，即如何用一种综合的方法来介绍计算机系统的相关内容，这样不但可以减少课时，也能让相关知识的衔接更加平滑，整体的知识体系更加系统化。CMU的《深入理解计算机系统》是目前比较成功的探索，国内外很多大学都已采用该教材作为课程的基础。我们翻译的这本书则是另一个有益的尝试。本书中计算机系统结构和操作系统的内容基本平衡，而《深入理解计算机系统》则明显偏向操作系统，对计算机系统结构的相关内容介绍相对较少。例如，本书对I/O中断处理专门安排了硬件实验，要求学生用硬件设计语言设计CPU并支持中断处理，这类实验对学生理解整个计算机系统是如何运作的非常重要，但在《深入理解计算机系统》中没有这部分内容。

我们希望本书的翻译出版，能够为国内的计算机系统教育提供一种新的选择。对希望未来研究、设计新型计算机系统的学生来说，本书提供了更加完整的基础。

本书的翻译是由我和我的学生完成的，我本人翻译了前言和第1章，汤雄超翻译了第2～4章，杨飞翻译了第5～7章，张峥翻译了第8～10章，朱晓伟翻译了第11～12章，陈力维翻译了第13章和附录。本人对全书进行了审校，因此书中的错误都应该由本人负责。

感谢机械工业出版社华章公司将这本书引入国内，感谢温莉芳副总经理，朱劼编辑和吴怡编辑在本书翻译过程中给予我们的极大耐心。

陈文光

2015年6月

为什么在计算机系统领域需要有一本新书

和高中生谈论计算机会让人感到兴奋。人们对“盒子（计算机机箱）里”有什么东西有一种神秘感，正是那个盒子里的东西使计算机能够完成诸如让用户玩有很棒图形的视频游戏，播放音乐（不管是 RAP 还是交响乐），发送即时消息给用户的朋友等功能。本书的目的就是与读者一起开展一段揭示盒子里有什么秘密的旅程。作为即将开展的旅程的一瞥，让我们在一开始就表明，让这个盒子变得有趣的并不仅仅是硬件，还包括软件和硬件是如何结合起来完成各种功能的。因此，本书所采用的途径是把软件和硬件放在一起观察，看它们是如何相互帮助以及如何协同起来让计算机变得有趣而且有用的。我们把这个过程称作“打开盒子”——即揭开盒子里有什么这个秘密。我们会看盒子内部并理解如何设计关键的硬件单元（处理器、内存以及外设控制器），理解要管理计算机中的所有硬件资源，包括处理器、内存、I/O 和硬盘，多处理器以及网络所需的操作系统抽象。为此，这是一本计算机系统教学的入门课程教材，采用了一种新颖的集成教学法来介绍相关内容。

本书的目标是让学生在本科生涯（计算机科学或计算机工程专业）的早期就在相关主题方面接触足够宽泛的知识。本书的内容是为用软硬件集成的方式进行课程教学而写的，这种方式使得学生可以了解计算机体系结构和系统软件之间的关系。书中的材料可以作为 4 学分的半年学期课程教材，或者作为 5 学分的季度课程教材，或是作为每季度 3 学分的两季度的课程系列的教材。基于本书的课程可以为学生打下很好的基础，以进一步深入学习计算机体系结构、操作系统和网络的高级课程或研究生课程，在这些领域进一步深造。此外，这类课程可以尽早激发学生对计算机系统的兴趣，对学生在本科期间参加研究工作也有帮助。

本书的主要特点（除了处理器和内存系统之外）如下：

1）详细介绍了存储系统；

2）专门用一章介绍了网络问题；

3）专门用一章介绍了多线程和多进程编程。

教学风格

本书采用的教学风格是“发现”而非“教导”或“灌输”。此外，内容是以“自顶向下”的方式展现的，读者首先看到我们要解决的问题，然后看到解决方案。以内存管理部分（第 8 章）为例。我们首先提出问题“什么是内存管理”，一旦理解了内存管理的需求，我们再开始探讨内存管理所需的软件技术和相应的硬件支持。因此，本书是以一种讲故事的方式来进行概念展现，学生们看起来很喜欢这种方式。在适当的地方，我们在不同章节用一些例题来阐明观点。

我们在撰写本书的时候始终以学生为中心。书中包含大量例题，可以帮助学生固化刚刚讨论过的概念。从我们作为教师的经验来看，学生确实喜欢了解历史背景（那些对计算的演化起到重要影响的著名的计算机科学家和机构）和现状，以及我们是怎么一步一步发展过来

的。这些历史片段遍布在全书中。除此之外，在必要的时候，在若干章我们都包括了一节从历史角度进行的回顾。我们从学生那里学到并采用的另一个措施是在文中直接给出参考文献，而不是在文末才给出。读者可以看到贯穿本书的大量脚注。此外，我们在每章末尾专门有一节给出外部链接（教材和开创性的著作），包括参考文献和扩展阅读的建议，这些内容在正文中不一定都被引用了，但是有助于增强学生的知识基础。今天，随着因特网上的信息日益丰富，为附加的信息提供URL链接是一件很有诱惑力的事情。但是，我们拒绝了这一诱惑（除了那些权威信息源的可靠链接）。尽管如此，我们知道现在学生在去图书馆之前会先搜索因特网，当然他们也应该这么做。在这种情况下，我们给学生一个提示：在利用因特网作为信息源的时候要慎重。通常，使用Google搜索是获取某种信息的最快方法。但是，必须对这些信息进行筛选以保证其准确性。作为经验法则，使用因特网上的信息来满足好奇心或是回答与流言有关的问题。（DEC是如何衰落的？为什么Linux成功了而Unix BSD却没有？Burroughs公司的历史是什么？计算机系统的真正先驱是哪些人？）对于技术问题（Pentium 4处理器的流水线结构是什么？VAX 11/780的指令集体系结构是什么？）则要从已出版的书籍、相关会议和期刊论文（当然它们中的大多数也可以在线获取）中寻求答案。

佐治亚理工学院计算机学院从1999年秋季学期开始，每学期都开设这门软硬件集成的课程，本教材就是这门课程的副产品。在一开始，本书作者为课程开发了完整的讲义和幻灯片，并使用两本标准的教材（一本体系结构教材和一本操作系统教材）作为课程的背景参考资料来补充课程的材料。从2005年春季开始，我们将课件转换成了本教材的手稿，因为学生一直想要一本与课程内容和风格匹配的教材。本教材的在线版本从2005年春季开始在佐治亚理工学院用于本课程。使用集成的方法介绍计算机系统。本课程每年开设3次（包括夏季学期），每学期有80多名学生选课。因此，书稿在付印之前经过了连续15个学期的教学，从选修本课的学生那里接受了持续不断的反馈与改进意见。

在设计产生本书的课程时，以及在撰写本书的时候，我们从其他机构开设的系统入门课程以及一些优秀教材中学到了很多东西。例如，MIT[⊖]的计算机系统入门课程拥有很长的历史和传统，而且是真正独一无二的。从这门课程中总结的教材[Saltzer，2009]对激发学生深入学习计算机系统来说是很好的资源。在撰写本书的时候，我们坦承受到了[Ward，1989]和[Kurose，2006]的教学法的启发。

本书的结构和可能的阅读路径

本书的知识内容可以分为5个模块。下面的路线图建议了一些可能的阅读路径。这些路径假设关于体系结构和操作系统的内容一样多。

1）**处理器** 本书的第一个模块是关于处理器以及与处理器相关的软件问题的。我们从探索如何设计盒子中的大脑[⊖]（处理器）开始。有哪些软件问题？既然计算机的大部分部件主要是使用高级语言编程的，我们考虑了高级语言结构是如何影响处理器的指令集的（第2章）。一旦理解了指令集的设计，我们就开始关注实现处理的硬件技术。我们从实现一个简单的处理器开始（第3章），然后考虑实现一个使用流水线技术的性能优化的处理器（第5章）。处理器是计算机系统中的宝贵资源，因此必须在多个相互竞争的程序间复用。正如第1章中视频游戏的例子所揭示的一样（见1.3节）。操作系统的职责就是保证资源的有效使用。本模块以

⊖ http://mit.edu/6.033/www/。

⊖ 本书封面中的解剖图用来表示计算类比为在人体内自然发生的网络分布式处理。

用于处理器调度的操作系统算法结束（第6章）。

我们预计第2、3、5和6章每章需要3小时的课堂讲授时间和1小时的练习题时间。

2）**内存系统** 第二个模块介绍了内存系统和内存层次。计算机程序包括代码和数据，并且都需要存放的空间。计算机的内存系统可能是决定性能最为关键的因素。如果内存系统不能以匹配处理器速度的方式提供执行程序所需的代码和数据，处理器速度（现在以Ghz为量度）就毫无意义。由于技术的进步，内存系统的大小一直在跨越式发展，但应用程序使用内存的胃口也在以同样的速度增长着，如果不是增长得更快的话。因此，内存也是宝贵资源，操作系统的作用就是保证用好资源。本模块的第一部分是关于有效管理内存的操作系统算法以及相应的体系结构支持的（第7章和第8章）；第二部分则介绍内存层次，可以帮助降低处理器在访问代码和数据时的延迟（第9章）。

我们预计第7、8和9章每章需要3小时的课堂讲授时间和1小时的练习题时间。

3）**存储系统** 第三个模块是关于I/O（特别是稳定存储）和文件系统的。只有与计算机进行交互才能让计算机有用且有趣。首先，我们讨论能够把处理器的注意力从当前执行的程序中脱离出来的硬件机制（第4章）。这些机制既包括外部事件也包括处理器执行程序时遇到的内部异常。与硬件机制相关的软件问题是解决正常程序执行的“不连续”性，包括记录原有程序的执行位置以及程序的当前执行状态。然后，我们介绍处理器与I/O设备的接口机制以及相应的底层软件技术（第10章），并特别强调了磁盘子系统。随后，我们完整地介绍了在稳定的存储设备（如磁盘）上如何构建文件系统（第11章）。

我们预计第4章和第10章每章需要3小时的课堂讲授时间和1小时的练习题时间，第11章需要6小时的课堂讲授时间和2小时的练习题时间。

4）**并行系统** 计算机体系结构是一个快速变化的领域。芯片密度、处理器速度、内存容量等在过去20年中都呈现出指数增长速度，并在可预见的未来仍然保持这样的增长速度。并行处理已不再是超级计算机独有的深奥概念。随着在一个芯片上集成多个CPU的多核技术的到来，并行性已经变得很常见。因此，理解与并行性有关的软件和硬件技术对于回答“盒子里有什么”这样的问题十分必要。这个模块包括多处理器中支持并行编程的操作系统问题以及相应的体系结构功能（第12章）。

我们预计第12章需要6小时的课堂讲授时间和2小时的练习题时间。

5）**网络** 在我们生活的世界上，单独一个盒子几乎没有任何用处，除非它与外部世界相连。与你的朋友在网络上对战多人视频游戏（在第1章介绍）是一个很好的例子。但即使在日常生活中，我们也需要利用网络来收发电子邮件或浏览因特网等。网络与其他输入/输出设备的不同之处在于，现在你的盒子得以连接世界了！你需要一种语言让你的盒子与外部世界交谈，并处理网络的各种情况，例如暂时或永久的连接中断。这一模块讨论了网络硬件的进化，以及用来处理各种网络状况的网络协议栈（操作系统的一部分）的功能（第13章）。

我们预计第13章需要6小时的课堂讲授时间和2小时的练习题时间。

总而言之，第2章～第10章每章需要用1周时间授课；第11章～第13章每章需要2周时间授课，正好在15周的一个学期里讲完。五个模块中的软件和硬件问题在本书中是一起介绍的，上述建议的阅读路径也是按照这种处理方式进行的。

读者也可以选择在体系结构和操作系统主题之间重点关注某部分的内容，而不会损失连续性。以处理器模块为例，第3章和第5章都是关于处理器的硬件实现问题的。对于偏重操作系统的课程，可以考虑少讲授或者完全跳过介绍流水线处理器实现（从5.7节开始）的第5

章，而不会损失课程的连续性。类似地，在偏重体系结构的课程里，可以跳过介绍处理器调度算法的第6章，而不会损失课程的连续性。

在内存模块中，第8章从操作系统角度涉及页式内存管理的细节。偏重体系结构的课程可以跳过这一章，而不会损失连续性。类似地，偏重操作系统的课程可以选择淡化第9章中对缓存的细节描述。

在存储模块中，面向体系结构的课程可以选择淡化第11章中文件系统的内容，而不必担心损失连续性。

在并行模块中（第12章），面向体系结构的课程可以跳过多线程的操作系统支持，以及一些高级主题，包括多处理器调度、死锁以及并发性的经典问题和解决方案；类似地，面向操作系统的课程可以选择跳过体系结构方面的高级主题，例如多处理器缓存一致性、并行机的分类以及互连网络等。考虑到并行性的重要性，在任何课程中，只要时间许可，应尽量覆盖这一章的全部内容。

在网络模块中（第13章），面向体系结构的课程可以跳过传输层和网络层的细节（分别是13.6节和13.7节）。面向操作系统的课程可以选择少讲一些协议栈的链路层（13.8节）和网络硬件（13.9节）的内容。

本教材在计算机科学课程体系中的位置

图P-1显示了计算机系统的抽象层次。我们可以尝试将图P-1中的不同层次的抽象与传统计算机科学课程体系中的课程相关联。诸如基础程序设计、面向对象程序设计、图形学以及HCI（人机交互）的课程通常使用较高层次的抽象。通常计算机科学和计算机工程的课程体系中包含数字电路和逻辑电路设计的基础课程，然后是计算机组成原理，介绍计算机的硬件设计。在计算机组成原理课程之上（在图P-1的抽象层次级别之上），大部分课程使用归纳方法：不同的课程分别覆盖计算机体系结构、操作系统和计算机网络的高级概念。



图P-1 计算机系统中的抽象层次

今天，设计计算机系统已经是软硬件集成的过程。这使人们对如图模式提出了质疑，特别是对计算机科学本科的课程体系中学生发展专业技能的早期。

以本书为基础围绕上述主题的课程是一种独特的尝试，用集成的方法在计算机系统的入门课程中介绍不同层次的概念（覆盖了图P-1中的深色区域——系统软件及其与计算机体系结构的关系）。这门课程将为渴望学习计算机体系结构、操作系统和网络中的高级主题（图P-2）的学生提供坚实的基础。

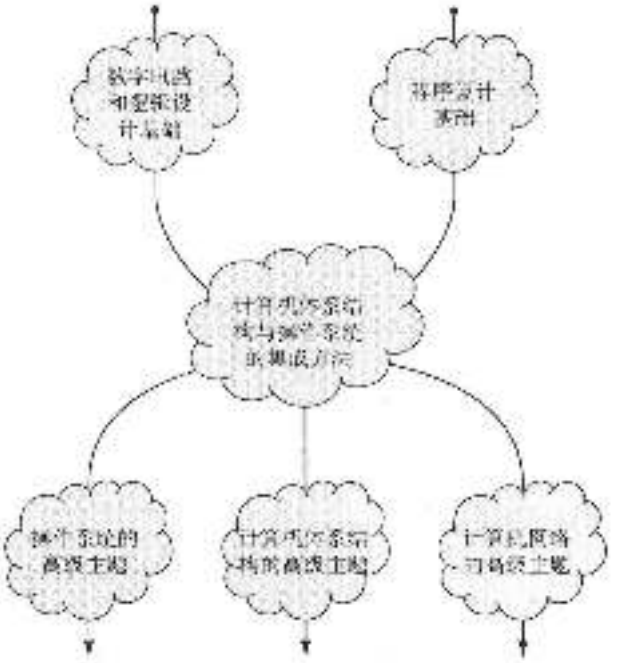


图 P-3 系统课型系列

使用本书内容的课程的先修课程很直接：逻辑设计基础和高级语言程序设计（最好是 C 语言）基础。换句话说，对在本书内容之上和之下的抽象层次需要有基本的理解（见图 P-1）。

在数字电路和逻辑设计基础以及程序设计基础方面都有许多优秀的教科书。类似地，在计算机体系结构、操作系统和计算机网络的高级主题方面，也有优秀的教科书。**唯独缺少的是对计算机系统进行简单、集成化的介绍，使其成为基础课程和高级主题之间桥梁的图书。本书的目标就是成为这样一座桥梁。**

计算机科学作为一门学科其边界已经扩展了。相应地，学习计算机科学的学生的兴趣也各不同。计算机科学的课程需要为学生在本科阶段的学习提供合理的选择。另一方面，课程也有责任保证，不论学生的选择是什么，都能学到计算机系统（广义）的核心知识。我们相信基于本书的课程可以满足这样一种系统核心知识的要求。如果正确地计划本课程，可以给学生提供充足的机会，通过其他课程来深入学习计算机系统。例如，我们建议在大学二年级开设将本书作为教材的课程。在大二的时候，学生可能可以学习更加面向设计的课程——针对体系结构、操作系统或是网络——以他们在大二从本书中学到的基本概念为基础。最后，在大四的时候，学生可以选修在这些领域中更具概念性的高级主题课程。

本书在体系结构和操作系统的内容方面是大致平衡的。我们认为，计算机科学专业的学生在本科期间应该对这两方面同等重视，不管他们的职业目标是什么。当然，希望成为系统架构师的学生必须了解本书中介绍的软件和硬件之间的互动。即使是希望进行软件开发的学生，了解这些知识对于成为更好的程序员也是必需的。但是，这取决于每个老师对这两个主题强调的程度。好消息是，本教材允许教师选择他们认为必需的课程深度，以与他们所在学校的课程结构相适应。例如，如果教师选择减少体系结构方面的内容，可以很轻松地简单介绍处理器实现的有关章节（第 3 章和第 5 章），而不必担心内容的衔接问题。在讨论本书结构的时候，我们已经对五个实践给出了类似的建议。

讲授系统的集成课程的补充材料

我们充分理解教师在讲授需要介绍体系结构、操作系统和网络的计算机系统的集成课程时所面临的难题。

为此，我们已经提供了一组在线资源。[⊖]我们已经讲授了11年本课程，每年3次，作为所有计算机专业学生的必修课，因此我们已经积累了相当多的在线资源。

1）我们有本课程所有内容的PowerPoint讲稿，能得准备课程和转换（从原有的幻灯模型）更加容易。

2）每个模块都有一个重要的实验部分。我们提供了这些已经迭代过多次的实验的详细描述，以及用于实验特定方面的软件模块（例如模拟器）。

3）除了每章后的练习题之外，我们还对本课程的不同实质还有附加的问题集、家庭作业以及本课程迄今为止的期中和期末考试题。

在补充材料中包含的样例实验

处理器设计

我们给学生提供一个完成了90%的处理器数据通路设计。通过完成数据通路可以帮助学生熟悉相关设计。然后他们要设计基于微码的控制逻辑（使用类似LogicWorks的逻辑设计工具），利用数据通路实现一个简单的指令集。这能帮助学生理解数据通路的工作原理并体会一些设计权衡。学生会得到真实电路设计的经验，并通过逻辑设计软件内置的模拟器对设计进行功能测试。

中断和输入/输出

学生在第一个实验的基础上增加电路以实现中断系统。然后他们（使用汇编语言）写一个中断处理程序。实验的电路设计部分再次通过LogicWorks软件系统实现并进行功能模拟。此外，我们还给学生提供了处理器模拟器，他们需要在其中增加中断支持，并与他们用汇编语言写的中断处理器程序一起工作。这个实验不仅可使中断系统的操作变得清晰，还展示了底层设备输入/输出的基本概念。

虚存子系统

学生在处理器模拟器上实现虚存子系统。在这个实验中，学生可通过实现和实验不同的页替换策略，获得开发操作系统中内存管理部分的经验。这个实验是用C语言实现的。

多线程操作系统

在我们提供的模拟器上，学生实现多线程操作系统的基本模块，包括CPU和I/O调度队列等。他们可实验不同的处理器调度策略。这个模块是用C语言和pthread实现的。学生可从实验中获得并行编程经验，并接触不同的CPU调度算法。

可靠传输层

学生在我们提供的一个模拟的网络层上实现一个简单的可靠传输层。在传输层必须处理

⊖ 关于本书教辅资源，用书教师可向培生教育出版集团北京代表处申请，电话：010-57355169/57355171，电子邮件：service.cn@pearson.com。——编辑注

的问题包括损坏的包、丢包以及乱序到达。这个实验也是用 C 语言和 pthread 实现的。

注意

在开始探索计算机系统内部的旅程之前，我们要提醒读者注意：在展示计算机系统设计的教科书中，习惯上会通过有数字的例子来说明和支持相关概念。历史可以揭示未来。如果说在技术发展中有什么东西不变的话，那就是变化。当你买了一辆新车，在车驶出展厅的那一刻，这辆车就变成了二手车。同样地，我们使用的任何有数字的例子中的数字，如处理器速度、内存容量或是外设的传输速率马上就会过时。真正不变的是原理，这也是本书的核心内容。一个让人欣慰的因素是，尽管绝对数字可能会随时间变化，从 MHz 到 GHz、从 MB 到 GB，相对数字随着技术的发展相对保持不变。这使得书中的数字示例也具有持久性。

致谢

我们极大地受惠于若干国内外同行，他们直接或间接促成了本书的出现。首先，我们要感谢 Yale Patt，从 2004 年夏天我们介绍了在佐治亚理工学院开设的这门课程后，他用具有无与伦比的说服力的方式告诉我们应该把课程的内容写成教材，因为大家迫切需要一本用集成方式介绍系统概念的图书。我们可以很诚实地说，如果没有他的鼓励，我们可能不会走上写书这条路。下面这些其他学校的同行也鼓励我们进行本书的写作，因此需要特别致谢：Jim Goodman（威斯康辛大学麦迪逊分校和新西兰奥克兰大学），Liviu Iftode（Rutger 大学），Phil McKinley（密歇根州立大学）以及 Anand Sivasubramaniam（宾州州立大学和 TCS）。我们要特别感谢 Jim Goodman，他仔细阅读了本书手稿的早期草稿，并提出了详细的反馈，使本书的叙述得到了很大的改进。除了这些人以外，我们还从其他学校的一些同行那里得到了很多对本书实验的积极支持。

写书的第一步是创建一份书稿供佐治亚理工学院的学生内部使用。对选择佐治亚理工学院 CS 2200 课程的学生，我们怎么感谢也不为过。从 2005 年春季学期开始，几代学生使用了本书的在线版本，并提出了反馈意见，对改进本书表达的清晰性、错误纠正、提供读者可能有兴趣的历史链接等起到了重要的作用。此外，有 3 名本科生帮助绘制了本书中的部分插图：Kristin Champion、John Madden 和 Vu Ha。

计算机学院的部分同事，包括 Nate Clark、Tom Conte、Constantine Dovrolis、Gabriel Loh、Ken Mackenzie 以及 Milos Prvulovic，对本书给予了建设和有洞察力的评论，帮助本书的叙述更加清晰。我们受惠于 Constantine Dovrolis 对本书网络一章早期版本的建议和反馈，使得我们不仅改进了内容，还改变了这一章的叙述顺序。Ken Mackenzie 的建议让我们在第 3 章的处理器设计中给出了一种简单的控制方法。Tom Conte 对流水线一章给出了详细的评论，帮助我们更清晰地表达内容。北卡州立大学的 Eric Rotenberg 为流水线一章的早期草稿提出了非常有意义的反馈。Junsuk Shin 写了本书附录中的简单客户端-服务器的套接字代码。我们向他们所有人表示特别的感谢。

我们要感谢佐治亚理工学院，以及计算机学院的远见卓识，鼓励我们在教学方面进行创新。实际上，从 1996 年开始对本科课程体系的改革使得我们开始批判性地思考应该如何教育本科生并了解在课程体系中缺少了什么。这最终导致我们开发了第一门集成方式的系统课程，包括体系结构、操作系统和网络等。

作为图书出版方面的新手，我们从成功的教科书作者那里学到了经验。我们需要特别感谢 Yale Patt（德州大学）、Jim Kurose（麻省大学）、Jim Foley（佐治亚理工学院）、Andy van Dam（布朗大学）、Sham Navathe（佐治亚理工学院）、Rich LeBlanc（佐治亚理工学院）和 Larry Snyder（华盛顿大学）等。我们怎么感谢他们都不为过，他们分享了很多经验，包括出版社的选取、与编辑的合作、为可能的评阅人编制问题，以及如何有效地利用评阅意见修订书稿。

书稿经过了几轮的外部评审。大部分匿名评阅人深思熟虑而且有技巧地精准指出了改进书稿的方式。我们对匿名审稿人付出时间和精力帮助本书最后成型表示非常感谢。

特别感谢 Addison-Wesley 出版我们这本教科书。Matt Goldstein 是一个极好的编辑，他负责本书的评阅流程，并建议我们如何修改书稿。他具有一种既能督促我们工作，又不显得傲慢的独特风格。当我们没有按计划完成任务时他表现出了极大的耐心，并对本书背后的愿景给予了毫无保留的支持。我们要感谢 Marilyn Lloyd，Pearson 的高级产品经理，他负责我们的教科书产品。我们还要感谢 Pearson 的 Jeff Holcomb、Chelsea Bell 和 Dan Parker。作为管理产品流程日常事务的项目经理，Aptara 公司的 Dennis Free 以及 Apatara 公司的职员，包括 Jawwad Ali Khan 和 Rajshri Walia，以及 Write With 公司的 Brian Baker，都值得特别提及。他们为本书尽快生产印刷做出了贡献。

最后，我们要感谢我们的家人，他们的爱、理解与支持使我们能够持续撰写本书。补充一点，Umakishore 的父亲是一位著名的小说家（笔名“Umachandran”），他著有多本小说，对他的回忆是写作本书的灵感。

出版者的话
译者序
前言

第 1 章 概述……1
1.1 盒子里有什么……1
1.2 计算机系统中的抽象层次……1
1.3 操作系统的作用……3
1.4 盒子里正在发生什么吗……5
1.4.1 在计算机上启动应用程序……7
1.5 计算机硬件的演化……7
1.6 操作系统的演化……9
1.7 本书导读……9
练习题……10
参考文献注释和扩展阅读……10

第 2 章 处理器体系结构……12
2.1 处理器设计涉及什么……12
2.2 如何设计指令集……13
2.3 常见的高级语言功能集……13
2.4 表达式和赋值语句……14
2.4.1 操作数放在哪里……14
2.4.2 在指令中如何指定内存地址……17
2.4.3 每个操作数应该有多宽……18
2.4.4 字节序……19
2.4.5 操作数打包以及字操作数的对齐……21
2.5 高级数据抽象……22
2.5.1 结构……23
2.5.2 数组……23
2.6 条件语句和循环……24
2.6.1 if-then-else 语句……25
2.6.2 switch 语句……26
2.6.3 循环语句……27
2.7 检查点……27
2.8 编译函数调用……27
2.8.1 调用者的状态……28
2.8.2 过程调用剩余的工作……30
2.8.3 软件惯例……31
2.8.4 活动记录……35
2.8.5 返回……36
2.8.6 栈指针……36
2.9 指令集体系结构选择……38
2.9.1 额外的指令……38
2.9.2 额外的寻址模式……39
2.9.3 体系结构类型……39
2.9.4 指令格式……39
2.10 LC-2200 指令集……42
2.10.1 指令格式……42
2.10.2 LC-2200 寄存器组……43
2.11 影响处理器设计的问题……44
2.11.1 指令集……44
2.11.2 应用程序对指令集设计的影响……45
2.11.3 其他驱动处理器设计的问题……46
小结……47
练习题……47
参考文献注释和扩展阅读……49

第 3 章 处理器实现……51
3.1 体系结构与实现……51
3.2 处理器实现涉及什么……51
3.3 重要的硬件概念……52
3.3.1 电路……52
3.3.2 数据通路的硬件资源……52
3.3.3 边沿触发逻辑……53
3.3.4 连接数据通路元件……54
3.3.5 基于总线的设计……57
3.3.6 有限状态机……59

3.4 数据通路设计……65
3.4.1 ISA 与数据通路宽度……61
3.4.2 时钟脉冲宽度……62
3.4.3 检查点……62
3.5 控制单元设计……62
3.5.1 ROM 加状态寄存器……63
3.5.2 FETCH 宏状态……65
3.5.3 DECODE 宏状态……68
3.5.4 EXECUTE 宏状态：ADD 指令（R 型指令部分）……68
3.5.5 EXECUTE 宏状态：NAND 指令（R 型指令部分）……71
3.5.6 EXECUTE 宏状态：JALR 指令（J 型指令部分）……71
3.5.7 EXECUTE 宏状态：LW 指令（I 型指令部分）……72
3.5.8 EXECUTE 宏状态：SW 和 ADDI 指令（I 型指令部分）……75
3.5.9 EXECUTE 宏状态：BEQ 指令（I 型指令部分）……75
3.5.10 设计汇编程序中的条件分支……78
3.5.11 再谈 DECODE 宏状态……79
3.6 控制单元设计的另一种选择……80
3.6.1 微程序控制……80
3.6.2 硬连线控制……81
3.6.3 在两种控制设计风格中选择……82
小结……82
历史回顾……83
练习题……84
参考文献注释和扩展阅读……86

第 4 章 中断、陷入及异常……87

4.1 程序执行中的不连续性……88
4.2 处理程序不连续性……89
4.3 处理程序不连续性的体系结构改进……91
4.3.1 修改 FSM……91
4.3.2 一个简单的中断处理过程……92
4.3.3 处理嵌套中断……92
4.3.4 从处理过程中返回……95
4.3.5 检查点……95
4.4 处理程序不连续性的硬件细节……96
4.4.1 中断的数据通路细节……96
4.4.2 获得处理过程起始的细节……97
4.4.3 保存/恢复线……99
4.5 信息汇总……100
4.5.1 体系结构和硬件改进总结……100
4.5.2 工作中的中断机制……100
小结……102
练习题……103
参考文献注释和扩展阅读……104

第 5 章 处理器性能与流水线处理器的设计……105

5.1 时间和空间性能指标……105
5.2 指令频率……107
5.3 基准测试程序……108
5.4 提升处理器的性能……111
5.5 加速比……112
5.6 提升处理器的吞吐量……114
5.7 流水线简介……115
5.8 指令处理流水线……115
5.9 简单指令流水线的问题……117
5.10 修正指令流水线里的问题……118
5.11 指令流水线的数据通路元件……120
5.12 针对流水线的体系结构与实现……121
5.12.1 指令穿过流水线的过程详解……122
5.12.2 流水线寄存器的设计……124
5.12.3 各个阶段的实现……125
5.13 冒险……125
5.13.1 结构冒险……126
5.13.2 数据冒险……126
5.13.3 控制冒险……135
5.13.4 冒险总结……141
5.14 在流水线处理器里处理程序不连续性……142
5.15 处理器设计的高级话题……144
5.15.1 指令级并行……144
5.15.2 更深的流水线……145

5.15.3 在乱序执行下再次讨论程序不连续性 …… 147
5.15.4 管理共享资源 …… 148
5.15.5 功耗 …… 149
5.15.6 多核处理器设计 …… 149
5.15.7 Intel Core 微架构：一个流水线 …… 150
小结 …… 151
历史回顾 …… 152
练习题 …… 152
参考文献注释和扩展阅读 …… 156

第 6 章 处理器调度 …… 157

6.1 引言 …… 157
6.2 程序和进程 …… 158
6.3 调度环境 …… 161
6.4 调度基础 …… 162
6.5 性能指标 …… 165
6.6 非抢占式调度算法 …… 167
6.6.1 先到先服务 …… 167
6.6.2 最短作业优先 …… 170
6.6.3 优先级 …… 171
6.7 抢占式调度算法 …… 172
6.7.1 轮转调度器 …… 175
6.8 结合优先级和抢占 …… 178
6.9 元调度器 …… 178
6.10 评价 …… 179
6.11 调度对处理器体系结构的影响 …… 180
小结和展望 …… 181
Linux 调度器——一个案例研究 …… 181
历史回顾 …… 183
练习题 …… 185
参考文献注释和扩展阅读 …… 186

第 7 章 内存管理技术 …… 187

7.1 内存管理器提供的功能 …… 187
7.2 内存管理的简单方案 …… 189
7.3 内存分配方案 …… 192
7.3.1 固定尺寸分区 …… 192
7.3.2 变长分区 …… 193
7.3.3 结束 …… 195
7.4 分页虚拟内存 …… 196
7.4.1 页表 …… 197
7.4.2 支持分页的硬件 …… 199
7.4.3 页表的规模 …… 199
7.4.4 虚拟和物理内存的相对大小 …… 200
7.5 分段虚拟内存 …… 200
7.5.1 支持分段的硬件 …… 204
7.6 分页和分段的比较 …… 204
7.6.1 解读 CPU 生成的地址 …… 206
小结 …… 207
历史回顾 …… 208
MULTICS …… 209
Intel 的内存体系结构 …… 210
练习题 …… 211
参考文献注释和扩展阅读 …… 212

第 8 章 页式内存管理 …… 213

8.1 按需分页 …… 213
8.1.1 按需分页的硬件 …… 213
8.1.2 页错误处理程序 …… 214
8.1.3 按需分页内存管理的数据结构 …… 214
8.1.4 页错误解析 …… 215
8.2 进程调度器和内存管理器间交互 …… 217
8.3 页替换策略 …… 218
8.3.1 Belady 的 Min 算法 …… 219
8.3.2 随机替换 …… 219
8.3.3 先进先出策略 …… 219
8.3.4 最近最少使用策略 …… 221
8.3.5 第二次机会页替换算法 …… 223
8.3.6 页替换算法回顾 …… 225
8.4 优化内存管理 …… 225
8.4.1 空闲页缓池 …… 225
8.4.2 颠簸 …… 226
8.4.3 工作集 …… 228
8.4.4 颠簸控制 …… 229
8.5 其他考虑 …… 229
8.6 旁路转换缓存 …… 230
8.6.1 TLB 的地址转换 …… 231

8.7 内存管理的高级话题 …… 232
8.7.1 多级页表 …… 232
8.7.2 局部页表项的访问权限 …… 234
8.7.3 反向页表 …… 234
小结 …… 234
练习题 …… 234
参考文献注释和扩展阅读 …… 236

第 9 章 分级存储体系 …… 237

9.1 缓存的概念 …… 238
9.2 局部性原理 …… 238
9.3 基本术语 …… 238
9.4 多级存储层次 …… 239
9.5 缓存结构 …… 241
9.6 直接映射缓存结构 …… 241
9.6.1 缓存查找 …… 243
9.6.2 缓存项中的字段 …… 244
9.6.3 对于直接映射缓存的硬件 …… 245
9.7 流水线处理器设计的影响 …… 247
9.8 缓存读/写算法 …… 247
9.8.1 CPU 对缓存的读访问 …… 248
9.8.2 CPU 对缓存的写访问 …… 248
9.9 处理器流水线中的缓存缺失处理 …… 251
9.9.1 在流水线性能上缓存缺失对内存延迟的影响 …… 252
9.10 利用空间局部性提高缓存性能 …… 253
9.10.1 增加块大小对性能的影响 …… 256
9.11 灵活的布局策略 …… 257
9.11.1 全相关缓存 …… 258
9.11.2 组相关缓存 …… 259
9.11.3 组相关的极端情况 …… 261
9.12 指令和数据缓存 …… 263
9.13 降低缺失损失 …… 264
9.14 缓存替换策略 …… 264
9.15 缺失类型简要说明 …… 266
9.16 TLB 和缓存结合 …… 268
9.17 缓存控制器 …… 269
9.18 虚拟索引物理标记的缓存 …… 270
9.19 缓存设计因素概述 …… 271
9.20 主存的设计因素 …… 272
9.20.1 简单的主存 …… 272
9.20.2 与缓存块大小相匹配的主存和总线 …… 273
9.20.3 交错式内存 …… 273
9.21 现代主存系统分析 …… 274
9.21.1 页式 DRAM …… 278
9.22 分级存储体系的性能影响 …… 279
小结 …… 280
现代处理器的分级存储体系（一个例子） …… 281
练习题 …… 281
参考文献注释和扩展阅读 …… 283

第 10 章 输入/输出和稳定性存储 …… 284

10.1 CPU 和 I/O 设备间的通信 …… 284
10.1.1 设备控制器 …… 284
10.1.2 内存映射 I/O …… 285
10.2 程控 I/O …… 287
10.3 DMA …… 288
10.4 总线 …… 290
10.5 I/O 处理器 …… 291
10.6 设备驱动 …… 292
10.6.1 例子 …… 293
10.7 外围设备 …… 295
10.8 磁盘存储器 …… 296
10.8.1 磁盘技术的传奇故事 …… 302
10.9 磁盘调度算法 …… 304
10.9.1 先到先服务 …… 305
10.9.2 最短寻道时间优先 …… 305
10.9.3 SCAN …… 305
10.9.4 C-SCAN …… 306
10.9.5 LOOK 和 C-LOOK …… 307
10.9.6 磁盘调度总结 …… 307
10.9.7 算法比较 …… 308
10.10 固态硬盘 …… 309
10.11 I/O 总线和设备驱动的演化 …… 310
10.11.1 设备驱动的动态负载 …… 311
10.11.2 信息汇总 …… 312
小结 …… 314
练习题 …… 314
参考文献注释和扩展阅读 …… 315

第 11 章　文件系统……317
11.1　概述……317
11.2　在嵌入式系统上实现文件系统的设计选择……321
11.2.1　连续分配……322
11.2.2　带有逻辑区段的连续分配……324
11.2.3　链接分配……324
11.2.4　文件分配表……325
11.2.5　索引分配……327
11.2.6　多级索引分配……328
11.2.7　混合索引分配……328
11.2.8　不同分配策略的比较……331
11.3　信息汇总……331
11.3.1　索引节点……336
11.4　文件系统的组件……336
11.4.1　创建、写入文件的剖析……337
11.5　各种子系统的交互……337
11.6　文件系统在物理媒介上的布局……340
11.6.1　内存中的数据结构……342
11.7　处理系统崩溃……343
11.8　其他物理媒介上的文件系统……343
11.9　现代文件系统一览……344
11.9.1　Linux……344
11.9.2　Microsoft Windows……348
小结……349
练习题……350
参考文献注释和扩展阅读……352
第 12 章　多线程编程与多处理器……353
12.1　为什么需要多线程……353
12.2　线程所需的编程支持……354
12.2.1　线程创建和终止……354
12.2.2　线程之间的通信……356
12.2.3　读/写冲突、竞争条件及不确定性……357
12.2.4　线程之间的同步……361
12.2.5　线程库中数据类型的抽象表示……365
12.2.6　简单的线程示例……366
12.2.7　死锁和活锁……369
12.2.8　条件变量……370
12.2.9　视频处理示例的完整解决方案……373
12.2.10　解决方案的讨论……374
12.2.11　重新检查条件……375
12.3　线程函数调用和多线程编程概念总结……377
12.4　线程编程的一些注意事项……379
12.5　使用线程作为软件结构抽象……379
12.6　POSIX pthread 库调用总结……379
12.7　操作系统对线程的支持……382
12.7.1　用户级线程……383
12.7.2　内核级线程……385
12.7.3　Solaris 线程：一个内核级线程例子……386
12.7.4　线程和库……387
12.8　在单处理器上的多线程的硬件支持……388
12.8.1　线程创建、终止以及线程间的迁移……388
12.8.2　线程之间的同步……388
12.8.3　原子的 Test-and-Set 指令……388
12.8.4　使用 Test-and-Set 指令的 Lock 算法……390
12.9　多处理器……391
12.9.1　页表……391
12.9.2　分级存储体系……391
12.9.3　保证原子性……393
12.10　高级话题……393
12.10.1　操作系统话题……393
12.10.2　架构话题……403
12.10.3　未来之路：多核与众核架构……412
小结……413
历史回顾……414
练习题……415
参考文献注释和扩展阅读……417
第 13 章　网络与网络协议基础知识……419
13.1　预备知识……419

13.2 基本术语……419
13.3 网络软件……423
13.4 协议栈……424
13.4.1 因特网协议栈……424
13.4.2 OSI 模型……426
13.4.3 分层的实际问题……427
13.5 应用层……427
13.6 传输层……428
13.6.1 停止并等待协议……429
13.6.2 流水线协议……431
13.6.3 可靠的流水线协议……432
13.6.4 处理传输错误……436
13.6.5 因特网上的传输协议……437
13.6.6 传输层总结……438
13.7 网络层……439
13.7.1 路由算法……439
13.7.2 因特网寻址……444
13.7.3 网络服务模式……446
13.7.4 网络路由与转发……449
13.7.5 网络层总结……450
13.8 链路层和局域网……450
13.8.1 以太网……451
13.8.2 CSMA/CD……451
13.8.3 IEEE 802.3……453
13.8.4 无线局域网与 IEEE 802.11……453
13.8.5 令牌环……454
13.8.6 其他链路层协议……456
13.9 网络硬件……456
13.10 协议栈各层之间的关系……460
13.11 用于数据包传输的数据结构……460
13.11.1 TCP/IP 包头……461
13.12 消息传输时间……462
13.13 协议层功能总结……466
13.14 网络软件与操作系统……466
13.14.1 套接字层……467
13.14.2 在操作系统中实现协议栈……468
13.14.3 网络设备驱动程序……468
13.15 使用 UNIX 套接字进行网络编程……469
13.16 网络服务与高层协议……474
小结……475
历史回顾……475
练习题……480
参考文献注释和扩展阅读……482

第 14 章 尾声：旅途回顾……483

14.1 处理器设计……483
14.2 进程……483
14.3 虚拟内存系统和内存管理……483
14.4 分级存储体系……484
14.5 并行系统……484
14.6 输入/输出系统……484
14.7 永久性存储……484
14.8 网络……485
结束语……485

附录 A 使用 UNIX 套接字进行网络编程……486

参考文献……495

索引……500

概　述

计算机无处不在，从手机到汽车、笔记本电脑、桌面电脑，再到 Google、eBay 和 Amazon 等搜索引擎背后的机器。计算机系统架构就是与如何设计上面每一类计算机器有关的内容。在计算技术的早期阶段，软件系统和硬件设计之间有清晰的区分。然而，若干因素正使得这种区分既不实际，也缺乏效率。从硬件方面来看，两项最重要却又相互纠缠的进展是芯片功耗和多核处理器。在过去 40 年间，处理器的处理能力一直在不断增长，实现了 Intel 的联合创始人 Gordon Moore 在 1965 年给出的预测：芯片密度（间接地，指处理速度）大概每两年翻一倍。芯片密度和处理速度增加的副产品是芯片的功耗也相应增加。因此，近年来系统架构师的主要精力同在设法将更多处理器放在同一芯片上以提高对增加的芯片密度的利用效率。这项技术用计算机技术的行话来说叫做多核（每个“核”都是一个独立的处理器）。与此同时，软件技术的复杂性也在增长：如今计算技术已经渗透到我们日常生活的方方面面。从软件方面来看，应用程序的复杂性、增长的交互性、实时响应，以及从一开始而非事后才需要考虑并行性是几个重要的因素。这些软件和硬件方面的发展意味着再也不能将对方视为黑盒。我们迫切需要培养新一代的系统架构师，能够理解系统软件和计算机体系结构之间的相互关系。

不管最后的职业追求是什么，我们越早将这种软硬件之间的互动介绍给学生，他们越能够作为计算机科学家更好地武装自己。



图 1-1　盒子里有什么

1.1　盒子里有什么

计算机是由处理器（也叫做中央处理单元，Central Processing Unit，CPU）、内存子系统、各种各样的外设（例如键盘、显示器、鼠标、硬盘，以及 DVD 播放器[⊖]）以及能够让你将盒子与外部世界相连接的网络接口组成的。另外计算机的系统软件（例如操作系统、编译器以及高级编程语言的运行时系统）能够让你在应用层做想做的事情。在本书中，我们会经常用“盒子”[⊜]来代表刚刚定义的计算机系统。

1.2　计算机系统中的抽象层次

让我们来看一个你可能熟悉的应用，比如说谷歌地球（Google Earth，见图 1-2）。你可以在图形用户界面（Graphical User Interface，GUI）的帮助下通过在地球表面上移动鼠标来在整个地球表面上浏览。你可以将鼠标移动到任何感兴趣的区域（比如说珠穆朗玛峰），然后单击

⊖ DVD 代表数字多用光盘（Digital Versatile Disk）或数字视频盘（Digital Video Disk），它使用光学技术存储海量多媒体数据，如电影文件。

⊜ 图 1-1 是一张有趣的图片，展现了人们在打开盒子往里看时的情景。

鼠标。你马上就可以在屏幕上看到世界上最高山脉的 3D 模型、卫星图片以及该区域的一些
[2] 照片等。在盒子里发生什么事情才能带给你这样的视觉体验呢？



图 1-2 谷歌地球的屏幕截图[○]

再考虑一个更复杂一些的例子，一个叫做“棒球”的多人视频游戏（见图 1-3）。游戏的目标非常简单，即比别的队获得更多的分数以赢得比赛。但是，实际比赛需要复杂的规则和处罚措施。

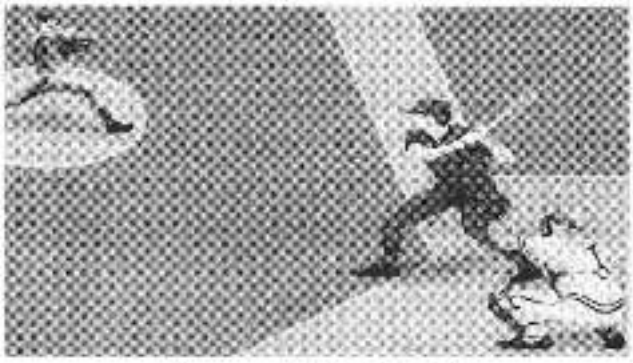


图 1-3 一个视频游戏（棒球）

我们考虑开发这个应用的软件体系结构。想象有一个（逻辑上的）中心软件组件，我们称其为服务器，它负责维护游戏的状态。每个玩家由一个软件组件代表，我们称其为客户端。由于这是一个多人视频游戏，客户端和服务器并不在同一台机器上执行，它们在由局域网连接起来的不同机器上执行。这类应用程序很自然地应该用高级语言（High Level Language，HLL）编程。

[3] 我们可能会给正在设计的游戏添加一些音效/视频的内容。正如你在图 1-4 中所见，要让视频游戏软件能运行，除了我们自己编写的代码（在图右侧的灰色方框中），还需要很多其他部分的协同。CPU 显然不懂机器语言外的任何其他语言，因此编译器必须将高级语言程序翻译成处理器能理解的指令集，程序才能在处理器硬件上执行。

现在我们来自底向上地了解一下处理器（图 1-4 的右侧）。在抽象层次的最底层，是构成半导体基底的电子和洞穴。晶体管抽象层给电子和洞穴的狂野世界带来了秩序。逻辑门由晶体管构成。组合和顺序逻辑单元是由基本的逻辑门组成的，并进一步组织成数据通路（datapath）。有限状态自动机控制着数据通路以实现处理器指令集体系结构中指令的能力。因

○ ©2010 Google Earth.

而指令集是软件和硬件的交汇点。处理器需要面向指令集生成可以在处理器上运行的代码；软件并不关心指令集在硬件上是如何实现的。类似地，硬件实现并不关心处理器上运行的程序是什么，而只是简单地在硬件上履行指令集体系结构所指定的软件和硬件之间的“合同”。

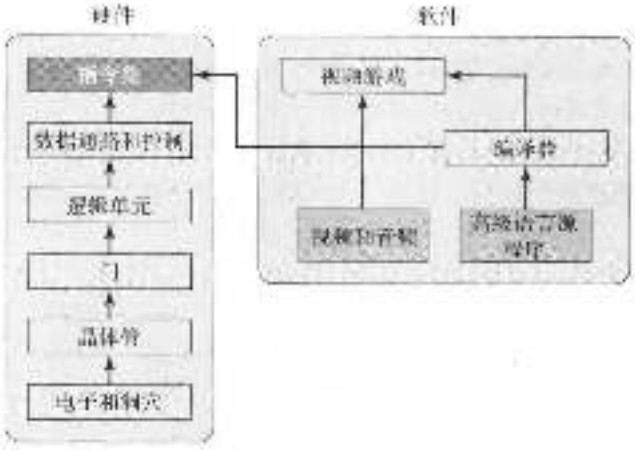


图 1-4 硬件/软件接口。左半边展示了硬件的抽象层次，从底部的电子和洞穴到顶部的指令集。指令集是硬件和软件之间的“合同”。右半边呈现了视频游戏这样的应用从概念到实现所需的软件组件

4

正如你看到的，连续的抽象层次（指令集、数据通路和控制、逻辑单元、门以及晶体管）允许我们使用高级语言程序控制半导体基层上的电子和洞穴的概率性行为。图 1-5 展示了联网的视频游戏是如何通过这些抽象层次来控制半导体基层上的电子和洞穴的。这就是抽象的威力。抽象是处理系统复杂性的一种核心方法，而不管是软件子系统还是硬件子系统。图 1-4 和图 1-5 都展示了通过一系列抽象层将高级语言程序转化为可以在处理器上执行代码的概念性步骤。

现在我们回到联网视频游戏的例子，了解一下操作系统在游戏开发生命周期以及玩家真正玩游戏时的作用。

1.3 操作系统的作用

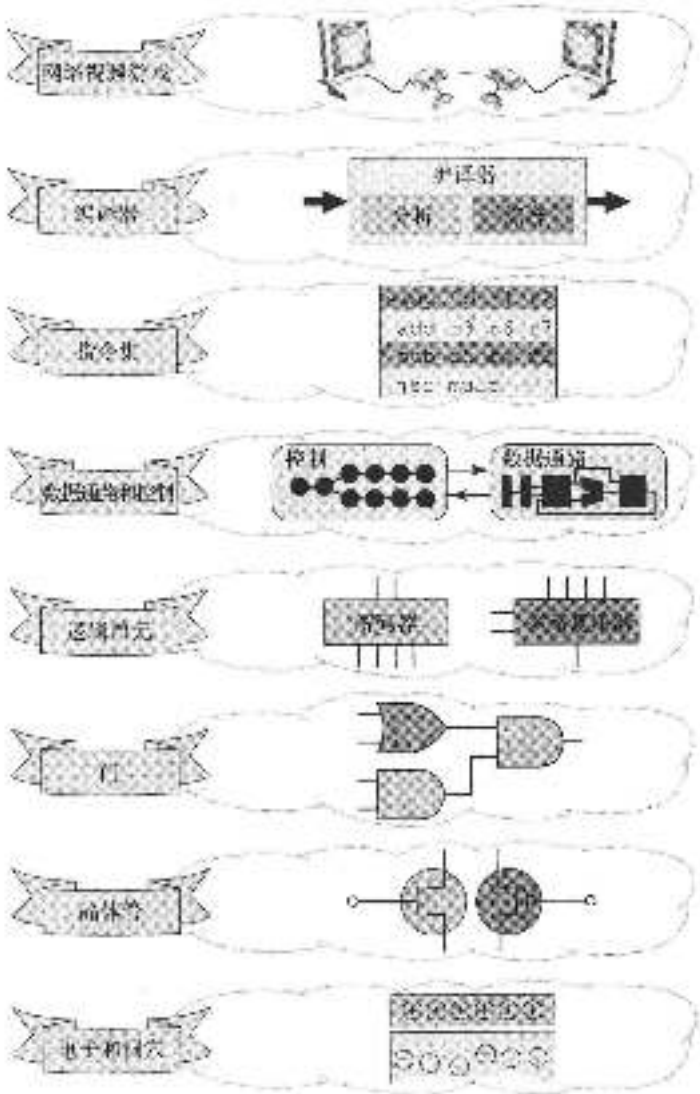
操作系统在网络视频游戏开发和实际使用中的角色是什么？操作系统是资源管理器，负责协调从游戏设计到实际运行游戏的过程中全部行为的硬件资源使用。

我们用联网视频游戏作为例子来理解程序的开发和部署生命周期。我们已经使用高级语言编写了客户端－服务器程序。我们可以用简单的文本编辑器，也可以使用复杂的程序开发工具如 Visual Studio 来开发视频游戏。一旦游戏开发完毕，我们就将程序编译成处理器的指令集。文本编辑器、编译器以及在程序开发中需要用到的其他程序工具都需要在处理器上运行。例如，编译器必须在处理器上运行，将高级语言程序作为输入，输出机器语言代码。操作系统要为每个程序的开发过程分配处理器资源。现在让我们来看看在玩游戏的时候会发生什么事情。

在视频游戏中，单击鼠标按键使得击球手三振出局，并显示在你的屏幕和其他每个玩家的屏幕上（见图 1-6）。这其中发生了什么事情？首先，你的计算机里的硬件设备控制器记录

了你的鼠标单击行为。控制器随后产生了一个处理器中断。请记住处理器正在执行你的游戏客户端程序。中断是一种硬件机制，通知处理器在正执行的程序之外发生了需要关注的事情。这种机制有点类似于房间的门铃。必须有人去看谁在按门铃，他要做什么。操作系统（也是一组程序）将自己调度到处理器上运行，以回应按铃。操作系统回应按铃，发现其来自鼠标，其目标是游戏客户端程序，并将这个中断传递给客户端程序。客户端程序将这个中断打包成一个消息，通过网络发送给服务器端程序。服务器端程序处理这个消息，使用新的消息更新游戏状态。客户端程序通过其各自的操作系统，更新各自的屏幕以反映新的世界的状态。我们可以看到，在从鼠标单击到显示屏幕更新的这段时间里，系统分配和释放了多种硬件资源（处理器、程序和数据所需的内存、鼠标、显示器、网络连接等）。操作系统负责协调所有这些动作。

5



6

图 1-5 从电子和洞穴到多用户视频游戏。视频游戏应用通过硬件抽象的各个层次驱动电子和洞穴完成它所希望的操作。

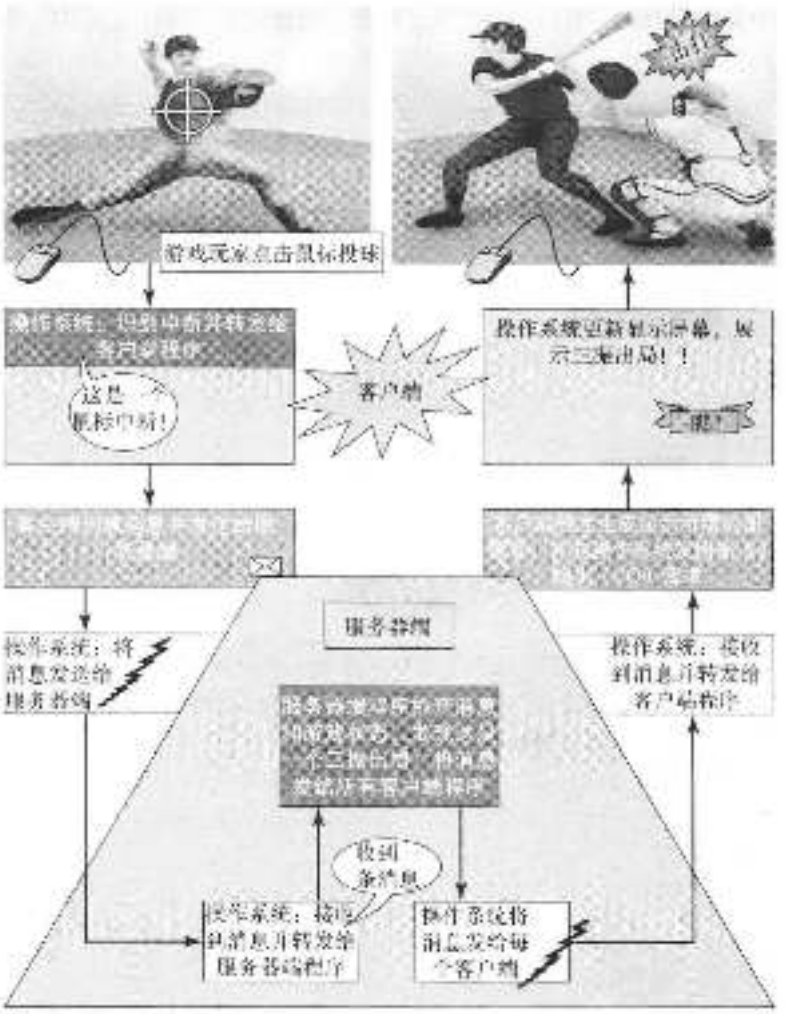


图 1-6 分布式视频游戏中的应用、硬件、操作系统交互。交互发生在客户端和服务器端的应用程序、操作系统以及下层执行的硬件之间

1.4 盒子里正在发生什么事

可以把视频游戏的例子作为问题来驱动我们进一步理解应用程序、操作系统和硬件之间的交互。对我们来说要更好地理解盒子里正在发生什么事，需要很好地掌握系统软件和硬件体系结构的行为。

首先，理解计算机系统有多种实现形式是有益的。计算机系统的实现形式包括手持设备（如手机或个人数字助理（PDA）[⊖]）、平板电脑、笔记本电脑、桌面电脑、并行计算机、集群计算机以及超级计算机等，如图 1-7 所示。

尽管这些计算机系统的外观和大小不同，但其内部的硬件组织结构在很大程度上是相同的。其中包括一个或多个中央处理单元（CPU）、内存以及输入/输出设备。将这些部件连接起来的管道叫做总线（bus）。设备控制器则在 CPU 和相关外设之间起中介作用。这些计算机的计

⊖ PDA（Personal Digital Assistant）即个人数字助理，现来泛指手机、掌对机等。

算能力、内存容量以及输入/输出（I/O）设备的种类和数量可能有很大不同。例如，PDA 拥有与其用途匹配的有限 I/O 能力，包括触摸屏、麦克风以及扬声器。用来运行大规模科学计算应用（如气候变化建模）的高端超级计算机则可能包括成千上万个 CPU、多达数个 TB[○]的内存和具有 PB 级存储容量的磁盘库列[○]。图 1-8 展示了典型的桌面计算机系统的硬件组织。



图 1-7 从 PDA 到超级计算机：各种计算机系统的实例，从手持设备到占据了整层建筑空间的超级计算机，例如像 Yahoo 和 Google 这样的搜索引擎公司或是在国家实验室里进行气候变化模拟的服务器集群

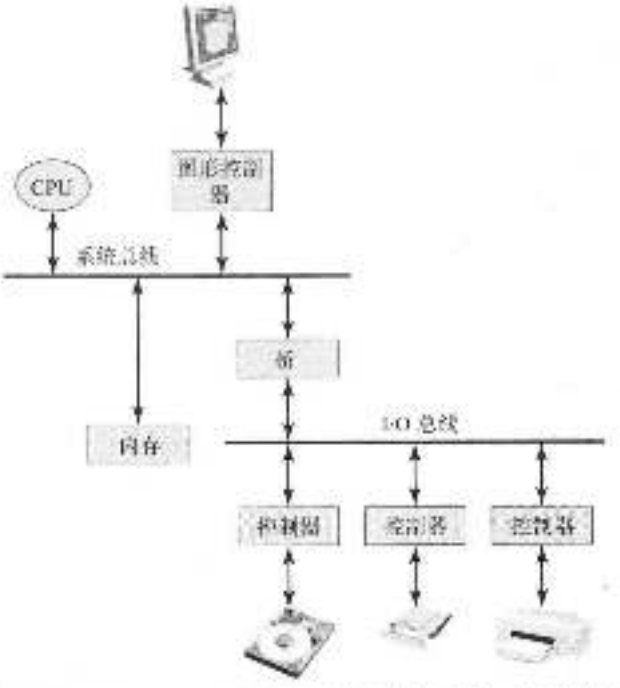


图 1-8 桌面计算机的硬件组织。尽管有多种多样的计算平台，但计算机系统的基本组成是非常相似的。注意这种组织形式使得可以在硬件组件上同时支持多个操作

○ 1 TB（Terabyte）= 2^{40} 字节（有时，1TB 也用来表示 10^{12} 字节）。

○ 1 PB（Petabyte）= 2^{50} 字节（有时，1PB 也用来表示 10^{15} 字节）。

计算机的硬件组织形式揭示了在硬件单元上同时进行操作的可能性（即并发性）。例如，在打印机打印文档的时候，硬盘可以读取 MP3[⊖]文件以播放你喜欢的音乐，而此时你正在用 Web 浏览器阅读 CNN[⊜]的新闻故事。CPU 是整个系统的大脑。计算机系统里发生的每件事都是在 CPU 上运行某些程序的结果。你可能会观察到，在你从计算机屏幕上看 CNN 的同时，文档编辑程序正通过打印机打印你的文档。Web 浏览器是一个应用程序，文档编辑器也是应用程序。操作系统为每个应用程序分配 CPU 时间，以触发其动作。因此，图 1-8 所示的计算机组织方式所支持的并发性在实际中得到了实现。

8-9

1.4.1　在计算机上启动应用程序

让我们来理解图 1-8 中的各个部件如何整合起来，与操作系统一起为你提供简单的计算体验——比如在显示设备上看视频。下面的描述为了讲解方便特意进行了简化。图中标有“内存”的方框存储着要在 CPU 上执行的所有程序。在没有任何用户程序的情况下，操作系统（其自身也是一个程序）总是在 CPU 上执行，并随时准备执行用户想要计算机系统执行的任务。首先，使用鼠标在显示设备上点击标有“电影播放器”的图标。鼠标的移动和鼠标点击都被操作系统接收，操作系统通过被点击的图标得知用户要执行的是哪个程序。所有的程序都在某个存储设备上保存着，最常见的情况是保存在计算机的硬盘上。操作系统将电影播放器的可执行镜像“加载”到内存中并将 CPU 的控制权转移以启动这个程序的执行。

电影播放程序的执行结果是，显示器上打开了一个图形窗口，并请你指定要看的电影文件。你可能会使用键盘来打出文件的名称，包括文件所在的盘符（例如 DVD 驱动器）。该程序在 DVD 驱动器上打开文件并播放，现在你可以在显示设备上观看喜爱的电影了。操作系统参与了给你提供观影体验的每一个步骤，包括：（a）刷新图形显示，（b）捕获用户的键盘输入并转交给电影播放器程序，（c）将数据从诸如 DVD 驱动器之类的存储设备移动到内存中。将数据在 I/O 设备和内存之间移动的实际机制取决于设备的特性。我们将在第 10 章介绍 I/O 子系统的时候在这方面展开更详细的讨论。

10

图 1-8 中的 I/O 总线和系统总线的作用是作为多种硬件单元之间数据移动的通道。正如高速公路和地面道路有不同限速一样，这些不同的总线在传输数据的时候也会有不同的速度特性。图 1-8 中标有“桥”的方框就是用于平滑计算机系统组成中不同通道的速度差异。

1.5　计算机硬件的演化

现在计算在日常生活中已无处不在，很难想象计算机还是稀罕物的时代。但要达到你现在花不到 1000 美元买的笔记本电脑的计算能力，不久之前要花费 100 万美元，需要一个大舞池的空间，还需要精心地制冷以及垫高的地板以便走线。

在 20 世纪 40 年代早期，ENIAC（Electronic Numerical Integrator and Computer，电子数值积分器和计算机）在宾夕法尼亚大学建成。ENIAC 被广泛认可为第一台可编程电子数字计算机（见图 1-9）。

ENIAC 由 18 000 只真空管[⊜]和 1000 位铁氧体磁芯（通常称作“磁芯存储器”）组成的随机存储器，功率大约是 170 千瓦，其计算能力与今天的音乐贺卡计算能力相当！可以看出在

⊖ MP3 表示 MPEG-1 音频层 3，是存储数字音乐的事实标准。

⊜ CNN 是一家总部在美国亚特兰大市的新闻网公司。

⊜ 真空管，由密封在小真空腔管（通常是玻璃制成）中的电线组成，在半导体革命前被用作数字开关设备。

ENIAC出现后60多年的时间内，计算技术的发展有多快。



图1-9 第一台电子数字计算机ENIAC[⊖]。由美国军方资助并在宾夕法尼亚大学秘密建造，是世界上第一台计算机，主要通过计算来支持二次大战中盟军的行动

[11]

计算机硬件技术的高速发展归功于物理、化学、电气工程、数学和计算机科学等多个领域的科学家和工程师的聪明才智。当然，半导体革命是推动计算机工业飞速发展的最显著的技术因素。20世纪40年代，数字计算机刚刚出现时使用真空管和磁芯存储器。1947年，贝尔实验室发明了一种叫做晶体管[⊜]的开关设备，半导体革命开始初现端倪。随后，使用分立晶体管建造数字计算机的方法让位于将多个晶体管集成在一个硅片上。微芯片的出现——20世纪80年代和90年代，基于CMOS晶体管[⊝]利用大规模集成（Very Large Scale Integration，VLSI）技术制成的单芯片处理器——可能是计算机硬件革命的引爆点（见图1-10）。今天，从手机到超级计算机的每个计算设备都使用微芯片作为基本构建单元，半导体内存（通常容量为数百兆乃至千兆字节）已经完全代替了磁

[12] 芯存储器（见图1-11）。



图1-10 从真空管到晶体管到微芯片，基本开关单元的演化使得单个处理器从整个房间大小缩小到硅片大小

⊖ 图片获得了宾夕法尼亚大学工程和应用学院的使用许可。

⊜ 1956年，晶体管的发明人John Bardeen、Walter H. Brattain和William Shockley由于在贝尔实验室所做的开创性工作获得了诺贝尔物理学奖。

⊝ CMOS（Complementary Metal-Oxide-Semiconductor）即互补型金属氧化物半导体，在集成电路（IC）上该技术广泛用于实现晶体管。

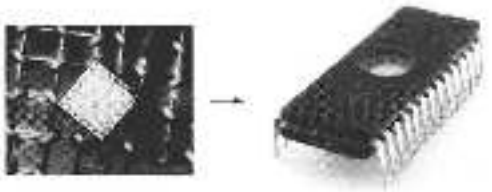


图 1-11 从磁芯存储器到半导体内存。内存技术的进化使得一块面积相当大的芯片可以存储数百万位

1.6 操作系统的演化

操作系统的演化与处理器的演化以及围绕处理器构建的计算机系统的演化相吻合。操作系统在 20 世纪 50 年代出现，例如 FMS（Fortran Monitoring System，Fortran 监控系统）和 IBSYS（IBM 7094 操作系统）。今天，操作系统存在于如图 1-7 所示的多种计算设备上。微软的 Windows 和 Mac OS 主导了桌面计算机市场。Linux 则在企业市场站稳了脚跟。嵌入式设备如手机或个人数字助理（PDA）有它们自己独特的需求，因此出现了满足其需求的专业化操作系统。专业化嵌入式操作系统的例子包括 Symbian OS（塞班操作系统）和 Blackberry OS（黑莓操作系统）。许多嵌入式操作系统是桌面操作系统的衍生物，例如 iOS 和 Windows CE。

可以根据操作系统所支持的计算机系统以及用户巨大增长的期望来追踪操作系统的演化。批处理操作系统支持大型主机系统。多任务操作系统能够更好地利用大型机和小型机的硬件资源。分时操作系统则用于满足用户交互式使用计算机系统的期望。随着个人计算机和图形用户界面的出现，PC 操作系统中集成了图形用户界面，例如微软的 Windows 95 及其后继者[9]。

最终，操作系统要向用户提供计算资源，如处理能力、内存、存储以及其他 I/O 设备等。最近出现的趋势是通过因特网访问这些计算资源。网格计算是这种趋势的开始，它是一个纯粹的科研行为，目标是通过因特网在不同管理主体之间共享高性能计算资源。网格计算这个词源自电力通过电网来传输并进入千家万户，象征着计算能力也应该像电力一样随处可得。今天，一些公司如亚马逊和微软正在通过 Web 提供计算资源（处理能力和存储）。云计算是一个商业界的流行语，用来描述这种给最终用户提供计算资源的新方式。 13

1.7 本书导读

对计算机系统爱好者来说，这是激动人心的时刻。本概述明确展示了计算机硬件和系统软件之间的紧密关系。与此相对应，在本书的其余部分也以一种集成的方式介绍了有关处理器、内存、I/O、并行系统以及网络的硬件和软件问题。

第一部分 处理器

第 2 ~ 5 章探讨了处理器设计和相关硬件问题。

第 6 章介绍了处理器调度问题以及如何用操作系统解决这些问题。

第二部分 内存子系统

第 7 ~ 8 章提出了内存管理问题，以及操作系统如何在相应的体系结构的支持下解决这

---

9 有兴趣的读者可以看一下名为《书呆子的胜利》（Triumph of the Nerds）的纪录片（由美国公共广播电台 1996 年制作并播出）以了解个人计算机革命：www.pbs.org/nerds/，可以通过 Google 视频找到这部纪录片。

的问题。

第9章介绍了内存层次，特别是关于处理器缓存。

第三部分 I/O子系统

第10章以概念为重点介绍了I/O的一般性问题，即如何与处理器接口。

第11章讨论了文件系统的设计和实现。文件系统是操作系统的一个主要组成部分，用来管理持久化存储。

第四部分 并行系统

第12章介绍了与并行处理器有关的概念、系统软件和硬件问题。

第五部分 网络

第13章讲解了设计网络协议栈时遇到的操作系统问题，以及相关的硬件支持。

练习题

1. 考虑谷歌地球（Google Earth）应用。打开程序，将鼠标放到地球表面上，点击珠穆朗玛峰以仔细观察这个山峰。确定并用非专业术语描述在这一系列动作中操作系统和硬件之间的交互。
2. 高级语言是如何影响处理器体系结构的？
3. 对下面的问题回答是或否，并说明原因：编译器设计者深入了解处理器实现的细节。 |14|
4. 解释计算机中的抽象层次，从硅芯片到复杂的多人视频游戏。
5. 对下面的问题回答是或否，并说明原因：计算机系统内部的硬件组织结构根据系统特性的不同有巨大差别。
6. 图1-8中计算机总线之间的“桥”的作用是什么？
7. 图1-8中“控制器”的作用是什么？
8. 使用因特网，研究并解释计算机硬件演化的5个主要里程碑。
9. 使用因特网，研究并解释操作系统演化的5个主要里程碑。
10. 对比网格计算和电网。解释为什么用电网来比喻网格计算是有道理的。同样，解释它们之间的不同。
11. 连线题。请在左右两边各选择合适的项目连线。

UNIX操作系统	Torvalds
硅芯片	Bardeen、Brattain 和 Shockley
FORTRAN语言	Kilby 和 Noyce
C语言	De Forest
晶体管	Lovelace
世界上最早的程序员	Thompson 和 Ritchie
世界上最早的计算机器	Mauchley 和 Eckert
真空管	Backus
ENIAC	Ritchie
Linux操作系统	Babbage

参考文献注释和扩展阅读

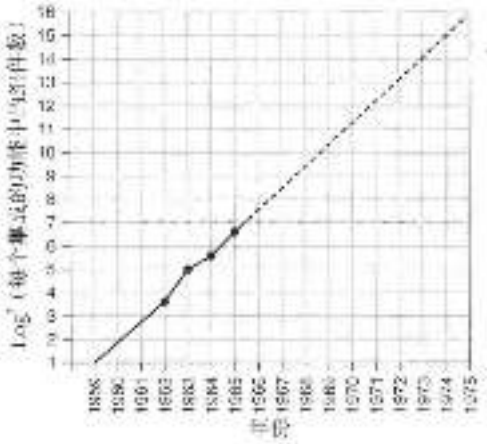
冯·诺依曼结构已经成为各种单片计算机的同义词，它由CPU及存储了指令和数据的内存构成。但计算机历史学家对这两者之间的联系还存在争论。冯·诺依曼模型的概念是由John von Neumann于1945年在一篇广为传播的论文中提出的，这个模型实际上受到了宾夕法尼亚大学的J. Presper Eckert和John Mauchly开发的ENIAC体系结构的启发。在ENIAC之前，一位名叫Alan Turing的数学家在1936

年写了一篇文章，假设了一种“通用计算机器”。这个假设的机器具有无穷内存以存储指令和数据。今天，这个假设的机器以他的名字被命名为“图灵机”，并已成为理论计算机科学的基础。Alan Turing 被认为是“理论计算机科学之父”。每年 ACM 都为在计算机科学领域做出持久性贡献的个人颁发久负盛名的图灵奖。图灵奖被认为与诺贝尔奖齐名。

阅读一些真正经典的文献以了解从前的思维过程总是很有意思。John Backus 和他的团队在 1953 年向他 IBM 的上司提出了 FORTRAN 语言，最早的关于 FORTRAN 语言的论文出现在 1954 年 [Backus，1954]。Backus 于 1977 年因此获得了图灵奖。ENIAC 结构是 Eckert 和 Mauchly 于 1943 年构思的，描述 ENIAC 的文章可以在 IEEE《Annals of the History of Computing》(计算历史年刊) 上找到 [Burke，1981]。 15

Charles Babbage 被认为是“计算机之父”，他生于 1791 年，卒于 1871 年，发明了第一台机械计算机（称作差分机）——使用轮子和齿轮来进行通过穿孔卡输入的简单计算。Ada Lovelace 常被称作世界上第一个计算机程序员，他生于 1815 年，卒于 1852 年，给 Babbage 的分析机（在其一生中从未完成）写了第一个算法。这些都发生在计算机革命的初期。

Jack Kilby 在 1958 年发明的集成电路或微芯片（Robert Noyce 在 6 个月后也独立发明了）则启动了“真正”的计算机革命。在 T. R. Reids 的书中有相应记载 [Reid，2001]。下面展示了 Gordon Moore 于 1965 年在《Electronic》(电子) 杂志上发表的文章中的图表 [Moore，1965] ㊀，显示了他对芯片密度与时间之间关系的函数的预测：



贝尔实验室的 Dennis Ritchie 和 Ken Thompson 开发了 UNIX 操作系统 [Ritchie，1974]。贝尔实验室的 Brian Kernighan 和 Dennis Ritchie 开发了 C 语言 [Kernighan，1978]。荷兰 Vrijie 大学（荷兰自由大学）的 Andrew Tanenbaum 开发了一个开源版本的 UNIX 操作系统，叫做 MINIX（Mini-uNIX 的缩写），并在 1987 年作为其广受推崇的操作系统教材的附录公布 [Tanenbaum，1987]，这促成了一个关于操作系统的用户社区的诞生。一个名叫 Linus Torvalds 的芬兰学生，受到 MINIX 的启发，开发了其自己版本的开源 UNIX 操作系统，并在 1991 年以 Linux 为名在 comp.os.minix 邮件组通过一条简短而低调的消息发布 [Torvalds，1991]。 16

我们用一些近期操作系统的参考文献来结束本章。Windows 7 是微软在 PC 平台上的最新产品 [Windows Version 7，2010] ㊁。微软的 Windows CE（CE 是简洁版的意思，Compact Edition）主要面向嵌入式应用，例如医疗设备、汽车以及移动电话 [Windows CE，2010]。苹果的 PC 平台 [Mac OS X，2010] 和 iPhone[iPhone OS X，2010] 有自己的操作系统。Symbian OS[Symbian OS，2010] 是一个在智能手机上流行的开源操作系统。黑莓 OS 是 RIM 公司为黑莓手机设计的专有软件平台 [Blackberry OS，2010]。 17

㊀ 见 Gordon Moore 在 ISSCC 2003 会议上的大会报告，“没有永远的指数（增长）”，摩尔定律 50 周年，Intel 公司，http://isscs.org/History/MooresLaw.htm。

㊁ 指本书英文版出版时。——译者注

第 2 章

Computer Systems: An Integrated Approach to Architecture and Operating Systems

处理器体系结构

处理器设计围绕两个体系结构的问题：指令集和机器结构。在计算机发展的早期（大约是 20 世纪 60 年代和 70 年代），曾有一段时间处理器设计完全被看作电子工程师的工作。计算机上大规模地使用汇编语言编程。因此，指令集越是花哨，应用程序就越趋向简单。这是当时流行的传统观念。随着现代编程语言的出现——如 20 世纪 60 年代的 Algol 语言——以及编译技术的快速发展，处理器设计显然不再仅仅是一项关于硬件的工作。特别地，指令集的设计与编译器如何有效地为处理器生成代码密切相关。在这个意义上，程序语言对于指令集的设计有相当大的影响。

让我们来了解程序语言是如何影响指令集设计的。高级语言中诸如赋值语句和表达式这样的结构会映射到算术/逻辑指令和加载/存储指令。高级语言支持的数据抽象需要指令集提供不同精度的操作数以及寻址模式。条件语句和循环结构需要条件和无条件跳转指令。更进一步来说，高级语言中像过程这样的模块化结构需要从处理器体系结构中获得附加的抽象支持。

应用对于指令集设计也有着重要的影响。例如，科学计算和工程计算在早期计算中占据主导地位。不应该，20 世纪 70 年代和 80 年代的商用系统在指令集上支持浮点运算。当前某些时候，手机和其他嵌入式系统的计算占了主导地位，随着计算融入社会的各个层面，毫无
[18] 疑问这个趋势将持续下去。音频和视频这样的流媒体应用在手持设备上变得很平常。自然地，这些应用的需求（例如，单独一条指令对许多数据进行操作）开始影响指令集的设计。

在硬件上直接支持一个具体的系统软件或应用的需求并非总是可行或划算的。例如，在计算机发展的早期，低端的计算机通过软件库使用指令集中可用的整数运算来实现浮点运算。直到今天，复杂的操作（例如求余弦）依然不应该由应用处理器的指令集直接支持。替代的方法是，一些专门的系统软件（称为*库*）通过将这些复杂操作映射为指令集中的简单指令来实现它们。

操作系统对于指令集的设计也有影响。一个处理器可能会同时运行多个程序。想想你的台式机或笔记本电脑，上面运行着若干个程序，但是却没有多个处理器。因此，每当我们切换到另一个程序之前，需要记住一个程序正在做什么。你可以想象一个动作麻利的厨师在 4 口锅里炒 4 个不同的菜，她记住了每道菜做到了哪个阶段并适时加入调料。操作系统是这样的一个软件实体（即它自身的一个程序），它像厨师处理不同的菜一样，安排不同的程序在处理器上执行。操作系统自身也对处理器设计有影响，在后面讨论程序不连续性和内存管理的章节中这会变得很显而易见。

2.1 处理器设计涉及什么

通过逻辑设计课程，我们掌握了寄存器、算术/逻辑单元这样的硬件资源，还有将它们连接起来的数据通路。当然，还有用来存放程序和数据的主存储器。在一组输入源中进行选择的多路选择器，连接处理器资源和主存储器的总线，用于将数据从数据通路放到总线上的驱动。我们很快将讨论数据通路。

我们将这些硬件资源比喻为英语中的字母表。单词使用字母表构成英语的字典。类似地，处理器的指令集使用硬件资源来构成处理器。正如自然语言中的单词让我们能够表达不同的思想和情感一样，指令集让我们能够安排处理器中的硬件资源做不同的事情。因此，指令集是区分 Intel x86、PowerPC 等处理器的关键。

作为计算机用户，我们知道可以在不同层次对计算机进行编程：在 C、Python 和 Java 的层次；在汇编语言的层次；直接使用机器语言的层次。

指令集就是计算机体系结构开出的处方，指定了这个计算机需要的能力，指令集应该对 19
机器语言程序员可见。因此，指令集是一个软件（即在计算机任何层级上运行的程序）和实际硬件实现之间的契约。指令集的实现有着许多选择，我们将在后面的章节中讨论这些选择。首先，我们将探索指令集设计的固有问题。

2.2 如何设计指令集

计算机由计算的机器演化而来。在早期的计算机设计中，指令集的选择很大程度上由硬件能否实现这些指令来决定。这是因为当时的硬件非常昂贵，而程序都是直接用汇编语言写成的。因此，指令集的设计很大范围内是电子工程师的工作，因为他们对硬件实现的可行性有非常好的想法。然而，硬件成本降低了，程序设计也日趋成熟，高级语言被开发出来，因此问题从硬件实现的可行性转移到指令是否实际有用上来，即指令是否有利于用高级语言编写出高效而紧凑的程序。

事实证明，指令集负责安排处理器内部做什么，而计算机用户很少需要直接和指令集打交道。毫无疑问，当你在玩视频游戏的时候，你不会关心处理器正在执行什么指令。用汇编语言写程序比用高级语言写程序容易出错是共识。

从人工手写汇编语言程序到编译器将高级语言程序转换为机器代码的变化是指令集体系结构演化的主要原因。这个变化意味着我们将寻求一个简单的指令集使高级语言结构能够转化为高效代码。

这里有一点需要注意，指令集的优雅是非常重要的，体系结构领域在这个方面上做了大量工作。然而，与此同等重要甚至需要首先关注的是指令集实现的效能。具体来说，在努力实现更简单、更快的指令集的时候，指令集体系结构的规律性是一个非常值得考虑的因素。我们在第 3 章和第 5 章讨论实现细节的时候还会再提到这一点。

虽然每种高级语言都有其独特的语法和语义特性，但我们依然可以找出一组在大部分高级语言中都存在的基本功能。我们首先要做的就是找出这样一组功能。我们以编译这样的功能作为讨论和开发指令集的基本动机。正如我们在本章开头提到的那样，除了编译高级语言结构之外，指令集还受到许多其他因素的影响。我们将在 2.11 节讨论这些因素。 20

2.3 常见的高级语言功能集

考虑如下功能集：

1）**表达式和赋值语句** 编译这样的结构揭示了许多指令集体系结构（Instruction-Set Architecture，ISA）中的细微之处，从算术/逻辑操作的种类到一个指令中操作数占的大小和位置。

2）**高级数据抽象** 编译一个简单变量的聚合（在高级语言中常称为结构（structure）或记录（record））揭示出更多 ISA 需要的细节。

3）**条件语句和循环** 编译这些结构使得程序的顺序执行发生变化，并且需要 ISA 有额外

的机制。

4）过程调用 过程让我们能够开发模块化且便于维护的代码。过程调用和返回的编译给指令集的设计带来了新的挑战，包括记录程序执行过程前后的状态、给过程传递参数、接收过程的返回值。

在 2.4 ~ 2.8 节，我们将从有效编译这些功能的角度依次考虑每个功能并开发出 ISA 所需的机制。在 2.10 节，我们通过展示 LC-2200 ISA 来总结前面的讨论。LC-2200 ISA 是一个简单的指令集，将作为后面探索处理器实现细节的基础。

2.4 表达式和赋值语句

我们知道任何高级语言（例如 Java、C 和 Perl）都有算术 / 逻辑表达式和赋值语句：

```
a = b + c; /* b 与 c 相加放入 a*/                    (1)
d = e - f; /* e 减去 f 放入 d */                     (2)
x = y & z; /* y 和 z 做 AND，结果放入 x*/             (3)
```

上面的每条语句都以两个操作数为输入，执行一次操作，然后将结果存到第三个操作数中。考虑在一个处理器指令集中的下面 3 条指令：

```
add a, b, c; a ← b + c                                (4)
sub d, e, f; d ← e - f                                (5)
and x, y, z; x ← y & z                                (6)
```

高级语言结构（1）、（2）、（3）分别直接映射为指令（4）、（5）、（6）。

21 这样的指令称为双操作数（binary）指令，因为它们都利用两个操作数进行工作来产生一个结果。它们也被称为三操作数（three-operand）指令，因为有三个操作数（两个源操作数和一个目的操作数）。是不是每个双操作数指令都需要三个操作数呢？简单地说，不是。在接下来的小节中我们将详细阐述这个问题的答案。

2.4.1 操作数放在哪里

让我们讨论一下上面提到的等式中的变量（*a,b,c,d,e,f,x,y,z*）的位置。图 2-1 是一个简单的处理器模型。

处理器内部是一个算术 / 逻辑单元，或者叫做 ALU，它执行 ADD、SUB、AND 和 OR 等运算。我们现在讨论这些指令的操作数放在什么地方。首先以一个比喻来开始。

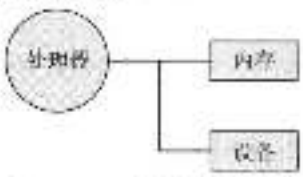


图 2-1 一个基本的计算机组成。处理器包含一个从内存中取指令和操作数的 ALU

假设你有一个工具箱，里面有很多工具。很多工具箱都有一个工具盘（tool-tray）。如果你在做某件事情（比如修理厨房的水龙头），将螺丝刀和水管扳手从工具箱移到工具盘中，然后将工具盘拿到厨房水槽处。接着你会进行修理，完成后将工具盘中的工具放回工具箱中。显然，你不会每次拿工具都跑到工具箱去拿，而是希望这个工具已经在工具盘里面了。换句话说，你通过将需要的最少工具放入工具盘，优化了跑到工具箱边的次数。

我们在设计指令的时候也想这样。我们已经知道术语寄存器用来描述处理器中可用的资源。它们就像内存一样，但是在处理器内部的，所以在物理上（因此也在电子上）更加靠近 ALU，被制造得比内存更快，如图 2-2 所示。因此，如果指令的操作数都在寄存器中，那么取得操作数就会比操作数在内存中的情况要快得多。但这还不是故事的全部。

使用寄存器还有另一个引人注目的原因，尤其对于现代拥有大内存的处理器更是如此。我们将这个问题称为操作数的可寻址性。回到工具箱和工具盘的比喻。假设你经营着一家汽车修理店，现在你的工具箱相当大。你的工作需要工具箱中几乎所有的工具，但会在不同时候用到。因此，在你工作的不同阶段，你会将工具盘中的工具送回工具箱，将下一阶段需要的工具放入工具盘中。显然，每个工具在工具箱中都有唯一的位置，但在工具盘中却没有。实际上你重复使用工具盘上的空间来放工具箱中的不同工具。

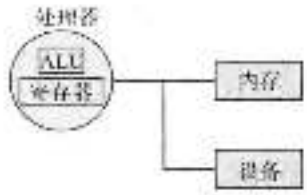


图 2-2 在处理器内部加入寄存器。寄存器与 ALU 的近距离减少了 ALU 取得操作数所需的时间

22

一个对操作数进行唯一寻址的体系结构将面临同样的困境。现代处理器有着非常大的存储系统。随着内存容量的增加，内存地址的大小（即用来表示内存中唯一地址的位数）也相应增加。因此，如果一条指令需要三个内存操作数，那么每条指令的大小都会增加。操作数的可寻址性是个大问题，每条指令都需要占据好几个内存单元以满足对所有内存操作数的唯一命名。

另一方面，有了少数寄存器配合当前程序所需（就像工具盘），我们就可以解决内存可寻址性问题，因为用来表示唯一寄存器地址所需的位数很少。作为内存可寻址性问题的推论，寄存器数量必须较小以限制用来寻址寄存器所需的位数（即便是芯片集成技术允许体系结构中包含更多的寄存器）。

所以，我们的指令看起来是这样的：

```
add r1, r2, r3; r1 ← r2 + r3
sub r4, r5, r6; r4 ← r5 - r6
and r7, r8, r9; r7 ← r8 & r9
```

另外，程序常需要使用常量。例如，将寄存器初始化为某个值是某个例程的要求。满足这个要求的最简单方法就是指令自身的某一部分作为常量。这样的常量值称为立即值。

例如，我们有一条这样的指令：

```
addi r1, r2, imm; r1 ← r2 + imm
```

23

在这条指令中，作为指令一部分的立即值成了第三个操作数。在编译高级语言时立即值非常方便。

例 2-1 给出下面的指令：

```
ADD Rx, Ry, Rz        ;    Rx ← Ry + Rz
ADDI Rx, Ry, Imm      ;    Rx ← Ry + 立即值
NAND Rx, Ry, Rz       ;    Rx ← NOT(Ry AND Rz)
```

如何达到下面这条指令的效果？

```
SUB Rx, Ry, Rz        ;    Rx ← Ry - Rz
```

答：

```
NAND Rz, Rz, Rz       ;    将 Rz 变为 Rz 的反码
ADDI Rz, Rz, 1        ;    将 Rz 变为 Rz 的补码
                      ;    现在 Rz 其实包含了 -Rz
ADD Rx, Ry, Rz        ;    Rx ← Ry + (-Rz)
                      ;    后面两条指令恢复了 Rz 的原始值
```

```
NAND Rz, Rz, Rz        ;      将Rz变为Rz内反码
ADDI Rz, Rz, 1         ;      将Rz变为Rz的补码
```

对于这些算术/逻辑运算来说，所有的操作数都在寄存器中。我们引入**寻址模式**的概念，寻址模式指的是在一条指令中如何指定某个操作数。这里使用的寻址模式，所有的操作数都在寄存器中，因此被称为**寄存器寻址**。

关于高级结构（1）、（2）、（3），现在探索程序变量a，b，c，d，e，f，x，y，z和这些处理器寄存器的关系。首先，假设这些变量都在内存中，由编译器放在众所周知的地方。因为变量在内存中但算术/逻辑运算指令只能使用寄存器，因此必须将变量从内存搬到寄存器中。所以，我们需要另外一些指令来将数据在内存和寄存器之间来回搬运。这些指令称为加载（从内存加载到寄存器）和存储（从寄存器存储回内存）指令。

例如，

24

```
ld  r2, b; r2 ← b
st  r1, a; a  ← r1
```

有了加载/存储指令和算术/逻辑指令，现在可以将这样的一个结构

```
a = b + c
```

"编译"为

```
ld  r2, b                    (7)
ld  r3, c                    (8)
add r1, r2, r3               (9)
st  r1, a                    (10)
```

也许有人感到奇怪，为什么不单纯使用内存操作数来避免寄存器的使用呢？毕竟，这一条单独的指令

```
add a, b, c
```

与（7）～（10）所示的4条指令序列相比是如此优雅而有效。

借助于工具箱比喻可以很好地理解其中的原因。你知道自己在工作中要多次使用螺丝刀，因此，并不是每次都到工具箱中取螺丝刀，而是花费一些代价将其移至工具盘中，然后多次重用它，直到将它放回工具箱为止。

内存就像工具箱，而寄存器就像是工具盘。你预计程序中的变量将会在多个表达式中使用。考虑如下的高级语言语句：

```
d = a * b + a * c + a + b + c;
```

可以看到，一旦a，b，c从内存被带到寄存器中，仅仅在这一个表达式求值中就被用了若干次。试着将这个表达式"编译"为指令序列（假设乘法指令有着和加法指令相似的形式）。变量在寄存器中的重用给我们带来了什么呢？答案是速度。正如本节中已经说明的那样，因为寄存器在处理器内部，因此我们访问程序变量的时间与每次都到内存中访问相比缩短了很多。

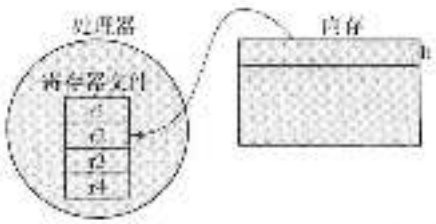


图2-3 从内存中装载寄存器。处理器在装载指令中指明了源内存地址和目标寄存器地址

在一条加载指令中，其中一个操作数是一个内存地址，另一个操作数是这条加载指令的目的寄存器（见图2-3）。同样，在一条存储指令中，目标是一个内存地址。

25

例2-2 一种体系结构有一个称为累加器（ACC）的寄存器，以及操作内存的指令，ACC如下：

```
LD ACC, a       ;       ACC ← 内存地址a的内容
ST a, ACC       ;       内存地址a的内容 ← ACC
ADD ACC, a      ;       ACC ← ACC + 内存地址a的内容
```

使用上面的指令，你如何实现下面的指令的语义？

```
ADD a,b,c; 内存地址a的内容 ← 内存地址b的内容 + 内存地址c的内容
```

答：

```
LD ACC, b
ADD ACC, c
ST a, ACC
```

2.4.2 在指令中如何指定内存地址

我们考虑如何使用指令的一部分来指定内存地址。当然，可以将地址直接嵌入指令中。但是这个途径有一个问题。在2.4.1节中提到，用来表示一个内存地址的位数已经很多了，并且随着内存容量的增大，情况只会变得更糟。例如，如果我们有一个PB级（大约2^{50}字节）的内存，则在指令中需要50位来表示一个内存操作数。进一步，正如将在2.5节中所见的那样，编译高级语言（尤其是面向对象的语言）写的程序时，编译器只知道复杂数据结构（如数组或对象）的每个成员的偏移量（相对于这个结构的地址）。所以，我们引入一种寻址模式来缓解每条指令都需要将整个内存地址操作数放入其中的情况。 26

这样的寻址模式称为**基址加偏移量**（base-offset）模式。在这种寻址模式中，指令中的内存地址为一个寄存器（基址寄存器）的内容与一个偏移量（以立即值形式包含于指令中）的和。通常表示为

```
ld  r2, offset(rb);     r2 ← MEMORY[rb + offset]
```

如果rb包含变量b的内存地址，而偏移量为0，则上面的指令等价于将程序变量b加载到寄存器r2中。

注意，rb可以是处理器中的任意一个寄存器。

如前所述，基址加偏移量寻址模式的威力在于它可以用来加载和存储简单的变量，很快我们将会看到，它还可以用于复合变量（比如数组和结构）的元素。

例2-3 给出下列指令：

```
LW Rx, Ry, OFFSET       ;       Rx ← MEM[Ry + OFFSET]
ADD Rx, Ry, Rz          ;       Rx ← Ry + Rz
ADDI Rx, Ry, Imm        ;       Rx ← Ry + 立即值
```

现在要实现一种新的寻址模式，称为自动递增寻址，用于具有下列语义的加载指令：

```
LW Rx, (Ry)+            ;       Rx ← MEM[Ry];
                        ;       Ry ← Ry + 1;
```

请给出一个解答，用给出的指令来实现上述LW指令。

答：

```
LW Rx, Ry, 0            ;       Rx ← MEM[Ry + 0]
ADDI Ry, Ry, 1          ;       Ry ← Ry + 1
```

2.4.3 每个操作数应该有多宽

操作数的宽度与其精度或者说动态有关。为了回答这个问题，我们需要回顾高级语言及
27 其支持的数据类型。我们用C语言作为一种典型的高级语言代表。C语言中的基本数据类型有short、int、long、char。这些数据类型的宽度与实现相关，一般来说，short是16位，int是32位，char是8位。我们知道char数据类型在C中用来表示字母数字字符。char类型宽8位的背后有着历史原因。为了在计算机和通信设备之间交换信息，人们使用ASCII码作为字母数字字符（打字机上能找到的那些符号）数字编码的标准。ASCII码使用7位来表示一个字符。也许有人觉得char数据类型应该是7位宽。然而，在C语言诞生的年代，流行的指令集都使用8位宽的操作数，所以对于C语言来说，使用8位的char类型非常方便。类似地，int数据类型为32位宽的原因是32位的处理器体系结构十分常见。

下一个问题是选择每个操作数的精度。这依赖于数据类型所需的精度。我们先给出数据精度的非正式定义。假设你在程序中需要一个变量x来存储取值范围在0～255的无符号整数，那么仅需要8位来表示这样一个变量。因此，x所需的精度是8位。类似地，如果在你的程序中有一个有符号整数y取值范围在-2^{15}～$2^{15}-1$，那么需要32位精度来表示这样一个变量（假设使用补码表示）。高级语言中的数据类型（如C中的int、short、char）给程序员提供了为不同需求的变量定义不同数据精度的灵活性。你可能好奇于为什么会提供这样的定制而不是简单地采取体系结构所支持的最高精度。答案是，这是一个在时间和空间上进行优化的机会。程序变量的精度越低，在内存中占用的空间就减小。此外，对于精度要求较低的算术/逻辑运算来说，在处理器和内存之间来回运送操作数也会有一定的时间优势，对于浮点算术运算来说尤为如此。所以，为了空间和时间上的优化，最好是指令中操作数的精度恰好满足数据类型的需求。这也解释了为什么处理器在指令集上支持多种精度，即字、半字和字节。字精度通常指的是体系结构在硬件上对算术/逻辑运算支持的最高精度。具体的精度类别允许在时间和空间上进行优化。

为了方便讨论，我们约定一个字是32位，半字是16位，字节是8位。这些精度类别正好对应于大部分C语言实现中的int、short和char类型。这样的选择是基于2009年前后大部分能使用字长都是32位。在2.4.1节中，我们已经介绍了操作数可寻址性的问题。当体系结构支持多种精度后，就出现一个内存操作数的可寻址性问题。这里的可寻址性指的是能够在内存中单独指定的最低精度的操作数。比如，如果一台机器是字节可寻址的，那么能够单独寻址的最低精度就是字节。如果是字可寻址的，那么能够单独寻址的最低精度就是字。我们
28 约定讨论中是字节可寻址的。

因此，一个字在内存中看起来是这样的：



每个字里面有4个字节。（MSB指最高有效字节（most significant byte）；LSB指最低有效字节（least significant byte））。例如，我们在程序中有一个整数变量为

```
0x11223344
```

那么它在内存中看起来是这样的：



每一个字节都能够单独被寻址。想要体系结构上会有在这个级别的精度上进行操作的指令。那么，指令集中应该包含操作不同精度操作数的指令，如下所示：

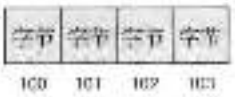
```
ld   r1, offset(rb);  从地址 rb+offset 处装入一个字到 r1 中
ldb  r1, offset(rb);  从地址 rb+offset 处装入一个字节到 r1 中
add  r1, r2, r3;      将寄存器 r2 和 r3 中的字操作数相加并将结果置于 r1 中
addb r1, r2, r3;      将寄存器 r2 和 r3 中的字节操作数相加并将结果置于 r1 中
```

支持多种精度的操作数的体系结构决策和处理器的硬件实现之间是有关系的。硬件实现包括确定数据通路的宽度以及数据通路中各种硬件资源（如寄存器）的宽度。我们将在第 3 章和第 5 章讨论处理器实现的更多细节。因为在讨论中我们认为一个字（32 位）是硬件支持的最大精度，为了方便，假设数据通路是 32 位宽。也就是说，所有的算术/逻辑运算的操作数都是 32 位的。相应地，假设寄存器的宽度是 32 位以恰好满足数据通路的宽度。需要注意的是，寄存器和数据通路的宽度正好与体系结构选择的数据宽度相同并不是必需的，但确实是方便且有效的。与此同时，体系结构和硬件实现也需要为操作较低精度操作数的指令提供一些便利。例如，像 addb 这样的指令是 8 位精度的，它使用源寄存器中的低 8 位进行加运算，然后将结果置于目标寄存器的低 8 位中。

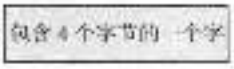
值得注意的是，现代的体系结构已经升级为 64 位整数运算。甚至连 C 语言都引入了 64 位精度的数据类型，但这些数据类型的名字在不同的编译器中可能有些不一致。然而，本章中我们对指令集设计进行的概念上的讨论，与实际硬件支持的精度是完全正交的。 [29]

2.4.4 字节序

字节可寻址的机器中存在着一个有趣的问题，就是一个字中各个字节排列的顺序。在字节可寻址的机器中，一个 4 字节的字，如果从地址 100 开始，那么这个字共实占据了内存中 100、101、102、103 这 4 个连续字节。

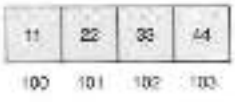


这 4 个字节组合起来就成了地址 100 处的一个字。



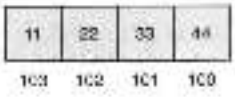
假设 100 处的这个字的值为 0x11223344，那么这 4 个字节在字中有两种可能的组织方式：

组织方式 1：



在这种组织方式中，该字的 MSB（包含 11_{hex}）位于字的地址即 100 上。这种组织方式称为大端模式。

组织方式 2：



在这种组织方式中，该字的LSB（包含44_{hex}）位于字的地址即100上。这种组织方式称为小端模式。

由此可见，字节序是根据哪个字节处于字的地址上来区分的。如果是MSB，就是大端；如果是LSB，就是小端。

原则上，对于使用高级语言编程来说，字节序其实是无所谓的，前提是严格按照所声明的数据类型要求的那样去使用程序中的变量。

[30] 然而，在C这样的语言中，数据类型的使用可能与其声明的不同。

考虑如下的代码片段：

```
int i = 0x11223344;
char *c;

c = (char *) &i;
printf("endian: i = %x; c = %x\n", i, *c);
```

我们来研究一下打印出来的c的值会是什么。这依赖于机器的字节序。在一个大端的机器上，结果将是11_{hex}；而在小端的机器上，结果则是44_{hex}。这个故事告诉我们，如果你声明了某种精度的数据类型却用别的精度去访问它，因为字节序的问题，这可能会成为灾难的根源。一些体系结构如IBM PowerPC和Sun SPARC是大端的，而Intel x86、MIPS和DEC Alpha则是小端的例子。一般说来，字节序对于程序的性能是没有影响的，但总能找到一些病态的例子使得某一种字节序比另一种具有更好的性能。这些例子通常为字符串操作。

比如说，考虑字符串“RAMACHANDRAN”和“WAMACHANDRAN”在大端机器中的内存布局（如图2-4所示）。假设前一个字符串的起始地址为100。

```
char a[12] = "RAMACHANDRAN";
char b[12] = "WAMACHANDRAN";
```

现在来考虑一下相同的字符串在小端机器中的内存布局，如图2-5所示。仔细观察图2-4和图2-5可以发现，在大端机器中，字符串从左往右排布；而在小端机器中则从右往左排布。为了比较这两个字符串，在两种体系结构中，程序员都可以利用字符串在内存中的布局来获得一些性能上的提高。

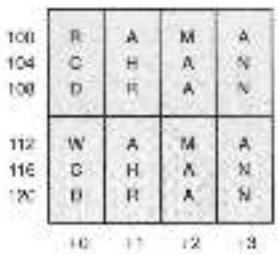


图2-4 大端布局，字的MSB处于字地址上（即字母R在内存地址100上）

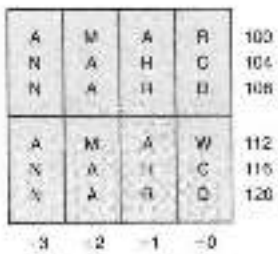


图2-5 小端布局，字的LSB处于字地址上（即字母R在内存地址100上）

正如前面提到的，如果按照声明的那样去操作数据类型，那么字节序对于你的程序是毫无影响的。然而，总有这样的情况，即使一个程序并没有违反上面的规则，字节序依然会影响程序行为。最常见的情况是网络有关的代码，因为它需要在多种不同的机器上工作。如果发送端是小端机器而接收端是大端机器，甚至连网络代码的正确性都会受到影响。正是由于

这个原因，网络代码中使用格式转换例程在网络格式和本机格式之间进行来回转换，以避免这样的陷阱。[⊖]

读者可能会感到奇怪，为什么计算机的制造者不都选择同一种字节序呢？问题在于，对不同的计算机制造者来说，字节序都是神圣的，而且目前也没有有关的标准，因此程序员只能适应多种不同字节序处理器共存的现实。

为了方便讨论，在本章接下来的部分我们都采用小端体系结构。

2.4.5 操作数打包以及字操作数的对齐

现代的计算机系统有大量的内存。因为有大量的内存可以使用，所以对内存的使用没有必要吝啬。然而，这并不完全正确。随着内存容量的增加，软件对内存的胃口也更大了。程序在内存中占据的空间通常被称为内存印迹。如果编译器在编译时非常简单粗暴，可能会尝试将程序在内存中的操作数打包以节约空间。具体来说，如果数据结构中包含了多种不同粒度的变量（即 int、char 等）且体系结构支持多种精度的操作数，那么这是很有意义的。顾名思义，打包的意思是将操作数相邻在内存中时保证没有空间被浪费。然而，在本章中我们将解释为何打包并非总是正确的途径。

首先，我们讨论编译器如何排布内存中的操作数以达到节约空间的目的。考虑下面的数据结构。

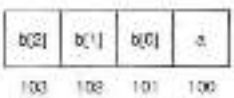
```
struct {
   char   a;
   char   b[3];
}
```

这个数据结构在内存中的一种可能布局以 100 为起始地址，如下图所示



让我们来确定这个数据结构最终需要占据的内存大小。因为每个 char 是 1 字节，所以数据结构的实际大小是 4 字节。但上面的布局却浪费了 50% 用于存储的空间。阴影部分就是浪费的空间。这就是未打包的布局。

有效的编译器将释放掉浪费的空间而将上面的数据结构打包，以 100 为起始地址，如下图所示：



编译器进行的打包与数据类型要求的精度和体系结构支持的可寻址性都是对应的。除了节省空间以外，这样的布局还能减少在处理器寄存器和内存之间来回搬运该数据结构（包含变量 a 和 b）所需的访存次数。因此，打包操作数在空间和时间上都是高效的。

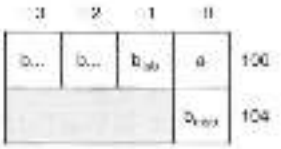
正如我们上面提到的，对于编译器来说，打包并非总是应当采取的策略。

⊖ 如果你接触过 Unix 源码，找一找例程 htonl 和 ntohl，即 host 到 network 以及 network 到 host 的格式转换例程。

考虑下面的数据结构：

```
struct {
   char    a;
   int     b;
}
```

我们来再次确定这个数据结构需要在内存中占据多少空间。一个 char 是 1 字节，一个 int 是 1 个字（即 4 字节）。因此，总共需要 5 字节来存储这个结构。我们来看看其中一种可能的布局，以 100 为起始地址。



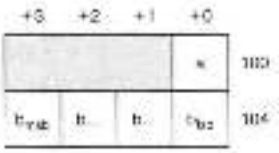
这种布局的问题在于，b 是一个 int，它始于地址 101，结束于 104。为了加载 b，需要从内存中读取两个字（分别从 100 和 104 两个地址读取）。无论由硬件还是软件来实现，这都是非常低效的。体系结构通常要求字操作数从字地址开始，这就是所谓的操作数与操作地址的对齐限制。

如果 address 不在字边界（100、104 等）上的话，那么下面的指令

```
ld   r2, address
```

就是一条非法指令。尽管编译器可以生成代码来加载两个字（在地址 100、104 处）并将它们在处理器中重新组成所需的 int 数据类型，但这在时间上是非常低效的。因此，典型的编译器在布局数据结构时会使得需要字精度的操作数位于字边界地址。

所以，编译器很可能将上面的数据结构按下图的方式来布局，起始地址为 100：



尽管这个布局浪费了 37.5% 的空间，但从访问操作数的时间这个角度来看，变得更加高效了。

你将会看到，计算机科学领域在第 1 章列出的所有抽象层次（从应用程序到体系结构）上都会表现出这种经典的时间—空间的权衡。

34

2.5 高级数据抽象

截至目前，我们已经讨论了高级语言中的简单变量，例如 char、int 和 float。我们将这些变量称为标量。这些变量需要的存储空间都是先验的。编译器可选择将标量变量放在寄存器或者内存中。然而，对于高级语言通常支持的数据抽象，如数组和结构，编译器只能把它们分配在内存中，除此之外别无选择。回想一下，由于可寻址性的问题，处理器中寄存器的数量通常只有十来个。所以，这些数据结构庞大的体积排除了将它们分配在寄存器中的可能。

2.5.1 结构

高级语言中的结构数据类型可以通过基址加偏移量的寻址模式来提供支持。

考虑如下 C 语言结构：

```
struct {
    int  a;
    char c;
    int  d;
    long e;
}
```

如果这个结构的基地址在某个寄存器 rb 中，那么访问结构中的任意字段都可以通过提供一个相对于基址寄存器的偏移量来完成。编译器知道每个数据类型需要多少空间，也知道每个变量在内存中的对齐情况。

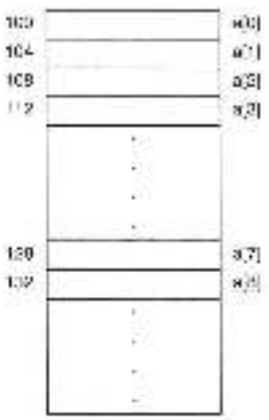
2.5.2 数组

考虑如下的声明：

```
int a[1000];  从 a[0] 到 a[999] 的一个整型数组
```

这里的 a 所指的并非是单个变量，而是变量 a[0]、a[1] 等组成的一个数组。由于这个原因，数组也常被称为向量。这种变量需求的空间在编译的时候可能知道也可能不知道，这取决于高级语言的语义。许多编程语言允许数组在运行时动态地决定大小而不是在编译的时候确定。这意味着，在编译期间编译器不知道数组所需要的存储空间。与之相反的是，标量在编译时是知道所需空间大小的。因此，编译器通常会使用内存来为这些向量变量分配空间。 35

编译器可能会将变量 a 在内存中按下图排布：



考虑下面这条操作数组的语句：

```
a[7] = a[7] + 1;
```

为了编译前面这条语句，假设指令集只允许 ALU 使用寄存器，那么首先我们需要将 a[7] 从内存中取出。显然这是可行的，使用我们已经介绍过的基址加偏移量寻址模式：

```
ld  r1, 28(rb)
```

rb 初始化为 100 时，上面这条指令就完成了将 a[7] 加载到 r1 中的工作。

一般来说，数组常在循环中使用。在这种情况下，可能有个循环计数器（设为 j），它被用来索引数组。考虑下面的指令：

```
a[j] = a[j] + 1;
```

在上面的语句中，相对于基址寄存器的偏移量是不固定的。它由循环当前的索引值得到。尽管还可以生成代码来加载 a[j] ⊖，在能够计算出 a[7] 的有效地址之前还需要额外的指令。所以，一些计算机体系结构提供了一种寻址模式允许有效地址来自两个寄存器内容之和。这被
36 称为**基址加索引**的寻址模式。

每条新指令和每种新寻址模式给实现增加了复杂性，因此需要非常小心地衡量其中的利弊。这通常由花费/性能分析来完成。例如，为了增加基址加索引寻址模式，我们需要问以下几个问题：

1）在程序的执行中，这种寻址模式有多常用？

2）从减少指令条数的角度来说，基址加索引寻址模式相对于基址加偏移量寻址模式有什么优势？

3）从执行时间的角度来说，使用基址加索引寻址模式的加载指令与使用基址加偏移量寻址模式相比需要付出什么代价？

4）为了支持基址加索引寻址模式，需要什么额外的硬件？

对上面四个问题的回答将给我们提供一个定量的标准来判断是否应该将基址加索引寻址模式包含进去。

我们在后面讨论处理器实现和性能影响时还会回头考虑如何评价向处理器中添加新指令和新寻址模式。

2.6 条件语句和循环

在谈论条件语句之前，理解程序执行时的控制流概念是非常重要的。在普通的控制流下，程序是顺序执行的：

```
100     I1
104     I2
108     I3
112     I4
116     I5
120     I6
124     I7
128     I8
132
```

指令 I_1 执行后紧跟着 I_2，然后是 I_3，I_4 等。我们使用一个专用的寄存器，称为**程序计数器**（PC）。概念上，我们可以想象 PC 指向正在执行的指令⊜。我们知道，程序并非总是沿着顺序
37 的控制流路径执行。

⊖ 简单地说，索引值需要乘以 4 然后加到 a 上来获得有效地址。只用基址加偏移量寻址模式来实现 a[j] 的代码留作给读者的练习。

⊜ 在第 3 章我们将看到，为实现的功能考虑，PC 包含着紧接着现在正在执行的指令的那条指令的内存地址。然而需要注意的是，这只是本书读中的一个简单的设计选择，并非从指令集设计的必然。

2.6.2 if-then-else 语句

考虑下面的语句：

```
    if (i == k) go to L1;
    a = b + c;
L1: a = a + 1;
```

我们来研究编译上面的语句时要什么。if 语句包含两部分：

1）对断言“i == k”求值：这可以通过我们已经看过的表达式求值用的指令来完成。

2）如果断言为真，那么它将控制流从下一条语句改变到目标 L1。到目前为止，我们的指令集还无法完成改变控制流的任务。

因此，我们有必要添加这样的新指令，它除了可以对算术和逻辑运算进行求值，还能改变控制流。我们引入这样一条新指令：

```
beq r1, r2, L1;
```

这条指令的语义如下：

1）比较 r1 和 r2。

2）如果它们相等，那么下一条被执行的指令在地址 L1 处。

3）如果它们不相等，那么下一条被执行的指令就是紧跟着这条 beq 指令的指令。

BEQ 是用于改变控制流的条件分支指令的一个例子。我们在指令中需要指明分支的目标地址。在描述算术/逻辑运算指令时，我们讨论了可寻址性（见 2.4.1 节）并认为将操作数放在寄存器而非指令中是一个好方法，因为这样减少了指令用来指明操作数的位数。这很自然地引出了分支指令中目标地址是否需要放在寄存器中而不是作为指令一部分这个问题。也就是说，地址 L1 应该放在寄存器中吗？一条分支指令通常将控制流从当前指令（即分支指令）带到另一条相距不是很远的指令。我们知道 PC 指向当前执行的指令，那么分支的目标可以用指令中表示的相对于当前分支指令位置的地址偏移量来表示。基于从当前指令到分支目标指令的距离不是太远，所以地址偏移量只需要分支指令中的少数几位来表示。换句话说，用分支指令的一部分作为内存地址（确切地说，地址偏移量）来指定分支目标是合适的。 38

因此，分支指令的格式如下：

```
beq r1, r2, offset
```

这条指令的效果有：

1. 比较 r1 和 r2。

2. 如果它们相等，那么下一条被执行的指令的地址为 PC+$offset_{signed}$ ⊖。

3. 如果它们不等，那么下一条被执行的指令就是直接跟在 beq 指令后面的指令。

我们不仅上新增了一种计算有效地址的寻址模式，这被称为程序计数器相对寻址。

指令集体系结构对于条件分支语句可能有不同的选择，如加 BNE（不等时跳转）、BZ（为零时跳转）以及 BN（为负时跳转）。

经常的情况是，在条件语句中会有一个 else 分句。

```
    if (i == k) {
        a = b + c;
    }
    else {
```

⊖ PC 表示当前 beq 指令的地址，$offset_{signed}$ 表示指令中带有正负符号的偏移量按照实现细节被解释的值。我们在下一章会谈到它。

```
        a = b - c;
    }
L1: ...
    ...
```

事实证明我们编译上面的代码不需要在体系结构中新增任何指令。条件分支指令非常有效地处理了上面的 if 语句有关的计算和分支。然而，当 if 分句的部分被执行完之后，控制流需要无条件地转移到 L1（紧接着 else 分句的语句块的开头）。

为了能够这么做，我们引入“无条件跳转”语句

```
j   rtarget
```

这里的 r_{target} 包含了无条件跳转的目标地址。

对无条件跳转指令的需求需要详细的阐述，毕竟我们已经有了条件分支指令了。无论如何，
39 我们可以通过条件分支指令 beq 来实现无条件分支。当 beq 指令的两个操作数使用同一个寄存器时（即 beq r1，r1，offset），结果就是无条件跳转。然而，这里有一个迷惑人的地方。条件分支指令的范围受到偏移量大小的制约。比如，一个 8 位的偏移量（假设偏移量使用补码表示，这样正负偏移量都可以表示），那么分支的范围被限制在 PC−128 − PC+127。这正是引入新的无条件跳转指令的原因，在这样的指令中，使用一个寄存器来表示跳转的目的地址。

读者应该可以明显感觉到，使用条件和无条件分支，我们可以编译任意多层级的嵌套 if-then-else 语句。

2.6.2 switch 语句

许多高级语言提供了特殊的条件语句，以 C 语言的 switch 语句为代表。

```
switch (k) {
   case 0:
   case 1:
   case 2:
   case 3:
   default:
}
```

如果 case 的数量有限或者比较稀疏，最好的方法就是将这个语句编译为多个 if-then-else 语句结构。另一方面，如果有很多连续的、不稀疏的 case，那么将它们编译为嵌套的 if-then-else 语句就会生成低效的代码。另一种选择是，使用一个跳转表记录所有 case 代码段的起始地址，这样就能产生高效的代码（见图 2-6）。



40 图 2-6 使用跳转表实现 switch 语句。表中的每一项指向与 case 值对应的第一条指令

原则上，完成这样的实现不需要新的指令。虽然不需要在指令集体系结构中添加新的指令以支持switch语句，但让无条件跳转语句支持一级的间接寻址会有很大好处。

这就是

```
J   *(rtarget)
```

这里的rtarget包含了（无条件）跳转的目标地址。

另外，一些体系结构提供了专用的指令用于边界检查。例如，MIPS体系结构提供了一个若小于则置位（set-on-less-than）指令。

```
SLT r1, r2, r3
```

它的效果是

```
if r2 < r3 then set r1 to 1
else set r1 to 0
```

这条指令对于实现switch语句时的边界检查是非常有用的。

2.6.3 循环语句

高级语言提供了不同形式的循环结构。考虑如下代码片段

```
        i = 0;
loop:   h = h + a[i];
        i = i + 1;
        if (i != 100) go to loop;
```

假设寄存器r1用于保存变量i，而寄存器r2中包含值100，我们可以按下面的方式编译前面的循环结构：

```
loop:   ...
        ...
        ...
        bne r1, r2, loop
```

因此，不需要新指令或寻址模式去支持上述循环结构。读者应该能够很容易地看到，其他一切形式的循环（如for、while、repeat等）都能用已经介绍过的条件和无条件分支指令类似地编译。

41

2.7 检查点

到目前为止，我们已经看到了下列高级语言结构：

1）表达式和赋值语句；

2）高级数据结构；

3）条件语句，包括循环。

为了有效编译所讨论的结构，我们还为指令集体系结构开发了以下功能：

1）使用寄存器的算术/逻辑指令；

2）在内存和寄存器间传输数据的指令（加载和存储指令）；

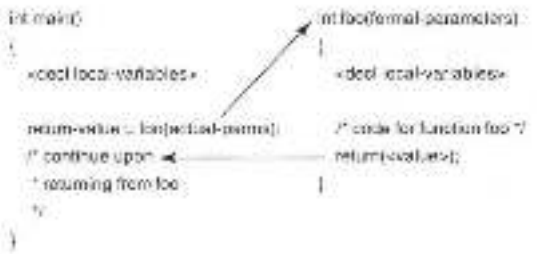
3）条件和无条件分支指令；

4）寻址模式：寄存器寻址、基址加偏移量寻址、基址加索引寻址、程序计数器相对寻址。

2.8 编译函数调用

编译高级语言中的过程调用或函数调用有一些需要特别注意的事情。

首先，我们来回顾程序员脑海中过程调用的模型。程序在 main 函数中调用了函数 foo。程序的控制流转移到了该函数的入口处。返回 foo 时，控制流返回到 main 函数中紧接着调用 foo 函数的语句。下面是这个模型的样子：



首先我们定义几个术语：调用者（caller）指的是做出过程调用的实体（在这个例子中是 main 函数）；被调用者（callee）是被调用的实体（例子中的 foo）。

让我们一一列出编译一个过程调用的步骤：

42 1）保证调用者的状态（即该调用者使用的寄存器）被保护好以便从过程调用中返回时得以恢复。

2）将实际参数传递给被调用者。

3）记住返回地址。

4）将控制权转交给被调用者。

5）为被调用者的局部变量分配空间。

6）从被调用者接收返回值并传给调用者。

7）返回调用点。

我们现有的指令集体系结构的功能能够满足前面这些要求吗？为了回答这个问题，我们讨论一下列表中的每一项。我们首先在 2.8.1 节中探讨保存调用者状态的结果。然后在 2.8.2 节考虑余下的条项。

2.8.1 调用者的状态

首先定义什么是调用者的状态。执行被调用者代码所需的资源有内存（被调用者的代码和数据）和处理器中的寄存器（所有的算术/逻辑指令会都用到它们）。编译器会确保调用者和被调用者使用不同的内存（例外情况是高级语言的语义要求的共享内存）。所以，处理器的寄存器中的内容才是我们需要担心的“状态”，因为当调用者调用被调用者时，它自身的某些结果正存储在寄存器中。因为我们不知道被调用者将使用哪些寄存器，谨慎的做法是在过程调用前先将它们保存起来，然后在返回时再恢复。

现在我们需要一个地方来保存这些寄存器。我们先尝试硬件的解决方案。我们引入一个影子寄存器组，我们将在调用前把寄存器保存到里面，从过程中返回时再将这个影子寄存器组中的值恢复到处理器的寄存器中（见图 2-7）。

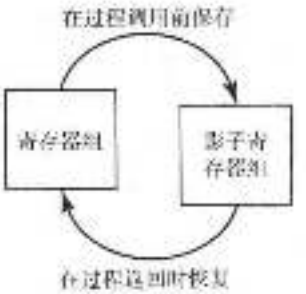


图 2-7 过程调用/返回时状态的保存/恢复。在调用过程时，将寄存器保存在影子寄存器组中，在返回时恢复

我们知道，在模块化的代码中，过程的调用是很频繁的，因此快速地保存/恢复状态非常重要。鉴于影子寄存器组位于处理器内部，而所有的保存/恢复操作都

发生在硬件层面，所以影子寄存器组看起来是个好主意。

这个想法的最大问题在于，它假设被调用的过程不会再进一步调用其他过程了。根据我们写高级语言程序的经验，这个假设完全站不住脚。实际上，我们需要与嵌套过程调用的层数一样多的影子寄存器组（见图 2-8）。

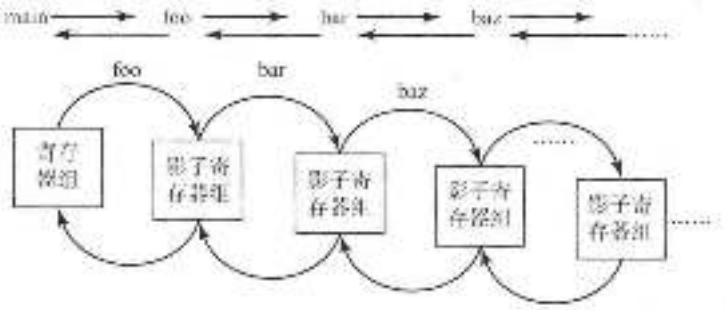


图 2-8 嵌套的过程调用中状态的保存和恢复。将影子寄存器组的想法扩展至一个嵌套的过程调用序列

我们来讨论硬件实现这些影子寄存器组的意义。嵌套过程调用（即图 2-8 中的链）的层级是程序的动态属性。硬件解决方案必须是有限的，与影子寄存器组的数量相关。并且由于成本和复杂性的原因，这个数量不能任意大。尽管如此，还是有的体系结构，如 SPARC，为此实现了一种称为寄存器窗口的硬件机制。SPARC 提供了 128 个硬件寄存器，但任意时刻都只有 32 个是可见的。那些不可见的寄存器就成了影子寄存器组，像图 2-8 所示的那样工作。

图 2-8 还向我们提示了一种可能的软件实现方案：使用你们可能已经在数据结构中学习过的栈。我们在调用点将状态保存入栈中，在返回的时候恢复。因为栈具有后进先出（LIFO）的特性，它正好满足了嵌套过程调用的需求。

栈可以通过软件的抽象在内存中实现，所以就不受嵌套层次的限制了。

编译器需要维护一个指向栈的指针以保存和恢复状态。这并不需要在体系结构上添加任何新东西。编译器会选择处理器中的一个寄存器作为专用的栈指针。注意，这并不是体系结构必需的要求，而仅仅是为编译器提供方便。另外，每个编译器都有选择不同寄存器作为栈指针的自由。这指的是编译器将不会使用这个寄存器来保存程序变量，因为栈指针已经被保留作为编译器内部的专用功能了。

43
|
44

刚才我们通过选定一个寄存器作为栈指针来说明了软件使用寄存器的实例。

硬件解决方案的一个好处是，它是用硬件实现的，所以很快。现在我们用软件方式来实现栈，代价是每次过程调用和返回时需要在内存和处理器寄存器之间来回搬运数据。想想看能不能既保持软件方式的弹性，又具有硬件方式的性能优势呢？软件方式无法达到硬件方式的高速，但确实可以减少那些浪费的部分。

先统计一下在每次调用和返回的时候是否真的需要保存或恢复所有的寄存器。之前的途径是调用者（即代表调用者的编译器）负责保存和恢复。如果被调用者根本没有使用任何寄存器，那么整个保存和恢复寄存器的工作都白做了。因此，我们可以把这个工作交给被调用者，让它在运行时保存将要使用的寄存器而在过程返回时恢复它们。同样地，如果调用者在过程返回后根本不使用这些寄存器里的值，那么这个工作也白做了。

为了走出这个困境，让我们再扩展一下软件惯例的想法。这里打个比方来帮助大家理解。这是一个关于两个懒惰的室友的故事。他们都想做尽可能少的家务，但是，他们每天都得吃饭，所以最终他们达成了一个协议。他们有一个公用的盘子，每个人有各自的几个碟子。他们决定遵守的规则如下：

- 自己的碟子永远都不需要清洗。
- 如果使用了别人的碟子，那么每次用完后必须洗干净。
- 盘子并不保证是干净的，所以每个人在用盘子之前，如果需要的话，最好是自己洗一遍。

有了这个惯例，每个人的工作都只剩一点了——如果他们不使用盘子和别人的碟子，那么根本就不需要工作！

过程调用的软件惯例采取的方式和这个懒人比喻差不多。当然，过程调用与这个比喻不同的是，它是不对称的（存在着调用者与被调用者的秩序）。调用者获得寄存器中属于他自己的一个子集（称为s寄存器组）。调用者可以任意使用这些寄存器，不用担心被调用者会冲掉里面的数据。被调用者如果需要使用s寄存器组，则需要保存和恢复它们。与盘子类似，还有一部分寄存器（称为t寄存器组）是调用者和被调用者所共有的。它们使用这些寄存器都不需要保存或恢复。现在，正如懒人比喻里的那样，如果调用者在过程返回后不使用t寄存器组中的值，那么在过程调用时就不需要干活。同样地，如果被调用者不使用s寄存器组，那么它也不需要考虑保存和恢复的事情。

寄存器的保存和恢复将在栈中实现。当我们讨论完过程调用和返回的其他工作之后，还
[45] 会接着补全关于过程调用和返回的软件惯例。

2.8.2 过程调用剩余的工作

1）参数传递　一种权宜之计是，使用处理器中的寄存器来传递参数。再一次地，编译器将建立一套软件惯例，将某些寄存器保留供传参专用。

当然，过程所需的参数个数可能会超过这些寄存器数量的限制。这种情况下，编译器会使用栈来传递多出的这些参数。软件惯例保证被调用者能够于栈指针知道在栈中的何处取得参数。

2）记住返回地址　我们在之前讲分支指令时提到过处理器中的程序计数器（PC）。迄今为止我们介绍过的高级语言结构都不需要记住自己位于程序何处。所以，现在需要一条新的指令将PC值保存在一个众所周知的地方，以便能在过程返回中使用它。

我们引入一条新的指令：

```
JAL r_target, r_link
```

这条指令的语义如下：

- 将返回地址保存在r_{link}中（可以是任意一个处理器的寄存器）。
- 将PC置为r_{target}中的值（即被调用者的起始地址）。

我们回到软件惯例的问题上来。编译器会指定一个寄存器作为r_{target}来保存目标例程的地址，而指定另一个寄存器作为r_{link}来保存返回地址。也就是说，这些寄存器将不能用来存放普通的程序变量。

因此，在调用点，过程调用被编译为

```
JAL r_target, r_link; /* r_target containing the address
```

从过程中返回的方式很直接，因为我们已经有无条件跳转指令了，

```
J r_link
```

完成了从过程调用中返回的任务[1]。

3）**将控制权移交给被调用者** 第3步是通过JAL指令将控制权移交给被调用者。

4）**被调用者局部变量的空间** 使用栈可以很方便地为被调用者所需的局部变量分配空间。软件惯例保证了被调用者借助于栈指针可以方便地找到局部变量的所在[2]。 46

5）**返回值** 一个方法是编译器保留某些寄存器用于返回值。与参数传递时相同，如果返回值超出了这些寄存器能够保存的能力范围，剩余的返回值将通过栈进行传递。软件惯例保证了调用者借助于栈指针能够找到返回值。

6）**返回到调用点** 正如前面提到的，简单的一条跳转到 r_{link} 的指令就能将控制权交回到调用点。

2.8.3 软件惯例

为了让讨论更具体些，我们这里介绍一套处理器寄存器及其使用的软件惯例：

- 寄存器 s0 – s2 是调用者的寄存器。
- 寄存器 t0 – t2 是临时寄存器。
- 寄存器 a0 – a2 是传参寄存器。
- 寄存器 v0 用于保存返回值。
- 寄存器 ra 用于保存返回地址。
- 寄存器 at 用于保存目标地址。
- 寄存器 sp 用作栈指针。

在展示如何使用这些软件惯例来编译过程调用之前，我们需要强调一些栈的细节。许多编译器采用的惯例是，栈从高地址往低地址增长。基本的栈操作如下：

- 压栈（push）减小栈指针并将值保存于栈指针指向的内存中。
- 弹出（pop）取出栈指针指向的值并增大栈指针。

图2-9 – 图2-20展示了编译器产生的运行时生成栈帧的代码。在所有的图中，低地址是栈的顶部而高地址是栈的底部，与栈从高地址往低地址增长一致。

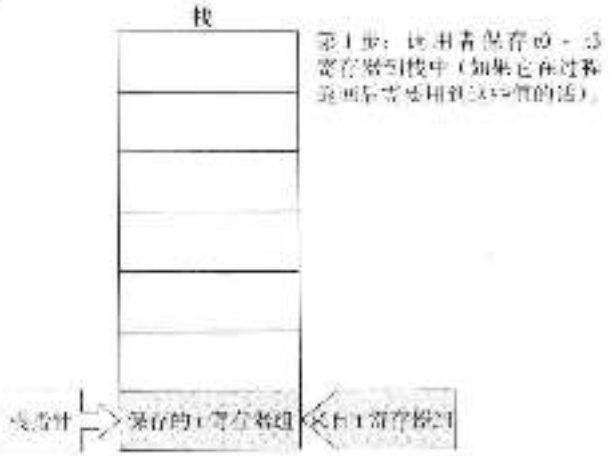
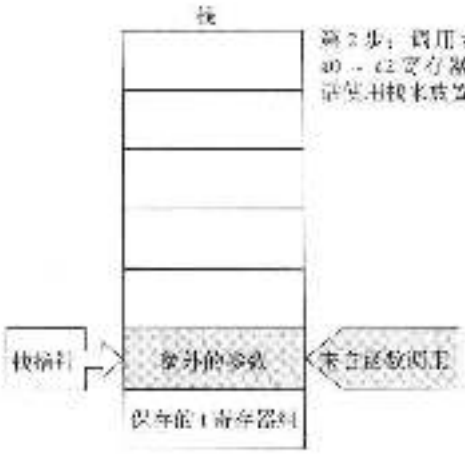


图2-9 过程调用与返回第1步

47

[1] 实际上，因为我们还有JAL指令，我们可以用这条指令来实现无条件跳转。JAL r_{link}，r_{link} 将无条件跳转到 r_{link} 后指向的位置，此处的 r_{link} 表示被忽略。

[2] 练习题18中有这个描述的一个变形。



48

图 2-10 过程调用与返回第2步

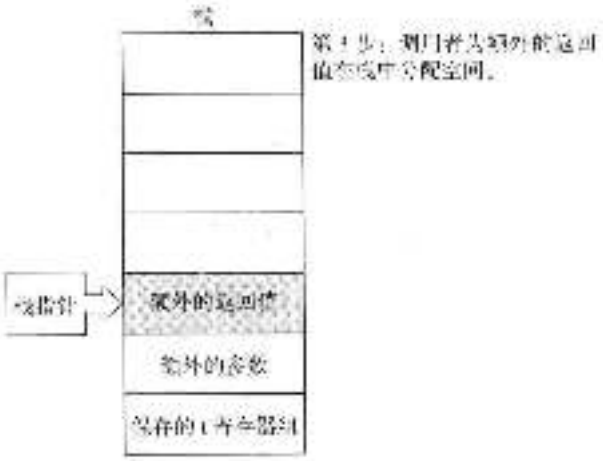
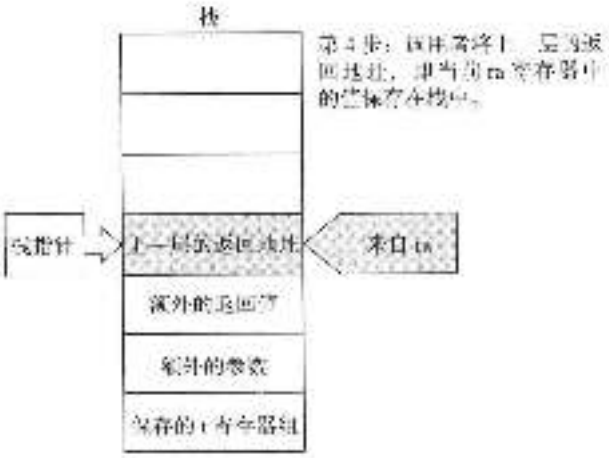


图 2-11 过程调用与返回第3步



49

图 2-12 过程调用与返回第4步

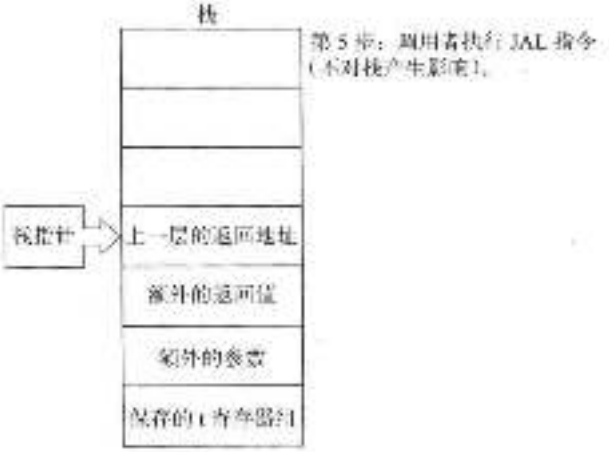


图 2-13 过程调用与返回第 5 步

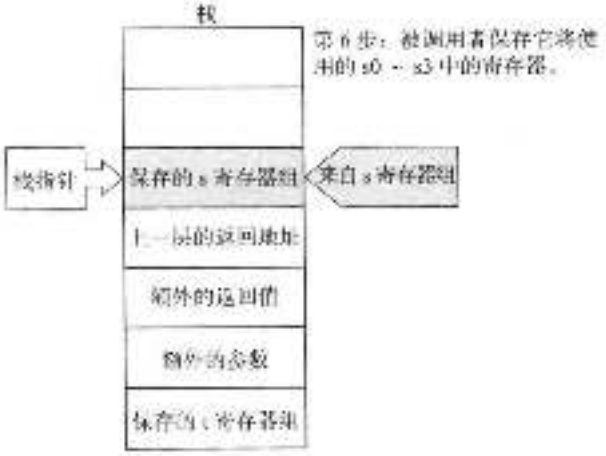


图 2-14 过程调用与返回第 6 步

50

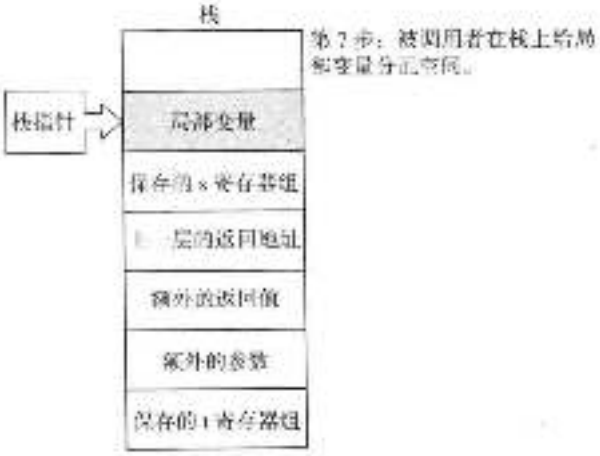


图 2-15 过程调用与返回第 7 步

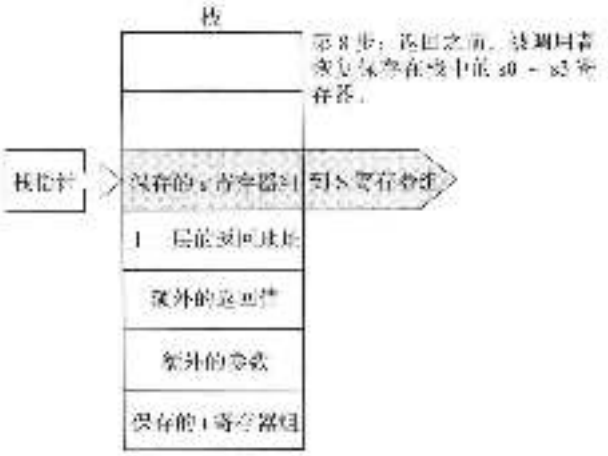


图 2-16 过程调用与返回第 8 步

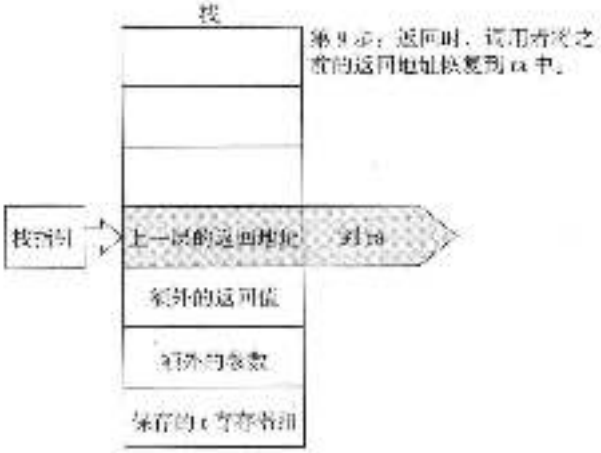


图 2-17 过程调用与返回第 9 步

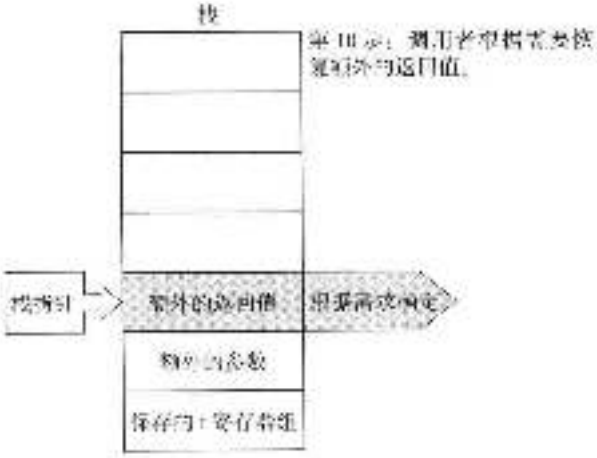


图 2-18 过程调用与返回第 10 步

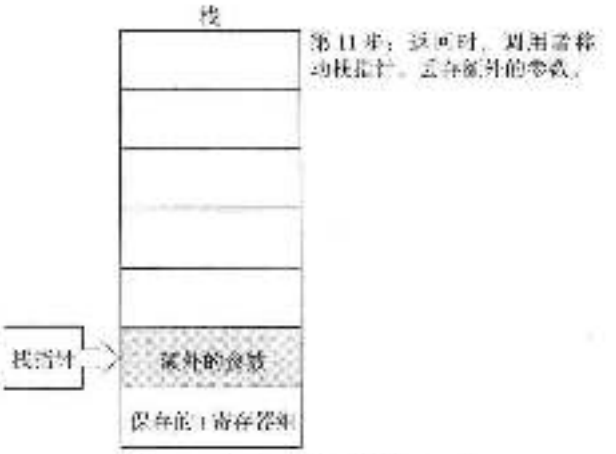


图 2-19 过程调用与返回第 11 步

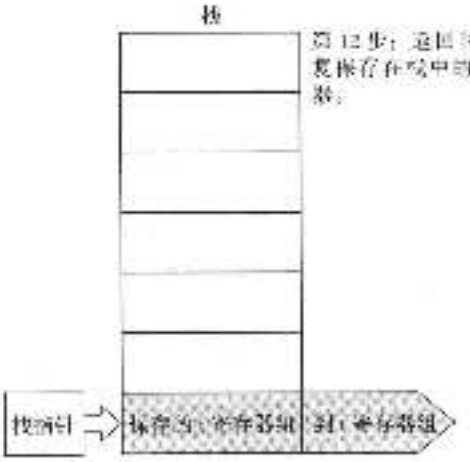


图 2-20 过程调用与返回第 12 步

53

2.8.4 活动记录

栈中与当前执行的过程有关的一部分区域被称为该过程的活动记录。活动记录是调用者与被调用者之间进行通信的区域。图 2-9 ~ 图 2-19 显示了调用者和被调用者之间如何建立活动记录，被调用者如何使用活动记录，以及返回时（调用者和被调用者）如何拆除活动记录。依赖于过程调用的嵌套，栈上可能会有多个活动记录存在。但是在任意时刻，都有且仅有一个与当前正在执行的过程有关的活动记录是在活动的。

考虑下面的序列：

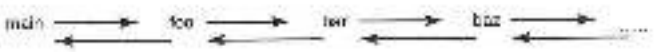
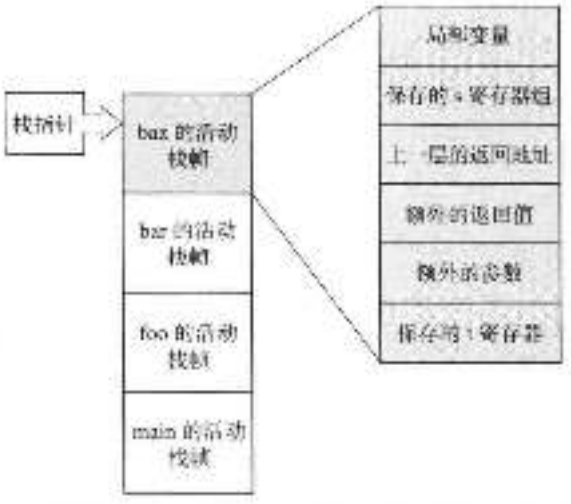


图 2-21 展示了前面这个调用序列的活动记录和栈的情况。



54 图 2-21 调用序列的活动记录。注意，只有最顶端的活动记录才是对当前执行的过程有意义的

2.8.5 递归

对程序员来说，最有力的工具莫过于递归了。我们不需要在指令集体系结构上增加任何东西就能支持递归。栈机制保证了过程的每个实例（无论它们是属于相同还是不同的过程）都拥有一个新的活动记录。当然，能被递归调用的过程必须写成支持递归的代码，这是程序员考虑的范围。指令集为了不需要做任何改变就能支持递归。

2.8.6 帧指针

在程序执行的过程中显而易见的是，必须能够定位放在栈上的所有东西的位置。在程序执行之前，它们的绝对位置是不知道的。很明显它们能够通过用栈指针的偏移量表示出来，但这条途径有一个问题。某些编译器会生成在函数执行过程中（即栈帧已经建好之后）修改栈指针的代码。例如，一些语言支持在栈上动态分配。尽管可以跟踪栈指针的移动，但是这带来的需要额外维护的信息以及在运行时间上的代价决定了这是一个馊主意。常用的解决方案是指定一个通用寄存器作为帧指针，包含在当前函数的活动记录中的一个已知的位置。在函数执行过程中，这个地址是不会改变的。用一个例子来理清这个问题和解决方案。考虑下面的过程：

```
int foo(formal-parameters)
{
   int a, b;

   /* 一些代码 */
   if (a > b) {                                          (1)
      int c = 1;                                         (2)
      a = a + b + c;                                     (3)
   }

   printf("%d\n", a);                                    (4)

   /* foo 的其余代码 */

   /*
```

```
    * 从foo返回
    */
    return(0);
}
```

我们假设栈指针（用 Ssp 表示）在调用 foo 的第 6 步（见图 2-14）之后的值是 100。在第 7 步中，被调用者为局部变量（a 和 b）分配了空间。按照栈从高地址向低地址增长的惯例，a 被分配在地址 96 而 b 被分配在地址 92。现在 Ssp 的值是 92 了。图 2-22 展示了这个情况。 55

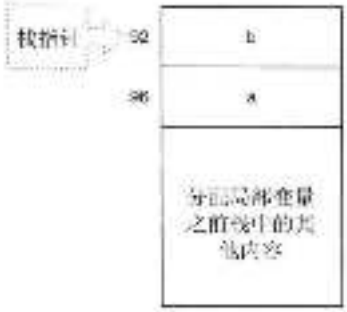


图 2-22 过程中的局部变量分配之后栈的状态。注意现在栈指针的值是 92。

foo 过程开始执行。“if”语句生成的代码需要将 a 和 b 装入寄存器中。这对编译器来说非常直截了当。下面的两条指令

```
ld r1, 4($sp); /* 将a存入r1，a的地址为 $sp+offset = 92 + 4 = 96*/
ld r2, 0($sp); /* 将b存入r2，b的地址为 $sp+offset = 92 + 0 = 92*/
```

将 a 和 b 分别装入寄存器 r1 和 r2 中。

注意，如果上一步计算正确的话，下一步会发生什么呢？现在程序新分配了一个变量 c，它也是在栈中分配的。然而，这个局部变量的分配并不在 foo 的执行前（见图 2-15，第 7 步）进行。这是个有条件的分配，只在 if 语句为真的时候进行。变量 c 的地址是 88，现在 Ssp 的值是 88。图 2-23 展示了现在的情况。



图 2-23 变量 c 分配之后栈的状态。此时对应右过程中的语句（2）。注意现在栈指针的值是 92

在 if 语句块中，我们可能需要加载和存储变量 a 和 b（见代码块中的语句 3）。生成变量 a 和 b 的正确地址是个诡异的问题。原来变量 a 对于 $sp 的偏移量是 4，现在，对于当前的 $sp 来说，偏移量则是 8。因此，在语句（3）中为了加载变量 a 和 b，编译器需要生成下

面的代码：

```
ld r1, 4($sp); /* 将a载入r1，a的地址为$sp-offset = 88 + 8 = 96*/
ld r2, 0($sp); /* 将b载入r2，b的地址为$sp-offset = 88 + 4 = 92*/
```

当if语句执行完毕之后，c被从栈中释放掉，$sp又变成了92。现在a对于当前$sp的偏移量又是4了。这与图2-22所表示的是一样的。

读者可以看出，栈能够扩大和缩小这个事实使得编译器更难写了。依赖于当前的栈指针，各个局部变量的偏移量都在改变。

这正是选择寄存器作为帧指针的原因。帧指针包含了栈中与被执行过程相关的活动记录的第一个地址，并且在这个过程的执行过程中都不会改变。当然，如果该过程又调用了别的过程，则它的帧指针也需要进行保存和恢复，这由被调用者来完成。因此，被调用者所做的第一件事就是将帧指针保存到栈中，并将当前的栈指针值复制给帧指针。如图2-24所示。

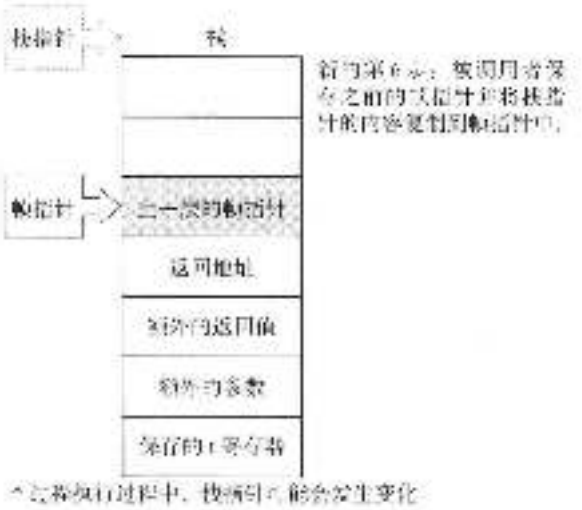


图2-24 帧指针

这个帧指针是栈上一个固定的套子（对于一个过程来说），它指向当前执行的过程的活动记录（AR）的首地址。

2.9 指令集体系结构选择

在本节中，我们总结关于指令集设计的体系结构的不同选择。这种选择存在于指令集中的算术/逻辑运算、寻址模式、体系结构类型，以及实际内存的布局（即指令格式）中。有时候，做出这些选择是由于当前的技术趋势和硬件的可行性考虑，有时候则是为了对高级语言结构提供精简而高效的支持。

2.9.1 额外的指令

有的体系结构提供了额外的指令，以提升编译出的代码的空间和时间的有效性。

- 例如，在MIPS体系结构中，加载和存储指令都是对于一整个32位的数进行操作的。然而，一旦指令被装到了寄存器中，特殊的指令就可以用来扩展其中某个特定的字节到另一个寄存器中；类似地，也可以在字中插入某个字节。

- DEC 的 Alpha 指令集包含了加载和存储不同数度操作数的指令：字节、半字、字和四字。
- 一些体系结构包含预定义的立即数（如 0、1 及其他小整数），可以直接在指令中使用。
- DEC VAX 体系结构可以用单条指令来存储/加载寄存器文件中的所有寄存器到内存或者从内存中取出它们。也许有人会认为，这样的指令有利于在过程调用的时候进行寄存器的保存和恢复。读者应该根据本章前面的内容，考虑一下这种指令在过程调用中的实用性。

2.9.2 额外的寻址模式

在我们已经讨论过的寻址模式之外，有的体系机构还提供了更花哨的寻址模式。

- 许多体系机构提供了间接寻址模式：

  ld @(ra)

在这条指令中，寄存器 ra 的内容是实际内存操作数的地址的地址。

- 伪直接寻址：

  MIPS 提供了这样一种类型的寻址模式，它使用 PC 的高 6 位以及它本身的低 26 位构成一个 32 位的有效地址：



早期的体系结构如 IBM 360、PDP-11 和 VAX11 支持的寻址模式远比我们在本章中讨论的多。大部分现代体系结构在访问的内存的寻址模式方面走的是最少化路线。这主要是因为，这么多年来，实际被编译器使用的复杂寻址模式非常少。甚至基址加偏移量在 MIPS 体系结构中也是没有的，尽管 IBM PowerPC 和 Inter Pentium 都支持这种模式。

2.9.3 体系结构类型

历史上曾经有过好几种不同的体系结构类型：

- **面向栈的体系结构** Burroughs Computers 公司引入了面向栈的体系结构，在这种体系结构中，所有的操作数都是在栈上的。所有的指令都操作位于栈上的操作数。
- **面向内存的体系结构** IBM 360 系列机着眼于面向内存的体系结构，大部分（如果不是全部的话）指令都是操作内存中的操作数。
- **面向寄存器的体系结构** 正如在本章讨论的，这种体系结构中的大部分指令处理的是寄存器中的操作数。随着编译技术日趋成熟，以及处理器中寄存器的有效使用，这种风格的体系结构最终保留了下来。DEC Alpha 和 MIPS 都是这种风格的体系结构。
- **混合类型** 通常来说，针对特定的应用，可以选择其中某一种类型的体系结构。因此，很自然的，这些类型的混合体非常受欢迎。IBM PowerPC 和 Intel x86 系列指令集都是面向内存和面向寄存器的混合类型。

2.9.4 指令格式

根据所有指令结构的不同，它们被分为以下几类：

1. 零操作数指令

例子包括：

58

- HALT（停止处理器）
- NOP（什么都不做）

另外，如果体系结构是面向栈的，对于大部分指令（除了压栈和出栈的值是显式的），它们的操作数都是隐含的。在这种体系结构中指令看起来是这样的：

[59] - ADD（弹出栈顶的两个元素，将它们相加，并将结果压入栈中）
- PUSH<operand>（将操作数压入栈中）
- POP<operand>（将栈顶元素弹出作为操作数）

2. 单操作数指令

这种类型的指令通常与高级语言中的一元运算符有关：

- INC/DEC <operand>（将指定操作数增加或减少某个常数值）
- NEG<operand>（取操作数的补码）
- NOT<operand>（取操作数的反码）

另外，无条件转移指令通常也只有一个操作数：

- J <target>（PC ← target）

此外，一些老式机器（如 DEC 的 PDP-8）使用一个隐含的操作数（称为**累加器**，ACC）和一个显式操作数。这种体系结构中的指令看起来是这样的：

- ADD<operand>（ACC ← ACC+operand）
- STORE<operand>（operand ← ACC）
- LOAD<operand>（ACC ← operand）

3. 双操作数指令

这类指令同样映射为高级语言中的二元运算符。其基本思想是在二元运算中，一个操作数既是源操作数也是目标操作数。

- ADD R1, R2（R1 ← R1+R2）

移动数据的指令也属于这一类：

- MOV R1, R2（R1 ← R2）

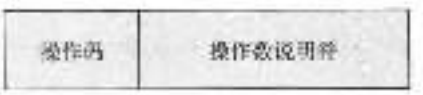
4. 三操作数指令

这是最常见的类型，在这一章中我们都能看到它的例子，比如：

- ADD R_{dst}, R_{src1}, R_{src2}（$R_{dst} \leftarrow R_{src1} + R_{src2}$）
- LOAD R, Rb, offset（R ← MEM[Rb+offset]）

指令格式指的是指令如何在内存中布局。一个体系结构会包含上述各种风格的指令，由许多要求不同数量操作数的指令组成。

典型的指令具有下面的格式：



[60] 在设计指令格式的问题上，我们要考虑实际的实现。因为指令格式的选择对于设计的空间和时间有效性来说是个关键点。与之相关的选择是指令中各字段的实际编码，即在某个给定字段上用怎样的位串表达怎样的语义。例如，表示 ADD 的位串，等等。

广义上讲，所有指令格式分为两类：

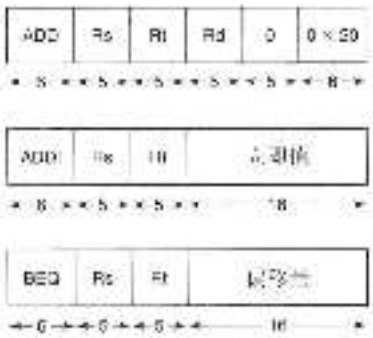
- **所有指令等长** 在这种格式中，所有的指令都有相同的长度（即一个内存字长度）。这

是是指令的某一位段是指令的不同会有不同的含义：

- 优势：
  - 指令长度固定简化了实现。
  - 只要拿到了指令，马上就可以对它的各个字段进行解释，因为所有的指令都是定长的，且长度相同。
- 劣势：
  - 因为各指令需要的空间其实是不一样的（例如单操作数指令和多操作数指令相比），这可能会造成空间的浪费。
  - 我们需要一些连接逻辑（例如译码器和多路选择器）来将指令的各字段分配到数据通路的各元素中去。
  - 指令集的设计者受限于所有指令长度固定（通常是一个字）的限制。这种限制体现在指令中立即值操作数的大小以及寻址时可指定的偏移量的范围。

MIPS 就是使用定长指令的一个例子。

这是 MIPS 的一些指令的例子：



在上面的 ADD 指令中，跟在 Rd 后的 5 位字段是没有用的，但因为定长的需求只能保留下来。

61

- 指令长度可变　在这种格式中指令是变长的，即一条指令可能占多个字。
  - 优势：
    - 不会有空间浪费，因为每条指令都只占据了它所需的空间。
    - 指令集设计者不再受限于有限的大小（例如，立即值的大小）。
    - 有机会根据编译器对指令的使用情况，为操作码、寻址模式、操作数选择不同的大小和编码。
  - 劣势：
    - 这样的格式使实现更复杂了，因为指令的长度只有在解释了操作码后才能确定。这会导致指令的操作码和操作数需要顺序解释。

DEC VAX 11 系列和 Intel x86 系列都是变长指令体系结构的例子。在 VAX 11 中，指令的长度在 1 ~ 53 字节变化。

需要强调的是，我们这里的目的并非暗示说本节介绍的五个体系结构类型和指令格式在今天都是可行的。我们的目的是让读者了解在指令集设计上有过许多曾经尝试过的选择。例如，过去曾经有过面向累加器的体系结构（使用零操作数格式）和基于累加器的机器（使用单操作数指令格式）。然而，这些体系结构已经不再是通用处理器的主流。

2.10 LC-2200 指令集

我们定义 LC-2200 作为这个简单体系结构的一个具体例子。这是一个面向寄存器的、小端的、使用定长指令格式的体系结构。有 16 个通用寄存器以及一个单独的程序计数器（PC）。所有的地址都是字地址。介绍这个指令集的目的有 3 个：

- LC-2200 作为一个简单的指令集的具体实例能够满足任何高级语言的需求。
- 它是我们第 3 章和第 5 章讨论实现细节的具体体系结构。
- 更重要的是，LC-2200 作为一个简单的、没有阻碍的工具，用来让我们在讨论中给它添加一些其他的特性，比如后面章节中的中断、虚拟内存和同步。这种增加功能对于一个学习工具来说是非常有吸引力的，因为它能引导读者走过根据某种需求给处理器增加某些特性的整个过程。

[62]

2.10.1 指令格式

LC-2200 支持 4 种指令格式。R 型指令包含 add 和 nand。I 型指令包含 addi,lw,sw 和 beq。J 型指令包括 jalr。O 型指令包括 halt。因此，LC-2200 共有 8 条指令。表 2-1 总结了这些指令的语义。

R 型指令（add, nand）：

28 ~ 31 位：操作码

24 ~ 27 位：reg X

20 ~ 23 位：reg Y

4 ~ 19 位：未使用（应为全 0）

0 ~ 3 位：reg Z



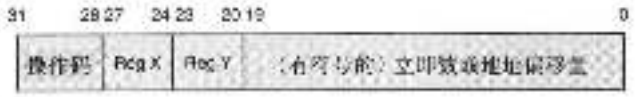
I 型指令（addi, lw, sw, beq）：

28 ~ 31 位：操作码

24 ~ 27 位：reg X

20 ~ 23 位：reg Y

0 ~ 19 位：立即值或地址偏移量（20 位的用补码表示的数，范围从 –524 288 ~ 524 287）



J 型指令（jalr）[⊖]：

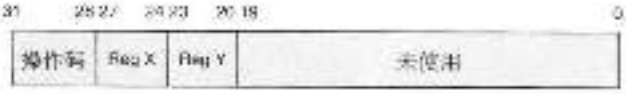
28 ~ 31 位：操作码

24 ~ 27 位：reg X（跳转目标）

20 ~ 23 位：reg Y（链接寄存器）

0 ~ 19 位：未使用（应为全 0）

⊖ LC-2200 并没有单独的无条件跳转指令，然而我们可以通过 JALR R_{link}, $R_{don't\ care}$ 来实现无条件跳转，这里的 R_{link} 包含目的地址而 $R_{don't\ care}$ 是一个你不关心其当前值的寄存器。



63

表 2-1 LC-2200 指令集

例子	格式	操作码	用寄存器传输语言表达的动作
add add $v0,$a0,$a1	R	0 0000_2	将寄存器 Y 的内容与寄存器 Z 的内容相加，结果存到寄存器 X 中 RTL: $v0 ← $a0 + $a1
nand nand $v0,$a0,$a1	R	1 0001_2	将寄存器 Y 的内容与寄存器 Z 的内容做与非，结果存到寄存器 X 中 RTL:$v0 ← ($a0 && $a1)
addi addi $v0,$a0,25	I	2 0010_2	将寄存器 Y 的内容与立即值相加，结果存到寄存器 X 中 RTL:$v0 ← $a0+25
lw lw $v0,042($fp)	I	3 0011_2	从内存中取值到寄存器 X，内存地址为 OFFSET 与寄存器 Y 的内容之和 RTL:$v0 ← MEM[$fp +0x42]
sw sw $a0,042($fp)	I	4 0100_2	将寄存器 X 存入内存中，内存地址为 OFFSET 与寄存器 Y 的内容之和 RTL:MEM[$fp +0x42] ← $a0
beq beq $a0,$a1,done	I	5 0101_2	比较寄存器 X 和寄存器 Y 的值。如果相同，则转移到地址 PC+1+OFFSET，PC 是当前 beq 指令的地址 RTL:if($a0 ==$a1) PC ← PC+1+OFFSET
注意，从程序员方便（及实现者麻烦）的角度来说，汇编器根据指令给出的值或符号以及汇编器自己对 PC 的理解来计算 OFFSET 值。在例子中，汇编器将 done-(PC+1) 存在 OFFSET 中，所以机器能够在运行时跳转到“done”标签			
jalr jalr $at,$ra	J	6 0110_2	首先将 PC+1 存入寄存器 Y 中，此处的 PC 是 jalr 指令的地址，然后转移到寄存器 X 所保存的地址。注意，如果 X 和 Y 是同一个寄存器，那么处理器首先将 PC+1 存入寄存器，因此会转移到 PC+1 RTL: $ra ← PC + 1; PC ← $at 注意，无条件跳转能够通过指令 jalr $ra, $t0 并含有 $t0 的值来实现。这就是为什么在 LC-2200 中没有单独的跳转指令
nop	n.a.	n.a.	实际上是一条伪指令（即汇编器会产生下面的指令：add $zero, $zero, $zero）
halt halt	O	7 0111_2	

64

O 型指令（中断）

28 ~ 31 位：操作码

0 ~ 27 位：未使用（应为全 0）



2.10.2 LC-2200 寄存器组

正如前面所说，LC-2200 有 16 个程序员可见的寄存器。事实证明，在编译高级语言时，0 是个非常有用的小的整型数。比如，它常用来初始化程序变量。因为如此，我们决定让寄存器 R0 一直保持 0 值。写到 R0 的操作会被体系结构自动忽略掉。

我们为这16个寄存器起了便于记忆的名字，同时与2.8.3节中描述的软件惯例相一致。不仅如此，因为汇编器的需要，我们在寄存器名字前都加了个$符号。寄存器，助记名，预期的作用以及软件惯例总结如表2-2所示。

表2-2 寄存器惯例

寄存器编号	名字	用途	是否由被调用者保存
0	$zero	恒0（硬件保证）	n.a.
1	$at	为汇编器预留	n.a.
2	$v0	返回值	No
3	$ao	参数	No
4	$a1	参数	No
5	$a2	参数	No
6	$t0	暂存	No
7	$t1	暂存	No
8	$t2	暂存	No
9	$s0	要保存的寄存器	Yes
10	$s1	要保存的寄存器	Yes
11	$s2	要保存的寄存器	Yes
12	$k0	为OS/陷入预留	n.a.
13	$sp	栈指针	No
14	$fp	帧指针	Yes
15	$ra	返回地址	No

65

2.11 影响处理器设计的问题

2.11.1 指令集

在本章中，我们一直把注意力集中在指令集设计上面。我们对指令集首先关心的是能够将高级语言结构编译为有效的机器码。这种关心在某些角度上是正确且有意义的。然而，这并不是Intel或AMD这类公司的体系结构工程师通宵达旦所挂念的事情。事实上，在20世纪80年代和90年代出现了许多ISA，虽然它们的优雅程度不一，但都是由我们前面讨论的问题驱动的。从优雅和性能的角度来说，Digital Equipment Corporation（DEC）的Alpha体系结构是最好的体系结构之一。DEC Alpha的体系结构工程师对编译器生成代码的直观性和有效性以及ISA的设计如何能有效实现进行了大量的思考。随着DEC公司这个20世纪80年代和90年代的微型计算机先锋的死去，Alpha体系结构也走到了尽头。

20世纪80年代出现了复杂指令集计算机（Complex Instruction Set Computers, CISC）和精简指令集计算机（Reduced Instruction Set Computer, RISC）之间的争论。使用CISC型ISA，编译器作者的任务就会变得更复杂，因为将高级语言结构编译为机器码有太多的选择。除此之外，ISA的复杂性对于硬件的有效实现也是个巨大的挑战。对于编译器作者来说，有选择大体上是好事，但如果程序员不懂得性能通常需要优先考虑的话，那这些选择就有问题了。随着编译器技术的日渐成熟，有人认为RISC型的ISA比CISC型的ISA更加易于编译器作者使用，同时也能更有效地实现。

我们都知道，Intel的x86是一个经得住时间考验的ISA，它是一个CISC型的ISA。目前，

x86 还是占主导地位的 ISA。与此同时，许多更优雅的指令集，如 DEC Alpha，已经消失了。原因在于决定指令集成败的其他因素有很多（市场压力是主要因素）。性能因素当然是很重要的考量，但真正好的指令集，如 Alpha，它们相对于 x86 的性能优势还没有大到让它们成为主导者。另外，尽管 x86 指令集的硬件实现很有挑战性，但 Intel 和 AMD 聪明的体系结构工程师已经做出了有效的实现，由于它在时钟频率上很占优，因此一个“好的”指令集的性能优势并不大，至少没有足够大来取代已经占据市场的指令集如 x86。

究其根本，一个指令集的成败很大依赖于市场对它的接纳程度。今天计算机软件支持着从商业到娱乐的一切。因此，主要的软件商（如 Microsoft、Google、IBM 和 Apple）对指令集的接纳程度成了决定指令集成败的关键因素。另一个同等重要的因素是计算机制造商（如 Dell、HP、Apple 和 IBM）对采用这种指令集的处理器的接纳程度。除了传统的市场（笔记本、桌面电脑和服务器），嵌入式系统（例如游戏机、手机、掌上电脑和汽车）也成了计算机领域中的重要部分。很难精确地描述为什么某个指令被或不被这些软件巨头、计算机制造商和嵌入式系统开发者接纳。尽管我们趋向于认为这是由指令集的优雅程度决定的，但是从计算机的历史上看，并非如此。这些决定通常是依据语用学 pragmatic 做出的，即是否有针对该 ISA 的好的编译器可用（尤其是 C 语言），是否支持遗留代码，等等。 66

2.11.2 应用程序对指令集设计的影响

应用程序在过去影响着指令集的设计，今后还会继续影响。在 20 世纪 70 年代，甚至到 20 世纪 80 年代早期，计算机主要用于处理数字的科学和工程应用程序。这些应用非常依赖于浮点算术。高端计算机（如 IBM 370 系列机和 Cray）在指令集中包含了这些指令，而当时所谓的小型机（如 DEC PDP 11 系列）则没有包含这些指令。曾经有成功的公司（如 Floating Point Systems 公司）制作附加的处理器用于为小型机提供浮点运算加速。如今，浮点指令已经是任何通用处理器的一部分了。用于嵌入式应用如手机和掌上电脑的处理器（如 StrongARM、ARM）可能没有这些指令，它们通过数学库使用整数指令来实现浮点运算。

另一个应用程序影响指令集设计的例子是 Intel 的 MMX 指令。有的应用程序专门处理音频、视频、图像这样的流数据，即像电影和音乐这样的连续数据。这些数据在内存中通常表现为数组。MMX 指令由 Intel 公司于 1997 年在奔腾系列处理器中首先引入，目标是让 CPU 能够高效处理流数据。这套指令背后是很简单的直觉。正如流数据这个名称所暗示的，音频、视频和图像应用需要在两个或多个流相应的数据上进行相同的操作（比如加法）。所以，应该有指令去模仿这种行为。MMX 指令最初是在奔腾处理器中引入的，奔腾的后继型号也继承了它。一共有 57 条指令，分成算术、逻辑、比较、转换、移位和数据传输，每个指令都有两个操作数（不是标量，而是许多元素构成的向量）。比如，一个加指令会把两个向量中对应位置的元素相加。[1]

一个更近的例子来自于游戏产业界。交互式的游戏已经变得非常复杂。实时游戏控制输出的图像和动画处理需求已经超出了通用处理器的处理能力。当然，你下次假期旅行时不可能拖着一台超级计算机去玩游戏！于是出现了图像处理单元（GPU），这是专用的外接处理器， 67

[1] 市场压力的一部分是必须支持遗留代码，即那些在同类处理器的老版本上开发的软件。这种处理器向后兼容性为 Intel x86 ISA 的保持做出了巨大贡献。

[2] 历史上，MMX 指令从一种称为单指令多数据（SIMD）的并行体系结构进化而来。20 世纪 90 年代中期以前这种体系结构非常流行，用于满足图像处理应用的需求。参见第 12 章中对不同并行体系结构类型的介绍。

用于完成游戏控制器所需的算术运算。基本上，GPU包含了许多功能单元（实现多像素渲染应用所需的基本操作）来并行处理流数据。最近Sony、IBM和Toshiba的合作项目公开了Cell处理器，进一步发展了GPU的概念。Cell处理器在一块芯片上集成了几个处理元素，每个都能被编程以完成特定的任务。Cell处理器体系结构已经应用于PlayStation PS3中。

2.11.3 其他驱动处理器设计的问题

指令集设计只是现代处理器设计中的问题之一，甚至不是最引人关注的问题。这里列出了更多且更新的问题，其中的一些将在后面的章节中详细阐述。

1）**操作系统** 我们提到过，操作系统在处理器设计中扮演着重要的角色。它的一个表现是，系统给程序员造成一种假象，让内存空间看起来比实际内存容量大得多。另一个表现是处理器对中断等外部事件的响应能力。在后面的章节我们将看到，为了满足操作系统的需求，处理器需要包含一些新的指令，以及一些从指令集层面看不出来的体系结构的新机制。

2）**对现代语言的支持** 大多数现代语言如Java、C++和C#都向程序员提供了动态增长和缩小程序的能力。这称为动态内存分配，这个特性从应用并发程序员和操作系统资源管理的角度来说都非常有力。当数据尺寸缩小的时候恢复内存，即垃圾回收机制，是资源管理的关键。处理器体系结构中自动回收废弃的机制是当今处理器设计中的一个热点。

3）**存储系统** 如你所知，处理器的速度在过去十年中按照接近指数的规律增长着。例如，1986年，一台Sun 3/50拥有一颗0.5 MHz的处理器；2007年，笔记本和桌面电脑拥有超过2 GHz的处理器。内存的密度也按指数增长着，但内存的速度却跟不上处理器的增长速度。处理器和内存速度的这种不一致被称为内存墙。在处理器设计中，使用聪明的技术来越过这道内存墙是最重要的问题之一。例如，设计高速缓存并将它集成到处理器中就是这类技术。我们在后面的存储器层次章与中会包括这些内容。 68

4）**并行性** 随着芯片密度的提高，且现芯片上常常可以集成数以百万计的晶体管，这使得在单个处理器上能够放置更多的功能单元。实际上，芯片的密度已经达到可以在同一块硅片上放置多个处理器的高度。这种称为多核和众核的体系结构，带来了一系列全新的处理器设计的问题。[⊖]其中的一些问题，例如并行编程和内存一致性问题，是从传统的多处理器机器（含有多个处理器的计算机）上移过来的。我们在后面中会讨论它们。

5）**调试** 程序变得复杂了。类似于Web服务器这样的应用，除了并行和拥有庞大的内存等特点之外，可能还包含接触网络和数据库的组件。写这样的程序自然就不简单。现代处理器设计的一个盲点就是有效地支持调试，尤其是并行程序。

6）**虚拟化** 随着应用程序复杂性的增加，它们的需求也越来越复杂了。例如，一个应用可能用到某些服务，而这些服务只存在于某些特定的操作系统中。如果你要同时运行多个程序，而每个程序又有各自的需求，那么就需要支持同时存在多个应用执行环境。由于某些原因，你可能会在笔记本上安装系统。如果能够让多个操作系统共存，且不需要来回切换的话就好了。虚拟化是这样一个系统概念，在同一个计算机系统上支持多个不同的运行环境。体系和编程人员已经注意到了在现代处理器设计中有效地支持这个概念。

7）**容错性** 随着硬件体系结构变得更加复杂，有了多核与众核以及庞大的层次存储系统之后，硬件出错的可能性增加了。体系结构工程师们现在更加注意设计让处理器对程序员隐

⊖ 在体系结构上说，多核和众核之间并没有很大区别。与之更为常见的需求是说众核是指具有比少数几个（多达16）更多的核。因此，它们的区别就是，多核有不超过8或16个核，比这更多的就是众核了。

暴这些失败的技术。

8）安全性　在这个时代，计算机的安全是个大问题。当提到保护计算机安全的时候，通常会想到网络的攻击。事实证明即使在计算机内部（在内存系统和CPU之间）也会出现安全问题。体系结构上要响应试图在处理器和内存的通信中采用加密技术来缓解这类问题。 69

小结

指令集是硬件和软件之间的契约。在本章中，我们从基础开始讲解了指令集设计中的问题。需要记住的重要内容总结如下：

- 在编译ISA时高级语言结构的影响。
- 编译算术/逻辑运算、条件语句、循环和过程调用需要ISA提供的最低支持。
- 影响ISA对寄存器使用的一些实际问题（如寻址和访问时间）。
- ISA中与高效编译高级语言结构的需求相称的取得内存操作数的寻址模式。
- 处理器使用有限寄存器资源的软件惯例。
- 软件栈的概念及其在编译过程调用中的使用。
- 最小的ISA的一些可能的扩展。
- 当今影响处理器设计的其他重要问题。

练习题

1. 有人认为，处理器拥有大的寄存器文件对性能是有害的，因为在高级语言的过程调用/返回中会有更大的开销。你同意这种看法吗？给出你的理由。
2. 描写出栈指针和帧指针之间的区别。
3. 在LC-2200指令集中，加法指令的操作数在什么地方？
4. 这个问题和字节序有关。你现在要写一个比较字符串的程序。你可以选择使用32位字节寻址的大端或小端体系结构来实现。在这种情况下，你会将4个字符打包到一个字中。你的选择是什么，程序员该如何实现？（提示：正常情况下字符串是一个个字符进行比较，如果能够一个个字地进行比较，那速度就会快很多。）
5. ISA可能会支持多种形式的条件分支指令，比如BZ（为零时分支）、BN（为负时分支）、BEQ（相等时分支）。请指出哪一种形式最适合于if语句中的哪种谓词表达式，给出几个if语句中谓词表达式的例子，并说明在这些不同风格的条件分支指令下，如何编译这些条件语句。
6. 我们说过，字节序不影响程序的性能和正确性，前提是（高级）数据结构的使用方式和声明中的一致。有没有这样的情况，即使你遵守了上面的说明，仍然会受字节序的影响呢？（提示：考虑跨网络边界的程序。） 70
7. 使用汇编具体实现C语言的switch语句，用跳转表和任意形式的条件分支指令。（提示：先确认变量值在switch变量的合法范围内，然后跳转到当前值对应代码段的起始地址，执行，跳到退出）。
8. 过程A在S寄存器组和T寄存器组中都保存有重要数据，当A调用过程B时，A需要将哪些寄存器保存到栈中？哪些寄存器由B来保存？
9. 考虑过程调用执行时栈的使用。检查所有在栈上的操作都发生在栈顶（通过push和pop）？给出一些在程序执行中访问栈内部的情况，并解释这是如何发生的。
10. 判断下面这句话是否正确：若没有帧指针，则过程调用/返回无法实现。
11. DEC VAX有一个单条指令可以将所有程序可见的寄存器从内存中加载或保存到内存中。你能说出一个使用这一对指令的原因吗？有什么好处和坏处？

12. 如何使用已有的LC-2200 ISA来模拟一条减法指令。
13. BEQ指令限制了你从PC的当前位置转移到目标的距离。如果你的程序需要跳转的距离超出了BEQ指令中偏移量的范围，你该如何使用已有的LC-2200 ISA来实现这样的长跳转呢？
14. ISA是什么，它为什么很重要？
15. 指令集设计都受到哪些因素影响？
16. 条件语句是什么？ISA如何处理它们？
17. 给寻址模式下个定义。
18. 在2.8节中，我们提到过程的局部变量是在栈上分配的。这种描述有利于简化我们的阐述，但现代的编译器并不是这样做的。上网找找看现代编译器是如何为过程的局部变量分配空间的。（提示：强调一下寄存器比内存要快，所以目标应该是将尽可能多的变量放在寄存器中。）
[71] 19. 我们提到栈时使用了术语“抽象”。这个术语是什么意思？“抽象”暗含着它是如何实现的吗？例如，过程调用/返回时使用的栈是硬件实现还是软件实现？
20. 给出如下指令：

```
BEQ   Rx, Ry, offset    if (Rx == Ry) PC = PC+offset
SUB   Rx, Ry, Rz        Rx ← Ry - Rz
ADDI  Rx, Ry, Imm       Rx ← Ry + Immediate value
AND   Rx, Ry, Rz        Rx ← Ry AND Rz
```

你该如何实现下面指令的功能：

```
BGT   Rx, Ry, offset    if (Rx > Ry) PC = PC+offset
```

假设寄存器和立即数字段都是8位宽，忽略因减法而可能造成的溢出。

21. 给出下面的加载指令

```
LW    Rx, Ry, OFFSET    Rx ← MEM[Ry + OFFSET]
```

如何实现一种新的寻址模式，称为间接寻址模式，用汇编语言表示如下：

```
LW    Rx, @(Ry)
```

这条指令的语义是寄存器Ry的值是一个指针的内存地址，而这个指针指向的内存操作数需要装入Rx中。

22. 将语句

```
g = h + A[i];
```

转化为LC-2200汇编器，假设A的地址在$t0中，g在$s1中，h在$s2中，i在$t1中。

23. 假设你设计了一台计算机，名为“大循环2000”，它从不进行过程调用，在运行到末尾时会自动跳回到内存的起始位置。这时还需要程序计数器么？给出你的理由。
24. 某个处理器满足下面的假设：
    - 所有的参数通过栈进行传递。
    - 寄存器V0用于返回值。
    - 寄存器组S预期是被保存的，即，调用者不需做任何事情，在过程调用后S的值应该与调用前一样。
    - 寄存器组T预期是临时使用的，即，在调用了子过程之后，T中的值可能就变了。

    考虑下面的程序

```
int bar(int a, int b)
{
  /* 使用寄存器T5, T6, $t1-$t2的代码 */
  return(1);
}
[72] int foo(int a, int b, int c, int d, int e)
{
  int x, y;
  /* 使用寄存器T5-t10, $t1-$t5的代码 */
  bar(x, y); /* 调用 bar */
```

```
  /* 使用寄存器TS和参数a, b, c的代码 */
  return(0);
}

main(int argc, char **argv)
{
  int p, q, r, s, t, u;
  /* 使用T9 - T10和S11 ~ S15的代码 */
  foo(p, q, r, s, t); /* 调用foo */
  /* 使用寄存器T9,T10的代码 */
}
```

下面是在 bar 执行时的栈，请标明栈中的每一项是在哪个过程保存的。

main	foo	bar	
———	———	———	p
———	———	———	q
———	———	———	r
———	———	———	s
———	———	———	t
———	———	———	u
———	———	———	T9
———	———	———	T10
———	———	———	p
———	———	———	q
———	———	———	r
———	———	———	s
———	———	———	t
———	———	———	x
———	———	———	y
———	———	———	S11
———	———	———	S12
———	———	———	S13
———	———	———	S14
———	———	———	S15
———	———	———	T6
———	———	———	x
———	———	———	y
———	———	———	S11
———	———	———	S12
———	———	———	S13 ◄ 栈顶

参考文献注释和扩展阅读

为了照顾第一次接触系统课程的学生，本章中对指令集设计的介绍尽量简单易懂。世界上还有很多有影响的 ISA。IBM 360 系列 [IBM System/360，1964] 就是大型机发展过程中的一个里程碑。之后不久，IBM 又推出了 IBM 370 系列 [IBM system/370，1978]。360 和 370 系列机是 CISC 型 ISA 设计的案例。在计算机的演化中另一个有影响力的公司是数字设备公司（DEC）[Bell Web Page，2010]。DEC 的 PDP-8 是 20 世纪 60 年代出现的 12 位机器。它的指令集都基于处理器中仅有的一个寄存器，即累加器

[PDP-8，1975]。DEC 的 PDP-11 是 PDP-8 的 16 位后继者，主导了 20 世纪 70 年代～90 年代的小型机市场。PDP-11 的 ISA 是围绕 8 个寄存器设计的 [Bell，1970]。DEC 在 20 世纪 70 年代中后期推出了 VAX 11 体系结构，作为中端计算机应用的一种 32 位的选择 [Strecker，1978]。它是经典的 CISC 型体系结构。

20 世纪 80 年代出现了 RISC 类型的体系结构。William Joy 是 Sun Microsystem 的共同创始人之一，他写了一篇有意思的文章[⊖]记录了 RISC 的演化。在 1980 年，David Patterson 教授启动了 Berkeley RISC 项目 [Patterson，1981]，这后来成了 Sun Microsystem 的 SPARC 体系结构的基础 [SPARC Architecture，2010]。MIPS 在 20 世纪 80 年代早期作为 Stanford 的一个校园项目出现。这个项目的领导者 John Hennessy 教授，在 1984 年创建了 MIPS Computer Systems 公司。这个公司后来被 Silicon Graphics (SGI) 收购。

IBM 801[Cocke，2000；Radin，1982]在 20 世纪 70 年代末作为一个实验项目出现，领导者是 John
74 Cocke，他是 IBM 的一个计算机先驱和图灵奖得主。在 20 世纪 80 年代，这个项目中的许多想法出现在 IBM 的商业产品 POWER 体系结构中。由 Apple、IBM 和 Motorola 组成的联盟将 POWER 体系结构推广到了 PC 中。这一系列的处理器，称为 PowerPC，用于 Apple 的 Mac 中，直到 2006 年。HP 在 20 世纪 80 年代中期也发明了自己的 RISC 体系结构——Precision 体系结构，或称 PA-RISC[Mahon，1986]。HP 的工作站都使用这种体系结构，直到 2008 年。Alpha 处理器是 64 位的 RISC 体系结构，出自 DEC[Sites，1992]。尽管这种体系结构很有意思，实现也很新奇，但由于非技术的原因（公司并购）而退出了市场。

Intel 的 80x86 体系结构出现于 20 世纪 80 年代，至今仍在市场中占主导地位。它是 CISC 型的体系结构。了解 x86 指令集细节最好的办法就是从跟头开始 [Intel Instruction set，2008]。

[Patterson，2008] 教材是理解计算机基本组成和设计的极佳来源，尤其是关于本章的主题，即指令
75 集设计以及编译器在设计中起的引导作用部分。

⊖ 见网址 http://www.cs.washington.edu/homes/lazowska/cra/risc.html。

处理器实现

上一章探讨了处理器中的指令集体系结构设计的问题。本章探讨的问题是，有了指令集之后，处理器的实现。指令集并没有描述处理器如何实现，它只是硬件和软件之间的契约。例如，指令集设计好后，编译器可以为不同的高级语言生成代码，这些代码可以在实现这个指令集的处理器上执行。当然，我们对于同一套指令集可以有不同的实现。在本章中我们将看到，有许多因素会影响具体选择哪种实现。

3.1 体系结构与实现

首先我们要理解为什么要区分体系结构和实现。

1）由于性价比的原因，同一体系结构可能有也应该有多种不同的实现以满足市场需求。例如，在服务器市场（例如，Web 服务器）应该有高性能的处理器，与此同时，嵌入式系统（如打印机）中则需要处理器的低端版本。这就是为什么会有遵循同一种体系结构的处理器家族，其中有些甚至直接就叫做某个系列（如 Intel Xeon 系列、IBM 360 系列、DEC PDP-11 系列等）。

2）另一个去除体系结构与实现之间耦合的重要原因是，系统软件和硬件可以并行部署。例如，通过去耦合，使得我们可以验证针对某种新体系结构的系统软件（如编译器、调试器、操作系统），即使这种体系结构还没有可用的实现。这极大地减少了推出一套计算机系统所需的时间。

3）高性能服务器的客户在软件上做出了巨大的投入。例如，Oracle 数据库是一套庞大而复杂的数据库系统。与处理器更新换代相比，这类软件系统的进化非常缓慢。Intel 的合作创始人 Gordon Moore 在 1965 年预测，单位面积上的晶体管数量每两年翻一番。实际上，技术进步的速度要更快一些，处理器的速度每 18 个月翻一番。这意味着每 18 个月就会有一款更快的处理器冲击市场。如果你注意过每年新出的处理器的速度，就会明显感受到这一点。软件的改变则比硬件技术慢得多。因此，遗留软件能在新版本的处理器上运行是很重要的。我们应该维持契约（即指令集）稳定，以便大部分的软件（如编译器及相关工具，还有调优后的应用程序）在处理器的换代过程中基本保持不变。体系结构与实现的非耦合特性使保持传统软件的二进制兼容性得以满足。

3.2 处理器实现涉及什么

实现一个处理器需要考虑以下一些因素：价格、性能、功耗、散热、操作环境等。例如，用于军事的处理器需要更加坚固的实现以抵抗恶劣多变的环境。用于笔记本的同样的处理器则不需要这么坚固的实现。

处理器的实现主要有两个方面需要关注。

1）第一个关心的问题是电子部件（ALU、总线、寄存器等）的组成如何满足处理器的性能价格定位。

2）第二个关心的问题与热学和机械问题有关，包括散热以及在印刷电路板（通常称为主板）上放置处理器的物理上的考虑。

这两个问题是与单芯片处理器相关的。当然，计算机中的硬件不仅仅是一个处理器。还有许多其他问题需要综合考虑，包括印刷电路板、背板、连接器、底盘设计等。总的来说，计算机系统设计是多方面的一种权衡。如果我们只考虑高端市场（超级计算机、服务器、台式机），那么大概就是性能与价格的权衡。然而，对于手机这样的嵌入式系统来说，功耗（power consumption）、性能（performance）、面积（area）这三者的综合（通常称为 PPA）才是设计中的主导原则。

[77]

超级计算机	服务器	台式机和个人计算机	嵌入式
高性能是主要的目标	中等的性能和价格	价格是主要的目标	小尺寸，低成本和功能都是主要目标

原则上，计算机设计是一种依靠经验的工作，在多个维度上进行权衡就像猜谜一样。

本章中，我们着眼于处理器实现。尤其是，处理器的数据通路和控制。本章中的设计是一个基础版本。在第 5 章中，我们会探索流水线处理器的实现。

现在我们复习逻辑设计过程中可能已经讲过的一些重要的硬件概念。

3.3 重要的硬件概念

3.3.1 电路

组合逻辑 这种逻辑电路的输出是输入的布尔组合。也就是说，这里没有状态（即记忆）的概念。这种电路由基本的逻辑门（AND、OR、NOT、NOR、NAND）组成。另一种认识这类电路的方式是，它们没有从输入回到输入的反馈。

考虑一个混合了许多麦克风的输入并给扬声器产生一个复合输出的插线板。扬声器的输出取决于接线板选择的麦克风以便产生复合声音。这个接线板就是一种组合逻辑电路的例子。在处理器的数据通路中能发现的组合逻辑电路包括多路复用器、解复用器、编码器、解码器和算术/逻辑单元。

时序逻辑 时序逻辑电路的输出是当前输入与当前状态的布尔组合。除了组成组合逻辑电路的那些基本的逻辑门外，组成时序逻辑电路还需要一种称为触发器的记忆元件作为关键部分。

考虑一个车库门开关控制电路。这个电路的输入就是一个按钮以及一些表示门开着还是关着的开关。电路的输出是一个控制电机升高或降低车库门的信号。动作的方向取决于门的当前状态。因此，车库开关门控制器是一个时序逻辑电路。处理器数据通路中的寄存器和内存也是时序逻辑电路的例子。

[78]

3.3.2 数据通路的硬件资源

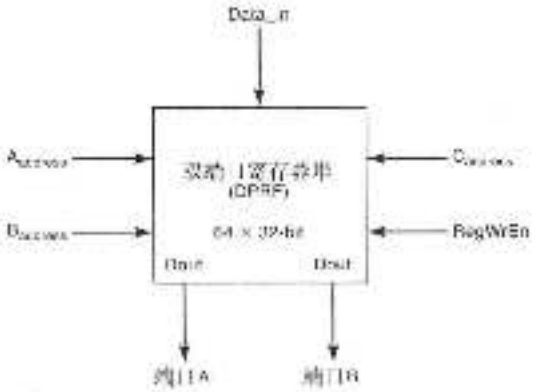
处理器的数据通路包含了组合逻辑与时序逻辑元件。根据第 2 章中给出的 LC-2200 指令集，我们来确定数据通路需要哪些资源。

我们需要内存来保存指令和操作数。我们需要算术/逻辑单元（ALU）来执行算术/逻辑指令。我们需要寄存器堆，因为它是大部分指令集体系结构中操作的重点。绝大部分指令使用寄存器堆。我们需要程序计数器（以后简称为 PC）来指向当前指令以及用于实现第 2 章中

讨论过的分支跳转指令。当一条指令从内存中取出后，它需要存储在数据通路的某个地方，所以我们引入了**指令寄存器**（IR）来保存指令。

顾名思义，寄存器堆就是体系结构中程序员可见的寄存器的集合。我们需要控制线路和数据线路来操作寄存器堆。这包括用来寻找某一特定寄存器的地址线以及读/写寄存器的数据线。只允许同时读单个寄存器的寄存器堆称为单端口寄存器堆（SPRF）。允许同时读两个寄存器的寄存器堆，称为双端口寄存器堆（DPRF）。例 3-1 给出了寄存器堆所需的所有控制线和信号线。

例 3-1 下图是一个双端口寄存器堆（DPRF），包含 64 个寄存器。每个寄存器 32 位长。$A_{address}$ 和 $B_{address}$ 是端口 A 和端口 B 所读寄存器的地址。$C_{address}$ 是数据 Data_in 写入的寄存器的地址。RegWrEn 是寄存器堆的写使能信号。图中的每个箭头各有几根线？



[79]

答：

a. Data_in 有 32 根线。

b. 端口 A 有 32 根线。

c. 端口 B 有 32 根线。

d. $A_{address}$ 有 6 根线。

e. $B_{address}$ 有 6 根线。

f. $C_{address}$ 有 6 根线。

g. RegWrEn 有 1 根线。

3.3.3 边沿触发逻辑

寄存器内容从当前状态改变到新状态是对时钟信号的响应（见图 3-1）。

输入变化引起输出变化的具体时间取决于这个元件是电平逻辑[⊖]还是边沿触发逻辑。在电平逻辑中，只要时钟信号是高电平，那么变化就会发生。而在边沿触发逻辑（见图 3-2）中，变化只会发生在时钟的上升沿或下降沿。如果状态变化发生在上升沿，则称为正边沿触发逻辑；如果变化发生在下降沿，则称为负边沿触发逻辑。

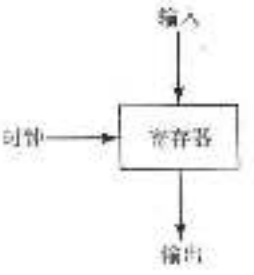


图 3-1 寄存器。只在响应时钟信号时输出才会变化

⊖ 习惯上所使用电平逻辑的存储器件称为锁存器。寄存器则通常用于指边沿触发的存储器件。

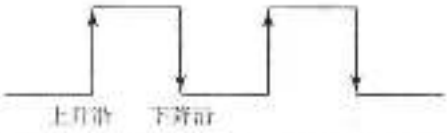
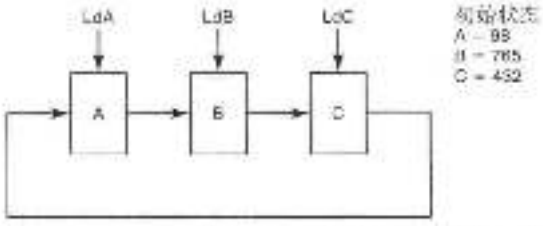


图 3-2 时钟。时钟周期指的是两个连续的上升沿（或下降沿）之间的时间间隔

在后面的讨论中，我们假设数据通路中所有的寄存器都使用正边沿触发逻辑。我们将在3.4.2节中讨论选择时钟周期宽度的细节。

例 3-2 寄存器 A、B 和 C 连接成了下面的电路：



LdA、LdB 和 LdC 是寄存器 A、B 和 C 的读入信号。如果在某个时钟周期内 A、B、C 的值如图中所示，那么下一个时钟周期它们的值又是什么？

答：

各个寄存器的输入都将会变成它们的输出，因此，

A = 432；B = 98；C = 765

内存元件比较特殊（见图 3-3）。正如第 1 章中看到的，在讨论计算机系统组成时，实际上，内存子系统是完全与处理器分离的。然而，出于简化处理器实现有关基本概念的目的，我们将内存包含在数据通路设计中。出于讨论的目的，我们认为内存不是边沿触发的。

例如，为了读取某个内存单元，你给内存提供“地址”和“读”信号，过了有限的一段时间（称为内存的读访问时间）后，该地址的内容就出现在“数据输出”线上。同样，为了写某个内存单元，你提供“地址”、“数据输入”，以及“读”信号，在一段有限时间（写访问时间）后，该内存单元的值就会变成通过“数据输入”写入的值。我们在第 9 章中会更详细地讨论内存系统。

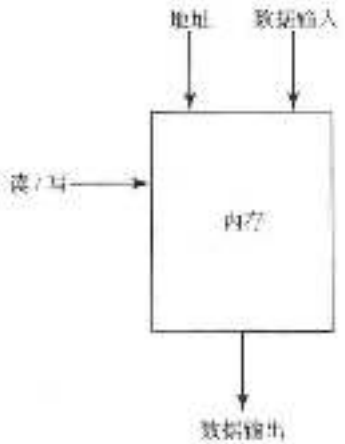


图 3-3 内存。在读操作时，给出“地址”一段时间后，指定的内存内容就会在“数据输出”线上出现

3.3.4 连接数据通路元件

让我们考虑执行 LC-2200 指令集中的 ADD 指令需要什么，并由此推出数据通路元件应该如何连接。

1）**步骤 1**：我们需要 PC 来指引指令在何处（见图 3-4）。

2）**步骤 2**：指令从内存中读出后，就保存到 IR 中（见图 3-5）。

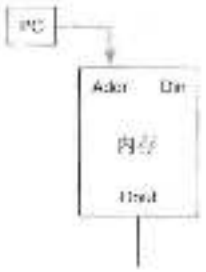


图 3-4 步骤 1：PC 给内存提供了指令地址

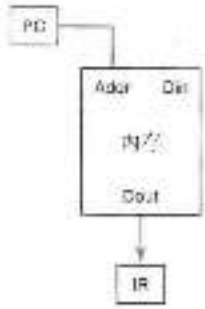


图 3-5 步骤 2：指令从内存中读出后，将时钟打入 IR

80
|
82

3）**步骤 3**：IR 中的指令可用后，就可以使用（IR 中的）指令中给出的寄存器号从寄存器堆（双端口，与例 3-1 中的类似）中读对应的寄存器。使用 ALU 进行加法操作，并将结果写回到寄存器堆中对应的寄存器（见图 3-6）。

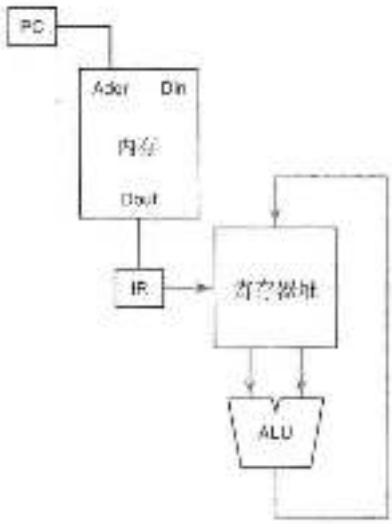


图 3-6 步骤 3：执行将两个寄存器值相加并写入第三个寄存器中

83

前面的 3 个步骤给出了 ADD 指令执行的路线图。我们看看这三步能不能在一个时钟周期内完成。前面提到，所有的存储元件（除了内存外）都是正边沿触发的。这意味着，在一个时钟周期内（如果不对门逻辑器件的延迟来说，时钟周期足够长的话），我们能将信息从一个存储元件传送到另一个存储元件（途中经过组合逻辑和内存）。所以，步骤 1 和步骤 2 可以在一个时钟周期内完成。但步骤 3 不能在同一周期内完成。在步骤 3 中，我们需要从 IR 中将寄存器号取出传送给寄存器堆。但由于 IR 的边沿触发特性，所以只有到下一个时钟周期后 IR 中的指令才可用（见图 3-7）。

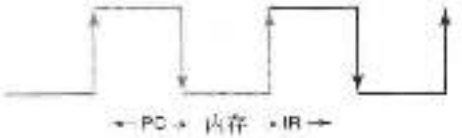


图 3-7 步骤 1 和步骤 2（第一个时钟周期），两个步骤都是在一个时钟周期内完成的

事实证明，步骤 3 可以在一个时钟周期内完成。在下一个时钟周期开始时，IR 的输出可以用来索引需要从寄存器堆中读出的具

体的源寄存器。寄存器值读出后（读寄存器的过程与读内存是类似的），将它们传递给 ALU，执行 ADD 操作，将结果写入目的寄存器（由 IR 给出）。图 3-8 阐明了步骤 3 在第二个时钟周期内的完成情况。

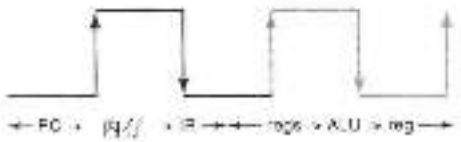


图 3-8 步骤 3（第二个时钟周期）：ALU 的结果在第二个时钟周期末出现在寄存器中

决定时钟周期 我们重新审视步骤 1 和步骤 2，我们说过它们可以在一个时钟周期内完成。要完成这些步骤，时钟周期需要多长呢？根据图 3-7，从第一个上升沿开始，我们可以枚举所有与这两个步骤相关的组合逻辑的延迟：

- 等待 PC 输出稳定到可以读取的时间（$D_{register-valid}$）。
- PC 输出传播到内存地址输入的线延迟（$D_{wire-PC-Addr}$）。
- 读取指定地址单元的内存访问时间（$D_{memread}$）。
- 内存读出值传播到 IR 输入的线延迟（$D_{wire-Data-IR}$）。
- 在第二个时钟上升沿前，IR 的输出需要达到稳定，这段时间称为建立时间（$D_{memsetup}$）。
- 在第二个时钟上升沿后，IR 输出需要保持不变一段时间，这称为保持时间（D_{hold}）。

[84]

完成步骤 1 和步骤 2 所需要的时钟宽度必须大于上述延迟的总和：

$$\text{时钟宽度} > D_{register-valid} + D_{wire-PC-Addr} + D_{memread} + D_{wire-Data-IR} + D_{memsetup} - D_{hold}$$

我们对每个时钟周期内所有可能的信号传播路径进行分析。然后，令时钟宽度大于整条路径的最坏情况下信号传播延迟。在 3.4.2 节中，我们将形式化定义计算时钟周期涉及的术语。

例 3-3 给出下面的参数（单位是皮秒（ps）），确定系统所需要的最小时钟宽度（仅考虑前面所说的步骤 1-3）：

$D_{register-valid}$	（PC 输出稳定）	20ps
$D_{wire-PC-Addr}$	（从 PC 到内存地址的线延迟）	250ps
$D_{memread}$	（内存读）	1500ps
$D_{wire-Data-IR}$	（从内存数据输出到 IR 的线延迟）	250ps
$D_{memsetup}$	（IR 的建立时间）	20ps
D_{hold}	（IR 的保持时间）	20ps
$D_{wire-IR-regfile}$	（从 IR 到寄存器堆的线延迟）	250ps
$D_{regread}$	（寄存器读）	500ps
$D_{wire-regfile-ALU}$	（从寄存器堆到 ALU 输入的线延迟）	250ps
D_{ALU-OP}	（执行 ALU 操作的时间）	100ps
$D_{wire-ALU-regfile}$	（从 ALU 输出到寄存器堆的线延迟）	250ps
$D_{regfilesetup}$	（写入寄存器堆的时间）	500ps

答：

步骤 1 和步骤 2 在一个时钟周期内完成。这两个步骤需要的时钟宽度是

[85]

$$C_{1,2} > D_{register-valid} + D_{wire-PC-Addr} + D_{memread} + D_{wire-Data-IR} + D_{hold} > 2060ps$$

步骤 3 占据一个时钟周期，所以所需要的时钟宽度为

$$C_3 > D_{wire-IR-regfile} + D_{regread} + D_{wire-regfile-ALU} + D_{ALU-OP} + D_{wire-ALU-regfile} + D_{regfilesetup} > 1850ps$$

最小时钟宽度 > 最坏情况下信号传播延迟

$$> MAX(C_{1,2}, C_3)$$

$$> 2060ps$$

例子中的数据是相当准确的数据（2007 年前后）。从例子中明显可以看出线延迟占主要地位。

3.3.5 基于总线的设计

为了执行 ADD 指令，为数据通路上的元件建立了专门的连接。为了实现其他指令（如 LD），我们需要建立一条从内存到寄存器堆的路径。以此类推，我们可以想象数据通路上的所有元件都相互连接。事实证明，这没有必要，也不是正确的方法。我们检查连接 ALU 到寄存器堆涉及什么。我们需要与位宽相应数量的线路来连接两个元件。对于 32 位的机器，需要 32 根线。当我们增加数据通路元件的连通性时，需要的线路就会快速增加。从占用硅片面积的角度来说，连线是非常昂贵的，所以我们需要减少连线数量以使硅片真正用于数据通路中活动的元件。而且，仅仅增加线路并不会带来性能的提高。例如，从内存到寄存器的线路对于 ADD 指令的实现一点帮助都没有。

所以，我们需要更加细致地考虑数据通路元件的连接问题。具体来说，前面讨论给我们的启发是，与其在每两个元件之间连线，不如设计数据通路让各元件共享线路。让我们来研究需要多少线以及如何共享它们。

单总线设计 最极端的一种情况是，只有一组总线，所有的元件都共享它。这就像小组开会一样，一个人说，大家都在听。如果有必要参与讨论，每个人都会轮到发言。如果同时有多人讲话，当然就会很混乱。这就是单总线系统（一组线路被所有元件共享）的工作方式。图 3-9 就是这样一个系统。

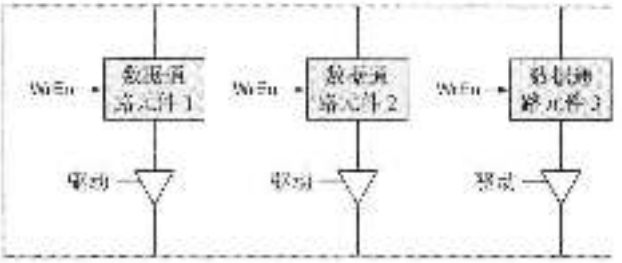


图 3-9 单总线设计。仅有一个数据通路元件能够将它的输出放到总线上，这是通过相应的“驱动”信号来实现的

总线说明这组线路是共享。第一点需要注意的是，灰线是一条电气单总线，也就是说，总线上的数据在总线的任意一段都是可用的。第二点需要注意的是，在元件和总线之间有些三角形。它们是驱动器（也称为三态线冲器[⊖]）。每个元件需要连接到总线的输出上，都会有这样一个驱动器。它们将数据通路元件和总线进行电气隔离。所以，为了“连接”元件 1 与总线，相应的驱动器必须是“开”的。这可以通过选择相应的“驱动”信号来完成。然后，我们就说数据通路元件 1 在“驱动”总线。同时存在多个元件驱动总线是错误的。所以控制逻辑的设计者需要保证任意一个时钟周期内仅有一个驱动器处于“开”状态。如果多个驱动器同时处于“开”状态，那么除了总线上的值变得无法预测之外，还可能对电路造成致命伤害。另一方面，总线上的元件在每个时钟周期都会尝试获取总线上的数据。为此，元件上对

⊖ 一个二进制信号通常处于 0 或 1 两种状态中。而驱动器总输出没有被使能时则是第三种状态，不是 0 也不是 1，是一种称为高阻态的状态，驱动器将总线与元件在电气上隔离开来。所以叫作三态线冲器。

应的 WrEn（写使能）信号必须为“开”。

双总线设计 图 3-10 展示了一个双总线设计。在这个设计中，寄存器是双端口的，类似于例 3-1 中的那样。因此，在一个时钟周期内可以同时读两个寄存器并传送给 ALU。顶部的灰线和底部的黑色点线都是传输地址和数据值的总线，具体传输的内容取决于在这个时钟周期内需要什么。然而，在名义上，灰色总线传输地址而黑色虚线总线在元件之间传输数据。

虽然图中没有画出，但在每个元件的输出端都有驱动器连接到总线上。完成 3.3.4 节中的步骤 1 ~ 3 需要几个时钟周期呢？每个周期分别发生了什么呢？

我们来探讨这两个问题。

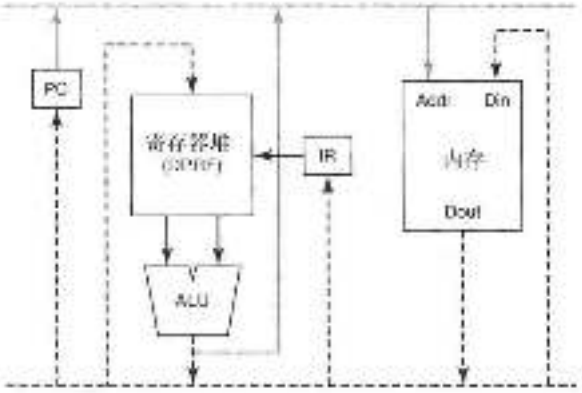
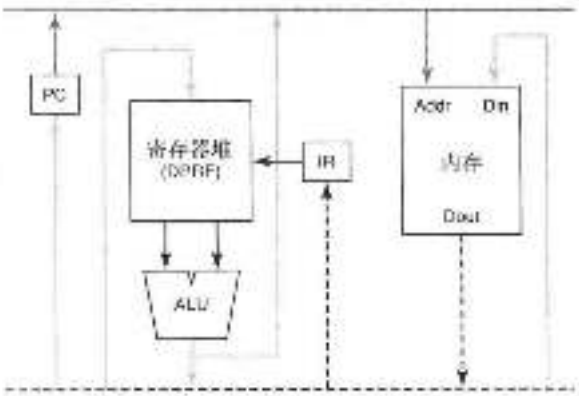


图 3-10 双总线设计。目的是让元件之间两个独立的会话能够同时进行

第一个时钟周期：

- 从 PC 到灰色总线（注意：在这个周期内没有其他元件能够驱动灰色总线）。
- 从深灰色总线到内存地址。
- 内存读取 Addr 指定的单元。
- 数据从 Dout 到黑色虚线总线（注意：此时没有其他元件能驱动黑色虚线总线）。
- 从黑色虚线总线到 IR。
- 时钟触发 IR。

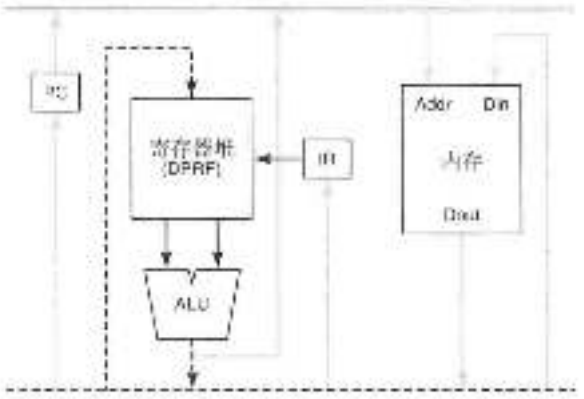
我们在一个时钟周期内完成了步骤 1 和步骤 2。



第二个时钟周期：

- IR 将寄存器号提供给寄存器堆（见 IR 到寄存器堆的那个箭头，它表示对应的线路），包括两个源寄存器和一个目的寄存器。
- 读寄存器堆，从两个源寄存器中取出数据。
- 寄存器堆将两个源寄存器的数据值提供给 ALU（见寄存器堆到 ALU 的箭头表示的线路）。
- 执行 ALU 的 ADD 操作。
- 将 ALU 结果提供给黑色虚线总线（注意，这个时钟周期内没有其他元件能够驱动黑色虚线总线）。
- 从黑色虚线总线到寄存器堆。
- 根据 IR 给出的目标寄存器号，写入寄存器堆。

在这个时钟周期内我们完成了步骤 3。



前面讨论中最关键的地方是，我们使用两根共享总线（以及寄存器堆到 ALU 的连接和 IR 到寄存器堆的选择线路）而不是每一对元件间的专设线路完成了步骤 1 - 3。

3.3.6 有限状态机

目前为止，我们总结了电路元件以及它们如何组装到处理器的数据通路中。这只是处理器设计的一部分。处理器设计中同等重要的一部分是控制单元。最好将控制单元理解为一个有限状态机（Finiut State Machine，FSM），因为将数据通路通过状态切换来完成指令的执行。

86 - 89

有限状态机，顾名思义，有有限个状态。在图 3-11 中，标记为 S1、S2、S3 的圆圈是 FSM 的状态。箭头则是状态之间的转移。FSM 是任意时序逻辑电路的抽象。它描述了电路的行为。FSM 的状态对应着时序逻辑电路的某些实际的物理状态。描述一个转移的两个参数是：1）触发状态发生变化的外部输入；2）电路在状态转移中产生的输出信号。因此，FSM 描述实际电路所有硬件细节是很方便的。

例如，图 3-12 中的简单 FSM 就表示了之前介绍的车库门开关控制电路。表 3-1 给出了这个 FSM 的状态转移表，包括导致转移的输入以及产生的相应输出。

状态“打开”表示门是开着的，而“关闭”表示门是关着的。输入是一个遥控器按钮。输出是控制马达开关门的信号。标记为“0”和“1”的转移表示是没有按下按钮的情况。标记为“2”和“3”的转移则表示按钮被按下。转移“2”产生一个控制门升起的输出信号，而转移“3”产生一个使门降下的信号。学过逻辑设计课程的人都知道，给出 FSM 和状态转移

表后，设计时序逻辑是一种很简单的练习。（见本章末尾与车库门开关控制器有关的两道题。）

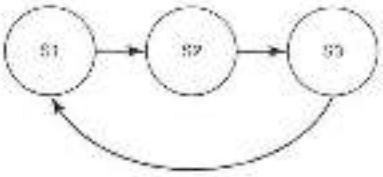


图 3-11 有限状态机（FSM）。圆圈表示状态，箭头表示状态转移

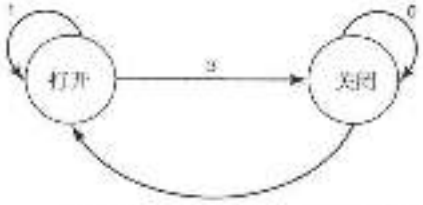


图 3-12 车库开关门控制器的 FSM。各个转移如状态转移表（见表 3-1）所示

表 3-1 图 3-12 中 FSM 的状态转移表

	状态			
转移号	输入	当前状态	下一状态	输出
0	无	关闭	关闭	无
1	无	打开	打开	无
2	按下按钮	关闭	打开	电机往上拉
3	按下按钮	打开	关闭	电机往下拉

我们知道时序逻辑可以是同步的或异步的。对于前者，状态转移是与时钟沿同步的，而后者的状态转移在输入变化时就立刻发生。

处理器的控制单元也是时序逻辑电路。我们用图 3-13 中的 FSM 表示控制单元。

FETCH：这个状态表示将指令从内存中取出。

DECODE：这个状态表示对读出的指令进行解码以确定需要什么操作数以及做什么操作。

EXECUTE：这个状态表示执行指令。

我们在 3.5 节中还会再讲到控制单元。

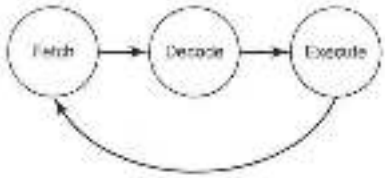


图 3-13 控制 CPU 数据通路的 FSM

3.4 数据通路设计

中央处理单元（CPU）包括数据通路和控制单元。数据通路含有所有的逻辑元件，而控制单元根据处理器的指令集为数据通路提供控制信号。

数据通路是硬件资源及其连接的结合体。我们看看数据通路需要什么硬件资源。我们看到过，指令集体系结构本身已经明确选择了一些硬件资源。一般来说，除了指令集显式要求的之外，我们还需要更多的硬件资源。

为了使讨论更具体，我们先给出 LC-2200 指令集要求的硬件资源：

1）能够进行 ADD、NAND、SUB 运算的 ALU。

2）包含 16 个 32 位寄存器的寄存器堆，如图 3-14 所示。

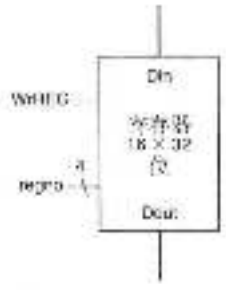


图 3-14 单输出端口的寄存器堆。4 位的“regno”信号唯一确定了 16 个寄存器中的一个；“WrREG”信号指定了对该寄存器进行读操作还是写操作

3）32 位的 PC。

4）2^{27} × 32 位字的内存。

内存是 LC-2200 指令集中用来存放指令和数据的硬件资源。内存的大小是一种实现上的选择。体系结构只是通过寻址能力限制内存空间的最大大小。LC-2200 具有 32 位寻址能力，所以最大的内存空间应该是 2^{32} 个字，而每个字为 32 位。

我们看看还需要什么额外的硬件资源。我们提到过，当指令从内存中读出后，它需要保存在数据通路的某个地方。IR 正是用于此目的。假设我们想用单条总线将所有这些元件连接起来，先不管总线的宽度，看看寄存器堆就能发现很明显的问题。我们只能从寄存器堆中获得一个寄存器的值，因为它只有一个输出端口（Dout）。ALU 操作需要两个操作数。所以，我们需要在数据通路中使用一些临时的寄存器来保存一个操作数。而且，使用单总线时，所有元件之间都只有一条通道。这就是我们将寄存器 A 和 B 放在 ALU 前面的原因。由于类似的原因，我们需要存放 ALU 结果的内存地址，于是出现了**内存地址寄存器**（MAR）。Z 寄存器（1 位寄存器）的作用在后面讨论指令集实现的时候自然会知道。在 Z 寄存器前面的零检测组合逻辑（见图 3-15）检测总线上的值是否为 0。根据指令集的硬件资源、数据通路的限制以及实现指令集的实际需要，我们最终得出了一个单总线设计，如图 3-15 所示。

90 ~ 92

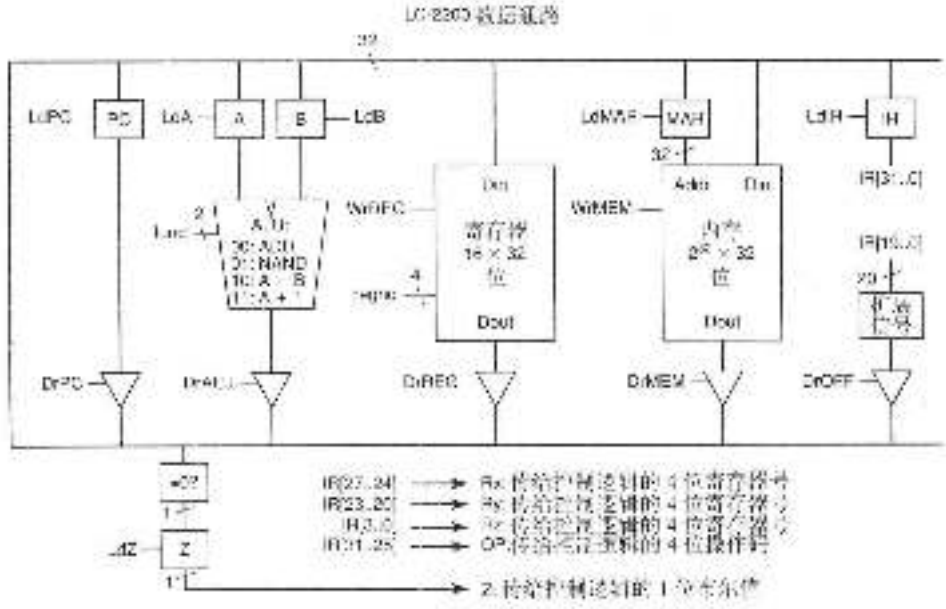


图 3-15 LC-2200 数据通路。有些资源是由 ISA 指定的，另一些则是由于单总线的限制而加上去的

3.4.1 ISA 与数据通路宽度

我们将 LC-2200 定义为一个 32 位指令集体系结构。因此，所有的指令、地址和操作数都是 32 位的。我们现在将探讨这种体系结构对数据通路设计带来的影响，还要对总线和 ALU 等其他部件的影响。

但从逻辑设计的角度来说，很容易实现两位精度的硬件实现高精度的算术和逻辑运算。例如，如果你愿意，你可以使用 1 位加法器来实现 32 位加法。虽然这会很慢，但确实可以实现。

93

同样，你可以让图3-15中的总线变得比32位窄一些。这样的选择会影响指令的执行。比如，如果你使用了8位宽的总线，那么你需要4次才能从内存中读出一条指令或内存操作数。再次为这个选择付出性能上的代价。

我们希望使用比ISA要求更低精度的硬件和更窄的总线，这又是一个性能与价格权衡的问题。因为大部分芯片面积都被连接线占据了，所以总线越窄、处理器的实现就越廉价。使用低精度的硬件也有同样效果，因为它会减小数据通路中连线的宽度。

所以，数据通路的设计体现出性能价格的权衡。这就是我们在3.1节中说的，芯片厂商会提供一种处理器位于性能价格曲线上不同位置的多个版本。

为了我们的讨论，假设数据通路上对体系结构可见的部分（PC、寄存器堆、IR、内存）都是32位宽。

3.4.2 时钟脉冲宽度

在3.3节中（见例3-3），我们非形式化地讨论了如何计算时钟周期宽度。现在我们形式化地定义与时钟周期计算有关的术语：

- 每个组合逻辑元件（例如，ALU或图3-15中的驱动门）都有一个从输入到输出传播值的一段延迟，这称为传播延迟。
- 类似地，从寄存器允许读取（例如，图3-15中，将regno值传送到寄存器堆）到将内容传送到输出端口（Dout）也有一段延迟（称为访问时间）。
- 对于要写入寄存器的情况，输入到寄存器在时钟上升沿到来之前一段时间要保持稳定（即输入值不再变化），这段时间称为建立时间。
- 类似地，在时钟上升沿到来之后，输入到寄存器还需要保持稳定一段时间，这称为保持时间。
- 最后，一个值从某个元件的输出通过线路出现在另一个元件的输入（例如，在图3-15中从驱动门的输出到PC的输入）的这段时间称为传输延迟（也称为线延迟）。

因此，如果我们希望在一个时钟周期内读取寄存器堆中的某个值并将它放入寄存器A，我们需要将一系列的延迟相加。我们计算所有需要在单个时钟周期内完成的数据通路操作的最坏情况的延迟。这就给出了时钟周期的下限。

94

3.4.3 检查点

到目前为止，我们回顾了下面的硬件概念：

- 基本的逻辑设计，包括组合逻辑电路与时序逻辑电路。
- 数据通路中的硬件资源，如寄存器堆、ALU和内存。
- 边沿触发逻辑和时钟周期宽度。
- 数据通路连接和总线。
- 有限状态机。

我们使用这些概念为LC-2200指令集体系结构建立了一条数据通路。

3.5 控制单元设计

如图3-16所示，管弦乐队指挥的作用是告诉乐队什么时候谁需要演奏或演唱。乐队成员自己知道自己要演奏什么，所以指挥不需要管理演奏内容，只需要保持节奏和次序就可以了。

如果说数据通路是乐队，那么控制单元就是指挥。控制单元为数据通路中的元件完成自己的工作提供一些提示。例如，如果 DrALU 线被断言（即，如果线上值为 1），那么相应的驱动器门就会将 ALU 的输出放到总线上。



图 3-16 一个管弦乐队的布置。指挥的作用近似于处理器中的控制单元。她为乐队中的每个成员给出“时间提示”

检查数据通路后，我们给出控制单元所需要的控制信号：

- **驱动信号**：DrPC、DrALU、DrREG、DrMEM、DrOFF。
- **加载信号**：LdPC、LdA、LdB、LdMAR、LdIR、LdZ。
- **写内存信号**：WrMEM。
- **写寄存器信号**：WrREG。
- **ALU 功能选择器**：func。
- **寄存器选择**：regno。

如何产生这些控制信号有许多可能的实现。所有的实现都是对于处理器控制单元的 FSM 的硬件实现。

3.5.1 ROM 加状态寄存器

我们来看一个非常简单的设计。首先，我们需要知道处理器处于什么状态。前面我们介绍了控制单元的 FSM，它包含 FETCH（取指）、DECODE（译码）和 EXECUTE（执行）状态。这是 FSM 抽象中处理器宏的状态。在真实实现中，根据数据通路的功能，需要许多微状态来表示宏状态的细节。例如，我们假设需要 3 个微状态来实现 FETCH 宏状态。我们将这些为微状态进行编码：

```
ifetch1   0000
ifetch2   0001
ifetch3   0010
```

现在我们引入**状态寄存器**，它的内容就是这些微状态的编码。所以，在任何情况下，这个寄存器的值都表示处理器的状态。

状态寄存器的引入让我们又从 FSM 抽象向硬件实现接近了一步。下面，为了在每个微状态下控制数据通路的各个元件，我们需要产生之前列出的控制信号。下面我们来讨论如何才能产生这些控制信号。

一种最简单的方法是，用状态寄存器作为表的索引。表中的每一项都包含该状态所需要的全部控制信号。用前面的乐队做比喻，指挥面前有整个音乐的乐谱。在演奏曲子时，指挥指导管弦乐队的演奏。指挥能从乐谱的每一行中看出，谁应该在哪一个时间点演奏哪一个音符。同理，我们能从控制单元表的表项看出，数据通路的元件在这个状态下应该做什么。每个演奏者都知道自己需要演奏什么。同样，每个数据通路元件也知道自己的功能。两者的相同之处是，他们都需要别人（别的东西）告诉他们现在该做什么。所以，指挥者和控制单元有着极其类似的工作，他们在时间上为演奏者和数据通路提供了必要的线索（解决了“什么时候”的问题），使其能够在正确的时间做相应的事情。这似乎很简单，所以表中的每个控制信号用1位表示。如果该位为1，则表示产生该控制信号；为0，则不产生。当然，func和regno字段的位数与它们在数据通路中的宽度有关（见图3-15，分别为2位和4位）。图3-17给出了控制信号表中的表项。

95 / 96

	驱动信号					加载信号						写信号			
当前状态	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	regno

图3-17 控制信号表的表项

控制单元需要从一个状态转移到另一个状态。例如，FETCH宏状态需要3个微状态，那么就会存在下面的情况：

当前状态	下一状态
ifetch1	ifetch2
ifetch2	ifetch3

如果将下一个状态也写在表中，那么状态转移就变得相对简单。现在来看看图3-18中的表。

	驱动信号					加载信号						写信号				
当前状态	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	regno	Next state

图3-18 将下一个状态加入控制信号表中

让我们来研究如何在硬件上实现这个表。

这个表其实就是存储元件。这种存储元件的特性是，一旦我们决定了某个状态所需要的控制信号，该表项的内容就被冻结。我们称这种存储器为只读存储器，或ROM。

所以，控制单元的硬件实现看起来就像图3-19那样。在每过一个时钟周期上，状态寄存器就转移到使用当前时钟周期内ROM表项输出指定的下一个状态。这是驱动数据通路中所有边沿触发存储元件的同一个时钟（见图3-15）。从ROM发出的所有加载信号（LdPC、LdMAR等）充当时钟信号的掩码，决定它们控制的存储元件是否在该时钟周期内被驱动。

下一件要做的事情就是将数据通路和控制单元结合起来。只要简单地将数据通路（见图3-15）的相应命名的实体与对应的ROM输出连接起来。

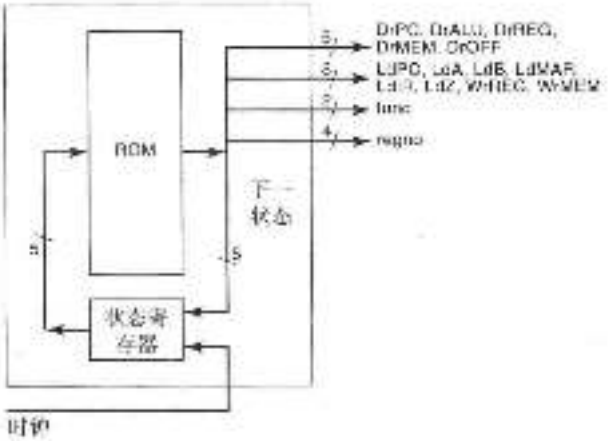


图 3-19 控制单元，使用状态寄存器和 ROM。ROM 的每一行都包含了当前时钟周期内需要产生供数据通路使用的控制信号（即“状态”）

控制单元的工作方式如下：

1）状态寄存器给出了本时钟周期内处理器的状态。

2）通过状态寄存器的值对 ROM 进行索引访问。

3）ROM 的输出就是当前传送给数据通路的控制信号集合。

4）在本时钟周期内数据通路执行控制信号指定的功能。

5）将 ROM 给出的下一状态传送给状态寄存器的输入，使之在下一个时钟周期开始时能够转移到下一个状态。

每个时钟周期都重复上述 5 个步骤。

图 3-13 将处理器的控制单元描述为一个 FSM。为了方便起见，我们再将它画在图 3-20 中。现在我们来检查图 3-20 中的每个宏状态需要发生什么，还有控制单元如何实现它们。对于每个宏状态，我们在旁边标出了数据通路的操作。

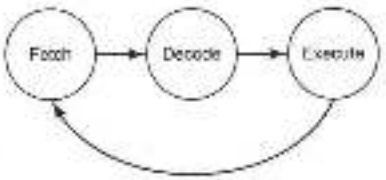


图 3-20 CPU 数据通路引进的 FSM

3.5.2 FETCH 宏状态

FETCH 宏状态从程序计数器（PC）所指出的内存中取出一条指令到指令寄存器（IR）中，为了读取下一条指令，还会将 PC 递增。

现在我们列出实现 FETCH 宏状态需要做什么：

- 将 PC 发送给内存。
- 读出内存的内容。
- 将内存的内容发送到 IR 中。

- 递增 PC。

很明显，如果使用单总线数据通路，那么这些步骤无法在一个时钟周期内完成。

- ifetch1
  PC → MAR
- ifetch2
  MEM[MAR] → IR
- ifetch3
  PC → A
- ifetch4
  A + 1 → PC

稍加思索，我们就能在少于 4 个时钟周期内完成 FETCH 宏状态的工作。观察在 ifetch1 和 ifetch3 中发生了什么。PC 的内容传递给了寄存器 MAR 和 A。这两个状态可以合并为一个状态，因为一旦 PC 的值输出到总线上，在一个周期内这两个寄存器都可以取得这个值。所以，我们可以将上面的序列简化为：

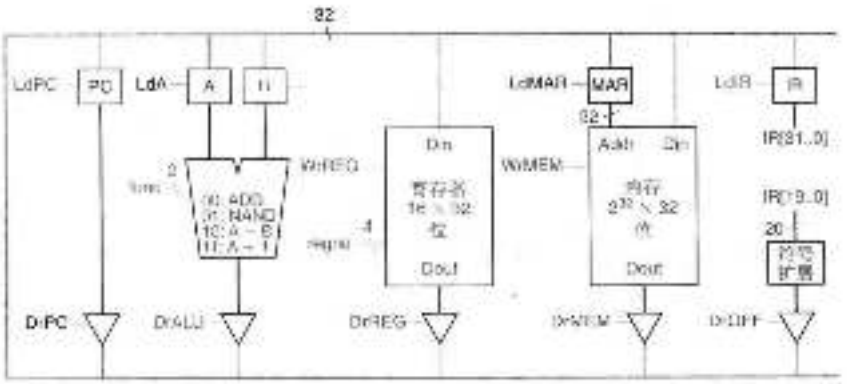
- ifetch1
  PC → MAR
  PC → A
- ifetch2
  MEM[MAR] → IR
- ifetch3
  A + 1 → PC

既然我们已经知道实现 FETCH 宏状态需要数据通路在每个微状态时做些什么，那么可以给出每个微状态需要的控制信号。对于每个微状态，我们将用到的数据通路元件和控制线高亮表示。

97
|
99

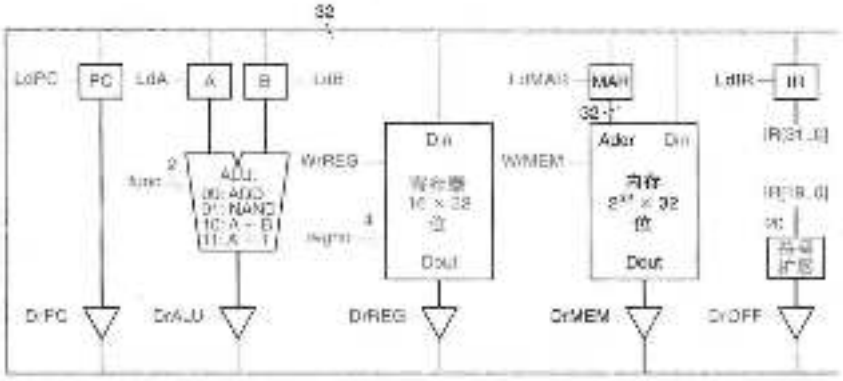
- ifetch1
  PC → MAR
  PC → A
  所需的控制信号：
  DrPC
  LdMAR
  LdA

下图显示了 ifetch1 微状态的数据通路活动。



- ifetch2
  MEM[MAR] → IR
  所需的控制信号：
  DrMEM
  LdIR

下图显示 ifetch2 微状态的数据通路活动。

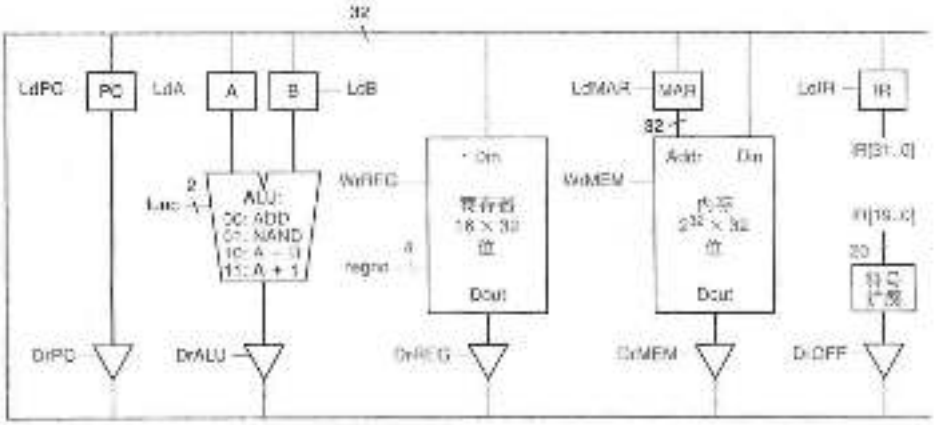


100

注意：根据数据通路的设计，对内存的默认类作为读类作（即 WrMEM 为 0）。而且，内存随式地读出了 MAR 指定地址的内存内容并在 ifetch2 中的 Dout 上给出了结果。

- ifetch3
  A + 1 → PC
  序列控制信号：
  func = 11
  DrALU
  LdPC

下图显示了 ifetch3 微状态的数据通路活动。



注意：如果 func 选择的信号为 11，那么 ALU 执行的操作是 A+1（见图 3-15）。

现在我们可以给出与微状态 ifetch1、ifetch2、ifetch3 对应的 ROM 的内容（见图 3-21，X 表示无所谓）。ifetch3 的下一状态字段临时标记为 TBD（To Be Determined，未决定）。很快我们会讨论这个字段。

这开始看起来像一个程序了，虽然它比我们第一节编程课上讲的程序要低层得多。每个 ROM 单元都包含驱动数据通路不同元件工作的命令集合。我们将每个表项称为一条微指令，并将 ROM 的整个内容称为一个微程序。每条微指令还包含下一条要执行的微指令的地址。现在，控制单元的设计变成了一个编程练习。它是一个最后的并发程序，因为我们在每条微指令中都利用了硬件的并行性。

101

		驱动信号					装载信号						写信号				
当前状态	State num	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	regno	下一状态
Ifetch1	00000	1	0	0	0	0	0	1	0	1	0	0	0	0	xx	xxxx	00001
Ifetch2	00001	0	0	0	1	0	0	0	0	0	1	0	0	0	xx	xxxx	00010
Ifetch3	00010	0	1	0	0	0	1	0	0	0	0	0	0	0	11	xxxx	TBD

图 3-21 一些填充了控制信号的 ROM 表单

注意每条微指令都是有结构的。例如，所有的驱动信号可以分成一组。类似地，所有加载信号也可以分成一组。这样的设计是为了节约表的空间需求。例如，可以将某些控制信号结合到一个编码字段中。因为我们知道任意时刻在总线上只有一个实体能够被驱动，所以我们可以将所有的驱动信号集成为 3 位宽的字段。这个字段的每个编码都唯一指定了总线上的一个应该被驱动的实体。这种方法在减少空间占用的同时，需要一个额外的解码环节，这会增加生成控制信号的延迟。我们无法将所有加载信号都集成在一起，因为可能会有多个存储元件在同一时钟周期内工作。

3.5.3 DECODE 宏状态

一旦将指令取出，就准备好进行解码了。所以，我们需要从 ifetch3 微状态转移到 DECODE 宏状态。

在这个宏状态中，我们检查 IR 的内容（位 28 – 31）以识别这是什么指令。知道是什么指令后，就需要找到实现这个指令的微程序。所有我们可以认为 DECODE（解码）过程是基于指令 OPCODE（操作码）的多路分支。为了更好地描述解码的多路分支特性，我们将控制单元的 FSM 重绘如下（见图 3-22）。多路分支的每一路都将 FSM 引向某一条指令的具体执行，为了简单起见，我们只画出 FSM 转移到不同类型的指令。

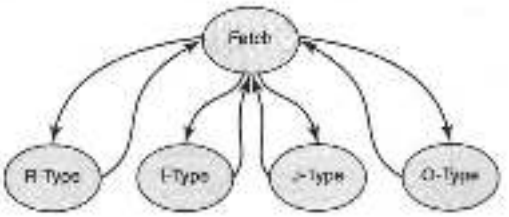


图 3-22 将 DECODE 宏状态展开后的 FSM

很快我们将回到如何在控制单元中实现多路分支的问题。现在我们先讨论每种指令的一些简单实现。

3.5.4 EXECUTE 宏状态：ADD 指令（R 型指令部分）

R 型指令具有下面的格式：



ADD 指令执行下面操作：

$R_X \leftarrow R_Y + R_Z$

为了实现这个指令，我们需要从寄存器堆中读取两个寄存器并将结果写入第三个寄存器。需要读取的寄存器指定为指令的一部分，且保存在 IR 中，在数据通路中是可用的。然而，从数据通路中可以看到，从 IR 到寄存器堆之间没有通路。这个"通路"是有原因的。根据我们要读或写的寄存器，我们需要将 IR 中的不同部分作为寄存器堆的 regno 输入。正如我们所见，多路复用器是完成这类选择的逻辑元件。

所以，我们需要将这个元件加入图 3-23 的数据通路中。

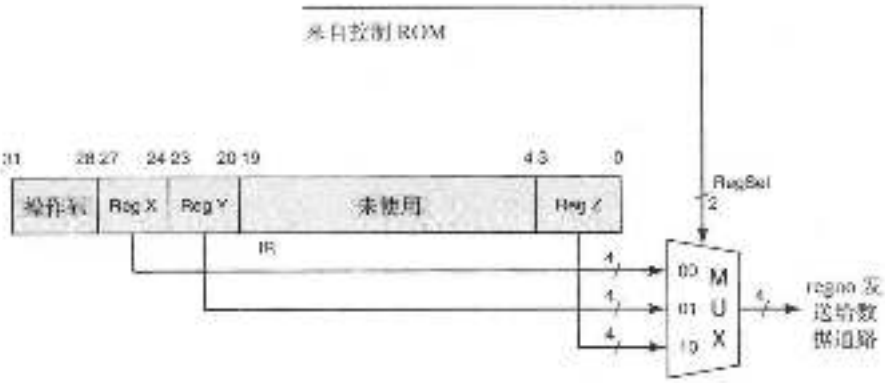


图 3-23 使用 IR 位字段对寄存器进行选择。多路复用器在 IR 中选择需要发送给寄存器堆 regno 地址的专用字段

103

RegSel 控制输入（2 位）来自于微指令。多路复用器的输入是 IR 中指明寄存器的 3 个字段（见第 2 章中 LC-2200 指令格式）。微指令从不直接对寄存器堆寻址。所以，我们在微指令中用 2 位 RegSel 字段来代替 4 位 regno 字段（见图 3-24）。

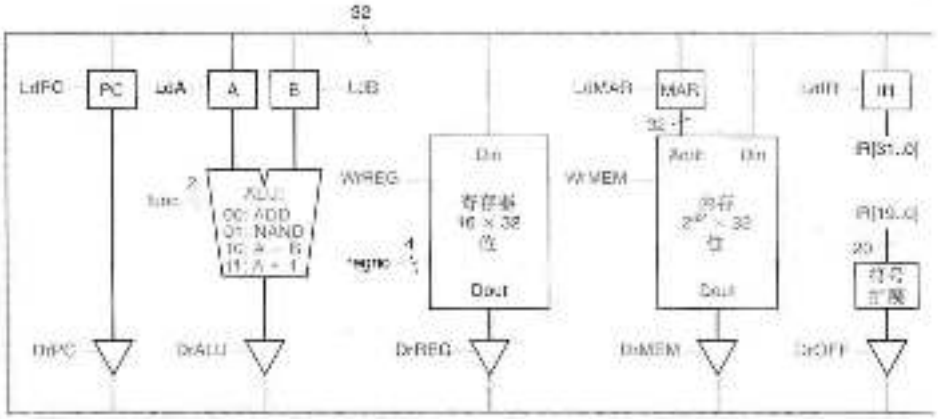
	驱动信号					加载信号						写控号				
当前状态	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	RegSel	下一状态

图 3-24 在 ROM 中添加了 RegSel 字段

现在我们写出实现 ADD 操作宏状态的微状态的数据通路活动和相应的控制信号。

```
• add1
      Ry -> A
      所要的控制信号：
      RegSel = 01
      DrREG;
      LdA
```

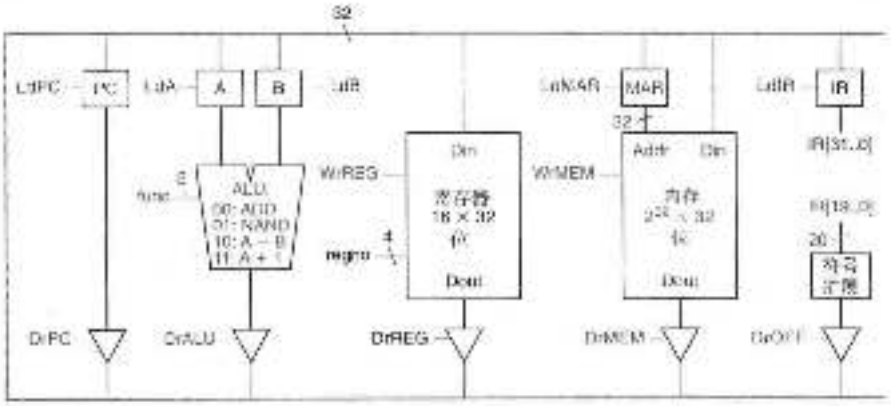
下图显示了 add1 微状态的数据通路活动。



104 注意：寄存器堆的默认操作是将 regno 指定地址的寄存器的内容读出到 Dout 上。

- add2
  Rx → B
  所需的控制信号：
  RegSel = 10
  DrREG
  LdB

下图显示了 add2 微状态的数据通路活动。



- add3
  A + B → Rx
  所需的控制信号：
  func = 00
  DrALU
  RegSel = 00
  WrREG

下图显示了 add3 微状态的数据通路活动。

ADD 宏状态由控制单元通过将 add1、add2、add3 微状态串联起来实现，最终返回到 FETCH 宏状态（见图 3-25）。

105

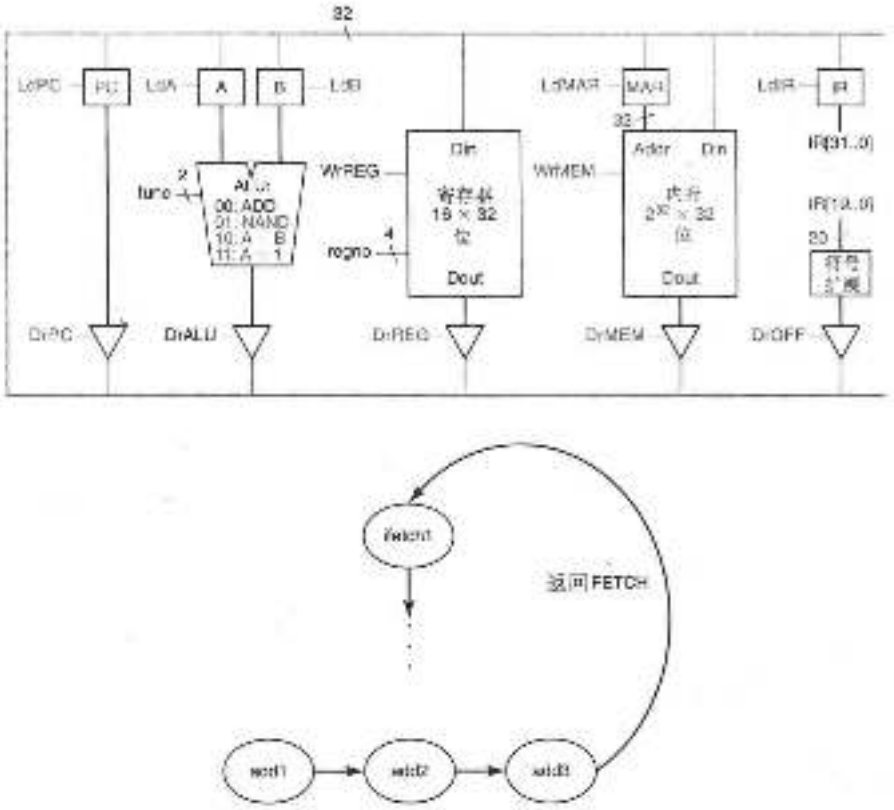


图 3-25 将 ADD 宏状态展开之后的 FSM

3.5.5 EXECUTE 宏状态：NAND 指令（R 型指令部分）

NAND 指令执行如下操作：

$R_X \leftarrow R_Y$ NAND R_Z

NAND 宏状态与 ADD 宏状态的相似之处是，有 nand1、nand2、nand3 三个微状态。与 ADD 相比，NAND 的三个微状态有哪些不同？我们将它作为练习留给读者。

3.5.6 EXECUTE 宏状态：JALR 指令（J 型指令部分）

J 型指令具有如下的格式：



106

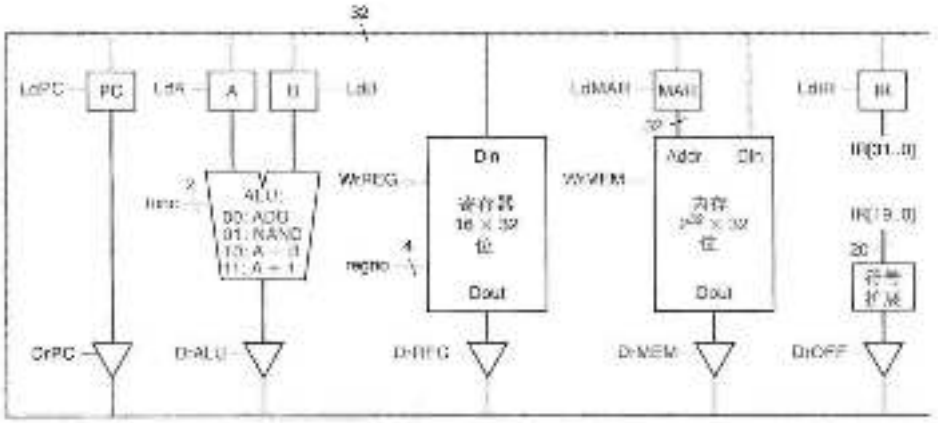
将 JALR 指令引入 LC-2200 指令集中用于支持高级语言的过程调用机制。JALR 将返回地址保存在寄存器中并将控制权交给子过程：

$R_X \leftarrow PC + 1$
$PC \leftarrow R_Y$

下面是 JALR 指令的微状态、数据通路活动和控制信号：

- jalr1
  PC → Ry
  所需的控制信号：
  DrPC
  RegSel = 01
  WrREG

下图显示了 jalr1 微状态的数据通路活动。

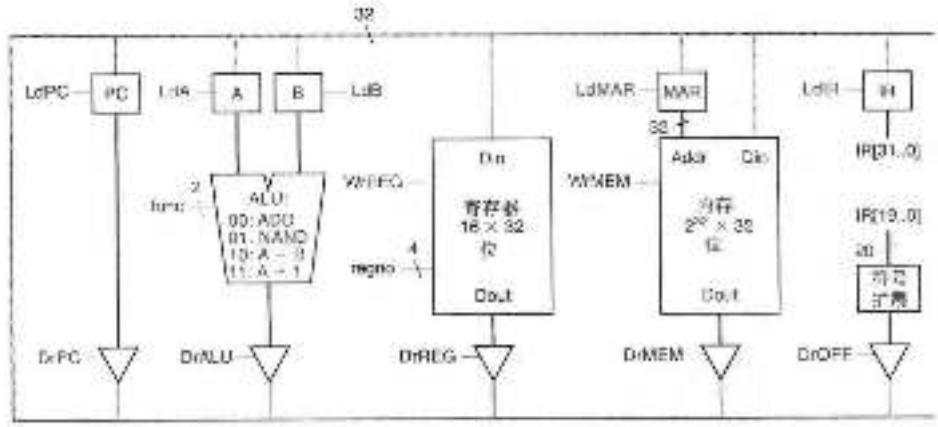


注意：PC+1 需要被保存到 Ry 中。我们在 FETCH 宏状态中已经递增了 PC。

- jalr2
  Rx → PC
  所需的控制信号：
  RegSel = 00
  DrREG
  LdPC

107

下图显示了 jalr2 微状态的数据通路活动。



3.5.7 EXECUTE 宏状态：LW 指令（I 型指令部分）

I 型指令如下格式：



LW 指令的语义是：

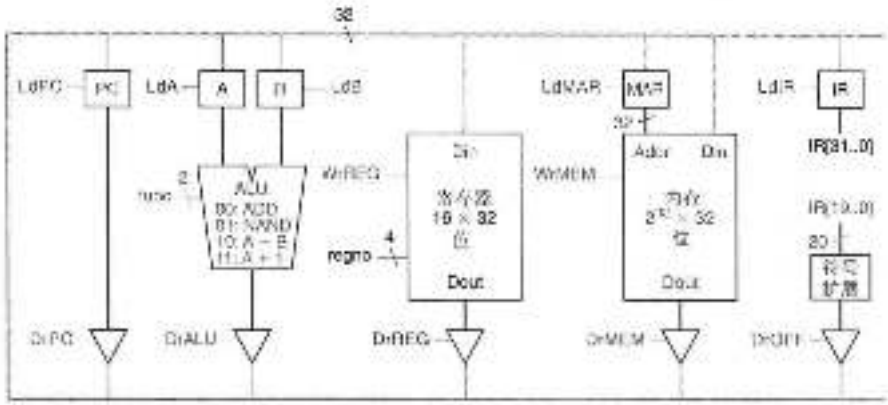
```
Rx ← MEMORY[Ry + aligned address-offset]
```

在 I 型指令中，带符号地址偏移量由指令的一部分作为立即值字段给出。立即值字段占据 IR 的 0 ~ 19 位。从数据通路中可以看出，有一个符号扩展硬件将这 20 位补码值转换为 32 位补码。DrOFF 控制线使得 IR 中的经过符号扩展后的偏移地址可以被放到总线上。

下面是 LW 指令的微状态、数据通路活动和控制信号：

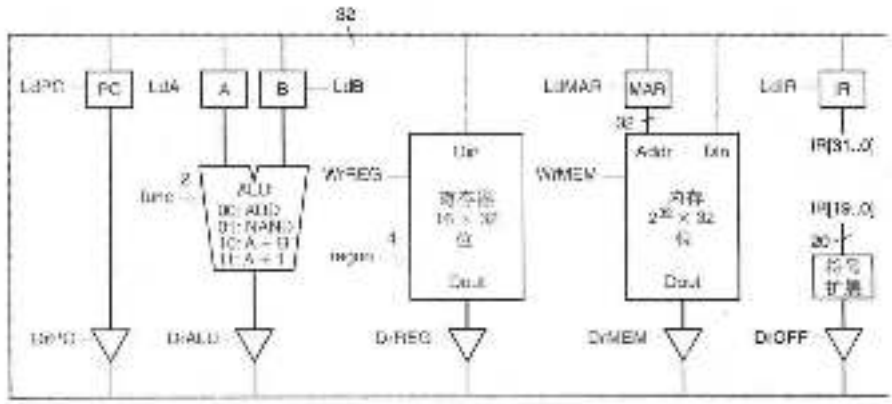
- lw1
  Ry → A
  后续的控制信号：
  RegSel = 01
  - DrREG
  LdA

下图显示了 lw1 微状态的数据通路活动。 108



- lw2
  符号扩展值移置→B
  后续的控制信号：
  DrOFF
  LdB

下图显示了 lw2 微状态的数据通路活动。

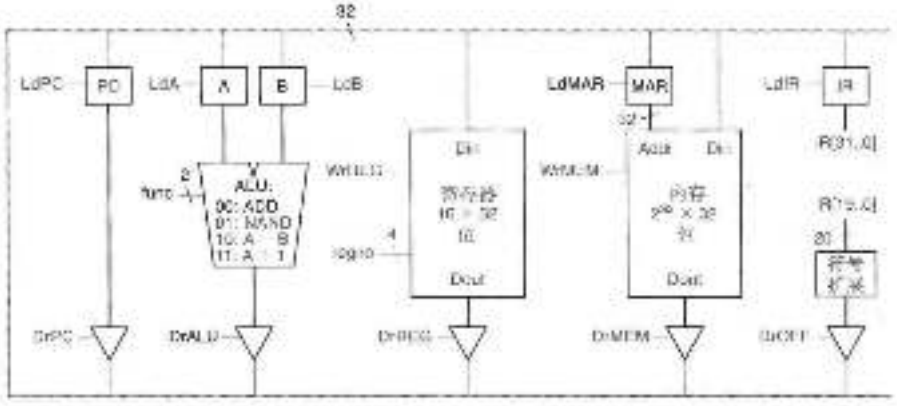


- lw3
  A + B → MAR
  后续的控制信号：

func = 00
DrALU
LdMAR

109

下图显示了 lw3 微状态的数据通路活动。



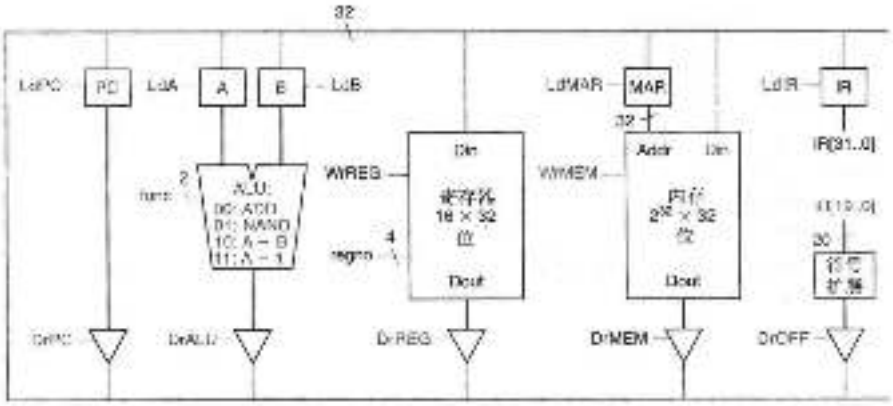
- lw4

MEM[MAR] → Rx

所需的控制信号：

DrMEM
RegSel = 00
WrREG

下图显示了 lw4 微状态的数据通路活动。

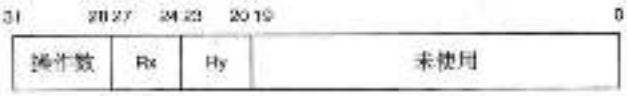


110

例 3-4 我们现在来给 LC-2200 增加一种新的寻址模式，称为**自动递增模式**。该模式对于 LW/SW 指令来说是有用的。采用此模式的 LW 指令语义为：

```
LW Rx, (Ry)+    ;   Rx ← MEM[Ry];
                ;   Ry ← Ry + 1;
```

指令格式如下：



写出使用该寻址模式实现的 LW 指令的序列。（你需要写出该指令的 EXECUTE 状态的序列。）对于每个微状态，给出数据通路活动（寄存器传输形式为 A ← Ry），并生成出关键数据通路所需的控制信

令（可选，DrPC）。

答：

```
IM1:   Ry → A, MAR
       控制信号：
              RegSel=01; DrReg; LdA; LdMAR

IM2:   MEM[Ry] → Rx
       控制信号：
              DrMEM; RegSel=00; WrREG

IM3:   A + 1 → Ry
       控制信号：
              Func=11; DrALU; RegSel=01; WrREG
```

3.5.8 EXECUTE 宏状态：SW 和 ADDI 指令（I 型指令部分）

SW 宏状态的实现与 LW 宏状态类似。ADDI 宏状态的实现与 ADD 宏状态类似，唯一的区别是第二个操作数来自 IR 中的立即值而不是另一个寄存器。这两个宏状态的实现就留给读者作为练习。

3.5.9 EXECUTE 宏状态：BEQ 指令（I 型指令部分）

111

BEQ 指令有如下语义：

```
if (Rx == Ry) then PC ← PC + 1 + 符号扩展偏移量
else 什么都不做
```

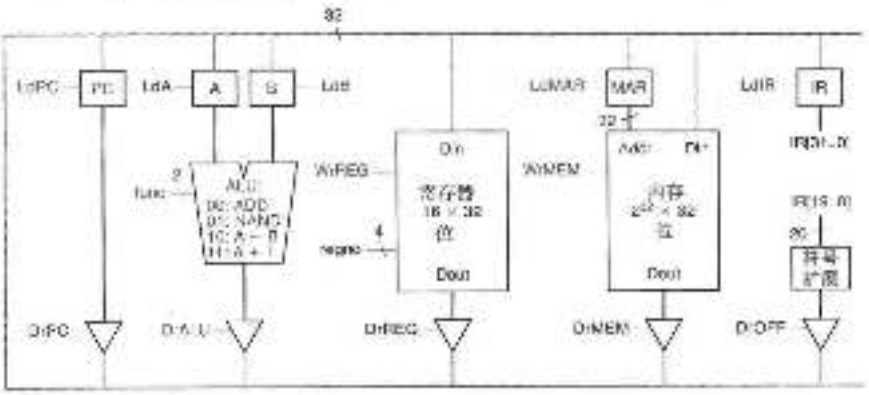
这条指令需要一些特殊处理。该指令的语义是，比较两个寄存器（Rx 和 Ry）中的内容，如果它们相等，则跳转到目的地址，而目的地址是符号扩展偏移量与 PC+1 的和（PC 是这条 BEQ 指令的地址）。

数据通路中，有硬件用于检测总线上的值是否为零。BEQ 的微状态使用这个逻辑根据两个寄存器的比较结果来设置 Z 寄存器的值。

下面是 BEQ 宏状态对应的微状态、数据通路活动以及控制信号：

```
• beq1
      Rx → A
      所需的控制信号：
            RegSel = 00
            DrREG
            LdA
```

下图显示了 beq1 微状态的数据通路活动。



- beq2

  By → B

  所需的控制信号：

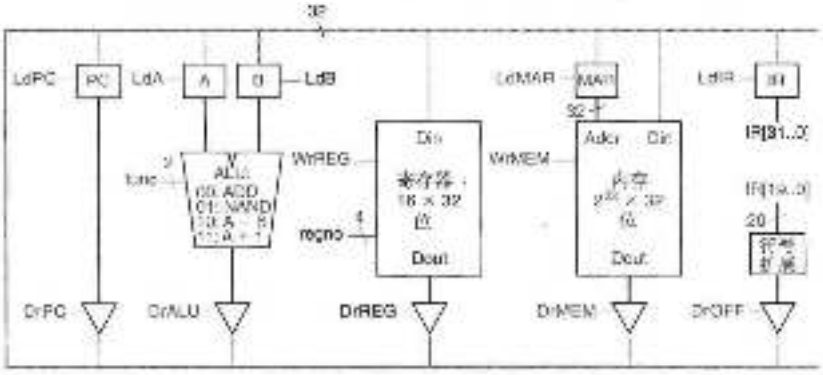
  RegSel = 01

  DrREG

  LdB

112

下图显示了 beq2 微状态下的数据通路活动。



- beq3

  A – B

  将条件表达式的结果放入Z寄存器

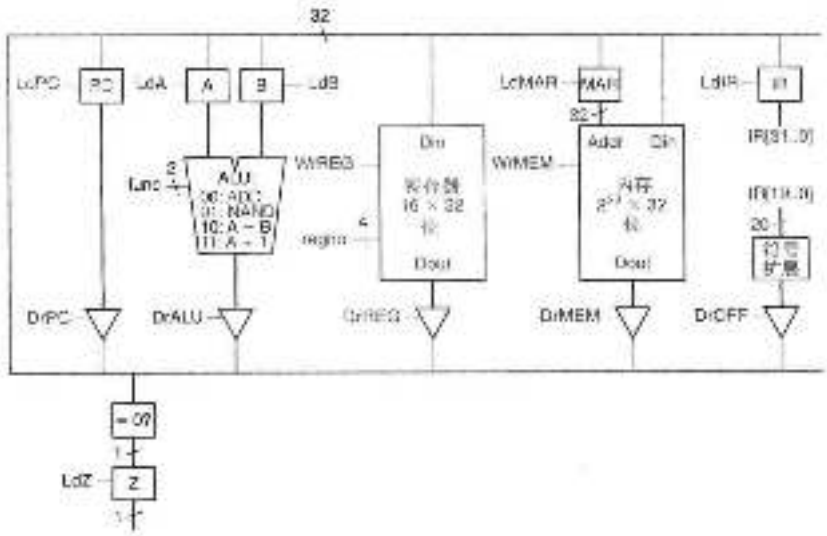
  所需的控制信号：

  func = 10

  DrALU

  LdZ

下图显示了 beq3 微状态下的数据通路活动。



113

注意：数据通路中的零检测逻辑元件一直在检测总线上的值是否为零。通过断言 LdZ，Z 寄存器（1 位寄存器）捕获检测结果以备以后使用。

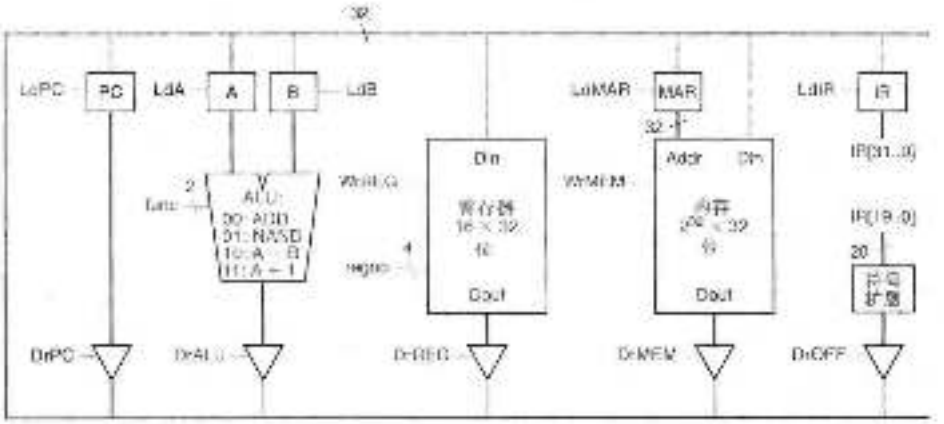
与其他指令的微状态相比，这个微状态的操作更具技巧性。对于其他指令，我们就是简单地按顺序走一遍各个微状态最后返回到 FETCH 宏状态。然而，BEQ 指令根据比较的结

果引起控制流转移。如果 Z 寄存器为 0（即 Rx!=Ry），那么我们就简单地返回到 ifetch1 并继续顺序执行下一条指令（PC 已经指向了该条指令）。另一方面，如果 Z 寄存器为 1，继续用 BEQ 微状态计算分支的目标地址。

首先让我们假设分支发生以完成 BEQ 的各微状态。

- beq4
  PC → A
  所需的控制信号：
  DrPC
  LdA

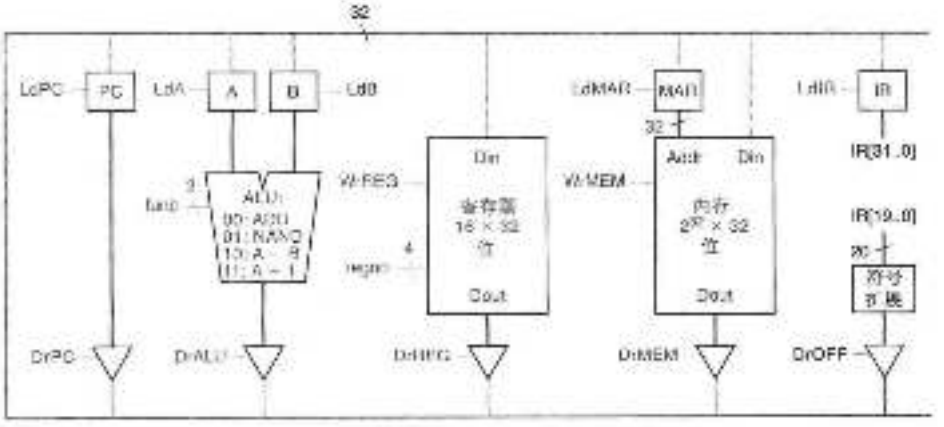
下图显示了 beq4 微状态下的数据通路活动。



- beq5
  偏移量的符号扩展 → B
  所需的控制信号：
  DrOFF
  LdB

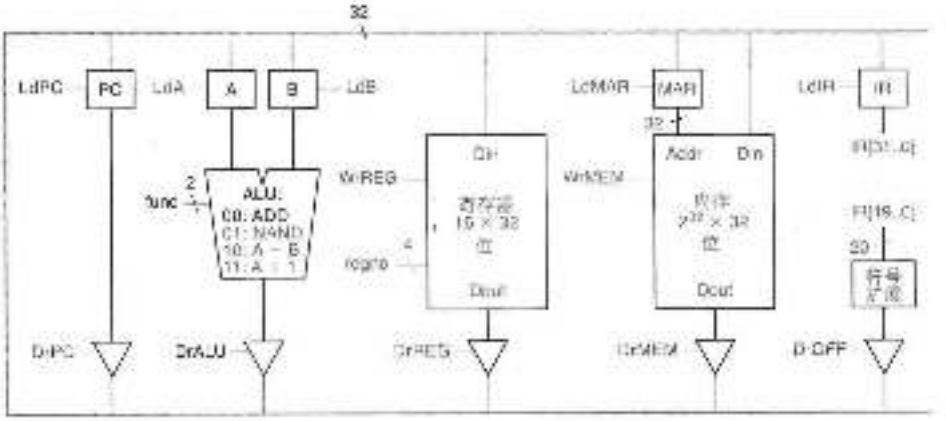
114

下图显示了 beq5 微状态下的数据通路活动。



- beq6
  A + B → PC
  所需的控制信号：
  func = 00
  DrALU
  LdPC

下图显示了 beq6 微状态下的数据通路活动。



注意：在 FETCH 宏状态中，PC 已经递增了，所以这里我们可以将 PC 与符号扩展偏移量相加来计算目标地址。

115

3.5.10 设计微程序中的条件分支

图 3-26 展示了 BEQ 宏状态的状态转移。beq3 微指令的下一状态（next-state）字段将包含 beq4。但是微指令中只有一个下一状态字段，我们需要让 beq3 根据 Z 寄存器的状态转移到 ifetch1 或者 beq4。一种时间上高效的办法是，在 ROM 上的其他单元复制一份与 ifetch1 有关的微指令。下面我们将解释这是如何做到的。

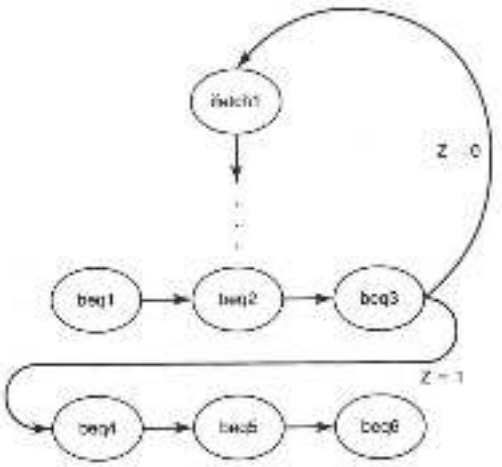


图 3-26 BEQ 宏状态的状态转移。我们可能会转移到 ifetch1 或 beq4，这依赖于 beq3 中计算的结果，这个结果在 beq3 结束前被 Z 寄存器捕获

我们假设状态寄存器有 5 位，beq4 的二进制编码为 Q10000，beq3 微指令的下一状态字段设置为 beq4。我们将 Z 寄存器的内容作为前缀，得到 6 位的 ROM 地址。如果 Z 为 0，则 ROM 地址为 001000；如果 Z 为 1，则地址为 101000。后一个地址（101000）是我们要存储与 beq4 微状态有关的微指令的地方。而 001000（称为 ifetch-clone）则会保存与原始 ifetch1 一样的微指令。图 3-27 显示了这个设计。

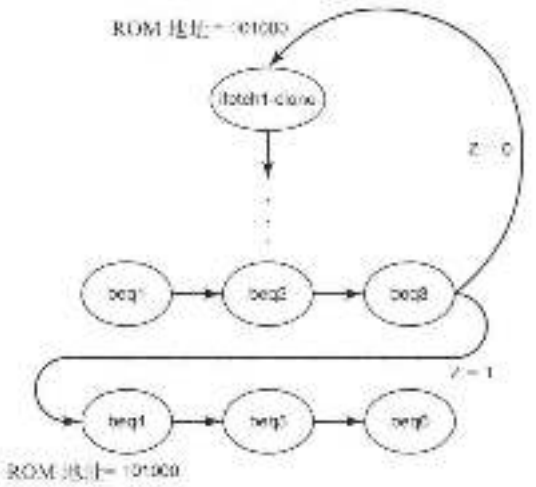


图 3-27 设计条件转分支。使用 5 位基地址（01000）并采用 Z 寄存器的输出作为地址前缀

我们还可以将这个想法（复制微指令）推广到任意需要在微程序中进行条件分支的情况。

3.5.11 再谈 DECODE 宏状态

我们回到 DECODE 宏状态。这是一个从 ifetch3 到 IR 指定的宏状态的一个多路分支。我们使用的技巧与之前根据 Z 寄存器内容为 BEQ 实现 2 路分支时的一样。

我们假设 10000 是通用的 EXECUTE 宏状态编码。

Ifetch3 的下一状态字段包含这个通用值。我们用 OPCODE 的内容（IR 的位 28 – 31）作为通用值的前缀来产生 9 位的 ROM 地址。因此，ADD 宏状态将从 ROM 地址 000010000 开始，NAND 宏状态将从地址 000110000 开始，ADDI 则从 001010000 开始，等等。如图 3-28 所示。

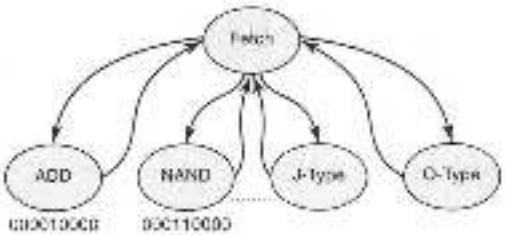


图 3-28 为 DECODE 宏状态产生多路分支。使用 5 位基地址（10000），与 IR 中的 OPCODE（4 位）作为前缀

116
~
117

将所有这些集合到一起，我们发现处理器的控制逻辑具有 10 位地址：

- 最高 4 位是 IR 的 28 – 31 位。
- 下一位是 Z 寄存器的输出。
- 最低 5 位来自 5 位状态寄存器。

最高 4 位（来自 IR 的 28 – 31 位）应当为 0，除非我们想在 FETCH 宏状态的末尾（即 ifetch3 微状态）进行多路选择。类似地，下一位（来自 Z 寄存器的输出）应该为 0，除了在 beq3 微状态下需要做 2 路分支。为了确保能正确地按照我们的选择修改 ROM 地址的高 5 位，

我们在 ROM 的每个表项中添加如外的两个 1 位字段，称为 M 和 T。（见图 3-29。）这些字段分别控制在这个时钟周期中是否要用来自 IR 的 4 位和来自 Z 寄存器的 1 位修改 ROM 地址。如何使用图 3-30 给出的修改控制电路来完成控制单元设计作为练习留给读者（见练习 17）。

	驱动信号					加载信号						写信号				下一状态和修改者		
当前状态	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	RegSel	下一状态	M	T

图 3-29 ROM 表项的最终设置。我们给 ROM 中的每一项添加了 M 和 T "下一状态和修改者"位

图 3-30 中控制单元的输入/输出信号直接与图 3-15 中的对应信号相连。

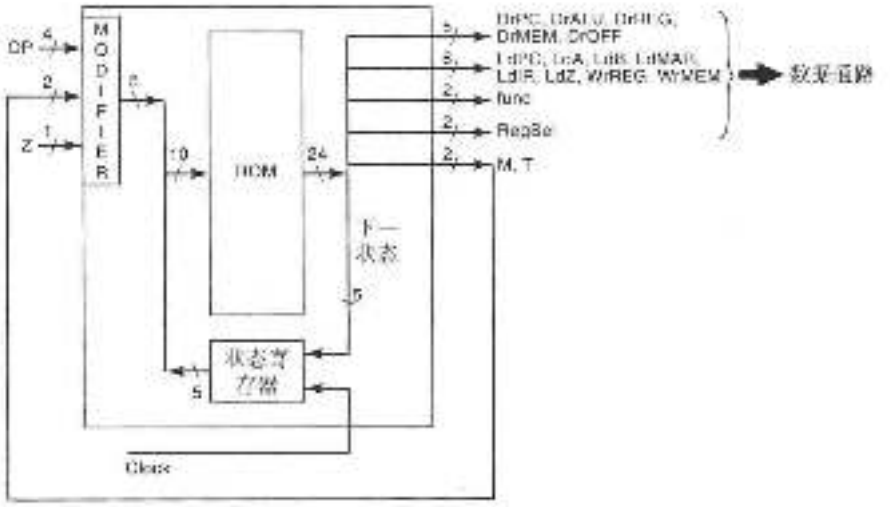


图 3-30 LC-2200 控制单元。这展示了完整的 ROM 寻址机制，以及所有需要产生并发送给数据通路的控制信号

3.6 控制单元设计的另一种选择

我们来考虑处理器控制单元不同风格的实现。

3.6.1 微程序控制

在 3.5 节中，我们给出了微程序风格的控制单元设计。这种风格的确很优雅，也很简单且易于维护。使用微程序设计的好处是，控制逻辑就像 ROM 中的一段程序，因此非常易于维护。但是，微程序设计有两个潜在的低效之处。第一个与时间有关。为了在某个时钟周期中产生控制信号，需要向 ROM 提供一个地址，然后再经过一段称为访问时间的延迟后，控制信号才在数据通路上可用。这个时间代价发生在每个时钟周期的关键路径上，因此成为了性能损失的一个原因。然而，通过预取，即在一条指令执行时提前取出下一条指令，这个时间代价可以被屏蔽。第二个与空间有关。前面小节中给出的设计属于微程序设计中的水平微码风格。在这种设计中整个控制通路中的每个控制信号都与微指令中的一位相对应。从之前讨论的情况来看，对于大多数微指令，这些位中的大部分都是 0，只有少数为 1，与该时钟周

制所需的控制信号吗合。例如，在 ifetch1 中，只有 DrPC、LdMAR、LdA 是 1，微指令中的其他位都是 0。水平微码在空间上的低效可以通过采用一种称为垂直微码的技术来克服，而垂直微码与写汇编相类似。不同微指令中同一位的位置表示了不同的控制信号，这又依赖于每条垂直微指令中的操作码（OPCODE）字段。垂直编码更具技巧性，因为各条微指令中同一位的位置表示的控制信号需要互斥。

3.6.2 硬连线控制

我们仔细研究 3.5 节中设计的使用水平微码的 ROM 究竟表示了什么是有益的。它实际上是一个**真值表**，给出了数据通路所需要的所有控制信号。ROM 的行表示状态，列表示的是函数（每一列对应一个控制信号）。基于前面对逻辑设计的介绍，读者应该知道如何合成每列所需要的最小布尔逻辑。这些逻辑函数比真值表更有效。

我们可以使用组合逻辑电路来实现控制信号对应的布尔逻辑函数。函数的功能就是在某个状态下给出的某些控制信号。

例如，DrPC 的布尔函数可能是：

```
DrPC = ifetch1 + jalr1 + beq4 + ...
```

使用 AND/OR 门，或者像 NAND/NOR 这样的通用门，我们可以生成所有的控制信号。我们将这种类型的设计称为**硬连线控制**，因为控制信号都是使用组合逻辑电路来实现的（不容易修改，所以称为硬连线）。这样的设计在时间和空间上都很有效（不需要 ROM 查找的访问时间，也不需要为时钟周期内没有产生的控制信号浪费空间）。过去有一些反对者认为，因为使用随机逻辑来实现布尔函数，所以这将导致噩梦般的维护工作。

然而，**可编程逻辑阵列**（PLA）和**现场可编程门阵列**（FPGA）的出现使得这些反对者自无力。例如，PLA 将所需要的逻辑组织成二维阵列结构，从而让随机逻辑有了结构。我们在图 3-31 中给出了这种结构。

118 / 119

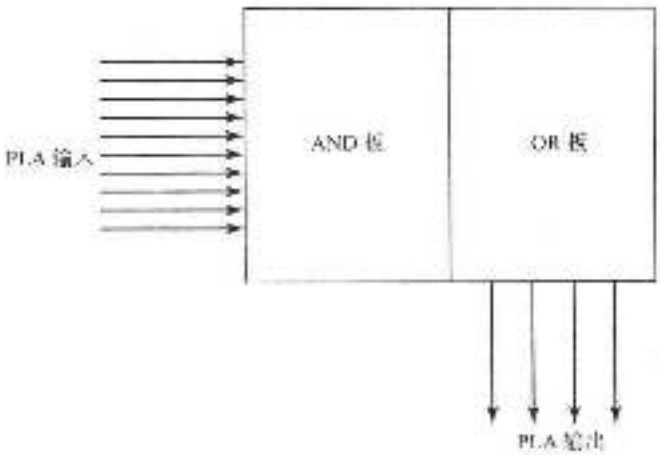


图 3-31 PLA。左边的输入来自状态寄存器和影响处理器（如 IR 和 Z 寄存器）状态的数据通路元件。每个输出是输入的 SUM-of-PRODUCT 的结果，这些输出就是数据通路所需的控制信号

输出是驱动数据通路所需的控制信号（DrPC、DrALU 等）。输入是处理器所处的状态

（ifetch1、ifetch2等）和数据通路中产生的条件（IR和Z寄存器的内容等）。AND板中的每个门都获得所有的输入（在正确而完整的版本中是这样的）。类似地，OR板中的每个门也都获得AND板产生的所有结果。PLA之所以称为PLA，是因为它上面的逻辑可以通过选择那些输入传送给AND板和OR板而“编程”。每个PLA输出都是**逻辑乘的逻辑加**（SUM-of-PRODUCT）的结果。**逻辑乘**（PRODUCT）是在AND板中实现的。逻辑加（ADD）是在OR板中实现的。这种设计将所有控制逻辑都集中在一起，就有了微程序设计的结构优势。同时，因为控制信号使用组合逻辑电路产生，所以它没有硬连线设计的不足。与随机逻辑的设计相比，它在空间的使用上稍微有些劣势，因为每个AND门的扇入数都与PLA的总输入数相同，而每个OR门的扇入数都与AND板中AND门的数量相同。无论如何，PLA的结构化优点和规则性远远超过了它的劣势，VLSI设计已经离不开它。

最近，FPGA变得非常流行，它用来为复杂的硬件设计迅速制造原型。FPGA其实是PLA的继承者，它含有逻辑元件和存储元件。这些元件之间的连接可以“现场”进行编程，FPGA正是由此得名。这样的灵活性使得设计上的缺陷更容易修正，甚至在部署之后依然如此。因此也提高了硬连线设计的可维护性。

3.6.3 在两种控制设计风格中选择

在这两种风格之间进行选择受到许多因素的影响。我们已经给出了两种控制风格的优势和劣势。也许由于FPGA的出现，许多针对硬连线控制可维护性的争论已经停止了。尽管如此，对于处理器的基本实现（即非流水线），或者对于复杂指令的实现（如x86体系结构中的那些），更倾向于使用微程序控制，因为它很灵活也很容易快速修改。另一方面，我们将在流水线处理器的有关章节中看到，高端流水线处理器的实现非常适合使用硬连线控制。表3-2总结了这两种设计思路的优劣。

表3-2 控制方法比较

控制方法	优势	劣势	备注	什么时候使用	例子
微程序	简单、易维护、灵活、快速开发	可能在时间和空间上都比较低效	在采用垂直风格时空间上的低效会得到改善，使用预取可以缓解时间上的低效	用于复杂指令和非流水体系结构的快速原型开发	PDP-11系列、IBM 360/370系列、Motorola 68000、Intel x86体系结构中的复杂指令
硬连线	适合于流水线实现，可达到更高性能的潜力	可能难以修改设计，设计时间更长	可以通过使用结构化元件（如PLA和FPGA）来改善可维护性	用于高性能流水线体系结构实现	大多数现代处理器，包括Intel Xeon系列、IBM PowerPC

120 ~ 121

小结

在本章中，我们感受了如何为给定的指令集设计处理器。第一步的实现是选择数据通路并将各个元件组装到数据通路上。我们在3.3节回顾了基本数字逻辑元件，在3.4节回顾了数据通路设计。当数据通路确定后，我们将注意力转移到驱动数据通路完成指令集的控制单元的设计。基于微程序控制单元的假设，3.5节大致浏览了实现LC-2200 ISA中各指令所需的微状态。在3.6节，我们评价了硬连线控制与微程序控制的区别。

历史回顾

我们现在来看看，历史上性能的改善如何受到经济与技术条件的影响。

在20世纪40年代和20世纪50年代，构建硬件和存储器所需的逻辑单元极为昂贵。当时采用的技术是电子管，后来是单独的晶体管来实现硬件。指令集体系结构都非常简单，一般都有一个单独的寄存器，称为累加器。那个时代的计算机以EDSAC和IBM 701为代表。

在20世纪60年代，开始出现少许的"集成"。1965年，推出的IBM 1130采用固态逻辑技术（Solid Logic Technology，SLT），这是后来集成电路的先驱。那个年代硬件价格出现大幅下降，但存储器依然使用磁芯（称为磁芯存储器），在计算机系统的成本中占了一大部分。

这种趋势持续发展，到了20世纪70年代，出现了SSI（小规模集成）、还有MSI（中规模集成）电路作为实现处理器的技术。半导体存储器在20世纪70年代开始取代磁芯存储器的地位。有趣的是，大约在1974年，磁芯存储器和半导体存储器每位存储的价格几乎是一样的（每位0.01美元）。从那时到现在，半导体存储器的价格一直迅速下降。

20世纪70年代经历了许多不同的体系结构，例如面向栈的、面向内存的、面向寄存器的等各种体系结构。那个时代机器的代表有IBM 360和DEC PDP-11，这些是面向内存和面向寄存器相结合的体系结构。还有面向栈的Burroughs B-5000。这些机器都属于存储程序计算机，即通常所说的以计算机先锋John von Neumann命名的冯·诺依曼体系结构。

在存储程序计算机发展的同时，计算机科学家还实验了非常新颖的体系结构，包括数据流和收缩系统。这些体系结构的目的在于让程序员编制多条指令并行执行，打破存储程序模型固有的指令串行执行方式。数据流系统和收缩系统都着眼于数项而非指令。数据流系统允许任何输入已经就绪的指令得到执行，并将执行结果传送给其他指令。收缩系统允许并行数据流通过功能元件的阵列（这个阵列经过预先的安排以完成某种特定功能）完成对数据流的计算（例如，矩阵乘法）。数据流体系结构是通用计算模型的一个实现，而收缩系统则是合成体系结构（synthesizing architectures）中的一种具体算法模型。虽然这些类型的体系结构并没有取代存储程序计算机，但它们对计算机整体，从算法设计到处理器实现，都产生了巨大的影响。

20世纪80年代，出现了一些有趣的发展。首先，使用双极晶体管的高密LSI（大规模集成）电路已很常见。这成了许多高端处理器采用的实现技术，如IBM 370和DEC VAX 780。另一个趋势是，使用CMOS晶体管（也称为场效应管或FET）的VLSI（超大规模集成）电路开始用于单芯片的微处理器。20世纪80年代末及90年代初，这些单芯片处理器在性价比上对高端机器市场产生了威胁。20世纪80年代，编译器技术也迅速发展，形成了实现上的系统软件（如编译器）与指令集设计之间的合作关系。这种关系为RISC（精简指令集计算机）体系结构铺平了道路。IBM 801、Berkeley RISC和Stanford MIPS号角了这场RISC革命。有趣的是，CISC（复杂指令集计算机）体系结构的追随者也很多，例如Motorola 68000和Intel x86系列处理器。因为新的集成技术能够将更多的晶体管放到一块硅片上，所以直到20世纪80年代还只存在于高端处理器上的流水线处理器设计（见第5章）开始进军微处理器。关于RISC和CISC的争论逐渐减少，问题变成了如何在一个流水处理器中达到每个时钟周期一条指令的吞吐量。

在20世纪90年代，微芯片（使用CMOS技术的单芯片微处理器）最终确立了它作为计算机工业界处理器实现技术的地位。20世纪90年代末，Intel x86和Power PC指令集（这两种指令集都属于采用了部分RISC体系结构实现技术的CISC体系结构处理器系列）成为制造桌面计

122

算机、服务器、超级计算机的工业标准。值得一提的是，那个年代最有希望的体系结构是 DEC Alpha。不幸的是，由于 DEC 在 20 世纪 90 年代后期死去，Alpha 体系结构也就安息了。

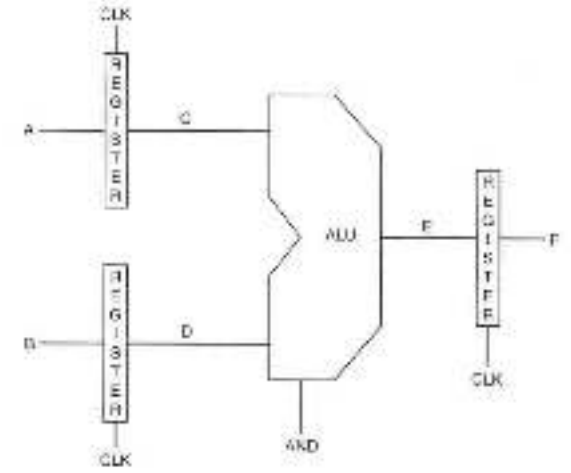
随着个人通信设备（手机、寻呼机、PDA）及游戏设备的出现，嵌入式计算平台从 20 世纪 80 年代开始越来越重要。大部分的嵌入式平台采用 RISC 体系结构的 ARM（Acorn RISC Machine 的缩写，最初由 Acorn Computer 设计）的各种型号。Intel 制造了从最初 ARM 体系结构衍生出的 XScale 处理器。值得一提的是，ARM 处理器一开始是打算为 PC 和工作站设计的。

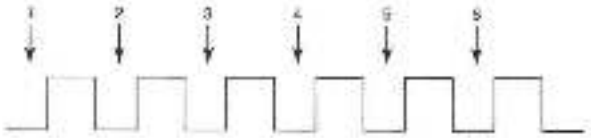
[123] 超标量和超长指令字体系结构（VLIW）处理器采纳了 RISC 革命中产生的技术。两者都是为了提高处理器的吞吐量。它们通常称为多发射处理器。超标量处理器依靠硬件来并行执行固定数量的不相互依赖的相邻指令。对于 VLIW，顾名思义，一条指令实际上包含了多个捆绑在一起的操作。VLIW 严重依赖于编译技术来降低硬件设计的复杂性并为其开发并行性。从 20 世纪 80 年代后期第一次提出以来，VLIW 技术在这些年间已经重新改进了许多次。VLIW 体系结构最近的一次改进是 Intel 的 IA-64，针对的是超级计算市场。VLIW 体系结构在高端嵌入式领域也很受欢迎，如 DSP（数字信号处理）应用。

我们已经迎来了新千年的第一个 10 年的结束。如今关于指令集的争论很少。大部分工作都位于微体系结构的层面，即，如何通过各种硬件技术来提高处理器的性能。同时，集成度的提高也允许将多个处理器放在单块芯片上。多核芯片开始出现在大部分我们购买的计算机系统上。

练习题

1. 电平逻辑与边沿触发逻辑的区别是什么？我们在实现 ISA 时用的是哪种？为什么？
2. 给出一个车库门开关控制器的 FSM 和状态转移表（见图 3-12 和表 3-1），用时序逻辑电路实现这个控制器。（提示：这个时序逻辑电路包含 2 个状态，产生 3 种输出，即下一状态、开门信号、关门信号。）
3. 使用本章介绍的 ROM 加状态寄存器方式重新实现练习 2 中的逻辑电路。
4. 对比控制逻辑的不同设计方法。
5. 对基于 ROM 的控制方式，一种减少空间需求的方法是将各个独立的控制信号用 ROM 中的一个编码字段表示。这种方法的优点和缺点是什么？哪些控制信号能聚集在一起，而哪些不能？给出你的理由。
6. 基于总线的数据通路设计有哪些优点和缺点？
7. 考虑一个三总线设计。你如何使用它来组织图 3-10 中的双总线数据通路上的那些元件？与双总线设计相比，它有什么好处？
8. 解释为什么内部寄存器，如指令寄存器（IR）和内存地址寄存器（MAR），不应由控制单元在 ISA 的
[124] 实现中当一数值的临时存储器。
9. 一位工程师希望将实现 FETCH 宏状态的微状态数量减少到 2 个。请问她该如何完成这个目标呢？
10. 定长指令的优势是什么？
11. 给出下图的数据通路。假设所有的线都是 16 位宽。填写下面的表格。
12. 在 LC-2200 处理器中，为什么在 ALU 后面没有寄存器？
13. 扩展 LC-2200 ISA 以使包含一条减法指令。给出该减法指令所需的微状态。假设数据通路如图 3-15 所示。
14. 在图 3-15 的数据通路中，为什么我们需要 ALU 前面的寄存器 A 和 B？为什么我们需要 MAR？什么情况下你不需要它们？（提示：考虑寄存器堆和总线上的额外端口。）
15. 芯存储器曾经是每位 0.01 美元。考虑你的计算机，按照这个价格，内存需要花多少钱？如果现在的内存还是这个价，会对计算机产业造成什么影响？
16. 如果计算机设计者只关注速度而忽略价格因素，那么如今的计算机产业将是怎样？谁会是消费者？如果仅仅考虑价格呢，又会怎样？





Time	A	B	C	D	E	F
1	0x42	0xFE	0	0	0	0
2	B	C				
3	0xC4FE	Ca1				
4	0	0				
5	0	0				
6	0	0				

17. 在 3.5.11 节中，我们引入了两个额外的字段，M 和 T，作为下一状态修改位。这两位允许我们在给定时钟周期中有选择地修改 ROM 地址的高位。使用这些下一状态修改位来完成图 3-30 中的控制单元逻辑。

18.（设计题）

考虑一个使用面向栈的指令集的 CPU。算术指令的操作数和结果都放在栈上。这种体系结构中没有通用寄存器。

下图所示的数据通路使用了两块分离的内存：一块 65 536（2^{16}）字节的内存用于保存指令和（非栈上的）数据，一块 256 字节的内存用来保存栈。栈使用常规内存和一个栈指针寄存器实现。栈从地址 0 开始，压入数据后在上增长（即往高地址增长）。栈指针指向栈顶的元素（当栈为空时，栈指针为 -1）。你可以忽略栈下溢或下溢的问题。

程序 / 数据内存使用的内存地址都是 16 位。所有的数据都是 8 位宽。假设程序 / 数据内存为字节可寻址，即每个地址对应一个 8 位的字节。每条指令包含 8 位的操作码。许多指令还包括一个 16 位的地址字段。指令集在下面给出。这里，“内存”指的是程序 / 数据内存（而不是栈内存）。

操作码	指令	操作
00000000	PUSH <addr>	将内存地址 <addr> 中的内容压入栈中
00000001	POP <addr>	将栈顶的元素弹出并存入内存地址 <addr> 中
00000010	ADD	将栈顶的两个元素弹出，并将它们相加之和压入栈中
00000100	BEQ <addr>	将栈顶的两个元素弹出，如果它们相等，则转移到内存地址 <addr>

125

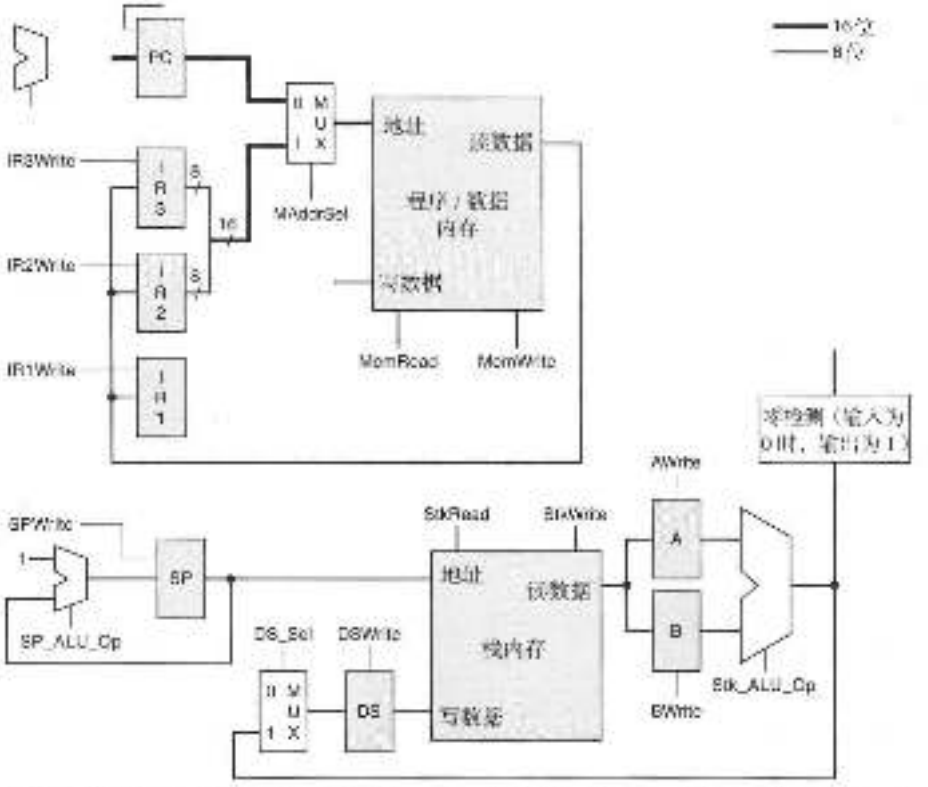
126

注意 ADD 指令只有 8 位，但其他指令都是 24 位。指令在内存中必须连续存放的（即不假设所有指令都是 24 位长）。

假设内存宽度为 8 位，即每次对内存的读/写操作都是对其中的 8 位指令或数据进行访问。这意味着取一条多字节的指令需要多次内存访问。

(a) 数据通路

补全下面的设计。



127

假设读或写程序/数据内存或栈内存都需要一个时钟周期来完成（实际上，应该更短一些，因为要留出时间来读/写寄存器）。同样，假设每个 ALU 算术运算需要的时间至少下一个时钟周期。零检测电路需要的时间可以忽略。

(b) 控制单元

给出控制单元的状态表，指出表中每个状态必须给出的控制信号。

参考文献注释和扩展阅读

逻辑设计基础方面有不少好书[Katz，2004；Mano，2007]。Patt 和 Patel[Patt，2004]的教材从基本逻辑设计开始，接着是机器语言编程，最终以介绍 C 语言结束。讲述计算系组成与设计的教材有很多[Patterson，2008；Stallings，2010；Tanenbaum，2005；Hamacher，2001]。Maurice Wilkes 是一位计算机先驱，他成绩很好，拿过很多次第一名。1951 年，他是组织处理器控制单元的微程序技术的发明者[Wilkes，1951]。该技术被工业界广泛用于实现复杂的计算机结构。Wilkes 获得过很多奖项，包括 1967 年 ACM

128 图灵奖、1980 年 Eckert-Mauchly 奖，其中后者是由 IEEE Computer Society 和 ACM 颁发的。

中断、陷入及异常

在前一章中，我们讨论了处理器的实现。本章，我们将讨论如何让处理器能够处理程序执行中的不连续性。在某种意义上，分支指令和过程调用也是一种不连续。然而，这些不连续都是程序员有意加入程序中的。我们在本章探讨的不连续性指的是那些无法预知的，甚至不属于程序本身的不连续性。

我们来看一个简单的比喻。在一个教室里，教授在讲授计算机体系结构。为了增强互动，他希望同学提问题。他有两种选择：1）每隔一段时间，他停下来询问学生是否有问题要问；2）他告诉学生，只要有问题，随时可以举手提问。显然，第一种方式抑制了学生的自发性。当教授让学生提问时，学生可能已经忘记问题是什么了。更坏的情况是，他们有许多问题，一个接一个的，导致现在已经完全跟不上课程了！第二种方式保证了学生及时得到回答，不会打断思路。但还有个小问题。教授应该什么时候让学生提问呢？他可以在学生举手时立即响应，但可能当前的话才说了一半。所以，他应该先说完现在的话，再接受学生提问。教授需要记住自己讲到哪里了，这样在回答问题后还能接着继续讲。如果在教授回答某个问题时，又有一个学生举手，又该怎么办？这样下去教授很快就会崩溃。所以，一条基本原则是，在教授回答完某个学生的提问前，不接受其他学生的提问。我们从这个教室的比喻中提炼出两点：记住课程进度；禁止进一步的提问。 129



我们设计的处理器能够执行指令，但如果它不能与外界进行I/O通信，那就没有用。基于前面的比喻，我们可以让处理器周期性地询问输入设备（如键盘）。一方面，轮询是易出错的，因为设备产生数据的速率可能要比轮询的速率更快。另一方面，如果设备并没有产生数据，那么处理器轮询设备就是浪费时间。所以，我们借鉴教室中的方法，让设备中断处理器，让处理器知道设备有话对它说。与教室类似，处理器需要记住当前程序执行到什么地方，且禁止其他中断直到处理完当前中断。

4.1 程序执行中的不连续性

在第3章中，我们定义了描述逻辑电路行为的术语：同步和异步。考虑一个现实生活中的例子。假设你走到冰箱前并拿出一瓶苏打水，这是同步事件，这是你计划中的一部分。另一方面，当你在房间里写作业时，你的室友突然闯进来给了你一瓶苏打水，这就是异步事件了，因为这并不在你的计划中。打电话是同步事件，而接电话则是异步事件。

我们将同步和异步的定义推广到系统的硬件和软件上。同步事件就是在规定好的时间发生的事件（如果真的发生了），属于系统计划中的活动。第3章讨论的各个微状态之间的转移都是硬件系统中同步事件的例子。同样，在程序中打开一个文件是软件的同步事件。

130

异步事件就是与系统正在进行的其他活动相关的在不可预料的时间发生的事件（如果真的发生了）。我们很快就会看到，中断是异步硬件事件。你在写作业时收到的新邮件提醒是异步软件事件。

系统可能既有同步事件又有异步事件。例如，硬件采用轮询方式（同步）来检测事件，并产生异步软件事件。

现在我们已经准备好讨论程序执行中的不连续性了，这包含3种形式：中断、异常和陷入。

1. 中断

中断是设备引起处理器注意的机制。这对于执行中的程序来说，是一种计划外的不连续性，与处理器的执行是异步的。而且，设备I/O可能会执行一个与当前正在执行程序完全不同的程序。为了简单起见，我们认为中断特指由外部设备引起的不连续性。

2. 异常

程序有时候会不小心执行一些非法指令（比如，除以零）或者执行路径与预想的不同。这些情况下，必须打断原有指令序列的执行，处理这个计划外的不连续性。这种情况称为异常。异常是内部产生的情况，而且与处理器执行是同步的。异常一般由当前程序引起，通常是执行中某些错误情况的结果。然而，诸如Java这样的编程语言定义了异常机制，允许错误在软件的各层中传递。在这种情况下，程序可以有意地产生异常作为非预期程序行为的信号。在这两种情况下，我们定义异常为一些脱离程序正常执行轨迹的情况（无论是故意还是无意的）。

3. 陷入

程序通常通过系统调用来读/写文件或请求其他系统服务。系统调用类似于过程调用，但需要专门的处理，因为用户程序会访问系统的某些部分，这些部分不仅与当前程序有关，还关系到系统的所有用户。而且，用户并不知道与此服务相关的过程位于内存何处但在调用时必须知道这个信息。陷入，顾名思义，允许程序往下再进操作系统中，然后由操作系统来确定用户程序做什么。在计算机著作中还常用术语软件中断来指代程序产生的陷入。为了我们讨论的目的，我们认为软件中断与陷入等价。与异常类似，陷入是内部产生的状况，并且是与处理器执行同步的。有的陷入是有意的——例如，程序显式地进行的系统调用。我们关心的是程序，从这个角度来说，有些陷入是无意的。我们在后面讨论存储系统的章节中会看到这类无意陷入的例子。

许多著作中对术语中断的表述相当标准一致，但是对另外两个术语就不是这样了。我们对这3个术语的定义和使用在全书中保持一致，但不保证与其他书一样。具体来说，在我们的定义中，陷入指的是当前运行的应用程序自己无法处理的一种内部情况，只能由操作系统处理。另一方面，异常的处理由当前运行的程序负责。

表 4-1 总结了这 3 种程序不连续性的特征。第二列表示它们是同步还是异步，第三列给出了不连续性来自于当前运行程序的外部还是内部，第四列表示不连续性的来源是否故意产生了它，最后一列给出了这种类型的例子。 [131]

表 4-1 程序不连续性

类型	同步/异步	源	有意	例子
异常	同步	内部	是或不是	溢出、除以零、非法内存地址、Java 的异常机制
陷入	同步	内部	总是不是	系统调用、软件中断、页错误、监视指令
中断	异步	外部	是	I/O 设备完成

4.2 处理程序不连续性

事实证明，程序的不连续性是一种有力的工具。第一，它允许计算机系统提供输入/输出功能。第二，它允许计算机系统为相互竞争的活动管理资源。第三，它允许计算机系统帮助程序员开发正确的程序。中断、陷入和异常分别提供了上述功能。

处理程序不连续性是处理器体系结构和操作系统之间的一种合作。我们先来理解这个分工。检测程序的不连续性是处理器的责任。引导处理器去执行处理不连续性的代码则是操作系统的责任。正如我们将在本书中看到的那样，各个子系统都需要这种硬件和软件的合作来支撑。

现在我们看出处理程序不连续性要做什么。准确地说，就是硬件隐式完成的活动和操作系统显式完成的活动。

事实证明，无论处理哪种不连续性，处理器要做的事情基本上都是一样的。需要强调的是， [132]
处理器只执行指令。为了处理这些程序的不连续性，处理器需要执行的指令并不是当前正在执行的。出现不连续时执行的过程就称为处理过程。处理过程的代码和其他过程的代码非常相像。在这个意义上，不连续性非常像过程调用。（见图 4-1）然而，它是计划外的过程调用（取决于不连续性的性质）。此外，我们也必须观察这种计划外过程调用的约定（过程调用惯例）以及它如何恢复正常的程序执行。其中大部分都很简单，就与普通的过程调用返回一样。

不连续性有 4 点很较脐的地方：

1）它们可以在指令执行的任意位置发生。因此，不连续性可能发生在指令执行的中间。

2）不连续性是计划外的，并且与当前程序是无关联。所以，硬件在将控制权交给处理过程之前，需要保存程序计数器的值。

3）在检测到不连续性时，硬件需要确定处理过程的地址以便将控制权交给它。

4）因为硬件隐式地保存 PC 值，所以处理过程需要懂得如何恢复之前的程序执行。

操作系统与处理器体系结构之间的合作使得前面 4 个问题的解决成为可能。这个合作的基础在于一个数据结构，它由操作系统维护，保存在内存中的某处，处理器也知道这个数据结构。这个数据结构是一个固定大小的处理过程地址表。表中的每一项对对应于一种预期的程序不连续

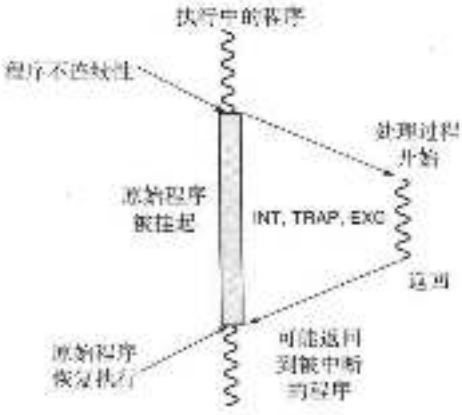


图 4-1 程序不连续性。程序不连续性与普通的过程调用之间有相同点也有不同点 [133]

性。（见图 4-2。）表的大小与体系结构有关。历史上，这个数据结构的名字叫做中断向量表（IVT）[⊖]。每种不连续性都有一个唯一的编号，通常称为矢量。这个编号用作 IVT 的唯一索引。操作系统在启动时建立这张表。这是为处理程序不连续性做准备的最后部分。表建好后，处理器在执行普通程序时若检测到不连续性，就用这张表来查找特定处理过程的地址。

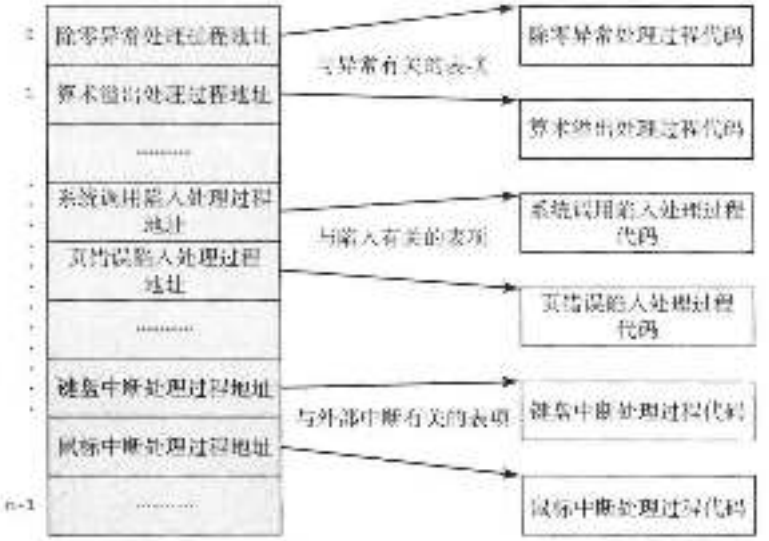


图 4-2 中断向量表（IVT）——在启动时 OS 建立这张表。这保证了当程序发生不连续时，硬件能够通过查询这张表确定应该去“哪里”

对于陷入和异常的情况，硬件内部产生它们的矢量。我们引入异常/陷入寄存器（ETR），它位于处理器内部，用于保存这个矢量（见图 4-3）。遇到异常或陷入时，与该异常或陷入相关的唯一编号就会放入 ETR 中。例如，当检测到“除以零”这个异常时，除法指令的 FSM 就会将该异常对应的矢量放到 ETR 中。类似地，系统调用由处理器支持的“陷入指令”产生。在这种情况下，陷入指令的 FSM 将对应系统调用的矢量放入 ETR 中。

134



图 4-3 异常/陷入寄存器——由处理器在检测到异常或陷入时设置，用于在 IVT 中索引处理过程地址

总之，操作系统与硬件之间为处理程序不连续性而进行的合作如下：

1）体系结构定义一系列异常和它们对应的编号（矢量值）。这些通常是程序运行中的运行时错误（比如，算术溢出和除以零）。

2）操作系统定义一系列异常（软件中断）和陷入（系统调用）以及它们的编号（矢量值）。

3）在启动时操作系统建立 IVT，给出处理各类异常、陷入、中断的处理过程地址。

4）在普通程序执行时，硬件检测异常和陷入，并将它们的矢量值填入 ETR。

5）在普通程序执行时，硬件检测外部中断并接收中断设备对应的矢量值。

6）硬件使用矢量值来索引 IVT，找到处理过程，将控制权从当前执行的程序转交给处理

⊖ 不同厂商对该数据结构的命名不同，Intel 将之称为中断描述符表（IDT），有 256 个表项。

过程。

对于外部中断，处理器需要做额外的工作才能确定与中断设备相对应的矢量，以便调用恰当的设备处理程序。4.3 节讨论为了处理程序不连续性而对处理器体系结构和指令集设计进行的改进。

4.3 处理程序不连续性的体系结构改进

首先我们理解为了处理程序的不连续性，需要哪些体系结构上的改进。因为对于各种类型的不连续性，处理器的处理机制是一样的，所以今后我们就用中断来指代不连续性。

1）处理器什么时候应该处理中断？这正是教室比喻中的问题。我们需要让处理器在进入处理过程之前是一个干净的状态。即使中断发生在某条指令执行的中间，处理器也应该等这条指令执行完成后才检查中断。

2）处理器如何发现有中断？我们将在处理器数据通路的总线中加入硬件线路。在每条指令执行完毕后，处理器在这条线上采样检查是否有中断等待处理。 135

3）我们如何保存返回地址？我们如何创建处理过程的地址？对于所有指令，如果在指令执行末尾出现了中断，则该指令的 FSM 进入一个特殊的宏状态，INT 宏状态。

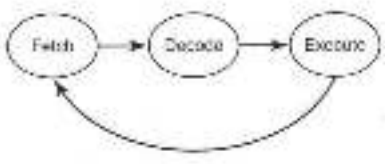
4）我们如何处理多个级联的中断？我们将在 4.3.3 节讨论这个问题。

5）我们如何从中断中返回？我们将在 4.3.4 节中给出这个问题的思路。

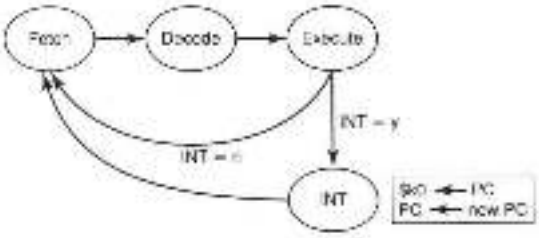
4.3.1 修改 FSM

在第 3 章中，我们讨论了实现处理器的基本 FSM 包含 3 个宏状态——取指（fetch）、解码（decode）、执行（execute），如图 4-4a 所示。

图 4-4b 显示了修改后的 FSM，它多了一个新的宏状态，用于处理中断。正如 4.2 节提到的，中断可能在当前指令执行的任意时刻产生。现在的 FSM 在一条指令结束时检查是否有中断。如果有中断（INT=y），则 FSM 转移到 INT 宏状态；如果没有中断（INT=n），则返回到 Fetch 宏状态，开始执行下一条指令。有一种可能是在每个宏状态后进行检查。这类比于教室中的教授先完成了他在思考才意识到有学生有问题。然而，在每个宏状态后检查中断是有问题的。在第 3 章中，我们看到处理器数据通路中包含多个在处理器指令集体系结构层次不可见的内部寄存器。我们知道，当指令执行完后，这些内部寄存器的值就无用了。因此，将中断检查推迟到当前指令完成后能使处理器处于干净的状态。为了让被中断的程序能够在处理完中断后恢复执行，有两个东西是必需的：程序可见寄存器的状态和程序恢复点。



a）处理器的基本 FSM



b）为处理中断而修改的 FSM 在发生中断，则处理器在当前指令完成后进入中断状态

图 4-4 FSM

例4-1 考虑下面的程序：

```
100     ADD
101     NAND
102     LW
103     NAND
104     RFQ
```

当处理器执行 ADD 指令时发生了一个中断。为了使能中断后恢复程序，需要记下的 PC 值是多少？

答：

虽然中断是在 ADD 指令执行过程中产生的，但它会在指令**执行完毕后**才被处理。所以为了使能中断后恢复程序，需要记下的 PC 值为 101。

现在我们讨论在 INT 宏状态中需要做什么。为了让讨论更具体，我们对 LC-2200 处理器进行改进以便处理中断。

1）我们需要将当前 PC 值保存在某处。我们为此保留一个寄存器 $k0（寄存器堆中的 12 号通用寄存器）。INT 宏状态将 PC 存入 $k0。

2）我们从设备接收到处理过程的地址，将其装入 PC，然后转移到 Fetch 宏状态。我们很快就会详细阐述完成这步的细节。

4.3.2 一个简单的中断处理过程

136 | 137

图 4-5 展示了一个简单的中断处理过程。保存和恢复处理器的寄存器与第 2 章中讨论的过程调用约定极为类似。

```
Handler:
    save processor registers;
    execute device code;
    restore processor registers;
    return to original program;
```

图 4-5 一个简单的中断处理过程

例4-2 考虑下面的程序：

```
100     ADD
101     NAND
102     LW
103     NAND
104     RFQ
```

当处理器执行 ADD 指令时发生了一个中断。此时，程序使用的寄存器只有 R2、R3 和 R4。中断处理器过程保存和恢复为寄存器是哪些？

答：

事不幸，因为中断可能在任意时刻发生，所以中断处理过程无法知道程序现在使用了什么寄存器。所以，它保守地恢复所有程序可见的寄存器。即使该程序仅仅需要保存或恢复 R2、R3 和 R4。

4.3.3 处理级联中断

图 4-5 给出的简单处理过程代码有一个问题。如果在处理当前中断时又发生了另一个中

断，那么我们就会失去原始程序的 PC 值（现在在 $k0 中）。这会导致无法返回到引起第一个中断的原始程序中。图 4-6 描述了这种情况。

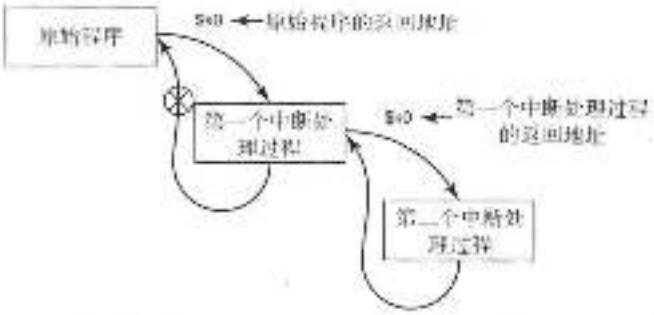


图 4-6 级联中断，因为返回地址保存在 $k0 中，所以嵌套中断会丢失引起第一个中断的原始程序的返回地址

在进入第二个中断处理过程时，我们就丢失了原始程序的返回地址。这种情况就好像教授在回答完第一个问题前又要回答第二个问题。在这个比喻中，我们简单地禁止学生在第一个问题回答完之前再次提问。但也许需要很好地回答第二个问题才能帮助大家理解原始问题。所以不接受多个中断对于计算机系统来说是不可行的。各个设备的速度迥异，比如硬盘的数据速率就要比键盘或鼠标高许多。所以，我们不能在处理一个中断时关闭其他中断。同时，每个处理过程都应要有一段无嵌套中断的时间，这样才能采取措施避免出现图 4-6 的情况。

所以，为了处理级联中断，需要两个东西：

1）关闭中断的新指令，称为**禁止中断**（disable interrupt）。

2）打开中断的新指令，称为**允许中断**（enable interrupt）。

而且，硬件需要在 INT 宏状态中隐式地关闭中断并将控制权交给处理过程。图 4-7 展示了修改后的 FSM，其中禁止中断已添加到 INT 宏状态上。

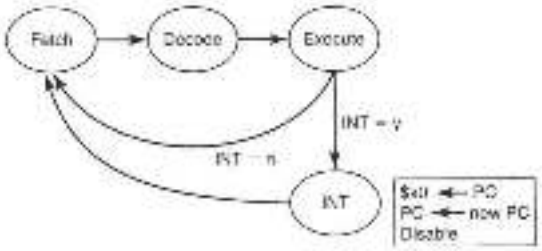


图 4-7 添加了关闭中断指令的修改后的 FSM。硬件在将控制权交给中断处理程序之前先关闭中断

138
:
139

现在让我们来研究处理过程应该做什么才能避免出现图 4-6 中的情况。处理过程将原始程序的返回地址保存在 $k0 中，此时中断是关闭的。这一步完成之后，它就打开中断，保证处理器不会错过更重要的中断。在离开处理过程之前，它恢复 $k0，此时中断也是关闭的。图 4-8 展示了修改后的中断处理过程。如第 2 章讨论的过程调用惯例，保存和恢复寄存器都使用栈。

```
Handler:
  /* 在进入处理过程时，中断是关闭的 */
  save $k0;
  enable interrupts;
  save processor registers;
  execute device code;
  restore processor registers;
  disable interrupts;
  restore $k0;
  return to original program;
```

图 4-8 修改后的中断处理过程。处理过程在返回地址保存好后，立即使用新指令显式地允许中断

例 4-3 考虑下面的程序：

```
100     ADD
101     NAND
102     IW
103     NAND
104     BEQ
```

在处理器执行 ADD 指令时发生了一个中断。处理该中断的处理过程代码如下：

```
1000    Save $k0
1001    Enable interrupts
1002    /* next several instructions save processor registers */
....
1020    /* next several instructions execute device code */
....
1102    /* next several instructions restore processor registers */
....
1120    restore $k0
112     return to original program
```

140

假设在指令“restore $k0”（PC=1120）处产生了一个中断，那么原始程序什么时候恢复呢？

答：

原始程序永远都不会恢复。注意第二个中断在“restore $k0”完成后马上被处理，此时处理过程有 $k0=101，这正是原始程序的恢复点。现在出现了第二个中断。不幸的是，第二个中断（见图 4-7）将第一个中断处理过程的恢复点（内存地址为 1121）保存到 $k0 中。因此，原始程序的恢复点（内存地址为 101）就永远丢失了。克服原因是或使中的中断处理过程设定图 4-8 中的关键的“禁止中断”指令。

需要注意的是，并不是所有情况在处理第一个中断时都需要这么谨慎地处理第二个中断。例如，在 4.4.1 节中我们将介绍多层中断的概念。根据它们的相对速度，设备会有不同的中断优先级。例如，与键盘之类的低速设备相比，硬盘这样的高速设备就有更高的优先级。当处理器在处理来自硬盘的中断时，它会暂时忽略来自键盘的中断。

硬件的角色是为处理器正确处理级联中断提供必要的机制。处理过程代码（它是操作系统的一部分）与处理器硬件的合作关系决定了如何最好地处理多个同时发生的中断，这依赖于这个时间点处理器正在做什么。

基本上，我们的选择包括两部分：

- 在一段时间内忽略中断（当操作系统正在处理一个更高优先级的中断时）。

• 正如本小节描述的那样，立刻处理新的中断。

暂时忽略中断可以是硬件优先级保证的隐式操作，也可以是处理过程通过“禁止中断”指令进行的显式操作。

4.3.4 从处理过程中返回

当处理过程完成后，它能使用存储在 Sk0 中的 PC 值返回到原始程序。乍一看，为了支持从中断返回，需要提供过程调用机制。例如，在第 2 章中，为了从过程中返回，我们引入了下面的指令[⊖]： [141]

```
J   Rtrn
```

自然地，我们企图使用相同的指令从中断返回：

```
J   $k0
```

然而，这有一个问题。我们在返回原始程序时中断应该是打开的。所以，我们应该考虑使用下面的指令序列从中断返回：

```
Xenable interrupts;
J   $k0;
```

使用上面的指令序列依然存在问题。我们在每条指令结束后应该检查中断。所以，在“Enable Interrupts”和“J $k0”两条指令的中间，可能会发生新的中断而废弃 Sk0 的值。

因此，我们引入下面的新指令：

```
Return from interrupt (RETI)
```

这条指令的语义是：

```
Load PC from $k0;
Enable interrupts;
```

需要特别注意的一点是，这条指令是原子的。也就是说，在该指令执行完之前不会有新的中断产生。有了这条新指令，图 4-9 给出了正确的能够处理嵌套中断的处理过程。

```
Handler:
  /* 进入处理过程时，中断是关闭着的 */
  save $k0;
  enable memops
  save processor registers
  execute device code;
  restore processor registers
  disable interrupts;
  restore $k0;
  return from interrupt;
  /* 从中断返回会打开中断 */
```

图 4-9 完整的中断处理过程 [142]

4.3.5 检查点

总之，我们为了使 LC-2200 能够处理中断，我们对体系结构做了如下改进。

1）LC-2200 中新增了下面三条指令：

⊖ LC-2200 没有单独的无条件跳转指令。同比在 LC-2200 中我们使用 JALR 来模拟它。

```
Enable interrupts
Disable interrupts
Return from interrupt
```

2）当中断发生时，将当前 PC 值隐式地存储到专用寄存器 $k0 中。

现在我们转向讨论支持这些体系结构改进所需要的硬件。我们已经给出了宏状态层上 FSM 的改进细节。我们给读者留一个练习，修改 LC-2200 数据通路和 INT 宏状态的细节问题。为了完成这个练习，读者需要识别完成 INT 宏状态需要哪些微状态，并给出各微状态下的控制信号。

4.4 处理程序不连续性的硬件细节

正如我们在 4.2 节中提到的那样，目前讨论的体系结构改进都与具体的不连续性类型（即异常、陷入、中断）无关。在本节中，我们将从整体上讨论处理程序不连续性的硬件细节，并具体讨论外部中断处理。我们已经介绍了中断向量表（IVT）和异常/陷入寄存器（ETR）。我们现在研究为了能够从外部设备接收中断向量，数据通路需要做哪些改进。我们有意保持简单的讨论，而实际上现代处理器中的中断体系结构是非常复杂的，我们在本章小结中会简单介绍。

4.4.1 中断的数据通路细节

我们将讨论处理中断的实现细节。在第 3 章中，我们介绍并讨论了连接数据通路各元件的总线这一概念。我们来扩展这个概念，这对于理解数据通路为处理中断进行的扩展是非常必要的。特别地，我们设想这样一条总线，它把处理器和内存、外部 I/O 设备连接起来。为了让处理器和内存通信，我们需要地址线和数据线。图 4-10 中的数据通路在总线上添加了额外的线路来支持中断。总线上有一条标记为 INT 的线。所有希望中断 CPU 的设备都断言该线。在第 3 章中，我们强调了保证任意时刻至多只有一个元件访问共享总线的重要性。另一方面，INT 线却不一样。可以有任意数量的设备同时断言该线，表明它们想和处理器通话（就像教室里有许多学生同时举手一样）。为了能够让多个设备同时断言 INT 线，这里采用了一种
143 称为线或（wired-OR）的技术。（该技术的细节已经超出了本书的范围，感兴趣的读者可以参考关于逻辑设计的教材（如 [Katz，2004；Mano，2007；Patt2004]）以了解更多细节。）

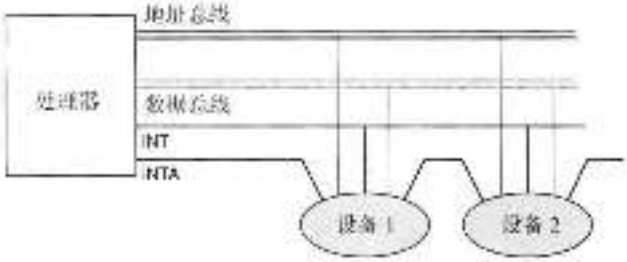


图 4-10 针对中断处理的数据通路改进。所有的设备通过“线或”方式连接到 INT 线上。从处理器发出的 INTA 线依次穿过所有设备（称为菊链）。如果多个设备同时中断处理器，则在电气上最接近处理器的中断设备优先处理。被选中的设备将“向量”放在数据总线上

在中断发生时，处理器对 INTA 线断言（在 INT 宏状态中）。尽管可能有多个设备着眼中断寄存器，但只能有一个设备得到确认。注意 INTA 线，它并不是共享线（像 INT 线那样），而是从一个设备到另一个设备的链，通常称为菊链。电气上距离最近的设备（图 4-10 中的设

备 1）最先得到 INTA 信号。如果它有中断请求，则它知道处理器已经准备好和它进行通话了。如果它没有中断请求，则它知道有其他设备正在等待与处理器通话，于是它将 INTA 信号通过链传递下去。菊链的优点是简单，但却需要忍受确认信号传播给中断设备的延迟，对于如今如此快速的处理器更是如此。由于这个原因，现代处理器并不使用这种方法。

这种设计原理的一种推广是允许总线上出现多条 INT 线和 INTA 线。每对不同的 INT 线和 INTA 线对应着一个优先级。图 4-11 表示了一个 8 层优先级的中断机制。需要注意的是，在这种设计中，某一时刻依然只能有一个设备能够与处理器对话（处理器的 INTA 线连接到有中断请求的设备中优先级最高的设备的 INTA 线）。设备优先级与设备速度有关。设备的速率越快，丢失数据的可能性越高，因此也越需要得到迅速的处理。因此，设备根据它们的优先级排布在中断线上。例如，硬盘的优先级比键盘高。然而，虽然有多个中断等级，但如果每个设备都需要专用的中断线，那也无法容纳所有的设备。另外，根据定义，高优先级的设备 144
比低优先级的设备重要。然而，有些设备的速率相近，因此处于同一优先级。比如，键盘和鼠标。因此，依然有必要将多个设备放在同一优先级上，如图 4-11 所示。

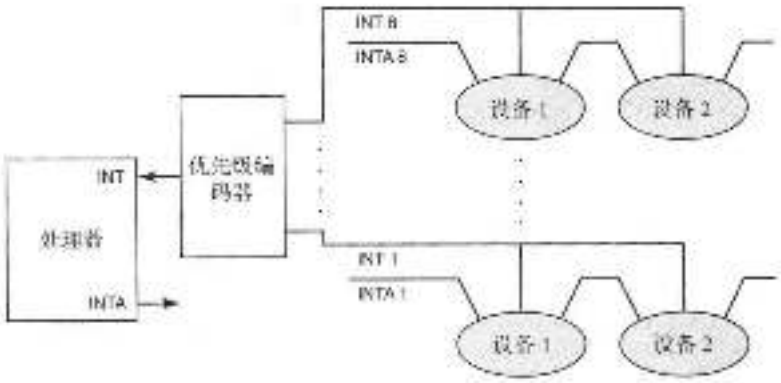


图 4-11 优先级中断。为了满足不同速率设备的需要，现代计算机系统有多个优先级。根据设备的速率将它们排布在不同的优先级上。每个优先级内部会有多个设备

正如前面所说，由于信号传播延迟太大，菊链设计并不令人满意。另外，处理器是珍贵的资源，如今的努力方向是将不必要的工作从处理器中转移出去，并支持外部的胶合逻辑。所以，在现代处理器中，决定哪个设备获得中断响应这个工作转移给了一个外部逻辑，称为中断控制器。中断控制器收集中断请求并选择优先级最高的那个报告给处理器，它还处理与设备间基本的信号交换。操作系统将一个优先级内的中断服务例程用链表链接起来，而不是在硬件上在采用菊链方式连接。现在处理一个中断需要遍历链表以确定第一个有中断请求的设备并进行处理。

你可能想知道这一切与你如何将设备（比如，记忆棒或耳机）插入笔记本之间有什么关系。实际上没有什么神奇的。设备的位置（以及它的优先级）已经预先由系统的 I/O 体系结构决定了（见图 4-11）。你在外部看见的就是一些可以插入设备的插槽。我们将在第 10 章中进一步讨论计算机总线，这与输入/输出有关。 145

4.4.2 获得处理过程地址的细节

我们现在来看看处理器如何从外部设备取得中断向量。正如 4.2 节中提到的那样，操作

系统在启动时建立的中断向量表（IVT）包含了所有外部中断的处理过程地址。虽然设备不知道它的处理过程代码在内存中的何处，但它知道表项中包含它。例如，键盘知道它的中断向量为 80，鼠标知道它的中断向量为 82[⊖]。在收到处理器发送的 INTA 信号时（见图 4-12），设备将它的向量放到数据总线上。需要强调的是，处理器此时还处在 INT 宏状态。处理器用这个向量作为索引到向量表中查找处理过程的地址，然后将地址装入 PC。操作系统为向量表保留一块内存区域（通常是低地址内存）。向量表的大小是操作系统设计时的一个选择。因此，通过中断向量表，处理器可以间接得到处理相应中断的处理过程代码在内存中的地址。

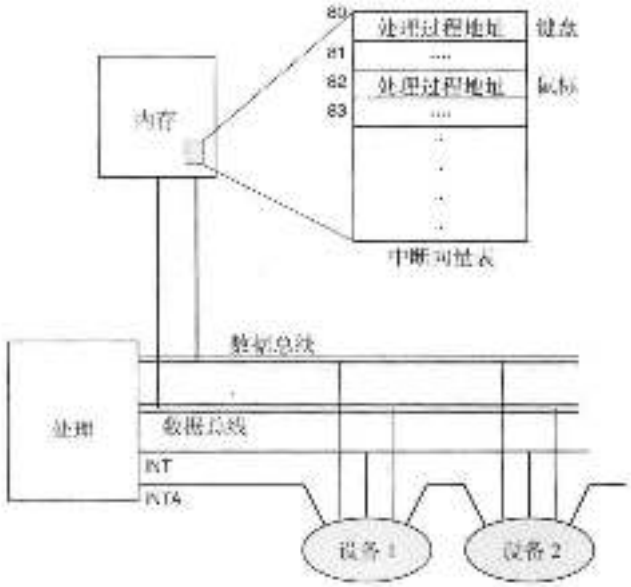


图 4-12 处理器与设备的连接及获取中断向量。处理器从送中的设备得到中断向量，并用它在内存中的 IVT 进行索引以得到相应处理过程代码的起始地址

处理器和设备之间的信号交换总结如下：

1）无论何时，如果设备想中断处理器，则对 INT 线断言。

2）处理器在完成当前指令（EXECUTE 宏状态（见图 4-4b））后，检查 INT 线（图 4-4b 显示 FSM 中为 INT-y/n）是否有中断请求。

3）如果有中断请求，即图 4-4b 的 FSM 中的 INT-y，则处理器进入 INT 宏状态并将总线上的 INTA 线断言。

4）设备在收到处理器发送的 INTA 信号后，把自己的向量放到数据总线上（比如，键盘将它的向量 80 放上去）。

5）处理器收到向量并在中断向量表中查找对应的表项。这里我们假设找到的处理过程地址为 0x5000。这就是处理当前中断所需要的处理过程的 PC 值。

6）处理器在 INT 宏状态中完成上面的动作后，如图 4-4b 所示，将 PC 保存在 $k0 中，并将中断向量表中取出的值填入 PC。

⊖ 操作系统通常为每个设备确定一张表项，该表项会被“编写”到设备接口中。我们在讨论输入/输出的第 10 章还会回到这个问题。

4.4.3 保存/恢复栈

图 4-9 是处理过程的代码，它包括保存和恢复寄存器的代码（与过程调用范例类似）。栈会起来很明显就是用来保存处理器寄存器的地方。然而有一个问题，处理过程怎么知道要用内存的那一部分作为栈呢？中断可能与当前运行的程序根本没关系。

鉴于这一点，常常可以见到体系结构拥有两个栈：用户栈和系统栈。通常，体系结构会指定某个寄存器作为栈指针。在进入 INT 宏状态时，FSM 执行以下操作：

现在我们来看看为了帮助完成栈切换，硬件需要做哪些改进：

1）**复制栈指针**：的确，我们需要做的就是复制体系结构指定作为栈指针的寄存器。在第 2 章中，我们指派一个寄存器 Ssp 作为栈指针。我们将复制这个寄存器：一个给用户程序使用，另一个给系统使用。在中断处理过程中保存的状态将使用系统版的 $sp。这样保存状态就不会弄乱用户栈，因为所有的保存和恢复都发生在系统栈上。在系统启动时，系统初始化系统版 Ssp 为系统栈的地址，申请足够的空间来处理中断（包括嵌套中断）。

2）**特权模式**：要记住中断处理过程其实也就是程序。我们需要让处理器知道在某个时刻应该使用哪个版本的 Ssp。由于这个原因，我们在处理器中引入一个模式位。处理器根据该位的值处于用户模式或内核模式。如果处理器处于用户模式，那么硬件就隐式地使用用户版本的 Ssp。如果处于内核模式，则使用内核版的 $sp。FSM 在 INT 宏状态中设置该位的值。因此，处理过程将运行在内核模式并使用系统栈。在返回用户程序前，RETI 指令将模式位设置为“用户”使得用户程序在恢复执行后能够使用用户栈。

模式位还有一个重要作用。我们引入了 3 条新指令来支持中断。允许所有的程序执行这三条指令是不谨慎的。例如，寄存器 $k0 有特殊意义，不应该允许任意程序写这个寄存器。类似地，也不应该允许任意程序打开或关闭中断。只有操作系统能够执行这些所谓的特权指令。我们需要一种方法来防止普通用户程序意外地或恶意地尝试执行这些特权指令。中断处理过程是操作系统的一部分，运行在“内核”模式。（FSM 在 INT 宏状态中将模式位设置为“内核”。）如果用户程序试图执行这些指令，将会引发一个非法指令陷入（illegal instruction trap）。

我们需要指出另外两个精妙的细节来完成对于中断处理的讨论。首先，中断是可以嵌套的。因为所有中断处理过程都运行在内核模式，只有当处理器从用户程序进入中断处理过程时，我们才需要在 INT 宏状态中进行模式切换（以及模式位引起的隐式栈切换）。而且，我们需要记住处理器当前的模式以便在执行 RETI 指令时采取恰当的操作（需要返回用户模式还是内核模式）。系统栈是一个记住处理器当前状态的方便工具。

```
INT macro state:
  $k0 ← PC;
  ACK INT by asserting INTA
  Receive interrupt vector from device on the data bus
  Retrieve address of the handler from the interrupt vector table;
  PC ← handler address retrieved from the vector table;
  Save current mode on the system stack;
  mode ← kernel /* 如果已经是内核模式了就什么都不做 */
  Disable interrupts;
```

图 4-13 INT 宏状态中的行为

INT 宏状态和 RETI 指令分别将处理器当前模式压入栈中或从栈中弹出。INT 宏状态和 RETI 指令采取与处理器当前状态相应的行动。图 4-13 总结了 INT 宏状态中的所有操作，图 4-14 总结了 RETI 指令的语义。

```
RETI:
  Load PC from $k0;
  /* 因为 RETI 是中断处理过程执行的，所以此时我们处于内核模式 */
  Restore mode from the system stack; /* 回到之前的模式 */
  Enable interrupts;
```

图 4-14 RETI 指令的语义

146 ~ 147

4.5 信息汇总

4.5.1 体系结构和硬件改进总结

为了处理程序不连续性，我们对 LC-2200 进行了如下体系结构和硬件的改进：

- 一个中断向量表（IVT），由操作系统初始化为各处理过程的地址。
- 一个异常/陷入寄存器（ETR），含有内部产生的异常和陷入的向量。
- 接收外部中断向量的硬件机制。
- 用户/内核模式和与之相关的处理器模式位。
- 与模式位有关的用户/内核栈。
- 一个硬件机制，用于在中断时隐式地将当前 PC 保存到特殊寄存器 Sk0 中，并利用向量（无论是内部产生的还是从外部设备接收的）从 IVT 中检索处理过程地址。
- 加入 LC-2200 的 3 条新指令：

148
/
149

```
Enable interrupts
Disable interrupts
Return from interrupt
```

4.5.2 工作中的中断机制

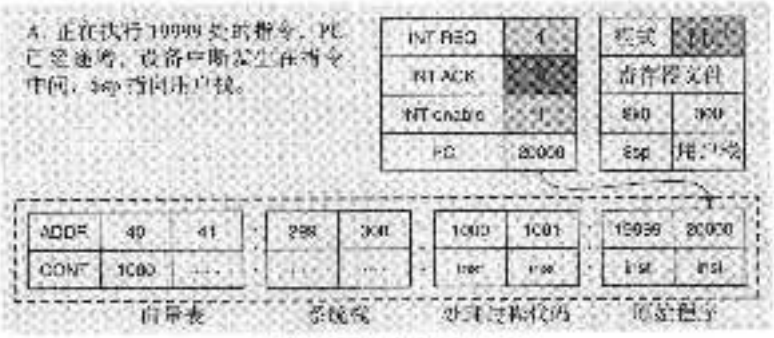
我们通过演示几个简单的例子来将这些概念综合到一起，让读者理解中断机制是如何工作的。为了让演示更加清楚，我们将系统版本的 $sp 称为 SSP，而用户版本的 Ssp 称为 USP。然而，在体系结构上（即从指令集的角度来看），它们指的是同一个寄存器。硬件（通过模式位）知道使用 USP 还是 SSP 作为 Ssp。

例4.2 图 4-15a 一直在执行程序，现在处理中断发出一系列步骤，称为 foo 的程序正在执行（见图 4-15a）。

现在设备 D 完成了处理。处理器正在执行单元 19999 的指令。它一直等到当前指令执行完毕。然后处理器进入 INT 状态（见图 4-15b）并将当前 PC 值（20000）存入 $k0。收到处理器的 INTA 信号后，设备将它的向量读到数据总线上。处理器使用数据总线接收设备的向量，如图 4-15b。

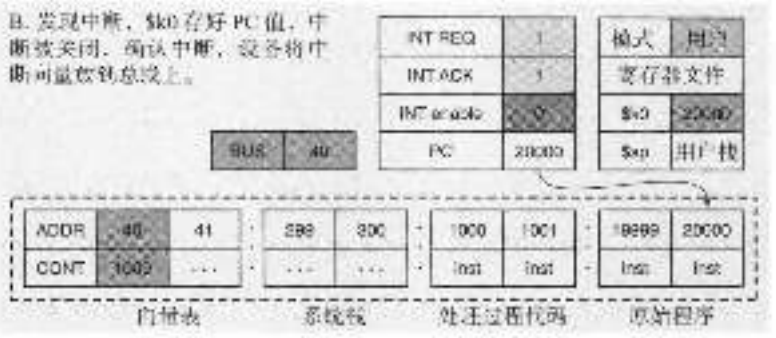
我们假设设备的向量是 40。处理器查找内存单元 40 以发现内处理过程地址（假设为 1000）。今 SSP 的内容为 300。在 INT 状态中，FSM 将 1000 载入 PC，将 300 载入 $sp，将当前模式保存在系统栈中，然后进入到 FETCH 状态。位于单元 1000 的处理过程代码（见图 4-9）开始执行，使用 $sp=299 作为它的栈（见图 4-15c）。

处理过程从中断返回后，原始程序再次在 PC=20000 恢复执行（见图 4-15d）。

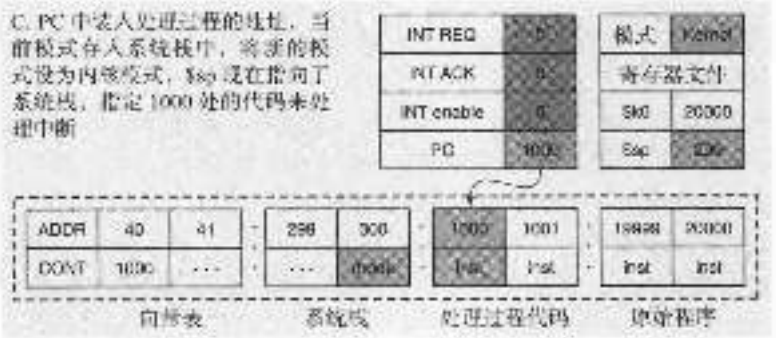


a）中断处理（按箭头指示顺序）

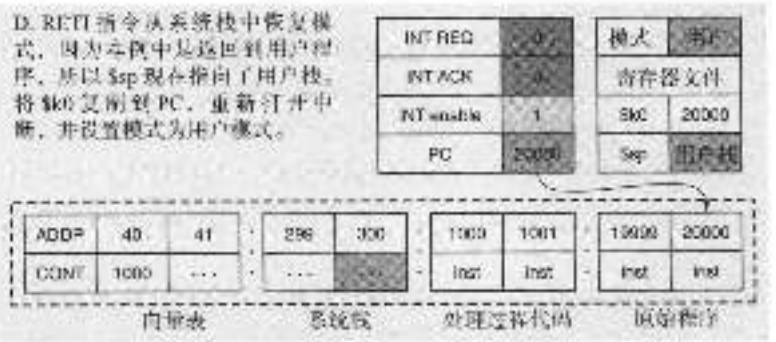
图 4-15 中断处理



b) 中断处理（INT：设备号——获取向量）



c) 中断处理（转交控制权给处理过程代码）



d) 中断处理（返回原始程序）

图 4-15 （续）

150
151

例 4-5 考虑如下情况：

假设内存地址是连续的整数。

用户程序执行内存单元 7500 的指令

用户栈指针（$sp）值为 18000

SSP 值为 500

键盘的向量为 80

硬盘的向量为 50

键盘中断处理过程的地址为 3000

硬盘中断处理过程的地址为 5000

a. 将上述信息用类似图4-15的方式画出。

b. 键盘产生了一个中断。表示当键盘中断处理过程即将开始执行时，处理器相关的状态（类似于图4-15c）。

c. 当键盘中断处理过程执行到打开中断后，硬盘产生了一个优先级更高的中断。假设键盘中断处理过程正在执行内存单元3023的指令，栈指针（Ssp）值为515。表示硬盘中断处理过程将要开始执行时，处理器的相关状态。

d. 表示当硬盘处理过程执行RETI指令时，处理器相关的状态（类似于图4-15d）。

e. 表示当键盘处理过程执行RETI指令时，处理器相关的状态（类似于图4-15d）。

这个例子留给读者作为练习。

小结

在本章中，我们介绍了一个重要的概念，中断。这使得处理器能够与外部世界进行通信。中断是程序不连续性的一种特定实例。我们讨论了支持嵌套中断所需要的最少的硬件改进，包括处理器内部的和总线层上的。

- 处理器的改进包括（3条）新指令、用户栈、系统栈、模式位以及INT宏状态。
- 在总线层上，我们介绍了专门的控制线，称为INT和INTA。这是为了让设备向处理器表明它想中断处理器并让处理器能够确认中断。

152 我们还回顾了陷入和异常，它们是同步的程序不连续性。有趣的是，处理这些不连续性所需要的软件机制是类似的。我们讨论了如何编写一个能够处理嵌套中断的通用的中断处理过程。

我们还特意简化了本章中中断的表示，便于让第一次上系统课程的学生接受。现代处理器中的中断机制相当复杂。例如，现代处理器将中断分为两类：可屏蔽的和不可屏蔽的。

- 前一种指的是能够通过关中断机制暂时关闭的中断（例如，设备中断）。
- 后一种指的是即使是关中断机制也无法关闭的中断（例如，系统检测到的内部硬件错误）。

我们介绍了一种机制，处理器使用这种机制来找到中断处理过程的起始地址（通过向量表），还有使用专用寄存器来保存被中断程序的返回地址。我们还介绍了一个简单的硬件方案，处理器识别中断设备的身份并确认中断。这些讨论的主要目的是告诉读者，设计这样的硬件是简单而直观的。

我们介绍了作为处理器内部状态特性的模式。这也是一个故意简化的版本。处理器状态可能还有其他一些位表示其他可用的属性信息（类似于模式位）。通常，处理器将所有这些位集合到一个寄存器中，称为处理器状态字（PSW）。遇到中断及中断返回时，硬件隐式地在系统栈上压入和弹出PC和PSW[⊖]。

我们还介绍了相当简单的中断处理过程代码来表达处理器需要做什么才能应付中断。典型的处理过程的工作远不止保存处理器的寄存器。在后面的章节中，我们将在操作系统功能，如处理器调度（第6章）和设备驱动（第10章）中，探究中断。

现代处理器中的中断体系结构远比这里给出的复杂。首先，因为处理器是珍贵的资源，所以大量与中断处理有关的事务，除了执行过程代码之外，都放在处理器的外部实现。例如，称为可编程中断控制器（PIC）的设备帮助处理器应对各种处理外部中断时的细节，包括：

- 应对多个中断级。

⊖ 在LC-2200中，我们指定一个寄存器$k0在INT宏状态中保存PC。许多现代处理器采取的另一种方法是直接将PC保存在系统栈上。

- 从设备收集实际的中断。
- 在产生中断的设备中选择优先级最高的设备。
- 获得被选中的对处理器发出中断的设备的身份（向量表索引）。
- 确认被选中的设备。

153

PIC 提供了处理器可读的寄存器，其中一个包含了发出中断的处理器发出中断的设备的身份。使用 PIC 简化了处理器在中断时需要做的工作。当到中断时，返回地址要么保存在系统栈上，要么保存在特殊的寄存器中，而控制权则简单地转交给操作系统设置好的一个地址，该地址对应于操作系统的通用第一级中断处理过程。这个处理过程只保存被中断程序的返回地址，并从 PIC 中读出中断设备的身份然后跳转到正确的处理代码。通常，操作系统的第一级处理过程是不可中断的且运行在一个称为中断模式的特殊模式中，因此操作系统不需要担心嵌套中断。一般地，设备驱动（实际操作设备的软件）在中断处理代码中做尽量少的工作是非常重要的。这是为了保证处理器不会一直被中断处理占用。设备驱动中只有时间敏感的代码才会写到中断处理过程中。例如，Linux 操作系统定义了 top-half 和 bottom-half 处理过程。根据定义，bottom-half 处理过程并没有 top-half 处理过程那么紧急。设备中非时间敏感的大量工作会在 bottom-half 处理过程中进行。

练习题

1. 遇到中断时，在将控制权转交给中断处理过程前，需要硬件上隐式地完成的工作有哪些？
2. 为什么不使用 JALR 从中断处理过程中返回？
3. 将下列步骤排成正确的顺序：

   处理过程的实际工作
   关闭中断
   打开中断
   从栈中恢复 Sk0
   恢复状态
   从中断返回
   将 Sk0 保存到栈中
   保存状态

4. 处理器如何知道哪个设备在请求中断？
5. 实现可被中断的中断需要什么指令？解释每一条的功能和用途，并解释如果没有这些指令会怎样。

   在下列中断处理过程代码中，选出不应位于其中的项。

   _____关闭中断
   _____保存 PC
   _____保存 Sk0 154
   _____打开中断
   _____保存处理器寄存器
   _____执行设备代码
   _____恢复处理器寄存器
   _____关闭中断
   _____恢复 Sk0
   _____关闭中断

_____恢复 PC
_____打开中断
_____从中断返回

7. 在下列 INT 密状态的操作中，选出不应位于其中的项。

_____保存 PC
_____保存 SP
_____Sk0 + PC
_____打开中断
_____保存处理器寄存器
_____通过断言 INTA 确认 INT
_____在数据总线上索引设备的中断向量
_____从中断向量表中索引 PC
_____从中断向量表中索引 SP
_____关闭中断
_____PC + 从向量表中得到的 PC
_____SP + 从向量表中得到的 SP
_____关闭中断

参考文献注释和扩展阅读

我们只是了解了处理器中断体系结构和操作系统高效中断处理机制的一些皮毛。对此感兴趣的读者可以阅读关于计算机组成的更高级的教材（如 [Patterson，2008]），以便了解现代处理器实现中断机制的有关细节。还有关于操作系统概念和实现的书 [Rubini，2001；Tanenbaum，2006；Silberschatz，2008]，以便获得对中断处理更深入的知识。在 Intel 的文档 [Intel System Programming Guide 3A，2008] 上可以找到关于 Intel 处理器中断体系结构很好的讨论。

155

处理器性能与流水线处理器的设计

在第3章中，我们介绍了一种简单的处理器设计，实现了LC-2200指令集。在我们讨论选择时钟周期的长度，以及减少在实现每个宏状态（取指、解码和针对特定指令的执行状态）时用到的微状态个数的时候，对如何提高性能有所涉及。

处理器的设计与实现可以进行定化的分析。体系结构的设计者们不停地评估加入某个特性对性能的影响有多大。因此，我们首先来讨论与处理器设计相关的性能指标。然后再看看各种用于改善处理器性能的方式。我们首先在第3章已经讨论了的简单处理器上下功夫，然后再来考虑在一个新的称做流水线的概念下的改进方式。

首先，让我们引入一些帮助我们理解处理器性能的指标。

5.1 时间和空间性能指标

如果说我们在建造一架飞机，你希望每趟飞行都能有若干乘客，你就得在机舱内提供足够的空间，足以容纳这么多乘客、他们的行李，以及路上提供给他们的食品。此外，你还得确保飞机在预定时间内从A地飞到B地。飞机上的乘客数也对飞行的耗时有影响：运载的乘客越多，需要运载的乘客和他们的行李的重量就越大，对于同样马力的引擎来说，飞行花费的时间就越多。

我们来把这个与处理器的性能做个类比。由于常见的对处理器的误解，我们在想到处理器性能的时候总是想到Mhz、GHz和THz。这些名词实际上当然描述的是处理器的时钟频率。我们在第3章知道时钟周期时间（即时钟频率的倒数）是由最坏情况下一个时钟周期里数据通路的延时来确定的。

理解为什么处理器速度不只是性能的决定因素也很重要。北方说，你写了一个程序foo，在某处理器上运行。你感兴趣的性能指标有两个：foo占用多少内存（空间指标），foo运行需要多久（时间指标）。为简单起见，空间的量化指标[⊖]，而执行时间是时间的量化指标。我们把前者定义为给定程序占用的空间，把后者定义为给定程序运行的时间。我们来看看这两个指标和我们关于处理器设计已经提及的部分有什么关系。

一个处理器的指令集架构对程序的内存印迹有影响。首先，我们得理解指标之间的联系。有人相信内存印迹越小，执行时间就会越短。在20世纪70年代普遍认为这个假定是正确的，于是人们便设计出了复杂指令集计算机（Complex Instruction Set Computer，CISC）体系结构。这是由于在20世纪70年代，编译器技术还处于起步阶段，高级语言和指令集架构之间有着广为人知的语义上的鸿沟。从事后来看，我们知道有效地编译一个程序并不需要异常复杂的指令；但是在当时，这个事实并不是那么显然。再加上那个时候内存非常昂贵，在内存使用方面尽量节省就成了指令集设计的目标。

CISC体系结构的评判标准是让数据通路和控制单元为每一条从内存中读取的指令做更多

⊖ 在第5章和第6章中我们将会看到，内存印迹表示操作系统为一个程序在整个执行时分配的虚拟空间。

[157] 差类作一个条准则导致指令集中不同复杂程度的指令完成执行需要经过不同数量的微状态。比如说，一个加法指令需要较少个微状态以完成执行，而一个乘法指令就需要多一些。类似地，一个使用寄存器操作数的加法指令就需要比较少个微状态，而使用内存操作数的加法指令就需要多一些。有了这些更复杂的指令，容易想象，对于给定的程序逻辑，只需要更少的指令就能实现，也就有了更小的内存印迹。

常识和早期计算机技术的进步削弱了这个关于内存印迹和程序性能之间联系的假设：

- 这是飞机类比开始失效的地方。在之前的类比中，乘客就类似内存中的指令。飞机中的每个乘客都需要被运到目的地，而且每个乘客都提高了飞机的总重量，后者又决定了飞机的飞行时间。但是，并不是程序里的每一条指令都一定会被执行到。比如说，众所周知很多产品级程序里一大部分代码是用来处理程序执行时可能遇到的异常情况的。你根据你自己的编程经验就知道好的软件工程实践是在系统调用之后检查返回代码。这部分检查错误处理的代码极少被执行到。为了把这一点说清楚，我们来想象一个包含一百万条指令的程序，其中有一个紧密的循环，只有 10 条指令却占用了 99% 的运行时间。这种情况下，程序到底有多大对执行时间当完全无影响。
- 第二，处理器实现技术的进步（主要是用流水线来执行指令的想法，我们会在本章后面部分加以讨论）模糊了复杂指令与一次完成简单工作的简单指令相比所拥有的优势。
- 第三，编程方式由汇编逐渐转向高级语言，指令集的有用程度的衡量标准变成了它对编写编译器的人来说多么有用。在很多方面来说，编译器技术的进步使得我们之前提及的语义鸿沟变得不那么可怕，也就使得指令集的设计逐步远离 CISC。值得一提的是，John Hennessy 和 John Cocke，两位精简指令集（随后会介绍）革命的先驱，都既研究体系结构，也研究编译器。
- 第四点，随着半导体技术和超大规模集成电路的进步，内存价格开始下降，也就使得程序大小没原来那么值得关心。CISC 关于空间的优势消失殆尽。随着这些进步，在 20 世纪 60 年代后期引入的缓存机制（我们将会在第 9 章中介绍）也由于半导体技术的进步而变得更加切实可行。随着缓存的到来，从主存读取数据到处理器的次数变少了，也就进一步地弱化了 CISC 的另一个根本假设，即读取一条复杂指令消耗的时间比读取几条简单指令要少。

以上几点使得精简指令集计算机（Reduced Instruction Set Computer，RISC）在 20 世纪 70 年代末到 20 世纪 80 年代初兴起。尽管当时有很多关于 CISC 和 RISC 哪个更好的争论，现在它们已经基本上不相干了。实际情况是两个阵营都从对方那里学到了一些好的特性。我们即将看到，在讨论流水线的时候，指令集对于保证处理器每个时钟周期执行一条指令，已经不重要了。比如说，现在具有统治性地位的指令集 Intel x86 是一个 CISC 体系结构的指令集，但是实现却用了 RISC 的哲学。从 Pentium Pro/Pentium II 开始，CISC 指令在 Intel x86 处理器内部被硬件转换为若干条 RISC 处理器。另一个具影响力的指令集是 ARM（Acron RISC Machine，或果精简机）。尽管 ARM 早期的时候是一个 RISC 的架构，当今流行的版本却已经包含了若干复杂指令。

[158] 我们将在讨论多级存储体系的第 9 章对内存印迹做更详细的讨论。现在，我们只需要注意程序的内存印迹和它的执行时间之间并没有太大的联系即可。

那么，是什么因素决定了程序的执行时间呢？处理器运行一个程序所执行的指令个数是执行时间的一个因素，另一个因素是执行每条指令所需要的微状态个数。由于每个微状态要

用一个 CPU 时钟周期来执行，每个指令的执行时间就可以由每条指令所用到的时钟周期个数（通常称作 CPI——Clocks Per Instruction，每指令时钟周期数）来测算。决定执行时间的第三个因素则是处理器的时钟周期时间。

如果 n 是程序执行的总指令数，那么就有：

$$\text{执行时间} = (\sum \text{CPI}_j) \times \text{时钟周期时间}，\text{这里 } 1 \leqslant j \leqslant n \tag{5-1}$$

有时候考虑一个程序所执行指令的平均 CPI 很方便。那么，如果 CPI_{avg} 是一个程序所执行的指令集的平均 CPI 的话，我们可以把执行时间表示成下式：

$$\text{执行时间} = n \times \text{CPI}_{avg} \times \text{时钟周期时间} \tag{5-2}$$

当然，很难定量地说平均 CPI 到底是多少，因为这个取决于不同指令的执行频率。我们将在下一节里更详细地讨论此问题。程序的执行时间是处理器性能的主要决定因素。也许更准确地说，由于时钟周期时间经常变化，时钟周期数（即，执行一个程序所用的时钟周期的个数）是一个更恰当的处理器性能的粗略标准。无论如何，应当说明处理器性能不仅仅是处理器速度。处理器速度无疑很重要，但是时钟周期数，即执行的指令个数与平均 CPI 的乘积，也是一个至少同样重要的决定程序执行时间的指标。

例 5-1 一个处理器有三类指令：

A, B, C; $\text{CPI}_A = 1$; $\text{CPI}_B = 2$; $\text{CPI}_C = 3$.

假如说，A 可能是算术和逻辑指令，B 可能是内存指令如加载和存储，C 可能是乘除法一类的复杂指令。

一个编译器产生两个不同的指令序列以完成相同的工作：

指令序列 1 执行 A 类指令 5 条，B 类指令 3 条，C 类指令 1 条； 159

指令序列 2 执行 A 类指令 3 条，B 类指令 2 条，C 类指令 2 条。

哪一个指令序列运行起来更快？

答：

指令序列 1 需要执行 9 条指令，总共需要 16 个时钟周期。

指令序列 2 需要执行 7 条指令，但是总共需要 17 个时钟周期才能执行完。

因此，指令序列 1 运行起来更快。

5.2 指令频率

知道程序里某个特定指令的执行频率是件很有用的事情。指令频率这个性能指标表示的就是这个量。静态指令频率是指特定指令在编译得到的代码中出现的次数，而动态指令频率是指特定的指令在该程序实际运行的时候被执行的次数。让我们来理解一下这两个指标的重要性。静态指令频率影响内存印迹，因此如果知道了某程序里某特定指令的出现次数特别多，我们就可以试图以巧妙的指令编码技术来减少它所占用的内存空间。动态指令频率则影响程序的执行时间。因此，如果知道了某个指令的动态频率特别高，我们就可以尝试对数据通路和控制部分进行改进以确保该指令的 CPI 尽可能小。

静态指令频率对通用处理器来说越来越不重要，因为减少内存印迹与提升处理器性能相比不是那么重要了。实际上，诸如特殊的指令编码这样用来改善静态指令频率的技术对性能的影响是负面的。特殊的指令编码打破了指令格式的统一，而指令格式的统一对于流水线处理器（见 5.10 节）是至关重要的。但是，静态指令频率对于需要在很受限制的内存空间里进

行优化的嵌入式处理器来说可能仍然是一个重要因素。

例 5.2 考虑下面的程序，其中包含了 1000 条指令。

160

```
I1:
I2:
..
..
..
I10:                  ┐
I11: ADD              │
I12:                  │ 循环
I13:                  │
I14: DONE OR I11      ┘
..
..
I1000:
```

ADD 指令在上面的程序中只出现了一次，指令 I_{10} 到 I_{14} 构成了一个循环 800 次的循环体。其他所有指令都恰好执行一次。

a. ADD 指令的静态频率是多少？

答：该程序内容只有 1000 条指令。在这 1000 条指令中，ADD 出现了多少一次。因此，ADD 指令的静态频率是 1 / 1000 × 100% = 0.1%。[⊖]

b. ADD 指令的动态频率是多少？

答：总共执行的指令数 = 在循环中执行的指令数 + 在循环外执行的指令数

= (800 × 5) + (1000 − 5) × 1

= 4995

每次循环会执行一次 ADD 指令，因此 ADD 指令执行次数 = 800。

ADD 指令的动态频率

= (ADD 指令的执行次数 / 总共执行的指令数) × 100%

= (800 / (995 + 4000)) × 100% = 16%

5.3 基准测试程序

我们现在来讨论如何比较不同机器的性能。经常能看到广告里大肆宣传诸如“X 处理器是 1GHz”或者“Y 处理器是 500MHz”的内容。既然执行时间并不完全是由处理器速度来决定的，我们如何确定哪一个处理器更好呢？基准测试程序是能够代表处理器负载的一组程序。例如，对于在游戏机里使用的一个处理器，基准测试程序也许是个视频游戏。对用于科学应用的处理器，矩阵操作可能是基准测试程序。通常，实际程序的核心被用做基准测试程序。比如说，矩阵乘法在若干科学应用里都会用到，并且在这些程序里往往是执行时间的主要部分。这时候，以矩阵乘法例程来对处理器进行基准测试就很有意义。处理器执行这样的核心程序的性能能很好地预测它执行整个应用程序的期望性能。

161

经常是由一组程序组成一个基准测试。有几种不同的方法来决定怎么使用这些基准测试程序评估处理器性能。

1）假设你有一组程序，它们必须一个接着一个运行。这时候，一个有用的总体指标是总执行时间，即这些单个程序的运行时间的累计总和。

2）如果你有一组程序，你想在不同的时候运行它们，而并不是同时运行所有的程序。这

⊖ 原文中 100 之后没有百分号，后面的类似公式中的 100% 的情况也类似。——译者注

种情况下，算术平均数（AM）是个有用的指标，即所有单个程序运行时间的平均数。值得一提的是，这个指标的汇总值偏向于较为耗时的基准测试程序（例如，程序执行时间：P1=100秒，P2=1秒；算术平均数AM是55秒）。

3）情况类似于上一种，但是你事先知道运行每个程序的频率。这种情况下，一个有用的指标是加权算术平均数（WAM），即所有单个程序运行时间的加权平均数。这个指标考虑了基准测试程序集中程序执行的相对频率（接着上面的例子，P1=100秒，P2=1秒，f_1=0.1，f_{10}=0.9，那么加权算术平均数WAM是0.1×100+0.9×1=10.9秒）。

4）如果你的情况类似于2），但是你完全不知道程序之间的相对执行频率。这时候，使用算术平均数可能会导致测量出来的处理器性能有偏向性。这种情况下，另一个汇总指标是几何平均数（GM），即p个数乘积的p次方根（接着上面的例子，P1=100秒，P2=1秒，几何平均数GM=sqrt(100×1)=10秒）。这个指标去除了算术平均数中存在的偏向于大数值的问题。

5）调和平均数（HM）也是一个有用的综合指标。从数学上说，它的计算方法是把数值的倒数的平均数算出来再取倒数。这也有助于矫正算术平均数中存在的偏向大数值的问题。我们一直在考虑的那个例子（程序的执行时间是P1=100秒，P2=1秒）的调和平均数HM是

$$\text{调和平均数} = 1/(\text{倒数的算术平均数})$$
$$= 1/(((1/100)+(1/1))/2) = 1.9801$$

162

在数据由比例构成时，调和平均数被认为特别有用。

如果所有的数值都一样，那么以上三个复合指标（算术平均数，几何平均数和调和平均数）得到的结果是相同的。一般来说，算术平均数趋向于偏向数据中的较大者，调和平均数趋向于偏向数据中的较小者，而几何平均数趋向于处在两者之间。一个有用的经验法则是在数值的绝对值很大的时候使用调和平均数，在绝对值很小的时候使用算术平均数。这里给我们的启示是，在使用单个综合指标来评价一个体系结构的时候，必须非常小心谨慎。

多年来，人们创建了若干个基准测试程序，用以评价体系结构。在工程/科学工作站最广为接受的是SPEC基准测试，由独立非营利机构标准性能评估公司（SPEC，Standard Performance Evaluation Corporation）开发。它的目标是“建立、维护和支持一组标准化的相关的基准测试程序，用于最新一代的高性能计算机。”[⊖] SPEC基准测试由一组实用的应用程序组成，包括了科学应用、事务处理、Web服务器等，代表了通用处理器的常见负载。[⊜]

处理器性能不只是由处理器时钟频率决定这一事实，使得进行基准测试很困难。比如说，除了时钟频率以外，内存系统的组织和处理器-内存总线带宽也都是关键性的决定因素。而且，不同的基准测试程序的行为对整体系统有着不同的需求。因此，当对比两个时钟频率相似甚至相同的处理器时，我们也许会发现其中一个在某些基准测试程序上表现较好，而另一个在另一些测试中表现较好。这就是为什么在不知道要运行什么样的负载的时候，综合指标很有用。但是，我们也必须谨慎地避免过度使用统计指标，正如名言所说，“谎言，该死的谎言，以及统计数字”。[⊝]

⊖ 参见www.spec.org/。

⊜ 参见www.spec.org/cpu2006/publications/CPU2006benchmarks.pdf，里面包含了对SPEC2006基准测试程序测量函数和浮点性能的介绍。

⊝ 参见www.york.ac.uk/depts/maths/histstat/lies.htm。

163 **例 5-3** SPECint2006 整数基准测试程序集由 12 个程序组成，用以定量测量处理器执行整数计算程序（与浮点运算相对）的性能。下表[⊖]展示了 Intel Core 2 Duo E6850 (3 GHz) 处理器运行 SPECint2006 的性能：

程序名	描述	运行时间（秒）
400.perlbench	Perl 语言的应用程序	510
401.bzip2	数据压缩	602
403.gcc	C 语言编译器	382
429.mcf	组合优化	228
445.gobmk	游戏人工智能（围棋）	549
456.hmmer	基因序列搜索	593
458.sjeng	国际象棋人工智能	679
462.libquantum	量子计算	422
464.h264ref	视频压缩	708
471.omnetpp	离散事件模拟	362
473.astar	寻路算法	465
483.xalancbmk	XML 处理	302

a. 计算算术平均数和几何平均数。

答：

算术平均数 = (510 + 602 + … + 302)/12 = 491.8 秒。

几何平均数 = (510 × 602 × … × 302)1/12 = 474.2 秒。

注意，算术平均数的结果简有于这组数据中的较大值。

b. 某系统用来以如下频率运行这 12 个程序：

- 10% 视频压缩
- 10% XML 处理
- 30% 寻路算法
- 50% 所有剩下的程序

计算此工作负载下的加权算术平均。

答：

此工作负载的 50% 时间被均匀分给 9 个程序，它们的平均执行时间

= (510 + 602 + 382 + 328 + 548 + 593 + 679 + 422 + 362) / 9

164 = 491.8 秒。

加权算术平均 = (0.1 × 708 + 0.1 × 302 + 0.3 × 466 + 0.5 × 491.8)

= 486.7 秒。

这些指标有用的原因之一是它们为对比不同体系结构、不同实现和不同硬件规格的机器提供了一个基准。但是，正如例 5-3 所示，单纯的数字难以用来对比不同的机器。因此，SPEC 基准测试的结果是以与某参考机器的比值的形式公布的。例如，如果一个基准测试程序在目标机器上运行需要 x 秒，而同一个程序在参考机器上运行需要 y 秒，那么这个基准测试程序在目标机器上的 SPECratio（SPEC 比值）就被定义为

SPECratio = 参考机器上的执行时间 / 目标机器上的执行时间 = y/x

SPEC 组织选择 Sun 微系统公司的 Ultra5_10 工作站作为 SPEC CPU 2000 测试的参考机

⊖ 来源：www.spec.org/cpu2006/results/res2007q4/cpu2006-20071112-02562.pdf。

器，它使用了一颗300MHz的SPARC处理器，有256MB的内存。这种机器也是SPEC CPU 2006的参考机器。

不同基准程序的SPECratio可以用几种统计学手段（算术、加权算术、几何、调和）中的一种来合并成单一的结果。因为我们在使用SPEC基准测试作为标准的性能报告手段时面对的数字是比例值，习惯上使用调和平均数。

SPECratio的好处是它是比较机器的基准。比如说，如果机器A和B的平均SPECratio分别是R_A和R_B，我们就可以立刻得出这两台机器的性能的对比结论。

5.4 提升处理器的性能

为了探索提升处理器性能的几条道路，一个好的出发点是那个关于执行时间的方程。让我们来独立地看它的每一项，理解它们分别提供了怎样的优化机会。

- **减少时钟周期时间** 时钟周期是由最坏情况的数据通路延时决定的。降低时钟周期时间也就是提升时钟频率。我们可以重新安排数据通路的元素以降低最坏情况的延时（比如说，在数据通路的布局上把它们摆放得更靠近一些）。我们还可以减少单个时钟周期里数据通路动作的个数。但是，这种对减少时钟周期的尝试对执行不同指令所需要的CPI个数是有影响的。如果还想要再减少时钟周期时间，就得设法减小单个数据通路元素的特征尺寸。要沿着这条优化的道路走下去，就需要设计出新的芯片制造工艺和装置技术，来减小特征尺寸。 [165]
- **改进数据通路组织以减小CPI** 在第3章中，实现使用了单一的总线。这样的组织减小了硬件中数据通路的元素的并发程度。我们暗示过，可以通过使用多条总线来提升硬件的并发性。再说一次，任何这类尝试都可能对时钟周期时间有着负面影响，因此需要仔细分析。设计处理器的微体系结构并优化实现来获得最高的性能，无论在学术界还是工业界都是一个多产的研究领域。

以上两点都集中在降低单条指令的延迟，以使得总的执行时间累积起来得以减少。另一个减少执行时间的机会是减少指令条数。

- **减少执行的指令条数** 一种减少程序里执行的指令条数的可能是，把简单指令换为更加复杂的指令。这也当能减少程序执行的总指令条数。我们已经看到过这个思路的反例。再重复说一遍，在试图引入新的复杂指令的时候，必须仔细权衡CPI、时钟周期时间和动态指令频率。编译器优化是另一个减少执行指令条数的优化方式，在现代计算机系统里，这对于决定长时间运行的程序的执行时间至关重要。

从前面的讨论中也应当看出，执行时间的三个组成部分是紧密相关的，必须一起进行优化，不能割离起来考虑。

例5-4 某体系结构有三种指令，CPI如下：

类型	CPI
A	2
B	5
C	1

某体系结构设计者计算出可以在不影响其他指令类型的CPI的情况下把B指令的CPI降低到3，但是同时也会增加该CPU的时钟周期时间。问，时钟周期时间增长多少比例以内，这个体系结构的修改有意义？假设该处理器上执行的所有的工作负载里A指令占30%，B指令占10%，C指

令占 60%。

答：

166 令 Co 和 Cn 分别为旧机器和新机器的时钟周期时间，令 N 为程序里执行的指令总数。

旧机器的执行时间是

$ET_{旧机器} = N \times (F_A \times CPI_{Ao} + F_B \times CPI_{Bo} + F_C \times CPI_{Co}) \times Co$

其中 F_A、CPI_{Ao}、F_B、CPI_{Bo}、F_C、CPI_{Co} 分别是各种指令的动态频率和 CPI。

旧机器的执行时间是：

$ET_{旧机器} = N \times (0.3 \times 2 + 0.1 \times 5 + 0.6 \times 1) \times Co$

$= N \times 1.7\ Co$

新机器的执行时间是

$ET_{新机器} = N \times (0.3 \times 2 + 0.1 \times 3 + 0.6 \times 1) \times Cn$

$= N \times 1.5\ Cn$

为了使新设计有意义，就必须有

$ET_{新机器} < ET_{旧机器}$

$N \times 1.5\ Cn < N \times 1.7\ Co$

$Cn < 1.7/1.5\ Co$

$Cn < 1.13\ Co$

因此时钟周期最多允许上升 13%。

5.5 加速比

比较处理器执行同一个程序或者基准测试程序集的执行时间，是理解一个处理器与另一个处理器相对性能的最显然的方法。类似地，我们可以比较进行某种被提议的改动前后的执行时间的改进程度，以量化这个修改所带来的性能提升。

定义

$$加速比_{A\ 比\ B} = \frac{处理器\ B\ 上的执行时间}{处理器\ A\ 上的执行时间} \tag{5-3}$$

处理器 A 与处理器 B 的加速比是处理器 B 上的执行时间与处理器 A 上的执行时间的比值。

类似地，

$$加速比_{改进} = \frac{改进前的执行时间}{改进后的执行时间} \tag{5-4}$$

167

改进带来的加速比是改进前的执行时间与改进后的执行时间的比值。

例 5-5 以下是某体系结构的指令的 CPI：

指令	CPI
ADD	2
SHIFT	3

其他 2（所有指令均予以，包括 ADD 和 SHIFT）

通过对程序的性能进行分析，一个体系结构设计者发现 ADD 指令后面紧跟着 SHIFT 指令的指令在整个程序中占据了 20% 的动态频率。他设计了一条新指令，一个 ADD/SHIFT 为组合，CPI 为 4。

如果把程序三的所有 {ADD，SHIFT} 都换成这条新指令，程序性能能提升多少？

答：

令原程序中执行的指令条数为 N，那么原程序的执行时间

= N × ADD/SHIFT 的频率 × (2 − 3)/2 + N × 其他指令的频率 × 2

= N × 0.2 × 2.5 + N × 0.8 × 1.875 = 2.0N

在把 {ADD，SHIFT} 换成新指令以后，新程序的总指令数量低到了 0.9N。在这个新程序中，组合指令出现的频率是 1/9，其他指令出现的频率是 8/9。

新程序的执行时间

= (0.9 N) × 组合指令的频率 × 4 + (0.9 N) × 其他指令的频率 × 2

= (0.9 N) × (1/9) × 4 + (0.9 N) × (8/9) × 1.875

= 1.9 N

程序的加速比 = 旧执行时间 / 新执行时间

= (2.0N) / (1.9N)

= 1.05 ⊖

另一个有用的指标是改进带来的性能提升比例

$$\text{执行时间的提升比例} = \frac{\text{原执行时间} - \text{新执行时间}}{\text{原执行时间}} \tag{5-5}$$

168

例 5-6 一个程序要执行 1000 条指令，平均 CPI 为 3，时钟周期时间为 2 ns。一个体系结构设计者提出了两个可能的方案：（1）可以将指令的平均 CPI 减少 25%，但时钟周期时间延长 10%；（2）可以将时钟周期时间减少 20%，但是平均 CPI 会增加 15%。

a. 体系变化决定选择那个选项的经理，给出体的决策考虑的理由。

答：

令 E0、E1、E2 分别表示原始机器、第一个选项和第二个选项下的执行时间。

E0 = 1000 × 3 × 2 ns

E1 = 1000 × (3 × 0.75) × 2 (1.1) ns = 0.825 E0

E2 = 1000 × (3 × 1.15) × 2 (0.8) ns = 0.920 E0

选项 E1 更好，因为它的执行时间比 E2 好。

b. 你选择的选项的执行时间与原设计相比，有多少的改进？

答：选项 1 相对原设计的改进 = (E0 − E1) / E0

= (E0 − 0.825 E0) / E0

= 0.175

因此改进比例为 17.5%。

另一种理解改进的效果的方式是，考虑它带来的改变影响执行时间的程度。例如，这个改动可能只影响到执行时间的一部分。一个以并行计算先驱吉恩·阿姆达尔（Gene Amdahl）命名的定律可以用来说明这个想法：

阿姆达尔定律

$$\text{Time}_{\text{改进后}} = \text{Time}_{\text{不受影响的部分}} + \text{Time}_{\text{受影响的部分}} / x \tag{5-6}$$

在上式中，$\text{Time}_{\text{改进后}}$ 是改动后的总执行时间，是不受改动影响的部分的执行时间（$\text{Time}_{\text{不受影响的部分}}$）加上受影响的部分的执行时间（$\text{Time}_{\text{受影响的部分}}$）除以这个改进对受影响部分的加速

⊖ 原书中的计算有误，已更正。——译者注

比（以 x 表示）。简单来说，阿姆达尔定律意味着提升处理器性能主要应当把资源花在对执行时间影响最大的关键指令上。[1]

169

表 5-1 总结了我们至今为止对论过的与处理器相关的性能指标。

表 5-1 性能指标小结

名称	记法	单位	备注
内存印迹		字节	程序在内存中占用的总空间
执行时间	$(\sum CPI_j) \times$ 时钟周期频率，对 $1 \le j \le n$	秒	执行恰好有 n 个指令的程序的运行时间
算术平均数	$(E_1 + E_2 + \ldots + E_p) / p$	秒	p 个基准测试程序执行时间的平均值
加权算术平均数	$(f_1 \times E_1 + f_2 \times E_2 + \ldots + f_p \times E_p)$	秒	p 个基准测试程序执行时间的加权平均值
几何平均数	$(E_1 \times E_2 \times \ldots \times E_p)$ 的 p 次方根	秒	p 个基准测试程序执行时间的乘积的 p 次方根
调和平均数	$1/((1/E_1) + (1/E_2) + \ldots + (1/E_p))/p)$	秒	p 个基准测试程序执行时间的倒数的算术平均数的倒数[2]
静态指令频率		%	指令 i 在编译出的代码中出现的次数
动态指令频率		%	指令 i 在执行的代码中出现的次数
加速比（M_A 比 M_B 的）	E_B/E_A	无量纲数	机器 A 与机器 B 相比的加速比
加速比（改进措施的）	$E_{旧}/E_{新}$	无量纲数	改进带来的加速比
执行时间的改进比例	$(E_{旧}-E_{新})/E_{旧}$	无量纲数	新与旧相比的改进比例
阿姆达尔定律	$Time_{改进后} = Time_{不改进的部分} + Time_{改进的部分}/x$	秒	x 是改进的加速比

170

例 5-7 某处理器把 20% 的时间花在 ADD 指令上。一个工程师提出将 ADD 指令的性能优化成原来的两等，这个改动对性能的提升多少？

答：

这个改进只对 ADD 指令有效，所以有 80% 的执行时间不受影响。

原来的标准化的执行时间 = 1

新的执行时间 = (花在 ADD 指令上的时间 / 4) + 剩下部分的执行时间

= 0.2 / 4 + 0.8

= 0.85

加速比 = 改进前的执行时间 / 改进后的执行时间

= 1 / 0.85

= 1.18

5.6 提升处理器的吞吐量

到现在为止，我们主要关注的是通过减少单条指令的延迟来提升处理器性能的技术。另一种提升处理器性能的思路则截然不同——不关注单条指令的延迟（即 CPI 指标），而关注吞吐量，即每单位时间处理器所执行的指令数。延迟回答的问题是处理器执行单条指令所花费的时间（即 CPI），而相反，吞吐量回答的问题则是处理器每个时钟周期执行多少条指令（可

[1] 阿姆达尔定律对并行机器上的程序的性能改进有严重影响。参见第 12 章的练习题 16，其中会涉及并行系统。

[2] 原文写了最后"的倒数"几个字。——译者注

称作 IPC，Instruction Per Clock cycle，每时钟周期的指令数）。这个概念就叫做流水线，它是本章剩余部分的焦点。

5.7 流水线简介

欢迎来到比尔的三明治店！比尔的商店里有各式各样的面包、调味品、奶酪、肉和蔬菜可供用户选购，这些食品分门别类地放在各个柜台里。在比尔创业早期，他的整个团队就是他自己一个人。他接到订单，按10序经过一个接一个的柜台，同时根据订单要求做出相应的三明治。现在他的生意已经扩大了，他有五个雇员，每个站在一个柜台前，组成一个三明治流水线：接订单、选择面包和调味料、选择奶酪、选择肉类、最后选择蔬菜。下表就展示了三明治流水线流程。

柜台1（接订单）	柜台2（选择面包）	柜台3（选择奶酪）	柜台4（选择肉类）	柜台5（选择蔬菜）
接订单（第5个）	第4个订单	第3个订单	第2个订单	第1个订单

每个柜台的雇员都在处理一个不同的订单，最后一个柜台（柜台5）在处理第一个订单的时候，第一个柜台已经在处理一个新的三明治的订单了。每个柜台在做了自己的事情以后都把三明治半成品以及订单传递给下一个柜台。比尔是个聪明的管理者，他并没有让一个雇员去处理整个订单，因为这样的话他的柜台上就得堆满做三明治的所有材料，也就毫无必要地增加了重复的原材料的量，因为每个顾客可能只用到原材料的一部分。相反，比尔有个地把工作分配成大致相等的工份，因此不会有哪个雇员无所事事。当然了，如果某个特定的三明治订单不需要某类材料，那么对应的雇员只需要传递一下半成品就行了。不过，大部分时间（尤其是高峰时间）里，比尔和雇员都源源不断地快速制作着三明治。

比尔将他供应三明治给消费者的速度提升到了原来的五倍。

5.8 指令处理流水线

你能猜得出来，我们拿比尔和三明治店要类比什么。在LC-2200的简单实现里，有限自动机每次执行一条指令，从取指令、解码、最后到执行，一路做下去，然后再进行下一条指令。这个方法的问题是数据通路没有得到良好的利用，这是因为，对于每个宏状态来说，并不是所有的资源都用上了。我们不必回想在第3章中实现LC-2200指令集的时候所使用的特定的数据通路。因为，不管数据通路的细节如何，我们知道任何一种LC-2200指令集的实现都需要以下的数据通路资源：内存、PC、ALU、寄存器堆、IR和符号扩展器。图5-1展示了我们在若干个指令的每个宏状态里使用到的硬件的数据通路资源。

宏状态	用到的数据通路资源			
取指令	PC	ALU	寄存器堆	内存
解码	IR			
执行（ADD）	RA	ALU	寄存器堆	
执行（LW）	RA	ALU	寄存器堆	内存　符号扩展器

图5-1　不同的宏状态使用到的数据通路资源

我们可以立刻得出以下两点观察结果：

1）IR在每个宏状态里都被用到了。这并不奇怪，因为IR包含了指令，然后执行中的每

一步都会用到IR的一部分。IR就等价于三明治流水线里的“订单”，需要传递过每个柜台。

171 ~ 172

2）在宏观的层面上来说，我们能看出这二个宏状态中需要做的工作是不同。有限自动机里的每个状态就等价于三明治流水线里的一个柜台。

我们想尽可能地一直利用数据通路的所有硬件资源。在三明治流水线里，我们通过同时制作多个三明治让所有的柜台都一直忙碌。我们因而尝试把三明治流水线的想法实施到处理器的指令执行上面去。一个程序就是一个指令序列，如图5-2所示。

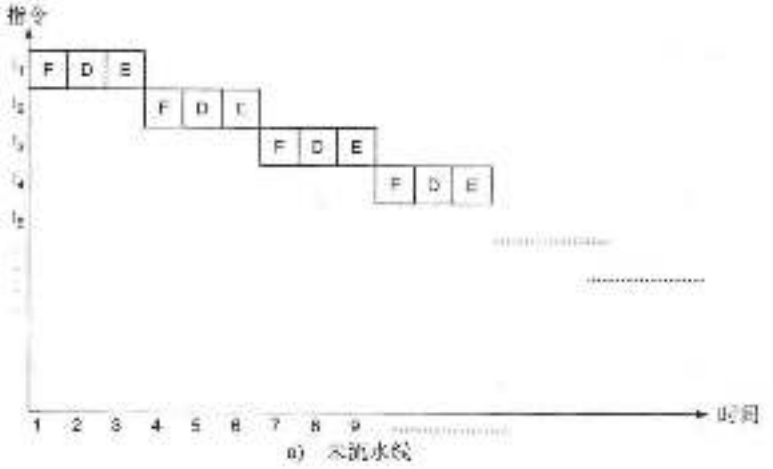
```
I1:  LW R1, MEM [1000];   R1 ← 内存地址 1000 处的数据
I2:  LW R2, MEM [2000];   R2 ← 内存地址 2000 处的数据
I3:  ADD R3, R5, R4;      R3 ← R4 + R5
I4:  NAND R6, R7, R8;     R6 ← R7 NAND R8
I5:  SW R9, MEM [3000];   R9 → 内存地址 3000 处的数据
I6   .....
I7   .....
I8
I9
I10
I11
I12
I13
```

图5-2 一个程序就是一个指令序列

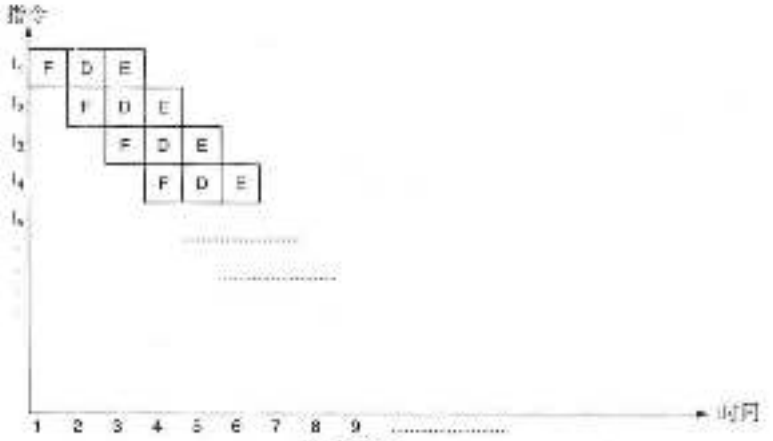
在使用有限状态机的简单实现里，处理器执行指令的时间轴看起来就像是图5-3a的样子。前一条指令执行完了，新一条指令才会执行。采用三明治流水线的想法，我们能看出，为了最大化数据通路资源的利用率，我们应该在指令流水线里同时执行多条指令，如图5-3b所示。问题就来了：这可能吗？如果你是三明治流水线中的雇员之一，制作你的三明治完全不取决于流水线里的前一个或者后一个顾客想要什么样的三明治。程序里的相邻指令是不是也是类似地互相不依赖呢？你会立刻觉得答案是不，因为程序是顺序执行的。然而，看看图5-2里的指令序列，尽管程序是顺序执行的，但是可以看出指令I_1到I_5这五条指令都碰巧互相不依赖。也就是说，一条指令的执行不依赖于它之前的指令的结果。我们很快会指出，这种令人开心的情况并不总是常态。我们会在5.13节中处理处理器流水线的这种依赖关系。但是，现在为了方便考虑，我们不妨认为指令互相不依赖，以说明将三明治流水线的想法应用于处理器流水线设计上的可能性。

173

注意，在图5-3b中的流水线执行的时间轴里，每当前一条指令进入下一个状态就开始处理一条新的指令。如果这是可行的，那么就能把指令的吞吐量提高到三倍。主要观察数据通路资源类似于三明治流水线中的各种食材，宏状态类似于三明治流水线中的各个柜台的雇员。



a) 无流水线



b) 流水线

图 5-3 执行的时间轴：a 中处理器一次处理一个指令，I_1 依次执行完 F、D、E 寄状态之后，I_2 才执行，以此类推。b 中，同时会有多条指令处在不同的执行阶段，比如说，在时刻 3，I_1 在 E 状态，I_2 在 D 状态、I_3 在 F 状态 [174]

5.9 简单指令流水线的问题

以下是简单地把三明治流水线应用到指令流水线上产生的问题。

1）不同的执行阶段经常会用到同一个数据通路的资源（如 ALU 和 IR）。

2）不同阶段的工作量不相同。例如，对比一下图 5-1 中解码和执行（LW）状态的工作量。前者是简单的组合函数，用以确定指令类型和所需资源；而后者却涉及计算地址、内存访问和把数据写入寄存器堆。一般来说，执行阶段所做的工作远远超过了其他阶段所做的工作。

让我们来理解一下第一个问题，即不同执行阶段之间存在对资源的争夺意味着什么。这常常被称做结构性冒险，它是由数据通路的局限性，如单一 IR、单一 ALU 以及单一连接数据通路的各元素的总线导致的。在三明治生产线中，一个订单（等价于 IR）就是一张纸，在柜台之间传来传去，而三明治半成品（等价于执行了一部分的指令）也在柜台之间直接传递（也就是说，并没有用到我们的数据通路例子里用到的集中“总线”）。我们可以通过使用类

似的想法来修正我们的指令流水线的问题。比如说，如果我们给取指令阶段增加一个额外的ALU，一个额外的IR以及一个额外的访存器，这个阶段与其他阶段之间就没有资源争夺了。容易理解为什么需要一个额外的ALU，因为取指令和执行两个阶段都需要用到ALU。但是我们需要理解拥有一个额外的指令寄存器意味着什么。好比三明治生产线里面订单从一个柜台传递到另一个柜台，我们把IR里的内容从取指令阶段传递到解码阶段，依次类推，以确保各个阶段互相独立。

另一种更严重的结构性冒险，由流水线的取指令和执行阶段引起。取指令阶段需要在每个时钟周期访问内存以取得指令。此外如果执行阶段有加载或存储指令，那么它也需要访问内存。我们怎样才能解决这个问题呢？一个简单的解决方案是规定取指令和执行阶段要访问不同的内存区域。有两个原因说明这样的设计是合理的：(a) 把这两者分离是个良好的编程实
175 践；(b) 绝大多数现代的处理器（比如说第7章将要讨论的Intel x86体系结构里采用的内存分段机制）把内存区域分隔成不同区域，以确保程序不会在无意中修改指令内存。因此，我们把整个内存分成I-MEM（即指令内存）和D-MEM（即数据内存），以使得取指令和执行阶段互相独立。现在，程序的指令来自I-MEM，程序所操作的数据结构来自D-MEM。因此，我们在设计上就消除了这个结构性冒险。

第二个问题意味着流水线的每个阶段所需的执行用时是不同的。回想一下，比尔精心设计了他的三明治流水线以确保每个雇员在他的柜台上所做的工作量大致相当。这里的缘由是流水线上最慢的环节限制了整个流水线的吞吐量。因此，为了让指令流水线也能这么高效，我们应当将指令执行的阶段切开，使得各个阶段所做的工作量大体相当。

5.10 修正指令流水线里的问题

现在来考虑解码阶段。这个阶段在目前的设计里是工作量最小的一个阶段。为了让工作量更加平均，我们必须得给这个阶段分配更多的工作。这里面临着一个两难问题，即我们在知道指令是什么之前，实在是什么都做不了。但是，我们可以投机性地做点事，只要不影响指令的实际语义就好。

我们认为大部分指令都会用到寄存器里的值。因此，我们可以提前从寄存器堆中读取寄存器内容，而不必实际了解到底是什么指令。在最坏的情况下，我们可以禁止使用从寄存器中读取的值。然而，为了能如此做，我们需要知道要读取哪个寄存器。因此，在设计指令集的时候，指令格式至关重要。如果回到第2章，去一看LC-2200里面用的寄存器的指令集（ADD、NAND、BEQ、LW、SW和JAL），你会发现表示源寄存器的位，不管是算术/逻辑指令的还是内存地址的，在指令的格式里总是占着相同的位置。我们可以利用这个事实，投机地在解码分析指令是什么类型的同时读取寄存器。根据同样的思想，我们可以把可能会做很多工作的执行阶段拆分成几个更小的阶段。

现在让我们根据以上的论证来把指令的处理过程划分为以下五个功能性组件，或者说阶段。

- IF　此阶段把PC所指向的指令从I-MEM读取出来放入IR中，然后把当前的PC加1以为取下一条指令做准备。
- ID/RR　此阶段将指令解码，并把正在解码的指令所需要的寄存器的值从寄存器堆中读出。LC-2200指令集有单操作数、双操作数及三操作数的指令。但是，值得一提的关键点是，任何一条指令都只需要至多两个寄存器的值。例如，ADD、NAND和BEQ需要从寄存器堆中读取两个源操作数。类似地，SW需要读取一个寄存器值来计算地址，读

取为一个寄存器值来得到要存进内存中的值。为了实现这样的功能，寄存器堆必须是双端口的，也就是说在同一个时钟周期内可以同时给寄存器堆两个寄存器地址而读取出对应寄存器的值。我们把这样的一个寄存器堆称作一个双端口寄存器堆（Dual-Ported Register File，DPRF）。现在，需要传递给双端口寄存器堆的寄存器地址取决于指令类型，因为不同类型（R 类、I 类和 J 类，参见 2.10.1 节）的指令里的寄存器地址在不同的部分。这不是个问题，因为我们可以用组合逻辑来根据指令的操作码字段在 IR 中找出表示寄存器地址的部分。因为这个阶段既包含解码的逻辑，也涉及读取寄存器堆，我们就给它了一个混合的名字。 176

- EX　本阶段负责处理所有指令中所需的算术和逻辑运算。你会在 5.11 节中看到，在 EX 阶段里，一个 ALU 可能不足以满足所有指令的需求。
- MEM　本阶段对于 SW 和 LW 指令分别负责读取内存和写回内存。不访问内存的指令不需要在本阶段进行操作。
- WB　本阶段对于需要将值写回寄存器的指令，执行将值写入目的寄存器的操作。在 LC-2200 中需要写入目的寄存器的指令包括算术/逻辑运算和加载指令。

图 5-4 图示了一条指令穿过一条流水线的过程。类似于我们心目中的三明治流水线，我们也希望每条指令都要流过指令流水线中的每个阶段。在任何时间，流水线上有五条指令正在执行：当指令 I_1 在 WB 阶段时，指令 I_5 在 IF 阶段。这是一条同步流水线，因为在每个时钟脉冲，每条指令的部分结果都被传递到下一个阶段。这里隐含的假设是时钟周期足够长，以至于流水线上最慢的部分也足以在时钟周期时间里完成它的功能。

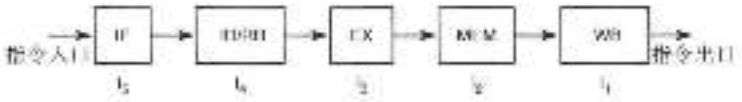


图 5-4　指令经过流水线的过程

每个阶段都以前一个时钟周期计算得到的部分结果为基础进行工作。并不是每条指令都需要每个阶段，比如 ADD 指令并不需要 MEM 阶段。但是，这仅仅意味着 MEM 阶段在得到 ADD 指令的部分结果时的这个时钟周期里什么也不做。（这里我们能明显看出它和三明治流水线的类似之处。）

你也许在看了特定指令的执行过程以后，会觉得这样的设计是低效的。比如说，这个设计给 ADD 指令的执行过程加上了一个空转的周期。但是，这个流水线设计的目标是提升指令处理的吞吐量，而不是减少每个指令的延时。回到三明治生产线的例子，其中类似的标准是保持客户排的队移动。每时间单位里服务的顾客数就类似于流水线处理器里每个时间单位里处理的指令条数。

因为每个阶段在处理不同的指令，每当一个阶段完成它的工作，它就必须得把它的结果放在下一个阶段知道的某个地方，以便它在下一个时钟周期可以获取这个结果。这被称作把一个阶段的结果缓冲起来。缓冲机制对于各个阶段的独立性至关重要。图 5-5 展示了添加了缓冲器的指令流水线。流水线寄存器是各个阶段之间的缓冲器的常用称呼。流水线寄存器和缓冲器在本章中意义相同，可以互换使用。在我们的三明治流水线的例子里，部分完成的三明治起到缓冲器的作用，给予各个阶段以独立自主。

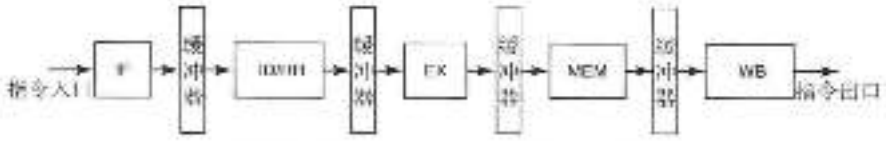


图 5-5 带缓冲器的指令流水线，缓冲器让每个阶段独立自主

5.11 指令流水线的数据通路元件

下一步是确定流水线的每个阶段所用到的数据通路元件，以及各个阶段之间的用于提供隔离的缓冲器的内容。

- 在 IF 阶段我们需要 PC、ALU 和 I-MEM。这个阶段的输出是从内存中获取的指令。因此，IF 和 ID/RR 阶段之间的流水线寄存器应当包含指令。
- 在 ID/RR 阶段，我们需要双端口寄存器堆；本阶段的输出是从寄存器堆里读取出的内容（记作 A 和 B）和指令解码的结果（指令的操作码，以及指令可能包含的偏移量）。这些就是 ID/RR 和 EX 阶段之间的流水线寄存器。
- EX 阶段要处理所有指令所需的算术运算。因为这是唯一一个要为指令进行算术运算的阶段，我们需要确定最坏情况下这个阶段所需的资源。这依赖于指令集。

在我们的例子（LC-2200）里，需要多于一个算术操作的唯一一个指令是 BEQ 指令。BEQ 指令需要一个 ALU 来进行比较（A==B），另一个 ALU 来计算有效地址（PC+带符号的偏移量）。因此，我们在 EX 阶段需要两个 ALU。

177
↓
178

BEQ 指令也带来了另一个需求。计算地址的算术运算需要知道对应于 BEQ 指令的 PC 的值。（碰巧的是，PC 的值还有一个指令也需要，即 JALR。）因此，PC 的值也应当被从前一个阶段沿着流水线传递到后一个阶段（连同其他需要传递的东西）。

EX 阶段的输出是算术操作的结果，也取决于特定的指令。EX 和 MEM 阶段之间的流水线寄存器里面存储什么内容依赖于指令。比如说，如果指令是 ADD，那么流水线寄存器里就会包含加法的结果、操作码，以及目的寄存器描述符（Rx）。我们可以看出，对于任何一条指令，PC 的值在 EX 阶段之后都不需要了。处理其他指令时流水线寄存器里面要存什么值的问题就留给读者作为练习。

- MEM 阶段需要访问 D-MEM。如果某个时钟周期里本阶段处理的指令不是 LW 或 SW，则输入缓冲器里的内容会被在时钟周期结束时简单地复制至输出缓冲器。对于 LW 指令，本时钟周期的输出缓冲器要包含读取到的内存内容、操作码，以及目的寄存器描述符（Rx）。而对于 SW 指令，输出缓冲器包含操作码。类似地，我们也容易看出，如果操作码是 SW 的话，WB 阶段里什么也不用做。
- WB 阶段要用到寄存器堆（DPRF）。这个阶段只与要写入值到目的寄存器的指令（如 LW、ADD 和 NAND）有关。这带来了一个有趣的两难局面。我们知道每个时钟周期里每个阶段都在处理不同的指令。因此，参见图 5-4，WB 在处理 I_1 的同时，ID/RR 在处理 I_4。这两个阶段需要处理不同的指令而同时访问 DPRF。（例如，I_1 也许是 ADD R1，R3，R4，而 I_4 也许是 NAND R5，R6，R7。）幸运的是，WB 是在写入寄存器，而 ID/RR 是在读取。因此，从两个阶段在进行的逻辑操作的角度来说，并没有发生冲突。并且，只要被读和写的寄存器不是同一个，两个操作就可以同时进行。在一个时钟周期里同时

读写寄存器是一个语义冲突（举个例子，考虑 I_1 是 ADD R1，R3，R4 而 I_2 是 ADD R4，R1，R6 的情况）。很快，我们将会在第 5.13.2 节里处理这种语义冲突。

我们在图 5-6 中图示不同阶段的资源的新的组织结构。注意 ID/RR 和 WB 阶段里提到的寄存器堆是数据通路里的同一个逻辑元素。在图 5-6 中，两个阶段都包含了寄存器堆以清楚地展示它们所需的资源。这里有一些关于我们的流水线处理器设计值得一提的事情。在稳定状态里，有五条不同的指令，分别在流水线的不同阶段里处理。我们知道，在 LC-2200 的简单设计里，有限状态机要穿过诸如 ifetch1、ifetch2 等这样的瞬状态。在一个给定的时钟周期内，处理器处在恰好一个状态里。而在流水线实现里，处理器同时属于由流水线的不同阶段表示的状态里。每个指令要用五个时钟周期来执行。每个指令都从 IF 阶段开始进入流水线，在 WB 阶段之后引退（即成功完成）。在理想情况中，流水线处理器每个时钟周期引退一条指令。因此，流水线处理器的等效 CPI 为 1。观察图 5-6，你可能会猜想 MEM 阶段应当是最慢 [179]
的。但是，这个问题的答案很复杂，我们将把关于在现代处理器内减小时钟周期时间的讨论留到第 5.15 节。

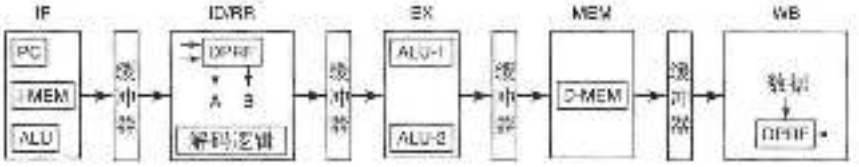


图 5-6 各个阶段的硬件资源的组织

内存系统对于流水线处理器的性能来说至关重要。内存访问时间是流水线处理器的延时中最重要的部分。为了隐藏此种延时，处理器应用了缓存。回忆一下第 2 章中提到的工具箱和工具托盘的类比。我们提到过，寄存器起到了工具托盘的作用，这体现在我们通过 load 指令，显式地把处理器需要的内存中的值加载到寄存器中。类似地，缓存起到了隐式的工具托盘的作用。换句话说，每当处理器从内存中取出点什么（指令或者是数据）时，它隐式地把它放入一个处理器的高速的存储区域，这个存储区域被称作缓存。通过给内存里的内容创建隐式的拷贝，处理器随后可以重用该内存地址里的值，而不需要再次从内存里获取。我们将在第 9 章讨论更多关于缓存的细节，包括它对流水线处理器实现的影响。而对于现在的关于流水线处理器实现的讨论，我们可以简单地认为缓存可以隐藏内存延时，让流水线处理器实现变为可行。尽管我们有缓存和高速寄存器，组合逻辑（ALU、多路选择器、解码器等）的延时仍然远远小于访问缓存以及通用寄存器的延时。出于确保所有阶段的延时大体相当的目的，现代处理器的实现包含了远远多于五个阶段。例如，访问存储元素（缓存、寄存器堆）也许会在流水线中用掉多个时钟周期。我们会在第 5.15 节讨论这类问题。因为现在是对处理器流水线实现的初次介绍，我们尽量让讨论简洁一些。

5.12 针对流水线的体系结构与实现

针对流水线的体系结构设计的几个关键点如下：

- **需要一个容易解码的指令格式** 这个属性使得实现可以在指令完全解码之前就做一些决定。一个对称的指令格式是一个具有此属性的例子。这种格式确保了指令里的特定 [180]
字段（如寄存器描述符、偏移量的大小和位置等）的位置对于一般类指令（如 LC-2200

中的R类指令和I类指令）来说保持不变，不依赖于具体是什么指令。我们已经看到，这是我们在LC-2200的ID/RR阶段里利用到的一个关键属性。

- **需要确保每个阶段的工作量相同** 这个属性确保了最理想的时钟周期时间，因为流水线里最慢的环节决定了它。

图5-6a展示了流水线的LC-2200实现的完整数据通路。

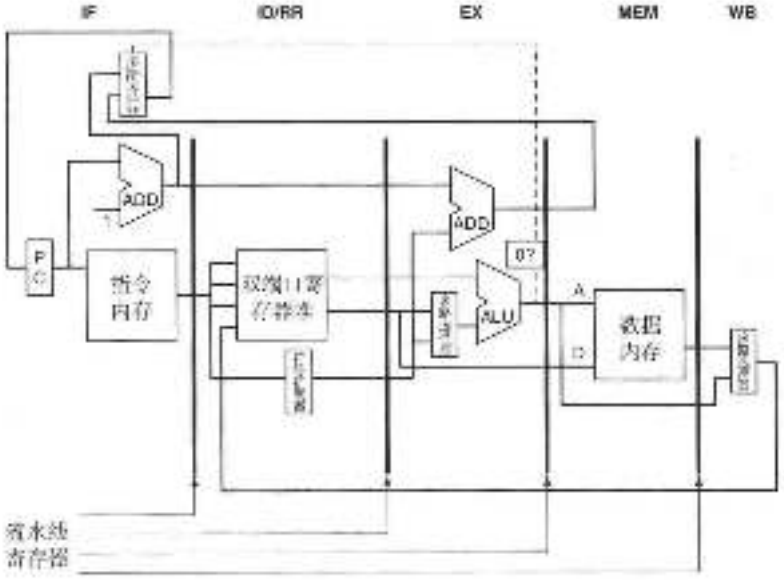


图5-6 a) 流水线的LC-2200的数据通路，展示了各阶段之间的所有联系，以及各阶段所需用到的资源

5.12.1 指令穿过流水线的过程详解

在本小节中，我们将会跟踪一条指令穿过五阶段流水线的全过程。我们给相邻阶段之间的寄存器分别取专有的名称，如图5-6b所示。

[181]

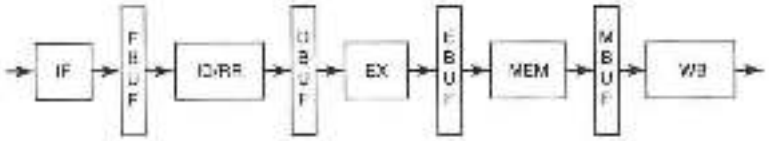


图5-6 b) 流水线寄存器的专有名称。每个阶段输出处的流水线寄存器包含着指令在该阶段执行的部分结果

表5-2总结了每个阶段之间的流水线寄存器的功能。

让我们来考虑一下ADD指令，它具有以下的句法和格式：

```
ADD   Rx, Ry, Rz;          Rx ← Ry + Rz
```

每个阶段分别执行下面总结的操作，向着完成这条加法指令的目标做贡献。

IF阶段（第1个时钟周期）：

```
I-MEM[PC] -> FBUF                    // 在PC这个内存地址存储的指令被取到FBUF
```

（其实 FBUF 不质「就是 IR）；之后 FBUF 的内容显示在图 5.6c) 中。

```
PC+1->PC                                    // PC 加一

ID/RR 阶段（第 2 个时钟周期）:
DPRF[FBUF[Ry]] -> DBUF[A];                  // 把 Ry 读入 DBUF[A]
DPRF[FBUF[Rz]] -> DBUF[B];                  // 把 Rz 读入 DBUF[B]
FBUF[OPCODE] -> DBUF[OPCODE];               // 把操作码从 FBUF 拷贝到 DBUF
FBUF[Rx] -> DBUF[Rx]                        // 把 Rx 寄存器描述符从 FBUF 拷贝到 DBUF

EX 阶段（第 3 个时钟周期）:
DBUF[A] + DBUF[B] -> EBUF[Result];          // 完成加法
DBUF[OPCODE] -> EBUF[OPCODE];               // 把操作码从 DBUF 拷贝到 EBUF
DBUF[Rx] -> EBUF[Rx];                       // 把 Rx 寄存器描述符从 DBUF 拷贝到 EBUF

MEM 阶段（第 4 个时钟周期）:
EBUF -> MBUF;                               // MEM 阶段对于 ADD 指令的执行毫无贡献，所
                                            以简单地把 EBUF 拷贝进 MBUF。[1]

WB 阶段（第 5 个时钟周期）:
MBUF[Result] -> DPRF[MBUF[Rx]];             // 把加法运算的结果写入由寄存器描述符 Rx 描
                                            述的寄存器中
```

182

表 5-2 流水线缓冲器及其内容

名称	各阶段的输出	内容
FBUF	IF	主要包含由内存读取的指令
DBUF	ID/RR	解码的 IR 和从寄存器堆读取到的值
EBUF	EX	主要包含 ALU 运算结果加上指令的其他部分，取决于具体指令
MBUF	MEM	如果指令不是 LW 或者 SW 的话与 EBUF 相同。如果指令是 LW 的话则缓冲器里包含内存指定地址中的内容

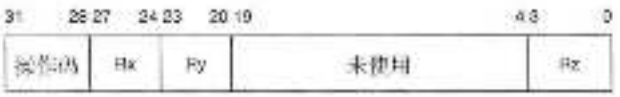


图 5.6 c) LC-2200 加法指令的句法和格式

例 5-8 只考虑 ADD 指令，定量分析流水线相邻阶段的各个缓冲区的大小。

答：

FBUF 的大小（与 LC-2200 的一条指令的大小相同）= 32 位

DBUF 的大小：

DBUF[A] 中存储为 Ry 寄存器的内容的大小 = 32 位

DBUF[B] 中存储的 Rz 寄存器的内容的大小 = 32 位

[1] 此处原文为 DBUF，应为 EBUF。——译者注

DBUF[opcode] 中的操作码	– 4 位
DBUF[Rx] 中的 Rx 寄存器描述符的大小	– 4 位
总大小（所有字段之和）	– 72 位
EBUF 的大小：	
EBUF[result] 中的加法结果	– 32 位
EBUF[opcode] 中的操作码	– 4 位
EBUF[Rx] 中的 Rx 寄存器描述符的大小	– 4 位
总大小（所有字段之和）	– 40 位
MBUF 的大小（与 EBUF 相同）	– 40 位

183

5.12.2 流水线寄存器的设计

尽管在前一节中我们看过了一条指令是如何穿过流水线的，应当说明流水线的每个阶段在每个时钟周期里都在处理不同的指令。读者应当弄懂针对 LC-2200 的不同指令在流水线每个阶段所采取的行动。（参见本章结尾处的练习题。）这个练习类似于我们在第 3 章中对 LC-2200 的串行实现的有限状态机的设计。在设计完成之后，每个阶段的流水线寄存器的尺寸可定为任何指令执行所需的流水线寄存器尺寸的最大值。在 ID/RR 输出处的流水线寄存器将会有最多的内容，因为我们还不知道指令是什么。对于流水线寄存器的这些内容的解读取决于这是哪个阶段，以及该阶段在处理的指令的操作码。

一个通用的流水线寄存器布局如图 5-6d 所示。操作码永远占用着每个流水线寄存器的相同位置。在每个时钟周期里，每个阶段依据操作码解读流水线寄存器的其余部分，并采取相应的数据通路动作（类似于我们在 5.12.1 节里为 ADD 指令所做的详细说明）。

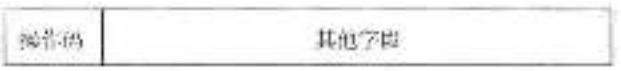


图 5-6 d) 流水线寄存器的通用布局

例 5-9 为 LC-2200 设计 DBUF 流水线寄存器。不要试图考虑重新使用寄存器的不同字段的方式来优化设计。

答：

DBUF 有以下字段：

操作码（所有指令都需要）	4 位
A（R 类指令需要）	32 位
B（R 类指令需要）	32 位
偏移量（I 类和 J 类指令需要）	20 位
PC 的值（BEQ 需要）	32 位
Rx 寄存器描述符（R、I、J 类指令需要）	4 位

DBUF 流水线寄存器的布局：



184

5.12.3 各个阶段的实现

在某种意义上，设计和实现一个流水线处理器可能比实现一个非流水线处理器还要简单。这是因为流水线的实现将设计给模块化了。这样的模块化带来了和把一个大的软件系统分解为若干个小模块一样的好处。正如大软件团队开发复杂软件系统（如微软 Word）时的情况一样，流水线处理器的模块化使得多个独立的硬件团队协同实现处理器成为可能，其中每个团队负责处理器的特定阶段。流水线寄存器的布局和解读类似于大型软件系统里定义良好的组件间接口。在完成了流水线寄存器的布局和解读之后，我们就在完全隔绝于其他阶段的情况下完成数据通路操作。更进一步地说，因为每个阶段的数据通路操作都在一个时钟周期中完成，整个阶段的设计完全是组合逻辑。在每个时钟周期开始时，每个阶段解读输入的流水线寄存器，用组合逻辑来进行数据通路操作，再把数据通路操作的结果写入它的输出流水线寄存器。

例 5-10 设计并实现 LC-2200 指令集的流水线中的 ID/RR 阶段的数据通路。你可以使用任何可用的逻辑设计来完成此题。

答：

图 5-6、图 5-6a 和图 5-6b 是解决此题的基础。数据通路元素已经放置好。根据指令的格式，读取寄存器文件得到的值会被放入 DBUF 的相应字段中；指令的操作码和操作码字段被从 FBUF 拷贝到 DBUF 中；PC 的值被放入 DBUF 的相应字段中。剩下部分留给读者作为练习。

5.13 冒险

尽管二明治流水线对于流水线处理器来说是个很好的类比物，指令流水线中还是有一些额外的问题，这些问题让它的设计变得复杂。这些问题被统称为流水线冒险。具体来说，有三类冒险：**结构性冒险、数据冒险和控制冒险。** 185

我们很快就会看到，所有的冒险的效果都一样，也就是它们会降低流水线的效率。换句话说，流水线会每个时钟周期执行少于一条指令。但是，回想一下，流水线是同步的，也就是说每个时钟周期每个阶段都在处理上个阶段刚刚放进流水线寄存器的指令。正如水管中的气泡，如果一个阶段无法发送一个合法指令给下一个阶段，它就应当把一个等价于“气泡”，即一个什么都不做的空指令传递过去。我们把这个称作 NOP（no-operation，无操作）指令。

我们将会在处理器的指令集中加入一种空指令。在接下来的关于冒险的讨论中，我们将讨论硬件如何根据遇到的冒险自动产生这些 NOP。但是，这不是唯一的办法。通过把流水线暴露给软件，我们可以让系统软件——即编译器——负责在代码中添加这些 NOP。我们将在 5.13.4 节中再讨论这种可能性。

另一种理解流水线中气泡效果的方式是认识到指令的平均 CPI 是大于 1 的。有必要对于使用 CPI 的记号加上一个警告：回忆一下程序的执行时间，如果以时钟周期数计的话，是 CPI 与总共执行的指令数的乘积。因此，CPI 本身并不能完全说明某体系结构或者它的某一实现效果有多好。实际上，编译器和体系结构互相合作，共同确定了程序的执行时间。比方说，一个编译器生成的未优化代码的 CPI 也许会比优化后的代码低一点，但是执行时间可能要比优化后的代码长很多。原因是编译器的优化阶段可能已经去掉了大量无用指令，也就减少了总共执行的指令数。但这可能是以增加了三类冒险出现的次数为代价的，因此，优化指令的平均 CPI 也就上升了。但是，优化代码的净效果仍然可能是执行时间减少。

5.13.1 结构性冒险

我们以前在提及不同阶段并发操作的可用硬件资源的限制时提到过结构性冒险。例如说，在非流水线版本里单一数据总线就是流水线实现的一个结构性冒险。类似地，单一的 ALU 是另一个结构性冒险。对于此种问题有两种解决方案：一种是接受它的存在，一种是修复它。如果估计这种冒险很罕见（比如只针对某个特定的同时穿过流水线的指令组合），那么可能不浪费硬件资源来修复是比较划算的。作为一个例子，我们不妨假设我们的尖头发老板[⊖]告诉我们，我们必须只能在 EX 单元里使用一个 ALU。那么，每一次遇到 BEQ 指令，我们得花两个时钟周期在 EX 阶段里，以处理所需的两个算术操作，一个用于比较，另一个用于计算地址。当然，得设法让前后的阶段知道 EX 阶段偶尔会花两个时钟周期进行操作。因此，EX 单元应当告诉它前面的阶段（即 IF 和 ID/RR）在下个时钟周期别发送新指令过来。简单地说，就是需要一个反馈线路，每个之前的阶段通过反馈线路里的内容来确定是否应当在当前阶段里暂停，还是应该做点有用的事情。如果一个阶段决定在某个时钟周期暂停，它只需不改变输出寄存器的值就好了。

186

而之后的阶段需要用跟之前阶段不同的方式来处理。具体来说，EX 阶段将会在它的输出缓冲器的操作码字段里写入一个 NOP 操作码。NOP 指令是让处理器执行一条对实际要执行的程序没有影响的空指令的便利方式。通过 NOP 指令，我们在流水线里 BEQ 指令和之前一条指令之间引入了一个气泡。

图 5-7 以一系列时间点的示意图展示了 BEQ 指令的执行过程。我们从第 2 个时钟周期开始，此时 BEQ 指令处在 ID/RR 阶段。反馈线路上的值为 STAY（二进制的 1），告诉之前的阶段留在同一条指令，而不要在时钟周期结束时把指令发送出去。流水线中这样的气泡的存在降低了流水线的效率，因为气泡使得流水线的等效 CPI 升到了 1 以上。

另一方面，如果这样的效率损失被认为是不可接受的，我们可以通过添加硬件来解决结构性冒险。这就是我们在之前讨论在 EX 阶段添加一个额外 ALU 来解决结构性冒险时所做的事情。

这里有三个关于术语的备注：

- 指令不能进入下一阶段的时候称作指令被拖延了。
- 拖延的结果是流水线中被引入一个气泡。
- 气泡在流水线中表现为一个 NOP 指令。一个执行 NOP 指令的阶段在这个时钟周期里什么都不做。它的输出缓冲器与上一个时钟周期相比不会改变。

你可能会发现，我们在本书中，拖延、气泡和 NOP 可互换使用，它们都表示相同的东西。

5.13.2 数据冒险

考虑图 5-8a。在第一个式子中，注意两条指令在程序中出现的顺序。I_1 读取寄存器 R2 和 R3 的值，然后将加法的结果写入寄存器 R1。后一条指令 I_2 则读取 R1（刚刚由前一条指令 I_1 写入）和 R5 中的值，并把加法的结果写入 R4。这种情况就是数据冒险，因为这两个指令之间存在依赖关系。更加细分的话，这种冒险称作写后读（RAW，Read After Write）数据冒险。当然了，这两个指令并不需要严格地挨在一起，只需要按照执行顺序执行的时候存在对数据的依赖关系，就构成了一个数据冒险。

⊖ 漫画《呆伯特》中主人公的老板。——译者注

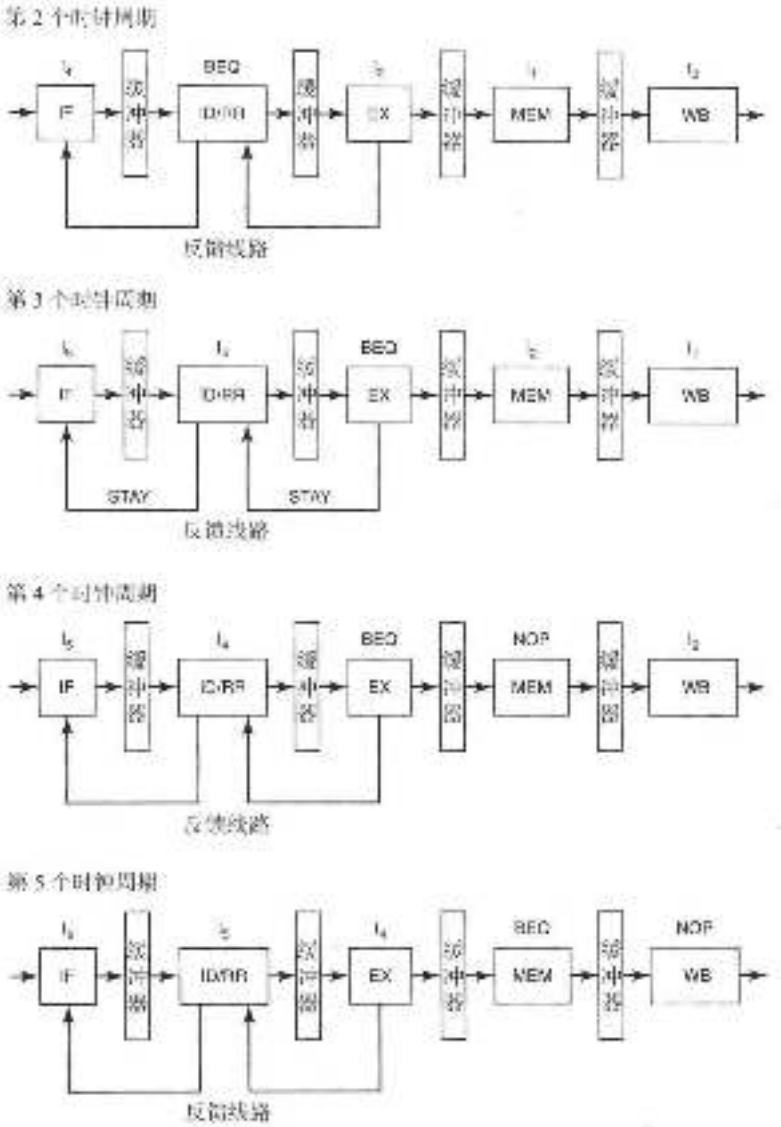


图 5-7 结构性冒险图解

还有另外两种类型的数据冒险。图 5-8b 中的情形被称作读后写（WAR，Write After Read）数据冒险，I_2 向寄存器写入一个值，而 I_1 从同一个寄存器读出一个值。图 5-8c 中的情形被称作写后写（WAW，Write After Write）数据冒险，I_2 要写入的寄存器也是前一条指令写入的目标。

这里是个很好的强调硬件和软件之间联系的地方。RAW、WAR 和 WAW 冒险是处理器流水线的属性。但是，这些冒险在程序执行时的出现依赖于程序本身的固有属性。导致这些冒险的程序属性分别是流依赖、反依赖以及输出依赖。

我们定义这些术语如下：

1）语句 S2 在满足下列两个条件的情况下被称作流依赖（或真依赖）于 S1：

• S1 在执行顺序中位于 S2 之前

• S1 修改了一个 S2 读取的资源[⊖]

2）语句 S2 在满足下列两个条件的情况下被称作反依赖于 S1：

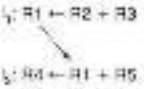
• S1 在执行顺序中位于 S2 之前

• S2 修改了一个 S1 读取的资源

3）语句 S2 在满足下列两个条件的情况下被称作输出依赖于 S1：

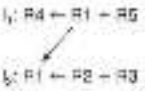
• S1 在执行顺序中位于 S2 之前

• S1 和 S2 修改了同一个资源



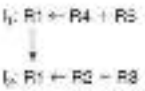
(5-7)

图 5-8 a) 写后读冒险



(5-8)

图 5-8 b) 读后写冒险



(5-9)

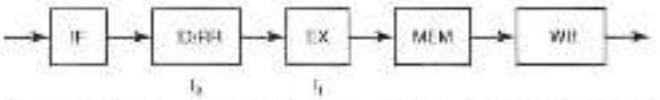
187 ~ 189

图 5-8 c) 写后写冒险

编译器进行数据依赖分析来识别这些程序属性，然后如果需要的话，通过重新排列指令来减少数据依赖对流水线处理器带来的负面效果。

如果指令按照程序顺序如同在一个非流水线处理器那般一个接着一个执行的话，这些数据依赖不会造成任何问题。但是，它们可能会在流水线处理器中引起问题，这将会在下一段进行解释。我们发现，对于我们考虑的简单流水线，WAR 和 WAW 冒险不会造成什么问题，我们将会在随后讨论解决它们的方案。我们首先来处理写后读冒险。

写后读冒险 让我们来跟踪图 5-8a 中式（5-7）所表示的指令序列流过流水线的过程：



当 I_1 在流水线的 EX 阶段时，I_2 正处在 ID/RR 阶段，即将读取寄存器 R1 和 R5 的值。这里就有问题了，因为 I_1 还没有计算出 R1 的新值来。实际上，I_1 只有执行到 WB 阶段才会把新计算的值写回到 R1 寄存器中。如果 I_2 被允许读取 R1 中的内容，如上图所示的情况，就会让整个程序的执行出错。我们把这种程序的本意与实际执行结果不符合的情况称作语义不一致。在非流水线的实现里绝对不会有这样的问题，因为它一次只执行一条指令。

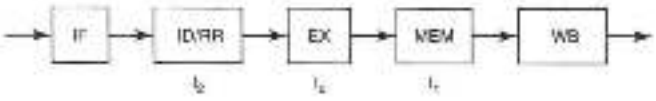
如果两条指令并不是一个紧接着一个的话，问题也许没那么严重。

比如说，考虑以下的序列：

I_1: R1 ← R2 + R3
I_2: R8 ← R6 − R7
I_3: R4 ← R1 − R5

⊖ 此处原文把 S2 误写成了 S1。——译者注

这种情况下，流水线看起来如下图：



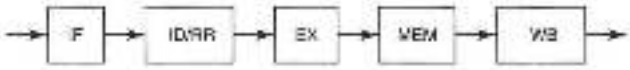
190

I_1 已经执行完毕。但是，只有在 I_1 达到 WB 阶段时才会把新的 R1 的值写入寄存器。因此，如果 I_1 之后的三条指令中任一条存在写后读冒险，就会导致语义不一致。

例 5-11 考虑下面四个指令的序列：

I_1: LW R1, 内存地址

<两条不相关指令>

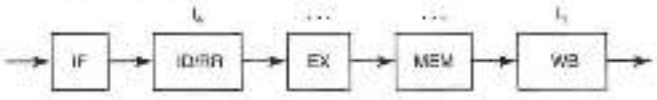


I_2: R4 ← R1 + R5

I_1 和 I_2 被两条不相关指令间隔，如上图所示。流水线执行此指令序列需要插入多少个气泡？

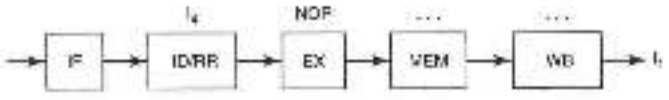
答：

I_1 达到 ID/RR 阶段时流水线的状态如下图：



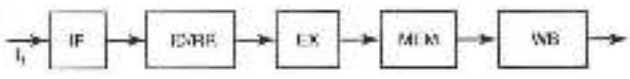
I_2 要到当前周期结束时才会把值写入 R1 中。因此，I_2 在这个时钟周期里读取寄存器的话是不能得到正确结果的。

因此，要有 1 个周期的延时，也就需要插入一个气泡（由 ID/RR 阶段传递给 EX 阶段一个 NOP 指令）。



例 5-12

I_1: R1 ← R2 + R3
I_2: R4 ← R4 + R3
I_3: R5 ← R5 + R3
I_4: R6 ← R1 + R8



如上图所示，指令序列 I_1 到 I_4 即将进入该五级流水线。

a. 在下表中写出指令逐步流过流水线直到全部四条指令都执行完毕且引退的整个过程。从流水线中引退是指一条指令不处在五个阶段中的任何一个。 191

答：

周期数	IF	ID/RR	EX	MEM	WB
1	I_1	–	–	–	–
2	I_2	I_1	–	–	–
3	I_3	I_2	I_1	–	–
4	I_4	I_3	I_2	I_1	–
5	–	I_4	I_3	I_2	I_1
6	–	I_4	NOP	I_3	I_2
7	–	–	I_4	NOP	I_3
8	–	–	–	I_4	NOP
9	–	–	–	–	I_4

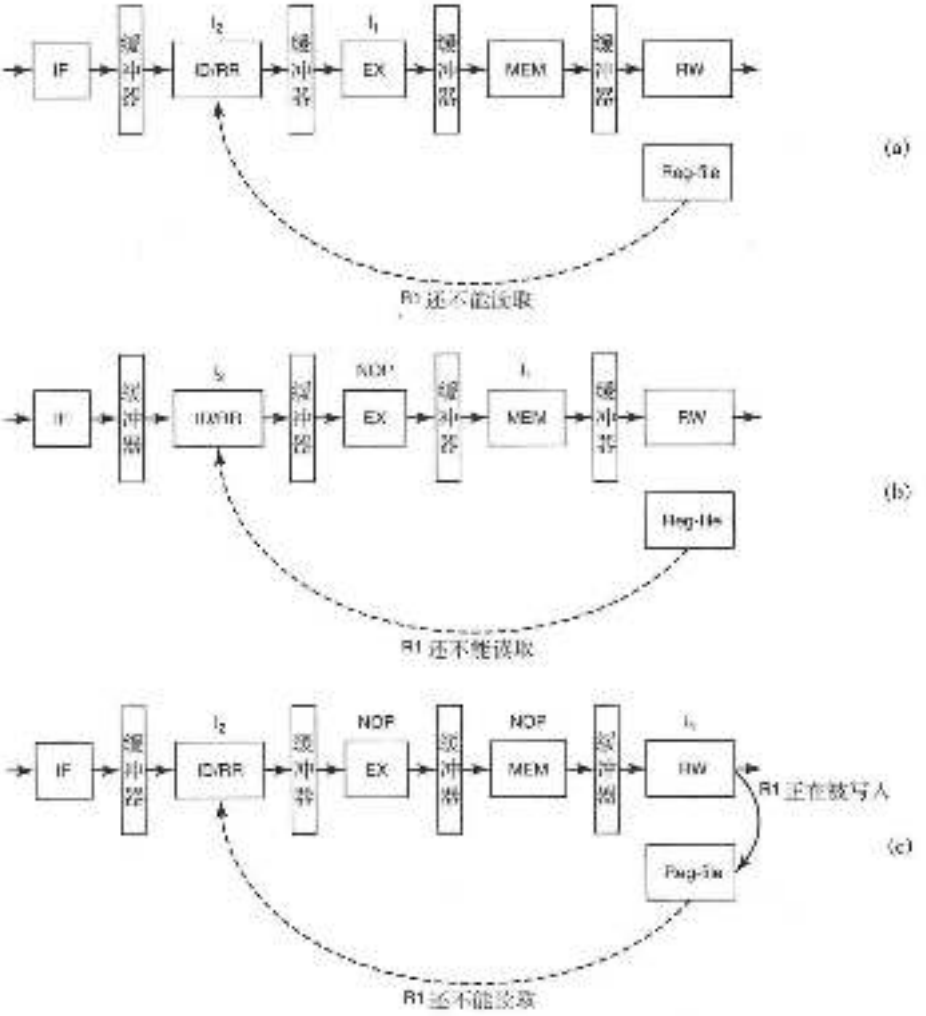
b. 假设程序员为各这四条指令，在之前的执行中的平均 CPI 是多少？

答：花了六个时钟周期执行了四条指令，因此，这几条指令的平均 CPI 是

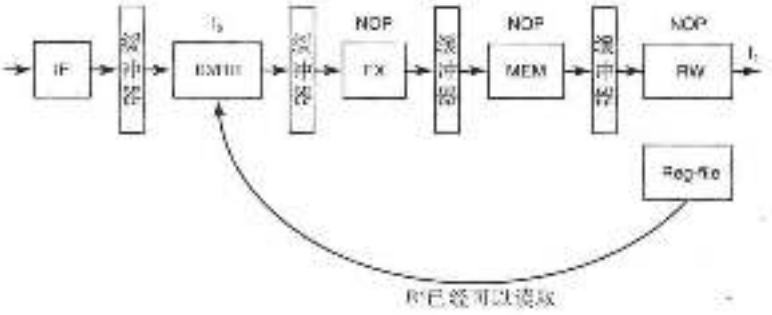
平均 CPI = 9/4 = 2.25

解决写后读数据冒险：数据前递 解决这个问题的一个简单方法类似于解决结构性冒险，我们能简单地把导致写后读冒险的指令拖延在 ID/RR 阶段直到寄存器的值已经可用为止。在图 5-8a 的式 (5-7) 中展示的情形里，ID/RR 阶段保持 I_2 三个时钟周期，直到 I_1 从流水线引退。在这三个时钟周期内，气泡（以 ID/RR 发送 NOP 指令的形式）被塞进流水线里。由于同样的原因，之前的阶段（IF）被告知停留在同一条指令上不取新指令。

192 图 5-9 中一系列的图示展示了流水线由于图 5-8a 中式（5-7）的数据冒险所引起的延时。



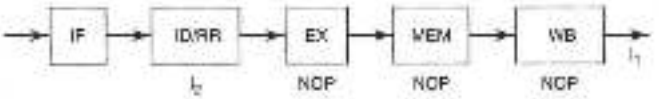
193 图 5-9 图 5-8 的式（5-7）中的写后读冒险



(d)

图 5-9 (续)

在下图中展示的时钟周期里，IF 阶段将会正常执行，开始读取新的指令。



ID/RR 阶段需要硬件帮助以确定它需要拖延。检测这一点所需的硬件十分简单。在图 5-10 中所展示的寄存器堆里，B 位就是寄存器的忙位，每个寄存器都有一个这样的位。ID/RR 阶段确定指令的目标寄存器之后就在寄存器堆里把相应寄存器的 B 位设为 1。图中的 I_1 应当已经把 R1 的 B 位设成 1 了。WB 阶段则负责在把新值写入寄存器堆的同时把对应的 B 位设回 0。因此，当 I_2 到达 ID/RR 阶段时，它会发现 R1 的 B 位被设置了，也就会自己一直拖延到三个时钟周期之后 B 位被清零为止。

寄存器堆

	B
	0
	0
	0
	0
	0
	0
	0
	0

图 5-10 寄存器堆里每个寄存器带有一个忙位

194

让我们来看看如何能摆脱写后读冒险带来的拖延。我们可能没法完全摆脱掉这种拖延，但是当然可以通过略微增加硬件的复杂性来减小化拖延的次数。原始的算法以它的发明者 Tomasulo 命名，在 20 世纪 60 年代的 IBM 360/91 处理器上初次使用。想法很简单，概括来说，给寄存器生成新值的阶段检查一遍是否有其他阶段正在等待这个值。如果是，它就把这个值前递给需要这个值的阶段。[⊖]

考虑我们的简单流水线，唯一会读取寄存器的阶段是 ID/RR 阶段。让我们来检查一下需要做什么来使用数据前递。我们给寄存器堆里面的每个寄存器增加一个称作 RP（Read Pending，等待读取）的位（参见图 5-11）。

寄存器堆

	B	RP
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0

图 5-11 寄存器堆里每个寄存器带有一个忙位和一个等待读取位

当指令进入 ID/RR 阶段时，如果它试图读取的寄存器的忙位被设置成 1 了，那么它就把该寄存器的等待读取位也设置成 1。如果 EX、MEM 或者 WB 阶段中的任何一个发现它计算新值的目标寄存器的等待读取位是 1，它就把生成的值提供给 ID/RR 阶段。由于必须得有线缆从这些阶段走回 ID/RR 阶段，硬

⊖ 我们所讲到的为简单流水线避免写后读冒险而采用的算法是受 Tomasulo 算法启发得到的，但是远不及原始的算法那么具通用性。

件就复杂起来了。人们可能会问，为什么不直接让这些阶段在等待读取位是1的时候把值写回寄存器堆呢？原则上说，当然可以这么做；但是，正如我们之前所说的，写入寄存器堆以及将忙位清零是WB阶段的职责。如果让所有的阶段都能写入寄存器堆的话，硬件的复杂性就会增加。而且，这样增加的额外硬件并不能带来任何性能的提升，因为在ID/RR阶段的指令需要数据时反正也能通过数据前递获得。

回头再看看图5-8a中的写后读冒险，我们看到数据前递能完全消除气泡，如图5-12所示。

考虑以下指令序列：

I_1

I_2

I_3

195 I_4

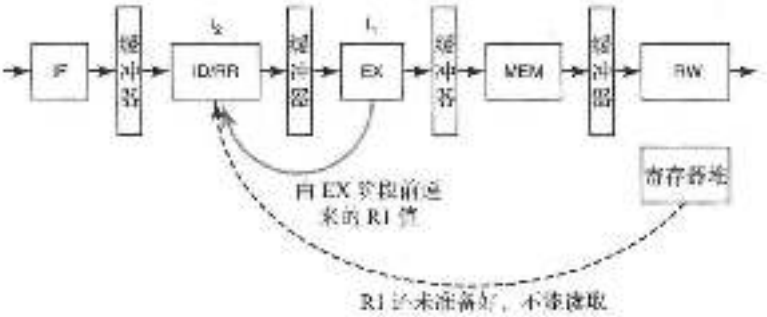


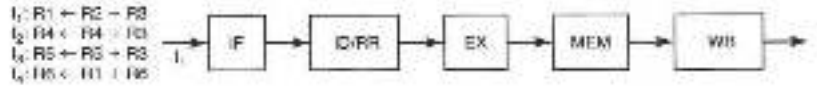
图5-12 用数据前递[⊖]来解决图5-8a中的写后读冒险

这里如果I_1是一条为某个寄存器产生新值的算术/逻辑指令，而随后的三条指令（I_2、I_3、I_4）中的任何一条需要该值，则前递将会消除由写后读冒险产生的拖延。表5-3总结了带前递和不带前递的流水线对于图5-8a中的写后读冒险所需要插入的气泡个数与I_1和I_2之间不相关指令条数的关系。

表5-3 图5-8a中写后读冒险产生的气泡个数

I_1和I_2之间的无关指令条数	不用前递时的气泡个数	用前递时的气泡个数
0	3	0
1	2	0
2	1	0
3或更多	0	0

例5-13 这里与指令序列与例5-12中的一样。假设采用了数据前递，在下表中给出指令的执行过程。这些指令的平均CPI是多少？



196

答：参见表5-3，使用数据前递以后，算术/逻辑指令使来自写后读冒险将不需要插入气泡，因为

⊖ 前递听起来很反直觉，因为箭头在向回指！但是这个可以这么理解：在程序的执行顺序里，I_1是在I_2的前面，而已经前递给了I_2。

从每个阶段都可以读指令序列之前的阶段。因此，从下表可以看出来，给定的执行序列里没有气泡。

周期编号	IF	ID/RR	EX	MEM	WB
1	I_1	–	–	–	–
2	I_2	I_1	–	–	–
3	I_3	I_2	I_1	–	–
4	I_4	I_3	I_2	I_1	
5	–	I_4	I_3	I_2	I_1
6	–	–	I_4	I_3	I_2
7	–	–	–	I_4	I_3
8	–	–	–	–	I_4

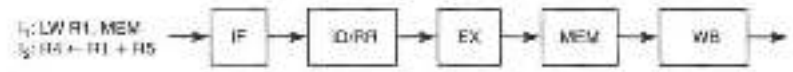
平均 CPI=8/4=2

处理读取内存的指令引入的写后读冒险 读取内存的指令也会引入数据冒险。考虑以下指令序列：

$$\begin{array}{l} I_1\text{: LW R1, 0(R2)} \\ \quad\searrow \\ I_2\text{: ADD R4, R1, R4} \end{array} \qquad (5\text{-}10)$$

在这种情况下，R1 的新值直到 MEM 阶段才可用。因此，如果读存指令读取的值在紧接着的指令中会用到，就算用上前递，一个时钟周期的拖延也无可避免，如式（10）所示。例 5-14 会详细介绍处理读取内存的指令在流水线中引入的气泡。

例 5-14 考虑以下指令序列：



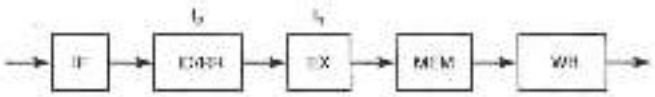
197

a. 如图，I_2 紧跟着 I_1。假设各个阶段都能前递到 ID/RR 阶段，之前的执行将会产生多少个气泡？

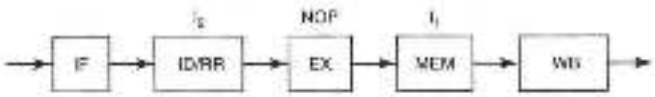
b. 如果没有寄存器前递的话，前面的执行将会产生多少个气泡？

答：

a. I_1 达到 ID/RR 阶段时的流水线状态如下：



I_1 只有在 MEM 阶段结束时才有 R1 的值。因此，有一个周期的延时，就好像一个气泡（ID/RR 阶段传递给 EX 阶段的 NOP 指令），如图：



MEM 阶段将会同时将它的结果写入它的输出缓冲器（MBUF）并将从内存读到的给 R1 寄存器的值传给 ID/RR 阶段，使得 ID/RR 阶段可以把该值（属于 I_1）写入 ID/RR 阶段的输出的流水线寄存器（DBUF）里。因此，之后流水线就没有延时了，尽管 I_1 只在 WB 阶段结束时将值写入 R1。下图展示了这两条指令在流水线里的执行情况。

周期编号	IF	ID/RR	EX	MEM	WB
1	I_1	–	–	–	–
2	I_2	I_1	–	–	–
3	–	I_2	I_1	–	–
4	–	I_2	NOP	I_1	–
5	–	–	I_2	NOP	I_1
6	–	–	–	I_2	NOP
7	–	–	–	–	I_2

198

因此，整个执行流程在有前推的情况下的气泡数是1。

b. 没有前进的话，I_1 在 I_1 把值写入R1之前是不能读取R1的。写入R1要等到WB阶段结束。注意写入R1的值只有在下一个时钟周期才可读取。下表展示了这两条指令在流水线里的执行情况。

周期编号	IF	ID/RR	EX	MEM	WB
1	I_1	–	–	–	–
2	I_2	I_1	–	–	–
3	I_3	I_2	I_1	–	–
4	I_4	I_3	NOP	I_1	–
5	–	I_2	NOP	NOP	I_1
6	–	I_2	NOP	NOP	NOP
7	–	–	I_2	NOP	NOP
8	–	–	–	I_2	NOP
9	–	–	–	–	I_2

因此，不同前进执行这两条指令所产生的气泡数=3。

表5-4总结了由于读取内存的指令产生的写后读冒险在有和没有数据前递的情况下，对于每种 I_1 和 I_2 之间的无关指令条数，分别引入的气泡个数。

表5-4 读内存指令引起的写后读冒险导致的流水线中的气泡

I_1 和 I_2 之间的无关指令条数	不用前递时的气泡个数	用前递时的气泡个数
0	3	1
1	2	0
2	1	0
3及以上	0	0

其他数据冒险的类型 其他两种冒险类型，即写后读和写后写冒险，对流水线处理器会造成各自特有的问题。但是，这些问题在影响流水线处理器的性能方面远不及写后读的问题严重。譬如说，读后写全然不是问题，因为需要读数据的指令已经在ID/RR阶段就把寄存器的值存到流水线缓冲器里了。对于写后写冒险，一个简单的解决方案是，如果一条指令将要把值写入寄存器，而它在ID/RR阶段时看到该寄存器的忙位为1时就推延它，直到把寄存器的忙位设置为1的指令在WB阶段把它清零为止。

让我们来理解以下写后写冒险的根源。写后写意味着一条指令要写入的寄存器将要被随

后的一条指令覆盖，并且两个指令之间并没有读取该寄存器的指令。我们可以安全地推论，前一次写入完全没有用。毕竟，如果在前一次写之后要有一次读取，那么就是写后读冒险，而不是写后写冒险了。这就提出了问题，为什么编译器要产生具有写后写冒险的代码呢？对于这个问题，有几种可能的答案。在之前，我们提及过导致流水线冒险的程序属性。流依赖是程序本身固有的属性，而反依赖和输出依赖主要是由编译器激进地使用寄存器存储程序变量而引入的。回忆一下，寄存器访问比内存访问快很多，外加寄存器的数量很有限。因此，编译器可能会重用寄存器去存储不同的程序变量，引入反依赖和输出依赖，从而导致流水线处理器里的写后读和写后写冒险。我们将在第 5.15.4 节中更加详细讨论这个问题，并且给出这个问题的可能的硬件解决方案。

5.13.3 控制冒险

这种冒险是指程序由于分支语句而打破了顺序执行。在基准测试程序中关于动态指令频率的研究表明，每 4 ~ 6 条指令中就有一条是条件分支指令。分支导致正常的控制流中断，因而对流水线处理器的性能有害。这个问题对于条件分支指令尤其严重，因为分支的结果通常直到流水线的很迟的地方才能被发现。

假设按照程序顺序进入流水线的指令流如下

```
BEQ
ADD
NAND
LW
...
...
```

一个保守的处理分支的方法是在解码阶段发现分支指令时就防止新指令进入流水线，直到分支被确定了，正常的执行再继续，可能是按顺序执行下去，也可能从跳转目标开始执行。图 5-13 展示了这样一个保守的做法下的指令流。对于 BEQ 指令，我们在 EX 阶段结束时知道它的结果。如果分支未被采取（也就是继续按着顺序执行），那么 IF 阶段已经有了从内存中读取的正确的指令，可以传递给 ID/RR 阶段。因此，我们将流水线推延一个周期，然后再继续执行图 5-13 中的 ADD 指令。但是，如果分支被采取，那么我们就不得不再从 BEQ 指令在 EX 阶段结束以后计算得到的 PC 地址开始取指令。因此，在第 4 个周期，我们开始读取分支目标处的正确的下一条指令。因此，如果分支被采取的话，根据前面的方案，就会有两个时钟周期的延时。

199 ~ 200

周期	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	ADD*	NOP	BEQ		
4	NAND	ADD	NOP	BEQ	
5	LW	NAND	ADD	NOP	BEQ

* 我们在取指令阶段实际上并不知道这条指令是什么。
- ADD 指令在 IF 阶段被推迟直到 BEQ 指令解决。之前的方案假设分支会不成功，而取下一条指令，让它在 BEQ 解决之后立刻进入 IF 阶段。如果分支成功的话，将会出现多一个时钟周期的延时来访问跳转到的指令。

图 5-13 一种处理分支指令的保守方案

例5-15 考虑以下指令序列：

```
        BEQ   L1
        ADD
        LW
        ...
L1      NAND
        SW
```

硬件采用保守策略来处理分支。

a. 假设分支未被采取，填下表以给出指令通过流水线的执行过程，直到3条指令成功地从流水线引退。这3条指令的实测CPI是多少？

b. 假设分支被采取，填下表以给出指令通过流水线的执行过程，直到3条指令成功地从流水线
[201] 引退。这3条指令的实测CPI是多少？

答：

a. 在分支未被采取时，给定的执行序列的时间表如下。注意ADD指令在IF阶段被流过了一个周期，然后BEQ指令在EX阶段结束时被解决了，ADD指令就可以继续向下传递了。

周期编号	IF	ID/RR	EX	MEM	WB
1	BEQ	–	–	–	–
2	ADD	BEQ	–	–	–
3	ADD	NOP	BEQ	–	–
4	LW	ADD	NOP	BEQ	–
5	–	LW	ADD	NOP	BEQ
6	–	–	LW	ADD	NOP
7			–	LW	ADD
8				–	LW

这三条指令的平均CPI是8/3 = 2.666。

b. 在分支被采取时，给定的执行序列的时间表如下。注意IF阶段读取的ADD指令在第4个时钟周期必须被转化为一个NOP，因分支被采取了。在第4个时钟周期开始要从分支的目标读取一条新的指令。正此原因导致了两个时钟周期。

周期编号	IF	ID/RR	EX	MEM	WB
1	BEQ	–	–	–	–
2	ADD	BEQ	–	–	–
3	ADD	NOP	BEQ	–	–
4	NAND	NOP	NOP	BEQ	–
5	SW	NAND	NOP	NOP	BEQ
6	–	SW	NAND	NOP	NOP
7	–	–	SW	NAND	NOP
8			–	SW	NAND
9				–	SW

[202] 这三条指令的平均CPI是9/3=3。

当然了，如果愿意为这个问题添加更多的硬件，我们可以做得更好一些。实际上，解决这个由流水线中的分支语句导致的麻烦有好几种方法。这是计算机体系结构社区里的热门研究领域。尤其是现代处理器可能采用深度流水线（有超过 20 个阶段），能够高速处理分支对于获得高性能来说极其重要。

在本章中，我们将会介绍解决该问题的诸多方案中的一部分。要想了解更多这方面的内容，你需要开始一门体系结构方面的高级课程。

处理流水线处理器中的分支 延迟分支和分支预测是解决流水线中此问题的两种方式。

1）**延迟分支** 这里的想法是假设分支指令之后的指令不管分支的结果，一定会被执行。这样就简化了硬件，因为就没有必要取消分支之后的指令。确保程序语义正确的责任就从硬件转移给了编译器。默认的方法是在软件（即程序在内存中的映像）中的每个分支语句之后加上一个 NOP。你也许会问这样的方法有什么用。答案是，聪明的编译器会对程序进行分析，找到一条有用而且放在分支语句之后不会影响语义的指令，以取代 NOP 指令。紧跟着分支指令的指令槽被称作延迟槽，相应地，这个技术被称作延迟分支。

图 5-14 和图 5-15 中的代码片段说明了一个编译器可以找到有用的指令塞进延迟槽里。在这个例子里，对于每个分支语句，无论分支的结果是什么都会执行紧跟着分支语句的指令。如果分支不成功，流水线就不用延时继续执行。如果分支成功了，那么（和分支预测失败情况一样）分支之后的除了第一条以外的指令就得被终止掉。

```
;     在此假设无论分支结果是否都执行分支后的指令
;     给一个地址为 a0 的 10 个元素的数组的每个元素加 7

addi  t1,  a0, 40              ; 当 a0=t1 时程序结束
loop: beq  a0, t1, done
      nop                      ; 分支延迟槽                    [1]
      lw   t0, 0(a0)                                           [2]
      addi t0, t0, 7
      sw   t0, 0(a0)
      addi a0, a0, 4                                           [3]
      beq  zero, zero, loop
      nop                      ; 分支延迟槽                    [4]
done: halt
```

列 5-14 延迟分支：延迟槽未被有 NOP

```
;     在此假设无论分支结果是否都执行分支后的指令
;     给一个地址为 a0 的 10 个元素的数组的每个元素加 7
addi  t1,  a0, 40              ; 当 a0=t1 时程序结束
loop: beq  a0, t1, done
      lw   t0, 0(a0)           ; 分支延迟槽                    [2]
      addi t0, t0, 7
      sw   t0, 0(a0)
      beq  zero, zero, loop
      addi a0, a0, 4           ; 分支延迟槽                    [1]
done: halt
```

图 5-15 延迟分支：把延迟槽里的 NOP 换成有用指令

参考图 5-14，编译器知道标注 [1] 和 [4] 的指令永远会被执行，因此，编译器一开始把 NOP 作为占位符放在这些槽里。在优化阶段中，编译器发现指令 [2] 从程序语义来说是良性[⊖]的，因为它只是把一个值加载进一个临时寄存器。因此，编译器把 NOP 指令 [1] 替换成图 5-15 中的读取内存的指令 [2]。类似地，循环里的最后一条分支语句是个无条件分支语句，不依赖于之前的 ADD 指令 [3]。因此，在图 5-15 中编译器把 NOP 指令 [4] 换成 ADD 指令 [3]。

有些机器甚至会使用多个延迟槽以在分支代价很高时提升流水线效率。延迟槽的数量越多，在成功分支时需要终止的指令条数越少。但是，这也增加了编译器寻找插入延迟槽的有用指令的负担，而且有时候根本就找不到这样的指令。

203
~
204

延迟分支似乎是个合理的想法，尤其是流水线不深（小于 10 个阶段）时，因为它把硬件简化了。但是，这个想法有几个问题。最明显的问题是它把微体系结构的细节暴露给了编译器作者，也就让编译器不仅仅要针对指令集，还要针对特定处理器实现。要不然，对于每一代处理器都得重写编译器的一部分。否则就得为了向后兼容而对微体系结构的变革加以限制。而且，现代的处理器采用深流水线。例如，最近的 Intel Pentium 处理器有超过 20 个流水线阶段。流水线变得更深，但是决定了程序中分支指令出现频率的基本块大小并没有发生变化。甚至，由于面对对象的编程，分支语句变得更加频繁了。因此，延迟分支在现代处理器实现中不受欢迎。

2）分支预测 这里的想法是假设分支的结果是某个方向，即使有分支指令也继续让指令进入流水线。例如，对于图 5-13 中的相同序列，让我们预测结果是不跳转（即顺序执行路径是赢家）。图 5-16 展示出采用这个预测情况下流水线中的指令流。在分支的结果被知道以后（BEQ 在 EX 阶段中），结果被传递回之前的阶段里。图 5-16 展示出结果符合预测时的美好情形，流水线完全没有延迟就继续执行了。

周期	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	NAND	ADD	BEQ		
4	LW	NAND	ADD	BEQ	
5	...	LW	NAND	ADD	BEQ

*我们在取指令阶段实际上并不知道这条指令是什么

图 5-16 分支预测

当然了，也有可能预测错误，我们得从这样的错误预测中恢复。因此，我们需要一种硬件机制来终止流水线中之前的阶段里执行了一部分的指令，而改从另一个分支开始读取指令。这种终止能力经常被称作冲刷。我们用一条额外的称作“冲刷”的反馈线路来实现这个硬件机制，如图 5-17 所示。在收到冲刷信号之后，IF 和 ID/RR 阶段放弃正在处理的部分执行的指令，向流水线中送入气泡。在正常执行开始前会插入两个气泡，即两个时钟周期的延迟，对应于图 5-17 中的 ADD 和 NAND 指令。

⊖ 这里是为了演示延迟分支的概念。严格来说，在循环结束以后执行读取内存指令有可能会访问到一个已经不存在的内存地址。

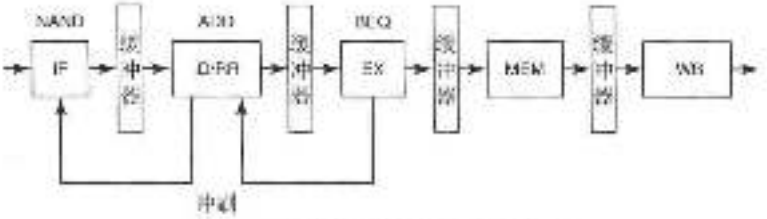


图 5-17 带着冲刷控制线路的流水线

[205]

例 5-16 考虑以下指令序列：

```
        BEQ    L1
        ADD
        LW
        ...
L1      NAND
        SW
```

假设使用了分支预测（预测分支不会被采取）。

a. 假设预测正确，填下表展现指令流过流水线的过程，直到 3 条指令成功引退为止。这 3 条指令的实际 CPI 是多少？

b. 假设预测错误，填下表展现指令流过流水线的过程，直到 3 条指令成功引退为止。这 3 条指令的实际 CPI 是多少？

答：

a. 在预测正确的情况下，给定序列的分时图表如下。分治预测逻辑把即序执行的指令吸进流水线，而且它们都成功执行完毕。

周期编号	IF	ID/RR	EX	MEM	WB
1	BEQ	–	–	–	–
2	ADD	BEQ	–	–	–
3	LW	ADD	BEQ	–	–
4	–	LW	ADD	BEQ	–
5	–	–	LW	ADD	BEQ
6	–	–	–	LW	ADD
7	–	–	–	–	LW

[206]

第 7 周期结束时，3 条指令完成了执行，因此平均 CPI 是 7/3=2.333。

b. 在预测错误的情况下，给定序列的分时图表如下。注意，ADD 和 LW 指令在 BEQ 确定分支的结果被错误预测时就被从各自所在的阶段给冲刷掉了。这就是为什么在第 4 个周期里 ADD 和 LW 被替换成了 NOP 指令。PC 在 BEQ 指令的 EX 周期（第 3 个周期）结束时被设置成跳转地址，以便第 4 个时钟周期时可以开始读取正确的指令。

平均 CPI 是 9/3=3。

注意平均 CPI 和我们之前不同分支预测且分支被采取时的平均 CPI 相同。

周期编号	IF	ID/RR	EX	MEM	WB
1	BEQ	–	–	–	–
2	ADD	BEQ	–	–	–
3	LW	ADD	BEQ	–	–
4	NAND	NOP	NOP	BEQ	
5	SW	NAND	NOP	NOP	BEQ
6		SW	NAND	NOP	NOP
7	–	–	SW	NAND	NOP
8			–	SW	NAND
9					SW

你也许会好奇，看起来分支被采取和不被采取的概率差不多，我们怎么能预测分支的结果呢？事实上，程序具有很多能帮助预测的结构。比如说，一个循环通常来说循环结束时会有一个条件分支语句来控制是返回循环的开头还是跳出循环。我们能立即看出，这个条件分支的结果严重偏向于跳回循环开头。因此，分支预测技术就依赖于这种程序的结构属性。

人们投入了相当多的精力来研究程序中分支的属性，并设计支持它们的预测方案。我们先前已经提到，循环和条件语句是产生分支语句的高阶结构。一种分支预测的手段是如果目标地址小于当前 PC 值，就预测分支会被采取。反过来，如果目标地址大于当前 PC 值，就预测分支不会被采取。这个策略背后的理由是循环一般来说都用到一个向后的分支语句以返回循环的头部（也就是回到一个较低的地址），而且这种事情发生的概率更高，因为循环会被多次执行。另一方面，向前的跳转通常对应着条件语句，它们相对而言较少被采取。

207

分支预测处在编译器的处理范围内，它以程序分析为基础进行。ISA 必须提供一种让编译器把预测传达给硬件的机制来支持编译器。因此，现代处理器往往给大部分以至全部的分支指令提供两个不同的版本，区别就在于是预测分支是被采取还是不被采取。编译器会针对给定的分支指令选择最适合它的需求的版本。

3）**带分支目标缓存的分支预测** 这个方法基于之前我们提到的分支预测。它使用一个被称作分支目标缓存（BTB，Branch Target Buffer）[⊖]的硬件装置来改进分支预测。BTB 本质上就是一张表，其中每一项包含三个字段，如图 5-18 所示。

分支指令地址	选取 / 未选取	分支指令目标地址

图 5-18 BTB 中的一项

BTB 把特定程序执行过程中遇到的分支语句的历史记录下来。BTB 可能项的个数不多（例如，100 条）。每次遇到一条分支指令，硬件就会查找 BTB。我们假设分支指令的 PC 值并不在 BTB 中（因为是第一次遇到）。这种情况下，在分支的结果被确定以后，就会在 BTB 中创建一个新项，包含该分支指令的地址、分支的目标地址以及分支的方向（被采取还是不被采

⊖ 我们已经在 5.11 节里介绍了缓存的概念。第 9 章包含对缓存的详细讨论。BTB 本质上就是把分支目标地址与分支指令地址相匹配的缓存。用来实现 BTB 的硬件类似于缓存的硬件。

取）。下次遇到同一条分支指令的时候，这份历史信息就能帮忙预测分支结果了。IF 阶段会检查 BTB，如果查找成功（即 BTB 中包含正在读取的分支指令的地址项，并且把对应的跳转地址发给了流水线），那么 IF 阶段就立刻从分支目标地址开始取指令。这种情况下，分支指令在流水线中不会引起任何气泡。当然，预测可能会失败。通常来说预测失败是因为在执行一个循环的第一次或者最后一次迭代的分支语句。数据通路里的冲刷线路负责处理这种预测错误。我们可以通过提供多于一位的历史来使历史机制更加健壮。 208

流水线处理器中处理分支的方式的总结 指令集的实现细节通常被称作处理器的微体系结构。在微体系结构里处理分支是获取高性能的关键，尤其是在现代处理器的流水线越来越深的情况下。我们已经提到过，编译出的代码中的分支频率可以高到每三四条指令中就有一条。因此，流水线有 20 个阶段的时候，流水线不同阶段中的部分执行了的指令很少可能是顺序指令。因此，对于分支语句的可能结果早做决定，使得流水线能塞满有用的指令，是在深流水线处理器上取得良好性能的关键。我们将在 5.15 节中讨论流水线处理器设计的现状。表 5-5 总结了我们在本章讨论过的各种用于处理分支的技术，以及使用了这些技术的处理器。

表 5-5 处理分支的技术总结

名称	优点	缺点	实际应用
抱过流水线	策略简单，不需要冲刷指令的硬件	性能损失	早期的流水线机器，如 IBM 360 系列
分支预测（分支未被采取）	只需少量额外硬件，因为按顺序执行所需取的指令已经在 IF 阶段中	需要在流水线中冲刷指令	绝大多数现代处理器，如 Intel Pentium、AMD Athlon 和 PowerPC 都使用这个技术。通常它们也使用复杂的分支目标缓存。MIPS R4000 采用一个延迟槽加上两个周期的分支不采取预测的混合策略
分支预测（分支被采取）	性能好，但是需要更正精巧的硬件设计	因为跳转的目标也就是 PC 的新值要一直等到分支指令在 EX 阶段才知道，这个技术需要更精巧的硬件支持才能投入实际应用	—
延迟分支	不需要任何额外硬件来抱延或者冲刷指令；通过把流水线延迟槽暴露给编译器来获取高性能	在现代处理器增加流水线深度以后编译器填充延迟槽就更加困难了；由于向后兼容性而限制了微体系结构的演化； 让编译器不仅要针对目标体系结构，还要针对特定硬件实现	较旧的 RISC 体系结构，如 MIPS、PA-RISC 和 SPARC

209 ~ 210

5.13.4 冒险总结

我们已经讨论了结构、数据和控制冒险，以及在微体系结构内解决它们的最基本的机制。我们主要介绍了检测和解决这些冒险的硬件方式。当冒险被硬件检测和处理的时候，它们经常被称作硬件互锁。但是，这份责任也完全可以由硬件转移给软件，即编译器。办法就是把微体系结构的细节暴露给编译器，让编译器作者可以确保（a）这些冒险在程序优化的时候就被消除了，或者（b）通过显式插入 NOP 指令来克服冒险。例如，对于我们一直在讨论的五阶段流水线，如果编译器总是确保被写入的寄存器在紧跟的至少三条指令中没有被用到的话，

就不会有写后读冒险。或者可以通过把其他有用的指令放在定义一个值和随后使用这个值的语句之间，也可以在找不到这种有用指令的时候把显式的 NOP 指令插进去。

把编译器的负担转移到编译器带来的好处是硬件就被简化了，因为既不需要检测冒险，也不需要采用类似数据前递的技术。早期版本的 MIPS 体系结构就没有硬件互锁，完全依靠编译器来解决这种冒险。但是，问题是编译时并不一定能了解这种依赖，比如说一个加载指令也许需要超过两个时钟周期才能取得数据，这取决于内存系统的状态。随着流水线变得更深，问题也愈加严重。因此，所有的现代处理器都采用硬件互锁来消除冒险。为了让硬件和软件合作，芯片生产商把微体系结构的细节公开出来，以帮助编译器作者利用这些细节来编写高效的编译器。有些微体系结构做得更过一步，VLIW（Very Large Instruction Word，超长指令字）微体系结构被设计成与编译器紧相合作，因此两者对于决定处理器的性能来说同等重要。

表 5-6 总结了可能导致流水线拖延的 LC-2200 指令，以及这样的拖延能被硬件方案解决到什么程度。

表 5-6 LC-2200 中冒险的总结

指令	冒险类型	可能拖延的周期数	加上数据前递之后	加分支预测（假设分支未被采取）
ADD, NAND	数据	0,1,2,3	0	不适用
LW	数据	0,1,2,3	0 或 1	不适用
BEQ	控制	1 或 2	不适用	0（成功）或 2（预测失败）

5.14 在流水线处理器里处理程序不连续性

在之前讨论中断时提到过，我们得等到有限状态机处在一个可以进入中断的干净状态。对于非流水线处理器，执行完一条指令时就是这样的状态。在流水线寄存器里，由于任何时间点都有多条指令在执行途中（即处于部分执行状态），难以定义这样的干净状态是什么。这就使得中断以及异常和陷入等其他导致程序不连续性的东西处理起来很复杂。

处理中断有两种可能的办法。一种是外部中断一到达，处理器就

1）停止继续向流水线发送指令（即 IF 阶段的逻辑来开始产生 NOP 指令向后传递）；

2）一直等到已经在流水线里的指令运行完（也就是把流水线排空）；

3）进入中断状态，类似于我们在第 4 章讨论的情形，然后做中断规定该做的事情（参见例 5-17）。

把流水线排空的坏处就是外部中断的响应时间可能很慢，尤其是现代处理器流水线很深的时候更是如此。另一种选择是冲刷流水线。中断到来的时候发送一个信号给各个阶段，让丢弃它们各自在处理的指令。这使得处理器可以立刻处理中断。当然了，这种情况下，最后完成的指令的地址就得存在 PCL 里，因为它是程序在处理完中断以后继续执行的起点。实现这样的方案有一些微妙的小地方得注意：要确保在流水线的任何阶段里，部分执行的指令不会篡改任何永久的程序状态（即寄存器值）。

211 现实中，处理器不会采用以上两种极端情况。中断会被流水线的特定阶段捕获，程序执行就会从这个阶段的指令那里开始。这个阶段之后的指令就继续执行到完成，而这个阶段之前的指令就被冲刷掉了。在任何一种情况下，都有一个有趣的问题："在流水线处理器里用硬件支持中断意味着会发生什么？"流水线寄存器是处理器内部状态的一部分。之前提到的几个

方法都会让处理器进入干净的状态，所以我们只需要关心把 PC 这个值记下来以在中断以后能正确恢复执行即可。需要保存的 PC 的值当然取决于选择的方法。

如果我们决定简单化处理，在进入 INT 宏状态（参见第 4 章）前排空流水线，那么只要把 IF 阶段里指向程序下一条指令的 PC 值冻结起来就够了。在中断的时候，硬件会把这个 PC 值告诉 INT 宏状态以便保存。

例 5-17 列举一个流水线处理器的硬件从中断开始到处理器开始执行中断处理代码的过程中采取的步骤。假设流水线遇到中断的时候会排空（只要接述出来就行了）。

答：

1）允许流水线中现存的指令（有用的指令）完成执行。

2）停止取新的指令。

3）开始从取指令阶段向流水线中发送 NOP 指令。

4）在所有指令都执行完毕之后，记录程序如复执行所需的 PC。（这个值就是最后一条完成的有用指令的内存地址加 1。）

5）后面 3 步是我们在第 4 章讨论过的中断状态行动。

6）进入 INT 状态，发送 INTA 信号，接收向量，禁止中断。

7）把当前模式存进系统栈里，把模式切换到内核模式。

8）把 PC 存进 Sk0，从中断向量里获得中断处理代码地址；载入 PC；继续流水线执行

冲刷流水线（不管是完全冲刷还是部分冲刷）需要让流水线寄存器带着每个指令的 PC 值，因为我们不知道什么时候会发生外部中断。[⊖]实际上，在流水线处理器里每条指令都必须把 PC 值带着，因为任何指令都可能导致异常 / 陷入。我们需要知道导致陷入的指令，才能处理完陷入 / 异常以后恢复执行。遇到中断以后，硬件会把第一个未完成的指令（即程序继续执行时开始的指令）的 PC 值告知 INT 宏状态以便保存。例如，如果中断被流水线的 EX 阶段捕获，那么 MEM 和 WB 阶段的当前指令被允许执行完，而对应于流水线里 EX 阶段的指令的 PC 值就是程序继续执行的位置。 212

例 5-18 一个 LC-2200 的流水线实现允许中断在五阶段流水线的 EX 阶段里被捕获，将流水线前面的指令排空，并冲刷棒之后的指令。假设解决与后读冒险采用了寄存器前递。考虑以下程序：

```
地址
100   LW   R1, MEM[2000];  /* 假设 MEM[2000] 的值是 2 */

101   LW   R2, MEM[2002];  /* 假设 MEM[2002] 的值是 5 */

102   NOP                ;  /* 给 LW R2 配了一个 NOP    */
103   ADD  R3, R1, R2;      /* 加法 R3 ← R1+R2   */
104   SW   R3, MEM[2004];  /* 把 R3 存入 MEM[2004]     */
105   LW   R1, MEM[2006];  /* 假设 MEM[2006] 里的值是 3 */
```

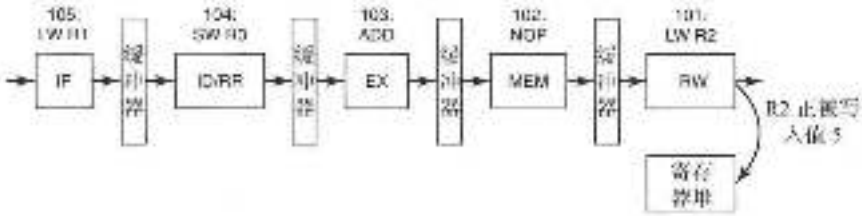
NOP（地址是 102）在流水线的 MEM 阶段时发生了中断。回答下列问题：

a. R1、R2、R3、MEM[2000]、MEM[2002]、MEM[2004] 和 MEM[2006] 里的值在进入 INT 状态时分别是多少？

b. 传递给 INT 状态的 PC 值是多少？

⊖ 回忆一下，在早先关于流水线寄存器的讨论中，我们提到过只有 BEQ 指令的执行需要带着 PC 的值。

答：当中断发生时，处理器状态如下：



因此，在 EX 阶段前面的阶段（即 MEM 和 RW）将会执行完，而 ID/RR 和 IF 阶段里的指令会被冲刷掉。程序将会在地址 103: ADD R3，R1，R2 处重新开始执行。

213

a. R1 = 2；R2 = 5；R3 = 值是未知（因为中断使得 ADD 指令结束）

MEM[2000] = 2; MEM[2002] = 5; MEM[2004] 未知；（因为中断使得 SW 指令被结束[⊖]）；MEM[2006] = 3

b. 传递给 INT 阶段的 PC 值就是 ADD 指令的地址即 103，也就是程序在中断结束以后开始运行的地方。

之前的讨论主要关心外部中断导致的程序不连续性。对于陷入和异常，我们别无选择，只能让之前的指令完成（即排干），而冲刷掉之后的指令，然后进入 INT 状态，把导致陷入/异常的指令的 PC 值传递给它。

5.15 处理器设计的高级话题

流水线处理器设计始于 20 世纪 60 和 70 年代，那是个高性能大型机和向量处理器的时代。很多在那个时代发明的概念在现代处理器设计里仍然很有用。在本节中，我们将会回顾处理器设计里的一些高级概念，包括当前流水线处理器的最新进展。

5.15.1 指令级并行

流水线处理器有一点很好的地方：就是它不用修改串行编程模型的观念。也就是说，从程序员的角度来说，第 3 章中介绍的一个简单处理器跟本章介绍的流水线处理器是没有区别的。程序的指令看起来执行的顺序与程序员所写的完全一样。原始程序里的指令出现的顺序被称作程序顺序。流水线处理器通过识别程序中相邻的相互独立的指令来缩短它的执行时间，因此，这些指令的执行时间可能会互相重叠。指令级并行（ILP，Instruction-Level Parallelism）就是给指令之间可能出现的重叠而起的名字。指令集并行是程序的一个属性，是一种通常被称作隐式并行的并行性，因为原始程序是串行的。在第 12 章中，我们将会讨论开发显式并行程序的技术，以及支持它的体系结构和操作系统。流水线处理器利用指令级并行来为串行程序获得性能提升。读者立刻可以看出，冒险限制了指令级并行。尤其是控制冒险，是试图利用指令级并行者的祸星。基本块是用来表示程序里被分支语句分隔的指令串的（参见图 5-19）。在图 5-19 中，第一个基本块中可用的指令集并行的量是 4，而第二个基本块的是 3。实际上流水线处理器可以利用到的并行性还要受本章中提到的其他类型的冒险（数据和结构）的限制。

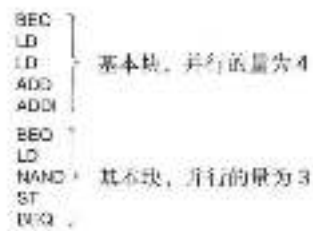


图 5-19 基本块和指令级并行

214

关于处理器的设计和实现，我们只讲了冰山一角。例如，

⊖ 此处作者误写为 ADD 指令。——译者注

对于避免流水线各个阶段之间的共享资源需求的冲突（例如寄存器堆和 ALU）我们只介绍了很简单的机制，使得处理器实现可以完全利用可用的指令集并行性。而且，我们还讨论了解决控制冒险的简单方案，让处理器可以利用跨越多个基本块的指令集并行。多发射处理器已经成为行业标准，但是对体系结构提出了新的挑战。

5.15.2 更深的流水线

现代处理器的深度经常远高于 5。例如说，Intel Pentium 4 处理器就有超过 20 个流水线阶段。但是，由于编译生成的代码里分支的频率很高，典型的基本块尺寸会很小（大约在 3 到 7 之间）。因此，要想让流水线处理器值得采用，用聪明的技术来利用基本块间的指令级并行就势在必行。微体系结构的领域里包含无数种提升处理器性能的手段，令人着迷，同时又不断演化。在第 3 章中，我们提到过多发射处理器，其中超标量和超长指令字是两种特例。指令发射是指把一个指令送进处理器流水线以执行的过程。在我们迄今为止考虑的五阶段流水线里面，每个时钟周期恰好发送一条指令。从多发射处理器的名字就可以看出，它们每个时钟周期里发射多条指令。作为近似，让我们假设硬件和编译器会确保同一个时钟周期内被发射的指令没有我们讨论过的任何冒险，所以它们可以独立执行。那么，处理器就得有多个解码单元和多个功能单元（如整数算术单元、浮点算术单元、加载/存储器等）来满足同一时钟周期发射的指令的不同需求（参见图 5-20）。取指令单元将会从内存读取多条指令以利用多个解码单元。解码单元也得能把解码的指令发给多个功能单元中的任意一个功能单元。 215

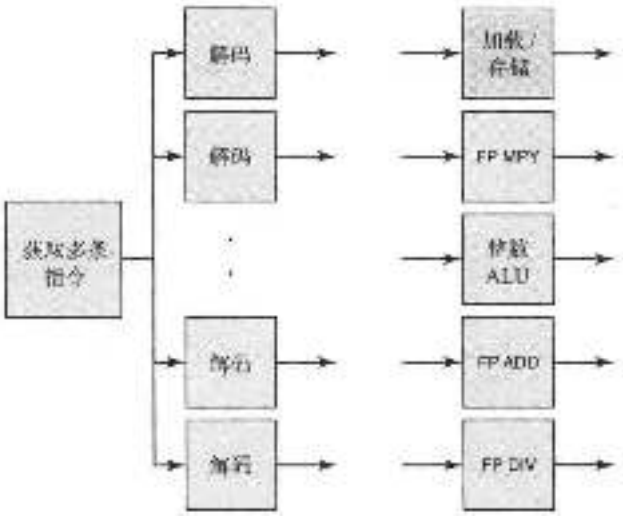


图 5-20 多发射处理器的流水线

这样的处理器的有效吞吐量接近于流水线级数和超标量度数的乘积。换句话说，现代处理器不但流水线很深，而且还有好几条流水线在并行执行。深度流水和超标量对体系结构和系统软件的开发者（尤其是编译器作者）提出了几个有趣的挑战。

你也许会怀疑深度流水线到底有没有必要。采用深度流水线有以下几个理由：

- **访存时间的相对增加** 在第 3 章，我们提到过数据通路里延迟的组成部分。由于芯片上的特征越来越小，线缆延迟（即在数据通路元素之间传递数据位的成本）取代逻辑操作成为决定时钟周期时间的瓶颈。但是，从寄存器或者缓存里取值（我们将在第 9 章中

讨论）所花费的时间还是比逻辑延迟和线缆延迟都要多。因此，时钟变快以后，允许从缓存读取数据花费多个周期可能是必要的。这就增加了需要从缓存取值的部件的复杂度，也就增加了整个处理器的复杂程度。把花费多个时钟周期进行的特定操作拆成多个阶段来执行是一种解决这类复杂性的办法。

- **访问微码 ROM** 在第 3 章我们介绍了指令的微程序实现。尽管该技术有它的问题，流水线处理器还是可以考虑采用它来实现指令集中的某些复杂指令。访问微码 ROM 可能会使流水线深度增加。
- **多个功能单元** 现代处理器的指令集包括整数运算和浮点运算。微体系结构通常会包含针对浮点加减乘除运算而与整数 ALU 区分的专门功能单元。简单五阶段流水线中的 EX 单元被一组功能单元的集合所替代（参见图 5-20）。为了调度多个功能单元，可能需要一个额外的流水线阶段。
- **浮点运算专用流水线** 浮点指令与相应的整数指令相比需要更多的时间来执行。在简单五阶段流水线里最慢的功能单元将会决定时钟周期时间。因此，自然会想把功能单元本身也弄成流水线，于是我们得到的结构类似于图 5-21，其中不同的功能单元有不同的流水线深度。这种差异化的流水线可以支持多个延迟特别长的操作（如浮点数的 ADD），而不会因为结构冒险拖延整个流水线。

216

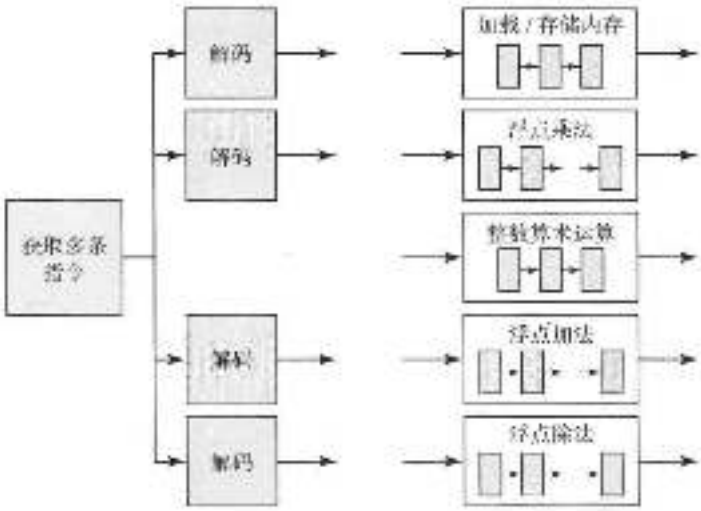


图 5-21 不同功能单元有不同深度的流水线

- **乱序执行和重排缓存** 回忆一下，流水线处理器无论怎么利用指令级并行，都应当保留程序的顺序执行。由于不同指令可能会在流水线中采取不同的道路，维持这种看上去的序列性就变得更复杂。因此，现代处理器区分发射顺序和完成顺序这两个概念。取指令单元按顺序发射指令。但是，由于不同的流水线深度不同以及其他一些原因（比如执行指令需要等待某个操作数），指令可以乱序执行和完成。我们发现，只要指令按照程序顺序从处理器引退，就不会有问题。换句话说，一条指令即使执行完毕，也会一直等到所有之前的指令都执行完才引退。为了确保这一点，现代处理器添加了一种可能是流水线的额外阶段的机制。具体来说是这样：流水线包含一个重排缓存（Reorder Buffer，ROB），用于将完成执行的指令按程序顺序引退。重排缓存确保指令完成的效

果（即写入对体系结构可见的寄存器、读取或者写入内存）的生效在之前的所有指令之后。它通过记录发射时的指令的程序顺序，并将需要发送到体系结构可见的寄存器和内存的数据与地址缓存下来来实现这一点。在有重排缓存存在的情况下，读取寄存器时仍然需要考虑写后读的依赖关系，如我们在 5.13.2 节中已经讨论的那样。

- **寄存器重命名**：之前我们提到过程序依赖，尤其是反依赖和输出依赖主要是由于编译器滥用地重用寄存器引起的（参见 5.13.2 节）。这表现为流水线处理器中的读后写和写后写冒险。为了克服这些冒险，现代处理器的物理寄存器比体系结构可见的寄存器数量多一些。处理器可能会花一个周期来检测某些资源冲突，然后通过一种称作寄存器重命名[⊖]的技术来区分对寄存器的不同使用。寄存器的重命名是在指令里标明的体系结构寄存器和指令用到的实际物理寄存器之间的一层间接对应关系。我们将会在本节的稍后部分，在介绍完 Tomasulo 算法后详细讨论这个问题。
- **基于硬件的投机执行**：为了克服控制冒险，完全利用多发射能力，很多现代处理器采用了基于硬件的投机执行，一种对分支预测的扩展。这个想法是不等待分支的解决，而同时执行不同基本块里的指令，然后设立机制来撤销由于投机而错误执行了的代码的效果。重排缓存和硬件寄存器重命名都有助于基于硬件的投机执行，这是因为，在物理寄存器或者重排缓存里的信息如果用于投机执行，并之后被发现这次执行的结果不该生效时，是很容易丢掉的。

5.15.3 在乱序执行下再次讨论程序不连续性

在 5.14 节我们讨论了流水线处理器里处理中断的简单机制。让我们来看看有乱序执行以后处理中断有什么变化。一些早期的处理器如 CDC 6600 和 IBM 360/91，用乱序执行来克服数据冒险和结构冒险导致的流水线拖延。基本的想法是按顺序发射指令，但是让指令一旦操作数可用了就立刻开始执行。这种乱序执行加上不同指令需要花不同时间这一事实，结果就是指令会乱序完成执行并引退。也就是说，指令不只是乱序地完成执行，而且还是乱序地更新处理器状态（即体系结构可见的寄存器和内存）。这样做有问题吗？看起来没有，因为这些早期的流水线处理器确实按程序顺序发射指令，也考虑了指令之间的数据依赖关系，也从来不投机地执行任何指令。外部中断对于乱序执行来说也构不成什么问题，因为我们可以采取一种非常简单的解决方案，即停止发射新指令，并让所有已经发射的指令完成执行，然后进入中断。

217
↓
218

但是，异常和陷入确实会有问题，因为引发异常的指令之后的指令可能已经执行完了。这种情况被定义为非精确异常，以说明发生异常时的处理器状态与程序顺序执行时并不同。早期的流水线处理器通过软件（即在异常处理例程中），或者通过最前检测长延时（如浮点运算）操作中的异常硬件技术，来恢复处理器异常状态。

我们在上一节中已经看到，现代处理器尽管乱序执行，但是会按照程序顺序进行引退。这自动消除了非精确异常的可能性。可能的异常会被缓存到重排缓存里，然后严格按照程序的顺序出现。

关于流水线处理器中中断的详细讨论超出了本书的范畴。有兴趣的读者可以参考计算机体系结构方面的高级课本 [Hennessy，2006]。

⊖ 注意，寄存器重命名也可以由编译器进行。编译器可以在进行程序优化时发现数据冒险并进行重命名。这时候流水线里就不需要一个额外的阶段了。

5.15.4 管理共享资源

有了多个功能单元以后，管理共享资源（如寄存器堆）就更加困难了。一个被 CDC6600 推广的技术是记分板，一种用于处理我们本章先前讨论过的各种类型的数据冒险的技术。基本的想法是有一个叫做记分板的集中装置，它在一条指令进入流水线时就记下它会用到的资源。在拖延一条指令和让它继续向下执行之间做出的决定取决于该指令当前所需的资源。例如，如果有写后读冒险，那么需要读一个即将被之前指令写入的寄存器的指令就会被一直拖延到记分板显示该资源已经可用（前一条指令把值写入寄存器时）为止。因此，记分板记录了流水线里所有指令的资源需求。注意，给寄存器堆里的每个寄存器配置忙位和等待读取位就已经为简单五阶段流水线实现了一记分板。

IBM 公司的 Robert Tomasulo 想出了一个聪明的算法（以他的名字命名）。该算法是个流水线处理器中资源共享和分配问题的分布式的解决方案。该解决方案首次被应用是应用于
219 IBM360/91，最早的（在 20 世纪 60 年代）采用流水线原理的计算机之一。基本的思想是（以本地寄存器的形式）把每个功能单元与存储相关联。在指令发射的时候，需要的寄存器值会被传递到这些本地寄存器[⊖]，从而避免了读后写冒险。如果（由于写后读冒险）寄存器值不可用，那么本地寄存器就记住它们应当从哪个单元得到这个值。在一条指令被执行完毕以后，这个控制单元会把新的寄存器值通过公共数据总线（CDB，Common Data Bus）发送给寄存器堆。其他等待这个值的功能单元（可能多于一个）把它从总线里读出来，开始执行各自的指令。由于寄存器堆总是与其他功能单元以相同方式工作，它也会记住那个单元将要给那个寄存器产生值，也就可以避免写后写冒险。这样，这个分布式的解决方案避免了我们迄今为止讨论过的所有潜在的数据冒险。

Tomasulo 算法的核心思想在于使用本地寄存器来作为体系结构可见的寄存器的代理。现代处理器通过把 Tomasulo 算法里用到的所有分布式的存储集中到一个大型的物理寄存器堆里来使用这个想法。我们先前提到的寄存器重命名技术产生了从体系结构可见的寄存器到物理寄存器的动态映射。例如，如果 R1 是个体系结构可见的寄存器，是某个存储指令的源寄存器，而实际分配到的物理寄存器是，比如说，P12，那么 R1 的值就在寄存器重命名的阶段被换成 P12。因此，寄存器重命名阶段负责动态为指令分配物理寄存器。本质上，寄存器重命名消除了读后写和写后写数据冒险。我们已经在 5.13.2 节讨论过数据前递怎么在流水线寄存器里解决写后读冒险。因此，寄存器重命名加上数据前递就处理了现代处理器中的所有数据冒险。负责寄存器重命名的流水线阶段留意任意时刻哪些寄存器在被使用，以及它们什么时候会由于指令的引退而被释放（与 CDC 6600 中的计分板技术何其相似）。

重排缓存的作用是确保指令按照程序顺序引退。寄存器重命名的作用是消除数据冒险，以及支持基于硬件的投机执行。一些处理器干脆把重排缓存也去掉了，把它的功能（即按程序顺序引退指令）集成到寄存器重命名机制里面去。

让我们重新回顾写后写冒险。我们已经看过在早期流水线机器里采用的技术（诸如计分板和采用 Tomasulo 算法），也看过现代处理器里使用的技术（诸如寄存器重命名和重排缓存），都能在乱序执行的情况下消除写后写冒险。投机执行会导致写后写冒险吗？答案是否，因为尽管用了投机执行，指令还是会以程序顺序引退。任何由于分支错误预测而产生的对寄存器
220 的错误写入都会在生效之前从重排缓存里删除。

⊖ Tomasulo 算法里的这些本地寄存器起到的作用与支持寄存器重命名的现代处理器中的大型存储集相同。

尽管我们有多发射处理器、投机执行和乱序执行，但是在5.13.2节中提出的问题还没解决，即，编译器为什么会生成有写后写冒险的代码？答案就在于编译器也许用前一个写入填充了一个延迟槽（如果该体系结构用了延迟分支的话），然后进行了一次未预测到的跳转，使得前一次写入没有用上。更一般地来说，写后写冒险可能由于源料之外的代码而出现。另一个例子是当前在执行的程序和陷入处理代码之间的交互。假设一个指令由于某种原因，如果遇到陷入的话会把值写入某个寄存器（第一次写入）。而陷入的处理代码的某处则写入到同一个寄存器（第二次写入）。该指令继续执行并完成把值写入寄存器（即程序顺序中的第一次写入，现在由于陷入处理的缘故变得不相干了）。如果写入不按照程序顺序发生的话，那么第一次写入将会覆盖掉陷入处理代码的第二次写入。检测和消除此类冒险是硬件的职责。

5.15.5 功耗

另一个处理器设计的有趣维度是关心功耗。即使在当今的技术水平下，现在的GHz微处理器仍然伴带着大量的电力，以至于保持系统冷却是个重要的工程挑战。而随着处理能力持续增加，能耗消耗也随之增加。体系结构设计者的挑战是追求更高性能的同时保持低功耗。

具有在一片硅片上塞进更多晶体管的能力是件幸事，但是它也对体系结构设计者带来了重大的挑战。首先，芯片中晶体管的密度提高之后，所有的延迟（回忆一下在第3章中讨论过的时钟脉冲宽度）都减小了。这包括进行逻辑运算花费的时间、线路延迟以及寄存器的访问时间。这就对体系结构设计者带来如下挑战：原则上说，因为延迟减小了，芯片可以提升到更高的频率。但是，提升频率会增加功耗。图5-22展示了提升时钟频率为几种流行的处理器带来的功耗提升。你能看到时钟周期时间与功耗之间的高度相关性。一块3.2GHz的IntelPentium 4处理器要消耗112瓦的功率。在表中，你能看到有些时钟频率更低的处理器消耗资源更高（比如说，比较一下1.8 GHz的AMD K8和2.2 GHz的Intel P4）。原因是功耗也取决于其他芯片上的资源，比如说处理器的字长和包括片上缓存在内的内存系统。（我们将在第9章讨论缓存的设计。）

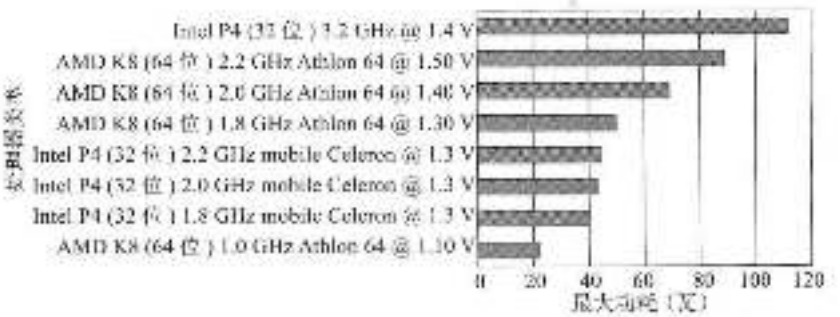


图5-22 CPU功耗[⊖]对时钟频率对功耗有重大影响。其他诸如处理器字长和其他片上资源（包括缓存）等因素也影响功耗

5.15.6 多核处理器设计

现实情况是，随着技术的进步，如果处理器的时钟频率调到技术允许的上限，我们很快就会有功耗等于核电站的笔记本电脑出现。当然了，解决方案不是停止制造密度更高、时钟频率 221

⊖ 来源：Intel的数据来自www.sandpile.org/impl/p4.htm，AMD的数据来自www.sandpile.org/impl/k8.htm。

更快的芯片。体系结构设计者在转向另一条不需要提升时钟周期而可以提升处理器性能的道路，即多处理器。该新技术已经以多核的名称被投入市场（如 Intel Core 2 Duo、AMD Opteron 四核处理器等）。每个芯片里装备多个处理器，而运算则被这多个处理器共同分担，系统的吞吐量（即性能）从而增加。多核处理器（也经常被称为芯片多处理器）的体系结构和硬件细节超出了本书的讨论范畴。但是，多核技术建立在并行计算的基础上，而后者的历史与计算机科学一样悠久。我们将会在第 12 章围绕多处理器更详细地讨论遇到的硬件和软件问题。

5.15.7 Intel Core™微架构：一个流水线

了解现代处理器的流水线结构是有益的。采用 Willamette 和 Galatin 微架构的 Intel Pentium 4 采用了 20 阶段的流水线，而基于 Prescott 和 Irwindale 微架构的则采用了 31 阶段的流水线。Intel 和 AMD 公司的产品（虽然两者都支持相同的 x86 指令集）之间的一个重大区别就是流水线的深度。Intel 的路线是采用更深的流水线来达到更高的吞吐量，而 AMD 则采用了相对较浅（14 阶段）的流水线。

Intel 处理器的一个系列包括 Intel Core 2 Duo、Intel Core 2 Quad 和 Intel Xeon 处理器采用了一个共同的内核微架构，参见图 5-23。值得一提的是该流水线结构被 20 世纪 90 年代中期的 Pentium "P6" 微体系结构中首次采用。图 5-23 被大大简化了，以展现现代处理器流水线的基本功能。读者有兴趣的话可以去 Intel 的网站参阅关于 Intel 体系结构的免费手册[⊖]。总的来说，该体系结构分为前端、执行核心以及后端三部分。前端的职责是按顺序从内存中获取指令，并用四个解码器来把解码后的指令（也被称作微操作）提供给执行核心。前端包含了取指令和预解码单元，以及微码 ROM。而流水线的中间部分是个乱序执行的核心。每个时钟周期在所需资源就绪（即没有写后读冒险）并且对应的执行单元可用（即没有结构冒险）的情况下可以发射高达 6 条微操作。这个中间部分包括寄存器重命名、重排缓存、保留站以及一个指令调度器。最后，后端负责按照程序顺序引退指令，以及更新程序员可见的体系结构寄存器。Intel Core 流水线体系结构里出现的不同功能单元的功能列举如下：

222

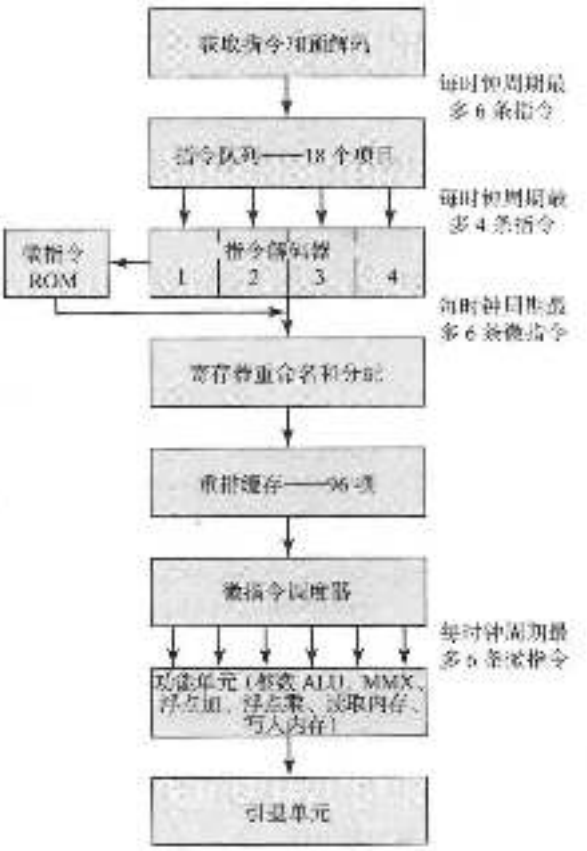


图 5-23 Intel Core 微体系结构的流水线功能描述

- **指令获取和预解码** 该单元负责两件事情：获取最可能要执行的指

⊖ Intel Core 是 Intel 公司的注册商标。

⊜ Intel 的网站，www.intel.com/products/processor/manuals/index.htm。

令，并且通过预解码来识别变长指令（因为 x86 体系结构支持变长指令）。预解码帮助指令获取器远在分支的结果产生之前就找出哪些是分支指令。一个精巧的分支预测单元（BPU，Branch Prediction Unit）是读取最可能执行的指令流的阶段的一部分。分支预测单元用专门的硬件来预测不同类型的分支指令（条件、直接、间接，以及调用函数和返回）的结果。预解码单元可以每个时钟周期向指令队列中写入至多六条指令。

- **指令队列** 指令队列取代了简单的五阶段流水线里的指令寄存器（IR，Instruction Register）。因为它装了多得多的指令（能装 18 条指令），指令队列能装下原始程序中的一小段（如一个小的循环）以加速流水线处理器的执行。而且，它能帮助省电，因为前端的剩下部分（即取指令单元）可以在执行循环的时候关掉。
- **解码和微码 ROM** 该单元包含四个解码器，因此可以每个时钟周期将指令队列中的四条指令解码。取决于指令，解码器单元可能会用微码 ROM 将一条指令扩展成多条微操作。微码 ROM 每个周期可以输出三条微操作。通过微码 ROM，在不拖慢执行简单指令的流水线的情况下实现了复杂指令。解码器还支持宏命令，将两条指令合并成一条微操作。
- **寄存器重命名 / 分配** 该单元负责将物理寄存器分配给微操作里用到的体系结构寄存器。它记录了体系结构寄存器与微体系结构中的实际物理寄存器之间的对应关系。它支持基于硬件的投机执行，消除了读后写和写后写冒险。
- **重排缓存** 该单元有 96 个项目，负责将原始程序的微操作按照程序顺序记录下来以便随后调度。它把在各个执行阶段的微操作都记录下来。由于它的尺寸原因，流水线中最多有 96 条微操作正在执行（即在执行的各个阶段）。 223, 224
- **调度器** 该单元负责把微操作分配给功能单元。它包括一个保留站，将所有微操作排队等待它们所需的资源就绪且需要用到的执行单元可用。它每个时钟周期可以调度至多 6 条微操作，当然这取决于有多少指令为执行做好了准备。
- **功能单元** 如名字所暗示的，这些是具体执行微操作的单元。它们中有的执行单元的延迟是一个周期（诸如整数加法），也有一些对于经常用到的高延迟微操作会采用流水线化的执行单元。还有流水线化的浮点运算单元和内存存取单元等。
- **引退单元** 该单元代表了微体系结构的后端，用重排缓存来按程序顺序引退微操作。另外，它按照程序的顺序更新体系结构的状态，并且管理代码执行的时候可能遇到的异常和陷入的顺序。它也与保留站通信以告知微操作在等待的资源是否可用。

小结

本章中，我们讨论了很多东西。我们在 5.1 节和 5.2 节中讨论了衡量处理器性能的标准；5.5 节讲了阿姆达尔定律和加速比的概念；在 5.7 节中，关于提升处理器性能的讨论引入了流水线处理器的概念。5.11 节和 5.12 节讨论了支持指令流水线所需的数据通路元素，以及得到适合流水线的体系结构和实现所需的最佳实践。流水线设计的祸根是冒险。我们在 5.13 节里介绍了流水线处理器里遇到的不同类型的（结构、数据、控制）冒险，以及相应的对策。流水线处理器实现遇到的另一个棘手的问题是处理程序的不连续性，我们在 5.14 节中讨论了该问题。在 5.15 节里讨论了一些与流水线处理器实现相关的高级话题。最后，我们以从计算机出现以来处理器实现的演化史作为本章结束。

历史回顾

我们当中的大多数人都对于自己身处何时何处，是否用着全天候供应的自来水，是否开着飞驰的小轿车，是否在掌中使用着高性能的计算机感到本就应该如此。当然了，大部分人也会很快意识到，在计算机技术短短的历史里，因为集成电路的突破式发展的惊人速度，我们其实已经走过了很长的一段路。

回头看看我们走过的道路是很有意思的。在20世纪60年代和70年代，流水线处理器设计被留给了当年最高端的计算机。在20世纪60年代，控制数据公司（Control Data Corporation）和IBM公司的研究者率先进行了流水线处理器设计的基础工作，并因此设计出
225 了诸如CDC 6600，IBM 360和370系列等高端计算机系统。这样的系统被称作大型机，估计是因为它们被塞进一个大金属机柜里的缘故。它们主要定位于商业应用。IBM 360系列的体系结构总设计师是吉恩·阿姆达尔。他的名字随着阿姆达尔定律一起名垂青史。他在设计WISC计算机时发现了流水线的原理，并在他的1952年于威斯康辛大学麦迪逊分校的博士论文里记录了下来。[1] Seymour Cray是高性能计算机的先锋，成立了Cray Research公司，该公司用以Cray-1开始的Cray系列计算机引领了向量超级计算机的时代。在成立Cray Research之前，Seymour Cray是控制数据公司的体系结构总设计师，而该公司在20世纪60年代是高性能计算领域的领先者，生产了CDC 6600（通常被认为是第一台商用的超级计算机）以及紧随其后的CDC 7600。

在高端计算机发展的同时，人们对小型计算机的开发也充满了兴趣，DEC公司的PDP系列以PDP-8型机引领道路，紧接着是PDP 11，再之后是VAX系列机器。这些计算机起初是面向科学和工程社区的，因此其优先目标是低成本而不是高性能。因此，这些处理器被设计成不采用流水线技术。

我们已经在第3章观察到，随着20世纪80年代“杀手级微处理器”的出现，加上在编译器技术和RISC体系结构中的开创性研究，为指令流水成为除了最低端的嵌入式处理器之外的所有处理器设计的常态铺平了道路。现在，即便是学步的小孩玩的游戏机，里面采用的处理器也是流水线的。

最后来澄清一下用词，超级计算机是被设计成用来求解有挑战性的科学和技术领域遇到的计算问题的。这些大挑战问题激发DARPA[2]开展研究项目以求刺激人们寻找突破性的计算机技术。当时，大型机用于商业、金融和其他方面的技术应用。而现今，这种高端计算机一般被称作服务器。服务器组成由高速网络相连的计算机的计算机组，也被称作机群。服务器既用于科学应用（如IBM的BlueGene的大规模并行体系结构），也用于技术应用（如IBM的z系列）。用于搭建这种服务器的处理器十分相似，并且也同样遵循我们在本章中讨论的流水线原则。

练习题

1. 判断对错，并给出理由：对于特定的负载和特定的指令集，降低所有指令的CPI（每指令的时钟周期数）
226 一定会提升处理器的性能。

2. 其体系结构有三类指令，它们的CPI如下：

---

[1] 来源：http://en.wikipedia.org/wiki/Wisconsin_Integrally_Synchronized_Computer。

[2] DARPA是一个美国联邦政府机构，其职能为推动军事技术的进步。

类型	CPI
A	2
B	5
C	3

一个体系结构设计师发现她可以用某种聪明的体系结构小技巧来将B类指令的CPI减少，而另外两种指令的CPI不变。但是，她发现这个改动同时也会把时钟周期时间增加15%。为了让这个改动有意义，B的最大可允许CPI（四舍五入到最近整数）是多少？假设她要在这个处理器上执行的所有负载中A占40%，B占10%，C占50%。

3. 如果处理器时钟频率是8MHz，且每个指令都需要花4个时钟周期的话，执行一个有2 000 000条指令的程序需要花多久？

4. 一个聪明的体系结构设计师重新实现了一个给定的指令集，将其中一半指令的CPI减少了一半，但是处理器的时钟周期时间上升了10%。这个新实现与原先相比快了多少？假设执行任何程序时所有指令被执行的概率都相等。

5. 在一个非流水线的（多周期）MIPS CPU中人们在考虑对ALU的一个改动。这个改动会让你可以在一个时钟周期里完成算术运算并把结果写入寄存器堆。但是，这么做会增加CPU的时钟周期时间。具体来说，原来的CPU频率是500MHz，但是新设计的速度只有400MHz。这个改动会提高还是降低性能？这个新设计与原先设计相比，快（慢）了多少？假设指令执行的频率如下：

指令	频率
LW	25%
SW	15%
ALU	45%
BEQ	10%
JMP	5%

原始设计中指令的CPI如下：

227

指令	时率
LW	5
SW	4
ALU	4
BEW	3
JMP	3

6. 以下是不同类型的指令的CPI：

类型	CPI
R类	2
I类	10
J类	3
S类	4

相同程序的两个不同实现的指令频率如下：

类型	实现1	实现2
R	3	10
I	2	1
J	5	2
S	2	3

哪种实现执行起来更快？为什么？

7. 静态和动态指令频率的区别是什么？

8. 给定以下指令以及对应的CPI，回答下列问题。

指令	CPI
ADD	2
SHIFT	2
其他	2（包括ADD和SHIFT在内的所有指令的平均值）
ADD/SHIFT	3

228

如果ADD紧跟着SHIFT的序列在程序中出现的动态频率中是20%，那么把所有的{ADD，SHIFT}这个序列换成新指令ADD/SHIFT的话，能够性能提升百分之多少？

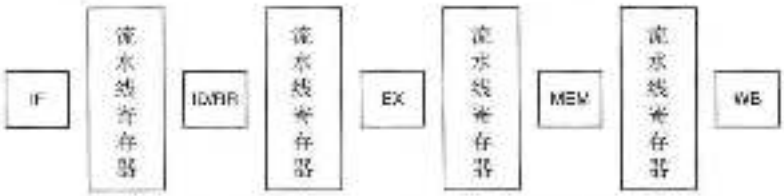
9. 比较结构、数据和控制冒险的异同。怎么消除它们对流水线性能可能造成的负面影响？

10. 怎么减轻或消除写后读冒险？

11. 分支目标缓存是什么？它是如何使用的？

12. 为什么LC-2200的五阶段流水线里的执行阶段需要第二个ALU？

13. 在一个如下图所示的五阶段流水线里（阶段之间有缓冲器），解释分支指令带来的问题，并给出解决方案。



14. 解释一下，为什么不管我们用保守策略，还是用分支预测（预测分支未被选取），只要分支被选取就一定会有两个周期的延迟（即，流水线中塞进了两个NOP指令），然后第5-13.3中的五阶段流水线才能继续执行。

15. 参见图5-6a，找出负责处理BEQ指令的数据通路元素，并解释它们的作用。解释每个时钟周期里数据通路上分别发生了什么。假设处理控制冒险采取的是保守方法。你的答案应该分为两种情况：分支被采取，以及分支未被采取。

16. 一个聪明的工程师决定将5阶段流水线中的两个周期的“分支被采取”的延迟降低为一个周期。她的想法是直接用EX阶段计算出的分支目标来取指令。（注意第5.13.3节中给出的方法要求目标地址首先存进PC。）

a. 说出对图5-6a要进行哪些修改来实现该想法。[提示：如果分支被采取，你得同时将目标地址传递给PC和指令内存。]

b. 尽管这么做能让流水线在分支被采取时的气泡数减少到一个，但未必是个好主意，为什么？

[提示：考虑时钟周期时间的影响。]

17. 在某流水线处理器里，每个指令都被分成五个阶段，每个阶段需要花 1ns 时间来执行。那么最好情况下执行完 1 000 000 000 条指令需要花多久？ 229

18. 用图 5.6(b) 中展示的五阶段流水线，回答下列问题：

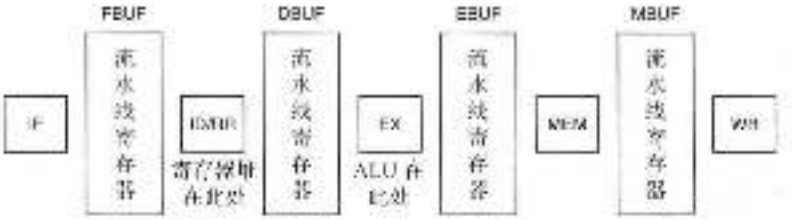
a. 展示 LC-2200 执行 BEQ 指令时每个阶段的行为（类似于第 5.12.1 节里的内容）。

b. 只考虑 BEQ 指令，计算 FBUF、DBUF、EBUF 和 MBUF 的尺寸。

19. 重复第 18 题，换成 SW 指令。

20. 重复第 18 题，换成 JALR 指令。

21. 给定下图所示的流水线处理器的数据通路：



一条加载一个字的指令有以下 32 位格式：

操作码	寄存器 A	寄存器 B	偏移量
8 位	4 位	4 位	16 位

指令的语义是

B ← 内存 [A + 偏移量]

把寄存器 A 的内容加上 16 位的偏移量，所得到的地址在内存中对应位置所存储的数据被赋值给寄存器 B。所有的数据和地址都是 32 位量。

写出为了执行该指令，流水线阶段之间的流水线寄存器 FBUF、DBUF、EBUF 和 MBUF 里面分别需要什么。清楚地给出每个缓冲器的布局。写出缓冲器中每个字段的宽度。对于本题你不需要关心体系结构中其他类型指令的格式或者需求。

22. 考虑以下两条指令： 230



如果 I_2 在流水线中紧跟着 I_1，不用前递的话，执行下去需要插入几个气泡（即 NOP 指令）？解释你的答案。

23. 考虑以下程序片段：

假设这些指令中没有冒险。现在，IF 阶段即将读取 1004 处的指令。

a. 说明五阶段流水线的当前状态。

b. 假设采用排空策略来处理中断，那么我们进入 INT 宏状态需要之前需要花费多少个周期？存储进 INT 宏状态的 PC 值会是多少？

c. 假设采用冲刷策略来处理中断，那么我们进入 INT 宏状态需要之前需要花费多少个周期？存储进 INT 宏状态的 PC 值会是多少？

地址	指令
1000	ADD
1001	NAND
1002	LW
1003	ADD
1004	NAND
1005	ADD
1006	SW
1007	LW
1008	ADD

参考文献注释和扩展阅读

Jim Thornton [Thornton，1964] 年写的论文是第一篇提出流水线基本原理的技术论文，读起来令人
231 被振。他的想法采纳进了第一台超级计算机 CDC 6600 中。IBM 的文档 [IBM System/360，1964；IBM System/370，1978] 是了解 IBM 的体系结构的很好的资源，它们对于流水线处理器的设计影响深远。Tomasulo 算法首次公开出现于一篇技术文献 [Tomasulo，1967] 被 IBM 360/91 采用。了解 VAX 11 体系结构的一个很好的资料是 [Strecker，1978]。Berkeley RISC 体系结构是在 [Patterson，1981] 中提出的。Stanford MIPS 处理器则是在 [Hennessy 1981] 中提出的。IBM 801 体系结构首次公开讨论是在 [Radin，1982] 中。Yale Patt，1996 年的 Eckert-Mauchly 奖得主，对指令级并行和超标量处理器设计做出了重大贡献，尤其是在分支预测领域 [Yeh，1992]。Acorn RISC Machine（ARM，绿果 RISC 机）是一种流行的 RISC 体系结构，专门为嵌入式环境量身定制。[ARM，1990]。Hennessy 和 Patterson 的书是了解更多关于流水线处理器设计高级概念的好资料 [Hennessy，2006]。计算机体系结构领域的主要会议有 ISCA ㊀、
232 Micro ㊁、ASPLOS ㊂和 HPCA ㊃。

㊀ ISCA 全称是国际计算机体系结构会议（International Symposium on Computer Architecture），网址是 http://isca2010.inria.fr/。

㊁ IEEE Micro：www.microarch.org/micro42/。

㊂ ASPLOS 的全称是编程语言和操作系统的体系结构支持（Architectural Support for Programming Language and Operating Systems），网址是 www.cse.cmu.edu/CALCM/asplos10/doku.php。

㊃ HPCA 的全称是高性能计算机体系结构（High Performance Computer Architecture），网址是 www.cse.psu.edu/hpeb/hpca16.html。

处理器调度

6.1 引言

处理器的设计与实现对我们来说已经不再是一个谜团了。在之前的章节中，我们了解了如何设计指令集、如何用有限状态机来实现指令集，以及如何用流水线等技术来改进处理器的性能。

Google Earth 的爱好者们体验了从高空观看一个国家或者一块大陆的很酷的效果，而且如果需要的话，还可以将他们有兴趣仔细探索的城市放大，以至于能够看到每条路和每个房子的名字。

我们刚刚对处理器也做了同样的事情。在了解了处理器 ISA 是怎么被设计出来之后，我们放大视图来看看实现处理器的细节。我们用 LC-2200 指令集来作为一个具体的例子，从硬件的角度了解了很多细节。现在让我们再将视图缩小，把处理器看作一个黑盒子，计算机系统中一种珍贵而稀缺的资源。几个程序不得不在该资源上运行（Google Earth、电子邮件、浏览器、即时通信，等等），系统软件必须有效管理该资源，以满足用户的需要。

因此，我们把注意力转移到一个补充话题上，即如何把处理器作为计算机系统里的一种资源进行管理。要做到这一点，我们并不需要了解处理器的内部结构。这是抽象的力量。把处理器看作一个黑盒子，我们会找出对于管理这一稀缺资源有益的软件抽象。操作系统中涉及这一功能的部分是处理器调度，这也是本章讨论的主题。高效实现该功能也许需要回顾（即“放大视图”）处理器的指令集，以及把指令集改进得更聪明。我们将会在本章的结尾处（参见 6.11 节）回到此问题。

考虑一个简单的类比。你有衣服要洗，有考试要准备，你得做晚饭，还得给妈妈打电话祝她生日快乐。但是世界上只有一个你，而且你得按时把这几件事情都完成。你会把这些事情分出个轻重缓急，但是你还知道，并不是所有这些事情都需要一直保持关注。比如说，你让洗衣机开始洗衣服以后，直到它在洗完衣服以后朝你蜂鸣为止，你都不需要把注意力放在它上。类似地，在我们的微波炉文化中，做晚餐只需把“电视晚餐”塞进微波炉里，等它加热好了开始蜂鸣为止。因此，以下是个合理地把这些都做完的计划：

1）开始洗衣机的清洗程序。

2）把食品塞进微波炉里，开始加热。

3）给妈妈打电话。

4）准备考试。

注意，做前两件事情你只需要集中精力一段很短的时间（跟第 3 个和第 4 个任务相比）。但是，有个问题，你不知道任务 3 和任务 4 会做多久。比如说，给妈妈打电话可能会打很久很久。有可能你还在给妈妈打电话，洗衣机或微波炉就朝你发出蜂鸣了。好吧，如果洗衣机朝你发出蜂鸣了，你可以礼貌地让你妈妈等一小会儿，然后去把洗衣机里面的东西装到烘干机里，然后继续打电话。类似地，你可以让妈妈等一会儿，去把食物从微波炉里拿出来，放

到饭桌上，准备食用。当你给妈妈打完电话，你可以安心享用晚饭，然后开始准备考试。你总共有8个小时用于学习，然后你有4门要准备考试的课程。所有考试对你的最终得分一样重要。在准备考试的过程中，你需要做一个决择，要么是每门课复习一会儿，循环往复，以确保你在所有课程上都有所进展，要么是按照考试的顺序一门一门地学习。

到这时候，也许你会挠挠头，问你自己“前面讲了这么多，跟处理器调度有什么关系？”当然你更有可能已经看出这里在发生什么了。你就是那个稀缺资源。你在把你的时间划分成几份，分配给不同的任务。你把给妈妈的电话置于了主准备考试更高的优先级。你随后会在我们讨论处理器调度算法时看到一个类似优先级的概念。你给开始洗衣机的洗衣程序和做饭时以与另外两个任务相比更高的优先级，因为你知道它们需要的时间非常少。你会在最短作业优先调度策略看到给予处理器的相似原则。在你打电话时洗衣机发出了蜂鸣，你会暂时让妈妈等一会儿，跑去处理洗衣机。之后你会在处理器调度的上下文中看到一个类似的概念，称作抢占。在准备考试时，你的第一个选择类似于我们即将看到的一种被称作循环调度的处理器调度策略，而第二种选择类似于处理器调度策略中的先到先服务策略。

234

6.2 程序和进程

让我们以对操作系统的简单理解开始对处理器调度进行讨论。它就是一个程序，它的唯一用途是为执行用户的程序提供资源。

为了理解用户程序需要的资源，让我们首先回顾一下我们是怎么创建程序的。图6-1展示了某诸如C语言一类的高级语言写成的程序的内存印迹的一种看起来合理的布局。

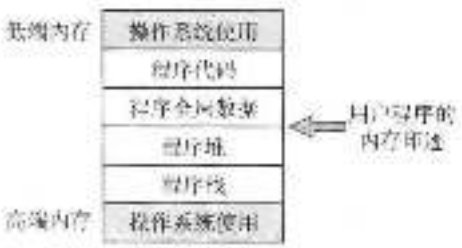


图6-1 内存印迹

我们用程序这个词表示很多种含义。但是，通常来说我们用这个词来表示对于特定问题的一个计算机解决方案。一个程序也许会以好几种形式存在。

图6-2展示了一个高级语言程序创建的生命周期。首先，我们有一个问题描述，并根据它设计了一个算法。我们把算法以某种编程语言（比如C）用一个编辑器编写出来。算法和C代码是同一个程序的不同表现形式，后者是一种把问题的解决方案代码化的方式。编译器将C语言代码编译成程序的二进制表示。这个二进制表示处理器仍然“执行”不了，因为我们写的程序用到了若干我们认为理所当然的工具，它们被“别人”提供给我们。比如说，我们调用终端I/O（比如scanf和printf），还调用数学运算（诸如正弦和余弦）。因此，下一步是将我们的代码与其他人提供的代码连接起来，以提供我们认为理所当然的那些功能。这就是链接器的功能。[1]链接器的输出仍然是程序的二进制表示，但是现在它已经处于处理器可以执行的状态了。程序的不同表现形式（文本、未链接的二进制，以及可执行的二进制）最终会进入你的硬盘。加载器，通常是操作系统的一部分，它负责读取硬盘上的内容，并创建图6-1所示的内存印迹。

编辑器、编译器和链接器都是独立的程序，而加载器则是一个更大的程序的一部分，这个更大的程序即操作系统。任何程序都需要资源来执行。这里说的资源就是处理器、内存和任何输入/输出设备。假设你要写一个简单的“Hello, world”程序。让我们来列举一下运行

[1] 在生成可执行文件的过程中，链接这步骤通常是编译器的一部分。

这个简单程序所需要的资源：你当然需要处理器和内存；此外，你需要显示器来显示你的输出。操作系统负责把程序所需要的资源交给它。

在迄今为止的讨论中，不难看出，操作系统和其他任何程序一样，也需要常驻内存中，有着类似于其他任何程序的内存印迹。图 6-3 展示了用户程序和操作系统的内存内容。

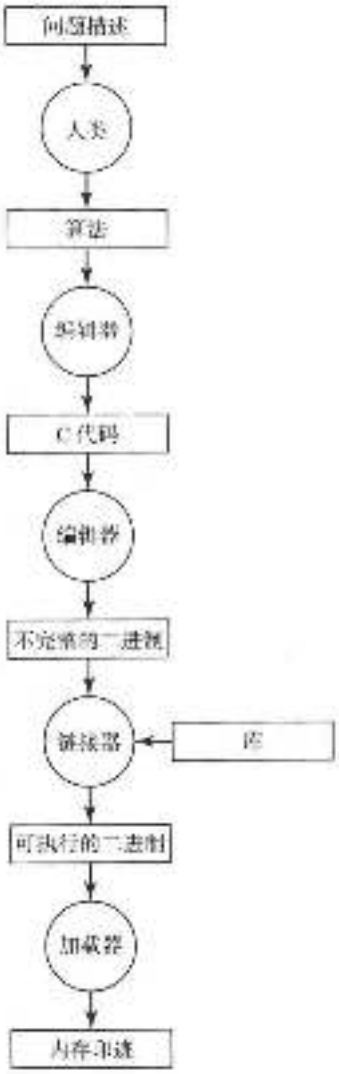


图 6-2 创建程序的生命周期



图 6-3 内存中的操作系统和用户内存

在本章中，我们将主要关注操作系统负责分配处理器资源给程序的部分，这部分被称作调度器。

调度器是操作系统中的一组例程，如其他程序一样，调度器也需要用处理器来工作——即选择在处理器上运行哪个程序。调度器包含一个决定一组程序中谁会赢得处理器时钟周期的算法。

你已经听说过进程一词。让我们来理解进程是什么，它跟程序有什么不同。进程是执行中的程序。参考图 6-1，我们把进程的地址空间定义为这个程序在内存中占用的空间。程序开始

235
:
237

在处理器上运行之后，程序占用的内存空间中的内容可能会由于程序对数据结构的操作而改变。此外，程序还能在执行中使用处理器寄存器。地址空间的当前内容加上寄存器的值，就构成了程序执行的状态（即进程的状态）。我们随后会看到如何具体地表示一个进程的状态。

进程常常是处理器调度的单位。调度器的输入是一组准备好在处理器上运行的进程，以及帮助调度器在这些进程中选择一个胜利者的其他属性，参见图 6-4。这些属性允许调度器在进程之间实现某种优先级。例如，期望运行时间、期望内存使用和期望 I/O 需求是与程序相关的静态属性。类似地，可用系统内存、程序到达时间和程序瞬时内存需求则是调度器可能用到的动态属性。紧迫性（可以表达为截止时间 / 或重要性）可能是调度器可以用到的另一类附加属性。有一些属性（诸如紧迫性）是显式地告诉调度器的，而另一些则是调度器可以隐式地推断出来的（诸如到达时间）。

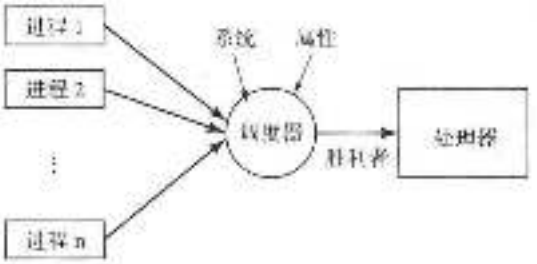


图 6-4 调度器——一个以当前可以运行的进程集合作为输入，根据系统状态和程序属性选择一个胜利者在处理器上运行

诸如任务和线程这样的词经常被用来表示工作的单位或调度的单元。我们会警告读者，虽然进程的定义在不同书里意思都一样，但是任务和线程就难说了。在大部分书里，任务的含义与进程一样。在本章中，我们只用任务一词表示工作单位。我们将会给出线程的一个基本定义，它对我们理解调度算法很有用处。

这里做个类比会很有用，如图 6-5a – c 所示。你取了晨报。它现在躺在餐桌上，没有人在读它，就像在内存中休眠的进程一样。你把它拿起来开始读。

现在有一个活动的实体在读报了，就是你。注意，根据你的兴趣，你会读报纸的不同部分。进程也是如此。取决于输入和程序的逻辑，进程可能会沿着程序的特定路径执行。这条路径为进程定义了一个控制线程。让我们回到读报纸的类比，来看看为什么在进程中有多个控制线程是有意义的。现在想象一下，在你读报纸的时候你的姐姐也来早餐桌前，和你一起看报纸，如图 6-5c 所示。取决于她的兴趣，也许她会读报纸的另一部分。现在有两个活动了（你和你姐姐），或者说有两个控制线程，在浏览晨报。类似地，一个进程中可以有多个控制线程。我们会详细说明为什么一个进程里中有多个线程是个好主意，还会在随后讨论多处理器和多线程程序的时候说明线程和进程的具体差异。现在，只需把线程理解为进程中的执行单元（也许还是调度的单元）就足够了。一个进程中的所有线程在同一个地址空间中执行，共享程序的内存印迹（图 6-1）中所展示的代码和数据结构。换句话说，进程就是程序加上所有在该程序中执行的线程。这类似于把报纸和你姐姐和你加在一起。

一个进程里可以有多个线程，但是从本章中所讨论的调度算法的角度来说，每个进程只有一个控制线程。关于调度的文献用作业来表示调度的单位。为了与此保持一致，我们决定在本章中把作业作为进程的同义词。我们把这些术语及其含义总结在表 6-1 中。



a) b) c)

图 6-5 你和你的姐姐在看报纸

表 6-1 作业、进程、线程和任务

名称	通常的含义	在本章中的使用方式
作业	调度单元	与进程同义
进程	执行中的程序；调度单元	与作业同义
线程	调度单元或进程内的执行	在本章的调度算法中未使用
任务	作业单位；调度单元	在本章中的调度算法中未使用，除了描述 Linux 的调度算法以外

6.3 调度环境

一般来说，处理器可能会用来做一组特定的任务。以手机之类的嵌入式设备的处理器为例来说，可能会有一个任务是负责响铃，一个任务是负责拨打电话，等等。在这种专用环境里，调度器可以简单地循环检查有没有可以执行的任务。

在大型面向批处理的计算的年代（跨越了 20 世纪 60 年代、20 世纪 70 年代和 20 世纪 80 年代早期），调度环境是多程序的，即多个程序从磁盘里读取到内存上，操作系统根据它们的相对优先级循环执行它们。那时候，你得把程序的描述（在一张磁盘上）以及以一种被称为作业控制语言（JCL，Job Control Language）的语言写成的它的执行需求交给一个人类操作员。一般来说你会几个小时以后再回来拿你的输出。在数据终端和小型机到来以后，交互式，或者说分时的环境就变得切实可行。这种情况下，处理器时间被坐在终端前使用计算机的用户所共享。值得一提的是，分时环境一定是多程序的，但是多程序的环境未必是分时的。这些不同的环境也产生了不同类型的调度器（参见图 6-6）。

高级调度器（long-term scheduler）通常用于面向批处理的多程序环境里，均衡内存中的作业以优化系统资源（处理器、内存、硬盘等）的使用。随着个人计算机和分时环境的到来，高级调度器对于所有实际的用途来说，在绝大多数现代操作系统中根本不存在。取而代之的是操作系统中一个叫做加载器的组件，在用户开始运行一个存储在磁盘上的程序（即，在笔记本电脑上点击图标，或者是在命令行下敲入程序名）后创建内存足迹。高级调度器（或者加载器）负责根据驻留在磁盘上的用户程序（u_i）创建驻留在内存的进程（p_i）。

中级调度器（medium-term scheduler）则在包括现代操作系统在内的很多环境里用到，它们紧紧跟踪当前正在 CPU 上执行的进程的动态内存占用量，以决定是否要增加或减少多道程序度，多道程序度的定义是同时在内存中存在并且竞争 CPU 的进程个数。该调度器主要负责控制一种被称作颠簸的现象，即当前的进程集合的内存需求超出了系统容量，导致进程在各自的执行中进展缓慢。中级调度器在系统吞吐量下降时，把程序在磁盘（在图 6-6 中以交换空间的形式展现）和内存之间来回移动。我们将会在第 8 章中更详细地回到颠簸这个概念上。

低级调度器（short-term scheduler）也在大部分现代操作系统中出现，首次出现是在分时

238 ~ 239

操作系统中。调度器负责在当前驻留在内存中的进程中选择一个来运行。本章的重点，包括介绍的算法在内，主要关注的都是低级调度器。最后一个东西是分发器，负责给低级调度器选择的进程设置好处理器寄存器的值，以让系统准备好执行该进程。高级调度器、中级调度器、低级调度器和分发器都是操作系统的组件，并只它们互相协调各自的行为。

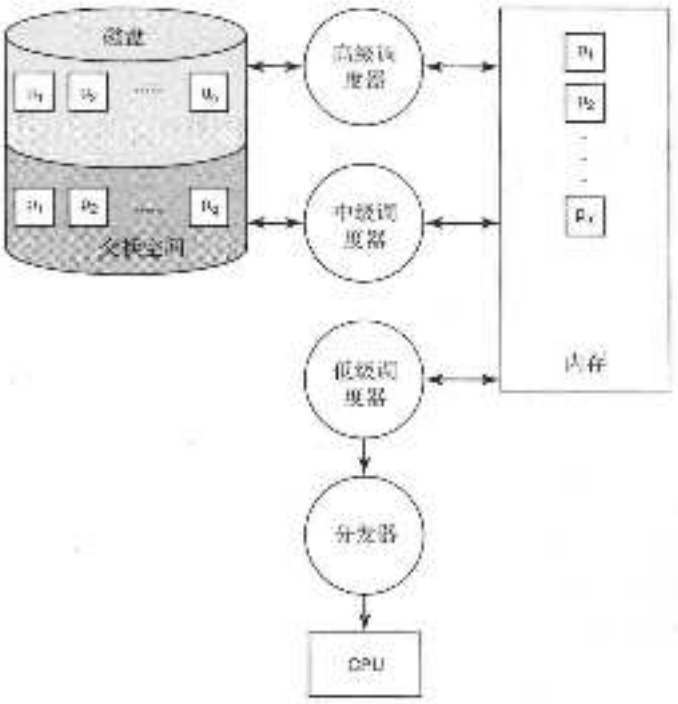


图 6-6 调度器的类型（u_i 表示磁盘上的用户程序；p_i 表示内存里的用户进程）

表 6-2 总结了不同环境里不同类型的调度器以及它们分别的作用。

表 6-2 调度器类型和作用

名称	环境	作用
高级调度器	面向批处理的操作系统	控制内存中的程序集合，以平衡系统资源的利用
加载器	所有操作系统	将用户程序从磁盘加载到内存
中级调度器	所有现代操作系统（分时的、交互式的）	平衡内存中的进程集合，以提高吞吐量
低级调度器	所有现代操作系统（分时的、交互式的）	调度内存中的进程在 CPU 上执行
分发器	所有操作系统	把低级调度器选中的进程的处理器状态装进 CPU 寄存器中

240
~
241

6.4 调度基础

在继续深究调度器之前先理解程序行为是很有用的。想象你的计算机上的一个从 CD 播放器播放音乐的程序。程序反复从 CD 盘中读取音轨（I/O 活动），然后把读到的音轨处理（处理器活动）后传给扬声器。我们发现，这就是程序的典型行为——在处理器上的突发性活动和 IO 设备上的突发性活动之间循环往复（参见图 6-7）。

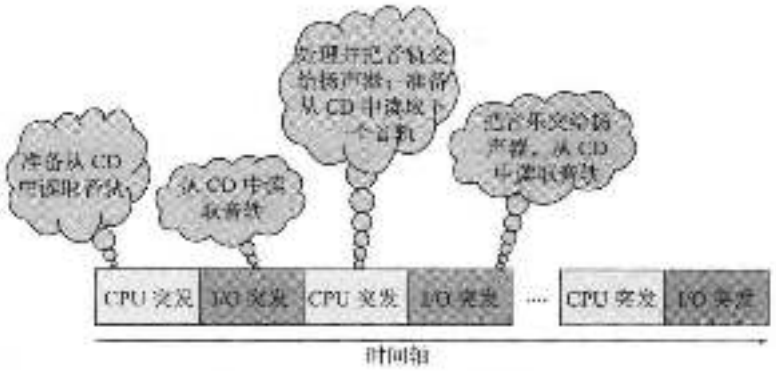


图 6-7 在 CD 播放器上播放音乐

我们将采用 CPU 突发这个术语来表示进程在不进行 I/O 调用的情况下所运行的一段时间。不正式地说，我们把一个进程的 CPU 突发定义为在进行 I/O 调用之前所连续进行 CPU 活动的时间区间。根据本章开始时的类比，CPU 突发就类似于你为了准备考试而连续阅读直到去冰箱里拿汽水的连续时间区间。类似地，我们采用术语 I/O 突发来表示一个进程用于处理 I/O 操作（譬如说，从 CD 中读取音乐文件）的一段连续的时间。值得一提的是在 I/O 突发中进程并不会用到处理器。在第 4 章中我们介绍了中断的概念，以及中断是怎么让处理器把注意力转移到诸如 I/O 完成一类的外界事件上的。在第 10 章中，我们将会讨论 I/O 设备和处理器之间的实际数据传输机制。而现在，为了让处理器调度的讨论尽量简单，我们不妨假设，进程在遇到 I/O 请求的时候就不再争夺 CPU 资源，直到 I/O 完成为止。

处理器调度器被划分为两个大类：非抢占式和抢占式。在非抢占式调度器里，一个进程要不然就一直执行到底，要不然就自己主动自愿地放弃处理器，以处理 I/O 请求。另一方面，在抢占式调度器里，调度器从当前进程手里把处理器抢走，交给另一个进程。不管是哪一种，调度的步骤均如下：

1）获得处理器的控制权。

2）把当前正在运行的进程的状态保存下来。

3）选择一个新的进程来运行。

4）把新选择的进程分发到处理器上运行。

最后一步，分发，是指把选定进程之前保存下来的状态加载到处理器寄存器的过程。

让我们来理解正在运行的程序，或者说是进程，它的状态是指什么。它包括程序执行到了什么地方（PC 值），处理器寄存器的内容是什么（假设这些寄存器中有一个就是栈指针），以及程序在内存中的印迹在哪里。除此以外，进程本身也许是刚刚被加载进内存等待运行，或者是等待 I/O，或者是正在运行，或者出于某种原因中止了执行，或者处于其他的什么状态中。

除此以外，调度器可能还了解进程的一些属性（内在的或者外来的），比如进程优先级、程序的到达时间，以及期望的运行时间。所有的状态信息都被聚集在一个数据结构中，称作 PCB（进程控制块，Process Control Block），如图 6-8 所示。每个进程都有一个 PCB，调度器则将所有 PCB 维护在一个链表中，该链表被称作就绪队列，如图 6-9 所示。

```
enum state_type {new, ready, running, waiting, halted};

typedef struct control_block_type {
   enum state_type state;          /* 当前状态 */
   address PC;                     /* 从哪里继续 */
   int reg_file[NUMREGS];          /* 通用寄存器的内容 */
   struct control_block *next_pcb; /* 链表指针 */
   int priority;                   /* 优先级 */
   address address_space;          /* 内存位置 */
          .
          .
          .
} control_block;
```

图 6-8 进程控制块（PCB）

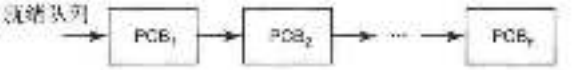


图 6-9 PCB 的就绪队列。PCB 保存了所有关于进程的信息。就绪队列通常则是个 PCB 的链表

PCB 包含了所有必需的描述进程的信息。它是操作系统的关键数据结构。我们将会在之后讨论内存系统和网络的章节中看到，PCB 是与进程相关的所有状态信息（如占用的内存、打开的文件以及网络连接）的集合。就绪队列是调度器中最重要的数据结构。该数据结构的有效表示和操作是调度器性能的关键。调度器的职责是快速地进行调度决策，然后离开，让 CPU 尽量用来运行用户程序。因此，一个关键问题是找出合适的用来衡量调度算法的效率的标准。直观地说，我们希望花在调度器里的时间占总 CPU 时间的百分比很小。我们将会在一个案例研究（见第 6.12 节）中看到，Linux 调度器是如何组织它的数据结构以确保高效的。

注意在处理器状态中并不包含 CPU 数据通路的内部寄存器（参见第 3 章和第 5 章）。原因将会在本小节的结尾处揭晓。

类似于进程控制块的就绪队列，等待 I/O 的进程的 PCB 队列也由操作系统维护（参见图 6-10）。

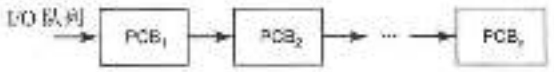


图 6-10 PCB 的 I/O 队列。在一个进程产生阻塞式 I/O 请求后，相应进程的 PCB 被移到 I/O 队列，等待 I/O 完成

出于本次讨论的目的，PCB 在就绪队列和 I/O 队列之间来回移动，取决于对应的进程是需要处理器还是 I/O 服务。CPU 调度器用就绪队列来调度用到处理器的进程。我们在本章中讨论的每个调度算法都假设存在这么一个就绪队列。就绪队列中的 PCB 的组织则取决于具体的调度算法。PCB 数据结构简化了我们在本节中之前的部分所提到的调度所用到的步骤。调度器知道哪个 PCB 对应于哪个正在运行的进程。把状态保存起来的步骤就是拷贝相关信息（在图 6-8 中列出）到当前运行的进程的 PCB 中的过程。类似地，调度器选择一个进程作为下一个运行在处理器上的候选者之后，把它分发到处理器上也就是简单地把处理器寄存器装上选定进程的 PCB 里所包含的信息。

从第 4 章我们了解到了系统调用（比如 I/O 操作）和中断都是不同类型的程序不连续性。

硬件对所有程序不连续性的处理都是类似的，即等待处理器到达一个干净的状态，然后处理该不连续性。一条指令执行完毕就是这里所说的干净的状态。处理器一旦达到这样的干净状态，处理器内部的寄存器（即程序员看不到的那些）也就不包含任何与当前程序相干的信息。因为调度器从一个进程切换到另一个进程的时机是明确定义的程序不连续性，使没有必要把程序员在指令集中看不到的处理器内部寄存器（在第3章和第5章中讨论过）保存下来。 242 ~ 244

表6-3总结了调度算法中重要的术语。

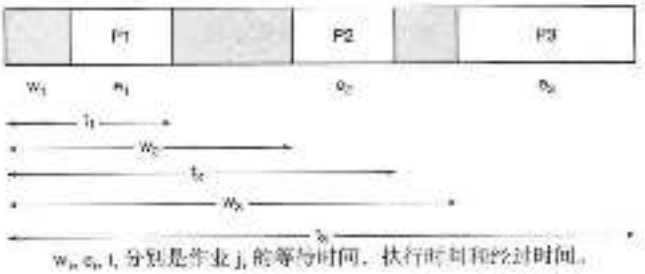
表6-3 调度术语

名 称	描 述
CPU突发	在请求I/O操作之前进程进行的连续CPU活动
I/O突发	CPU在I/O设备上发起的活动
PCB	进程控制块，保存进程（即运行中的程序）的状态
就绪队列	PCB队列，由表示准备好在CPU上运行的非阻塞内存里的进程的PCB组成
I/O队列	PCB队列，其表示已在等待发起I/O操作或者等待结束I/O操作而处于阻塞状态在内存里的进程PCB组成
非抢占式算法	允许当前调度在CPU上的进程自愿让出处理器（通过结束执行，或者进行I/O系统调用）的算法
抢占式算法	在发生外部事件（比如I/O完成中断，或者定时器中断）时会从当前调度的进程中强行抢走处理器的算法
颠簸	在就绪队列中的进程的动态内存使用超过了系统总内存容量时发生的一种现象

6.5 性能指标

在讨论调度算法时，我们认为术语作业和进程是同义词。作为操作系统的一部分，调度器也是程序，也需要在处理器上运行。调度器的最终的目标是运行用户程序。因此，运行用户程序时处理器在被合理利用，而在运行操作系统本身的时候就没有得到利用。这里就产生了问题，即评价调度算法的效率的指标是什么？CPU利用率是指处理器繁忙时间的百分比。尽管这个百分比是个有用的指标，但它并没有说明处理器到底在做什么。所以，让我们看看有没有其他指标。指标既可以是以用户为中心的也可以是以系统为中心的。吞吐量是个以系统为中心的指标，表示每单位时间内所完成的作业个数。平均周转时间是另一个以系统为中心的指标，用于测量作业进入和离开系统平均所花的时间。平均等待时间是另一个以系统为中心的指标。作业的响应时间则是以用户为中心的指标，测量给定作业的经过时间。

图6-11显示了在处理器上调度三个进程P1、P2和P3的时间轴。假设所有进程都在时刻0开始。为了本次讨论方便，我们假设在阴影部分里处理器忙着做别的事情，与运行这些进程无关。 245



w_j, e_j, t_j 分别是作业 j 的等待时间、执行时间和经过时间。

图6-11 调度三个进程P1、P2和P3的时间轴

参照图 6-11，我们将刚刚定义的术语量化如下：

吞吐量 = 3/t_3 作业 / 秒

平均周转时间 = (t_1 + t_2 + t_3) / 3 秒

平均等待时间 = ((t_1 − e_1) + (t_2 − e_2) + (t_3 − e_3)) / 3 秒

推广到 n 个作业，我们得到

吞吐量 = n/T 作业 / 秒，其中 T 是这 n 个作业完成的总经过时间

平均周转时间 = (t_1 + t_2 + ... + t_n) / n 秒

平均等待时间 = (w_1 + w_2 + ... + w_n) / n 秒

响应时间就是每进程的周转时间，因此，有下列等式：

$R_{P1} = t1$

$R_{P2} = t2$

$R_{P3} = t3$

...

$R_{Pn} = tn$

响应时间的方差[○]也是个有用的指标。除了以上定量指标以外，还应当提一下关于调度算法的两个定性指标：

246

- **饥饿** 在任何作业组合中，调度策略都应该确保所有的作业一直有进展。如果出于某种原因，一个作业并没有任何进展，我们就把这种情况称作饥饿。这种情况的定量表现是特定作业的响应时间没有上界。
- **护送效应** 在任何作业组合中，调度策略应当努力预防长时间运行的作业完全占据 CPU 的使用。如果出于某种原因，作业的调度符合固定的规律（类似于军队中的护卫），我们就把这种情况称作护送效应。这个现象的定量表现是作业的响应时间的方差很大。

我们将会在随后几节中讨论若干调度算法。在如此做之前，请先注意以下几点：

- 在所有调度算法中，我们假设从一个进程切换到另一个进程的时间可以忽略不计，以简化调度的时序图。
- 我们提到过，进程可能会在生命周期中，在 CPU 和 I/O 请求之间来回切换。自然地，I/O 请求在不同时间可能是关于不同设备的（输出到屏幕、从磁盘读取、从鼠标输入，等等）。但是，由于本章的重点是 CPU 调度，为了简便起见我们只画出一个 I/O 队列。
- 还是为了把重点放在 CPU 调度上，我们假设调度 I/O 请求采用一种简单的模型（先到先服务）。换句话说，CPU 调度器在调度进程时使用的内在或者外来属性并不适用于 I/O 调度。I/O 请求就按照它们被进程请求的顺序处理。

表 6-4 总结了从调度的角度来说有意义的性能指标。

表 6-4 性能指标小结

名称	记法	单位	描述
CPU 利用率	—	%	CPU 忙碌的时间百分比
吞吐量	n/T	作业 / 秒	系统中心指标，表示时间 T 中执行的作业数 n
平均周转时间 (t_{avg})	(t_1 + ... t_n)/n	秒	系统中心指标，表示作业完成的平均时间

○ 响应时间的方差是可能的响应时间离期望值的差的平方的平均数。

（续）

名称	记法	单位	描述
平均等待时间 (w_{avg})	$((t_1-e_1)+(t_2-e_2)+\ldots+(t_n-e_n))/n$	秒	系统中心指标，表示作业经历的平均等待时间或者 $(w_1+w_2+\ldots+w_n)/n$
响应时间 / 周转时间	t	秒	用户中心指标，表示特定作业 t 的周转时间
响应时间方差	$E[(t_i-e_i)^2]$	秒^2	用户中心指标，表示给定进程的实际响应时间与其期望值的统计差异
饥饿	—	—	用户中心的定性指标，表示特定的一个或一组进程由于调度器的某种内在特性而拒绝服务
护送效应	—	—	用户中心的定性指标，表示由于调度器的某种内在特性而对某些进程产生负面效果

6.6 非抢占式调度算法

我们之前已经提到过，非抢占式算法中，在当前进程被调度到 CPU 上运行之后，调度器就不控制当前进程了。只有在当前进程通过终止或者发出阻塞式系统调用（比如文件 I/O 请求）时调度器才能取回控制权。在本节中，我们将会考虑属于此类的 3 种不同算法：FCFS、SJF，以及优先级调度。

6.6.1 先到先服务

本算法中用到的内在属性是进程的到达时间，即运行一个应用程序的时间。例如，如果你在 t_0 时间运行 winamp，之后在 t_1 时间运行 realplayer，那么 winamp 的到达时间对于调度器来说就较早。因此，在两个程序的整个生命周期中，只要两个程序都可以运行，winamp 一定会被调度器选中执行，因为它的到达时间比较早。记住，这个 winamp 的“优先权”甚至在它从 I/O 完成之后返回就绪队列时也会持续起作用。例 6-1 展示了先到达的进程与就绪队列里其他进程相比所享受的优势。

247 / 248

图 6-12 展示了一组进程，并在图的上半部分画出了它们的活动。每个进程的活动都在 CPU 突发和 I/O 突发之间交替。例如说，P2 进行 1 个单位的运算，然后 2 个单位的 I/O，并在整个生命周期里一直重复下去。图 6-12 的下半部分给出了对这些进程在 CPU 和 I/O 上进行先到先服务（FCFS，First-Come First-Served）调度的时间轴。在任何时间，有恰好一个进程在 CPU 上执行，另一个在进行 I/O 活动。假设每个进程需要处理两个 CPU 突发，中间穿插一次 I/O 突发。所有 3 个进程在时间轴的开始处均准备好运行。但是，P1 是第一个达到的，然后再是 P2 和 P3。因此，在可以选择的时候，由于先到先服务原则，调度器总是会优先选择 P1，然后是 P2 和 P3。P1、P2 和 P3 的等待时间分别是 0、27 和 26。

这个算法有个很好的性质，即任何进程都不会饥饿——也就是说，算法中没有会导致任何进程拒绝服务的内在偏向。我们随后会看到，并不是所有算法都具有此性质。但是，由于该属性的本质，响应时间的方差可能会很大。例如，如果在一个长作业到达之后紧接着来了一个短作业，则短作业的响应时间就会很糟糕。该算法也由于图 6-12 中描绘的护送效应而导致低下的处理器利用率。术语护送效应（convoy effect）主要来自军事意义上的护卫（convoy），护卫表示一组车辆一起行动。当然了，在军事里，护卫是个好事，因为这些车辆在紧急情况

下可以互相援护。在高速公路上你在单车道路段被挡在一辆慢车后面时，就不自觉地属于一个“护卫”。对CPU来说，短作业（P2和P3）就被长作业（P1）挡到了后面。护送效应是先到先服务调度算法所特有的问题，源自于这种调度思想的本质。我们中的很多人都经历过在结账时发现前面的顾客有满满一车的货物，而自己只买那么几样东西的情况。不幸的是，护送效应根源于先到先服务的调度思想，因此根据它的本质，它就没有对较短的作业给予任何首选考虑。

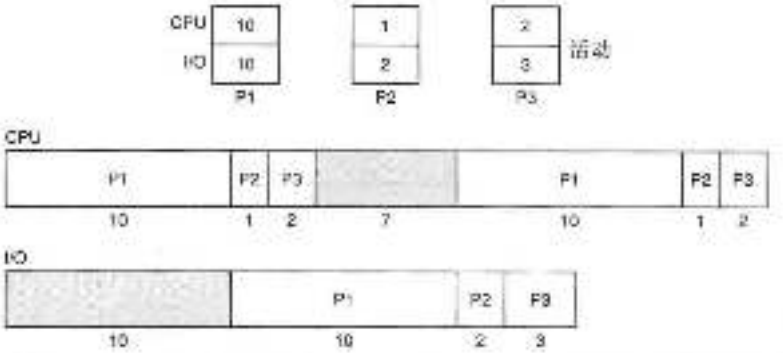


图6-12 先到先服务算法的护送效应，短作业（P2，P3）由于先到先服务算法的特点而卡在一个长作业（P1）之后

249

例6-1 考虑一个非抢占式的先到先服务（FCFS）进程调度器。在调度队列中有3个进程，到达顺序是P1、P2和P3。在选择接下来运行哪个进程时总是根据到达顺序进行选择。调度从t=0开始，CPU和I/O突发时间分别如下所示：

	CPU突发时间	I/O突发时间
P1	8	2
P2	5	5
P3	1	5

每个进程都在完成以下三个动作的操作之后结束：

1. CPU突发
2. I/O突发
3. CPU突发

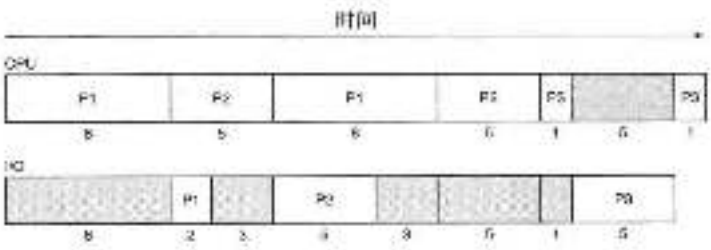
a. 展示FCFS调度算法导致的CPU和I/O时间轴，从t=0开始，直到三个进程均完成为止。

b. 各个进程的响应时间是多少？

c. 各个进程的等待时间是多少？

答：

a.



注意，在 t=8 时，P2 和 P3 都在就绪队列中，而 P3 进行了一次 I/O 请求，让出了处理器。调度器根据到达时间选择了到达较早的 P2 来处理器上运行。t=10 时，P1 完成 I/O，回到就绪队列；P3 这时候已经在就绪队列里了。但是，就算是不抢占的调度，P1 在就绪队列中等待 P3 之前，凭什么 t=13 时调度器选择运行的是还在运行的 P1。 250

b. 根据每个进程从到达到离开系统的总时间，计算响应时间。

响应时间（P1）= 21

响应时间（P2）= 26

响应时间（P3）= 33

c. 每个进程在 CPU 上执行完毕后进行 I/O 操作时都不会进行有用作业。因此，我们通过在总周转时间（完整地运行时间）中减去花费在有用作业的时间，来计算等待时间。例如说，对 P1 来说，有用作业时间

= 第一个 CPU 突发 + I/O 突发 + 第二个 CPU 突发

= 8 + 2 = 8

= 18

因此，P1 的等待时间如下：

等待时间（P1）=（21-18）= 3

类似地，

等待时间（P2）=（26-15）= 11

等待时间（P3）=（33-7）= 26

如同我们之前已经提到的，调度器在进程的整个声明周期里，选择哪个进程在 CPU 运行时都会考虑到达时间。而且，这个内在属性只对 CPU 调度有意义，而不用于 I/O 调度。这两点可以从以下例子中看出。

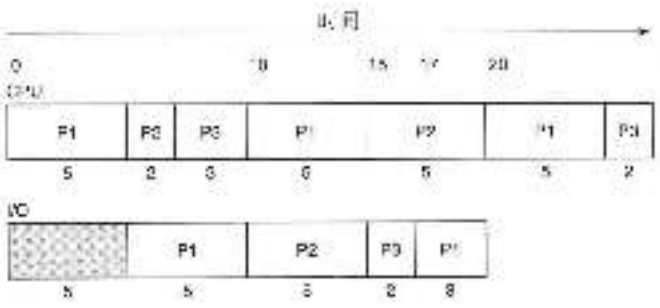
例题 2 考虑一个先到先服务的调度器。假设调度器共有 3 个进程，它们都是各自运行。调度算法要求调度器在选择进程时总是依照到达时间进行选择。假设 P1、P2 和 P3 依照这个顺序进入系统，调度从 t=0 时刻开始。

这三个进程的 CPU 和 I/O 突发如下所示：

	CPU	I/O	CPU	I/O	CPU	
P1	5	5	5	3	5	P1 完成
P2	2	5	5			P2 完成
P3	3	2	2			P3 完成

从 t=0 开始，画出先到先服务算法为该组进程分配 CPU 和 I/O 时间的图，并列出 3 个进程完成的时间。 251

答：



注意：

1. 在 t=15 时刻，P3 和 P1 都准备进行 I/O。但是，P3 先进行 I/O 请求（在 t=10 时刻），而 P1 则

在之后才发出请求（在 t=15 时刻）。我们之前已经提过，I/O 请求是按照请求顺序来处理的，因此，P3 先被处理，然后才是 P1。

2. 在 t=20 时刻，P1 和 P3 都需要在 CPU 上运行。实际上，P3 在 t=17 时刻完成 I/O，而 P1 在 t=20 时刻刚刚完成 I/O。但是，P1 在前争口挑出，被选择在 CPU 上运行，因为 CPU 调度器在进行调度决定时总是遵照进程到达的时间顺序。

6.6.2 最短作业优先

术语最短作业来自于店里的调度决定——比如说，在汽车修理店。不幸的是，这个术语用在 CPU 调度上单从字面意思解释有时候就词不达意了。因为 CPU 调度器并不知道某个程序的具体行为是怎么样的，它只能利用已知的部分知识来进行调度。在最短作业优先（SJF，Shortest Job First）算法的情形中，所用到的知识就是所需的 CPU 突发时间，这是每个进程都具有的内在属性。我们知道，每个进程依次遇到 CPU 和 I/O 突发活动。最短作业优先调度器查看当前可以运行的进程集合所需的 CPU 突发时间，并选择需要 CPU 突发最短的一个。回忆本章开始时的类比，你在给妈妈打电话之前启动洗衣机和微波炉。最短作业优先调度器采用相同的原则挑选短的作业先做。现实中，调度器在程序启动时不知道它的 CPU 突发时间，但是它可以根据之前的 CPU 突发来推断进程的期望突发时间。

最短作业优先算法会让较短的作业获得更好的响应时间。而且，它也不会有先到先服务算法的护送效应，因为它优先选择较短的作业。实际上该算法也被证明了能得到最好的平均等待时间。图 6-13 展示了由最短作业优先来处理图 6-12 中的相同进程的时间轴。三个进程 P1、P2 和 P3 的等待时间分别是 4、0、9。

252

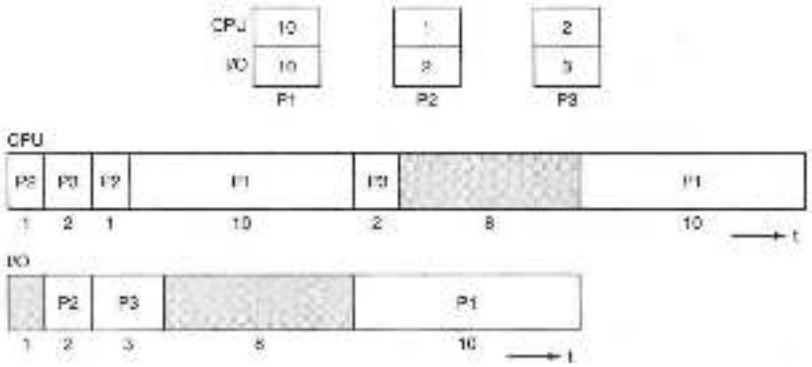


图 6-13 最短作业优先算法通过优先对待较短作业，避免了先到先服务的护送效应

注意最短作业 P2 在这种调度策略下得到了最好的服务。在 t=3 时刻，P2 刚刚完成它的 I/O 突发。幸运的是，P3 也刚刚完成了它的 CPU 突发。这时候，尽管 P1 和 P2 都准备好在 CPU 上运行了，调度器却会因为 P2 需要的 CPU 突发较短而优先选择 P2 而不是 P1。在时刻 t=4，P2 完成了它的第一个 CPU 突发。由于没有其他更短的作业可供调度，P1 得到了在 CPU 上运行的机会。在 t=6 时刻，P3 刚刚完成它的 I/O 突发，准备好在 CPU 上调度执行。但是，P1 正在处理器上执行。由于调度器是非抢占式的，P3 只好等待 P1 主动放弃 CPU（P1 在 t=14 时这么做了）。

最短作业优先调度可能会导致长作业的饥饿。在前面的例子里，在轮到 P1 运行之前，更新的较短的作业可能会进入系统。因此，P1 可能会等待很长时间，甚至一直等下去。为了解

决此问题，一个被称作老化的技术会优先选择已经待在就绪队列中等待了很久的作业（与其他较短的作业相比）。基本上这里的想法就是让调度器为每个作业增加一个新的属性，即它进入调度列表中的时刻。当一个作业的年龄超过了一定界限，调度器就会忽略最短作业优先原则，而优先选择这样的作业来调度。

例 6-3 考虑一个非抢占式的最短作业优先（SJF）进程调度器。假设在调度队列里有三个进程，且都准备好运行。该调度算法要求每次都选择可以运行的进程中最短的一个。调度从 t=0 时刻开始。三个进程 253
的 CPU 和 I/O 突发执行如下所示：

	CPU	I/O	CPU	I/O	CPU	
P1	4	2	4	2	4	P1 完成
P2	5	2	5			P2 完成
P3	2	2	2	2		P3 完成

每个进程在上面列出的 CPU 和 I/O 突发完成以后就退出系统。

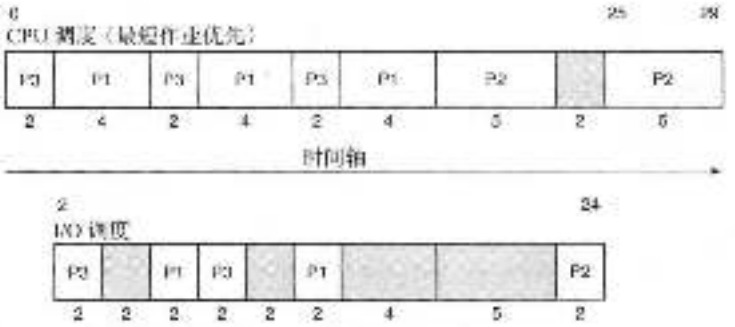
a. 画出在最短作业优先调度下，从 t=0 开始为 CPU 和 I/O 时间轴，来列三个进程全部退出系统。

b. 各个进程的等待时间是多少？

c. 整个系统的平均吞吐能是多少？

答：

a.



b. 我们采用例 6-1 相同的方法计算各个进程的等待时间：

等待时间（P1）=（18 - 16）= 2

等待时间（P2）=（30 - 12）= 18

等待时间（P3）=（14 - 10）= 4

c. 总时间 = 30

吞吐量 = 完成的进程个数 / 总时间

= 3 / 30

= 1 / 10 进程每单位时间

6.6.3 优先级

由于调度的目的，多数操作系统都会给每个进程赋予一个外来属性——优先级，这是一个小整数值，用来表示它与其他进程相比的相对重要性。比如说，在 Unix 操作系统中每个用户级进程开始时都有固定的默认优先级。就绪队列包含多个子队列，每个对应着一个优先级，如图 6-14 所示。

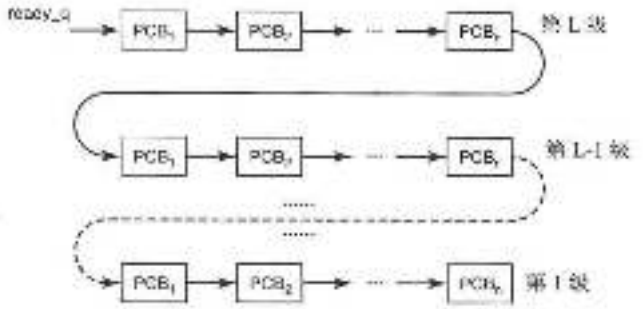


图 6-14 优先级调度器的多个就绪队列。调度器可能随着时间推移将进程在不同的优先级之间移动

每个优先级内部的调度是先到先服务的。新的进程放在与它们优先级相对应的队列中。因为优先级是个外来属性，调度器可以完全控制不同进程的优先级分配。这个调度方式与先到先服务或者最短作业优先相比十分灵活。例如，调度器可以根据用户的级别来分配优先级。从数据中心[⊖]运营者的角度来说这可能特别有吸引力，那里不同的用户可能会愿意为他们各自的作业支付不同的价格。

优先级是个给不同用户提供差异服务的自然方式。不仅仅在处理器调度中是如此，在日常生活的几乎每个方面都是这样。一个例子是，公司负责处理客服请求的呼叫中心会把不同的拨打者依据它们的档案放进不同的队列里。难缠的拨打者被塞进与更受欢迎的拨打者相比更慢的队列里。类似地，航空公司用头等舱、商务舱和经济舱作为给客户划分优先级的方式。

254 ~ 255

只需回想一下，就能发现最短作业优先其实只是优先级调度的一个特例，其中优先级

$$L-1/\text{CPU 突发时间}$$

因此，类似于最短作业优先，优先级调度器也有低优先级进程会饥饿的问题。显然，我们可以通过给每个进程分配显式的优先级的方法来解决这个问题。类似于我们在讨论最短作业优先时所提到的，在一个进程的年龄达到某个界限以后，调度器会人为地提升进程的优先级。

想想看就会意识到先到先服务调度也是一种基于优先级的算法。不过调度器是用进程的到达时间来作为它的优先级的。因此，在先到先服务算法中新进程的优先级一定不会比原有的进程高，这也就是为什么先到先服务算法不会有饥饿问题的原因。

由于与先到先服务算法的相似性，基于优先级的算法也可能展现出护送效应。让我们来看看这为什么会发生：如果一个高优先级的进程恰好是个长时间运行的进程，我们可能就会处于类似于先到先服务调度中的状况。但是，在先到先服务算法里，进程的优先级一旦确定了就绝不更改，但是基于优先级的调度器可不必如此，用来克服饥饿问题的机制（即根据进程年龄来提高优先级）也有助于消除护送效应。

6.7 抢占式调度算法

这一类调度算法同时意味着两件事情。一方面，调度器可以随时得到处理器的控制权，而完全不用让正在运行的进程知道。另一方面，调度器能够把当前运行的进程的状态保存下

⊖ 作为一种没有高性能计算资源的用户获取该资源的方式，数据中心变得越来越普及了。请想想乌班、微软、惠普和 IBM 等公司都处在提供此类服务的第一线。云计算是提供此类服务的业界流行语。

来，以便能够正确地从被抢占的地方开始恢复执行。回到本章开始时我们的类比上去，当你在给妈妈打电话而洗衣机发出蜂鸣的时候，你会让妈妈等一会儿，而脑中则记下待会儿继续通话的时候该从哪里说起。

原则上，上一节中讨论的任何算法都能被改造成抢占式的。为了达到这个目的，在先到先执行算法中，每当一个进程在完成I/O重新进入就绪队列时，调度器可以决定抢占当前正在执行的进程（如果它的到达时间比前者要晚）。类似地，对于最短作业优先和优先级调度，调度器也在一个新进程或者刚刚完成I/O的进程进入就绪队列时重新进行评估，以决定要不要抢占当前进程的执行。

最短剩余时间优先（Shortest Remaining Time First，SRTF）是最短作业优先调度器的一种特殊情况，但是加入了抢占的概念。调度器估计每个进程的运行时间，当一个进程回到就绪队列，调度器计算作业的剩余处理时间。依据它的计算结果，调度器将进程放在就绪队列的合适位置。如果该进程的剩余时间比当前在运行的进程还要少，调度器就抢占后者以运行前者。

例6-4 考虑以下四个争夺CPU的进程。调度器采用最短剩余时间优先算法。表中给出了每个进程的到达时间。 256

进程	到达时刻	执行时间
P1	T_0	4 ms
P2	T_0 + 1 ms	2 ms
P3	T_0 + 2 ms	2 ms
P4	T_0 + 3 ms	3 ms

a. 给出从T_0时刻开始的调度方案。

答：

1）在T_0时刻，P1开始运行，因为没有其他进程。

2）在T_0+1时刻，P2到达。下表展示了此刻P1和P2的剩余所需运行时间：

进程	剩余时间
P1	3 ms
P2	2 ms

调度器切换到P2。

3）在T_0+2时刻，P3到达。下表展示了此刻P1，P2和P3的剩余所需运行时间：

进程	剩余时间
P1	3 ms
P2	1 ms
P3	2 ms

调度器继续运行P2。

4）在T_0+3时刻，P4到达。P2完成并离开。下表展示了此刻P1、P3和P4的剩余所需运行时间：

进程	剩余时间
P1	3 ms
P3	2 ms
P4	3 ms

257 调度器选择P3，并且在接后的2 ms中一直运行至P3结束。

5）在T_0+5时刻，P1和P4是仅有的两个进程，剩余所需运行时间如下：

进程	剩余时间
P1	3 ms
P4	3 ms

平手了，调度器就根据到达顺序（P1在先，P4在后）来进行选择，这种选择机会降低平均等待时间。下表展示了最少剩余时间优先调度的结果：

从T_0开始的时间	0	1	2	3	4	5	6	7	8	9	10	11	12
在运行的进程	P1	P2	P2	P3	P3	P1	P1	P1	P4	P4	P4		

b. 进程的平均等待时间是多少？

答：

$$\text{响应时间} = \text{完成时间} - \text{到达时间}$$
$$R_{p1} = 8-0 = 8\text{ms}$$
$$R_{p2} = 3-1 = 2\text{ms}$$
$$R_{p3} = 5-2 = 3\text{ms}$$
$$R_{p4} = 11-3 = 8\text{ms}$$
$$\text{等待时间} = \text{响应时间} - \text{执行时间}$$
$$W_{p1} = R_{p1}-E_{p1} = 8-4 = 4\text{ ms}$$
$$W_{p2} = R_{p2}-E_{p2} = 2-2 = 0\text{ ms}$$
$$W_{p3} = R_{p3}-E_{p3} = 3-2 = 1\text{ ms}$$
$$W_{p4} = R_{p4}-E_{p4} = 8-3 = 5\text{ ms}$$

c. 平均等待时间是多少？

答：

$$\text{总等待时间} = W_{p1} + W_{p2} + W_{p3} + W_{p4} = 10\text{ ms}$$
$$\text{平均等待时间} = 10/4 = 2.5\text{ ms}$$

258 d. 如果抵成先到先服务调度策略，处理相同的一组进程的调度方案会是什么样子？

答：

从T0开始的时间	0	1	2	3	4	5	6	7	8	9	10	11	12
在运行的进程	P1	P1	P1	P1	P2	P2	P3	P3	P4	P4	P4		

e. 先到达先处理算法的平均等待时间是多少？

答：

$$\text{响应时间} = \text{完成时间} - \text{到达时间}$$
$$R_{p1} = 4-0 = 4\text{ms}$$
$$R_{p2} = 6-1 = 5\text{ms}$$
$$R_{p3} = 8-2 = 6\text{ms}$$
$$R_{p4} = 11-3 = 8\text{ms}$$
$$\text{等待时间} = \text{响应时间} - \text{执行时间}$$
$$W_{p1} = R_{p1}-E_{p1} = 4-4 = 0\text{ ms}$$
$$W_{p2} = R_{p2}-E_{p2} = 5-2 = 3\text{ ms}$$
$$W_{p3} = R_{p3}-E_{p3} = 6-2 = 4\text{ ms}$$
$$W_{p4} = R_{p4}-E_{p4} = 8-3 = 5\text{ ms}$$

答：

总等待时间 = $W_1 + W_2 + W_3 + W_4$ = 12 ms

平均等待时间 = 12/4 = 3 ms

注：可以看出，与先到先服务相比，最短剩余时间优先算法的平均等待时间是较低的。

6.7.1 轮转调度器

让我们找出适合分时环境的调度器的特点。分时环境的名字就隐含了一点，即每个进程都应该得到处理器时间的一部分。因此，非抢占式的调度器就不适合这种环境。

分时环境特别适合采用轮转（Round Robin，RR）调度器。假设有 n 个就绪的进程，调度器把处理器分成时间段，一般称作时间片，然后分配给各个进程（参见图 6-15）。你能看出这类调度器与本章开始时在关于准备考试的类比里提到的第一种策略是有联系的。轮转调度器在就绪进程之间轮流切换也不是没有代价的。选择合适的时间段 q 时一定要考虑上下文切换时间。

259

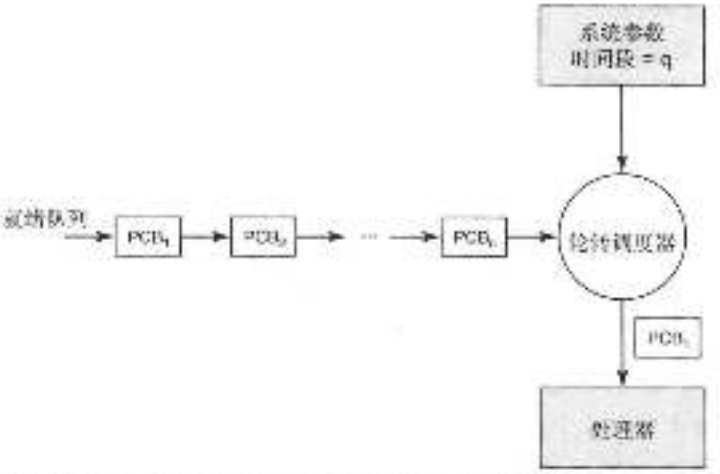


图 6-15 轮转调度器。每个进程得到一个时间段的时间以在处理器上运行，然后被上下文切换走，以使运行下一个进程

就绪队列里的每个进程都得到处理器的一个时间片 q。当时间片用完了，当前调度的进程就被放在就绪队列的尾部，而就绪队列中的下一个进程则被调度到处理器上。

我们也可以在轮转调度器和先到先服务调度器之间找出联系。先到先服务调度器是轮转调度器的一种特殊情况，即时间段是无穷长。处理器分享也是轮转调度的一种特殊情况，此时每个进程都得到处理器的 1 个单位的时间，因此每个进程都能假想自己独占运行在一个速度是 1/n 的处理器上。共享处理器的观念对于证明调度方面的理论结果很有帮助。

让轮转调度器工作 让我们考虑一个轮转调度器的例子。图 6-16 给出了三个进程的 CPU 和 I/O 突发情况。假设进程的到达顺序是 P1、P2、P3；并且调度器用的时间片大小为 2。每个进程都在图中所示的 CPU 和 I/O 突发完成之后离开系统。

	CPU	I/O	CPU	I/O	
P1	4	2	2		P1 完成
P2	2	2	2		P2 完成
P3	2	2	4	2	P3 完成

图 6-16 轮转调度器示例中的 CPU 和 I/O 突发

在时刻 t=0 时，调度队列如图 6-17 所示。需要注意的关键地方是此刻的顺序不一定会一直保持下去，因为进程在 CPU 和 I/O 队列之间来回移动。当一个进程重新加入 CPU 或者 I/O 队列时，总是会插到队尾，因为进程没有任何内在的优先级。

260

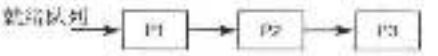


图 6-17 图 6-16 中例子里 t=0 时刻的就绪队列

图 6-18 展示了前 17 个时间单位（从 t=0 到 t=16）的调度情况。注意 P2 只用到了分配给它的 2 个时间单位中的 1 个，因为它在 3 个单位的 CPU 突发之后就进入了 I/O 突发。换句话说，在轮转调度中，时间片是进行下一次调度决策之前连续使用 CPU 的时间上限。而且，在 t=12 时刻，P2 刚刚完成它的 I/O 突发，准备好回到 CPU 队列。此时 P1 正在它的 CPU 时间片当中，而 P3 也在就绪队列里。因此，P2 回到就绪队列里，排在 P3 之后，如图 6-19 所示。



图 6-18 图 6-16 中例子的轮转调度结果

例 6-5 如果采用轮转调度，图 6-16 中的三个进程的等待时间分别是多少？

答：

类似于例 6-1 和例 6-3。

等待时间 =（响应时间 - 在 CPU 或者 I/O 上花费的使用时间）

P1 的等待时间 = (13 – 8) = 5

P2 的等待时间 = (17 – 7) = 10

P3 的等待时间 = (17 – 10) = 7

轮转算法的细节 让我们把注意力转移到调度器如何得到处理器的控制权上。有一个硬件装置，即定时器，会在经过时间 q 之后中断处理程序。处理定时器中断的中断处理程序是轮转调度器的一部分。图 6-20 展示了这个系统的各个层面。在任何时候，处理器要不然在运行调度器，要不然在运行某个用户程序。考虑某个正运行在处理器上的用户程序。在遇到中断时，时钟中断处理程序会获得处理器的控制权（参考在第 4 章中阐述的发生中断时的处理步骤）。处理程序将正在运行的用户程序的上下文保存进相应的 PCB，并把控制权移交给调度器。这个切换过程被称作上行调用，它使得调度器可以运行它自己的调度算法，以选择下一个分发到处理器上的进程。一般来说，上行调用是指从系统软件的低层调用上层的某个系统函数（即过程调用）。

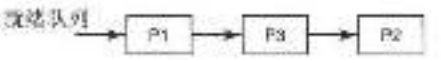


图 6-19 图 6-18 的例子中 t=12 时刻的就绪队列

图 6-21 总结了轮转调度算法。算法由五个过程组成：分发器、定时器中断处理程序、I/O 请求陷入、I/O 完成中断处理程序、以及进程终止陷入处理器。

分发器只需要把就绪队列头部的进程分发到处理器上，并且把定时器设定到时间段 q 就好了。当前调度的进程可能会以以下三种方式放弃处理器：进行 I/O 请求、结束运行，或者是用完在处理器上分配到的时间。I/O 请求陷入是第一个选项的表现。比如说，如果当前调度的进程产生了一个从磁盘读取文件的请求。这种情况下，I/O 请求陷入会把当前调度的进程的状态保存到进程控制块里，把它移动到 I/O 队列去，然后上行调用分发器。定时器中断处理程序则是当前进程时间片用完的表现。硬件定时器会中断处理程序，从而调用该处理程序。如图 6-21

所示，定时器中断处理程序会把当前进程的状态保存到进程控制块里，把进程控制块移动到就绪队列的末尾，然后上行到用分发器。在 I/O 请求完成的时候，处理器会得到一个 I/O 完成中断。该中断会导致 I/O 完成处理程序被调用。这个处理程序的作用就有点复杂了。某个进程还在处理器上执行，而且它虽然在执行，就说明它的时间片还没用完。处理程序简单地把当前执行的程序的状态保存到进程控制块。I/O 完成代表的是当初进行 I/O 请求的那个进程。处理程序把该进行 I/O 请求的进程对应的进程控制块移动到就绪队列的末尾，并上行调用分发器。分发器会做必要的操作，以分发被 I/O 完成中断打断的那个进程（因为它还有处理器时间）。最后，进程终止处理程序只需要把进程的控制块释放，从就绪队列中移除，然后上行调用分发器即可。

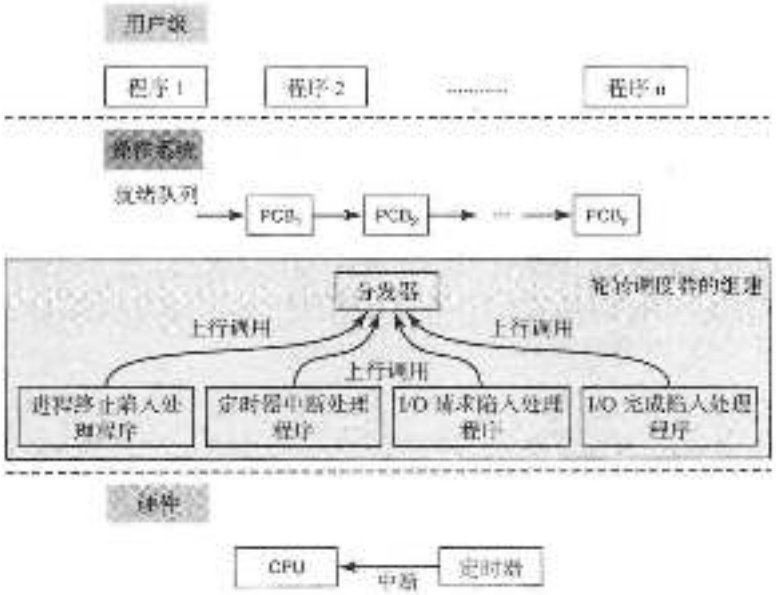


图 6-20 采用了轮转调度器的系统的不同层面。轮转调度器的不同组件在当前运行的进程主动要求或者系统状态变化时调用

```
分发器：
  得到就绪队列队首的进程控制块
  设定定时器
  分发
定时器中断处理程序：
  把上下文保存到进程控制块
  把进程控制块移到就绪队列尾部
  上行调用分发器
I/O 请求陷入：
  把上下文保存到进程控制块
  把进程控制块移到 I/O 队列；
  上行调用分发器
I/O 完成陷入处理程序：
  把上下文保存到进程控制块
  把完成了 I/O 的进程的控制块移到就绪队列；
  上行调用分发器
进程终止陷入处理程序：
  释放进程控制块
  上行调用分发器
```

图 6-21 轮转调度算法

6.8 结合优先级和抢占

诸如Unix和Microsoft Windows XP等通用系统在CPU调度算法中结合使用优先级和抢占的概念。进程有对应的优先级，在创建的时候由操作系统决定。它们被安排进一个多级的就绪队列，类似于图6-14中的那样。另外，对于两个特定优先级的进程来说，操作系统共同一个固定时间段的轮转调度器。在没有更高优先级的进程时将会处理低优先级的进程。为了避免饥饿，操作系统可能会周期性地提升长时间没有被执行的进程的优先级（即老化）。另外，操作系统可能会给高优先级进程更大的时间片。这样的调度器的软件系统的结构如图6-20所示。

6.9 元调度器

在有些要应对多种需求的系统中，调度环境可能由多个就绪队列组成，每个都有各自的调度算法。例如，有个队列用来处理前台的交互式作业（并且有个相应的调度器），另一个队列用来处理后台的批处理作业（并且也有个相应的调度器）。交互式的作业按照轮转的方式来调度，而后台作业则采用一个基于优先度的先到先服务算法进行调度。在它们之上则是一个元调度器，把时间片分给这些调度器（如图6-22所示）。参数Q是元调度器给下一层的两个调度器分配时间段。元调度器通常也提供因为应用程序（被不同进程所代表）需求的动态变更而把作业在两个低级调度器的就绪队列之间移动的功能。

261
·
264

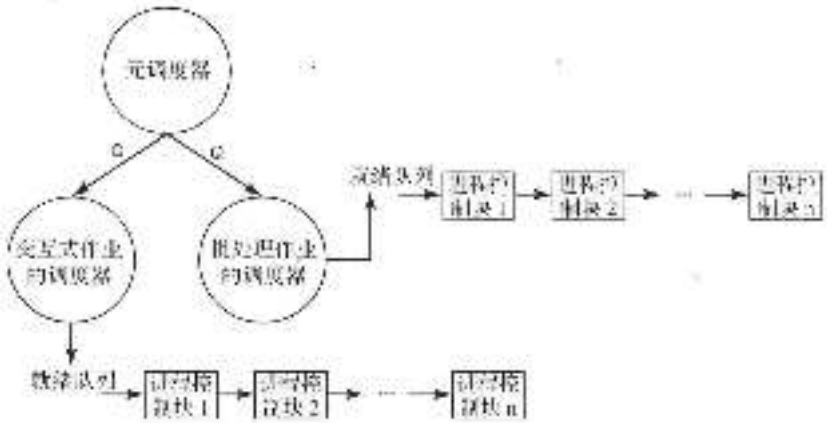


图6-22 元调度器。每类作业有一个调度器。元调度器给这些进程级调度器分配时间片

此元调度器为一个推广被用在网格计算中，这是一种有点游牧性的基于互特网的计算基础设施，因它能给供给用户使用高性能计算资源而不必真的拥有它们，而在获得吸引力。网格计算使用可能在地理上分散，并且穿过多个行政单位的计算资源。网格这一术语象征对计算资源的安排类架与分配电力的“电网”。网格的用户把作业提交给这个环境，然后作业被分布式的计算资源所执行。术语高性能指的是可能有几台甚至几千个计算资源协同工作以满足应用的计算需要。例如说，一个这种基础设施的应用可能是全球气候模式。建议有兴趣的读者参阅关于网格计算的专业书本[Foster，2003]。

6.10 评价

我们应当怎么评价调度器的效率呢？任何一个操作系统实体的关键属性是快速提供用户所请求的资源，然后让开。调度算法根据我们之前在讨论具体的部署环境时提到的性能指标进行评价。根据诸如分时的、面向批处理的和多道编程的等属性刻画不同的环境。这些环境 265
针对不同的市场力量或者计算的应用领域。例如，我们现在可以区分出以下计算扮演着重要角色的领域：

- **桌面和膝上型计算机** 我们对这种个人计算机领域再熟悉不过了。我们用这个类别来称呼那些用来做字处理、软件开发一类事情的计算机。
- **服务器**：这是邮件服务器、文件服务器和 Web 服务器的领域。
- **商用**：这类包括电子商务和金融应用，经常被用作企业计算的同义词。
- **高性能计算**（HPC，High Performance Computing） 这是需要高性能计算资源来解决科学和工程问题的领域。
- **网格** 该领域包含高性能计算的所有元素，外加一点：计算机可能是地理上分开的（比如说，一些在东京而另一些在亚加罗尔），并且可能跨过行政界限（比如说，一些属于佐治亚理工而另一些属于麻省理工）。
- **嵌入式** 随着包括手机、iPOD 和 iPhone 等在内的个人数字助手的流行，本领域正在成为支配地位的领域。本领域也包含汽车、飞机和空间探索中用到的专门的计算系统。
- **普适计算** 这个正在兴起的领域结合了高性能计算和嵌入式计算的元素。机场部署摄像头网络来进行视频监控就是此领域的一个例子。

我们采用负载一词来表示特定领域中典型应用的特点。可以把负载宽泛地分为两大类，即 I/O 受限的和计算受限的。但是，读者应当注意这个宽泛的分类并不总是有用的，尤其是在 I/O 的情形时。这是因为不同领域的应用中的 I/O 的本质差别很大。例如，你可以说商务应用和桌面/膝上计算机都是 I/O 密集的。商务应用要操作大型的数据库，因此会涉及大量的 I/O；桌面计算也是 I/O 受限的，但是和商务计算的不同之处在于它是交互式的 I/O，而不是与大容量存储设备（比如磁盘）之间的交互。服务器、高性能计算和网格领域的应用都趋向于是计算密集型的。嵌入式和普适计算与前述的领域很不一样。这一类的领域部署需要快速反应的传感器和执行元件（比如摄像头、麦克风、温度传感器和警报器），类似于交互式的负载；同时，对传感器数据的分析（例如，摄像头图像）又倾向于计算密集。

表 6-5 总结了这些不同领域的特征。

表 6-5 不同的应用领域

领域	环境	负载特征	调度器类型
桌面	分时、交互、多道编程	I/O 受限	中期、短期、分发器
服务器	分时、多道编程	计算受限	中期、短期、分发器
商务	分时、多道编程	I/O 受限	中期、短期、分发器
高性能	分时、多道编程	计算受限	中期、短期、分发器
网格	批处理的、分时、多道编程	计算受限	长期、中期、短期、分发器
嵌入式	分时、交互、多道编程	I/O 受限	中期、短期、分发器
普适	分时、交互、多道编程	两种受限的都有	中期、短期、分发器

我们可以采用三种不同的手段来评价调度器：建模、模拟和实现。这三种手段各有优劣。

建模是指推导该系统的数学模型（采用诸如排队论一类的技术）。模拟是指开发一个计算机程序以模拟该系统的行为。最后，实现是指把该算法实际部署到操作系统中去。建模、模拟和实现，以这个顺序代表了递增的投入。对应的是，这些手段获得的回报（在理解该系统的性能潜力这方面）也与投入相当。同时，建模和模拟使得我们可以问“如果……会怎样？”的问题，而实现则只能锁定固定的算法，使得试验不同的设计选项变得困难。一般来说，系统性能的早期评估用的是建模和模拟，然后才是投入实际实现。

6.11 调度对处理器体系结构的影响

迄今为止，我们都是把处理器当作一个黑盒子。在讨论了各种处理器调度算法之后，自然会考虑操作系统的这个功能是否需要处理器体系结构的特殊支持。当一个进程被调度在处理器上执行以后，ISA 就得符合在运行的程序的需求。我们在第 2 章就已经讨论过这部分是怎么处理的了。进程调度算法本身也类似于用户级的程序，因此在 ISA 这方面来说它们的需求也没什么特别之处。因此，是在迁移过程中——即从一个进程上下文切换到另一个进程的过程中——处理器体系结构可以为操作系统提供一些特别的支持。

266
|
267

让我们把它分开考虑，以便能理解处理器体系结构对调度运行支持的机会在哪里。首先，因为现代操作系统支持抢占，处理器体系结构必须得提供一种中断当前执行的程序的方法，以使得操作系统可以获得处理器的控制权，并运行调度算法而不是用户程序。我们已经在第 4 章中了解了这种处理器功能。具体来说，处理器体系结构需要为调度器提供一个定时器装置以便它进行调度决定，并且为进程设置时间段。另外，指令集要提供特别指令来开关中断以确保中断处理器执行一组指令的原子性[⊖]。（参见第 4 章关于此主题的详细讨论。）

执行特权指令要求处理器处在特权状态中——这是为了确保普通的用户程序不被允许运行这些指令。我们在第 4 章就看到，处理器提供用户 / 内核模式的操作，正是为了这个原因。

调度算法会修改操作系统私有的数据结构（比如进程的控制块）。必须有内存空间的分隔来给操作系统开辟一块用户进程不能访问的区域，以确保操作系统的完整性，防止用户程序无意或者恶意地对操作系统数据结构的毁坏。这一点在图 6-20 中表现了出来。在随后的章节（参见第 7、8 和 9 章），我们将会更加详细地讨论内存管理和内存层次结构，它们会帮助实现用户程序与内核代码的隔离。现在这个时候，只需要说提供内核模式的操作是个实现用户程序与内核代码之间的分隔的方便机制就足够了。

最后，让我们考虑上下文切换——也就是把当前运行进程的寄存器值保存到进程控制块——并且用下一个分发到处理器上的进程的控制块中的寄存器值填充寄存器的过程。我们知道，这一步必须得快，操作系统才能高效。这里是处理器指令集提供帮助的机会。某些体系结构（如 DEC VAX）提供单一指令负责从内存某处加载所有寄存器值，以及类似地，把所有寄存器值存到内存。虽然在过程调用时保存和恢复哪些寄存器的值可以是有选择性的（参见 2.8 节），在上下文切换的时候操作系统必须假设当前运行进程的所有寄存器都与该过程有关，从而为在指令集中包含这条指令提供了依据。某些体系结构（如 Sun SPARC）则提供寄存器窗口（参见 2.8 节），可以用来维持不同进程的上下文不同，并且消除在上下文切换时保存 / 恢复所有寄存器的需要。在这种情况中，调度器在上下文切换时只需要切换到与新进程相联系的寄存器窗口就行了。

⊖ 不管是这里还是其他地方，我们采用原子性一词都是用来表示一个操作是不可分的。一组不会被中断的指令序列就是一个原子性动作的例子。我们会在第 12 章讨论线程同步的时候回顾这个概念。

小结和展望

表 6-6 总结了本章讨论的不同调度算法的特征。对于感兴趣的读者来说，有几个调度方面的高级主题。本章讨论的策略不保证任何特定的服务质量。有几种环境可能会需要这种保证。比如说，火箭发射器、飞机上的控制系统，或者是核反应堆的控制系统。这种系统，俗称实时系统，需要确定性的保证；在调度方面的高级话题就包括怎么提供这种实时性的保证。比如说，截止时间调度器给调度器提供关于任务一定要完成的截止时间的提示。调度器将利用这些截止时间作为一种调度时用来决定进程优先级的手段。 268

表 6-6 调度算法的比较

名称	属性	调度标准	长处	短处
先到先服务	本身是非抢占式的； 可以在 I/O 完成事件时改成抢占式	到达时间（内在属性）	公平；无饥饿	响应时间方差高；护送效应
最短作业优先	本身是非抢占式的； 可以在新任务到达或者 I/O 完成事件时改成抢占式	期望运行时间（内在属性）	优先选择最短作业被证明可响应来说是最理想的响应时间的方差低	可能会饥饿； 对长时间运行的计算不利
优先级	可以是非抢占或者抢占式的	作业的优先级（外来属性）	高度灵活；由于优先级不是内在属性，可以根据调度环境的需求而调节作业的优先级	可能会饥饿
最短剩余时间优先	类似于最短作业优先但是是抢占式的	作业的期望剩余运行时间	与最短作业优先类似	与最短作业优先类似
轮转	抢占式 给每个作业以处理器的相同份额	时间段	所有作业机会均等	作业间上下文切换的额外开销

其实，很多现代应用程序也都需要实时性的保证。在 iPOD 上播放音乐、在 XBOX 上玩游戏，或者在笔记本上看电影，这些应用程序都需要实时性的保证。但是，错过截止时间的影响应该比核反应堆里面小得多。这种应用经常被称作软实时应用，已经成为通用计算机上运行的应用程序的一部分。为这类应用程序进行调度也是一个可以探索的有趣话题。 269

嵌入式计算正在成为一种占主导地位的环境，取得了不少进展，比如手机、iPOD 和 iPhone 等。感兴趣的读者可以探索此类环境中的调度问题。

超越单处理器的环境，在多处理器环境上调度带来了自己的一组挑战。我们把关于此话题的讨论推迟到本书后面的章节。最后，在分布式系统中调度是另一个激动人心的话题，但是超出了本书讨论的范畴。

Linux 调度器——一个案例研究

我们已经提到过，现代通用操作系统采用本章中所介绍技术的一个组合。作为具体的例子，让我们来看看 Linux 中是怎么调度的。

Linux 是一个开源的操作系统项目，这意味着一个开发者社区志愿地向该操作系统的开发贡献了代码。该操作系统的新版本的推出有一定的规律性。比如说，在 2007 年 12 月左右，

内核编号是2.6.23.12。本节中的讨论适用于在Linux的2.6.x版本中采用的调度框架。

Linux给我们提供了一个有趣的案例研究机会，因为它同时在尝试满足两种不同的环境：(a)桌面计算以及(b)服务器。桌面计算意味着有一个交互式环境，其中响应时间很重要。这意味着调度器可能必须得经常进行上下文切换以满足交互性要求（鼠标点击，键盘输入，等等）。另一方面，服务器则处理计算密集的负载，因此上下文切换越少，服务器能完成的作业就越多。Linux在它的调度算法里试图满足以下目标：

- 高效率：意味着花在调度器自身的时间要尽可能少，这是服务器环境的一个重要目标。
- 支持交互性：这对于桌面环境的交互式负载来说很重要。
- 避免饥饿：以确保计算性的负载不会因为存在交互性负载而受到影响。
- 支持软实时调度，以满足带有实时性要求的交互式应用程序的需求。

Linux对进程和线程这两个术语的使用与它们传统的含义相比稍微有点不标准。因此，我
270 们将采用任务一词来称呼Linux的调度算法。

Linux的调度器支持三类任务：

- 实时先到先服务
- 实时轮转
- 分时的

调度器有140个优先级别。它把0～99保留给实时任务，剩余的留给分时任务。数字越低，表示优先级越高；因此，优先级0是系统中的最高优先级。调度器采用实时先到先服务和实时轮转算法来处理交互式负载，而用分时算法来处理计算性负载。实时先到先服务任务享有最高的优先级。调度器不会抢占正在执行的实时先到先服务任务，除非有个更高优先级的实时先到先服务任务进入了就绪队列。实时轮转任务与实时先到先服务任务相比优先级低一些。如同名字所暗示的一样，该调度器给每个轮转任务分配一个时间片。相同优先级的轮转任务的时间片也相同，而轮转任务的优先级越高，相应的时间片也越长。分时任务与实时轮转任务相似，只是它们的优先级更低一些。

调度器的主要数据结构（参见图6-23）是一个运行队列。运行队列包含两个优先级数组。一个是活动数组，一个是期满数组。每个优先级数组都有140项，对应于140个优先级别。每一项都指向该级别的第一个任务。同级别的任务则以一个双向链表的形式连在一起。

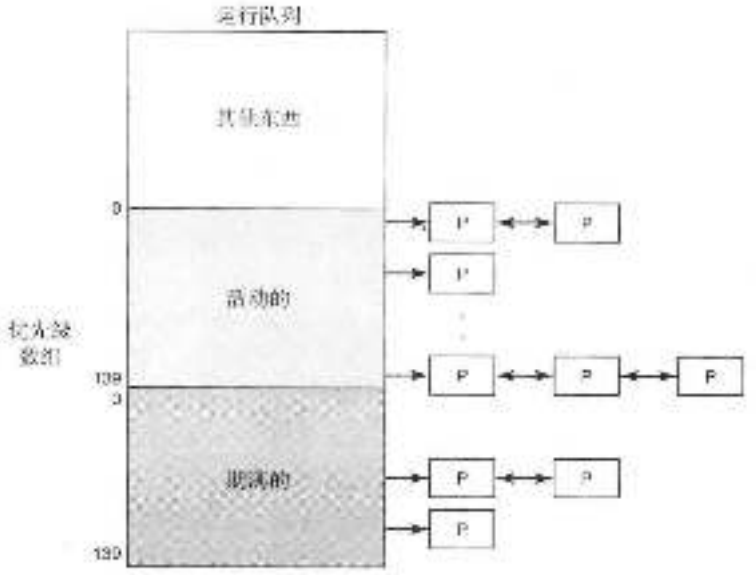
调度算法很直观：

- 从活动数组中选择最高优先级任务中的第一个来运行。
- 如果该任务阻塞了（由于I/O），那么把它放到一边，运行下一个。
- 如果当前调度的任务的时间片用完了（对先到先服务的任务不适用），把它放进期满数组中。
- 如果一个任务完成了I/O，把它放到活动数组的相应优先级的项目中。调整它的剩余时间段。
- 如果活动数组中不再有任务了，就交换活动和期满数组的指针，继续进行调度算法（也就是说，期满数组变成了新的活动数组，反之亦然）。

前述算法中第一件应当注意的事情是，优先数组保证了调度器能在常数时间内做出调度决定，而不依赖于系统中的进程数。这达到了我们之前提到的效率目标。由于这个原因，该调度器也被称作O(1)调度器，意味着调度决定不依赖于系统中的任务数。

第二件需要注意的事情是，调度器通过实时先到先服务和实时轮转调度来给交互式任务

以特殊处理，以满足软实时性的要求。


运行队列
其他类型
0
活动的
优先级数组
139
0
期满的
P
P
P

操作系统的进化与处理器的进化密不可分。Charles Babbage（查尔斯·巴贝奇）（1792—1871）建造了第一台计算机（他把它称作“分析机”）。而只用到了机械部件：齿轮、滑轮和轮子。直到第二次世界大战为止计算机都没有多少其他进步。虽然让人伤心，战争刺激技术创新却是真的。宾夕法尼亚州立大学的 John Mauchly 和 Prosper Eckert 在 1944 年用真空管建造了 ENIAC（Electronic Numerical Integrator and Calculator，电子数字积分计算机）。他们的工作是由美国陆军赞助的，用以进行破译德军密码的计算。ENIAC 和其他该时代（1944 ~ 1955）的早期机器主要都用独占模式，而不需要任何操作系统。实际上，这些早期计算的“程序”不过是被设计成用来反复进行特定计算的接线电路。

在 20 世纪 50 年代末出现的采用固态电子器件的大型机产生了我们今日所知的计算机器的最早形象。IBM，大型机竞技场的最早玩家，推出了 FORTRAN 编程语言，以及可能是第一个用来支持它的操作系统 FMS（Fortran Monitoring System，FORTRAN 监控系统）。IBM 后来推出了 IBSYS，IBM 的 IBM 7094 计算机的操作系统。这是批处理操作系统的年代：用户以一叠打孔卡片的形式提交他们的作业。打孔卡片包含一种采用任务控制语言（JCL，Job Control Language）所描述的程序所需资源，以及用 FORTRAN 语言写的程序本身。作业在计算机上一个接着一个运行，并且需要不少人类操作员的手工操作来加载程序所需的磁带和磁盘一类的资源。用户在一段时间以后再从计算机上得到程序的运行结果。

计算机的早期岁月中有两个明显不同的用户社区：科学的和商用的。在 20 世纪 60 年代中期，IBM 推出了 360 系列计算机，用以把两个社区的需求合二为一。它也推出了为该系列计算机所设计的 OS/360 操作系统。该操作系统的最重要的创新是多道程序设计，它确保了系统在为某个用户进行 I/O 的时候，CPU 可以处理别的进程。尽管有多道程序设计，但是从外
273 部看起来仍然是个面向批处理的系统，因为用户在某个时刻提交任务，而之后再来收集结果。

为了分析结果而改进程序以及为了调试，产生了对得到提交的作业的交互式响应的渴望，而这渴望就导致了操作系统中的下一场革命，即分时。它源自 MIT 在 1962 年开发的 CTSS（Compatible TimeSharing System，兼容分时系统），运行在 IBM 7094 之上。MIT 的 CTSS 的一个后续项目是 MULTICS（MULTiplexed Information and Computer Service，多路信息和计算机服务），考虑到那个时候可用的计算设施的简陋程度，它在信息服务和交换的概念这方面可能远远地领先于时代。MULTICS 引入了几个创新性的操作系统概念，涉及信息组织、共享、保护、安全和隐私，影响持续到今日。

MULTICS 是 1974 年由 Dennis Ritchie 和 Ken Thompson 于贝尔实验室开发的 UNIX 操作系统开发的种子。他们把该系统命名为 UNIX，以体现他们希望开发一个精简的单用户版本的 MULTICS 的意图。UNIX[⊖]立刻在教育机构、政府实验室和诸如 DEC、HP 等公司中流行起来。从 UNIX 间接发展出 Linux 这件事情也是个有趣的故事。随着 UNIX 的流行以及它被不同实体的广泛改编，很快出现了该操作系统的几个不兼容的版本。在 1987 年，Vrije 大学的 Andrew Tannenbaum[⊜]开发了 MINIX，作为 UNIX 的一个小克隆，以用于教学目的。Linus Torvalds 从 MINIX 出发开始编写 Linux，作为 UNIX 的一个免费产品版本，很快由于 GNU 基金会提倡的开放软件模式，而开始了它自己的生涯。虽然有这些不同的基于 UNIX 的系统的故事，有一件东西还是一样的，也就是这些不同风格的 UNIX 背后的操作系统概念。

⊖ 它一开始被命名为 UNICS（Uniplexed Information and Computer Service，单路信息和计算机服务），后来改名为 UNIX。

⊜ 是多本著名操作系统和网络系统教材的作者。

前面提到的所有操作系统都是为了支持大型机和小型机而在不断演进。与这种进化平行的是微型计算机进入市场，逐渐地改变了计算设施和使用模型。计算机不再一定是像大型机和小型机通常意味的那样是个共享资源。相反，它是个人计算机（PC），用来给一个用户独占使用。在20世纪70年代，开始有早期的微型计算机，它们采用Intel 8080/8085单芯片处理器。这样的个人计算机应当有什么样的操作系统呢？一种简单的操作系统，被称作CP/M（Control Program Monitor，控制程序监控器）是这些微机的行业标准。

IBM在20世纪80年代早期开发出了IBM PC，一个叫做微软的小公司为IBM公司提供了一个操作系统，称作MS-DOS。MS-DOS的早期版本和CP/M有着惊人的相似处，但是有着简化的文件系统，并可提供更高的性能。有了IBM这样的工业巨人的庇佑，MS-DOS很快占领了PC市场，把CP/M赶到了二线。一开始，由于PC的预期用途，MS-DOS的功能相当原始。但是它缓慢地开始加入来自UNIX系统的多任务和分时的想法。值得一提的是，虽然 274
MS-DOS是从一个小的轻型操作系统开始演化，逐渐加入来自UNIX操作系统的想法，而苹果的Mac OS X同样针对PC（苹果的Macintosh系列），则是UNIX操作系统的直接后裔。

苹果在Mac计算机中采用了图形用户界面（GUI，Graphical User Interface）以后，微软紧随其后，在MS-DOS之上提供Windows，作为用户与PC交互的方式。早期的Windows是MS-DOS之上的一层包装，直到1995年微软推出Windows 95，之后推出Windows NT（NT表示New Technology，即新技术）、Windows NT 4.0、Windows 2000、Windows XP、Windows Vista，再到Windows 7（截止到2009年）。但是，操作系统中的核心抽象层面的基本概念在版本的更迭中变化不大。如果说有变化的话，那么就是这些概念成熟到了与UNIX操作系统中的那些难以区分的程度。类似地，基于UNIX的系统也吸收和加入了源自PC世界的图形用户界面的想法。

练习题

1. 对比进程和程序的异同。
2. 进程的状态包括哪些东西？
3. 在分时环境中哪个指标最为用户中心？
4. 考虑一个抢占式的优先级处理器调度器。要执行的作业包含有三个进程，P1、P2和P3。它们具有以下特点：

进程	到达时间	优先级	活动
P1	0秒	1	8秒的CPU突发，然后 4秒的I/O突发，然后 6秒的CPU突发后退出
P2	2秒	3	64秒的CPU突发后退出
P3	4秒	2	2秒的CPU突发，然后 2秒的I/O突发，然后 2秒的CPU突发，然后 2秒的I/O突发，然后 2秒的CPU突发，然后 2秒的I/O突发，然后 2秒的CPU突发后退出

P1、P2和P3的平均周转时间分别是多少？

这组作业的平均等待时间是多少？

[275] 5. 先到先服务的 CPU 调度策略有何缺点？

6. 解释先到先服务调度策略中发生的护送效应。

7. 先到先服务调度的好处是什么？

8. 依据以下标准讨论不同的调度算法：

(a) 等待时间

(b) 饥饿

(c) 周转时间

(d) 周转时间的方差

哪个调度算法以周转时间的方差大而著称？

9. 是不是任何一个调度算法都能改成抢占式的？什么样的算法特性会使它适合抢占式的调度？需要处理器体系结构为允许改内进行怎样的支持？

10. 总结处理器调度为处理器体系结构带来的增强。

11. 考虑以下进程，按照顺序到达：

	CPU 突发时间	IO 突发时间
P1	3	2
P2	4	3
P3	8	4

分别采用先到先服务、最短作业优先和轮转算法，说明处理器和 I/O 的活动情况。

12. 用最短作业优先和轮转（时间片 =2）重做例 6-1。

13. 用先到先服务和轮转（时间片 =2）重做例 6-3。

参考文献注释和扩展阅读

Bobrow 等人所写的分时操作系统的经典论文 [Bobrow，1972] 对早期商用操作系统如 DEC 公司的 TOPS-20 的设计有着重大影响。那些年里，其他有影响力的操作系统包括 DEC 的 VAX/VMX 操作系统，以及 IBM 的 OS/360。关于 Unix 的原始论文发表在 1974 年 [Ritchie，1974]。处理器调度的话题有几本教材讲得很好 [Silberschatz，2008；Tanenbaum，2007]。要了解 Linux 中 CPU 调度的细节，Bovet 和 Cesati 写的书是个很好的资源 [Bovet，2005]。很多操作系统教材 [Silberschatz，2008；Tanenbaum，2007] 包括了对几个曾经有影响力的操作系统的案例研究，也包括诸如 Linux、Windows XP 和 Symbian
[276] OS 之类的现代系统。Foster 和 Kesselman 写的书 [Foster，2003] 是个学习网格计算中的开发的很好资源。

内存管理技术

让我们回顾一下已经了解了哪些东西。在硬件方面，我们看到了处理器指令集、中断，以及处理器的设计。而在软件方面，我们看到了如何把处理器当作一种资源，调度它来运行不同的程序。我们已经熟悉的软件实体包括位于操作系统之上的编译器和链接器，以及位于操作系统之中的加载器和进程调度器。

现在，我们希望我们已经对“盒子”中的魔法般的某些东西进行了揭秘。在本章中，我们把视角放在计算机系统的另一个重要组成部分，即内存，以继续我们揭秘这个“盒子”的过程。

内存系统中的硬件和系统软件之间的相互联系比其他任何子系统都要强。在描述内存系统的某个软件方面的时候，经常没法不同时提一提相应的硬件支持。我们用包括本章在内的3章来讲内存系统。本章主要讲操作系统管理内存的不同策略，以及必需的体系结构支持。在第8章中，我们深入讨论基于页机制的内存系统的细节，尤其是页替换策略。最后在第9章，我们讨论内存层次结构，尤其是缓存以及主存，即物理内存。

7.1 内存管理器提供的功能

让我们来了解一下什么是内存管理。为了加以区分，我们需要指出本书中所说的内存管理和Java、C#等编程语言中的自动内存管理是不同的。这些编程语言的运行时系统会自动释放程序未使用的内存。垃圾回收是这个功能的另一个称呼。有一些垃圾回收器里的技术与操作系统内存管理里的方法类似。关于这两者的异同点的讨论超出了本书的范畴。如果读者对垃圾回收有兴趣，可以参考其他资料学习[Jones，1996]。

在本书中，我们将重点放在操作系统如何管理内存上。如处理器一样，内存也是一种珍贵的资源，而操作系统则负责确保它的有效利用。内存管理是操作系统里的一个部分，它提供以下功能。

1）**提高资源利用率** 因为内存是一种稀缺资源，它最好能被到按需分配。我们可以把它类比成办公室的空间。系里的每个教师都可以给实验室和他的学生申请若干空间。系主任可能没有办法一次性就把教师申请的空间分配给他，但是可以随着教师学生数的增加而逐渐给他增加空间。类似地，尽管程序的内存印迹[⊖]包括了堆，却没有必要在程序启动的时候就把堆空间分配给它，而只需要在程序动态地申请的时候给它分配就行了。如果一个教师的小组缩小了，那么他不再需要分配到的所有空间，系主任就会收回一部分空间，分给其他更需要空间的教师。类似地，如果一个进程没有主动使用分配给它的空间，也许最好就应该把它的空间释放掉，给其他需要空间的进程使用。渐进式分配内存和动态分配两个点子都可以改进内存的利用率。我们用一个简单的图示来表示合理分配内存资源的重要性。下图左侧的部分是一个笔记本电脑上的任务管理器的截图，显示了它实际在运行的应用程序列表。而右侧部分

⊖ 在第5章和第6章中，我们把术语内存印迹来表示程序在载入时所占用的静态空间。在第7章和第8章，我们则把内存印迹的定义扩展为包括程序执行时在堆里动态分配的空间。

显示了该笔记本电脑上实际运行的进程列表的很小一部分。一共有 4 个应用程序在运行，但是却有 123 个进程！我们展示这张图是希望指出，在用户运行的进程以外，操作系统和其他实用程序会产生很多后台进程（常被称作守护进程）。因此，在任何时候，所有进程对内存系统产生的需求都是很大的。例子中的笔记本电脑有 2GB 内存，其中 58% 在使用中，尽管只
278 有 4 个应用程序在运行。



2）**独立性和保护** 我们提到过，内存里任何时候都同时驻留着多个进程。某程序可能会包含 bug，以至于失控地写入到不属于它的内存印迹的内存区域去。防止有 bug 的进程破到它自己和其他进程是明智的。进一步说，在现今计算机病毒和蠕虫肆虐的时代，恶意程序可能有意尝试破坏其他正常程序的内存。因此，内存管理器为进程提供独立性，并且在它们之间提供保护。我们再考虑一下空间的类比。如果你在有兄弟姐妹的家庭中长大，很可能就能联想到对独立性和保护的需要。你要么已经有，要么是想有一个你自己的房间，在你不想让兄弟姐妹（以及父母）进来的时候可以锁起门来！

3）**从资源限制中解放** 当一个教师被雇用了，系主任可能会许诺他想要多少空间就有多少空间，说不定比整个系现有的未占用空间还要多。这就给了新老师计划建立和扩展研究计划与学生团队的自由。类似地，对程序员来说理想情况是开发程序的时候不用担心物理内存的总量。想象一下一个有 10 个学生的教师，但是实验室只能放下 5 个人。学生可能只能换到工作，共享实验室。类似地，想象一下你要写一个多玩家的视频游戏，而经理说你总共只能使用 10KB 的内存。比如说你的程序的代码和数据结构所需要的内存空间远超过给定的 10KB。那么你作为程序员就必须找出不会同时用到的代码和数据结构，并以一种分时共享的方式使用内存装下它们（本质上来说，就是把这些数据结构分多层放在内存里）。如果被迫这么做的话，程序就会变得很丑。为了解决这个问题，内存管理软件和体系结构协同工作，设计出一种机制，给程序员以拥有很大内存的假象。实际物理内存可能比程序员看到的假想中
279 的内存小得多。

4）**并发进程共享内存** 你可能大部分时候都想把兄弟姐妹关在门外，但是偶尔你也会希望他们进入你房间，也许是来玩视频游戏。类似地，尽管进程间的内存保护是必要的，进程有时候也许会想共享内存。既可以是隐式的也可能是显式的。比如说，你可能在桌面上打开了多个浏览器窗口，每个可能在访问不同的网页，但是如果它们都在运行同一个浏览器应用程序，它们就在共享代码。这是进程间隐式共享的例子。把一张图片从一个幻灯片程序拷贝到一个字处理程序则是显式数据共享的例子。内存管理帮助进程在需要的时候进行内存共享。

前面提到的所有功能都有各自的成本，特别是，我们在引入 CPU 和内存之间的一个代理，如图 7-1 所示。该代理是一块硬件，提供实现内存管理策略所需的机制。原则上说，它

把CPU生成的（逻辑）内存地址映射到实际内存地址以便访存。代理的复杂程度取决于内存管理器的功能。

迄今为止，我们还没有在LC-2200中引入这样的功能。但是，既然我们有野心，想让LC-2200在从游戏机到高性能计算机的各种设备中称霸全世界，这种功能显然必不可少。

一个好的内存管理器有三主目标。它应当：

1）需要尽量少的硬件支持；

2）对内存访问的影响尽量小；

3）让内存管理的额外开销小（指的是内存的申请和释放）。

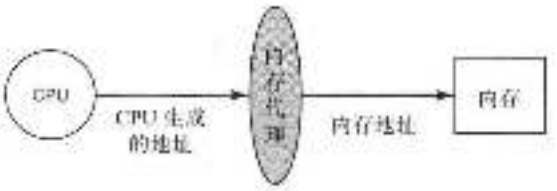


图7-1 内存管理器的通用图示。内存管理器的行为像是CPU生成的地址和内存之间的代理

7.2 内存管理的简单方案

在本节中，让我们考虑一些内存管理的简单方案，以及所需的对应的硬件支持。本节本着共同探索的精神，讨论如何达到之前提到的目标，同时也实现上节提到的功能。前两个方案和对应的硬件支持（栅栏寄存器和界限寄存器）只是为了说明目的而提出，我们不知道有任何机器体系结构用到了这样的方案。而第三个方案（基址寄存器和限长寄存器）则在多种体系结构中被广泛采用，包括CDC 6600（第一台超级计算机）以及IBM 360系列。这三个方案在达到前一节所提出的功能需求方面都有欠缺之处，因此催生了现代体系结构中为更加复杂的内存管理机制，即分页和分段。 280

1）**用户和内核的分隔** 考虑一个非常简单的内存管理方案。我们之前已经看到，操作系统和用户程序共享所有可用的内存空间。操作系统所用的空间是内核空间，而用户程序所用的则是用户空间。作为一种近似，我们希望确保两种空间之间有一个分界线，从而用户程序不会跨进操作系统的内存空间里去。图7-2展示了一个用以实现这种分隔的简单机制。图中的阴影部分对应着图7-1中内存代理所做的工作。该方案用到了三个体系结构元素：一个模式位用来表示程序是在用户还是内核模式，一条特权指令用来翻转该位，以及一个栅栏寄存器。你可以想象，该寄存器的名字来源于对保护财产的物理意义上的栅栏的类比。内存管理器在调度用户程序时设置栅栏寄存器。硬件通过把访存地址与栅栏寄存器做对比来验证处理器产生的内存地址的有效性。这个简单的硬件机制为用户程序和内核之间提供了内存保护。

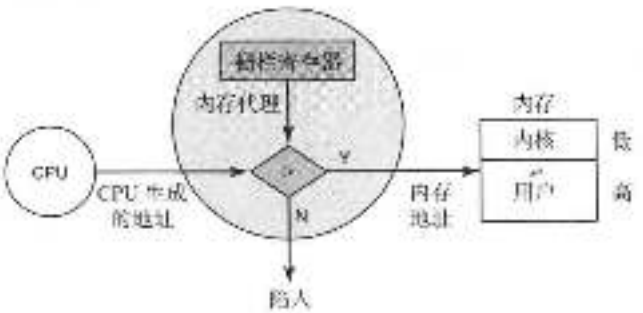


图7-2 栅栏寄存器，使得用户程序只能访问高于栅栏寄存器所指定的内存地址 281

举例来说，假设栅栏寄存器被设置为10000。这就意味着内核占据着0 – 10000的内存区

域。如果CPU在用户模式中产生的地址高于10000，硬件就认为它是合法的用户程序地址。任何小于或等于10000的地址都是内核地址，从而会产生访问违例的陷入。CPU必须在内核模式里才能访问内核内存区域。为了理解CPU是怎么进入内核模式的，回忆一下第4章中介绍的陷入的概念，一个同步的程序不连续性，通常是由希望进行系统调用（比如读取文件）的程序引发的。这样的陷入会导致处理器自动进入内核模块，这是陷入指令实现的一部分。因此，CPU隐式地在系统调用里进入内核模式，并且此时就可以访问为内核所保留的内存区域了。当操作系统完成系统调用，它可以通过体系结构提供的特权指令显式地返回用户模式。这条指令是特权指令是因为它仅限于在内核模式里使用。任何在用户模式中执行该指令的尝试都会引发异常，另一种同步程序不连续性（我们也在第4章介绍过了），由用户程序的非法操作引起。你也许已经猜到了，向控制寄存器进行写入也是一条特权指令。

2）**静态重定位** 之前已经讨论过，我们希望多个用户程序可以同时在内存中共存。因此，内存管理器应当保护共存的进程，防止它们互相影响。图7-3展示了实现该保护的硬件机制。图中阴影部分仍然是表示图7-1中的内存代理所完成的工作的硬件部分。

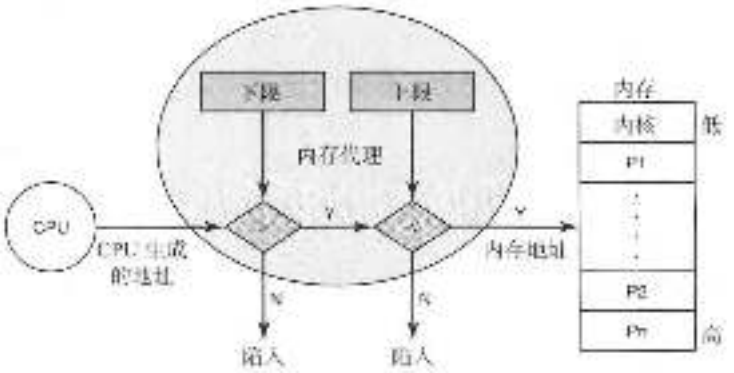


图7-3 界限寄存器：使得用户程序只能访问在下限和上限寄存器的值之间的内存地址

静态重定位指的是一个进程的内存界限在链接并创建可执行程序的时候被设定。当可执行文件创建好之后，在程序执行过程中内存地址就不能改变了。为了支持进程的内存保护，体系结构提供两个寄存器：上限和下限。界限寄存器是进程控制块（也就是PCB，在第6章介绍过）的一部分。向界限寄存器中进行写入是体系结构提供的特权指令。内存管理器在分发进程的时候把PCB中的值设置进界限寄存器中。（关于如何在CPU上调度进程，参见第6章。）在链接时，链接器为特定进程分配一个特定的内存区域[⊖]。加载器在加载的时候把下限和上限寄存器的值确定下来，之后在程序执行过程中再也不更改它们。假设链接器为P1分配了地址范围10001 – 14000。在这种情况下，调度器会把下限和上限寄存器分别设置为10000和14001。从而在P1执行的时候，如果CPU生成的内存地址在10001 – 14000之间的话硬件就会允许该访问，而如果越出该界限的话就会产生一次访问违例陷入。

282]

我们在第6章中提及过替换的概念。把一个进程替换走是指将一个非活动进程（比如说，在等待I/O完成的进程）从内存里移动到磁盘上。内存管理器这么做是想把原本被非活动进程占据的内存空间分配给其他活动进程来有效利用。类似地，在一个进程再次成为活动进程时

⊖ 注意现代操作系统采用的是动态链接，其中给进程选择地址的决定会推迟到载入进程的时候来进行。动态链接器在选择加进程的界限寄存器的时候便可以考虑当前内存的使用状况。

（比如说，它的 I/O 请求完成了），内存管理器会把该进程从磁盘替换入内存（在确保所需内存分配给将要替换入内存的进程之后）。

考虑在有静态重定位支持时把一个进程从磁盘替换入内存。由于界限是确定的，内存管理器会把替换出的进程放进内存里和原来完全一样的位置中。如果该内存空间已经被其他进程使用了，那么被替换出的进程此时将无法被加载到内存，这就是静态重定位的主要缺点。

现实中，编译器生成代码时会假设程序会驻留在内存的某个已知的位置[⊖]。因此，如果操作系统不能对此假设做出更正的话，进程本质上来说就是不可重定位的。如果一个进程的地址在加载内存和执行时均不能改变，我们把它称作不可重定位的。我们用静态重命名这一名称来称呼在进程加载时定位到与编译时不同的位置上的技术。也就是说，程序中用到的地址在程序载入内存时就确定下来（即固定），在执行过程中不再变更。IBM 在它们 20 世纪 283
60 年代早期的大型机中采用了这种静态重定位的一个版本。在把进程加载进内存时，加载器会在内存中检查哪些地方未被使用，并决定把进程放在哪里。然后它会“修复”可执行程序中的所有地址，以使得程序可以在它的新家中正常工作。例如，如果原始程序占据了地址 0 – 1000，加载器决定把程序放在地址 15000 和 16000 之间。这种情况下，加载器就会在加载程序时向程序中的每处地址上加上 15000。可以想象，这是个非常笨重费力的工作。加载器知道可执行程序的布局，因此它知道常数值和地址之间的不同之处，因此可以进行这种修补工作。

3）**动态重定位** 静态重定位对内存管理有着太多约束，并且导致内存利用率低下。这是因为在创建之后，可执行程序就在内存中占据着一个固定区域。两个完全不同的程序如果碰巧有着相同的或者重叠的内存界限，就不能在内存中共存，哪怕内存中当前还有其他未被占用的区域。这就好比两个小孩争抢同一个玩具，尽管其实有很多其他玩具可以玩一样！这不是我们想要的情况。动态重定位就是指能够把可执行程序加载到内存中任意能装得下该进程的区域。让我们来看看这与静态重定位有什么不同。有了动态重定位，程序生成的内存地址可以在程序执行的过程中被改变。这就意味着在把程序加载到内存的时候，操作系统可以根据当前内存使用情况来决定把程序放在哪里。根据之前关于静态重定位的讨论，你可能会认为这就是动态链接器让我们做的事情。但是，这里的区别是有了动态重定位，如果进程被替换出内存了，当它之后被换回内存时它不一定必须回到之前它待过的位置。静态重定位时程序中产生的地址在执行中是固定的，而用了动态重定位以后可以在执行过程中改变它们。

现在需要找出动态重定位所需的体系结构支持。让我们尝试一种略微不同的硬件机制，如图 7-4 所示。和之前一样，图中阴影部分表示图 7-1 中内存代理所完成的工作的硬件部分。体系结构提供两个寄存器：基址寄存器和限长寄存器。CPU 产生的地址总是被加上基址寄存器的值。由于这个偏移操作合理地发生在程序执行时，这种体系结构的增强就达到了动态重定位的需求。和在静态重定位中一样，这两个寄存器是每个进程的 PCB 的一部分。每次一个进程被加载到内存（既可能是加载程序也可能是替换入程序），加载器都为该进程给基址寄存器和限长寄存器分配好值。内存管理器则将这些加载器分配的数值写入 PCB 中该进程的相应字段中。类似于静态重定位中的界限寄存器，向基址寄存器和限长寄存器写入是体系结构提供的一个特权指令。当内存管理器分发一个特定进程时，它将基址和限长寄存器的值设为该进程的 PCB 中的值。假设 P1 的内存印迹是 4000。如果加载器给 P1 分配的内存起始 284

⊖ 在大部分现代编译器中，程序从地址 0 开始，直到某个系统预定的最大地址为止。

是 10001 ~ 14000，那么内存管理器就会在 P1 的 PCB 中向基址寄存器赋值 10001，而向限长寄存器赋值 14001。因此，在 P1 执行时，任何 CPU 产生的地址都会自动被硬件向上偏移 10000。只要偏移之后的结果小于界限寄存器里的值，硬件就认为该次访问合法，允许该次访问。任何超出限长寄存器的值都会导致访问违例陷入。读者应当能够确信动态重定位会得到比静态重定位更好的内存利用率。

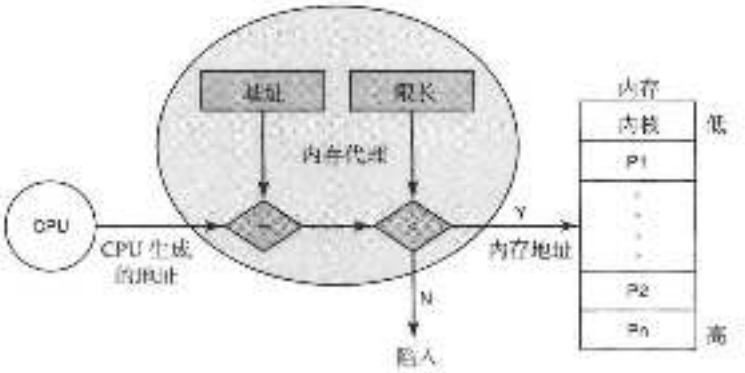


图 7-4 基址和限长寄存器。CPU 生成的地址被偏移基址寄存器所指定的数值移动，而限长寄存器则用作当前进程所能访问的内存地址上限

我们为静态和动态重定位所介绍的体系结构增强都需要在处理器里添加两个额外的寄存器。让我们回顾一下这两个机制以及对应的实现。对比图 7-3 和图 7-4 里需要的数据通路操作，图 7-3 里需要进行两次比较操作来检查界限，图 7-4 里需要进行一次加法操作，然后进行一次比较操作。因此，在两个方案里都需要进行两次算术操作来从 CPU 地址得到内存地址。因此，两个机制甚至在硬件复杂性和延迟上都差不多。但是，采用基址和限长寄存器带来的内存利用率的好处是巨大的。这就是一点点人类的才智可以帮助我们在不增加投入的情况下带来巨大收益的例证。

7.3 内存分配方案

假设有硬件支持，并且内存管理器也采用了基址－限长寄存器的方案，现在来讨论内存
285 分配的策略。在每种情况下，我们都给出进行内存管理所需要的数据结构。

7.3.1 固定尺寸分区

在该策略中，内存管理器把内存划分为固定尺寸的分区。让我们来理解内存管理器所需的数据结构。图 7-5 展示了一种可行的数据结构，即一个保存在内核空间里的分配表。效果就是，内存管理器通过该数据结构来管理用于用户程序的那部分内存。为了本分配策略，表中包含了三个字段，如图 7-5 所示。被占位表示该分区是否已经被使用。如果该分区已经被分配，则该位为 1；否则的话，则该位为 0。当一个进程请求内存时（就可能是在加载时也可能是在执行过程中），选择一个尺寸大于等于当前需求量的固定尺寸分区给它。比如说，如果内存管理器的分区尺寸有 1KB、5KB 和 8KB。某进程 P1 请求了一块 6KB 大小的内存块，那么它就会被分配到一块 8KB 大小的分区。内存管理器也就会将表中对应的位设置为 1，并且在该进程返回该内存块时把该位清零。在 P1 请求 6KB 内存之后，分配表如图 7-5a 所示。注

意在 8KB 分区中有 2KB 空间被浪费掉了。

分配表

被占位	分区大小	进程
0	8K	XXX
0	8K	XXX
0	1K	XXX

内存

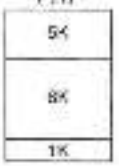


图 7-5 固定尺寸分区的分配表

不幸的是，即使另一个进程申请 2KB 内存，也不可能把浪费的空间分配给它。这是因为分配表以固定尺寸的分区为基础维护着总结信息。这个现象被称作内部碎片，指的就是固定尺寸分区内部浪费空间，而这种浪费会导致内存利用率低下。一般来说，内部碎片是内存分配的粒度与实际请求内存尺寸的差。 286

$$\text{内部碎片} = \text{固定分区大小} - \text{实际内存请求} \tag{7-1}$$

分配表

被占位	分区大小	进程
0	5K	XXX
1	8K	P1
0	1K	XXX

内存

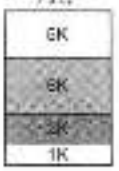


图 7-5 a) 在 P1 的请求被满足后的分配表

例 7-1 一个内存管理器以 4KB 的固定尺寸块为单位进行内存分配。可能的最大内部碎片是多大？

答：

进程所请申请的最小内存大小为 1 字节，此时内存管理器会分配一个 4KB 的分区以满足此要求。因此，最大内部碎片 = 4KB − 1

= 4096 − 1

= 4095 字节

假设在 P1 有 8KB 分区时另一个内存分配请求需要 6KB。那么这时该请求无法满足，虽然累计来说（把 5KB 和 1KB 的分区加起来）还是有 6KB 的内存空间，却无法满足该新请求，因为这两个分区不是连续的（而且进程的请求所要求的是连续的一段内存）。这个现象被称作外部碎片，这也会导致内存利用率低下。一般来说，外部碎片是所有内存系统可用的不连续内存块的总和。

$$\text{外部碎片} = \sum \text{所有不连续的内存分区} \tag{7-2}$$

7.3.2 变长分区

为了克服内部碎片的问题，我们来讨论一下分配可变长度的分区以适应内存请求的需求的内存管理器。假设内存管理器总共有 13KB 的内存可用。这一回内存管理器并不像上一个方案那样拥有一张静态的分配表，而是在运行过程中动态地建立分配表。图 7-6 展示了在进行任何分配之前分配表的初始状态。



图 7-6 变长分区的分配表

图 7-6a 则展示了内存管理器处理一系列申请内存的请求之后的分配表。注意管理器在满
287 足 P1、P2、P3 的请求之后还剩下 2KB 的剩余空间。

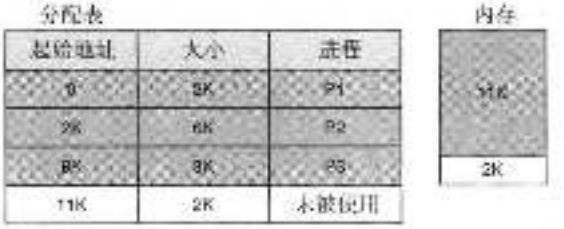


图 7-6 a) 处理了来自 P1(2KB)、P2(6KB) 和 P3(3KB) 的请求之后的变长分区的分配表

图 7-6b 则展示了在 P1 完成之后，被 P1 占据的 2KB 分区也被标记成了未被使用。



图 7-6 b) 在 P1 完成之后的变长分区的分配表

假设新进程 P4 新申请了 4KB 内存。不幸的是，该请求不能被满足，因为 P4 申请的空间需要是连续的，但是可用的空间却是碎片状的，如图 7-6b 所示。因此，变长分区虽然解决了内部碎片的问题，却不能解决外部碎片的问题。

随着过程的结束，内存中也会产生可用空间组成的洞。分配表记录着这些可用空间。如果分配表中的相邻项都是可用的，那么管理器将会把它们合并成为一大块，如图 7-6c 和图 7-6d 所示。



288 图 7-6 c) 在 P2 释放内存前的变长分区的分配表

分配表

起始地址	大小	进程
0	8K	未被使用
8K	3K	P3
11K	2K	未被使用

内存



图 7-6 d) 在 P2 释放内存后的变长分区的分配表

问题 7-2 图 7-6b 和图 7-6d 中所示情况下最大的外部碎片分别是多少？

答：

在图 7-6b 中，有两个 2KB 的分块不连续，因此

外部碎片 = 4 KB

在图 7-6d 中，有两个分别为 2KB 和 8KB 的分块不连续，因此

外部碎片 = 10 KB

在接到一个新的内存请求时，内存管理器进行内存分配有多种选择。这里列出两种可能性。

1）**最佳适应** 管理器检查一遍分配表，找出大小最适合新请求的分块。比如说，参见图 7-6d，如果请求的大小为 1KB，则分配器将会通过分割 2KB 的分区来满足请求，而不是使用 8KB 的那块空间。

2）**首次适应** 管理器找出第一个适合新请求的分块。比如说，参见图 7-6d，分配器会分割 8KB 的分区来满足 1KB 的请求。

分配算法的选择需要进行权衡。最佳适应算法在分配表很大时的时间复杂度会很高。但是最佳适应算法的内存利用率会较高，因为外部碎片较少。

7.3.3 缩并

内存管理器会在外部碎片超出可容忍极限以后采用一种叫做缩并的技术。例如，参见 289
图 7-6d，内存管理器可能会把 P3 的内存重新放到从地址 0 开始的位置，从而创建出一段连续的 10KB 空间，如图 7-6e 所示。缩并是个代价高昂的操作，因为 P3 的内存范围里的所有嵌入地址都得进行调整，才能保持语义不变。不仅昂贵，而且实际上在大部分体系结构里根本就是不可能。回忆一下 20 世纪 60 年代的早期内存管理方案，就是为了引入动态重定位，IBM 360 才引入了基址寄存器[⊖]（跟图 7-4 里的方案差不了多少）。OS/360 操作系统会在加载程序的时候进行动态重定位。但是，即使用了这个方案，当一个进程被加载到内存中以后，缩并内存还是需要做不少事情，比如说把进程暂停以便进行重定位。更进一步地说，缩并的代价随着需要重定位的进程个数上升而上升。因此，就算在体系结构里进行内存缩并是可行的，内存管理器也很少这么干。缩并通常是跟替换一起进行的——也就是说，内存管理器会在进程从硬盘换回内存的时候顺便进行重定位。

分配表

起始地址	大小	进程
0	3K	P3
3K	10K	未被使用

内存



图 7-6 e) 缩并内存以创建大的连续空间

⊖ 来源：参见 www.research.ibm.com/journal/rd/441/amdahl.pdf。另请读 Gene Amdahl 关于 IBM 360 的原始论文。

7.4 分页虚拟内存

随着内存容量持续攀升，外部碎片就变成了很严重的问题。我们需要解决它。

让我们回顾一下基础知识。在用户眼中，程序在内存中占据着连续的一段。我们迄今为止讨论的对内存管理的硬件支持至多就是把程序重定位到与用户所看到的地址不同的地方去。我们需要绕开这个用户视角里固有的关于连续内存的假设。虚拟内存的概念就有助于绕开此假设。分页则是实现这一概念的手段。

想法就是让用户保留关于程序得到的是一段连续内存的感觉，因为这样让程序编写起来容易一些。内存代理（参见图 7-1）则把这个连续的概念分割成相等大小的逻辑实体。这些逻辑实体被称作页。类似地，物理内存由页帧组成，我们将会简单地称呼为物理帧。逻辑页和物理帧的大小相同且固定，称作页大小。物理帧中承载着一个逻辑页。



a) b) c)

图 7-7 关于相框的类比

[290] 让我们考虑一个类比。教授想熟悉整个大班的所有学生。为了达成目的，他用了以下手段：他把班上学生的照片收集下来。他在办公室有个空相框（见图 7-7a）。当一个学生在他的办公时间来访，他就把该学生的照片放进相框（见图 7-7b）。当下一个学生来访，他就再在相框中换一次照片（见图 7-7c）。教授并没有给每个学生都准备一个相框，只是为不同的学生重用同一个相框。他也不需要给每个学生准备一个单独的相框，因为他在办公时间见到学生总是一个一个地见到。

把物理内存分割成很多页帧的过程与这个简单的例子有很多相似之处。相框可以装载任何照片。类似地，给定的物理帧可以用来承载任何逻辑页。内存代理维护着用户的逻辑页和物理内存的物理帧之间的映射关系。不难猜到，给每个程序创建这样的映射关系是内存管理器的职责。一个被称作页表的实体用来存放从逻辑页到物理帧的映射关系。页表的效果就是把用户所看到的内存和物理上的内存组织区分开来。因此，我们把用户看到的内存称作虚拟内存，而把逻辑页面称作虚拟页。CPU 产生对应着用户视角的虚拟地址。代理把虚拟地址通过查表（参见图 7-8）转换到物理地址。既然我们已经区分清楚了用户视角和物理组织，虚拟内存和物理内存的相对大小就无所谓了。例如，从用户的角度看到的虚拟内存比实际的物理内存大得多是完全合理的。实际上，这就是现在



图 7-8 页表、内存代理可以把用户程序产生的虚拟地址按页表得到物理地址

大部分内存系统的常态。较大的虚拟内存消除了由于物理内存受限而引起的资源限制，给用户程序以内存很大的幻象。

内存代理只需要在同一页中维持用户关于连续内存的假设即可。不同页面则不必在物理内存中处于连续的位置。图 7-9 展示了一个程序，它已经通过分页技术把四个虚拟页面映射到了两个物理帧上。注意分页技术解决了外部碎片的问题。但是内部碎片还是可能存在。由于帧大小是固定的，如果请求的内存大小只能填充帧的一部分，还是会产生内部碎片。

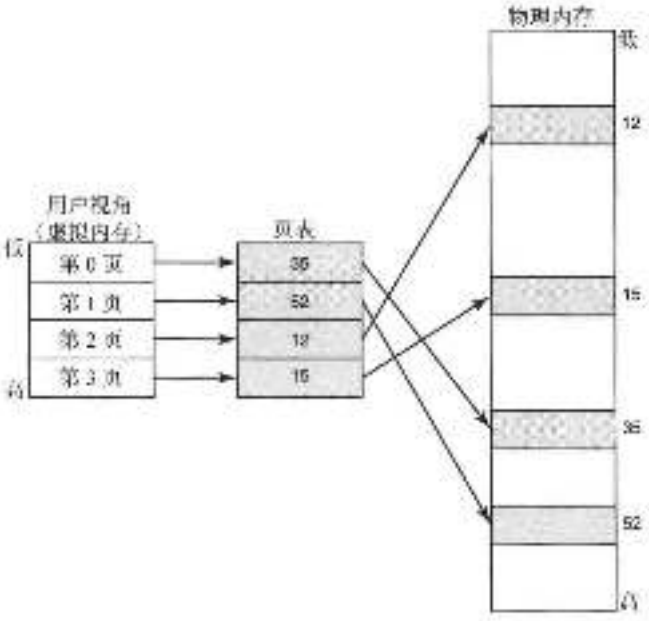


图 7-9 打破用户的连续虚拟内存的视角

291 / 292

7.4.1 页表

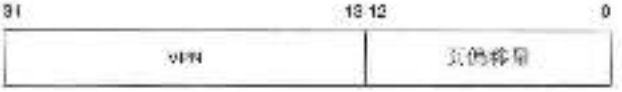
让我们深入看看分页的概念。既然虚拟页（或者物理帧）的大小固定，而同一页中的地址是连续的，我们可以把 CPU 生成的虚拟地址看成由两部分组成：虚拟页号（VPN，Virutal Page Number）和页中的偏移量。为了解释方便，我们假定页面大小是 2 的幂数次幂。硬件首先把虚拟地址分成两部分。这件事情简便易行。记住页中所有的地址都是连续的。所以，虚拟地址的偏移量肯定就来自它的低位。偏移量所需的位的个数直接由页大小就可以知道。比如说，如果页大小为 8KB，则页中有 2^{13} 个不同的字节，因此我们需要 13 个位来给每个字节进行寻址，这也就是偏移量的长度。虚拟地址的剩下高位就组成了虚拟页号。一般来说，如果页大小为 N，则虚拟地址的 $\log_2 N$ 个低位组成页偏移量。

例 7-3 考虑一个有 32 位虚拟地址的内存系统。假设页大小为 8KB。画出虚拟地址分成虚拟页号和页偏移量的布局。

答：

每个页有 8KB，我们需要 13 个位来寻址页中每个字节进行寻址。由于页中的字节是连续的，这 13 个位也就是虚拟地址的最低位（即第 0 ~ 12 位）。

虚拟地址的剩下 19 个高位（即第 13 ~ 31 位）则组成虚拟页号。



把虚拟地址转换为物理地址就是查页表来得到对应于虚拟页号的物理帧号（Physical Frame Number，PFN）。图 7-10 展示了这个转换过程。图 7-10 的阴影部分是内存代理所做工作的硬件部分。硬件查找页表以把虚拟地址转换为物理地址。那么，页表应该放在内存的什么地方呢？既然硬件必须得在每次内存访问的时候查找页表来进行地址转换，似乎把它放在 CPU 数据通路里是个好想法。让我们来检查一下这个想法的可行性。我们对于每个虚拟页号都需要一个表中的项目。在例 7-3 里，我们需要 2^{19} 个页表项目。因此，把页表实现为处理器数据通路的一部分是不可取的。而且这样的话，系统里就只能有一个页表。而为了内存保护，
293 每个进程都需要自己的页表。

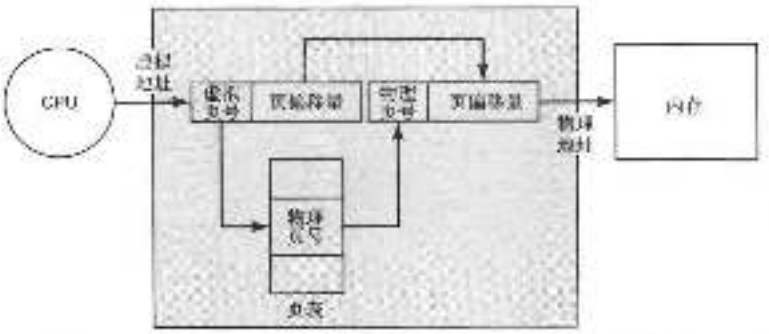


图 7-10 地址转换。页表用的是虚拟地址的虚拟页号的部分作为查找索引，用来查找物理帧号。虚拟地址的页偏移量被附加在物理帧号之后，以得到物理地址

因此，每个进程都有一个页表驻留在内存里，如图 7-11 所示。CPU 需要知道内存中的页表的位置，以便可以进行地址转换。出于这个目的，我们向 CPU 的数据通路里添加一个新的寄存器，页表基址寄存器（PTBR，Page Table Base Register），其中包含当前运行进程的页表的基地址。PTBR 是进程控制块的一部分。在
294 进行上下文切换时，寄存器值会从新分发的进程的 PCB 中加载。

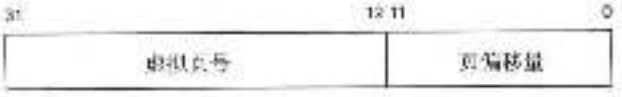


图 7-11 物理内存中的页表

例 7.4 考虑一个有 32 位虚拟地址和 24 位物理内存的内存系统。假设页大小是 4KB。(a) 展示虚拟和物理地址的布局。(b) 页表会有多大？该内存系统中包含多少个页帧？

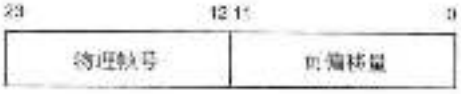
答：

a. 假设页大小为 4KB，那么 32 位虚拟地址的低 12 位就是页偏移量，而剩下的高位（20 位）则是虚拟页号。



由于物理地址是 24 位，它内有 12（即 24–12）位被用来表示物理帧号。

物理地址的布局：



23	12 11 0
物理帧号	页偏移量

b. 页表项数 $= 2^{\text{页表项数的位数}} = 2^{20}$

假设每个项是一个字，即 32 位（4 字节）。

$$\text{页表的大小} = 4 \times 2^{20} \text{ 字节} = 4\text{ MB}$$

$$\text{页表的个数} = 2^{\text{物理帧号的位数}} = 2^{12} = 4096$$

7.4.2 支持分页的硬件

支持分页需要用什么硬件一目了然。我们需要向数据通路添加一个新的寄存器，即页表基址寄存器。在每次内存访问的时候，处理器根据页表基址寄存器里的值，算出对应于虚拟地址的页表项的地址。从这个页表项中读取到的是物理帧号，把它与页偏移量连接在一起以后就得到了物理地址。这就是在流水线处理器的 FETCH 和 MEM 阶段中分别读取指令和数据时都要用到的转换过程。添加到处理器以支持分页的新增硬件出人意料得少，尤其是与它为内存管理带来的巨大好处相比而言。让我们回顾一下采用分页给内存访问带来的额外开销。本质上说，硬件每次访存得访问两次内存：第一次是去读取物理帧号，而第二次则是去读取内存内容（指令或者是数据）。这似乎相当低效，从维持高性能处理器流水线的角度来说是完全不可采取的。幸运的是，可以大幅度地减少这种低效，从而让分页实际上可行。这里的关键在于记住最近地址转换的结果，因为我们很可能会去访问同一个物理页的很多内存位置。处理器首先查询一个被称作旁路转换缓存（TLB，Translation Lookaside Buffer）的表。只有在没有找到映射关系的时候，处理器才会从物理地址里面去读取物理帧号。想知道更多关于 TLB 的信息，请参阅下一章（见 8.6 节）。 [295]

7.4.3 页表的建立

内存管理器在进程启动的时候设置页表。从这个意义上来说，页表有着双重职责。硬件用它来进行地址转换。而它也是内存管理器控制之下的一个数据结构。通过设置页表，内存管理器把进程的页表基址寄存器的值存在进程控制块中。图 7-12 就展示了加上页表基址寄存器之后的进程控制块。

```
typedef struct control_block_type {
            enum state_type state;
            address PC;
            int req_file[SUMREGS];
            struct control_block *next_pcb;
            int priority;
            address PTBR;
            ....
            ....
} control_block;
```

图 7-12 加上了页表基址寄存器字段的进程控制块。只需要加上这一个字段，就足以让操作系统确定进程的内存印迹

7.4.4 虚拟和物理内存的相对大小

根据迄今为止的讨论，看起来虚拟内存的意义就在于把程序从可用物理内存的限制中解放出来。自然，这让我们觉得虚拟内存应该总是比物理内存大。这么考虑当然是完全符合逻辑的。尽管这么说，让我们也来看看，让物理内存比虚拟内存大有没有意义。我们立刻可以看出，这样做对于单个程序来说没有好处，因为程序可以用到的地址空间是受限于虚拟内存的大小的。比如说，如果虚拟地址空间是32位的，给定的程序就只能访问4GB的内存。即使系统的物理内存超过4GB，单个程序也无法拥有超过4GB的内存印迹。但是，更大的物理内存方便让操作系统可以装下更多的消耗内存的进程。这也是Intel体系结构的物理地址扩展（PAE，Physical Address Extension）把物理地址从32位扩展到36位的意义所在。结果就是这个特性让系统可以拥有至多64GB的物理内存，而操作系统可以（通过页表）给特定进程映射4GB虚拟地址空间，以居留在64GB物理内存中的不同部分。[1]

有人也许会这么推断：随着支持多于32位物理地址的技术的到来，处理器体系结构也应该支持更大的虚拟地址空间了。他们错对了。实际上，包括Intel在内的商家已经设计出了64位的体系结构。Intel提供物理地址扩展特性的原因只是允许仍在应用中的运行着遗留应用程序的32位平台使用到更多的内存。

7.5 分段虚拟内存

让我们来考虑一个类比。图7-13展示了一个房子的规划图。它有一个客厅，一个起居室、一个餐厅，可能还有一个书房和一个或多个卧室。换句话说，我们先从逻辑上把房屋的空间划分成若干功能单元。然后，我们可以根据房屋的总可用空间来把实际的物理空间分配给各个房间。为此，我们可能会决定让客人的卧室比比方说小孩的卧室略大一些。我们也许会有一个与正式的餐厅相比更加温馨的早餐区域，等等。这种对空间的功能性组织有多个好处。如果你有访客，你不必重整家里的任何东西，只需要让他们住进访客卧室即可。如果你决定带朋友来家里过夜，只需与该朋友共享你的卧室即可，而不用打扰到家里的其他成员。让我们把这个类比应用到程序开发上去。

在前一章中（参见6.2节），我们把进程的地址空间定义为程序的内存印迹占用的空间。在前一节中，我们强调了维持用户程序的内存印迹是连续的这个景象是必需的。我们现在更进一步，强调让地址空间从0开始到某个最大值的重要性，因为它对编译器生成代码来说很便利。虚拟内存可帮助实现这个景象。

让我们研究一下地址空间对程序来说是否足够。与房屋空间安排的类比在这里很有用。我们把房屋从逻辑上根据用途划分成不同的房间。这些空间彼此独立且受到保护（门和锁等）。这确保了没有人可以在不事先告知的情况下闯进对方的私人空间。在某个层面上，构建程序类似于设计房屋。尽管我们最后得到的是单一的程序（在UNIX术语中称作a.out），源代码是有逻辑结构的。不同的数据结构和过程组织起来以负责提供特定功能。如果这是一个团队项目，你甚至会属于一个开发团队，其中每个人都负责整个程序的不同功能。你可以想象一下，开发诸如微软Word一类的复杂软件时参与的软件工程师数目。因此，有多个地址空间可用会有助于把程序从逻辑上组织得更好。尤其是有了面向对象编程之后，有多个地址空间

296

[1] 如果想了解更多信息，建议感兴趣的读者参阅Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. [Intel System programming guide 3A, 2008]

的好处怎么强调都不为过。



图 7-13 房屋户型布局设计[3]。Edenlane Homes, Inc. 公司版权所有并保留所有权利。本书经过许可使用该图

让我们再深入一点，理解有多个地址空间会如何帮助开发者。即使只是考虑基本的内存印迹（参见第 6 章的图 6-1），我们可以让内存印迹的不同部分——即，代码、全局数据、堆和栈——被放在不同的地址空间里。从软件工程的角度来看，这种安排让我们可以给不同地址空间以不同属性（比如说，代码段是只读的，等等）。此外，能够给不同的地址空间分配属性的能力对于调试程序来说极其有用。

在大型程序的开发中使用多个地址空间的需要就更加迫切。比方说，我们要写一个视频监控系统。它可能有多个组件，如图 7-14 所示。你能看出有多个组件，并且每个组件拥有各自的地址空间的程序与图 7-13 中的房间布局方案之间的相似之处。

297
~
298

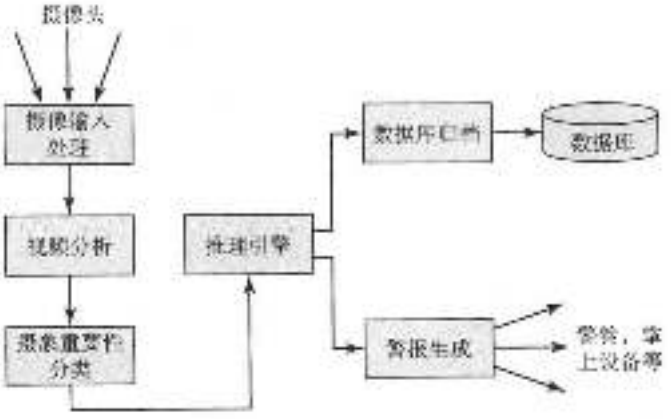


图 7-14 一个样例应用程序：视频监控系统。这类复杂的应用程序会由若干模块组成，并且很可能是由一支软件工程师团队负责开发

[3] 资料来源：www.edenlanehomes.com/images/design/Typical-Floorplan.jpg。

这是个足够复杂的工作，以至于得要很多人一起为项目工作。团队成员在使用具有良定义的接口的情况下，可以彼此独立地开发图 7-14 所示的各个方框。让这些组件分别在不同的地址空间运行，并且配上合适的保护和共享级别（类似于在房屋设计中的门和锁），会对开发和调试给予极大的帮助。更进一步地说，这也使得维护这样的应用程序变得容易。如果你只是要给厕房铺地砖，就不必临时搬进酒店住一阵子。类似地，你可以重写该应用程序的特定功能模块，而不影响其他部分。

分段就是实现前述景象的技术。与所有系统级机制一样，本技术也是操作系统与体系结构的合作的成果。

内存在用户眼中的景象并不是单一的线性地址空间，而是由多个不同的地址空间组成。每个这样的地址空间都被称作一个段。段有两个属性：

- 唯一的段编号
- 段大小

每个段都从地址 0 开始，直到（段大小 -1）未知。CPU 产生的地址由两部分组成，如图 7-15 所示。

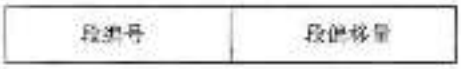


图 7-15 分段的地址

299

和分页一样，内存代理在 CPU 和内存之间，通过查找分段表来把地址转换为物理地址（见图 7-16）。与分页一样，也是操作系统负责给当前运行的进程设置段表。

现在你很可能会想，除了名字从页变成段以外，看起来分页和分段之间没有多少区别。在深入讨论两者的差异以前，让我们回到图 7-14 中的样例应用程序。采用分段，我们可以将这个应用程序安排成图 7-17 那样。注意每个功能模块都在自己的段中，而各个段分别有一个大小，取决于对应组件的功能。图 7-18 展示了这些段在物理内存中的布局。

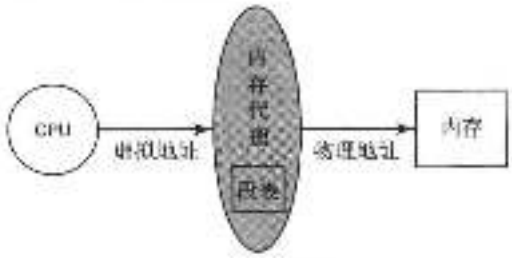
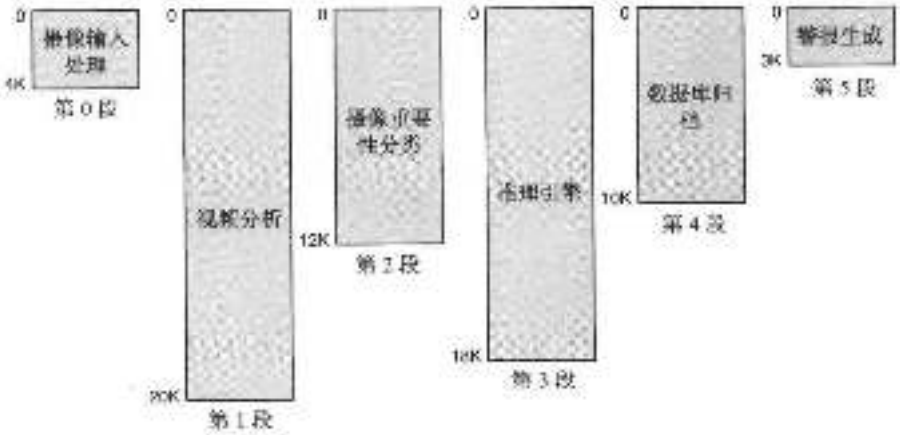


图 7-16 段表



300

图 7-17 样例程序被组织成多个段



图 7-18 图 7-17 中应用程序的各个段映射到物理内存中

例 7-5 某程序有 10KB 的代码空间和 3KB 的全局数据空间。它需要 5KB 的堆空间和 3KB 的栈空间。编译器给这个程序的每个组件分别分配了一个段。物理内存分配如下所示：

代码起始地址	1000
全局数据起始地址	14000
堆空间起始地址	20000
栈空间起始地址	30000

a. 画出该程序的段表。

答：

301

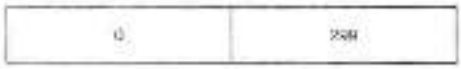
段编号	起始地址	大小
0	1000	10 KB
1	14000	3 KB
2	20000	5 KB
3	30000	3 KB

b. 假设内存按字节寻址，画出内存布局。

答：

c. 给出与以下虚拟地址对应的物理地址：





[302]

答：

1. 偏移量 299 是在第 0 段的大小范围内（10KB）。

2. 物理内存地址

= 第 0 段起始地址 + 偏移量

= 1000 + 299

= 1299

7.5.1 支持分段的硬件

支持分段所需的硬件很简单。段表是个类似于页表的东西。段表里的每一项被称作段描述符。段描述符给出了段的起始地址和大小。每个进程都有自己的段表，由操作系统在创建的时候负责分配。类似于分页的是，这个机制也需要在 CPU 中的一个专门的寄存器，称作段表基址寄存器（Segment Table Base Register，STBR）。该硬件采用该寄存器以及段表来在进程执行的时候进行地址转换（参见图 7-19）。硬件首先会进行边界检查，以确保提供的偏移量在该段的限制范围内，然后再继续处理内存访问。

读者应当会回忆起 7.3.2 节中讲到的变长分区的内存分配方案。分段也会遇到和变长分区同样的问题，也就是外部碎片。这可从图 7-18 中看出。

7.6 分页和分段的比较

现在我们已经准备好来理解分段和分页的区别了。两个都是实现虚拟内存的技术，但是在细节上差别很大。我们在表 7-1 中总结这两个方法的异同。

乍一看，你会觉得分段有很多好处。因此很容易得出结论，认为体系结构应该选用分段来作为实现虚拟内存的载体。不幸的是，表格的最后一行，即外部碎片，才是真正重要的一点。也还有其他的考虑，比如说，采用分页时 CPU 生成的虚拟内存地址占用一个内存字，但是采用分段的话也许必须得用两个内存字才能指定一个虚拟地址。这是因为我们可能想要让每个段都能与总可用地址空间一样大，以求尽量大的灵活性。这意味着我们需要一个内存字

来表明段编号。另一个内存字来表明该段内的偏移量。另一个重要的系统级的考虑是平衡整个系统。让我们详细讨论这句话意味着什么。实际上，应用程序和系统软件对内存的需求一 303
直持续增长。桌面出版应用程序和浏览器的每个版本的内存印迹的增长是这种增长的例证。原因当然是给最终用户提供更多功能的渴望。现实是我们永远没法提供足够的物理内存满足我们的胃口。因此，虚拟内存必须得远大于物理内存。页和段得能够"按需"地从硬盘上被加载进物理内存中。虚拟内存把内存系统从物理内存扩展到磁盘中。因此，从磁盘向内存按需传输数据必须得高效，才能使整个系统高效运转。这就是我们所说的对系统整体进行平衡。由于页大小是个系统属性，采用分页的话对系统整体进行平衡会较为容易。由于用户对段的大小可以进行控制，在使用分段的情况下对系统整体进行优化就困难一些。

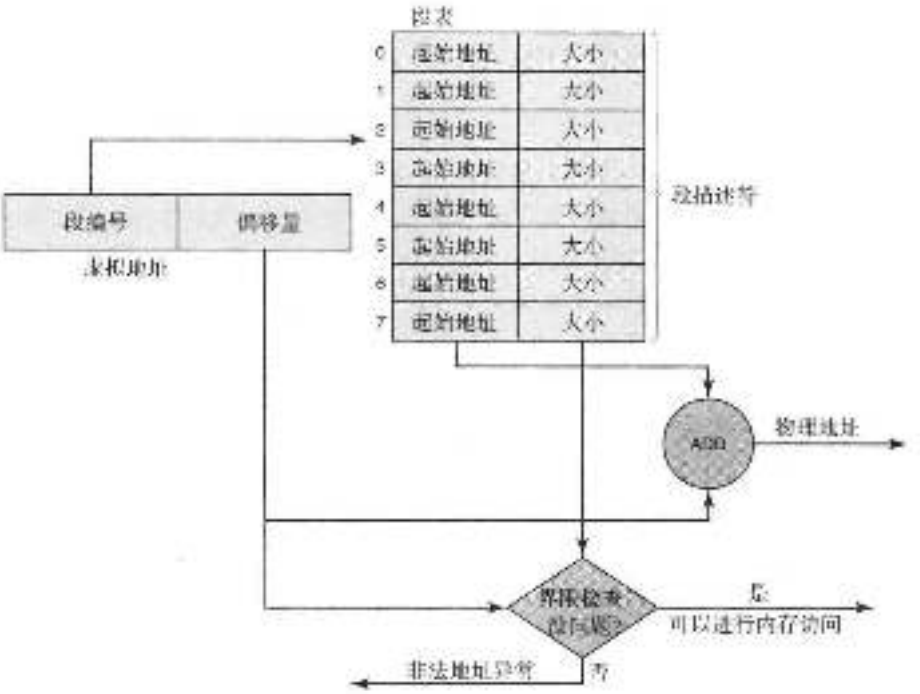


图 7-19 采用分段机制的地址转换。虚拟地址中的段编号用作段表中的下标来获得段的起始地址。虚拟地址中的偏移量则加到起始地址之上，以生成实际的物理地址发送给内存 304

表 7-1 分页和分段的对比

属性	分页	分段
用户受到保护，不直接受到选择内存大小的限制	是	是
与物理内存的关系	物理内存可能比虚拟内存多或者少	物理内存可能比虚拟内存多或者少
每个进程的地址空间	一个	多个
对用户的可见性	用户不知道分页的存在；用户有使用单一线性地址空间的错觉	用户知道有多个地址空间，都从 0 开始

（续）

属性	分页	分段
软件工程	没有明显的好处	允许程序组件根据用户需求组织成多个段；使得模块化设计成为可能 增加可维护性
程序调试	没有明显的好处	受到模块化设计的帮助
共享和保护	用户没有直接的控制权 操作系统可以在地址空间之间实现共享和保护，但是从用户的角度来看没有意义	用户可以直接控制多个段的共享和保护 对面向对象编程和大型软件开发较为有用
页和段的大小	被体系结构固定下来	可变。用户可以为每个段分别选择
内部碎片	可能有内部碎片。因为地址空间里某个页的一部分可能用不到	无
外部碎片	无	可能有外部碎片。因为变长的段必须在可用的物理内存中分配，从而产生问题（参见图7-18）

305

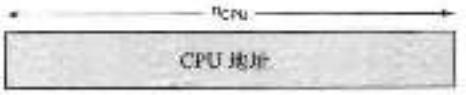
由于这些原因，本节描述的真正的分段不是实现虚拟内存的可行方案。一种解决外部碎片的方案是我们在7.3.3节中描述的那样，即使用内存缩并。但是，我们也观察过了实践中实现内存缩并的难点。一个更好的方法是采用一个组合技术，即页式分段。用户得到的是一个分段的系统，如本节描述。而在内部，操作系统和硬件采用分页，如上节所描述的，以消除外部碎片的不良影响。

关于这样的页式分段技术的深入讨论超出了本书范围。我们会在之后的章节里从历史的视角讲述分页和分段，还有一个商用的页式分段的例子，拿Intel Pentium体系结构作为案例研究。

7.6.1 解读CPU生成的地址

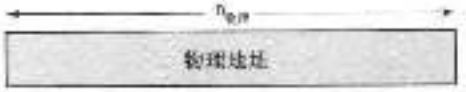
处理器生成简单的线性地址来在内存中寻址。

CPU生成的地址：



CPU生成地址中的位数取决于处理器的寻址能力（通常与处理器字长以及对内存操作数的访问的最小粒度相关联）。比如说，对于按字节寻址的64位处理器来说，$n_{CPU}=64$。

物理地址的位数取决于物理内存的实际大小。



如果采用的是按字节寻址的内存，则有

$$n_{phy}=\log_2(\text{物理内存的大小，以字节计})$$

例如，如果物理内存大小是1GB，那么$n_{phy}=30$位。

把CPU生成的线性地址解释为虚拟地址的具体方法取决于内存系统体系结构（即，分页还是分段）。相应地，物理地址的计算方式也要变更。表7-2总结了与分段和分页内存系统相关的主要公式。

306

表7-2 分段和分页内存系统中的地址计算

内存系统	虚拟地址计算	物理地址计算	表大小
分段	n_{seg} 段编号 | n_{off} 段偏移量 $n_{off} = \log2(\text{段大小})$ $n_{seg} = n_{va} - n_{off}$	段起始地址[⊖] = 段表[段编号] 物理地址 = 段起始地址 + 段偏移量	段表大小 = $2^{n_{seg}}$个项目
分页	n_{vpn} 虚拟页号 | n_{off} 页偏移量 $n_{off} = \log2(\text{页大小})$ $n_{vpn} = n_{va} - n_{off}$	物理帧号[⊜] = 页表[虚拟页编号] 物理地址： n_{pfn} 物理帧号 | n_{off} 页偏移量 $n_{off} = \log2(\text{页大小})$ $n_{pfn} = n_{物理} - n_{页偏移量}$	页表大小 = $2^{n_{虚拟页号}}$个项目

307

小结

内存系统的重要性怎么强调也不为过。系统整体的性能关键上依赖于内存系统的效率。组成内存系统的硬件和软件之间的交互使得对内存系统的研究令人着迷。迄今为止，我们讲了若干个不同的内存管理方案以及实现这些方案所需的硬件需求。在本章的开始处，我们找出了内存系统的四个标准：提高资源利用率、进程内存空间的独立和保护、从内存资源限制中的解放，以及并发进程的内存共享。从操作系统管理内存这种稀缺资源的效率的角度来说，这些都很重要。从关于分段的讨论中，我们给内存管理添加了另一个同样重要的评判标准，即促进好的软件工程实践。这个标准是说，内存管理方案除了要达到系统级的标准以外，还要能帮助我们开发灵活、易维护、持续发展的软件。让我们依照这些标准对这些方案进行总结。表7-3给出了这些内存管理方案的定性对比。

表7-3 内存管理方案的定性对比

内存管理标准	用户/内核分离	固定分区	变长分区	分页虚拟内存	分段虚拟内存	页式分段虚拟内存
提高资源利用率	无	不超过分区大小的内部碎片，外部碎片	外部碎片	不超过分区大小的内部碎片	外部碎片	不超过分区大小的内部碎片
独立和保护	否	是	是	是	是	是
从资源限制中解放	否	否	否	是	是	是
并发进程的共享	否	否	否	是	是	是
促进好的软件工程实践	否	否	否	否	是	是

308

⊖ 这是通过在段表中以段编号为索引查找得到的。

⊜ 类似地，这是通过在页表中以虚拟页编号为索引查找得到的。

只有采用页式分段的虚拟内存符合所有标准。因此，有必要回顾一下那些方案在当前内存管理的前沿中还有用处。表7-4总结了本章提到的内存管理方案，以及所需的硬件支持以及它们在现代系统中的适用性。

表7-4 内存管理方案总结

方案	硬件支持	是否仍在使用中
用户/内核分隔	模式寄存器	有
固定分区	界限寄存器	未使用在任何产品操作系统中
变长分区	基址+限长寄存器	未使用在任何产品操作系统中
分页虚拟内存	页表和页表基址寄存器	是，在绝大多数现代系统中
分段虚拟内存	段表和段表基址寄存器	纯静态分段未使用在任何商业上流行的处理器中
页式分段虚拟内存	分页和分段的硬件的组合	是，在绝大多数基于Intel x86的操作系统中[⊖]

历史回顾

大约在1965年前后，IBM推出了System/360系列大型机。该体系结构提供了基址和限长寄存器，也就为支持动态重定位的内存管理铺平了道路。任何一个通用寄存器都可以用作基址寄存器。编译器则会选择一个特定的寄存器来用作基址寄存器，从而任何以高级语言编写的程序都可以被操作系统动态重定位。但是，有个小问题。程序员可以找出哪个寄存器被用作基址寄存器，便可以利用这一点来把基址寄存器的值"藏起来"以便插入该高级语言程序的汇编代码使用。[⊜]这么做的结果就是程序在执行之后就不再能重定位了，因为程序员可能已经在程序里硬编码了地址。你也许会奇怪，他们怎么会做这种事情呢？对于追求从系统中榨取每一点性能的程序员来说，往高级语言代码里塞进去一些汇编代码实在是司空见惯的事情。我们当中的一些至今还在这么做！由于这些原因，动态重定位一直没能像IBM所希望的那样良好运行。主要的原因就是在体系结构里用来进行地址偏移的寄存器是个程序员可见的通用寄存器。

309

大约在1970年前后，IBM在它们的System/370系列大型机中引入了虚拟内存。[⊝] System/370与System/360本质上的不同之处就是体系结构支持动态地址转换，也就消除了之前提到的System/360的问题。System/370代表了IBM第一次对虚拟内存概念的支持。System/370系列的后继型号则通过扩展寻址能力改良了虚拟内存模型。该体系结构采用分页虚拟内存。

在这个语境下，值得花一下术语静态和动态的使用随着操作系统或者体系结构的定位而略有不同。早期时（参见7.2节）我们从操作系统的视角定义了静态和动态重定位是什么。带着这个定义，你可以说如果硬件支持在运行时修改虚拟到物理的映射，该程序就是动态可重定位的。根据这个定义，IBM的360系列的基址加限长寄存器是支持动态重定位的。

体系结构设计师则以更细粒度观察程序运行时的单个内存访问，使用术语地址转换。如果从虚拟到物理的映射可以在程序执行的任意阶段被修改，这个体系结构就支持动态地址转换。根据这个定义，IBM 360的基址和限长寄存器就只能支持静态地址转换，而分页才支持

⊖ 应当注意，Intel的分段与纯形式的分段大为不同。我们随后将会讨论Intel的页式分段方案。

⊜ 来自与University of Wisconsin-Madison的James R. Goodman的私人通信。

⊝ 想要对这些描述System/360和System/370的权威论文，请参阅www.research.ibm.com/journal/rd/255/ibmrd2505D.pdf。

动态地址转换。操作系统对静态和动态的定义是从整个程序来说的，而体系结构对静态和动态的定义则是从单个内存访问来说的。

甚至在 IBM 进入虚拟内存的世界之前，Burroughs 公司就在它们的 B5000 系列机器中引入了分段虚拟内存的概念 [Oliphint，1987]。通用电气与 MIT 的 MULTICS 项目合作于 20 世纪 60 年代中叶在它们的 GE 600 系列机器中引入了更复杂分段机制 [Schroeder，1971]。IBM 通过推出 VM/370 操作系统以支持虚拟内存，以及持续不断地对 System/370 系列机器的分页虚拟内存的改进，迅速确立了 20 世纪 60 和 20 世纪 70 年代大型机之战的胜利。这场革命持续到今天，你甚至在支撑企业级应用 IBM z 系列大型机[⊖]中仍然能看出它与早期机器之间的联系。 310

MULTICS

有些学术性的计算机项目对所在领域的发展一直保持着深远的影响。MIT 的 MULTICS 项目就是一个这样的例子。该项目源自 20 世纪 60 年代，并且你很容易就能看出我们所知道的计算机系统中有很多东西出生于该项目（UNIX、Linux、分页、分段、安全、保护，等等）。在某种意义上来说，MULTICS 项目中提出的操作系统概念超前于时代，而那个时候的处理器体系结构则还没有准备好支持 MULTICS 所鼓吹的内存保护这种高级观点。MULTICS 项目是本书主旨的绝佳例子，即系统软件和机器体系结构的相连性。

MULTICS 引入了页式分段的概念[⊖]。图 7-20 画出了 MULTICS 所实现的方案。

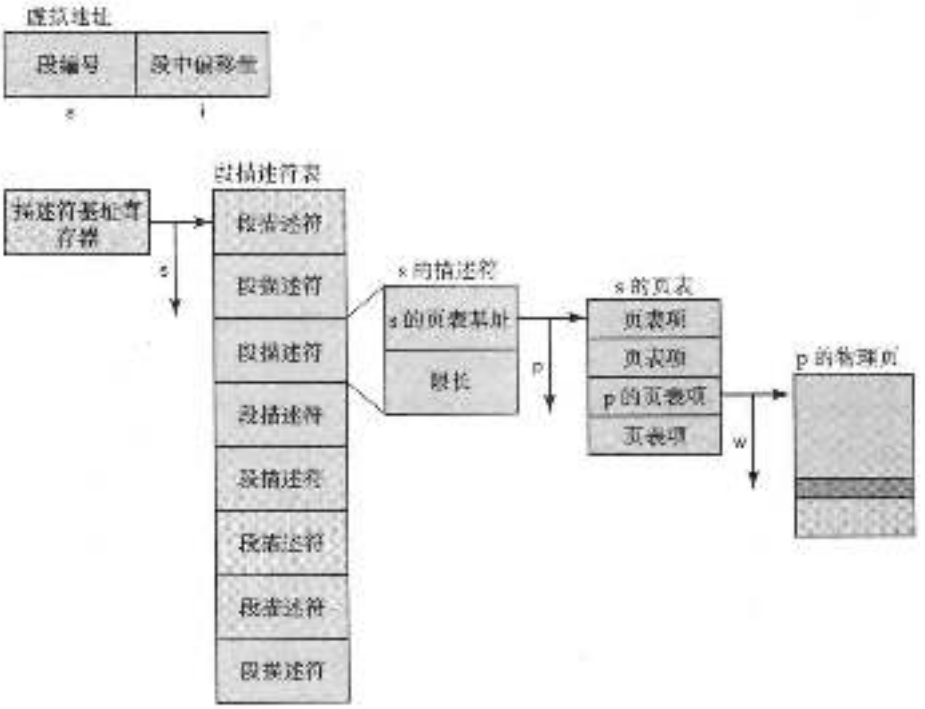


图 7-20 MULTICS 中的地址转换 311

CPU 生成的 36 位虚拟地址由两部分组成：18 位的段编号（s）和 18 位的段内偏移量（i）。

⊖ 资料来源：www-03.ibm.com/systems/z/。

⊖ 参见原始的 MULTICS 论文：www.multicians.org/multics-vm.html。

每个段都可以任意大，只要不超过段大小的上限 $2^{16}-1$ 即可。为了避免外部碎片，每个段都由页组成（在 MULTICS 里，页的大小是 1024 个字，每字 36 位）。每段都有自己的页表。在 MULTICS 中，动态地址转换分为两步。

- 定位到对应于段编号的段描述符：硬件通过把保存在一个叫做描述符基址寄存器的寄存器中的段表基址与段编号相加以进行此次查找。
- 段描述符包含该段页表的基址。根据虚拟地址中的段偏移量，以及页大小，硬件计算出对应于该虚拟地址的特定的页表项。最后，通过连接物理页编号和页内偏移量（就是段内偏移量除以页大小的余数），就得到了物理地址。

如果某虚拟地址中段编号为 s，而段内的偏移量为 i。那么，我们在寻找的内存位就处在该段的第 p 个页的偏移量 w 处。其中：

$$w = i \bmod 1024$$

$$p = (i - w) / 1024$$

图 7-20 展示了这种地址转换的过程。

Intel 的内存体系结构

Intel 的 Pentium 系列处理器也采用了页式分段。但是，它的组织结构与 MULTICS 的简单方案相比更加复杂。学术项目与工业产品之间争论的实际情况之一就是，后者的处理器系列必须得考虑它的向后兼容性。向后兼容性意味着新处理器作为之前型号的后继者，必须能够运行以前的代码，而完全不需要对其进行修改。这是个麻烦事，对新处理器的设计造成了很多限制。Intel Pentium 当前的内存体系结构从 Intel 的史早期的 x86 体系结构发展而来，比如 80286。因此，它保留有旧处理器中的分段机制，外加对于给软件开发提供大量虚拟地址的美学要求。

我们有意对本节的讨论进行了简化。作为一种近似，虚拟地址是段选择器加上偏移量（参见图 7-21）。Intel 的体系结构把总的段空间分成两半：*系统*和*用户*。系统段对于所有进程来说是共通的，而用户段则每个进程各不相同。你可能已经猜到了，由于对所有进程都一样，系统段是给操作系统用的。对应的是，有一个对所有进程都一样的描述符表，称作全局描
312 述符表（Global Descriptor Table，GDT），还有一个每个进程独有的表，称作局部描述符表（Local Descriptor Table，LDT）。Intel 中的段选择器类似于 MULTICS 中的段编号，只有一个区别：段选择器中有一个位表示该虚拟地址所用到的段名是系统段还是用户段。

和在 MULTICS 中一样，选定段的段描述符包含用来把虚拟地址中指定的偏移量转换成物理地址的详细信息。这里的区别是可以选择是使用不带任何分页的简单分段（以与更早的处理器相兼容）还是使用段页式管理。这可以从图 7-21 中看到。（地址转换是采用 GDT 还是 LDT 是由选择器的用户 / 系统位来决定的。）计算出来的实际地址就是物理地址。在页式分段的情形中，描述符中存储的基址是对应于此段编号的页表基地址。地址转换的剩余部分和我们之前给 MULTICS 描述的一样，通过选定段的页表进行转换（参见图 7-20）。某个全局控制寄存器用来控制是采用纯分段还是页式分段。

如果你想了解更多关于 Intel 的虚拟内存体系结构的知识，请参阅 Intel 的系统编程指南[⊖]。

⊖ Intel® 64 and IA-32 Architectures Software Developer's Manual Volumn 3A: System Programming Guide [Intel System programming guice 3A, 2008].

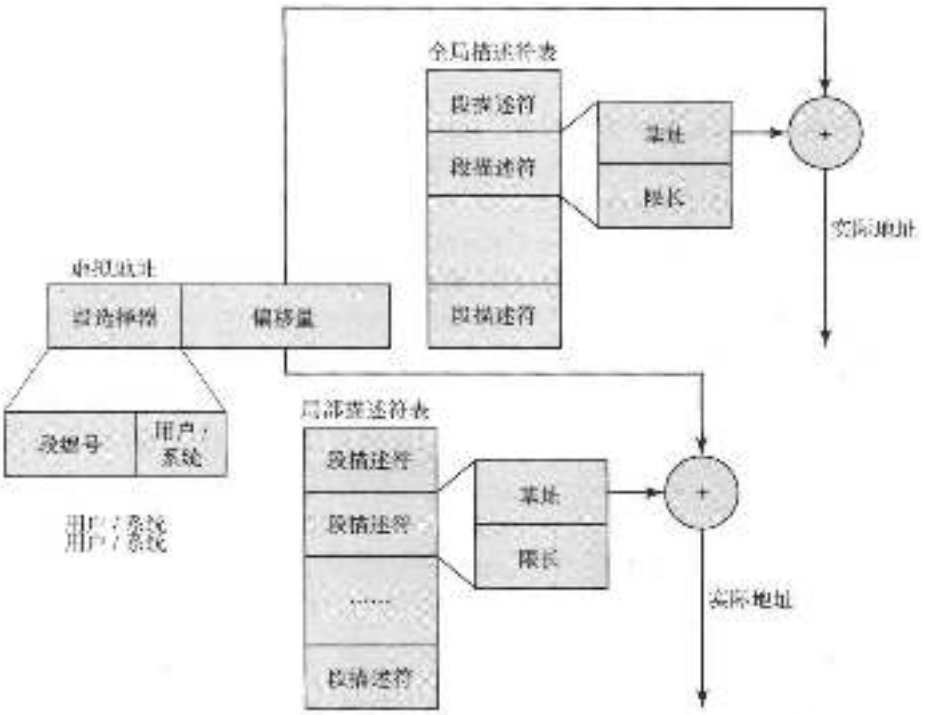


图 7-21 在 Intel Pentium 中采用纯分段的地址转换 313

练习题

1. 内存管理的主要目标有哪些?
2. 说出你是支持还是反对以下观点，并给出理由：既然内存很便宜，而且我们能弄到很多内存，也就不再需要内存管理了。
3. 比较并说出内部和外部碎片的异同。
4. 内存管理器以固定的 2048 字节大小为单位分配内存。当前的分配结果如下所示：

   P1 1200 字节

   P2 2047 字节

   P3 1300 字节

   P4 1 字节

   根据以上分配结果，请问由于内部分配总共浪费了多少内存?
5. 判断正误，并给出理由：内存缓冲通常是与固定尺寸分区的内存分配方案一起用的。
6. 判断正误，并给出理由：用基址和限长寄存器来进行管理，与用界限寄存器相比，没有什么特别的好处。
7. 假设某体系结构用基址和限长寄存器进行内存管理。内存管理器采用变长分区分配。当前的内存分配如下所示：



314

有个需要 9KB 内存的新申请。此次申请能否被满足？如果不能的话，为什么？图中所表示的情形中外部碎片的量是多少？

8. 在分页内存系统中页大小与帧大小有什么联系？

9. 从所需硬件资源（新增的处理器寄存器个数，以及用来针对给定的 CPU 生成的内存地址计算出物理内存地址的额外电路）这个角度，对比基址加限长寄存器与分页虚拟内存这两个解决方案的异同。

10. 分页虚拟内存系统为什么能消除外部碎片？

11. 推导分页内存系统在页大小为 p 的情况下的最大内部碎片。

12. 一个系统中虚拟地址有 20 位，页大小为 1KB。页表中有多少项？

13. 一个系统中物理地址为 24 位，页大小为 8KB。物理帧的最大数目是多少？

14. 说出分页虚拟内存和分段虚拟内存的差别。

15. 给定以下段表：

段编号	基址地址	大小
0	3000	3 KB
1	15000	3 KB
2	25000	5 KB
3	40000	8 KB

对应于以下虚拟地址的物理地址是多少？

1	399

16. 从所需硬件资源（新增的处理器寄存器个数，以及用来针对给定的 CPU 生成的内存地址计算出物理内存地址的额外电路）这个角度，对比分页和分段的内存系统的异同。

参考文献注释和扩展阅读

Elliot Organick 的书 [Organick，1972] 是一本很好的历史文献，其中介绍了 MULTICS 项目。该项目探索了包括页式分段在内的若干个先锋想法，历经了时间的考验。IBM 在内存系统发展中所扮演的角色在 [IBM system/360，1964] 和 [IBM System/370，1978] 中有记载。[Intel System programming guide 3A，2008] 是关于 Intel 的内存体系结构的一份良好的文档。

页式内存管理

本章我们主要讨论**页式内存管理**（page-based memory management）。这项技术对于支持虚拟内存的绝大多数处理器和操作系统都很重要。正如前文提到的，即使是支持段式管理的处理器（如英特尔奔腾处理器）也使用页式管理来消除外部碎片。

8.1 按需分页

正如我们在第 7 章中提到的，程序启动时内存管理器会为处理器建立一个页表。我们首先来弄明白程序启动时内存管理器在内存中分配整个程序的哪些部分。生成的程序包含应用程序的功能部分和算法逻辑部分，以及在程序执行过程中出现错误的情况下的非功能部分。这样，可以预测对运行情况良好的程序，只在内存加载整个程序中很小的一部分就可以正常执行。所以，当程序启动时内存管理器不将整个程序加载进内存是一件需要谨慎考虑的事情。这需要对在内存中执行一个不完整的程序意味着什么有深入的理解和思考。基本想法是加载那些不在内存中但又有需要的程序部分。这种按需分页的技术会有更好的内存利用率。



图 8-1　页表项，按需分页需要在页表项中添加一个有效位

首先，让我们来了解按需分页在硬件和软件上都发生了什么。

[316]

8.1.1 按需分页的硬件

在第 7 章中（见 7.4.1 节），我们提到硬件从页表中选取物理帧号（PFN）作为地址转换的一部分。但对于按需分页，页也许还没有加载到内存中。所以我们需要页表中的额外信息来了解页是否在内存中。我们向每个页表项添加一个有效位。如果有效位是 1 就说明这项的 PFN 字段是有效的；否则是无效的，表示页面不在内存中。图 8-1 展现了支持按需分页的 PTE。硬件所扮演的角色是识别出无效的 PTE 并帮助操作系统执行正确的操作，即向内存加载缺失的页。这一情况（PTE 无效）是一个程序中断意外，因为在原程序中是没有错误的。操作系统为了节省内存资源决定不加载这部分程序，这种程序中断表现为页错误异常或陷入。

操作系统通过从断点引入缺失页来处理这种错误。一旦页被加载进内存，程序会准备好从缺失处恢复执行。所以，为了支持按需分页，处理器应该能够重新启动那些在执行过程中由于页错误被暂停的指令。

图 8-2 表示处理器流水线。IF 和 MEM 阶段因为涉及内存访问所以很容易受到页错误的影响。

为指令重启的硬件　我们首先来理解在硬件中会发生什么。假设指令 I_2 在 MEM 阶段有一个页错误，并且在流水线局部执行中有一些指令。在处理器进入 INT 阶段解决中断问题前（参见第 4 章硬件如何在 INT 阶段解决中断问题），必须注意那些已经在流水线局部执行中的指令。在第 5 章，我们简要地提及了流水线处理器处理中断的措施。指令 I_2 在 MEM 阶段经

历的页错误异常也是相同的。处理器会完成 I_1 指令并且在进入 INT 阶段前压缩指令 $I_2 \sim I_5$ 的执行。为了在页错误之后能够重启指令，INT 状态需要保存涉及指令 I_2 的 PC 值。注意，目前为止压缩指令 $I_3 \sim I_5$ 没有坏处，因为它们没有改变程序的持久状态（在处理器的寄存器和内存中）。页错误存在一个有趣并且关键的副作用（任何其他的异常也会这样）：流水线寄存器（见图 8-2 中的缓冲区）包含异常（EX 阶段的运算或者 MEM 阶段的页错误）事件里指令执行时的 PC 值。我们在第 5 章已经讨论了在流水线处理器中处理陷入和异常的硬件后果。

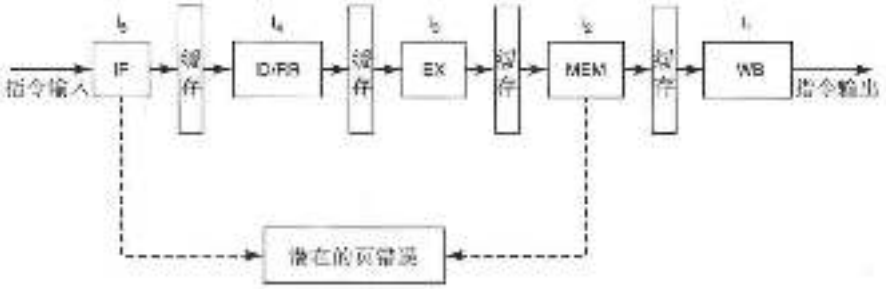


图 8-2 处理器流水线潜在的页错误。IF 和 MEM 阶段访问内存操作数，所以容易遇到页错误

8.1.2 页错误处理程序

页错误处理程序和前面讨论过的其他中断处理程序很类似。我们知道任何处理程序都需要采取的基本措施（状态保存 / 状态恢复）；我们在第 4 章已经了解过。在这里，我们关心处理程序为了纠正页错误会进行哪些具体工作：

1）搜索一个空闲的页帧。

2）从磁盘向空闲页帧加载出错的虚拟页。

3）为缺页异常进程更新页表。

4）重新将进程控制块（PCB）放入调度器的准备好队列。

在接下来的章节中我们将讨论这些细节。

8.1.3 按需分页内存管理的数据结构

现在我们来探讨按需分页的数据结构和算法。首先来看数据结构。我们知道页表是内存管理器维护的针对每个进程的数据结构。除了应用了页表，内存管理器针对页错误也用如下数据结构：

1）**空闲页帧表** 这个数据结构包含内存管理器目前没有用到的页帧信息，这些页帧都用来处理页错误。空闲页帧表不包括本身；空闲页帧表中的每个节点仅包含页帧号。例如（见图 8-3），页帧 52，20，200，…，8 是目前没有使用的页帧号。所以，内存管理器可以使用列表中的任何页帧来处理一个页错误。注意，当机器启动时，因为没有用户进程，空闲列表包含用户空间的所有页帧。内存管理器会分配和释放内存，空闲列表针对进程的页错误会随之减少或增加。

317
↓
318

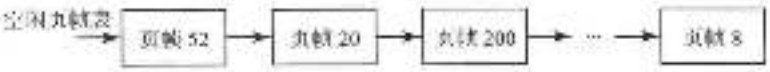


图 8-3 空闲页帧表。这些是内存管理器用于处理页错误的空闲帧

2）**页帧表（FT，Frame table）** 这个数据结构包含反向映射。给定一个页帧号，页帧表将

返回进程 ID(PID)和目前占用页帧的虚拟页号（见图 8-4）。例如，页帧 1 目前处于空闲状态，而页帧 6 被进程 4 的虚页 0 占用。接下来很快来会讨论内存管理器如何用这种数据结构。

3）磁盘映射（DM，Disk map） 这种数据结构将进程的虚拟空间映射到包含页内容（见图 8-5）的磁盘位置上。磁盘映射和页表类似，每个进程都有一个这样的数据结构。

为了讨论清楚，我们展示的每个给定的数据结构互不相同。通常内存管理器会为了提高效率或因为它们有相同的数据结构而合并其中的一些数据结构，并通过多种角度的观察来模拟功能行为。在我们讨论页替换策略之后会有更好的理解（见 8.3 节）。

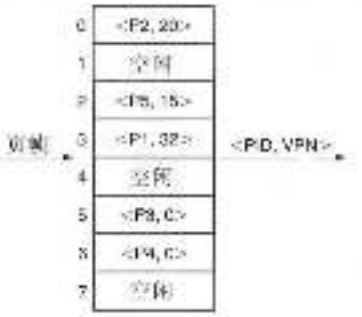


图 8-4 页帧表。这是内存管理器用于反向查询的数据结构，例如，给定一个页帧，通过这个页帧表会找到对应的进程和虚页 [319]

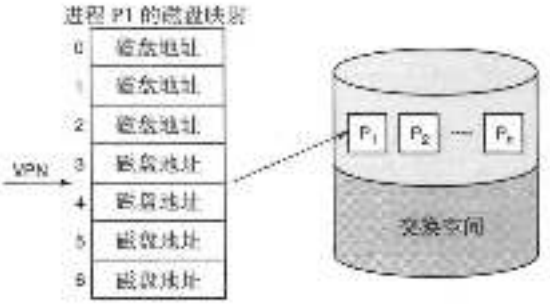


图 8-5 进程 P1 的磁盘映射。这种数据结构允许内存管理器通过磁盘块定位反向查找进程的虚页

8.1.4 页错误解析

让我重新来看页错误异常处理程序发现页错误时是如何通过下面的数据结构工作的。

1）发现一个空闲页帧 页错误处理程序（内存管理器的一部分）查找空闲页帧表。如果页表是空的则出现问题，这意味着所有的物理页帧都被使用了。然而，为了保证缺页的进程能够继续运行，内存管理器不得不将缺失的页从磁盘装入实际物理内存中。这意味着我们要在物理内存中为缺失的页留出空闲空间。所以，内存管理器会挑出一些物理页帧作为被替换页，用于处理缺失的页。关于被替换页的选择策略在 8.3 节进行讨论。

2）挑选被替换页 在选择被替换页帧的过程中，内存管理器决定包含被替换页的被替换进程。页帧表在做决定时就会派上用场。我们要区分干净页和脏页的概念。干净页指的是程序从磁盘引入内存中就再也没有被改变的页面，所以，磁盘中对应的部分和干净页是一样的。另一方面，脏页指的是程序从磁盘中引入内存后改变过的页面。如果被替换页是干净页，内存管理器要做的是将页表项（PTE）中的有效位设为无效，即这一页的内容不需要保存。然而，如果这一页是脏页，内存管理器需要通过被替换进程的磁盘映射来获取磁盘地址信息，将此页写回磁盘中（通常指冲刷到磁盘，flushing to the disk）。

3）加载缺失页 内存管理器通过缺页异常处理程序的磁盘映射从磁盘中读出缺失的页并保存在选定的页帧中。 [320]

4）**为缺页异常进程和帧表更新页表** 管理器设置缺页异常进程的 PTE 的映射指向被选中的页帧，并将有效位设为有效。同时也要更新帧表来处理页帧映射的变化。

5）**重启缺页异常进程** 此时缺页异常进程已经准备好了重新启动。内存管理器将缺页异常进程的控制块（PCB）放入调度器的准备好队列中。

例 8-1 假定进程 P1 正在执行，在虚拟页码（VPN）为 20 的地方经历了一个页错误，而空闲页帧表是空的，管理器选择页帧号 PFN=52 作为被替换帧。这个帧目前在进程 4 中编号为 33。图 8-6a 和图 8-6b 显示了处理页错误前后页表和页帧表的变化情况。

答：

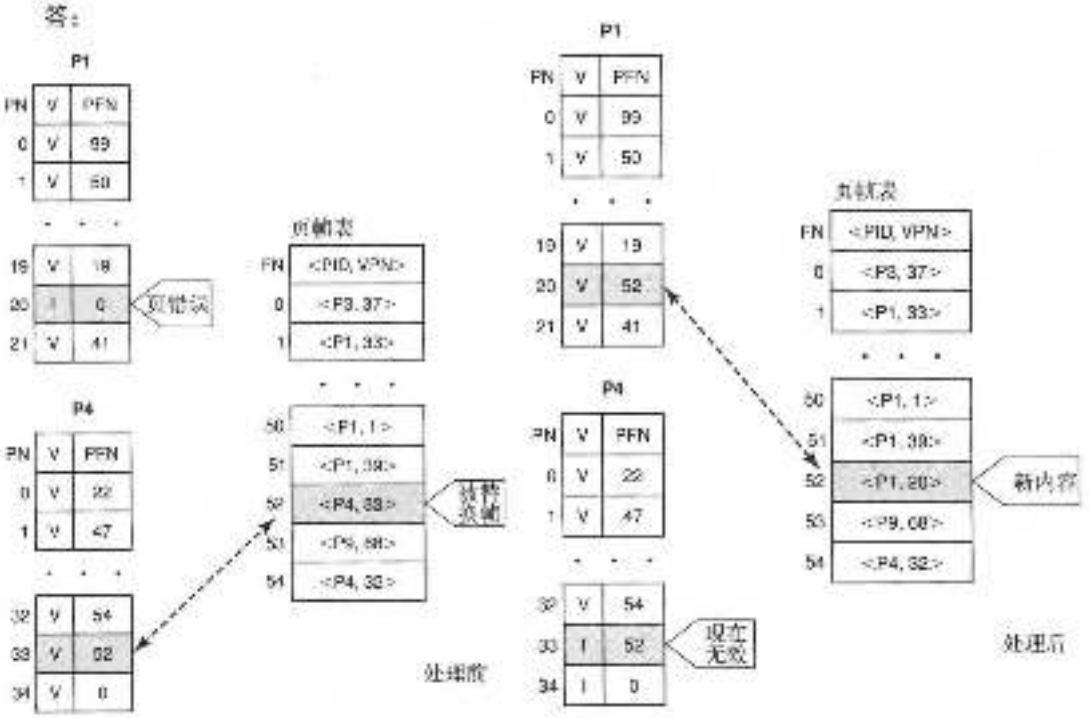
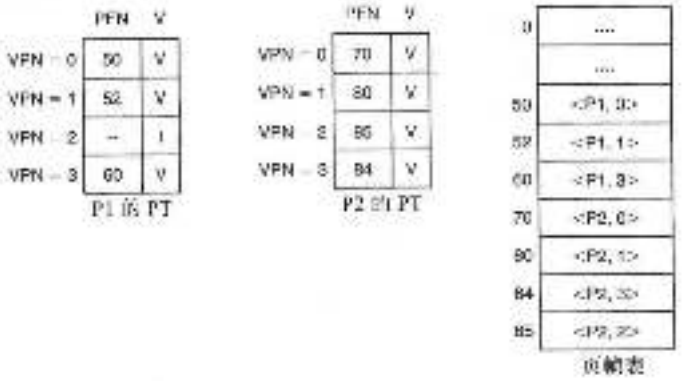


图 8-6 a）进程 P1 正在执行，并且虚页号为 20 的地方出现页错误

图 8-6 b）进程 1 的页错误处理完成

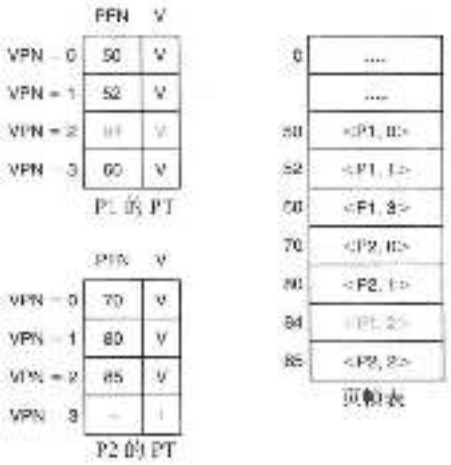
例 8-2 给定发生页错误前页管理器的数据结构，显示在 P1 进程中会在 VPN=2 的地方发生页错误。被替换页帧经页替换算法计算选定为 PFN=84。注意下图中只显示了页帧表中涉及内容。

发生页错误前的结构



处理发P1进程VPN=2处的页错误的数据结构的内容

答：



321
-
323

注意，页错误处理程序和其他用户进程一样也是一段代码。然而，不允许页错误处理程序本身发生页错误。操作系统会确保操作系统中特定的部分，如页错误处理程序，一直保存在内存中（即不会从物理内存中去除）。

例 8-3 当内存中没有空闲帧发生页错误时会执行下面 7 个操作中的 5 个。选择其中 5 项正确的操作并识别出其他 2 个不正确的操作。

a. 利用页请求发生有页错误的进程。

b. 利用缺页异常处理程序的磁盘映射从磁盘中将被替换帧装入缺失的页。

c. 选择一个被替换为页用作替换（和相关的被替换帧）。

d. 更新缺页异常处理程序的页表和页帧表来反映被替换帧的映射变化。

e. 如果是脏页，通过被替换进程的磁盘映射将被替换页写回磁盘。

f. 恢复页帧表识别出被替换进程并设置被替换页表中被替换页的有效位为无效。

g. 检测缺失的页是否在物理内存中存在。

答：

第 1 步：c

第 2 步：f

第 3 步：e

第 4 步：b

第 5 步：d（注意：只要被选步骤的相对顺序保持不变，这一步也许会出现在第 3 步或第 5 步。）

操作 a 和 g 不属于页错误处理过程。

8.2 进程调度器和内存管理器间交互

图 8-7 演示了 CPU 调度器和内存管理器之间的交互。在任何情况下，CPU 或者执行一项用户进程，或者执行操作系统下子系统中的一项操作，如 CPU 调度，或者进行内存管理。调度器、内存管理器和其他所有涉及数据结构的程序代码都会保存在内核内存空间中（见图 8-7）。用户进程保存在用户内存空间中。一旦 CPU 调度器分配一个进程，它会一直运行，直到下面的事件发生：

324

1）硬件计时器中断 CPU，可能会产生一个用于进程上下文切换的 CPU 调度器的回调（图中 1）。回调（见第 6 章）指从系统软件的较低层向较高层的一项功能调用。CPU 调度器会采取恰当的措施调度 CPU 的下一个进程。

2）进程发生页错误，产生一个内存管理器的回调（见图中的 2），用于处理前文中提及的页错误。

3）进程发出系统调用（例如请求 I/O 操作），导致另一个子系统（图中未标明）回调并采取相应措施。

虽然这 3 种事件处于操作系统的不同阶段，它们共享 PCB 的数据结构，这些综合起来就是目前进程的状态。

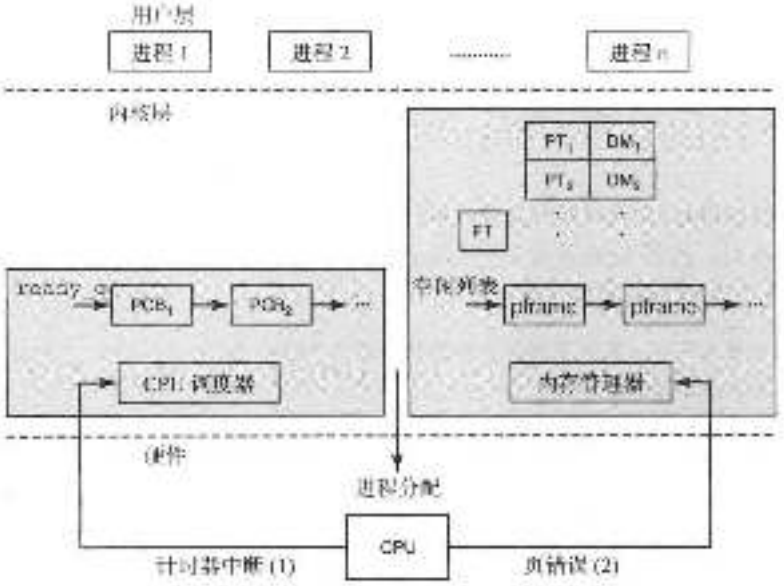


图 8-7 CPU 调度器和内存管理器建的交互。除了计时器中断，CPU 在探测目前分配的进程是否出现页错误的情况下也会回调操作系统

[325]

8.3 页替换策略

现在我们来讨论当发生页错误并且空闲页帧表为空时如何从物理内存中选出被替换页。如何从物理内存中选出一个被替换页的过程叫做页替换策略。我们首先来了解好的页替换策略会有哪些特性。

1）对于给定的连续的页访问，该策略能产生最少的页错误。这项特性也保证了操作系统中处理页错误耗费时间的降低。

2）理想情况下，一旦特定的页引入物理内存，该策略应该努力确保相同的页不再发生页错误。这项特性保证了页错误处理程序会顾及用户程序的访问模式。

在选择一个被替换页时，内存管理器有 2 个选择：

- **局部被替换选择** 思路是从缺页进程中抽取物理帧做替换，来满足缺页请求。具有一定的简洁性。例如，这种策略不需要页状表。然而，局部替换会导致较低的内存利用率。
- **全局被替换选择** 思路是从所有进程的帧中选择一个物理帧，而不需要是发生缺页异

常的进程中。具体选择哪个被替换进程和被替换页取决于具体的算法。由于是在全局范围内选择被替换页，这种策略会有不错的利用率。

我们用全局被替换选择来改善内存的利用情况。理想情况下，如果没有页错误，内存管理器永远不会被调用，并且处理器在绝大多数时间都会执行用户程序（除了上下文切换的情况之外）。所以，降低页错误率是任何内存管理器的目标。有以下两个原因使得内存管理器需要降低页错误率：（1）因为发生页错误时从下一级存储读取页会很耗时，这样会严重影响程序的性能；（2）宝贵的处理器循环不应该频繁用在类似页替换这样的开销上。

在余下的讨论中都默认采用全局页替换策略，尽管为了简单起见所举的例子主要集中在单进程的页调度行为上。内存管理器基于进程的页面调度进行被替换页的选择。所以，当我们提及一个页时，指的是虚页。一旦内存管理器确定一个虚页是被替换页，保存此虚页的物理帧将作为被替换选择。对于每种页替换策略，如果有必要我们会确定出所需的硬件支持、所需的数据结构、算法的细节和不同缺页错误数目预期的性能。

8.3.1 Belady 的 Min 算法

如果知道未来这些页被访问的情况，那么选择替换那些未来最长时间内不会被访问的页是最好的选择。这种替换策略不可行，因为内存管理器并不知道未来某个进程的访问情况。然而，在 1966 年，Laszlo Belady 提出了最优替换算法（optimal replacement algorithm），命名为 Belady 的 Min 算法。这个算法后来成为评价任何页替换算法性能的基准。 326

8.3.2 随机替换

最简单的策略是页面随机替换。乍一看，这也许不是一个很好的方法。这种策略的优点是内存管理器不需要任何硬件支持，也不需要保存当前页的细节信息（如时间戳或访问顺序）。在对未来未知的情况下，了解随机策略对任意顺序访问的性能分析是很有价值的。正如 Belady 的 Min 算法是页替换策略性能的上限一样，随机替换可以作为替换策略性能的下限。换句话说，如果一个页替换策略要求有硬件支持或者需要内存管理器维护细节信息，它应该比随机策略表现得要好，否则就没必要增加额外开销。实际应用中，内存管理器在没有足够多的细节信息做决定时都会默认使用随机替换算法（见 8.3.4 节）。

8.3.3 先进先出策略

这是最简单的页替换策略之一。先进先出（FIFO，First In First Out）算法如下：

- 当一个页装入物理内存时添加一个时间戳。
- 如果有页需要替换，选择时间戳最久的页面作为被替换页。

有趣的是，对于此策略我们不需要任何硬件支持。我们稍后将看到，内存管理器会用它的数据结构记录页存入物理内存的顺序。

我们首先来理解内存管理器需要的数据结构。内存管理器用队列记录存入物理内存中页面的存入顺序来模拟时间戳。我们用一个带头指针和尾指针的循环队列来进行模拟（见图 8-8，头指针和尾指针均初始化为 0）。我们向队尾插入元素。所以驻留时间最长的页面是在队首的页面。此外，内存管理器还设置队满标志（full flag，初始化为 false）来指示队列是否满。队列中所表示的元素是当前内存中正在使用的物理帧。所以，我们要将队列的长度设置为实际物理内存能够容纳的总帧数目。循环队列中的每个元素都和一个特定的物理帧对应。

初始时队列为空（队满标志位为 false），表示所有的物理帧都没有被使用。随着内存管理器按需分页的进行，会分配物理页帧来满足页错误需求（每做一次分配就增加一个队尾元素）。当没有额外的页帧进行分配的时候（头和尾指针相等）就认为队列满了（队满标志位为 true）。
327 这种情况下如果出现页错误会进行页替换。内存管理器会替换队首的页，因为它的驻留时间最久。需要注意的是循环队列同时起到了空闲页面表和页帧表的作用。

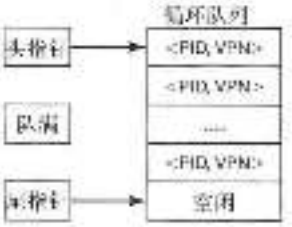


图 8-8 先进先出页替换算法的循环队列图。尾指针指向保存第一个空闲物理页帧的序号。每个队列项都和一个物理页帧相对应。同时保存页帧的 <PID，VPN>（进程号和虚页号）。队首保存驻留时间最长页。

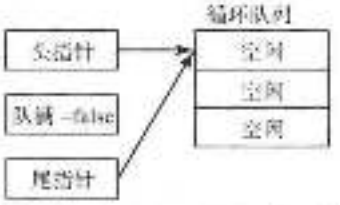
例 8-4 假设进程访问一系列的页：

访问编号：	1	2	3	4	5	6	7	8	9	10	11	12	13
虚页号：	9	0	3	4	0	5	0	6	4	5	0	5	4

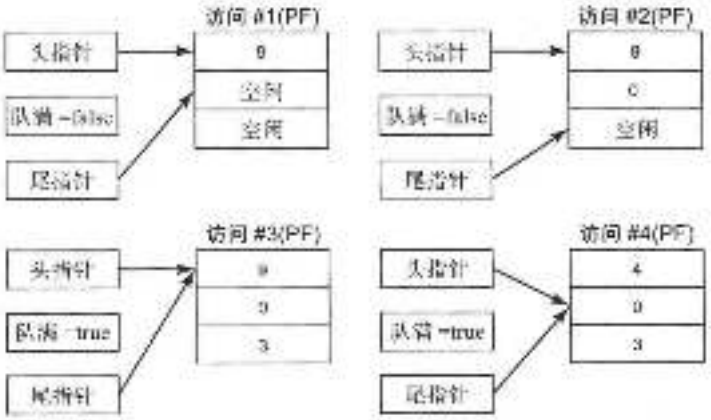
假设有 3 个物理页帧。用类似图 8-8 的循环队列来显示每个页面访问后队列状态。

答：

初始时，循环队列如下：



当出现页错误时，首先从队尾指针插入新页。类似地，被替换页也是队首指针所指的当页面，因为对于 FIFO 策略，队首的页总是指向最先进来的页面。一旦选中被替换页，队首指针将指向下一个 FIFO 候选者。当队为满时，队首指针和队尾指针都需要移动来处理一个缺页异常。下面的数据结构类图表示前 6 个访问之后的队列的状态（PF 代表页错误；HIT 代表访问命中，即没有页错误）：





给定的序列模拟了 FIFO 的行为：第 6 个访问替换掉了页 0，因为它是当前在内存中最早的页面。请注意在第 8 次访问时页 5 命中。然后，页 0 接下来很快会被访问到（访问 7）。

通过观察前面例子的内存访问顺序，我们可以知道页 0 是常被访问。一个有效的页替换策略不应该试图替换页 0。不幸的是，我们不能提前知道哪些页面会最常被使用。让我们看看有没有其他方法比 FIFO 表现得好。

8.3.4 最近最少使用策略

即使在真实的生活中，我们也常用过去的经验来预测未来。所以，尽管我们不知道进程在未来会访问哪些内存，但我们可以分析进程之前访问的内存。我们首先看看如何将其应用在页替换策略上。最近最少使用策略（LRU）假设页如果在过去很久都没有被访问，有很大的概率它在未来也不会被访问。所以，LRU 策略选中的被替换页是最长时间未被使用的页。

328 - 329

让我们首先来了解 LRU 策略的硬件支持。硬件需要跟踪 CPU 的每次内存访问。图 8-9 演示了栈的数据结构。每次访问时 CPU 都会把最常被访问的页放在栈顶；如果页在栈之外的其他地方存在，CPU 也会把它移除。所以，栈底的页是最不经常被使用的页面，而且会被替换用来处理页错误。如果我们想跟踪所有页帧的访问，那么栈的大小应该和实际物理页帧总数一样大。

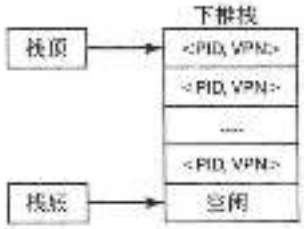


图 8-9 LRU 替换策略的下推栈。最常使用的页在栈顶。最不常被使用的页在栈底

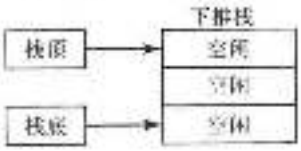
接下来，我们来探讨 LRU 策略的数据结构。内存管理器应用类似图 8-9 中的硬件栈来选择栈底的页作为被替换页。当然，软件从栈底读取数据需要一些指令集上的支持。除了页表，硬件栈也可用于虚页到物理页的转换。

注意，内存管理器需要维护额外的数据结构，例如空闲列表和帧表，来处理页错误。

例 8-5 我们在这里讨论的页访问顺序和 FIFO 例子中进程访问内页顺序是一样的：

访问编号：	1	2	3	4	5	6	7	8	9	10	11	12	13
虚页号：	9	0	3	4	0	5	0	6	4	5	0	5	4

假设有 3 个物理页帧，初始化栈如下所示：



330

如下栈图显示了前 6 次访问后栈的状态（PF 代表页错误；HIT 代表访问命中，意味着没有页错误）：



比较例 8-4 和 8-5。在前 6 次访问中都经历了 5 次页缺失异常。不幸的是，我们不能改善这一点，因为这些缺失的页（9、0、3、4、5）都不在物理内存中，它们确实是页缺失，任何策略都不能避免这一点。然而，值得注意的是 LRU 策略在第 6 次访问时替换了页面 3（而不是 FIFO 例子中的页 0）。所以，在访问 7 的 LRU 策略会有命中。换句话说，LRU 能够避免在 FIFO 策略中遇见的异常情况。

近似 LRU 的实现 #1：一个简单的硬件栈 LRU 策略虽然在概念上很吸引人，但从具体应用的观点上来看并不容易实现，主要原因有如下几点：

1）因为物理帧数目很多，栈需要很多项。如果物理内存是 4GB 并且页大小为 8KB，那么栈的大小则有 0.5MB。在流水线处理器的数据通路中增加如此大的硬件结构会极大地增加处理器的时钟周期。由于这一原因，在处理器的数据通路上增加一个如此大的硬件栈是不实际的。 331

2）为了将当前的访问页存在栈顶，每次访问硬件都要对栈进行修改。这种开销很大的操作降低了处理器的速度。

因为这些原因，真正的 LRU 策略不适合实际应用。还有一个更重要的原因，在某些应用中 LRU 会对性能产生很不利的影响。例如，假设一个程序在顺序访问 N+1 个页之后循环地访问这些页面。如果内存管理器为此任务分配的空闲页帧池大小为 N，这样如果应用 LRU 策略每次访问都会有页错误发生。这个例子表现出来的病态性具有很强的现实意义，因为科学计算用的数组规模通常都非常大。

可行且对性能影响较小的办法是采用近似的 LRU 方式。栈的大小可以设定为一个较小的数字（例如 16），而不是等于物理内存中实际帧的数目。这样，栈会保存处理器调用最近的 16 条历史访问记录（更久的访问会从栈底弹出）。算法会随机挑选一页作为被替换页，但这不是在硬件栈中发生的。典型情况下，算法会保护最近被访问的 N 个页不被替换，而 N 是硬件栈的大小。

实际上，一些模拟运行情况研究发现，真正的 LRU 可能比近似的 LRU 算法还要糟糕。这是因为除了 Belady 的 Min 算法外其他的算法都是对页的访问进行猜测，所以很容易失败。

从这个意义上来说，一个不要求任何软件和硬件支持的单纯的随机替换算法（见 8.3.2 节）用于完成某些工作确实表现得较好。

近似 LRU 的实现 #2：每个页帧增加访问位 从实现一个高速的流水线 CPU 的角度来看，追踪每次内存访问是不现实的。所以我们要从其他方法去寻找近似的 LRU 算法。

一种可行策略是从页的层次去记录访问而不是每次单独访问。思路是给每个页帧增加一个访问位。当 CPU 访问这一页的任何位置时硬件上都会对这一位进行设置；而在软件上对它进行读取和复位。硬件通过页表协调对物理内存的访问。所以，我们在页表中会有引用位。

让我们将注意力转移到选择被替换页上。下面是应用访问位的算法：

1）内存管理器为每个页帧维护一个名为访问计数器的位向量。

2）内存管理器定期读取所有页帧的访问位，并把它们转储在每个帧相应访问计数器的最高有效位（msb，most significant bit）中。计数器通过右移将引用位装入各自的 msb 位置。图 8-10 显示了这一过程。在每次读取访问位之后，内存管理器会清除访问位。每隔单位时间就会重复该过程。所以每个计数器会维护着最近 n 个时间间隔内访问的快照（图 8-10 里的 n=32）。

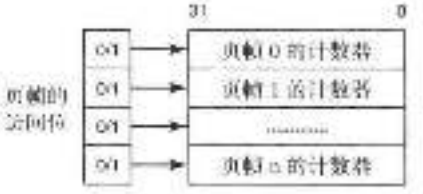


图 8-10 页帧访问计数器：每一页的访问位是 0 或 1

3）访问计数器绝对值最大的页是最常被访问的页。而访问计数器值绝对值最小的页是最不经常被访问的页，也是选作被替换的页。

分页守护进程是内存管理器的一项，它每隔一定时间间隔（由时间段决定）唤醒来执行前面的算法步骤。

8.3.5 第二次机会页替换算法

这个算法对 FIFO 进行了扩充，在 FIFO 策略中加入了访问位的思想。正如其名，这一算法会给每个页一次不选为被替换页的机会。基本的想法是利用硬件上的访问位作为指示给每个页面第二次留在内存中的机会。算法步骤如下：

1）初始时，操作系统会清空所有页的访问位。随着程序的执行，硬件为程序的每次页引用设置访问位。

2）如果页面要被替换，内存管理器以 FIFO 的方式选择被替换页。

3）如果被替换页的访问位被设置，管理器会清除访问位，并给它一个新的到达时间，并重复步骤 1。换句话说，这个页被移进 FIFO 队列的队尾。

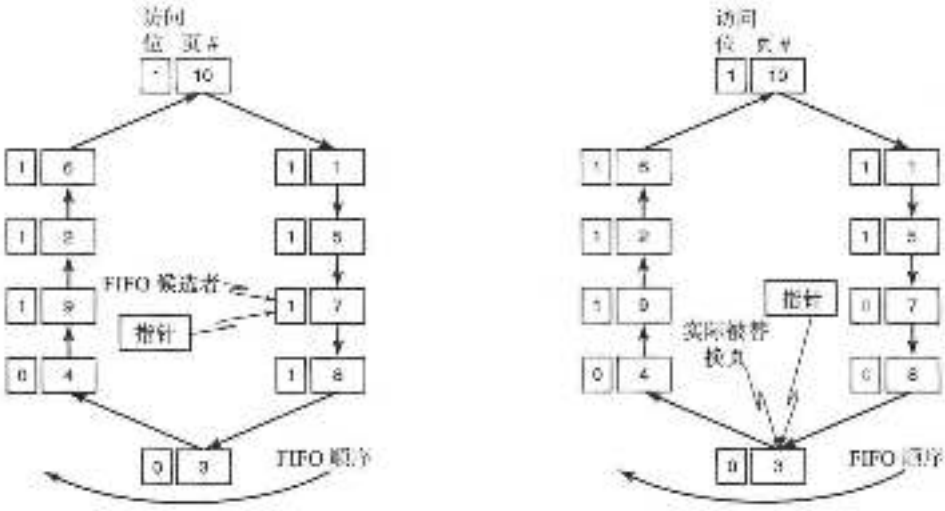
4）被替换页是 FIFO 队列中队首访问位没有被设置的页。

当然，如果所有的页面都进行了访问位设置，算法会退化为简单的 FIFO 算法。

一个可视化并实现该算法的简单方式是将页设想成一个循环队列，如图 8-11a 所示。指针指向 FIFO 候选者。当要选一个被替换页时，指针会前进直到发现一个访问位没有被设置的页。在它向前移动到最终的被替换页前，内存管理器会清除它遍到的页的访问位。如图 8-11a 所示，第一个选作 FIFO 候选者的是页 7。然而，因为它的访问位有设置，指针会前进直到发现页 3 并把页 3 作为被替换页。因为它的引用位没有被设置（FIFO 队列中第一个搜索到的未被设置的页）。算法在遍历 FIFO 队列的过程中会清除页 7 和页 8 的访问位。注意算法在遍历的过程中没有改变其他未遍到页的访问位。可以发现指针扫描的过程很像时钟的指

332
333

针围绕圆圈在走，所以这个算法也叫做时钟算法（clock algorithm）。将页3作为被替换页之后，指针会前进直到选中下一个FIFO候选者（图8-11（b）中的页4）。



a）第二次机会替换算法——内存管理器在算法开始时保存指向FIFO候选者的指针。注意有的页帧的访问位被设置而有的访问位被清空。即使引用位被设置为1的页表示从上次内存管理器进行扫描后被程序访问过

b）第二次机会替换算法——算法会遍历访问位为1的域（遍历过程中会清除该位），直到发现一个帧的访问位没有被设置，并将它作为被替换页

图 8-11

例8-6 假设我们只有3个物理帧，并且我们应用第二次机会页替换算法。请展示按下列页访问序列执行程序帧中所存放的虚页号：

访问编号：	1	2	3	4	5	6	7	8	9	10
虚页号：	0	1	2	3	1	4	5	3	6	4

答：

接下来的图显示每次访问处理后页帧和相应引用位的状态。第一项永远是FIFO策略选中为被替换页。注意当页被引入页帧时访问位会进行设置。

为了理解一个特定访问的被替换页的选择，可查看前一次访问之后页面的状态。

访问1－3：没有替换。

访问4：页0选为被替换页（因为所有页的访问位都有设置，所以FIFO候选者是被替换页）。

访问5：没有替换（页1的访问位被设置）。

访问6：页2是被替换页（页1，FIFO候选者，因为访问位还有一次机会）。

访问7：页1是被替换页（页3，FIFO候选者，因为访问位还有一次机会）。

访问8：没有替换（页3的访问位被设置）。

访问9：页4是被替换页（因为所有页的访问位都有设置，所以FIFO候选者是被替换页）。

访问10：页3是被替换页（也是FIFO候选者；它的访问位为空）。

334
-
335

页帧	访问	注解

访问 1

0	1

访问 2

0	1
1	1

访问 3

0	1
1	1
2	1

访问 4

1	0
2	0
3	1

访问 5

1	1
2	0
3	1

访问 6

3	1
1	0
4	1

访问 7

4	1
3	0
5	1

访问 8

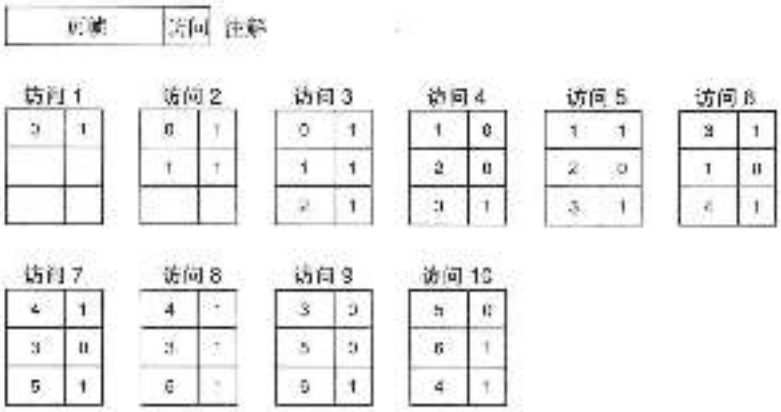
4	1
3	1
6	1

访问 9

3	0
5	0
5	1

访问 10

5	0
6	1
4	1



8.3.6 页替换算法回顾

表 8-1 对页替换算法进行了总结，并说明了相应的硬件支持。结果发现使用访问位的近似 LRU 算法在减少页错误率上确实表现得很好，表现得几乎和真正的 LRU 一样好。这个例子很好地说明了创造力可帮助我们从确切的解决方案中得到最大收益。

表 8-1 每种页替换算法的比较

页替换算法	需要的硬件支持	需要保存的信息	备注
Belady 的 MIN	Oracle	无	基本上会有最好的性能；不靠硬件实现；是性能比较的上界
随机替换	无	无	最简单的策略；可以用作性能对比的下界
FIFO	无	每个虚拟页引入物理内存的时间	可能会有异常出现；通常比随机替换表现出的性能还要差
真正的 LRU	下推栈	保存指向 LRU 栈底的指针	预期性能接近最优；因为空间和时间复杂性无法用硬件实现；最坏情况下甚至低于 FIFO
近似 LRU#1	小的硬件栈	保存指向 LRU 栈底的指针	预期性能接近最优；最坏情况下与 FIFO 性能相似甚至低于 FIFO
近似 LRU#2	每个页帧增加访问位	每个页帧访问计数器	预期性能接近最优；或极硬件复杂性最坏情况下与 FIFO 性能相似甚至低于 FIFO
第二次机会替换	每个页帧增加访问位	每个虚拟页引入物理内存的时间	预期性能优于 FIFO；内存管理和 LRU 策略相比较为简单

8.4 优化内存管理

在前面的章节，我们展示了初级的虚拟内存分页管理技术。在本节，我们来讨论一些提高系统性能的内存管理器的应用策略。值得注意的是，优化策略并不针对某项特定的页替换算法。它们可以应用于前面章节讨论的基本的页替换策略。

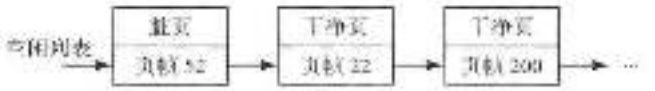
8.4.1 空闲页帧池

当出现页错误之后才挑选被替换页并不是一个很好的方法，内存管理器总是准备好一定

最少数量的页帧随时用于处理页错误。分页守护进程每隔一段时间间隔会被唤醒来查看空闲页帧表（图 8-3）中的空闲帧是否低于最小门限值。如果低于最小门限值，它会执行页替换算法来腾出更多的空闲帧以满足最低门限要求。有时在空闲页帧表中空闲帧的数目会大于等于门限值。当一个进程终止时，所有的页帧都会出现在空闲列表中，用来解决此问题。

处理过程中的 I/O 重叠 在 8.1.4 节，我们指出内存管理将一个页帧作为被替换页前，如果要处理的是一个脏页，内存管理器要把当前页面的内容写回磁盘。然而，因为内存管理器只是简单地将页帧加入空闲页帧表，并不会同时进行保存操作。我们将在后面的章节看到高速 I/O（如磁盘）和 CPU 活动同时执行。所以，内存管理器要在把帧加入空闲页帧表之前将脏的被替换页进行 I/O 写回操作。由于这个原因，内存管理器也许会跳过空闲页帧表中的脏页来处理页错误（见图 8-12a）。这种策略有助于减轻由页错误造成的 IO 写等待延时问题。

336
?
337



a) 空闲页帧表——出现页错误时，内存管理器可能会选择页帧 22 而不是页帧 52 作为被替换页。因为页帧 22 是干净页，而页帧 52 是脏页



b) 为空闲页帧表中的页帧维护的反向映射 <PID，VPN>

图 8-12

页表的反向映射 为了满足最低门限要求，页面守护进程有可能会取走目前正在运行的进程的页帧。当然，这些缺失页面的进程有可能因为页面被抽走运行时出现缺页异常。事实证明，我们可以向每个空闲列表的节点增加一个额外的位来缓解这一情况。如果内存管理器没有把这一物理页帧分配给其他进程，那么我们可以把它从空闲页帧表中取回并分配给缺页进程。为了实现这一优化，我们在空闲页帧表中每项里增加了反向映射（类似页帧表），显示它最后保存的虚页（见图 8-12b）。

当遇到页错误时，内存管理器会把缺失进程的 <PID，VPN> 和每个空闲页帧表进行匹配。如果匹配成功，内存管理器会用找到的页帧为发生页错误的缺页异常进程重建初始时页表中的映射。这项优化策略排除了通过 I/O 读取从磁盘引入缺失页的需要。

这种加强策有意思的地方是，如果在第二次机会替换算法之后就应用，相当于给了页面留在系统中的第二次机会。

8.4.2 颠簸

一个我们经常遇到的涉及计算机系统性能的术语是颠簸（thrashing），颠簸用来表示系统没有把有用的工作完成。例如，假设程序的多道运行程度（在第 6 章被定义为同时在内存中运行并占用 CPU 计算资源的进程数目）很高，但我们仍观察到很低的处理器利用率。我们也许会试图增加多道运行程度来让处理器更忙。表面上看，这也许是个好主意，但我们略向深层次挖掘一下。

很有可能目前所有执行的程序都I/O受限（意思是说它们花费在I/O上的时间比在CPU上运行的时间要多）。在这种情况下增加多道运行程度也许是个好主意。然后，也有可能程序是CPU受限的。乍一看，处理类似的CPU受限的工作时处理器利用率降低很奇怪。简单的回答是有太多的分页活动。我们来阐述这一观点。内存管理器需要为每个进程分配足够的空间才能完成相应工作。换句话说，会有更多的页错误。所以，如果内存中同时存在太多的进程（即程序多道程度很高），那么进程很有可能会在物理内存中不停分页。这样，所有的进程都没有前进。在这种情况下，增加程序的多道程度并不是个好方法。实际上，我们应该降低多道程度。图8-13显示了不同多道程度下CPU利用率的预期表现。在某个点之后CPU利用率会迅速降低。

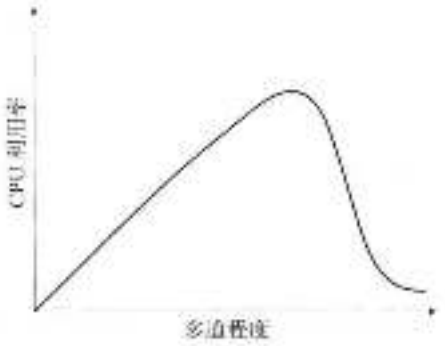


图8-13 CPU颠簸现象。当超过一定多道程度之后CPU利用率会急剧下降

当处理器花费更多的时间用于分页而不是计算时就会出现颠簸现象。分页是操作系统中进程的一种形式I/O行为。过多的分页可能会让一个原本为计算受限的进程变成一个I/O受限的进程。我们从这些讨论中得到的一个很重要的教训是，CPU调度应该考虑进程的内存利用情况。CPU的调度策略只基于处理器的利用率是不对的。幸运的是，这种情况可以通过调整CPU的调度系统和操作系统的内存管理来进行纠正。

让我们来讨论如何控制颠簸。当然，我们可以把整个程序装进内存，但这并不是一种有效利用资源的方法。这一方法在于确保每个进程不会有频繁的页错误，每个进程会分配足够的页帧。我们可以应用局部性原理来帮忙。一个进程也许会有范围很广的内存印记。然而，如果我们看不同时间的访问窗口，我们会发现进程的访问只集中于整个内存印记中的一小部分。这就是局部性原则。当然，程序的访问会随着时间变化，如图8-14所示。然而，这种变化也是渐进的而不是急速变化的。例如，在时间t1程序访问的页面为{p1，p2}；在时间t2访问的页面是{p2，p3}。

338
|
339

我们不想让读者认为程序访问的位置总是连续的页。例如，在时间t7，程序访问的页面是{p1，p4}。值得注意的是，一定时间间隔内程序的访问活动被局限于小部分页面（见例8-7）。

应用这项原则来减少页错误也很简单。如果当前程序访问的位点在内存中，相关的进程在位点改变前通常不会出现页错误。例如，如果程序访问的页面在时间t1和t2保持不变，并且假设进程p1和p2在物理内存中，那么在时间t1和t2间程序通常不会经历页错误。

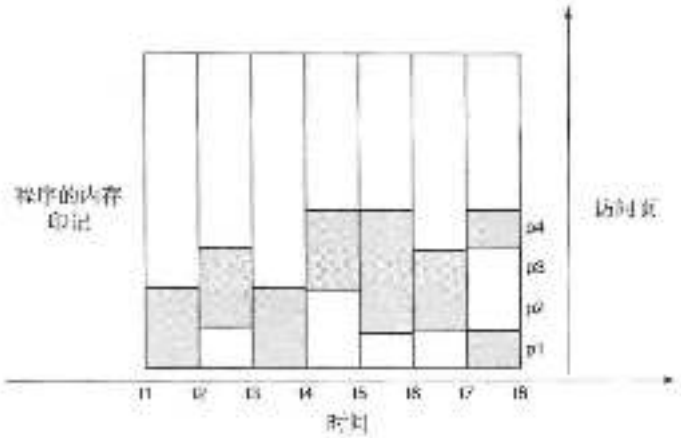


图 8-14 程序访问位点随时间的变化：程序在不同时间 t1、t2 等的访问页见图；假设程序访问的位点在时间间隔，如 t1 ~ t2、t2 ~ t3 等，保持不变。

8.4.3 工作集

为了判定程序活动的位点情况，我们定义并使用工作集的概念。工作集（working set）是定义程序活动位点的集合。当然，工作集并不是保持不变的，因为程序活动的位点会随时间改变。例如，参考图 8-14，

Working $set_{t1,t2}$ = {p1, p2}
Working $set_{t2,t3}$ = {p2, p3}
Working $set_{t3,t4}$ = {p1, p2}
Working $set_{t4,t5}$ = {p3, p4}
Working $set_{t5,t6}$ = {p2, p3, p4}

340

工作集的大小（WSS，Working Set Size）表示进程在一个时间窗口里访问的特定页的数目。例如，在时间间隔 t1 ~ t2 中 WSS 是 2，而在时间间隔 t5 ~ t6 中 WSS 是 3。

系统内存的压力是当前所有会竞争资源的进程的 WSS 的总和。

$$\text{总内存压力} = \sum_{i=1}^{n} \text{WSS}_i$$

例 8-7 在时间间隔 t1 ~ t2 中，3 个进程 P1、P2 和 P3 的访问虚拟页次序如下：

P1：0，10，1，0，1，2，10，2，1，1，0

P2：0，100，101，102，103，0，101，102，104

P3：0，1，2，3，4，5，0，1，2，3，4，5

a. 对于给定的这 3 个进程，在此时间间隔内的工作集是什么？

b. 在此时间间隔三系统的总内存压力是多少？

答：

P1 的工作集 = {0,1,2,10}

P2 的工作集 = {0,100,101,102,103,104}

P3 的工作集 = {0,1,2,3,4,5}

b. P1 的工作集大小 WSS_{p1}=4

P2 的工作集大小 WSS_{p2}=6

P3 的工作集大小 $WSS_{P3}=6$

总内存压力 = 所有进程的工作集数目之和

= 4+6+6

= 16 个物理页帧

8.4.4 颠簸控制

1）如果系统表现出来的总内存压力比可用内存的总量大，内存管理器会降低程序的多道程度。如果总内存压力比可用物理内存的总量小，内存管理器会增加程序的多道程度。 341

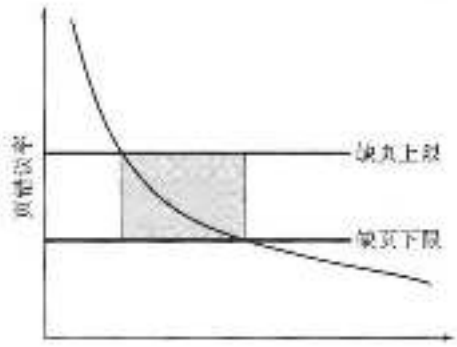


图 8-15 有颠簸控制时的页错误率。图中阴影部分是表现出的最佳系统性能。当页错误率低于缺页下限时可以考虑增加程序的多道运行程度；当页错误率高于缺页上限时可以考虑降低多道程度防止出现颠簸现象

一种测量进程 WSS 的近似方法是对相关的物理帧使用访问位。每隔一定时间间隔 Δ，守护进程会被唤醒并对每个进程物理帧的访问位进行采样。守护进程会记录相应访问位设置过的页号；它之后会把这些页的访问位清零。这种记录页号的方法可让内存管理器获取在任何时间间隔 t – t+ Δ内的工作集和给定进程的 WSS。

2）另一个控制颠簸的方法是将观察到的页错误率作为衡量颠簸的指标。内存管理器会设置两个限制，页错误的下限和上限（见图 8-15）。当页错误率超过缺页上限时意味着有过多的分页活动。在这种情况下，内存管理器会降低多道程度，这样可以有效地增加每个进程的可用物理帧数。另一方面，当页错误率低于页错误页下限时意味着内存管理器可以增加程序的多道程度，这样可以有效地减少每个进程的可用页帧数。

图 8-15 中阴影部分显示出推荐的内存管理器最佳情况下所表现出的性能。当页错误率比缺页上限高时分页守护进程会增加空闲物理页帧池，当低于缺页下限时如果有需要会增加程序的多道程度。 342

8.5 其他考虑

操作系统采用一些其他的措施来减少页错误率。例如，当内存管理器换出一个进程时（为了减少程序的多道程度），它会保存当前进程的工作集。当内存管理器换入一个进程时会把相关的工作集引入进来。这种优化措施称为预约式页面调度，在进程启动时减少了中断次数。

系统的 I/O 活动和 CPU 活动类似，可以同时进行。这导致系统的内存和 I/O 子系统之间

会发行有趣的交互。例如，I/O 子系统可能会进行初始化操作将给定的物理帧存入磁盘。同时，分页守护进程也可能会由于页错误选择同样的物理帧作为被替换页。就像 CPU 调度器和内存管理器相互配合工作一样，I/O 和内存子系统间也会配合工作。通常，为了防止内存管理器将要交换的页面作为被替换页，I/O 子系统会锁住物理内存中的页帧，并保存一定时间。页表作为 I/O 子系统和内存管理器之间的桥梁用来记录信息，例如需要锁住哪些物理内存中的页帧。我们将在第 10 章中讨论 I/O 的细节信息。

8.6 旁路转换缓存

迄今为止需要明确一点：页错误对于系统性能有很坏的影响。内存管理器很艰难地试图去避免缺页。为了各进程独立执行，上下文切换时间（从一个进程切换到另一个）大约为几十条指令执行时间；页错误处理时间（不算磁盘 I/O）也约有几十条指令执行时间。换句话说，花费在磁盘 I/O 上的时间范围是毫秒级的，对于一个 GHz 级别的处理器这相当于执行了百万行的程序指令。

然而，尽管我们利用各种优化措施减少了页错误的数目，每次内存访问会有两次内存中的实际操作：一次用于地址转换，另一次用于实际指令或数据的读取。这很不理想。幸运的是，运用一些工程技巧，我们可以将它们合成一个。分页的概念消除了用户对程序使用连续内存的想法。然而，实际情况是，一个页面中的内容在内存里是连续存放的。所以，如果我们对一个页面做地址转换，那么这个地址转换适用于页面中所有的内容。这表明可以向 CPU 中增加一些硬件来记录地址转换信息。然而，我们知道程序总是有范围很广的内存印迹。而我们之前提到过的局部性原理现在又发挥作用了。如图 8-14 所示，我们对于一个特定程序一次只需记住一部分翻译信息，忽略它的整个内存印记。这就是旁路转换缓存（TLB，Translation Look-aside Buffer）的答题思路。设计一个小的 CPU 可以缓存最近翻译信息的硬件（见图 8-16）。每个页面的地址转换至少要做一次。所以，当处理器启动时表中所有的项都是无效的。这就是每个
[343] 项中设置有效位的原因。PFN 字段提供那一项 VPN 对应的物理页帧号。

用户态/内核态	VPN	PFN	有效/无效
U	0	122	V
U	XX	XX	I
U	10	152	V
U	11	170	V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

图 8-16 旁路转换缓存（TLB）保存最近处理器用到的地址转换

请注意 TLB 是如何划分为两部分的。一部分用于保存和用户地址空间相关的翻译，另一部分应用于内核空间。对于上下文切换，操作系统简单地使所有的用户空间翻译无效，调度新的进程，并建立那一部分的表。内核空间转换独立于处理器正在运行的用户进程，是有效的。为了更好地进行 TLB 管理，指令集提供专门的 TLB（purge TLB），一种可在内核模式中执行的特权指令。

8.6.1 TLB 的地址转换

图 8-17a 和图 8-17b 显示了 TLB 中的 CPU 地址转换。硬件会首先检查对于 CPU 产生的地址转换在 TLB 中是否有效。我们将 TLB 中一次成功的查询称为命中。如果没有命中则称为缺失。如果命中，TLB 中的 PFN 会帮助产生物理地址，这样避免了去内存中做地址转换。如果缺失，内存中的页表会提供 PFN，硬件会把翻译的结果保存在 TLB 中为将来访问相同的页做准备。

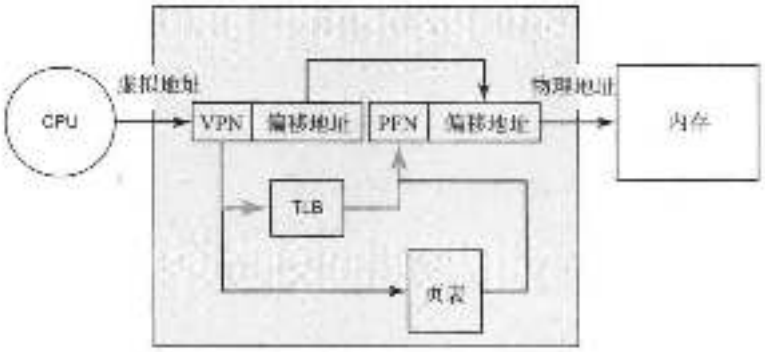


图 8-17 a) 地址转换（TLB 命中）。TLB 中保存当前 VPN 到 PFN 的映射，避免了从页表中查询的需要

344

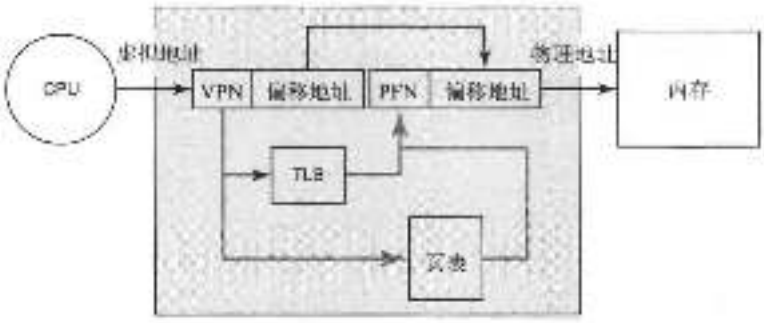


图 8-17 b) 地址转换（TLB 缺失）。VPN 向 PFN 的映射目前不在 TLB 表中，需要在页表中进行查询

当然，很有可能地址转换并不是完全由软件实现的。你也许会好奇这怎么可能。通常当没有在 TLB 表中发现需要的翻译时硬件会发出 TLB 缺失异常。如果这一页没有出现在内存中则会升级为真正的 TLB 错误。任何情况下，一旦完成了 TLB 缺失页的处理，操作系统就会把地址转换信息保存在 TLB 中，便于以后操作。例如 MIPS 和 DEC Alpha 等体系结构处理这样的 TLB 缺失是完全在软件层面实现的。通常，这些体系结构的 ISA 对于修改 TLB 项都有特殊的指令集。我们习惯将它们称为软件管理的 TLB。

TLB 是一种特殊的内存，和我们以前遇到的任何硬件都不同。TLB 是一个保存 VPN 和 PFN 映射的散列表。对给定的 VPN，硬件需要对每个表进行查找进行匹配。我们将这种硬件设施作为内容可寻址存储器（CAM，Content Addressable Memory）或关联存储器（Associative Memory）。TLB 的硬件实现细节取决于它的组织结构。

TLB 是缓存（caching）的一种特殊的表现形式，关于缓存，我们将在第 9 章多级存储体系中进行详细表述。TLB 满足缓存的定义，即任何子系统用一个小表将最近用到的表项保存起来。从这一点来说，这一概念更像在第 2 章中介绍的类似工具箱或工具托盘的概念。下面举一些具体的例子：

a. 处理器高速缓存（processor cache）指在处理器设计中为了保存最近访问的内存地址而
345 设计的硬件设备。

b. 在处理器设计中将最近使用的地址转换保存在 TLB 中。

c. 在磁盘控制器设计中将最近访问的磁盘块放进内存里（见第 10 章）。

d. 在文件系统设计中，将最近访问的文件在磁盘上的物理位置信息保存在内存的数据结构中（见第 11 章）。

e. 在文件系统设计中用内存中的软件缓存保存最近磁盘访问过的文件（见第 11 章）。

f. 在 Web 浏览器设计中将最近访问的网页保留在本地。

换句话说，缓存是将对保存项目的一小部分而不是永久性地保留全部信息的一种系统的概念。我们将在第 9 章讨论更多关于处理器缓存实现的信息，也会谈到多级存储体系。

8.7 内存管理的高级话题

我们提到过，内存管理器的数据结构包括进程的页表。让我们来算一下，假设每个页面有 40 位的字节可寻址虚拟地址和 8KB 大小的页面，对于每个进程的页表我们需要 2^{27} 个页面项。这对每个进程形成了一个高达 128 兆的页表项。机器每个物理内存的大小也许不如一个进程的页表大。

我们将展示一些初步的解决物理内存中管理页表大小的思想。操作系统中一个更高级的课程将对这一话题进行更深入的讨论。

8.7.1 多级页表

基本思想是将一个单级页表划分为多级页表。为了进行更具体的讨论，可以想象一个有 32 位虚拟地址、4KB 大小的页面。页表有 2^{20} 项。让我们考虑一个 2 级的页表，如图 8-18 所示。虚拟地址的 VPN 有两部分。VPN1 会选择第一级页表 2^{10} 个页表项中的一项。还会有一个第二级页表（用 VPN2 来索引）来表示第一级页表中特定的项。这样就有 2^{10}（1024）个二级页表，每个二级页表有 2^{10} 个项。二级页表中存有对应虚拟地址 VPN 的 PFN。

这种页表需要多大的空间？如果是单级页表，我们需要 2^{20} 项（1M 项）。如果是 2 级页表，我们需要 2^{10} 个一级页表项和 2^{10} 个二级页表，每个二级页表有 2^{10} 个项。那么二级页表
346 的空间大小是 1K 项（一级页表的大小），总空间大小超过了单级页表的结构。[6]

我们从这种二级结构中得到了什么？这种结构使得对需要一直保存在物理内存中的内核数据结构的需求急剧减少。让我们来看看为什么。每个进程的一级页表有 1K 个项。这对于物理内存中每个进程的数据结构大小来说很合理。第二级页表没有必要装入物理内存。内存管理器将它们保存在虚拟内存中，并且在程序局部性原理的基础上对二级页表进行分页。

现代操作系统提供适合 64 位处理器体系结构的多级页表（即多于 2 级）。不幸的是，如果有更多级的页表，也就意味着对物理内存会有更多次的潜在访问。尽管有这样的事实，幸

[6] 注意二级页表的页表项的大小和单级页表结构的不一样。然而，为了使讨论简单，我们不考虑这些细节。为了更好地阐述这一点请参考练习 11。

运的是，TLB使接下来对相同页面的内存访问的翻译不耗费资源。

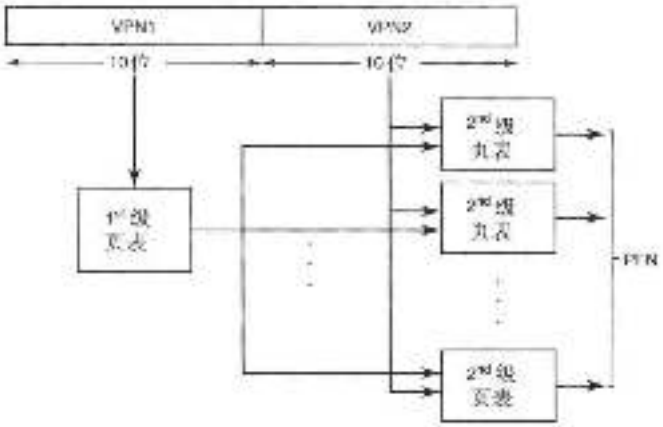


图8-18 二级页表。VPN的最高位（VPN1）用于对一级页表进行检索，从而获取二级页表的内存地址。每个二级页表保存VPN最后几位（VPN2）指定的虚页的映射

例8-8 考虑64位虚拟地址空间，8KB页面大小的内存系统，我们使用五级页表。每个进程第一级页表保存2K（2048）个页表项，剩下的4个级别的页表都会保存1K（1024）个页表项。

a. 给出系统中与这一多级页表结构相应的虚拟地址的布局。

b. 每个进程需要的总页表空间是多少（即所有级别页表的总和）？ 347

答：

a. 对于8KB的页面大小，每页的偏移地址位数是13。

那么VPN的位数=64 − 13 = 51位

用于第一级页表的位数（2048个项）= 11。

剩下4级页表每个需要的位数（每个页表有1024项）= 10。

虚拟地址布局如下：



b. 第一级页表中项的数目 = 2^{11}。

第二级页表中项的数目 = 2^{10}。

利用图8-18类似，有2^{11}个二级页表（每个和一级页表项对应）。所以总的二级页表项的数目是2^{21}。

三级页表中项的数目 = 2^{10}。

共有2^{21}个这样的三级页表（每个对应二级页表中的一项）。所以三级页表项的数目 = 2^{31}。

四级页表中项的数目 = 2^{10}。

总共会有2^{31}个这样的四级页表（每个对应三级页表中的一项）。所以，四级页表项的数目 = 2^{41}。

五级页表中项的数目 = 2^{10}。

总共会有2^{41}个这样的五级页表（每个对应四级页表中的一项）。所以，五级页表项的数目 = 2^{51}。

所以虚拟内存系统页表的总大小 = $2^{11} + 2^{21} + 2^{31} + 2^{41} + 2^{51}$。

对于这样的虚拟内存系统如果采用单级页表的形式那么每个页表表示有2^{51}项。

8.7.2 局部页表项的访问权限

页表项通常包含的内容除 PFN 和有效位之外还有其他信息。例如，它会包含特定页的访
348 问权限，如只读、读写，等等。和第 7 章内存管理器的功能（见 7.1 节）阐述的观点一致，这点很重要。特别注意内存管理器要提供内存保护和进程间的独立性。更进一步，进程还要保护自己不受程序错误的影响（例如，程序无意中对错个内存印记的访问）。最后，进程间当有需要时应能够共享内存空间。页表项的访问权限信息是硬件和软件交互协作的另一种方式。在进程创建时，内存管理器会在页表中为程序内存印记中的页设置访问权限。例如，对于包含代码的页会设置为只读，对于数据的页会设为读写。硬件在地址转换时会对每次内存访问的权限进行检查。当检测出越权访问时，硬件会进行纠零。例如，如果一个进程试图去写一个只读页，会导致访问进入陷入，并将控制交由操作系统来采取措施。

页表项也是操作系统为每个页存放其他相关信息的地方。例如，页表项可以包含发生页错误时情况下从磁盘引入页面的信息。

8.7.3 反向页表

因为虚拟内存通常比物理内存大很多，一些体系结构（例如 IBM 的 Power 处理器）会用反向页表，这实际上就是页帧表。反向页表减轻了对每个进程页表的需要。更深一步，反向表的大小和物理内存的大小（帧数上）相等，而不是虚拟内存。不幸的是，反向页表使硬件关于逻辑地址向物理地址的翻译变得复杂。所以，在这样的处理器中，硬件通过 TLB 机制进行地址转换。当遇到 TLB 缺失时，硬件会把控制（通过陷入）转交给操作系统来解决软件上地址转换的问题。操作系统也会负责更新 TLB 表。体系结构通常会提供特殊的指令集用于特权模式下的读、写和清除 TLB 中的项。

小结

现代操作系统的内存子系统包括分页守护进程、替换管理器、页错误处理程序等。子系统与 CPU 调度器和 I/O 子系统进行密切协调。下面是我们对本章进行的快速小结：

- 按需分页基础，包括硬件支持和操作系统用于分页的数据结构；
- 处理页错误时 CPU 调度器和内存管理器之间的交互；
- 349 页替换策略，包括 FIFO、LRU 和第二次机会替换策略；
- 用于减少页错误损失的技巧，包括保存一个页帧池可以随时为页错误处理分配帧，尽可能延迟将必要的替换页写回磁盘以及替换帧到被替换页的反向映射；
- 颠簸和进程用于控制颠簸的工作集；
- 为了保持流水线处理器持续工作用于加速地址转换的旁路转换缓存；
- 有关内存管理器更深入的探讨，包括多级页表和反向页表。

我们在第 7 章中观察到，现代处理器支持基于页的虚拟内存。相应地，用于现代处理器的操作系统如 Linux 和 Microsoft Windows（NT、XP 和 Vista）都实现了基于页的内存管理。8.3.5 节讨论的第二次机会页替换策略是一种很流行的策略，和其他策略相比它具有简便性和相对高效性。

练习题

1. 对于有 5 个阶段的流水线处理器，当发生页错误时，为了指令重启硬件上需要做哪些工作？

2. 描述分页内存管理器中页面表和磁盘映射数据结构所扮演的角色。

3. 描述页错误处理的步骤。

4. 描述进程调度器和内存管理器之间的交互。

5. 第二次机会页替换算法和简单的 FIFO 算法有什么区别？

6. 考虑一种体系结构，其中 TLB 中的每项都有

1 个访问位（当 CPU 访问相关 TLB 中的项进行地址转换时硬件会自动进行设置）。

1 个脏位（当 CPU 访问相关 TLB 中的项进行内存访问时硬件会自动进行设置）。

这些位和 8.6 节讨论的关于 TLB 其他方面的完全一样。该体系结构提供了 3 条特殊指令：

- 一个用于对特定 TLB 项的访问位进行采样（Sample_TLB(entry_num)）；
- 一个用于对特定 TLB 项的访问位进行清除（Clear_refbit_TLB(entry_num)）；
- 一个用于对所有 TLB 项的访问位进行清除（Clear_all_refbits_TLB(ALL)）。

350

使用 TLB 的额外信息提出页替换的实现策略。给出用于维护并实现页替换策略的算法的数据结构和伪代码。

7. 进程有如下的内存访问序列：

1 3 1 2 3 4 2 3 1 2 3 4

给定的序列（表示进程的虚页号）在进程执行中重复进行。假设有 3 个物理页面。

给出真正的 LRU 页替换效率的分页活动。给出 LRU 将在前 12 次访问所替换的页。给出哪些访问会命中，哪些访问会出现页错误。

8. 运型有如下的内存访问序列：

4 3 2 2 3 4 2 4 2 2 3 4

给定的序列（表示进程的虚页号）在进程执行中重复进行。最佳页替换策略前 12 次访问的分页活动应该是怎样的？给出哪些访问会命中，哪些访问会出现页错误。

9. 处理器要得到虚拟内存地址 0x30020 的内容。使用简单的分页策略将它分解为 VPN=0x30 和偏移地址 0x020。

PTR（一个用于保存页表地址的 CPU 寄存器）的值为 0x100。表明这个进程的页表从地址 0x100 开始。

机器采用按字寻址。页表的每项为单字长：

PTR=0x100	
VPN	Offset
0x30	0x020

其中的地址对应页中的内容如下：

物理地址	内容
0x00000	0x00000
0x00100	0x00010
0x00110	0x00000
0x00120	0x00040
0x00130	0x00020
0x10000	0x03737
0x10020	0x02424
0x20020	0x01111
0x22020	0x02222
0x25000	0x03535
0x25020	0x06666

351

地址是如何计算的？

当缺页异常处理器时的内容是什么？

最坏情况下会有多少内存访问？

10. 在时间间隔 t1 ~ t2 中，3 个进程 P1，P2 和 P3 分别记录了如下的虚页访问：

```
P1: 0, 0, 1, 2, 1, 2, 1, 1, 0
P2: 0, 100, 101, 102, 103, 0, 1, 2, 3
P3: 0, 1, 2, 3, 5, 1, 2, 3, 4, 5
```

给定的 3 个进程在此时间间隔内的工作集分别是什么？

此时间间隔中系统的总内存压力是多少？

11. 对于一个有 20 位页编数的虚拟内存系统，结用对于一级页表和二级页表分配策略的实际不同。对于二级页表，假设布局和图 8-18 类似。我们只对驻内存中的页表感兴趣。对于一个二级页表，存放在内存中的是第一级页表。对于单级页表，整个页表都需要存储在内存中。你需要算出不同组织结构（单级或二级）的 PTE 细节来计算每种组织结构总的页表需求。

参考文献注释和扩展阅读

Peter Denning 开创了工作集模型及其对性能影响的研究工作 [Denning, 1968]。FIFO 替换算法的缺点首先由 Belady 等发表于 [Belady, 1969]。用 Belady 的名字命名的最优页替换算法 "Belady 最小" 算法最早发表于 [Belady, 1966]。Carr 和 Hennessy[Carr, 1981] 提出了 WSCLOCK，是本章讨论的基本时钟算法的增强版，其中利用了工作集的概念。Silberschatz 等在教科书 [Silberschatz, 2008] 中对页替换算法进行清晰的介绍。Bryant 和 O'Hallaron 在其教科书中从程序员的角度对虚拟内存进行了很好的
352 讲解 [Bryant, 2003]。

分级存储体系

我们首先来了解什么是分级存储体系。目前为止，我们一直把物理内存当作黑盒对待。在对 LC-2200（第 3 章和第 5 章）进行讨论时，我们把内存看作数据通路的一部分。这种安排有一个隐含假设，即访问内存花费的时间和其他数据通路类操作所花费的时间一样多。我们稍微深入讨论这一假设。当今，处理器时钟速率达到了 GHz 的程度，这意味着 CPU 时钟周期小于 1ns。我们把它和当前速度最快的内存进行比较（circa 2009）。应用动态随机访问存储器（Dynamic Random Access Memory，DRAM）技术的物理内存的时钟周期的数量级是 100ns。我们知道在流水线处理器应用中最慢的部分决定处理器的时钟周期。在给定流水线内存访问的 IF 和 MEM 阶段，我们要想办法衔接 CPU 和内存之间存在的 100:1 速度上的差距。

在内存系统中定义两个有关频率的术语——访问时间（access time）和时钟周期（cycle time）——是很有用的。访问时间的意思是从向内存提交一个请求到获取数据之间的时间延迟。向内存系统提交的两次连续的请求之间的时间差称为时钟周期。有许多因素会对访问时间和时钟周期造成影响。例如，DRAM 技术用单晶体管存储 1 位。读取该位会耗尽存储的电量，所以在下次读取该位前要进行补充操作。这就是在 DRAM 中时钟周期和读取访问时间之间有差距的原因。除了实现内存系统所用的特定技术外，在连接内存系统和处理器之间的总线上的传输延迟也增加了访问时间和时钟周期之间的差距。

让我们重新来看看 LC-2200 的处理器数据通路。它包含一个寄存器堆，寄存器堆也是一种存储器。一个小的 16 元素寄存器堆的访问时间与访问其他数据通路元素的速度相当。有两个原因会造成这种情况。第一个原因是寄存器堆使用了不同的技术，即静态随机访问存储器（Static Random Access Memory，SRAM）技术。这项技术的优点在于速度快。SRAM 在速度 353
上优于 DRAM 的原因在于它使用了 6 个晶体管存储每一位，这样就不需要读后进行充电。基于同样的原因，SRAM 的访问时间和时钟周期没有差别。作为一条经验法则，SRAM 的时钟周期是 DRAM 的 8 ~ 16 倍。正如你所猜测的，因为 SRAM 每位有 6 个晶体管（而 DRAM 每位有 1 个晶体管），所以 SRAM 通常比 DRAM 的体积大。同样的原因，SRAM 也会耗费更多的电量。毫无疑问 SRAM 每位相对于 DRAM 也会贵一些（大约是 8 ~ 16 倍）。

物理内存比寄存器堆慢的第二个且更具说服力的原因是巨大的尺寸。每个寄存器堆的容量通常是 16，32 或 64 元素项。另一方面，度量内存容量通常用 KB、MB 或 GB。换句话说，即使我们在物理内存中使用了 SRAM 技术，但与寄存器堆相比，更大的结构也会导致更慢的访问时间。不考虑使用的技术（即 SRAM 或 DRAM），简单的现实是你可以拥有很快的速度或很大的容量，但不能两者均占。鱼和熊掌不能兼得。

从现实角度考虑，SRAM 不能用来实现大型内存系统，原因有很多，包括电量损耗、芯片尺寸、大规模 SRAM 不可避免的时间延迟，最重要的原因是用这项技术实现大内存的花费。另一方面，与 SRAM 相比，DRAM 技术能量消耗小且能有大规模的集成度。例如，引用 2007 年的数据，单个 DRAM 芯片可以有 256Mb，访问时间为 70ns。DRAM 技术的优点是尺寸。因此，使用 DRAM 技术实现大规模内存系统是经济且合理的选择。

9.1 缓存的概念

有少量的快速存储器和大量的慢速存储器是合理的。理想情况下，我们想要拥有大容量慢速存储器在尺寸上的优点和小容量快速存储器在速度上的优点。在前面讨论的关于速度和尺寸的前提下，我们选择用SRAM技术实现小容量快速存储器，用DRAM技术实现大容量慢速存储器。

分级存储体系同时实现了这两个目标。图9-1说明了分级存储体系的基本思想。主存是计算机指令集最底层的物理存储器。缓存，正如其名，是隐式存储器。在第8章（见8.6节）讨论将TLB作为存储地址转换信息的特殊情况时，我们就已经引入了缓存的概念和它在计算机系统不同方面的应用。特别地，在处理器进行内存访问的上下文中，该思想是指将从内存中取出的信息保存在缓存中。缓存比主存小，因此，也更快。

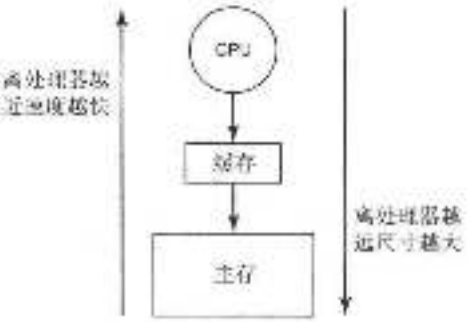


图9-1 一个基本的分级存储体系。图中显示一个二级存储体系。现代处理器可能会有更深层的存储体系（在主存下面有3级缓存）

我们的目标是：CPU从缓存中查找它在主存中需要的数据。如果数据不存在，它再从主存中检索它们。如果缓存能够满足大部分的CPU请求，那么我们就能够获得较好的缓存提速效果。

9.2 局部性原理

我们来了解为什么缓存首先工作。答案是我们在前面章节中讨论过的局部性原理（见8.4.2节）。

大致来讲，程序倾向于访问内存中相对较小的区域，在给定时间间隔中不会考虑实际的内存印记。虽然活动区域会随时间变化，但这些变化是渐进的。局部性原理专注于这种程序趋势。

局部性原理有两个维度，即空间和时间。空间局部性指的是程序有很高的概率访问邻近的内存单元，如果它访问i，就很有可能访问$i-3$、$i-2$、$i-1$、$i+1$、$i+2$、$i+3$等。这一现象很直观，程序的指令占用连续的内存单元。类似地，像数组或记录这样的数据结构占用连续的内存单元。时间局部性指的是对于目前经常访问的单元i在不久的将来会有很高的概率被访问。这一现象也很直观，考虑如下场景，在迭代算法中，通过循环重复执行相同的指令来更新数据结构。我们会在下面章节中利用这些局部特性来进行缓存设计。

9.3 基本术语

354
~
355

我们现在引入一些很直观的用于描述分级存储体系性能的术语。在引入前，有必要重新回忆一下第2章提到的工具箱和工具托盘。如果你需要一个工具，你首先会在工具托盘中寻找。如果工具托盘中有需要的工具，那就节省了去工具箱中寻找工具的时间；如果工具托盘中没有，你就要去保存工具箱的仓库中拿工具，使用工具，并将它放在工具托盘中。通常，如果在工具托盘中有需要的工具，那么整个过程的时间会很短。当然，有时工具托盘满了的时候你也要将一部分工具放回工具箱中。从数学的角度来讲，如果我们知道能够从工具托盘

中找到工具的概率，那么1减去这个概率就是去工具箱中寻找工具的概率。

现在我们来定义一些基本的术语。

- **命中**（hit）：指的是CPU在缓存中发现了内存地址的内容，这样就避免了从更深层的分级存储体系中读取，这就像在工具托盘中找到了工具。命中率（h）是CPU在缓存中成功找到数据的概率。
- **缺失**（miss）：指的是CPU没有在缓存中找到要找的数据，因此导致从更深层的分级存储体系中读取数据，这就像在工具托盘中没有找到工具，要去工具箱中寻找。缺失率（m）是CPU没有在缓存中找到数据的概率，等于$1-h$。
- **缺失损失**（miss penalty）：指的是由于在分级存储体系的任何一层中发生缺失而造成的时间损失，这就像在工具托盘中没有找到工具而去工具箱中寻找所造成的时间损失。
- **有效内存访问时间**（Effective Memory Access Time，EMAT）：指的是CPU的有效访问时间。EMAT由两部分组成：

  a. 在缓存中查找CPU要访问的内存单元所花费的时间，定义为**缓存访问时间**（cache access time）或**命中时间**（hit time）。

  b. 当缓存中缺失时，从更深层的存储中查找并读取缺失数据的时间，定义为**缺失损失**（miss penalty）。

每次访问内存，CPU都会有第一部分的时间。第二部分，即缺失损失，是由分级存储体系中深层的访问时间来决定的，它由等待缺失处理时CPU所经历的时钟周期的数目决定。缺失损失由许多因素决定，包括缓存的组织结构和主存系统的设计细节。在后面的章节中我们会对这些因素进行讲解。由于CPU只有在发生缺失时才会有这样的时间损失，所以为了计算第二部分，需要计算CPU所需的内存单元不在缓存中进而需要从更深层存储中进行调用所花费的时间和概率（由缺失率（m）决定）。

m代表缺失率，T_c代表缓存访问时间，T_m代表缺失损失，那么

$$\mathrm{EMAT} = T_c + m \times T_m \tag{9-1}$$

356

9.4 多级存储层次

现代处理器有不同等级的缓存。例如，最先进的处理器（circa，2009）在一个芯片上至少有二级缓存，称为第一级（first-level，L1）缓存和第二级（second-level，L2）缓存。你也许会想，如果第一级缓存和第二级缓存都在同一个芯片上，为什么不可以合并为一个更大的缓存？答案是实际上我们关心两个因素——对缓存的较短访问时间（即命中时间）和较低的缺失率。一方面，我们希望命中时间越短越好并与处理器的时钟周期保持同步。因为缓存的尺寸对访问时间有直接的影响，所以我们希望缓存尽量小。另一方面，由于处理器周期时间和主存访问时间之间的差距逐渐拉大，所以我们希望缓存容量大一些。为了解决这两个互斥的因素，我们采用多级缓存。第一级缓存是为了速度上的加速，即与处理器的时钟周期时间同步，所以很小。第二级缓存的速度仅影响第一级缓存没有命中造成的缺失损失，不直接影响处理器的时钟周期时间。于是，第二级缓存的设计主要考虑降低缺失率，所以大一些。处理器安放在集成电路板即主板上，上面有主存（物理内存）[⊖]。企业级机器（数据库或服务器）的高端CPU甚至有一个不在芯片上的第三级缓存（L3）。考虑到访问速度，这些多级缓存通

⊖ 不同级别缓存的访问时间或大小就是技术的进步不断变化。

常应用 SRAM 技术。

由于前述原因，当缓存离处理器较远时缓存的尺寸较大。如果 S_i 表示第 i 级缓存的大小，那么

$$S_{i+1} > S_{(i-1)} > \cdots > S_2 > S_1$$

相应地，当缓存离处理器较远时缓存的访问时间较长。如果 T_i 是第 i 级的访问时间，那么

$$T_{i+1} > T_{(i-1)} > \cdots > T_2 > T_1$$

将 EMAT 术语作为一个整体推广到分级存储体系，T_i 和 m_i 分别表示分级存储体系各级的访问时间和缺失率。对分级存储体系中的第 i 级存储，有效内存访问时间由下式递归得出：

$$\mathrm{EMAT}_i = T_i + m_i \times \mathrm{EMAT}_{i+1} \tag{9-2}$$

我们现在来定义分级存储体系的概念：

357 分级存储体系定义为包含一个处理器能够直接或间接访问的指令和数据的所有存储器。

直接访问的意思是存储器对于 ISA 来说是可见的。间接访问的意思是对于 ISA 来说是不可见的。图 9-2 说明了这一定义。在第 2 章，我们介绍了寄存器的概念，它是对于从 ISA 直接访问的离处理器最近也是最快的数据存储。通常情况下，ISA 中的存取 load/store 指令和算术/逻辑指令访问寄存器。L1、L2 和 L3 是处理器每次从主存读取指令或数据时都会访问的不同级别的缓存。通常 L1 和 L2 在芯片上，L3 不在芯片上[⊖]。主存用来存储程序的指令和数据。处理器显式地访问内存读取指令（通过程序计数器）和数据（通过 load/store 指令和其他使用内存操作数的指令）。值得注意的是有些结构还允许处理器通过 ISA 直接访问缓存（例如，刷新缓存中的内容）。二级存储器用作程序的整个内存印记的存储，主存中只保存其中的一部分，这与第 8 章中讨论的工作集原则相一致。换句话说，二级存储器用作虚拟内存。处理器遇到页错误时就隐式地访问虚拟内存以便将缺失的虚页引入主存（即物理内存）中。

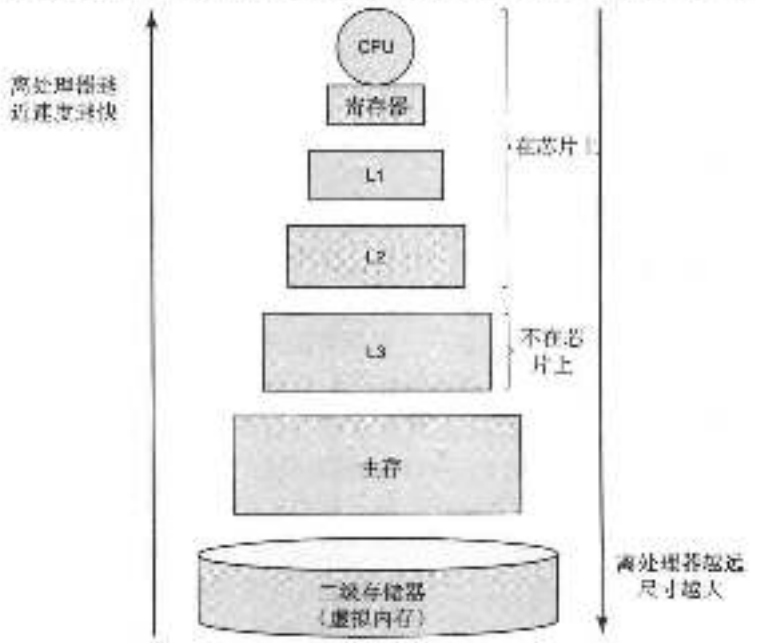


图 9-2 从处理器寄存器到虚拟内存的整个分级存储体系

⊖ 较新的设计将 L3 缓存放在芯片上（见本章小结后的例子）。

本章主要讨论包括缓存和主存的部分分级存储体系。从前面的章节（第2、3和5章）中，我们已经对于寄存器有了较好的认识。类似地，在第7、8章中我们对于虚拟内存有了较好的了解。在本章，我们主要关注缓存和主存。因此，在本章剩下的部分我们用术语缓存层次结构（cache hierarchy）和分级存储体系（memory hierarchy）来表示同一事物。

例9-1 考虑如图9-3所示的三级存储体系。计算有效内存访问时间。

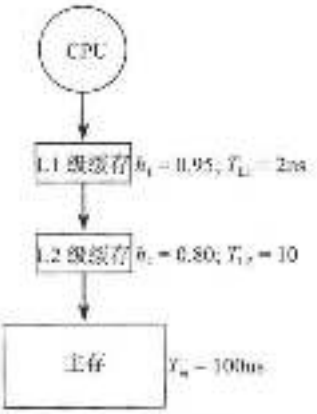


图9-3 三级存储体系

答：

$$EMAT_{L2} = T_{L2} + (1-h_2) \times T_m = 10 + (1-0.8) \times 100 = 30\text{ns}$$

$$EMAT_{L1} = T_{L1} + (1-h_1) \times EMAT_{L2} = 2 + (1-0.95) \times 30 = 2 + 1.5 = 3.5\text{ns}$$

$$EMAT = EMAT_{L2} = 3.5\text{ns}$$

358 : 359

9.5 缓存结构

缓存有3个需要注意的方面：布局策略、查找算法和有效性。

这3个方面分别与下面的3个问题相对应：

1）从内存中读取的数据放在缓存中的什么位置？

2）如何查找放在缓存中的数据？

3）如何知道缓存中的数据是有效的？

一个用于内存地址到缓存的映射函数是回答第一个问题的关键。除了映射函数外，第二个问题的答案涉及每个缓存项中用于精确确定缓冲内容的元数据。第三个问题引出了为了辅助查找算法，给每个缓存项设置一个有效位的必要性。

本书中我们通过一个简单的直接映射（direct-mapped）缓存例子来说明这3个方面的内容。在9.11节，我们将了解其他的缓存结构，即全相关（fully associative）和组相关（set-associative）。

9.6 直接映射缓存结构

直接映射的内存单元和缓存单元有一对一的关系[⊖]。也就是说，给定一个内存地址，在缓存中有唯一确定的单元存放该地址中的内容。为了更清楚地表述直接映射的工作，我们考

⊖ 当然，由于缓存比内存小，所以在一些内存单元和给定的缓存单元之间存在着多对一的关系。

虑一个简单例子——16字的内存和8字的缓存（如图9-4所示）。图中的阴影显示内存单元0～7分别对应于缓存单元0～7；类似地，内存单元8～16也分别对应于缓存单元0～7。

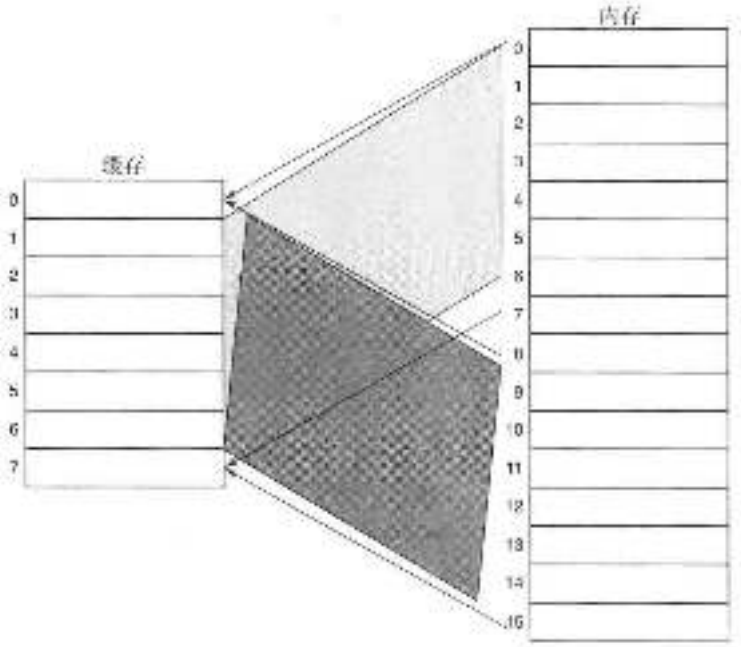


图9-4 直接映射缓存。在内存单元和缓存单元之间存在一对一的映射

在回答命中和有效性问题之前，我们首先了解直接映射的缓存中关于内存单元的布局策略。为了更好地进行说明，假设缓存为空，考虑如下的内存地址访问：

0,1,2,3,1,3,0,8,0,9,10（均为十进制地址信息）

因为缓存最初为空，所以前4个访问（地址0，1，2，3）在缓存中均为缺失，CPU从内存中读取信息并将它们保存在缓存中。图9-5显示了处理完前4次内存访问后的缓存。这些缺失都是不可避免的，称为强制缺失（compulsory miss），因为初始时缓存为空。

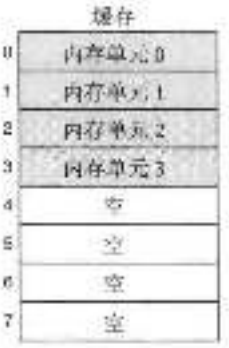


图9-5 在前4次访问后的缓存内容

接下来的3个CPU访问（地址为1，3，0）在缓存中命中，这样避免了从内存中查找数据。我们看看在接下来的CPU访问中将发生什么（地址8）。这个访问在缓存中缺失，CPU从内存中读取数据。现在我们需要弄清楚系统从内存单元中读取的数据将放在缓存中的什么位置。缓存在单元4～7有空间。然而，使用直接映射缓存单元0是存储内存单元8的唯一位置。图9-6显示了缓存的这一状态。这也是强制缺失，因为内存单元8的内容初始时没有在缓存中。

考虑下一个访问（地址0）。这个访问在缓存中也是缺失的，CPU需要从内存中读取它并把它存储在缓存单元0，这也是内存单元0的唯一位置。图9-7显示了缓存中的新内容。这个缺失的发生是由于内存单元0和8都映射到缓存中的同一位置造成冲突所引起的，因此这也称为冲突缺失。尽管在缓存中有空闲的空间，但由于直接映射还是会有冲突缺失发生。注意前面的缺失（图9-6中的单元8）也会造成冲突，因为单元0的数据已经保存在缓存中。忽略

这种情况，第一次访问内存单元会造成缺失。这也是为什么我们把图 9-6 中的缺失归为强制缺失。我们在 9.15 节再详细讨论这两种不同种类的缺失。



图 9-6 内存单元 0 被替换为 8



图 9-7 内存单元 8 被替换为 0（冲突缺失）

9.6.1 缓存查找

到目前为止，我们知道了 CPU 和内存之间的信息交换，或者称为握手（handshake）：CPU 提供地址和命令（例如，读）并从内存中读取数据。缓存的引入（见图 9-1）改变了简单设置。CPU 首先查找缓存确定需要的数据是否在缓存中，只有发生缺失才会使 CPU 进行标准的 CPU 和内存间的握手。

下面介绍 CPU 如何在缓存中查找对应的内存单元。特别地，我们需要搞清楚 CPU 中的索引如何通过直接映射提交给缓存。重新看看图 9-4，我们可以看到 CPU 地址如何映射到它们对应的缓存索引。

可以按照下式计算缓存索引的数值：

$$\text{内存地址} \bmod \text{缓存大小}$$

例如，给定内存地址 15，缓存索引为

$$15 \bmod 8 = 7$$

类似地，内存地址 7 所对应的缓存索引为

$$7 \bmod 8 = 7$$

基本上，为了建立缓存索引，我们简单地取内存地址中能够表示缓存大小的最低几位。在我们前面的例子中，因为缓存有 8 项，所以需要 3 位作为缓存索引（内存地址的最低 3 位）。

假设 CPU 需要从内存地址 8 读取数据，1000 是内存地址的二进制表示，缓存索引是 000。CPU 在索引 000 处查找缓存单元。我们需要一些方法来了解该缓存项的内容是来自内存单元 0 还是 8。所以，除了数据外，我们在每个缓存项中还需要获取额外的信息来处理多个内存地址对应同一缓存地址的情况。内存地址用于产生缓存索引的位已经可以用来实现这一目的。我们将这一额外的信息（存储在缓存中）称为标记（tag）。例如，内存地址 0000 和 1000 的最高有效位的标记分别为 0 和 1。所以，对于每个缓存项需要 1 位标记（见图 9-8）。如果 CPU 需要访问内存单元 11，它就在缓存中查找单元 11 mod 8——缓存中的单元 3。这个单元的标记是 0。所以，缓存项中包含的数据对应于内存单元 3（二进制地址为 0011），而不是内存单元 11（二进制地址为 1011）。

假设CPU产生内存地址0110（内存地址6）。我们假设这是内存地址6第一次被CPU访问。所以，它不可能出现在缓存中。我们看看当CPU试图读取内存单元6时会发生什么。CPU会首先查找缓存中单元6（6 mod 8）。如果标记恰巧为0，那么CPU会假定这个数据对应于内存单元6。那么，在这个例子中CPU就不会去内存中查找数据；这里标记位恰巧为0。所以，在这个缓存项中的数据并不与实际内存单元6相对应。我们可以看出这是错误的。并且它说明对每个缓存项我们需要更多的信息来避免这种错误。标记对于消除缓存中当前的内存单元歧义很有用，但它并不能确定缓存项是否有效。为了解决这一问题，我们为缓存中的每项增加了有效位（见图9-9）。

	标记	数据
0	1	内存单元8
1	0	内存单元1
2	0	内存单元2
3	0	内存单元3
4		空
5		空
6		空
7		空

图9-8 每项中带有标记字段和数据字段的直接映射缓存。标记字段指一块标识给定缓存单元中的实际内存单元

	有效位	标记	数据
0	1	1	内存单元8
1	1	0	内存单元1
2	1	0	内存单元2
3	1	0	内存单元3
4	0	X	空
5	0	X	空
6	0	X	空
7	0	X	空

图9-9 每项含有有效字段、标记字段和数据字段的直接映射缓存。标记字段中标为“X”代表当前条件下“不关心取值”

9.6.2 缓存项中的字段

对上文进行总结，每个缓存项都包含3个字段（见图9-10）。因此，从查找缓存的角度，CPU产生的内存地址有两部分：标记和索引。索引是CPU产生的包含内存地址的具体缓存单元；标记是地址的一部分，它有助于消除具体缓存项的内容的歧义（见图9-11）。我们用内存地址的最低有效位（即最右边的位）作为缓存索引来利用空间局部性原理。例如，在我们的简单缓存中，如果用内存地址的最高3位作为缓存索引，那么内存单元0和1将映射到缓存中的同一地址。



图9-10 每个缓存项的不同字段



图9-11 对CPU产生的内存地址进行转换，以便缓存查找

图9-4有8项直接映射缓存，考虑内存地址0，1，0，1的访问序列。假设内存地址的最高3位用作缓存索引，最低位（因为在本例中内存地址只用4位表示）用作标记。图9-12显示了每次访问后缓存的内容。注意，即使剩余的缓存为空，内存访问序列中的相同缓存项

(缓存中的第一行)是如何被重用的。每次访问都产生缺失，在该访问模式下会替换缓存中前一次的内容。

图 9-12 中的情况很不理想。回忆 9.2 节提到的局部性原理(时间和空间局部性)。连续的内存访问应该映射到不同的缓存单元中。这就是选择图 9-11 中地址转换的原因。 364 | 365



图 9-12 将内存地址中的最高有效位作为缓存索引的内存访问序列

9.6.3 用于直接映射缓存的硬件

我们首先对之前讨论的想法进行总结。图 9-13 显示了直接映射缓存的硬件结构。

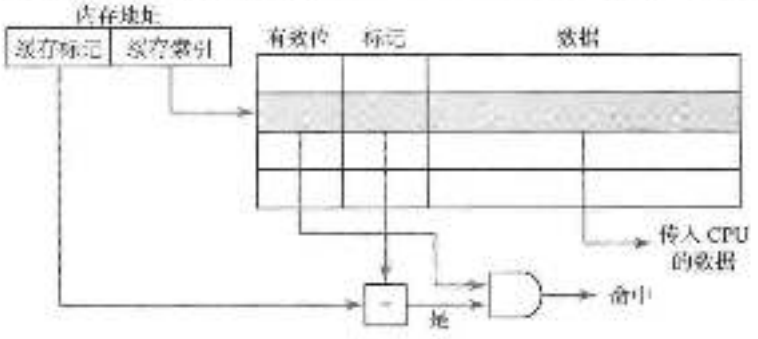


图 9-13 用于直接映射缓存的硬件。灰色项是通过缓存索引选中的缓存单元

内存地址的索引部分选择缓存中的特定项(图 9-13 中灰色的缓存项)。图 9-13 中的比较器将这一项的标记字段与内存地址的标记字段进行比较。如果是匹配的且这一项是有效的，那么它会发出一个命中信号。因为命中，所以缓存将选中项的数据字段(也称为缓存行)发送给 CPU。缓存块(cache block)是另一个与缓存行类似的术语。我们目前已经用了 3 个类似的术语：缓存项、缓存行和缓存块。虽然有这么多的术语对应同一个事物，但是还是请读者记住这些名词，因为计算机体系结构书中经常互换地使用这些术语。 366

注意，缓存中需要的真实存储空间比缓存的数据部分大。有效位和标记字段称为元数据(metadata)，用来管理存储在缓存中的真实数据，它也代表空间开销。

目前为止，我们已经把内存单元对应到缓存的数据字段中。内存单元的大小由指令集所允许的内存访问粒度决定。例如，如果体系结构是按字节寻址的，那么 1 字节就是内存操作数最小的可能尺寸。通常，在这样的体系结构中，字宽为几个整数字节。我们可以将每个字

节放在不同的缓存行中，但从9.2节我们知道，空间局部性原理建议如果访问字的一个字节，那么很可能会访问同一字的其他字节。所以，即使体系结构是按字节寻址的，在每个缓存行中保留完整的字也是很有意义的。因此，CPU产生的内存地址将被转换成3个字段（如图9-14所示）：缓存标记、缓存索引和字节偏移量。字节偏移量定义为区分一个字中不同字节的地址位。例如，如果字宽是32位且体系结构是按字节寻址的，那么地址中的最低2位就是字节偏移量。



图9-14 当单个缓存块包含多个字节时对CPU产生的内存地址进行转换

例9-2 我们考虑某类内存系统的直接映射缓存的设计。假设CPU产生一个32位的按字节寻址的内存地址。每个内存块包含4字节。一次内存访问将整个字读入缓存中。直接映射缓存采64KB（这就是缓存能够存储的数据量）。每个缓存项包含被读的一个字。计算缓存中用于存放位和标记字段的额外存储空间。

答案：

假设使用小端存储。0是地址中的最低有效位。用这种标记法，地址的最低两位，即位1和位0，区分一个字地址中的字节。一个缓存项保存字中全部4个字节。于是，地址的最低两位用来唯一地决定一个字中的字节，不需要将一块识别特定的缓存项。所以，这些位不用来做缓存查找的索引部分。

缓存大小与每项中保存的数据大小的比是缓存项数：

$$64\text{KB} / (4\text{字节} / \text{字}) = 16\text{K}\text{项}$$

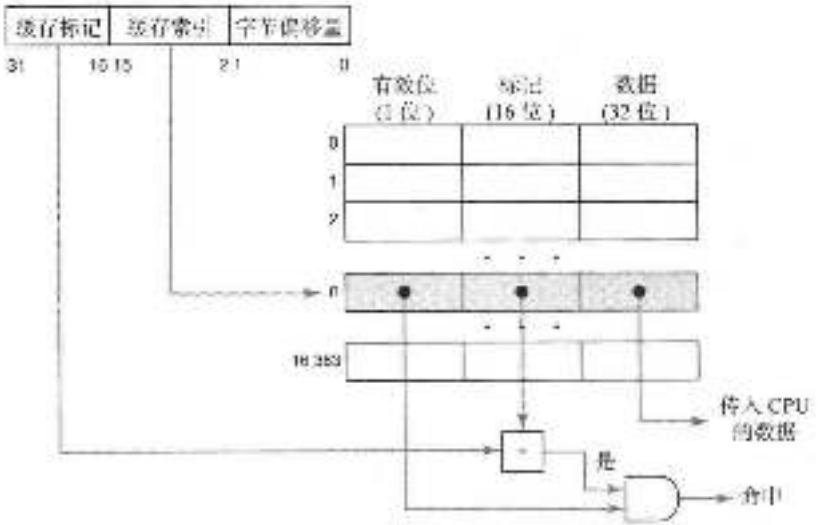
16K项需要14位来表示。因此位2～15形成缓存索引。剩下的位16～31形成标记。这样，每个缓存项都有16位标记。

额外的元数据：16位标记位+1位有效位=17位。

因此，元数据需要为额外存储空间为

$$17\text{位} \times 16\text{K}\text{项} = 17 \times 16\,384 = 278\,528\text{位}$$

下图显示了这个问题的缓存设计：



缓存需要的总空间（真实数据 + 元数据）
=64KB + 278 528
=524 288 + 278 528
=802 816

空间开销 = 元数据 / 总空间 = 278 528/802 816 = 35%

我们暂停如何减少空间开销。在图 9-2 中，每个缓存行保存一个内存字。一种减少空间开销的方式是修改缓存设计，使每个缓存行保存多个连续的内存字。例如，考虑在例 9-2 中每个缓存行保存连续的 4 个内存字。这将把缓存行的数目降低到 4K。块大小（block size）用来指出每个缓存行中连续的数据数。在例 9-2 中块大小是 4 字节。如果每个缓存行包含 4 个字，那么块大小就是 16 字节。为什么想要一个更大的块大小呢？而且，这将如何帮助我们减少空间开销？这能帮助提升内存系统的整体性能吗？读者先思考这些问题。9.10 节再进行块大小对缓存设计影响的详细讨论。

367
|
368

9.7 流水线处理器设计的影响

前面介绍了内存和处理器之间的缓存，我们重新来看流水线处理器设计和缓存中指令的执行。图 9-15 是第 6 章（见图 6-6）流水线处理器的图。

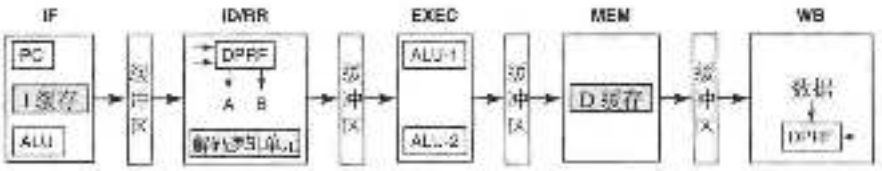


图 9-15 带缓存的流水线处理器

注意我们已经将部分存储器做了替换，在 IF 和 MEM 阶段的 I-MEM 和 D-MEM 分别换成了 I 缓存和 D 缓存。假设访问命中，缓存使 IF 和 MEM 阶段与其他流水线阶段有类似的周期时间。我们看看如果缓存访问缺失会发生什么。

- **在 IF 阶段缺失**：当 I 缓存缺失时，IF 阶段向内存发送访问来读取指令。正如我们所知，内存访问时间是 CPU 时钟周期的 10 倍。直到指令从内存中读入缓存，IF 阶段才停止向下一阶段发送空指令（NOP）。
- **在 MEM 阶段缺失**：在 D 缓存中缺失只与内存访问指令（load/store）相关。与 IF 阶段类似，直到内存访问完成，MEM 阶段的缺失才停止向 WB 阶段发送空指令。在执行缺失指令前它也不会执行目前正在执行的从较去阶段读取的其他指令。

我们将内存延迟（memory stall）定义为等待需要完成的内存操作所浪费的处理器周期。内存延迟有两方面：处理器读访问缓存造成的读延迟（read stall）；处理器写访问缓存造成的写延迟（write stall）。我们将在下一节定义和讲解这些延迟，同时也会介绍如何避免它们，因为这些对处理器流水线性能起决定性作用。

369

9.8 缓存读 / 写算法

在本节，我们讨论读 / 写缓存的策略和机制。不同级的缓存会选择不同的策略。

9.8.1 CPU 对缓存的读访问

处理器需要访问缓存来读取内存单元中的指令或数据。在 LC-2200 ISA 五阶段流水线中，这可能由于在 IF 阶段读取指令或在 MEM 阶段为了加载指令而读取操作数决定。处理器和缓存采取的步骤如下：

- **步骤一**：CPU 将内存地址的索引部分（见图 9-16）发送给缓存。缓存进行查找，如果成功（缓存命中），它将数据发送给 CPU。如果缓存发送缺失信号，那么 CPU 将地址通过内存总线发送给主存。原则上，所有这些动作都发生在同一周期中（流水线的 IF 或 MEM 阶段）。

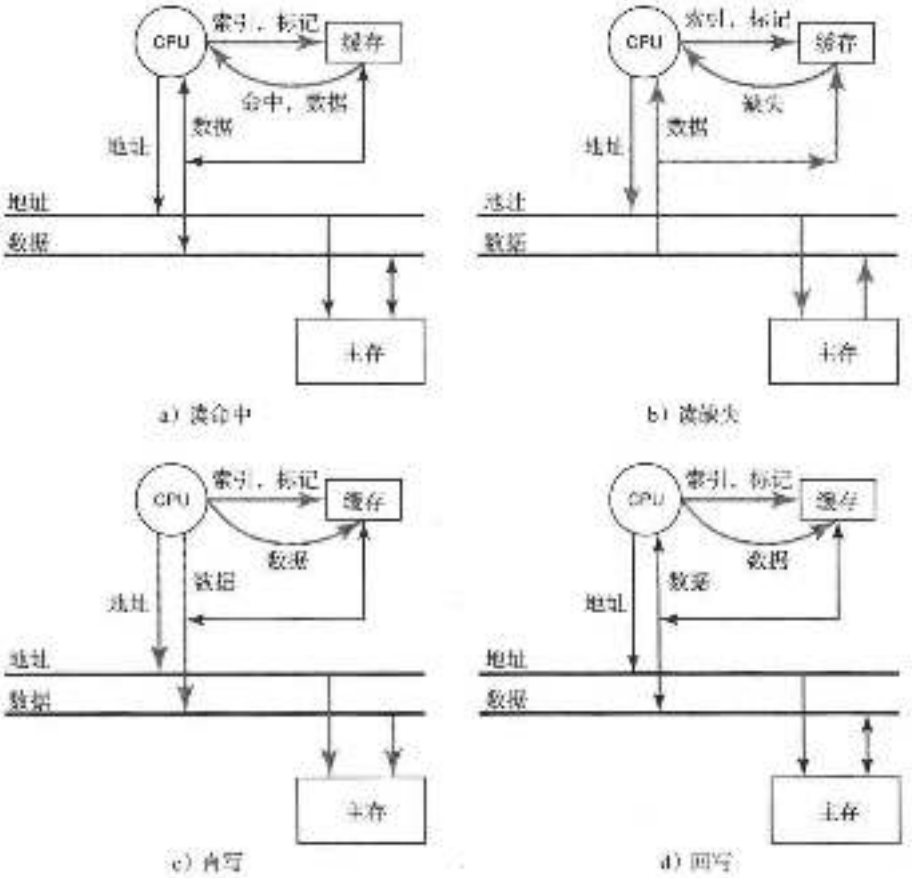


图 9-16 读写操作时，CPU、缓存和内存间的交互

- **步骤二**：在向内存发送地址后，直到收到来自内存的数据，CPU 会一直向下一阶段发送空指令（NOP）。读延迟定义为处理一个读缺失所花费的处理器时钟周期数。正如前面提到的，由于内存速度较慢，这可能会花费多个 CPU 周期。缓存分配一个缓存块来接收内存块。最后，主存将数据传送给 CPU，同时用数据更新分配的缓存块。缓存修改这个缓存项的标记字段并将有效位设置为有效。

9.8.2 CPU 对缓存的写访问

当有对内存单元进行写操作的指令时，处理器会对缓存进行写访问。在我们实现的 LC-

2200 ISA 五阶段流水线中，这可能发生在 MEM 阶段中，用于存储内存操作数。处理器对缓存的写访问有两种处理方式：直写（write through）和回写（write back）。

直写策略。主要思想是对于每个 CPU 写操作都更新缓存和主存。处理器和缓存采取的基本步骤如下：

- **步骤一**：在每次写操作（LC-2200 中的存储指令），CPU 都简单地向缓存写数据。没有必要检查有效位或缓存标记。缓存会更新相关项的标记字段并设置有效位。这些操作都在流水线的 MEM 阶段发生。
- **步骤二**：同时，CPU 向主存发送地址和数据。当然，因为内存访问要耗费好几个 CPU 周期去完成，所以这很影响性能。为了减轻这个性能瓶颈，通常会在 CPU 和内存之间的数据通路中安置一个写缓冲区（write buffer），如图 9-17 所示。写缓冲区是一个用来缓解 CPU 和内存之间速度差异的小硬件存储器（和寄存器堆类似）。对于 CPU 而言，一旦将地址和数据都放在写缓冲区中，写操作就完成了。因此，该操作发生在流水线的 MEM 阶段，不影响流水线的性能。

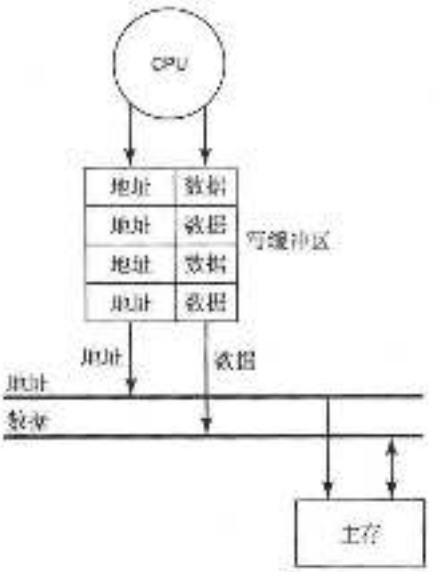


图 9-17 为了缓和 CPU 与主存速度上的差距，使用 4 个写缓冲区单元的直写策略。只要地址和数据已放入写缓冲区中，处理器写操作就完成了，并且流水线可以恢复工作。写缓冲区中的数据会在后台被传送到内存，这是与流水线操作同时进行的。如果写缓冲区满，那么流水线停止

- **步骤三**：写缓冲区独立于 CPU，完成主存的写操作。注意，如果在处理器试图向写缓冲区写数据时写缓冲区满，那么流水线会受阻直到写缓冲区中的一项被存入内存为止。写延迟定义为写操作浪费的处理器时钟周期数（忽略命中或在缓存中缺失的情况）。

采用直写策略，当发生写缺失时，采取不同的缓存块分配策略写延迟会有不同的表现。对于缺失情况，有两种缓存块分配策略：

- **写分配**：这是一种通常处理写缺失的方法。直觉上正在写的数据在将来也会被程序用到，所以应该将它存入缓存中。然而，因为在缓存中这个块也是缺失的，所以我们不得不分配一个缓存块，并将缺失的内存块写入其中。从这点上看，写缺失和读缺失的

370
|
372

操作类似。在直接映射缓存中，接收内存块的缓存块是提前确定的。然而，正如我们后面将看到的灵活布局策略（见9.11节），分配策略由其他的设计因素决定。

- **非写分配**：这是一种不常用的处理写缺失的方法。因为处理器不需要数据，所以写访问能够很快地完成。处理器只是简单地将要写的数据写入写缓冲区中以便完成流水线中MEM阶段需要的操作。于是，因为缺失的内存块不需要从内存中读出，所以也有写延迟。

回写策略。核心思想是当CPU写操作时只更新缓存。处理器和缓存采取的主要步骤如下：

- **步骤一**：CPU与缓存之间的交互和直写策略相同。我们假设内存单元已经在缓存中（即写命中）。CPU向缓存中写数据。缓存更新相关项的标记字段并设置有效位。这些操作都发生在流水线的MEM阶段。
- **步骤二**：选中的缓存项和对应的内存单元中的内容是不一样的。对CPU而言这不是问题，因为在内存读操作之前首先要检查缓存。因此，CPU总是获取缓存中的最新数据。
- **步骤三**：我们考虑在何时更新主存。缓存总是比主存小。因此，在某个时刻，可能需要替换现有的缓存项，为现在没在缓存中的内存单元腾出空间（我们将在9.14节讨论缓存的替换策略）。在进行替换时，如果CPU写入要被替换的缓存项，那么相应的内存单元要用该缓存项的最新数据更新。

处理器对写缺失的处理和读缺失一样。在采取了必要的处理读缺失的步骤之后，进行前面介绍的写操作。

我们需要一些机制来判断缓存项中的数据是否比相应内存单元中的数据新。每个缓存项中的元数据（有效位和标志字段）没有为缓存决策提供必要的信息。所以，我们给每个缓存项增加一个新的元数据——脏位（dirty bit），见图9-18。

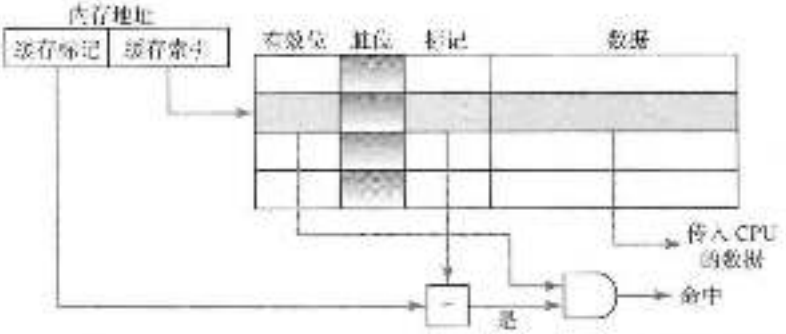


图9-18 采用回写策略的直接映射缓存结构。每项都有一个额外的字段（脏位）用来指示块是否为脏。水平阴影部分的块是缓存索引选择的块

缓存用下述方法使用脏位：

- 当处理缺失（缺失将内存单元装入缓存项）时，缓存清除脏位。
- 当CPU进行写操作时缓存设置脏位。
- 当替换时，缓存将缓存中的数据写到相应内存单元中。注意这与虚拟内存系统（见第8章）中将物理页换回写到磁盘的操作类似。

我们介绍了直写策略中的写缓冲区的概念。结果是对于回写策略，写缓冲区也是很有用的。注意，我们的目的是让处理器更高效。这意味着应该尽可能快地将缺失的内存块引入缓

存。换句话说，将缺失的块引入缓存比回写要替换的脏块更重要。这时，写缓冲区也派上了用场，优先从内存读数据，而不是向内存写数据。把要被替换的块（如果脏了）存放在写缓冲区中，它最终会被写回。但发送给内存的立即请求是对目前的缺失进行处理（读或写缺失）。当CPU没有要处理的读或写缺失时，来自写缓冲区的写请求才会传送到内存。这容易让人想起内存管理器优先从磁盘读取缺失页，而不是将脏的页面写入磁盘（见8.4.1节）。

两种写策略的对比。缓存应该采用哪种写策略呢？答案是取决于很多因素。一方面，直写策略能保证存中的数据永远都是最新的。然而，这是以每次写操作都要向主存发送数据为代价的。可以采取一些优化措施，例如，如果总是向写缓冲区中的同一内存单元重复地进行写操作（旧还没有被传入内存），那么它可以用一个新的写操作代替。然而，在程序中发生这种情况的概率很小。另一种优化，称为写合并（write merging），是将要写入同一内存块的不同部分的独立写操作合并在一起。回写策略的优点是速度快，并且重要的是，CPU每次写访问都不使用内存总线。因此，对于同一内存单元的重复写操作并不会导致内存总线上的流量过载。只有当被替换时缓存才会用内存单元中最新的数据更新内存。在这种情况下，注意对于回写策略，为了保存需要回写到主存的替换数据，写缓冲区也是很有用的。

直写策略的另一个优点是缓存永远是干净的。换句话说，当需要把缓存中的块替换为缺失块时，缓存永远不需要把被替换的块写入更低级的存储器中。这样对缓存进行直写操作，设计上更简单并且速度比回写缓存更快。所以，如果处理器有多级缓存，那么对于离处理器较近的缓存通常采用直写策略。例如，大部分现代处理器对L1级D缓存采用直写策略，对于L2和L3级缓存采用回写策略。

我们将在第10和12章中看到，写策略的选择会对I/O和多处理器系统的设计产生影响。采用不同的策略各有利弊。回写策略会降低内存总线的流量；同时，直写策略能保持内存总是最新的。

373
|
374

9.9 处理器流水线中的缓存缺失处理

内存访问造成处理器流水线不能连续地进行。所以要尽可能减轻缺失对处理器流水线的负面影响。不能在流水线的IF阶段隐藏缺失，但可以在MEM阶段隐藏（hide）缺失。

- **MEM阶段的读缺失：**考虑如下的指令序列。

```
i1: lw    r1, a       ;  r1 ← 内存单元a
i2: add   r3, r4, r5  ;  r3 ← r4 + r5
i3: nand  r6, r7, r8  ;  r6 ← r7 NAND r8
i4: add   r2, r4, r5  ;  r2 ← r4 + r5
i5: add   r2, r1, r2  ;  r2 ← r1 + r2
```

假设lw指令造成D缓存中的缺失。CPU不得不在MEM阶段等待这条加载（load）指令（停止先前阶段的工作，并向WB阶段发送空指令（NOP）），直到内存用数据响应为止。然而，指令i5使用加载到r1的值。让我们想想如何利用这点信息阻止对执行指令i2、i3和i4的执行。

在第5章中，我们给寄存器堆中的寄存器引入忙标位（busy bit）的概念来处理险象。对于修改寄存器值的指令，在ID/RR阶段将这个位置位（见图9-15）；当写操作完成时，清除该位。我们可以将这种思想扩展到内存加载中。

- **MEM阶段的写缺失：**写缺失问题的难度取决于写策略和缓存单元分配策略。首先，我们考虑采用直写策略的情况。如果缓存块分配策略是非写分配，那么因为有写缓冲区，

所以流水线就不会产生任何延迟。处理器只是简单地把写操作放在写缓冲区中以便完成 MEM 阶段需要的操作。然而，如果缓存块分配策略是写分配，那么这就需要进行和读缺失一样的操作。所以，处理器的流水线在 MEM 阶段就会有写延迟。缺失的数据块
375 需要在写操作完成前传入缓存，尽管有写缓冲区。对于回写策略，因为写缺失需要像读缺失一样处理，所以 MEM 阶段的写延迟是不可避免的。

9.9.1 在流水线性能上缓存缺失对内存延迟的影响

让我们重新考虑程序执行时间。在第 5 章中，我们将程序执行时间定义为：

$$\text{执行时间} = \text{执行的指令数目} \times \text{CPI}_{Avg} \times \text{时钟周期}$$

流水线处理器试图让 CPI_{Avg} 等于 1，因为它尝试每个周期处理 1 条指令。然而，结构、数据和控制险象都会对流水线造成影响，所以 CPI_{Avg} 的值大于 1。

分级存储体系加剧了这一问题。每条指令都至少要访问内存一次，即读取指令。另外，对于内存访问指令可能还有额外的内存访问。如果这些访问导致了缺失，流水线上就会出现空指令。我们将这些由分级存储体系造成的额外空指令称为存储器延迟周期（memory stall cycle）。

这样，关于执行时间我们有了更精确的描述：

$$\text{执行时间} = [(\text{执行的指令数目} \times (\text{CPI}_{Avg} + \text{内存延迟}_{Avg})] \times \text{时钟周期} \quad (9\text{-}3)$$

我们将处理器的有效 CPI 定义为：

$$\text{有效 CPI} = \text{CPI}_{Avg} + \text{内存延迟}_{Avg} \quad (9\text{-}4)$$

程序经历的总内存延迟定义为：

$$\text{总内存延迟} = \text{指令的数目} \times \text{内存延迟 Avg} \quad (9\text{-}5)$$

每条指令的平均内存延迟数目定义为：

$$\text{内存延迟}_{Avg} = \text{每条指令的缺失数目}_{Avg} \times \text{缺失损失}_{Avg} \quad (9\text{-}6)$$

当然，如果读和写造成不同的缺失损失，我们需要对它们区别对待（见例 9-3）。

例 9-3 有一个流水线处理器，若不考虑内存延迟它的平均 CPI 值为 1.8。I 缓存的命中率为 95%，D 缓存的命中率为 98%。假定内存访问指令占所有执行指令的 30%。这其中，80% 是加载指令，20% 是存储指令。平均来说，读缺失损失是 20 个周期，写缺失损失是 5 个周期。考虑内存延迟，计算处理器的有效 CPI。

答：

该问题的解答使用了前面的式（9-4）和式（9-6）。

$$\begin{aligned}\text{指令缺失的代价} &= \text{I 缓存缺失率} \times \text{读缺失损失}\\ &= (1 - 0.95) \times 20\\ &= 1 \text{ 周期 / 指令}\end{aligned}$$

$$\begin{aligned}\text{数据读缺失的代价} &= \text{程序中内存访问指令的比例} \times\\ &\quad \text{加载占内存访问指令的比例} \times\\ &\quad \text{D 缓存的缺失率} \times \text{读缺失损失}\\ &= 0.3 \times 0.8 \times (1 - 0.98) \times 20\end{aligned}$$

$$
\begin{aligned}
&= 0.096 \text{ 周期/指令} \\
\text{数据写缺失的代价} &= \text{程序中内存访问指令的比列} \times \\
&\quad \text{存储的内存访问指令的比列} \times \\
&\quad \text{D 缓存的缺失率} \times \text{写缺失损失} \\
&= 0.3 \times 0.2 \times (1-0.98) \times 5 \\
&= 0.006 \text{ 周期/指令} \\
\text{有效 CPI} &= \text{基本 CPI} + \text{I 缓存对 CPI 的影响} + \text{D 缓存对 CPU 的影响} \\
&= 1.8 + 1 + 0.096 + 0.006 \\
&= 2.902
\end{aligned}
$$

减少缺失率和减少缺失损失是减少内存延迟的关键，从而增加了流水线处理器的效率。

有两种方法用于降低缺失率，我们将在 9.10 节和 9.11 节中讨论它们。在 9.12 节中，我们将讨论降低缺失损失的方法。

9.10 利用空间局部性提高缓存性能

降低缺失率的第一个方法利用空间局部性原理。基本想法是当内存单元 i 发生缺失时将相邻内存单元中的内容引入缓存。因此，到目前为止，缓存中的每一项都与指令集结构中的一个内存访问单元相对应。为了利用缓存设计中的空间局部性，我们通过内存和缓存之间的内存传输单元的指令对内存访问单元进行去耦合。处理器的指令集体系结构决定了内存访问单元。例如，如果体系结构的指令能够按字节寻址，那么处理器的内存访问单元就是一个字节。另一方面，内存传输单元是分级存储体系的设计参数。特别地，这个参数总是内存访问单元的整数倍，以使利用空间局部性。我们将缓存和内存间的转换单元称为块大小（block size）。当遇到缺失时，缓存会把包含缺失内存访问的整个块大小的块读入内存。

376 ↓ 377

图 9-19 是这种结构下内存状态的例子。在这个例子中，块大小是 4 个字，其中一个字是 CPU 指令内存访问单元。如图所示，一个块的内存地址从块的边界开始。例如，如果 CPU 缺失对内存单元 0x01 数据的访问，那么缓存会引入组成块 0 的 4 个内存字（从地址 0x00 开始），块 0 包含单元 0x01。

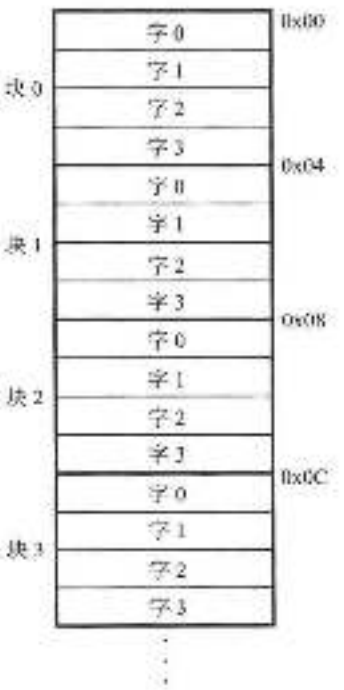


图 9-19 当遇到缺失时，主存按块进行了划分，这些块的位置是连续的，以便使内存和缓存进行信息传递

连续的字组成块，块组成缓存中的项。CPU 产生的地址有如下 3 部分：标记位、索引和块偏移量，如图 9-20 所示。

注意块偏移量是模拟包含多个连续内存单元的缓存块所需要的位数。

例如，如果块大小是 64 字节，那么块偏移量是 6 位。给定块的所有数据都包含在一个缓存项中。标记和索引字段的意义与前面一样。下面介绍图 9-20 中不同字段所需位数的通用计算方法。

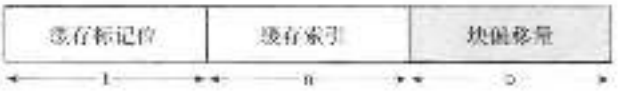


图 9-20 对 CPU 产生的包含多个字的缓存块的内存地址进行翻译

设 a 表示内存地址的位数，S 是缓存按字节计算的总大小，B 是块大小。

图 9-20 中不同字段的计算表达式如下：

$$b = \log_2 B \tag{9-7}$$

$$L = S/B \tag{9-8}$$

$$n = \log_2 L \tag{9-9}$$

$$t = a-(b+n) \tag{9-10}$$

L 是直接映射缓存中的行数，缓存的大小为 S，每个块为 B 字节。图 9-20 中 b 是内存地址的最低有效位；t 是内存地址的最高有效位。n 是内存地址的中间位（见例 9-4 关于这些字段的计算）。

现在我们在多字字块大小的环境中重新回顾基本的缓存算法（查找、读和写）。图 9-21 显示了直接映射缓存的结构。

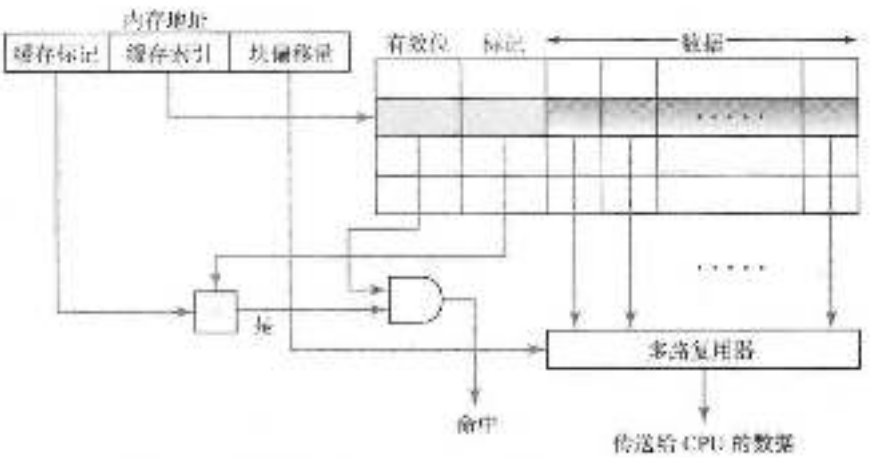


图 9-21 一个多字直接映射缓存的结构。利用多路复用器，块偏移量从选定的块中选择特定的字发送给 CPU。水平阴影部分的块是缓存索引选择的块

1）**查找**：如图 9-20 所示，内存地址的中间部分用于缓存查找。缓存项包含一个数块（如果地址中的缓存标记和特定项中的标记确定它命中）。地址的最低 b 位确定处理器请求的块中的特定字（或字节）。多路复用器利用这 b 位从块中选择特定的字（或字节），并将它发送给 CPU（见图 9-21）。

2）**读**：当读时，缓存取出对应于缓存索引的整个块。如果标记位的比较结果是命中，那么多路复用器选择块中的特定字（或字节），并将它发送给 CPU。如果是缺失，CPU 启动内存的一个块传送。

3）写：我们不得不修改写算法，因为对于整个的缓存行只有一个有效位。与读缺失类似，当遇到写缺失时，CPU 启动内存的一个块传送（见图 9-22）。

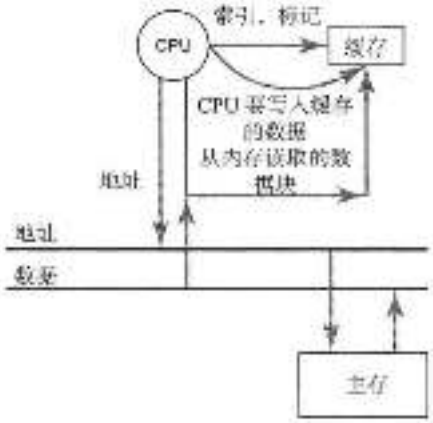


图 9-22 处理写缺失时 CPU、缓存和内存之间的交互。缺失的块首先从内存传送到缓存，然后 CPU 要写入的特定单元在缓冲块中进行更新

378 / 380

当命中时，CPU 将特定的字（或字节）写入缓存。根据写策略，实现会要求额外的元数据（以脏位的形式），并可能采取额外的措施（例如，将修改的字或字节写入内存）来完成写操作（见例 9-4）。

例 9-4 考虑一个 64KB 的多字直接映射缓存。CPU 生成 32 位按字节寻址的内存地址。每个内存字包含 4 字节。块大小是 16 字节。缓存使用每个字有一个脏位的回写策略。缓存对于每个数据块有一个有效位：

a. CPU 如何翻译内存地址？

答：

参考式（9-7），

块大小

$$B = 16 \text{ 字节}$$

所以

$$b = \log_2 16 = 4 \text{ 位}$$

我们需要 4 位（内存地址内位 0 – 3）用于处理块偏移量。

参考式（7-8），缓存行的数目为

$$L = 64\text{KB}/16 \text{ 字节} = 4096$$

参考式（7-9），索引位的数目

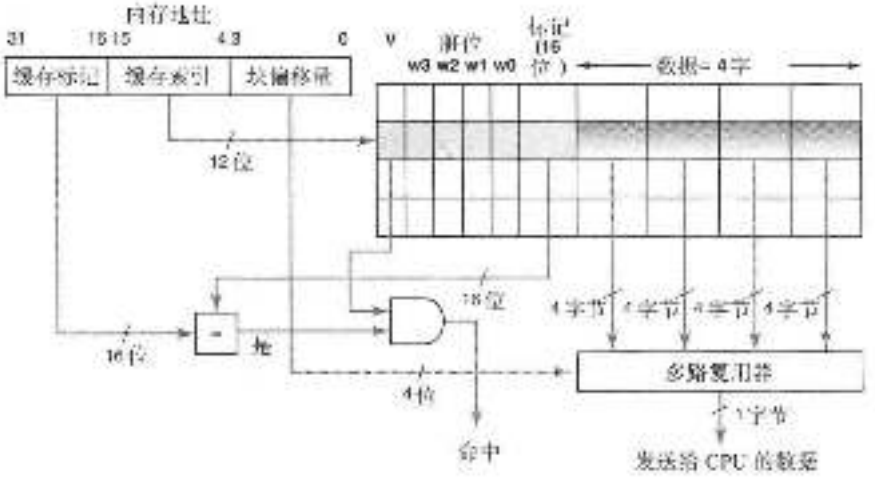
$$n = \log_2 L = \log_2 4096 = 12$$

参考式（7-10），标记位的数目

$$t = a-(n+b) = 32-(12+4) = 16$$

所以，我们需要 12 位（内存地址中为位 4 – 15）用作索引，剩下的 16 位（内存地址为位 16 – 31）用作标记。下图显示了 CPU 对内存地址的翻译：

381



b. 计算实现缓存所需的总的存储大小（即实际数据加上元数据）。

答：

每个缓存行有：

数据	16字节 ×8位/字节 =128位
有效位	1位
脏位	4位（每字有1位）
标记	16位
	149位

缓存的总空间 =149×4096缓存行 = 610 304位

元数据需要的空间 = 总空间 − 实际数据

= 610 304 − 64KB

= 610 304 − 524 288

= 86 016

空间开销 = 元数据/总空间 = 86 016/610 304 = 14%

重复一下，例9-2使用4字节的块，对于同样大小的缓存，空间开销是35%。换句话说，增加块的大小能够降低元数据的需求，因此也减少了空间开销。

382

9.10.1 增加块大小对性能的影响

理解增加块大小对缓存性能的影响很有指导性。增加块大小主要是利用了空间局部性原理。所以，对于给定总容量的缓存，我们希望通过增加块大小来降低缺失率。在极限条件下，我们可以仅用一个缓存块（缓存块的大小为缓存的总容量）。因此，关于块大小，有如下两个问题：

1）缺失率总是能够下降吗？

2）当我们增加块大小时，处理器的性能总能提升吗？

对于第一个问题，答案是不总是的。实际上，当缺失率下降达到某个拐点后会开始上升。我们在第8章中讲的工作集概念是这一现象的原因。回想一下，程序的工作集总是随时间不断变化。缓存块包含连续的内存地址。然而，如果程序的工作集发生了变化，那么大的块尺

寸也会增加缺失率（见图 9-23）。

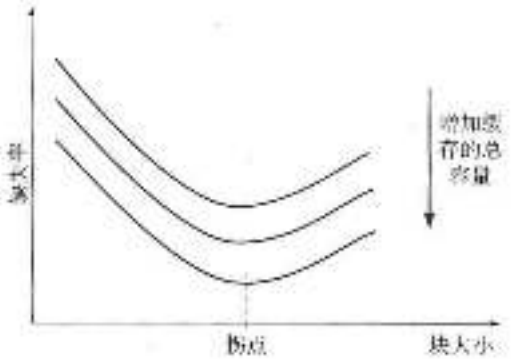


图 9-23 缺失率随块大小变化的曲线。当增加块大小时缺失率会降低，但对于一个特定大小的缓存，我们会达到一个拐点，在拐点之后缺失率会上升

对于第二个问题，答案有点儿复杂。如图 9-23 所示，对于给定的缓存大小，在达到拐点前缺失率确实随着块大小的增加而下降。这可以让处理器有更少的内存延迟，并提高性能。然而，当块大小超过拐点后，处理器会有更多的内存延迟，并且性能会降低。处理器性能的衰退可能出现在比图 9-23 中拐点位置更早的时间。虽然对于给定的缓存大小，缺失率在达到拐点前随着块大小的增加而降低，但是增加的块大小可能会对降低缺失损失带来负面效应。减少程序的执行时间是主要的目标，这一目标主要是靠减少分级存储体系中流水线的延迟来实现。块大小越大，当遇到缺失时花费在从内存将数据传送到缓存的时间开销也越大，这样也增加了内存延迟。我们将简要讨论降低缺失损失的技术。需要注意的是，由于设计参数（块大小和缺失损失）是相互关联的，所以对一个参数进行优化并不总能使整体性能得到提升。换句话说，仅仅着眼于缺失率，并把它作为性能优化的尺度可能会造成缓存设计上的错误。

在流水线处理器设计中（第 5 章），我们理解了单指令延迟和处理器整体吞吐量之间的区别。类似地，在缓存设计中，利用空间局部性原理可以降低后面指令的潜在缺失，而块大小的选择会影响单指令的延迟（导致缺失）和程序整体吞吐量之间的平衡。现实中的一个例子是所得税。每个人要交税（类似延迟），交税减少了个人的财富，但有助于社会整体的建设（类似吞吐量）。当然，平衡好这两者的关系最好。平衡点的位置取决于所采用的策略方法。

现代处理器对于这些问题有了更复杂的处理方法。处理器的微体系结构，即 ISA 的实现细节，是非常复杂同时引人入胜的。只要保持程序的语义，指令就没必要按照程序的顺序执行。缓存缺失不一定会阻碍处理器的运行，这种缓存称为非锁定缓存（lock-up free cache）。这些和其他微观级优化以无法预测的方式彼此相互作用，当然，它们对处理器上的工作负载也很敏感。简单地说，当把缓存块的大小变为原来的两倍时，需要装入内存的数据也会变为原来的两倍。内存系统通常跟不上这种需求。结果是在图 9-23 的平衡点之前性能下降了。

9.11 灵活的布局策略

在直接映射缓存中，内存地址和缓存索引之间有一对一的映射关系。因为这种严格的映射，缓存不能将新的内存单元放在缓存当前未占用的单元中。由于程序的自身特性，这种严格的对应关系降低了性能。图 9-24 说明了直接映射缓存随着程序工作集的改变而表现得很没

有效率。

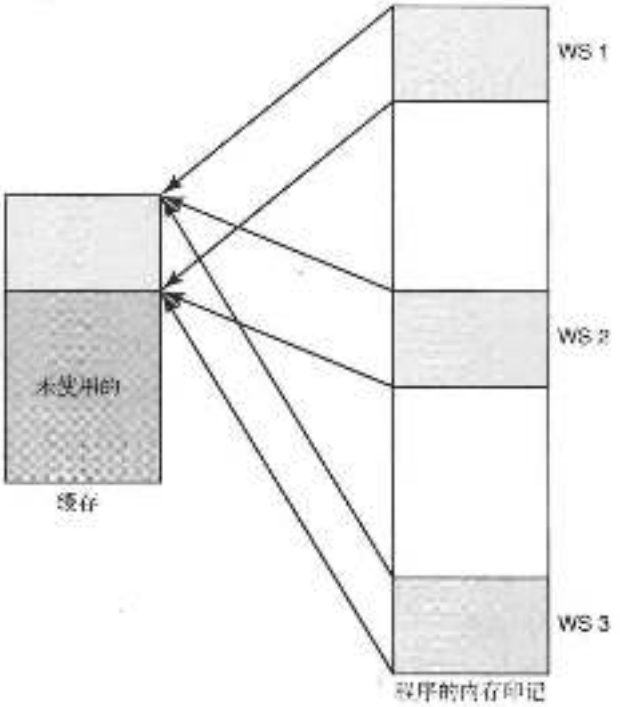


图 9-24 程序的不同工作集占用直接映射缓存的相同部分

在程序执行的过程中，程序频繁地在 3 个工作集（WS1、WS2 和 WS3）之间跳转。这 3 个工作集恰巧映射为图中的同一缓存区域。假设每个工作集都是缓存总大小的 1/3。所以，原则上说，缓存内有足够的空间加载全部 3 个工作集。但由于是严格的映射，缓存中的工作集是一个一个地进行替换，导致性能较低。理想情况下，我们希望程序的全部 3 个工作集能够驻留在缓存中，这样除了必要的一个外就不会有缺失。

让我们来讨论如何实现这一目标。缓存的设计应该考虑程序的局部性会随时间改变。我们将首先讨论一种完全可以避免这种错误的极其灵活的替换策略。

9.11.1 全相关缓存

在这种设置中，没有从内存块到缓存块的唯一映射。因此，缓存块可以存放在任意内存块中。所以，这种结构下的缺失只有强制缺失和容量缺失。缓存翻译 CPU 给出的内存地址，如图 9-25 所示。注意翻译中没有缓存索引。

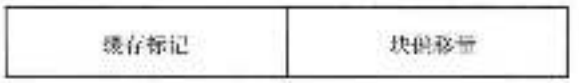


图 9-25 全相关缓存的内存地址翻译

这是由于，如果没有唯一映射关系，那么内存块可以驻留在任何缓存块中。因此，在这种结构下，为了查找，缓存需要搜索所有项来查看内存地址中的缓存标记和任何有效项的标记位是否匹配。一种可能的策略是连续地搜索每个缓存项。从处理器性能的角度看，这是站不住脚的。所以，硬件为每一项增加一个重复的比较器，这样可以并行地对所有的标记位进行比较来查看是否命中（见图 9-26）。

383
，
385

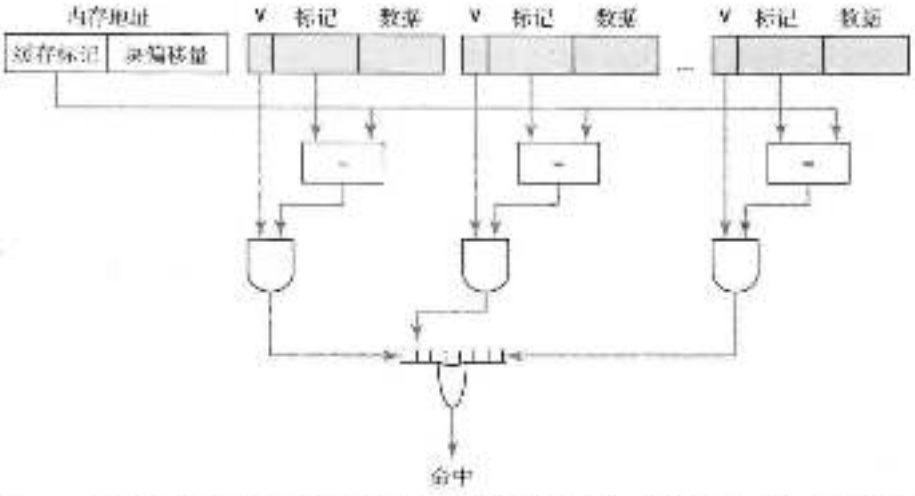


图 9-26 全相关缓存的并行标记匹配硬件。缓存的每块代表一个独立的缓存。标记匹配需要对所有缓存是否命中并行执行。这样，给定的内存块可以放在任何一个缓存块中（图中阴影部分）。当命中时，与内存地址成功匹配的块将数据传送给 CPU

匹配硬件的并行标记的复杂性使全相关缓存不适用于任何合理大小的缓存。乍一看，由于它的灵活性，对于给定的工作负载和缓存大小，全相关缓存似乎可以更好地降低缺失率，但事实并不是这样的。缓存，作为一种宝贵的高速资源，大多数情况下的利用率接近饱和。这样，缓存中的缺失不可避免地导致将缓存中一些已有的东西替换出去。如何选择被替换项对缓存缺失率有巨大的影响（因为我们不知道未来 CPU 会访问哪些内存地址），也会对被替换的全相关缓存中缺失行的灵活性产生不利影响。我们在后面会简要地讨论缓存替换策略的细节（见 9.14 节）。可以肯定地说，在这一点上，由于上述原因，除非在非常特殊的环境中，我们很少在实际中使用全相关缓存。第 8 章讲的旁路转换缓冲器（TLB）由于容量较小，是全相关的结构。全相关的名字来源于内存块可以和任何一个缓存块相关联。 386

9.11.2 组相关缓存

组相关缓存是直接映射和全相关的折中。这种结构的名字来源于一个内存块可以与一组缓存块相关联。例如，2 路组相关缓存给缓存中的每个内存块 2 个可能存放的位置。类似地，4 路组相关缓存给缓存中的每个内存块 4 个可能的存放位置。相关程度（degree of associativity）定义为给定内存块在缓存中拥有的存放位置数。2 路组相关缓存的相关程度是 2，4 路组相关缓存的相关程度是 4，以此类推。

关于组相关缓存，一种简单的理解方式是将它想象成多个直接映射缓存。为了使讨论更具体，考虑一个 16 个数据块的缓存。我们可以将这 16 个数据块组织成直接映射缓存（见图 9-27a），也可以组织成 2 路组相关缓存（见图 9-27b），或者 4 路组相关缓存（见图 9-27c）。当然，每个数据块都有相关的元数据。

在直接映射缓存中，给定索引（如 3），缓存中就会有一个位置对应于该索引值。2 路组相关缓存在缓存中有两个位置（图 9-27b 中的阴影）对应于同一个索引。4 路组相关缓存有 4 个位置（见图 9-27c 中的阴影）对应于同一个索引。对于 16 个数据块的缓存，第一种结构需要 4 位的索引来对缓存进行查找，而第二种结构需要 3 位的索引，第三种需要 2 位。给定一

个内存地址，缓存同时查找所有可能的位置，也就是与地址中索引值相匹配的组。组相关缓存所需要的标记匹配硬件的数目等于相关程度。

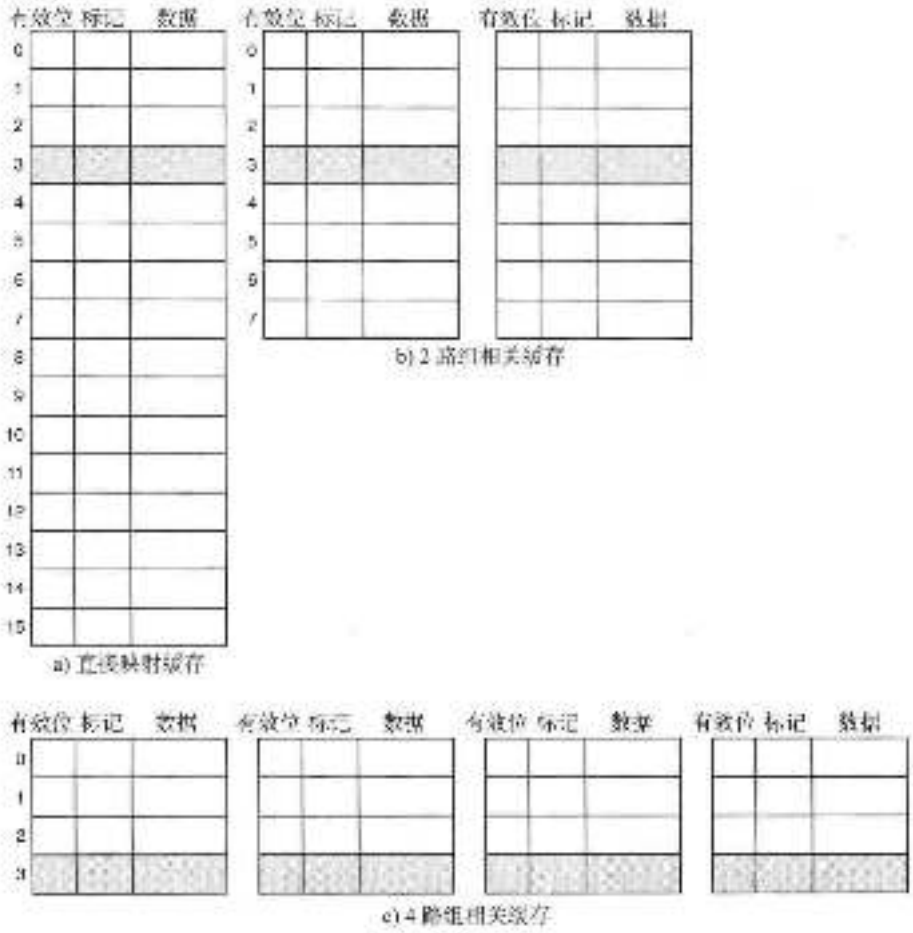


图 9-27 16 个缓存数据块的 3 种不同结构。根据相关程度不同，给定的内存块可以存放在不同的缓存块中。上面的 3 个图中的阴影部分表示给定内存块所能存放的位置

图 9-28 是块大小为 4 字节的 4 路组相关缓存的完整结构。

缓存将从 CPU 得到的内存地址转换成标记、索引和块偏移量，类似于直接映射的结构（见图 9-29）。

让我们来讨论在缓存查找中如何将内存地址分解为索引和标记位。对于直接映射结构，缓存的总大小决定了索引位的数量，即 $\log_2(S/B)$，S 是缓存大小，B 是块大小，都以字节为单位。对于总缓存大小相同的组相关缓存，索引位的位数为 $\log_2(S/pB)$，p 是相关联程度。例如，对于总缓存大小为 16 个数据块的 4 路组相关缓存，要求的位数为 $\log_2(16/4)=2$ 位（见图 9-27c）。

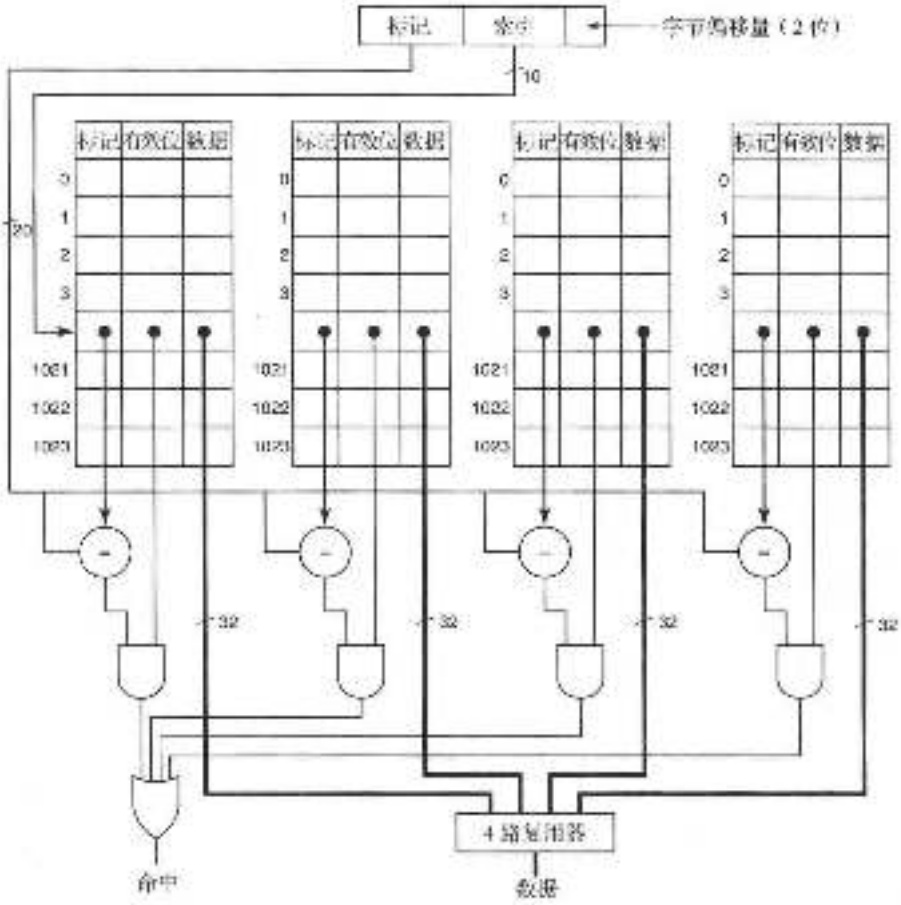


图 9-28 4路组相关缓存结构



图 9-29 翻译 CPU 产生的内存地址对组相关缓存和直接映射缓存没有区别

9.11.3 组相关的极端情况

直接映射和全相关缓存都是组相关缓存的特殊情况。考虑缓存的总大小为 S 字节，块大小为 B 字节，缓存中数据块的数目为 N，则 $N=S/B$。我们将这些数据块组织成 p 个并行缓存，称为 p 路组相关缓存。缓存有 N/p 个缓存行（或称为组[⊖]），每个缓存行有 p 个数据块。缓存需要 p 个并行硬件用于标记比较。

如果 $p=1$ 会是什么情况？在这种情况下，结构变为直接映射缓存。缓存有 N 个缓存行，每个组有 1 块。

如果 $p=N$ 会发生什么呢？这种情况下，结构会变为全相关缓存。缓存只有 1 个有 N 块的缓存行。

⊖ 所以目前为止我们有 4 个术语指的是同一件事：缓存行（cache line）、缓存块（cache block）、缓存项（cache entry）和组（set）。

有了以上讨论，我们重新来看式（9-7）～式（9-10）：

总缓存大小 $=S$ 字节

块大小 $=B$ 字节

内存地址 $=a$ 位

相关程度 $=p$

缓存中的总数据块数为

$$N=S/B \tag{9-11}$$

我们将计算缓存行的式（9-8）用下式替换：

$$L=S/pB=N/p \tag{9-12}$$

式（9-8）是当 $p=1$ 时式（9-12）的特殊情况。索引位的数目 n 由式 $\log L$ 得到（见式（9-9））。

正如前文所说，全相关缓存主要用于 TLB。相关程度通常由分级存储体系中的缓存级别决定。直接映射缓存或2路组相关缓存是 L1 级缓存（离 CPU 较近）的典型应用。更深层的分级存储体系也会有更高的相关程度。我们在 9.19 节会重新来看看这些问题。

例 9-5 考虑一个数据大小为 64KB 的 4 路组相关缓存。CPU 生成 32 位按字节寻址的内存地址。每个内存字包含 4 字节。块大小是 16 字节。缓存采取直写策略。每个数据块有一个有效位。

387～390

a. CPU 如何解释内存地址？

b. 计算实现缓存的总的存储大小（即实际数据加上元数据）。

答：

a. 内存地址为位数

$$a=32 \text{ 位}$$

因为块大小是 16 字节，所以使用式（9-7），块偏移量

$$b=4 \text{ 位（内存地址中的位 0 ～ 3）}$$

因为是 4 路组相关缓存（见图 9-27c），

$$p=4$$

缓存行的数目（见式（9-12））

$$L=S/Pb=64\text{KB}/(4\times16) \text{ 字节}=1\text{K}$$

索引位的数目（见式（9-9））

$$n=\log L=\log_2 1024=10 \text{ 位}$$

标记位的数目（见式（9-10））

$$t=a-(n+b)=32-(10+4)=18 \text{ 位}$$

所以，内存地址最高位为 31，最低位为 0，则

标记位	18 位（31 ～ 14）
索引位	10 位（13 ～ 4）
块偏移量	4 位（3 ～ 0）

b. 4 个并行缓存的每一个中的块（或者加元数据）包含：

数据：16×8 位	128 位（每个缓存块有 16 位）
有效位	1 位（每块为 1 位）

标记　　-18 位

总位数　　-147 位

每个缓存行有 4 个这样的块 -147×4=588 位。

在整个缓存中有 1K 个这样的缓存行，缓存的总大小 -

588 位 / 缓存行 ×1024 缓存行 = 602 112 位

例 9-6 考虑一个 4 路组相关缓存。

- 缓存的总数据大小 =256KB。
- CPU 生成 32 位按字节寻址的内存地址。
- 每个元素字由 4 字节组成。
- 缓存块的大小为 32 字节。
- 每个缓存行有一个有效位。
- 缓存采取回写策略，每字有一个脏位。

a. CPU 如何翻译内存地址。（哪些位用作缓存索引，哪些位用作标记，哪些位用作块偏移量？）

b. 计算缓存的总大小（即数据加上元数据）。

答：

a. 和例 9-5 类似，

标记	16 位（31 - 16）
索引	11 位（15 - 5）
块偏移量	5 位（0 - 4）

b. 四个并行缓存中的块包含如下：

数据：32×8 位	=256 位（每个缓存块 32 字节）
有效位：	=1 位（每块有 1 个有效位）
脏位：8×1 位	=8 位（每字有 1 个脏位）
标记	=16 位
总位数	=281 位

每个缓存行包含 4 个这样的块 =281×4=1124 位。

整个缓存中有 2K 个这样的缓存行，缓存的总大小 -

1124 位 / 缓存行 ×2048 缓存行 =281KB

9.12 指令和数据缓存

在处理器流水线中（见图 9-15），我们介绍了两个缓存：一个在 IF 阶段，一个在 MEM 阶段。表面上看，前一个用于处理指令，后一个用于处理数据。有些程序可能需要大的指令缓存，而另一些程序可能需要大的数据缓存。

我们很想把这两个缓存合并为一个大的、统一的缓存。当然，对于给定大小的缓存，当不考虑是数据还是指令模式的访问时会增加命中率。

然而，合并也有一个缺点。我们知道 IF 阶段在每个时钟周期都要访问 I 缓存，而 D 缓存只对内存访问指令时（加载 / 存储）起作用。采用统一的缓存可能会导致结构上的险象并降低流水线的性能。实证研究说明统一的缓存产生的结构上的险象所带来的不利影响降低了整个

392 流水线的性能，尽管提高了命中率。

有一些硬件技术（例如，多个读端口的缓存）用于避开将I缓存和D缓存合并所带来的不利影响。然而，这些技术增加了处理器的复杂性，这反过来会影响流水线的时钟周期。回想一下，缓存是为了缓解访问速度和缺失率之间的关系而产生的。正如我们在开始所提及的（见9.4节），L1缓存设计的主要目的是为了将命中时间与处理器时钟周期相匹配。这表明避免L1缓存不必要的设计复杂性是为了保持较低的命中时间。另外，由于对指令和数据的访问方式不同，所以I缓存和D缓存在设计上考虑的因素（如相关程度）也不一样。除此之外，由于芯片密度的提高，现在可以设计独立的有足够空间的I缓存和D缓存来弥补将缓存分开所带来的缺失率的开销。最后，I缓存不用支持写操作，这会让I缓存更简单、速度更快。由于这些原因，通常芯片上的I缓存和D缓存是分开的。然而，由于L2缓存设计的主要目的是降低缺失率，所以通常会有一个统一的L2缓存。

9.13 降低缺失损失

缺失损失是当发生缺失时数据从内存传输到缓存的服务时间。正如我们前面所观察到的，读操作和写操作损失是不一样的，并且损失由数据通路上的其他硬件确定，例如写缓冲区，允许与内存传输时同时进行计算的写缓冲区。

通常，主存系统的设计都要考虑缓存的结构。特别是，要支持向/从CPU填充缓存的块传输。连接主存和CPU的内存总线在确定缺失损失方面起着关键作用。处理器和内存之间每次数据传输所需要的时间，称为总线周期时间。处理器和内存之间每个时钟周期传输的数据量称为内存带宽（memory bandwidth）。内存带宽用于度量处理器和内存之间传递信息的吞吐量。带宽由处理器和内存之间的数据线的数目决定。根据总线的位宽限制，内存系统可能需要多个总线周期来传递一个缓存块。例如，如果块大小是4个字，内存总线位宽只有一个字，那么需要4个总线周期来完成块传输。作为一阶近似，我们可以将缺失损失定义为从内存传输一个缓存块到缓存的总时间（用CPU时钟周期度量）。然而，对于一个单独的缺失，处理器所经历的实际延迟可能比块传输时间短。这是因为内存系统可以先为处理器访存缺失提供特定的数据，再传输包含缺失访问的内存块的余下部分。

尽管在内存系统中支持这样的块传输，但当块大小超过一定值时会有其他不利影响。例
393 如，如果处理器在单元x处发生读缺失，缓存子系统读入包含x的整个块，那么可能首先让处理器服务x。根据处理器和内存之间带宽的不同，在接下来内存系统可能会花费多个总线周期来传输块的其他部分。同时，处理器可能会在不同缓存块的另一个内存单元y引入第二个缺失。现在，因为系统忙于完成在x上发生缺失时的块传输，所以内存系统不能立即处理第二个缺失。这就是我们在9.10节观察到的，不能只将缺失率作为设计缓存块大小的参考指标的原因。这是每个计算机子系统设计时都会遇到的关于延迟和吞吐量之间关系的经典问题。我们在第5章介绍了这一内容，现在我们在内存系统中来看看这部分内容，在第13章我们将从网络的角度重新审视这部分内容。

9.14 缓存替换策略

在直接映射缓存中，替换策略提前确定了被替换项。因此，这是没有办法选择的。

在组相关或全相关缓存中，可以选择被替换的项。利用时间局部性原理，我们建议采用LRU策略。对于全相关缓存，缓存对所有的块用LRU策略选择被替换项。对于组相关缓存，

选作被替换的项局限于能够加载当前缺失内存访问的组。

为了记录 LRU 信息，缓存需要额外的元数据。图 9-30 显示了记录 2 路组相关缓存的 LRU 信息的硬件结构。每组（或缓存行）都有一个与它相关的 LRU 位。



图 9-30 在 2 路组相关缓存中每组有 1 位 LRU。对于给定的行，相关的 LRU 位说明哪个缓存（C0 或 C1）最近被访问

对于每次访问，硬件都对当前访问的内存块所在组的 LRU 位进行设置。假设对于给定组的两个块都有有效位，那么硬件根据缓存 C0 还是 C1 中的访问命中来设置 LRU 位为 0 或 1。 394
如果 LRU 位为 1，则替换 C0 中的块；如果 LRU 位为 0 则替换 C1。

2 路组相关缓存需要的硬件很少，但因为每次内存访问（影响 IF 和 MEM 阶段的流水设计）都要更新 LRU 位，所以也有时间损失。

用于更高相关联程度的 LRU 硬件会变得更加复杂。假设我们将 4 个并行缓存标记为 C0、C1、C2 和 C3，如图 9-31 所示。基于 2 路组相关缓存，每组有一个 2 位字段。这 2 位字段给出最近访问的块。不幸的是，它告诉我们最近访问的块，但它没有告诉我们组中哪块最近很少访问。对于每组，我们真正需要的是一个次序矢量，如图 9-31 所示。例如，组 S0 的次序列表示 C2 最不常用，C1 最常用。也就是说，S0 中的块按访问时间次序降序排列为：C1，C3，C0，C2。这样，此时，如果需要替换 S0 中的块，根据 LRU 可判断替换 C2 中的内存块。每次访问 CPU 都会更新当前访问组的次序。图 9-32 显示了进行一系列映射到组 S0 的内存访问时 LRU 次序向量的变化情况。每行表示根据当前访问的块，被替换项是如何变化的。

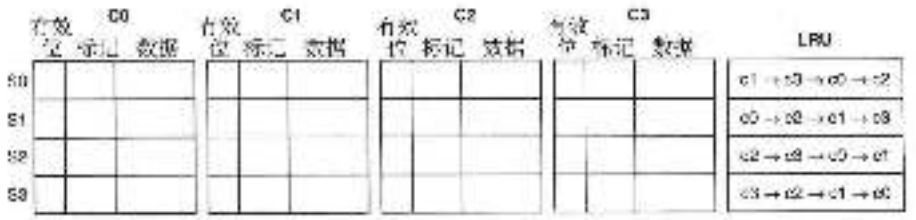


图 9-31 4 路组相关缓存的 LRU 信息

在硬件上应用该方案会怎么样？4 个并行缓存的访问次序矢量可能有 4！=24 个。所以，我们需要 5 位计数器来对次序矢量的 24 种可能情况进行编码。8 路组相关缓存需要计数器的

位数史大以便编码 8 阶乘的状态。每个组都需要维护一个能够记录状态变化的与有限状态机相关的计数器，如图 9-32 所示。我们可以看到应用这种策略的硬件复杂度会随着关联程度和缓存中行数的增多而加大。还有其他的与真正的 LRU 方案近似但更为简单的编码方式。对于很多真实的程序序列，我们有足够的经验表明低复杂度的替换策略可能事实上比真正的 LRU 表现得更好（即导致更低的缺失率）。

	至 S0 的 LRU	
访问 C1	c1 → c3 → c0 → c2	被替换项：当前在 C2 中的块
访问 C2	c2 → c1 → c3 → c0	被替换项：当前在 C0 中的块
访问 C3	c3 → c2 → c1 → c0	被替换项：当前在 C0 中的块
访问 C0	c0 → c3 → c2 → c1	被替换项：当前在 C1 中的块

图 9-32 当遇到一系列映射到组 S0 的访问序列时，LRU 矢量的变化

9.15 缺失类型简要说明

我们定义了 3 种类型的缓存缺失：强制缺失（compulsory）、容量缺失（capacity）和冲突缺失（conflict）。正如其名，强制缺失是由于程序在执行中第一次访问给定内存单元造成的。通常该单元不在缓存中，缺失是不可避免的。我们用发动机是冷是热进行类比（在启动时），将这样的缺失称为冷缺失（cold miss）。

另一方面，考虑这一情况，CPU 访问的内存单元 X 本来是在缓存中的，但现在不在了，发生了缺失[⊖]。这可能有两个原因：在发生缺失时缓存是满的，所以不得不腾出一些空间给 X。这就是称为容量缺失的原因。或者可以想象缓存不满，但映射策略将 X 引入当前被其他内存单元占用的缓存行，这就是所谓的冲突缺失。由定义我们知道，在全相关缓存中不会出现冲突缺失，因为内存单元可以放在任何位置。所以，在全相关缓存中的缺失类型只有强制缺失和容量缺失。

有时候，我们很难对缺失进行分类。在全相关缓存中，假设 CPU 第一次访问内存单元 X，此时缓存也是满的，而这时缓存在 X 处发生缺失，这时是容量缺失还是强制缺失？我们可以说都是。所以，将这个缺失归为强制缺失或容量缺失或两者都是。

注意容量缺失可能出现在直接映射缓存、组相关缓存或全相关缓存中。例如，考虑有 4 个缓存行的直接映射缓存。缓存初始化为空，每个缓存行仅保留一个内存字。CPU 访问如下的内存访问序列：

395
/
396

0, 4, 0, 1, 2, 3, 5, 4

我们将前述地址的内存字表示为 m0，m1，…，m5。第一次访问（m0）是强制缺失。第二次访问（m4）也是强制缺失，并且由于直接映射结构它将导致 m0 被替换出缓存。再次访问 m0，也会出现缺失。因为在引入 m4 时已经将 m0 换出，这是典型的冲突缺失。尽管缓存有其他空闲行但还是将 m4 替换为 m0。继续访问，m1、m2 和 m3 均导致强制缺失。

之后我们访问内存单元 m5。这在之前的缓存中没有出现过，所以 m5 导致的缺失应该是强制缺失。然而，缓存此时已经有 m0、m1、m2 和 m3，是满的，所以我们也可以称之为容

⊖ 我们不知道 X 为什么最初会被换出，但从分析当前缓存缺失的角度来看，这一点并不重要。

强缺失。所以我们可以称这种缺失为强制缺失或容量缺失或两者都是。

最后，我们又访问内存单元 m4。这可以称为冲突缺失，因为 m4 之前在缓存中被引用过。然而，此时缓存被 m0，m5，m2 和 m3 填满，所以我们也可以称它为容量缺失。即我们可以称这种缺失为冲突缺失或容量缺失或两者均是。

强制缺失是不可避免的。所以，这种缺失支配着其他类型的缺失。换句话说，如果缺失可以分为强制缺失或其他缺失，我们将它归为强制缺失。当缓存满时，独立于结构，我们引入一个当前不在缓存中的内存单元上的缺失。换句话说，如果缺失可以被归为冲突缺失或容量缺失，我们选择容量缺失。

例 5-7 假设下列情况：

- 在 2 路组相关缓存中，块的总数 = 8。
- 采用 LRU 替换策略。

C1	C2
0	
1	
2	
3	

处理器以下列顺序访问内存单元 18 次：

0, 1, 8, 0, 1, 16, 8, 8, 0, 5, 2, 1, 10, 3, 11, 10, 16, 8

对于给定的 2 路组相关缓存，用表格的方式给出将占用内存单元的缓存；占用的具体缓存索引和缺失类型（冷 / 强制缺失、容量缺失、冲突缺失）。 397

内存单元	C1	C2	命中 / 缺失	缺失类型

注意：

- 缓存初始为空。
- 发生缺失时，如果两个位置（C1 和 C2）均为空，则将缺失内存单元放入 C1。
- 缺失为容量缺失或冲突缺失时，将缺失的类型归为容量缺失。
- 缺失为冷 / 强制缺失或容量缺失时，将缺失的类型归为冷 / 强制缺失。

答：

内存单元	C1	C2	命中 / 缺失	缺失类型
0	索引 = 0		缺失	冷 / 强制缺失
1	索引 = 1		缺失	冷 / 强制缺失
8		索引 = 0	缺失	冷 / 强制缺失
0	索引 = 0		命中	
1	索引 = 1		命中	
16		索引 = 0	缺失	冷 / 强制缺失
8	索引 = 0		缺失	冲突缺失
8	索引 = 0		命中	
0		索引 = 0	缺失	冲突缺失
5		索引 = 1	缺失	冷 / 强制缺失
2	索引 = 2		缺失	冷 / 强制缺失

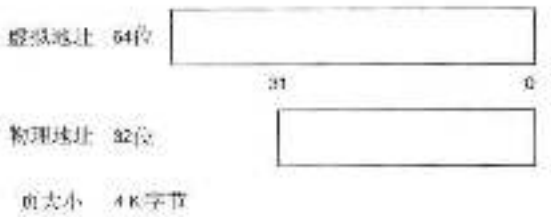
（续）

内存单元	C1	C2	命中/缺失	缺失类型
1	索引=1		命中	
10		索引=2	缺失	冷/强制缺失
3	索引=3		缺失	冷/强制缺失
11		索引=3	缺失	冷/强制缺失
10		索引=2	命中	
16	索引=0		缺失	容量缺失
398 8		索引=0	缺失	容量缺失

9.16 TLB 和缓存整合

在第 8 章中，我们引入了用来保存地址信息的 TLB 概念。对于给定的虚页号（VPN），如果 TLB 中有对应的物理页帧号（PFN），则返回此物理帧号。出于访问速度的考虑，TLB 通常非常小，但虚页的空间非常大。和处理器缓存类似，对于给定 VPN，需要查询 TLB 的映射函数。设计 TLB 时的考虑因素与处理器缓存设计时类似，即 TLB 可以被组织成直接映射或组相关的结构。根据结构，为了使于 TLB 查找，将 VPN 划分为标记字段和索引字段。下面的例子说明了这一点。

例 9-8 假设：



有 512 项的直接映射 TLB。

a. TLB 中每项的标记字段有多少位？

b. TLB 中需要多少个比特存储页帧号？

答：

a. 页大小为 4KB，页偏移量的位数 = 12。

所以，VPN 所需的位数 = 64 − 12 = 52。

查找一个容量为 512 的直接映射缓存所需的索引位数 =9。

所以，TLB 中标记的位数 = 52 − 9 = 43 位。

b. TLB 中保存 PFN 所需的位数等于 PFN 的大小。

页大小为 4KB，那么 PFN 为 32 − 12 = 20 位。

现在我们将 TLB 和分级存储体系放在一起得到一个整体的结构图。图 9-33 显示了如下 CPU 访问内存的路径（IF 阶段或 MEM 阶段）：

- CPU（在流水线的 IF 或 MEM 阶段）生成虚拟地址（VA）。
- TLB 完成虚拟地址到物理地址（PA）的转换。如果在 TLB 中命中，那么流水线不停顿地继续执行。如果发生缺失，流水线暂停直到处理完缺失。
- 该阶段使用 PA 来查找缓存（I 缓存或 D 缓存）。如果在缓存中命中，那么流水线不停顿

地继续工作。如果发生缺失，流水线暂停直到处理完缺失。

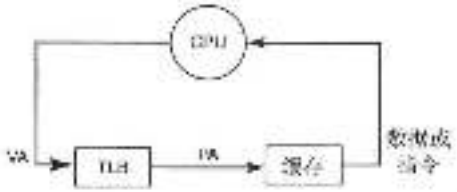


图 9.33 内存访问的路径。在缓存查询开始前，CPU 虚拟地址需要先通过 TLB 查找转换成物理地址

注意流水线的 IF 和 MEM 阶段也可能同时访问 TLB。出于这一点考虑，大多数处理器将 TLB 划分成指令部分和数据部分（I-TLB 和 D-TLB），所以两个地址转换能够并行进行。如图 9-33 所示，TLB 在处理器时钟周期中起着关键作用，因为每次内存访问都需要先经过 TLB 然后再经过缓存。所以，TLB 很小，对于 I-TLB 或 D-TLB 通常仅有 64 ~ 256 项。

9.17 缓存控制器

缓存控制器是处理器与缓存内部和内存系统其他部分的硬件接口，具有如下的功能：

- 当处理器发出请求时，缓存控制器查询缓存以确定是否命中，当命中时将数据传给处理器。
- 当发生缺失时，它初始化总线事务以便从更高级的分级存储体系中读取缺失的块。
- 根据内存总线的细节设计，请求的数据块相对于该请求可能会异步到达。在这种情况下，缓存控制器接收数据块并将它保存在缓存的合适位置。
- 我们将在下一章看到，控制器给处理器提供详细说明内存的哪些区域是“不可缓存”的能力。当我们处理 I/O 设备到处理器的接口时，这种需求变得很明显（见第 10 章）。

399
|
400

例 9-9 考虑如下两分级存储体系：

- 将 128 项的全相关 TLB 分为两部分：一部分用于用户进程，另一部分用于内核。TLB 每个时钟周期允许访问一次。TLB 的命中率为 95%。缺失发生时会访主存以便完成地址转换。
- L1 缓存有 1 个周期访问时间，命中率为 99%。
- L2 缓存有 4 个周期访问时间，命中率为 90%。
- L3 缓存有 10 个周期访问时间，命中率为 70%。
- 物理内存有 100 个周期访问时间。

计算分级存储体系的平均内存访问时间。注意页表项本身也可能在缓存中。

答：

回顾 9.4 节的公式：

$EMAT_i = T_i + m_i \times EMAT_{i+1}$

$EMAT_{物理内存} = 100$ 周期

$EMAT_{L1} = 10 + (1 - 0.7) \times 100 = 40$ 周期

$EMAT_{L2} = (4) + (1 - 0.9) \times (40) = 8$ 周期

$EMAT_{L3} = (1) + (1 - 0.99) \times (8) = 1.08$ 周期

$EMAT_{TLB} = (1) + (1 - 0.95) \times (1.08) = 1.054$ 周期

$EMAT_{内存访问} = EMAT_{TLB} + EMAT_{L1} = 1.054 + 1.08 = 2.134$ 周期

9.18 虚拟索引物理标记的缓存

如图 9-33 所示，我们看到每次内存访问都会查找 TLB，然后查找缓存。TLB 有助于避免在主存中进行地址转换。然而，TLB 查询在 CPU 路径的关键位置上。这意味着虚拟地址到
401 物理地址的转换在时间上要先于在缓存中查找内存单元对应的数据。换句话说，由于缓存查询在 TLB 查询之后，所以在 CPU 得到缓存中的内存访问是否命中前会有很明显的时延。我们希望能够并行进行 TLB 的地址转换和缓存中的查找。换句话说，我们不希望地址转换“打扰”CPU 访问缓存。即我们希望绕过 TLB 来获取 CPU 地址并查找缓存。起初，这似乎不可能实现，因为我们需要用物理地址来查找缓存。

让我们重新看看图 9-34 中的虚拟地址。地址转换完成从 VPN 到 PFN 的改变。然而，虚拟地址的页偏移量部分是不变的。



图 9-34 虚拟地址

直接映射缓存或组相关缓存用物理地址的最低有效位作为查找的索引（见图 9-11）。

这给我们启示，如果我们从虚拟地址的不变化部分（即页偏移量部分）得出缓存索引，那么我们就可以并行进行缓存查找和 TLB 查找。我们将这样的结构称为虚拟索引的物理标记的（virtually indexed physically tagged）缓存（见图 9-35）。缓存使用虚拟地址中的索引，但标记位从物理地址获得。

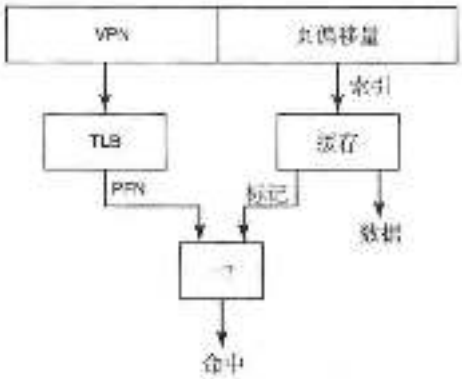


图 9-35 虚拟索引的物理标记的缓存。使用在地址转换时保持不变的虚拟地址中的位（页偏移量），缓存查找可以与 TLB 查找并行执行

如果再多思考一下，不难看出这种策略的局限性。虚拟地址的不变化部分限制了缓存的大小。例如，如果页大小为 8KB，那么缓存索引的位数最多是 13 位，通常会更小，因为最低有效位会指定块偏移址。尽管有这些限制，但增加组相关程度可以增加缓存大小。然而，因
402 为增加相关程度也增加了硬件设计的复杂度，所以增加相关程度也是有限制的。

软件和硬件之间的合作关系有助于减轻这种限制。尽管硬件实现地址转换，但内存管理器是建立 VPN 到 PFN 映射的软件实体。通过仔细选择转换进程和 VPN 到 PFN 映射，内存管理器使用称为页面着色（page coloring）的技术来保证虚拟地址的更多位保持不变（见例 9-7）。页面着色允许处理器使用更大的、虚拟索引的和物理标记位的缓存，并独立于页面大小。

另一种解决地址转换问题的方法是使用虚拟标记的（virtually tagged）缓存。在这种情况下，缓存使用虚拟索引和标记。读者需要考虑遇到的挑战，例如结构等。关于这种缓存的讨

论超出了本书的范围。[⊖]

例9-10 考虑虚拟索引的和物理标记的缓存：

1）虚拟地址是32位。

2）页面大小为8KB。

3）缓存的特性如下：

- 4路组相关缓存。
- 总大小＝512KB。
- 块大小＝64字节。

内存管理器使用页面着色来获得大的缓存大小。

1）对于分级存储体系，虚拟地址需要多少位保持不变？

2）用图描述地址转换和缓存查找，标记其中使用的虚拟地址和物理地址。

答：

页面大小8KB表明页偏移量是13位，剩余19位是VPN。

缓存中：4块/组×64字节/块＝256字节/组，且

512KB缓存总容量/256字节/组＝2K组，即索引需要11位。

所以缓存的内存访问分解为：

标记，15位；索引，11位；偏移量，6位。

现在内存管理软件需要以如下方式将地址分配给页面：VPN最低的4位必须和PFN最低的4位相等。 403



9.19 缓存设计因素概述

目前我们已经介绍了很多概念，在讨论主存前很有必要对这些概念进行一一列举：

1）时间和空间局部性原理（9.2节）。

2）命中、缺失、命中率、缺失率、周期时间、命中时间、缺失损失（9.3节）。

3）多级缓存及其设计考虑因素（9.4节）。

4）直接映射缓存（9.6节）。

5）缓存读/写算法（9.8节）。

6）空间局部性和块大小（9.10节）。

7）全相关和组相关缓存（9.11节）。

8）I缓存和D缓存考虑因素（9.12节）。

9）缓存替换策略（9.14节）。

10）缺失类型（9.15节）。

11）TLB和缓存（9.16节）。

12）缓存控制器（9.17节）。

13）虚拟索引的和物理标记的缓存（9.18节）。

⊖ 见高级计算机体系结构关于这部分的更深入探讨（例如[Hennessy, 2006]）。

现代处理器有片上 TLB、L1 和 L2 缓存。TLB 和 L1 设计时的主要考虑因素是减少命中时间，这两者设计时主要的考虑因素是一致的。TLB 通常是一个小的用于地址转换的全相关缓存，大概有 64 ~ 256 个缓存项。TLB 通常分为系统部分（用于保留上下文切换）和用户部分（当上下文切换时刷新用户部分）。有些处理器在 TLB 项中提供进程标志来避免上下文切换时的刷新。L1 缓存用于优化访问速度，通常，它分为 I 缓存和 D 缓存，具有很小的相关程度（通常为 2），并且与更高级的分级存储体系相比大小更小（I 缓存和 D 缓存在 2008 年前后大小通常都小于 64KB）。L2 缓存设计的主要目的是为了减少缺失率，通常它的 I 缓存和 D 缓存是一体的，并且相关程度更大（4 路和 8 路比较常见，也有 16 路的）。为了降低缺失率，L2 缓存的块大小可以比 L1 大。2008 年前后处理器设计中 L2 缓存大小为几百 KB 到几 MB 之间。大多数的现代处理器提供片外 L3 缓存，设计时的考虑重点与 L2 类似，但容量更大（2008 年
404 前后处理器中约为几十 MB）。

9.20 主存的设计因素

处理器内存总线的设计和物理内存的结构在分级存储体系的性能中扮演着重要角色。正如我们前面提到的，与 CPU 相比，应用 DRAM 技术的物理内存有大约 100:1 的速度差。当发生缓存缺失时这种设计可能要多次访问物理内存，取决于处理器内存总线的宽度和缓存块的大小。

本书希望能带动读者一起去发现这些有趣的概念，我们从主存系统设计中一些非常简单的设计思想讨起。一开始，我们希望读者理解这些思想远没有今天我们看到的计算机里的内存系统复杂，之后我们将进行关于现代上流内存系统设计的讨论。

首先，我们考虑 3 种不同的内存总线结构和相应的缺失损失。为了讨论方便，我们假设 CPU 产生 32 位的地址和数据；缓存块大小是 4 个字，每个字 32 位。

9.20.1 简单的主存

图 9-36 展示了简单的内存系统的结构。当发生缓存缺失时它发送块读请求。CPU 简单地将块地址发给物理内存。物理内存在内部对块的连续地址进行计算，从 DRAM 中获取相关的字，并将它们一个一个地发送给 CPU。

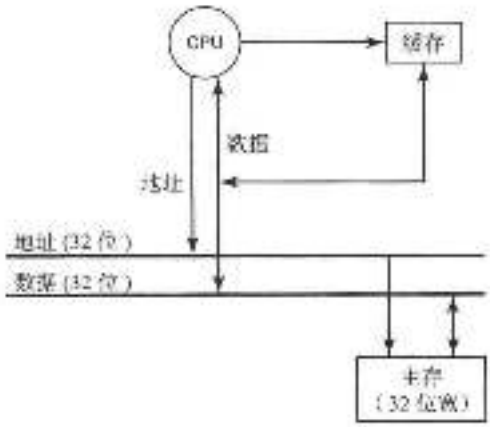


图 9-36 简单的内存系统。CPU 的最大内存访问单元和内存的传输单元是一样的（都是
405 32 位的数据）

例 9-11 假设 DRAM 的访问时间是 70 个周期。CPU 与内存之间地址或数据的总线周期时间是 4 个周期。计算块大小为 4 个字的块传输时间。假设在数据传输给 CPU 之前这 4 个字是每一次从 DRAM 中获取。

答：

从 CPU 到内存的地址传输时间 = 4 个周期。

DRAM 访问时间 = 70×4 = 280 个周期（4 个连续字）。

从内存到 CPU 的数据传输时间 = 4×4 = 16 个周期（4 个字）

块的总传输时间 = 300 个周期。

9.20.2 与缓存块大小相匹配的主存和总线

为了降低缺失损失，我们将处理器内存总线和物理内存与块大小进行匹配。图 9-37 展示了该结构。这种结构用单总线周期将块从内存传输到 CPU，并且对 DRAM 只进行一次访问。块的全部 4 个字构成 DRAM 中的一行，这样可以通过单个块地址进行访问。然而，因为我们需要 128 位宽的数据总线，所以这是以复杂的硬件设计为代价的。

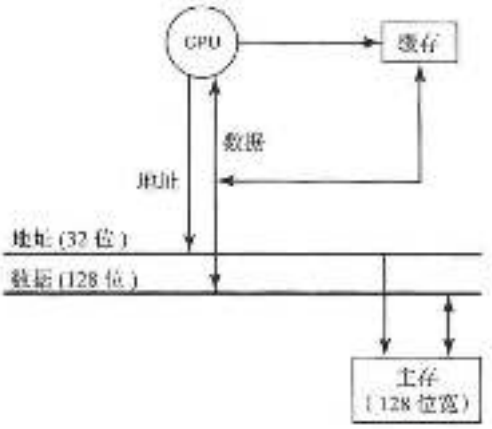


图 9-37 与缓存块大小相匹配的主存结构。内存是按块组织的，当发生缺失时，使用更宽的数据总线（128 位），传输包含缺失内存字的整块 406

例 9-12 假设 DRAM 的访问时间是 70 个周期，CPU 和内存之间的地址或数据的总线周期时间是 4 个周期。计算内存系统的块传输时间，这里总线宽度和内存结构与 4 个字的块大小相匹配。

答：

从 CPU 到内存的地址传输时间 = 4 个周期。

DRAM 的访问时间 = 70 个周期（全部 4 个字通过单次 DRAM 访问获取）。

从内存到 CPU 的数据传输时间 = 4 个周期

块的总传输时间 = 78 个周期。

9.20.3 交错式内存

从硬件角度考虑，在前一种设计中增加总线宽度的做法是不现实的。幸运的是，还有其他方法能够获得前一种设计中的性能优势，我们可以通过一种称为内存交错（memory interleaving）的工程技巧实现。图 9-38 显示了交错式内存系统的结构，主要思想是设计多个内存库（bank）。每个库负责提供缓存块的特定字。例如，缓存块由 4 个字构成，我们有 4 个内存库，M0、M1、M2 和 M3。M0 提供字 0，M1 提供字 1，M2 提供字 2，M3 提供字 3。

CPU 发送块地址，块地址被 4 个库同时接收。内存库是并行工作的，每个内存库在 DRAM 阵列中查找它负责的块中的字。当这些字被检索到时，利用与第一个简单的主存结构类似的标准总线，内存库轮流向 CPU 发送数据。

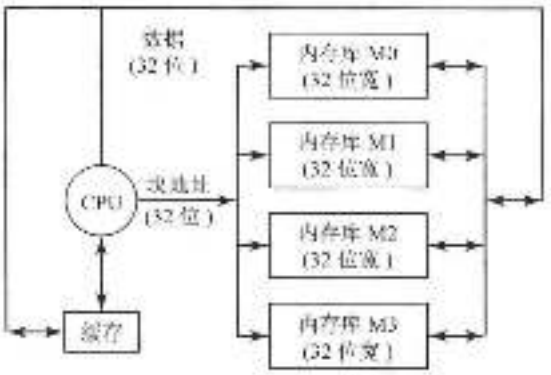


图 9-38 交错式主存。当收到块地址时，每个内存库依次发送块中的字（它保存的字）

交错式内存系统主要考虑 DRAM 访问是处理器内存交互中最耗时的部分。这样，交错式内存在避免硬件复杂性的情况下达到了接近宽内存时的性能情况。

例 9-13 假设 DRAM 的访问时间是 70 个周期，CPU 和内存之间地址或数据的总线传输时间是 4 个周期。计算图 9-38 中的交错式内存系统的读传输时间。

答：

从 CPU 到内存的地址传输时间 = 4 个周期（4 个内存库同时接收地址）。

DRAM 的访问时间 = 70 个周期（4 个字被 4 个库并行检索）。

从内存到 CPU 的数据传输时间 = 4×4 个周期（内存每轮流将各自的数据发送到 CPU）。

块的总传输时间 = 90 个周期。

目前，在内存系统设计中，我们考虑的主要因素是让处理器尽可能地保持繁忙状态。这也意味着，当发生缓存读缺失时，从内存到处理器的数据传输要尽可能快。向交错式内存系统写数据和向常规内存系统写数据没有什么差别。大多数情况下，处理器利用一些技术（例如 9.8.2 节讨论的写缓冲区技术）避免了向内存写数据带来的延迟。然而，许多处理器内存总线支持在交错式内存系统中运行得较流畅的块写操作，特别是对于整个缓存块的回写。

9.21 现代主存系统分析

现代内存系统与前面介绍的简单思想相距甚远。交错式内存已经过时。利用现代技术，交错思想现在主要体现在 DRAM 芯片本身中。让我们来解释这是如何工作的。在 2010 年前后，DRAM 芯片能够在一片上容纳 4G 位。

407
|
408

然而，为了更好地进行表述，我们假设 DRAM 芯片有 64 × 1 个位容量。也就是说，如果我们假设这个芯片有一个 6 位地址，每个地址对应 1 位数据。DRAM 存储的每 1 位称为单元（cell）。在实际中，DRAM 单元是按照矩阵排列的，如图 9-39 所示。正如图中所描述的，6 位地址被分为行地址 i（3 位）和列地址 j（3 位）。为了访问 DRAM 芯片中的一位，你必须首先提供 3 位行地址 i（称为行访问选通信号，或 RAS，请求）。DRAM 芯片会选择整个第 i 行，如图 9-39 所示。然后你必须提供 3 位列地址 j（称为列访问选通信号，或 CAS，请

求）。它选出了6位地址对应的特定位，并将它传送给内存控制器，内存控制器会将它传给CPU。

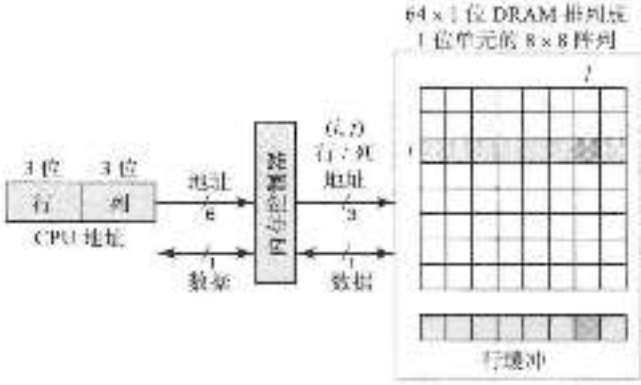


图9-39 访问64×1位DRAM。通过行地址将DRAM中的整个行选出；通过列地址选出该行中的特定位

对于一个1G位的芯片[9]，我们需要一个32K×32K的单元阵列。这种情况下行缓冲区的大小是32Kb。很有必要知道DRAM的周期时间是如何构成的。正如前面所讲，DRAM中的每个单元都是一个电容电荷。当选定某行时，会有电路（图中没有标明）感应选中行每个单元的电容电荷，并将行缓冲区中的相应位的值缓存为0或1。从这点考虑，读DRAM是破坏性的操作。通常在读取选中行后，DRAM电路需要对该行重新充电将它恢复为原来的样子。这种破坏性的读之后跟着进行充电的操作过程中伴随着行和列的地址解码，以及计算DRAM周期时间的时间累加。将阵列中的特定行读到缓存是整个操作中最费时间的部分。你可以很快发现，在完成这些操作后，这行中只有与列地址对应的1位被使用，其他位都被丢弃。我们在稍后将会看到（见9.21.1节）如何在不将它们全部丢弃的情况下使用行缓冲区中尽可能多的数据。

我们可以重新设计DRAM的结构，这样每个单元(i, j)不会对应1位，而是对应k位。例如，一个1M×8位的DRAM有一个1K×1K的单元阵列，阵列中每个单元包含8位。地址和数据通过芯片上的引脚（pin）传送给DRAM芯片（见图9-40）。芯片设计的一个主要考虑因素是减少这样的输入/输出引脚。我们发现驱动芯片中的逻辑单元所需的电流非常小，但将逻辑信号传入和传出芯片需要相对较大的电流。这意味着芯片边缘繁重的信号驱动电路消耗掉了本来可以用作其他应用的电能（如逻辑或内存操作）。由于这个原因，DRAM中的单元以方形阵列进行存储，而不是线性阵列，所以同样一组引脚以时间复用方式将行和列地址发送给DRAM芯片。这就是来自DRAM芯片用语的行地址选通（Row Address Strobe，RAS）和列地址选通（Column Address Strobe，CAS）。对行和列地址使用共享引脚的缺点是它们需要按顺序发送给DRAM芯片，这就增加了DRAM芯片的周期时间。

409

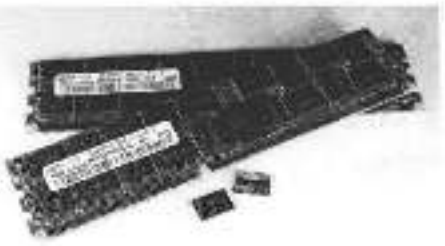


图9-40 三星的2Gb DDR3 DRAM芯片。单独的2Gb DDR3芯片如图所示（顶部和底部视图）；印刷电路板上设计相关的电路并将这些芯片装配在一起实现一个8GB的内存模块

[9] 1G位是2^{30}位。

下面的例子说明了如何用这些基本的 DRAM 芯片来设计一个主存系统。

例 9-14 利用如下信息，设计一个主存系统：

- 处理器到内存总线
  - 地址线 = 20
  - 数据线 = 32
- 每次处理器到内存的访问都跨过一个总线指定为 32 位字。

410

- DRAM 芯片的细节信息：1M × 8 位

答：

主存系统的总大小 = 2^{20} 字 × 32 位/字 = 1M 字 = 32Mb。

所以我们需要 4 个 DRAM 芯片，每个 DRAM 芯片有 1M × 8 位，采用如图 9-41 所示的结构。

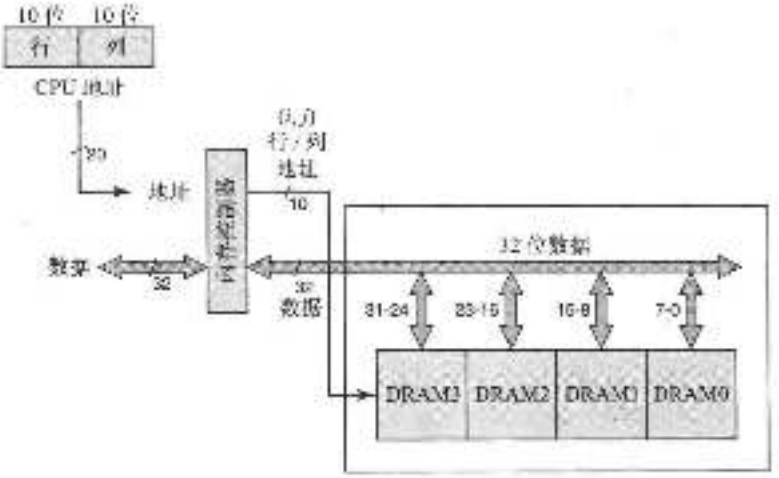


图 9-41 使用 1M × 8 位 DRAM 芯片的 32M 位内存系统。内存控制器依序向 4 个 DRAM 芯片提供 10 位行地址和列地址。每个 DRAM 芯片利用 10 位行地址从各自阵列中选出 1024 × 8 位放入它们特定的行缓冲区中（行缓冲区大小 = 8 192 位）。每个 DRAM 芯片利用 10 位列地址将对应行缓冲区中的唯一 8 位字节读出放在数据总线上

很容易将上述设计扩展成按字节寻址的内存系统。例 9-15 说明了如何利用 1Gb 的 DRAM 芯片建立一个 4GB 的内存系统。

例 9-15 利用如下信息，设计一个 4GB 的主存系统：

- 处理器到内存总线
  - 地址线 = 32

411

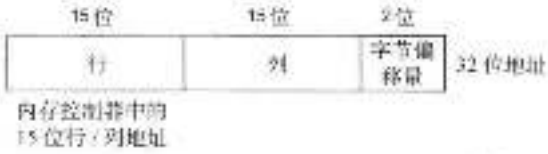
  - 数据线 = 32
- 每个 CPU 字都是由 4 字节构成的 32 位内字。
- CPU 支持按字节寻址。
- 地址线的最低 2 位指定 32 位字中的字节。
- 每个处理器对内存的访问都按字地址返回一个 32 位的字。
- DRAM 芯片的细节信息：1Gb（由 2^{28} × 4 位构成）。

答：

在32位地址中，最高30位用作字地址。

主存系统的最大小 = 2^{30} 字 ×4 字节 / 字 = 4GB = 32Gb。

所以，如图9-42所示，为了排列成二维阵列，我们需要32个DRAM芯片，每个DRAM芯片包含1Gb。为了对字中的字节进行写操作，内存控制器（图中没有显示）将选择二维阵列中合适的行，并将15位RAS和CAS请求以及其他控制信息一起传送给那个行。为了读取一个32位的字，它将向所有的DRAM传送15位RAS和CAS请求，这样控制器就得到了一个完整的32位字。



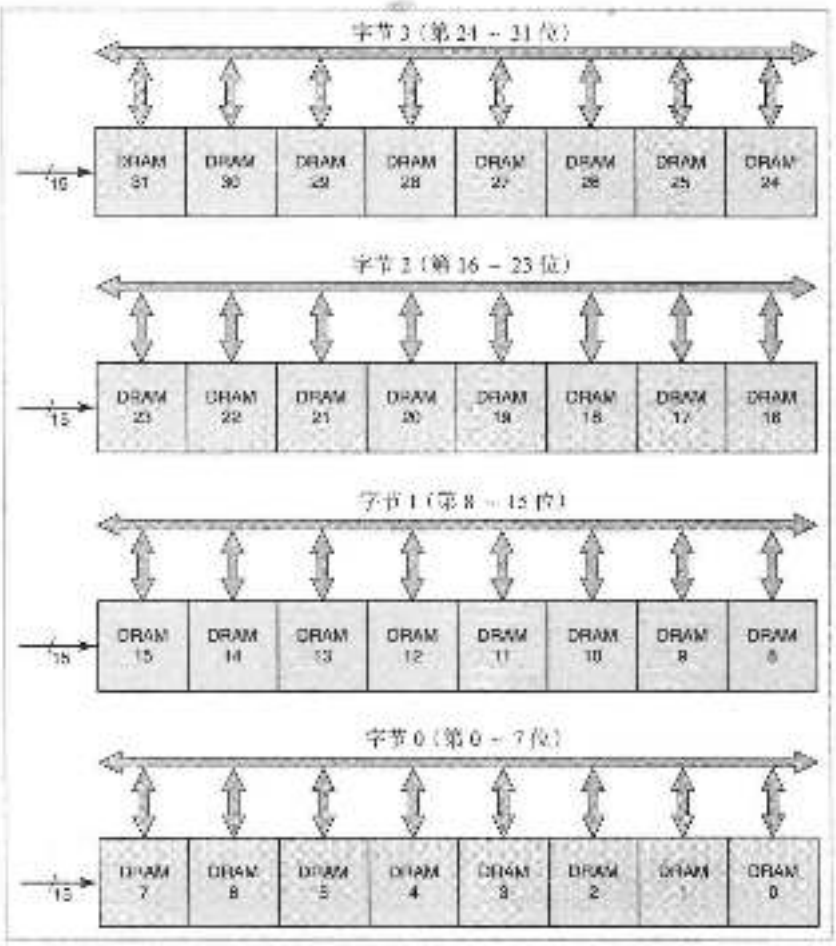


图9-42 使用1Gb DRAM芯片的4GB内存系统（使用类似于例9-15中的结构图）。为了读取32位的字，内存控制器同时向所有行（4行）顺序传输15位的RAS和CAS。同一行上的每个DRAM芯片都从各自的阵列中利用RAS和CAS地址选出特定的位。这样每行提供所需的32位字的8位以便响应内存控制器从CPU接收的地址

生产厂商将DRAM芯片封装在双列直插式存储模块（Dual In-line Memory Module，

DIMM）中。图 9-43 显示了 DIMM。通常，DIMM 是包含 4 ~ 16 个 DRAM 芯片、8 字节数据通路的小型印刷电路板。目前，DIMM 是内存系统的基本构件。



图 9-43 双列直插式存储模块

9.21.1 页式 DRAM

回想一下，当对 DRAM 单元进行读操作时会发生什么？内存控制器首先向 DRAM 提供行地址。DRAM 将整个选中行读入行缓冲区中。然后内存控制器提供列地址，DRAM 在行缓冲区中选择特定的列，并将数据传送给内存控制器。这两部分组成了 DRAM 的访问时间，并代表大部分 DRAM 周期时间。正如我们本节前面提及的那样，一旦选中的列数据传入控制器，行缓冲区中的其他都会被丢弃。在图 9-41 和图 9-42 中我们看到，同一行的连续列地址映射为 CPU 生成的连续内存地址。所以，从内存获取一个数据块也就是从 DRAM 获取相同行的连续列。回想一下设计交错式内存使用的技术（见 9.20.3 节）。每个内存库都保存同一块中的不同字，并以连续的总线周期在内存总线上将它传送给 CPU。DRAM 通过一种称为快速页模式（Fast Page Mode，FPM）的技术支持相同的功能。该技术能够做到在不增加额外 RAS 请求的情况下，允许在连续的 CAS 周期中访问行缓冲区的不同部分。例 9-16 很好地阐述了这一概念。

例 9-16 例 9-15 中的内存系统通过一个块大小为 16 字节的处理器缓存进行了增强。解释当发生缓存缺失时内存控制器如何将请求的块传送给 CPU。

答：

图 9-44 的上半部分显示了 CPU 生成的地址。CPU 芯片的内存控制器显示在图的下半部分。注意列地址的最低两位是 CPU 地址中块偏移量的最高两位。块大小为 16 字节，或者说是 4 字。假设缓存块为连续的字当第 i 行 4 个连续列给出，列地址仅最低两位发生变化。

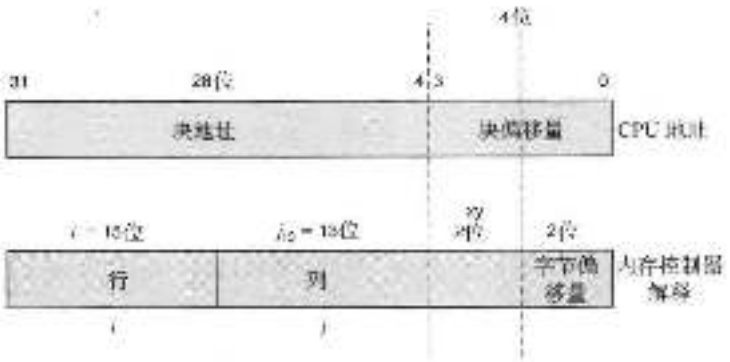


图 9-44 内存控制器对 32 位 CPU 地址进行解释。xy 地址位代表块中的字。在图 9-42 中，DRAM 芯片的每行提供一个字的一个字节。来自内存控制器的 RAS 请求选出内存块中空间相邻的字，并将它们放入 DRAM 的行缓冲区中。为了获得给定内存块的连续字，内存控制器需要向 DRAM 库发出与不同二进制组合 xy 相对应的 CAS 请求，这样才能从 DRAM 的行缓冲区中读取块中的连续字

例如，假设我的 CPU 地址是 (i,j)，$j=0100000110001101$。

我们进行下述说明：

j 由 $j_{hi}xy$ 构成，其中 j_{hi} 代表列地址 j 的最高 13 位。

为了从 DRAM 芯片中读取整个块，内存控制器进行下列操作：

1）为了地址 (i,j)，向 DRAM 阵列发送 RAS/CAS 请求。

2）利用地址 $j_{hi}xy$ 多发送 3 个额外的 CAS 请求，这里 $xy=00$，10，11。

每个 CAS 请求都将使 DRAM 芯片发送连续的 4 个字（返回的第一个字是地址被失的真实地址）。内存控制器在 4 个连续内存总线周期中传输这 4 个字，将它们返回给 CPU。

经过这些年的不断发展，DRAM 技术有了很大提高。本节介绍了该技术的部分内容。希望读者通过这部分内容的学习能够激发更多的兴趣，关注这一领域在本书范围之外的更高级的主题。

9.22 分级存储体系的性能影响

CPU 会与分级存储体系有显式或隐式的交互：处理器寄存器，缓存（多个等级）、主存（在 DRAM 中）和虚拟内存（在磁盘上）。离处理器越远的存储容量越大，速度越慢。在分级存储体系中离处理器越远每字节的价格也越便宜。

正如我们所讨论的，尽管相对速度和容量大致保持相同，但真实容量和速度每年都保持着持续增长。表 9-1 就是一个具体的例子，它对 2006 年前后不同等级的分级存储体系的相对延迟和容量进行了总结。2006 年前后 2GHz Pentium 处理器的时钟周期时间是 0.5ns。

412 ~ 415

表 9-1 2006 年前后分级存储体系的相对大小和延迟

存储类型	典型大小	读取一个 4 字节字的 CPU 时钟周期的近似延迟
CPU 寄存器	8 – 32	通常，直接访问（0 – 1 个时钟周期）
L1 缓存	32KB – 128KB	3 个时钟周期
L2 缓存	128KB – 4MB	10 个时钟周期
主存（物理内存）	256MB – 4GB	100 个时钟周期
虚拟内存（在硬盘上）	1GB – 1TB（千兆字节）	1 000 – 10 000 个时钟周期（不考虑处理页错误的软件开销）

分级存储体系在系统性能中扮演着重要的角色。我们可以看到，对于当前正在执行的程序，缺失损失会影响流水线处理器的性能。更重要的是，内存系统和 CPU 调度器的设计需要在设计决策时了解分级存储体系的概念。例如，内存管理器的页替换策略从分级存储体系的各级中删除相关物理页的内容。所以，经历页面错误的进程在错误从磁盘引入物理内存之后可能会有明显的性能损失，直到页面的内容填满了邻近级的分级存储体系，性能损失才会得到缓解。

CPU 调度对系统性能也有类似影响。上下文切换的直接开销包括保存和加载被取消调度的进程和新调度的进程的进程控制块（PCB）。刷新被取消调度的进程的 TLB 是直接开销的一部分。因为是分级存储体系，所以上下文切换是间接开销。开销表现为从缓存到物理内存分级存储体系的不同级的缺失。一旦新调度的进程的工作集达到了分级存储体系中离处理器较近的级，进程将使处理器的真正性能发挥出来。这样，在计算 CPU 调度器使用的时间量子（time quantum）时有必要考虑由于分级存储体系所带来的对上下文切换性能的真正影响。

小结

416 表 9-2 提供了本章中的重要术语和概念。

表 9-2 与分级存储体系相关的概念总结

种类	词汇	备注
局部性原理（9.2 节）	空间	访问连续的内存单元
	时间	重用已经访问的内存单元
缓存结构	直接映射	一对一映射（9.6 节）
	全相关	一对多映射或一对一映射（9.11.1 节）
	组相关	一对多映射（9.11.2 节）
缓存读/写（9.8 节）	读命中/写命中	CPU 访问的内存单元在缓存中存在
	读缺失/写缺失	CPU 访问的内存单元不在缓存中
缓存写策略（9.8 节）	直写	CPU 对缓存和内存进行写操作
	回写	CPU 只对缓存进行写操作，当发生替换时更新内存
缓存参数	缓存总大小（S）	按字节计缓存的总数据大小
	块大小（B）	一个数据块中连续数据字大小
	相关程度（p）	给定的内存块在缓存中能够保存的单元数
	缓存行的数目（L）	S/pB
	缓存访问时间	缓存中完成查找命中/缺失的 CPU 时钟周期时间
	CPU 访问单元	CPU 和缓存之间交换数据的大小
	内存传输单元	缓存和内存间交换数据的大小
	缺失损失	处理缓存缺失所花的 CPU 时钟周期时间
内存地址解释	索引（n）	$\log_2 L$ 位，用来查找确定的缓存行
	块偏移量（b）	$\log_2 B$ 位，用来选择块中的特定字节
	标记（t）	$m-(n+b)$ 位（m 是内存地址的位数），用来与缓存中的
417		标记字段进行匹配
缓存项/缓存块/缓存行/组	有效位	表示数据块是有效的
	脏位	对于回写，表示数据块是否比内存中的新
	标记	用于将标记与内存地址进行比较判断是否命中
	数据	实际的数据块
性能指标	命中率（h）	CPU 访问缓存命中的百分比
	缺失率（m）	$1-h$
	平均内存延迟	每条指令访存次数 $_{avg}$ × 缺失损失 $_{avg}$
	第 i 级内存访问时间（EMAT$_i$）	$\text{EMAT}_i = T_i + m_i \times \text{EMAT}_{i+1}$
	有效 CPI	CPI_{avg} + 内存延迟 $_{avg}$
缺失种类	强制缺失	CPU 第一次访问该内存单元
	冲突缺失	由于相关程度有限造成的缺失，尽管缓存没满
	容量缺失	缓存满时造成的缺失
替换策略	FIFO	先进先出
	LRU	最不常访问的
内存技术	SRAM	每位有 6 个晶体管的静态 RAM
	DRAM	每位有一个单晶体管的动态 RAM
主存	DRAM 访问时间	DRAM 读访问时间
	DRAM 周期时间	DRAM 读和刷新时间
	总线周期时间	CPU 和内存之间的数据传输时间
	使用 DRAM 的数据交错技术	使用 DRAM 的页模式位

现代处理器的分级存储体系（一个例子）

现代处理器采用多级缓存。处理器有片上的 L1 缓存（分开的指令和数据部分）和指令与数据部分为一体的 L2 缓存，这种设计很常见。在处理器之外，主存之前可能会有 L3 缓存。 418
采用多核技术，内存系统正变得越来越复杂。例如，AMD 在 2006 年[⊖]引入了 Barcelona[⊜]芯片。这个芯片有 4 个核，每个核有自己的 L1 缓存（分开的 I 缓存和 D 缓存）和 L2 缓存。而之后的 L3 缓存被所有核共享。图 9-45 显示了 Barcelona 芯片的存储结构。L1 缓存是 2 路组相关缓存（指令部分 64KB，数据部分 64KB）。L2 缓存是 16 路组相关缓存（总大小为 512KB，指令部分和数据部分是一体的）。L3 缓存是 32 路组相关缓存（2MB 所有核共享）。

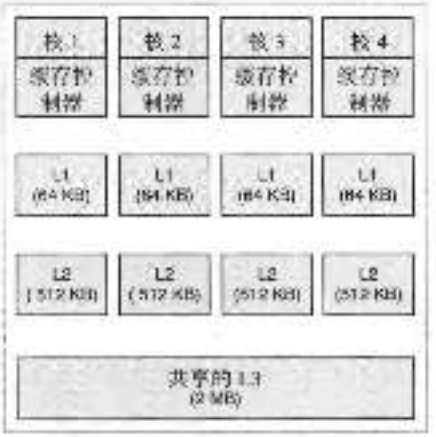


图 9-45 AMD 的 Barcelona 芯片，这个 4 核芯片有 3 个片上缓存

练习题

1. 将时间局部性和空间局部性进行对比。
2. 将直接映射、组相关和全相关缓存设计进行对比。
3. 有一个同学设计了这样一个缓存，最高有效位用作索引位，最低有效位用作标记位。你觉得这样的缓存性能如何？为什么？
4. 在标记位为 t 位的直接映射缓存中，你觉得会有多少个标记比较器？它们一次比较操作会比较多少位？ 419
5. 解释在缓存中为什么要设计脏位。
6. 给出充足的理由说明为什么要使用多级缓存分层结构。
7. 缓存设计考虑的主要因素是什么？讨论这些考虑因素如何影响分级存储体系的不同级。
8. 增加缓存块大小的动因是什么？
9. 采用组相关缓存设计的动因是什么？
10. 判断正误，并给出理由：与 L2 缓存相比 L1 缓存通常有更高的相关程度。
11. 给出 3 个理由说明 L1 缓存需要分为 I 缓存和 D 缓存。
12. 给出充分的理由说明在更深级的缓存分层结构中需要使用统一的 I 缓存和 D 缓存。
13. 判断正误，并给出理由：只要有 L2 缓存，L1 缓存设计就可以只关注将它的访问速度提升到与处理

⊖ Phenom 是 AMD 为台式机生产的芯片品牌。

⊜ AMD 的 Phenom 处理器数据表在：http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf。

器的时钟周期时间相匹配。

14. 你的工程团队告诉你，你有一个总大小为 2MB 的片上缓存。你可以选择将它设计成一个大的 L1 缓存，或两级 L1 和 L2 缓存，在不同级缓存中划分 I 缓存和 D 缓存，等等。你会做出什么样的选择？为什么？在做决定时请考虑命中时间、缺失率、相关程度和流水线结构。对于这个问题，你有足够的理由给出设计决定的定性解释。

15. 对于 4 路组相关缓存，使用真实的 LRU 替换策略，你需要用多大的计数器？

16. 考虑下述分级存储体系：

- L1 缓存：访问时间 =2ns；命中率 =99%
- L2 缓存：访问时间 =5ns；命中率 =95%
- L3 缓存：访问时间 =10ns；命中率 =80%
- 主存：访问时间 =100ns

计算有效内存访问时间。

17. 分级存储体系的资源如下：

L1 缓存	2ns 的访问时间	98% 的命中率
L2 缓存	10ns 的访问时间	？？
内存	60ns 的访问时间	

假设对于 L1 和 L2 缓存，需要使用查找操作来看看访问是缺失还是命中。为了确保有效内存访问时间不超过 3ns，L2 缓存的命中率需要保持为多少？

420 18. 你需要为 32 位处理器设计缓存。内存是按字进行组织的，但按照字节进行寻址。你被告知使用的缓存为 2 路组相关缓存，每块有 64 字（256 字节）。你允许使用总大小为 64K 字（256KB）的数据（除去标记位、状态位等）。

描绘缓存的布局。

说明为了进行缓存查找，CPU 生成的地址如何解释为块偏移量、索引和标记位。

19. 下列关于缓存的说法，哪些是正确的？

- 可以用处理器的指令集架构像操作寄存器那样去使用缓存。
- 通常不能直接使用处理器的指令集架构对缓存进行操作。
- 通常缓存使用的技术和主存一样。
- 缓存通常比寄存器堆大，但比主存小。

20. 区分冷缺失、容量缺失和冲突缺失。

21. 进行如下的改变，重做例 9-7。

- 总共 16 个块的 4 路组相关缓存
- LRU 替换策略
- 访问序列如下：

0,1,8,0,1,16,24,32,8,8,0,5,6,2,1,7,10,0,1,3,11,10,0,1,16,8

22. 进行如下改变，重做例 9-8。

- 32 位虚拟地址，24 位物理地址，8KB 页大小，64 项的直接映射 TLB

23. 解释术语虚拟索引的物理标记的缓存（virtually indexed, physically tagged cache）。这种设计的优点是什么？这种设计的局限性是什么？

24. 解释术语页着色（page coloring）。这项技术解决了什么问题？它是如何工作的？

25. 进行如下改变，重做例 9-10。

- 虚拟地址 64 位
- 页大小 8KB

缓存参数：2 路组相关缓存，总大小 512KB，块大小 32 字节 421

26. 进行如下改变，重做问 9-15。

- 地址线和数据线均为 64 位
- 64 位 CPU 字
- 使用 1Gb DRAM 芯片

27. 解释术语 3 维或 DRAM。

参考文献注释和扩展阅读

从 20 世纪 60 年代早期缓存发明以来，分级存储体系统一直是广大作者探讨的热点。计算机先驱 Maurice Wilkes 写了第一篇讨论缓存思想的技术论文 [Wilkes，1971]。为了深入理解缓存，我们向读者推荐 Alan Jay Smith 的论文 [Smith，1982]。Hennessy and Patterson[Hennessy，2006] 对许多先进的能增强缓存性能的优化技术进行了总结。Bryant and O'Hallaron[Bryant，2003] 对缓存设计基础和缓存对程序性能的影响进行了总结。内存技术在不断发展变化，有这样一个 DRAM 发展规律，"DRAM 容量每 3 年增长 4 倍"。基于这样的现实情况，获取最新内存技术的渠道是生产商的网页介绍。例如，Samsung、Kingston、Hynix 和 Micron 等公司引领着 DRAM 市场，所以这些供应商的网页是查找内存技术最新发展状况的好地方。 422

输入 / 输出和稳定性存储

为了更好地理解计算机，接下来我们探讨 I/O 子系统。在第 4 章中，我们讨论了中断，I/O 设备通过中断进行处理器的任务切换。在本章中，我们将说明处理器和 I/O 设备间更多的交互细节，以及处理器、内存和 I/O 设备之间不同数据的传送方式。

我们首先讨论 CPU 和 I/O 设备间通信的基本范式。之后我们讲解完成通信所需的硬件机制，以及 CPU 和 I/O 间用作数据传输通道的总线细节。对硬件机制的补充是称为设备驱动的操作系统实体，设备驱动完成 CPU 和每个特定 I/O 设备间的真正通信。稳定性存储在大多数计算机系统中是通过硬盘（hard disk）提供的，毫无疑问这是计算机中最重要、最复杂的 I/O 设备之一。我们将讨论硬盘的细节设计，包括精心设计的 I/O 调度算法。

10.1 CPU 和 I/O 设备间的通信

尽管我们已经讨论了计算机系统中的处理器部分，但大部分计算机用户可能还没有意识到他们正在使用的很多小工具中就存在着处理器。例如，手机或 iPod 中就有处理器。我们使
423 用 iPod 是因为它有吸引我们的功能。iPod 或手机都是通过输入 / 输出设备来完成交互功能的。因此，了解 I/O 设备如何与计算机系统的其他组件进行交互是解开计算机神秘面纱的关键（见图 10-1）。尽管有多种 I/O 设备，但它们与系统其他部分的连接方式非常类似。正如我们所见，处理器执行指令集中的指令。LC-2200 并没有直接对 CD 播放器或话筒进行操作的特殊指令。接下来我们将学习设备如何工作，比如 iPod 如何播放音乐。

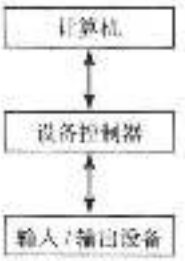


图 10-1 设备控制器和其他组件间的关系

I/O 设备和计算机之间有一个特殊的硬件，称为设备控制器（device controller），它扮演着两者之间桥梁的作用。设备控制器知道如何在 I/O 设备和计算机间进行交互。

10.1.1 设备控制器

为了讨论得更具体，我们考虑一个非常简单的设备，键盘（keyboard）。当敲击键盘时设备里有电路完成将按下的键映射成它所代表的二进制字符编码。这种二进制编码，通常称为 ASCII（American Standard Code for Information Interchange，美国标准信息交换码格式），需要传送给计算机。为了实现这种信息交换，有两件事是必需的。首先，我们需要临时空间来保存输入的字符。其次，我们需要引起中断来将字符传给处理器。这就是设备控制器产生的原因。我们来思考一下键盘设备控制器至少需要什么。它有两个寄存器：数据寄存器和状态寄存器。数据寄存器是保存键盘输入字符的存储空间。正如其名，状态寄存器用于将设备与计算机信息交换的当前状态进行汇合。

对于键盘，我们有如下陈述。

- 有一位就绪位（ready bit），用于回答问题“数据寄存器中的字符是不是最新的（即处理

器还没有发现它”。

- 有一位中断允许位（Interrupt Enable，IE），用于回答问题“处理器允许设备中断它吗”。
- 有一位中断标志位（Interrupt Flag，IF），用于回答问题“控制器准备好中断处理器了吗”。 424

图 10-2 显示了简易的键盘控制器。取决于设备的复杂程度，我们还需要包含其他的额外状态信息。例如，如果设备的输入速率超过了信息传递给处理器的速率，那么控制器中需要有数据溢出标志位（data overrun flag）来记录状态。

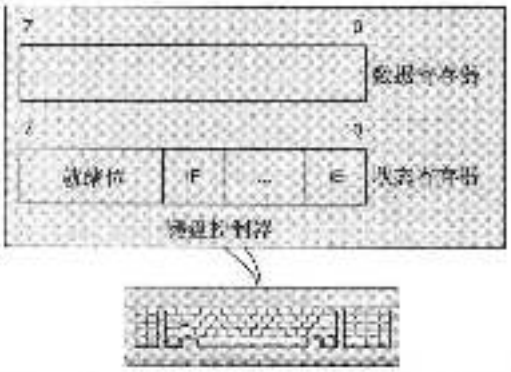


图 10-2 键盘控制器和把一个数据寄存器和一个状态寄存器，数据寄存器保存键盘最后发送的字符，状态寄存器保存 CPU 和设备间交互的当前状态

有时，很有必要将来自设备的状态和来自处理器的命令进行区分。键盘设备非常简单，来自处理器的命令只有打开和关闭中断允许位。一个更复杂的设备（如照相机）可能需要额外的命令（如调焦、摇摄和移轴等）。总体上说，设备控制器通过一组寄存器和处理器进行交互。

10.1.2 内存映射 I/O

接下来，我们来讨论计算机如何与设备控制器进行连接。我们必须以某种方式让控制器中的寄存器对处理器可见。一种简单实现这一目标的方法是使用处理器内存总线，如图 10-3 所示。处理器使用存取（load/store）指令对内存进行读和写。如果控制器中的寄存器（数据寄存器和状态寄存器）以内存单元的形式呈现给 CPU，那么处理器可以简单地使用同样的存取（load/store）指令对这些寄存器进行操作。这项技术称为内存映射 I/O（memory mapped I/O），允许处理器在不发生任何变化的情况下与设备控制器进行交互。

将设备寄存器作为内存单元使用的技术很简单。我们给设备寄存器唯一的内存地址。例如，我们随意选取内存地址 5000 赋值给数据寄存器，将内存地址 5002 赋值给状态寄存器。键盘控制器里有智能组件（即电路）对这两个地址总线上的地址进行操作。例如，假设处理器执行如下指令：

```
LW r1, Mem[5000]
```

控制器识别出地址 5000 号数据寄存器相对应，然后将数据寄存器的数据放在数据总线上。处理器并不明白这些内容来自控制器的一个特定寄存器。它仅将数据总线上的值存入寄存器 r1 中。

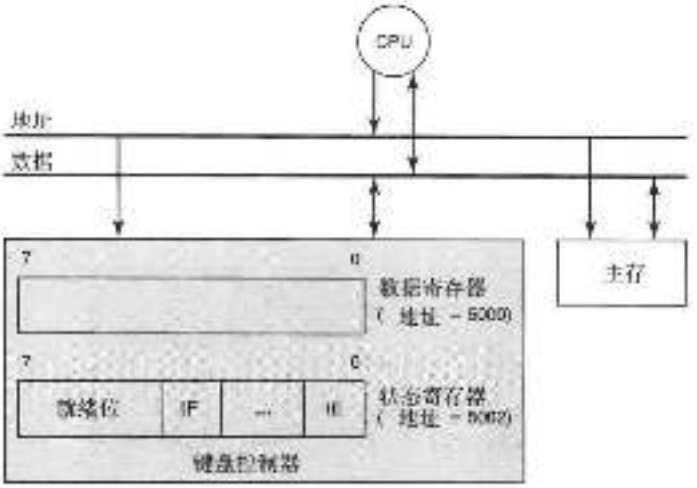


图 10-3 将设备控制器与处理器内存总线相连。通过给控制器中的寄存器提供唯一的内存地址，使控制器中的寄存器地址在 CPU 看来就像内存地址

这就是内存映射 I/O 的魅力所在。在不增加指令集中新指令的情况下，我们已经将设备控制器集成到了计算机系统中。内存也与总线上的地址进行交互，你也许会怀疑这项技术是否会使内存和设备控制器发生混乱。基本思想是在地址空间中为设备控制器保留一部分空间。假设有 32 位处理器，并且我们希望为 I/O 设备留出 64KB 的地址空间。我们可以在 0XFFFF0000 ~ 0xFFFFFFFF 中任意为 I/O 设备划分地址，并在这个范围内为我们希望为连接到总线上的设备分配地址。例如，对于键盘控制器，我们希望将地址 0xFFFF0000 作为数据寄存器，地址 0XFFFF0002 作为状态寄存器。键盘设备控制器的设计会考虑这种分配，并对这些地址做出反应。这意味着每个设备控制器都有电路对总线上的地址进行解码。如果总线上的地址和这个控制器的地址相匹配，在总线上它会表现得像内存一样执行相关命令（读和写）。相应地，内存设计会忽略划分给 I/O 的地址范围。当然，至于划定多少地址范围给 I/O 这只是一个习惯问题。通常习惯将高地址空间留给 I/O 设备寄存器。

425
|
426

你也许会惊讶如何将这一信息与我们在第 9 章中刚刚学过的分级存储体系的细节相协调。如果分配给设备寄存器的内存地址出现在缓存中，处理器会不会从缓存中获取一个旧数据，而不是设备寄存器中的内容呢？这是一个关键问题，正是由于这一原因缓存控制器才将内存的特定区域设定为“不可缓存的”。即使处理器像读内存地址那样去读设备寄存器，提前设置为先验的缓存控制器也不会对这些地址进行缓存，但每次读它们处理器都会重新对它们进行访问。

内存映射 I/O 的优点是不需要额外的 I/O 指令；缺点是部分内存地址空间分配给设备寄存器，所以这部分地址空间对用户和操作系统的代码和数据来说是不可用的。有些处理器提供特殊的 I/O 指令，并将 I/O 设备连接到独立的 I/O 总线上。类似的设计（称为 I/O 映射 I/O）在嵌入式系统中非常常见，因为内存空间有限。然而，因为现代通用处理器使用 64 位地址空间，在这么大的地址空间中为 I/O 保留一小部分是很合理的做法。所以内存映射 I/O 是现代处理器设计的选择，尤其是因为它很容易融入处理器体系结构中，并且不需要新的指令。

10.2 程控 I/O

既然我们已经知道了设备控制器与处理器是如何连接的，让我们将注意力转向在设备与处理器之间是如何相互传递数据的。程控 I/O（PIO）指的是编写一个计算机程序来完成数据传输。为了让讨论更具体，我们举前面介绍的键盘控制器的例子。

让我们来对键盘控制器的步骤进行总结：

1）当新的字符进入数据寄存器时它设置状态寄存器的就绪位。

2）当 CPU 读取数据寄存器时，控制器清除就绪位。

使用键盘控制器的给定讲义，我们编写一个简单的程序，将来自键盘的数据转移到处理器中（如图 10-4 所示）。

- 步骤 1：检查就绪位（如图 10-4 所示）。
- 步骤 2：如果就绪位没有设置，返回步骤 1。
- 步骤 3：读数据寄存器中的内容（见图 10-4）。注意读这个寄存器会自动清除就绪位。
- 步骤 4：将读入的字符存入内存（见图 10-4）。
- 步骤 5：返回步骤 1。

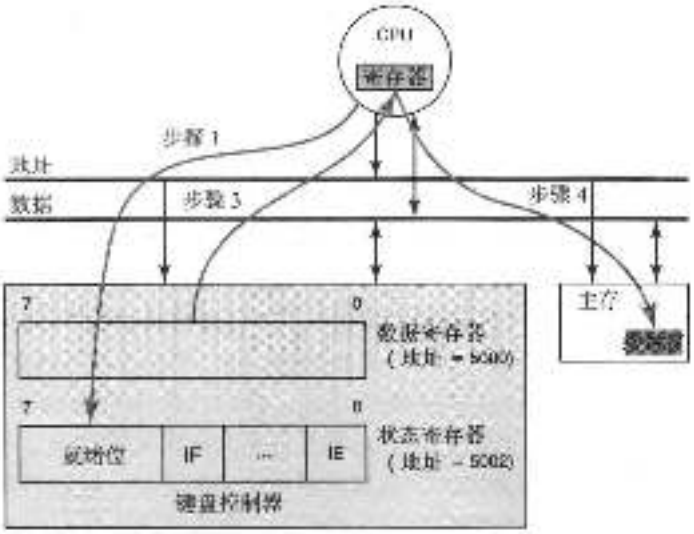


图 10-4 PIO 数据传输示例。控制器设置就绪位向 CPU 表明数据寄存器中有新数据。CPU 使用简单的存取（load / store）指令将数据从数据寄存器传输到内存缓冲区中

步骤 1 和 2 构成了处理器和设备控制器之间的握手。通过这两步，处理器不断地检查设备是否有新数据。换句话说，处理器轮询设备看看是否有新数据。考虑处理器与类似键盘的设备之间的速度差异。1GHz 处理器执行一条指令大约为 1ns。即使打字速度非常快，假设每分钟输入 300 个字符，控制器每 200ms 输入一个字符。处理器在获得输入字符前要对状态寄存器轮询数百万次。这对处理器资源的利用很没有效率。可以给设备设置一个中断位来代替轮询操作，当发生中断时，执行前述的指令来完成数据传输。操作系统调度其他程序在 CPU 上运行，通过上下文切换来处理中断，如第 4 章所述。

通过轮询或中断方式的程序数据传输被低速设备使用（例如，键盘和鼠标），这些低速设备通常是异步生成数据的。即数据的生成和任何时钟频率不合拍。然而，类似硬盘的高

427
↓
428

速设备是同步产生数据的。也就是说数据是按照一定时钟频率产生的。当设备准备就绪时，如果处理器没有及时获取数据，那么很有可能设备会用新的数据覆盖掉旧的数据，造成数据丢失。目前最先进的处理器的内存总线带宽为200MB/s。总线上的所有实体（处理器和设备控制器）共享带宽。最先进的硬盘驱动以150MB/s的速率产生数据。考虑到数据生成速率和可用内存总线带宽之间的差距，并不适合用程控I/O在内存和像硬盘这样的高速同步设备之间传输数据。

而且，即使对慢速设备，通过程控I/O使用处理器对数据传输进行编排也是对处理器资源的浪费。在下一节，我们引入一个新的技术来完成数据传输。

10.3 DMA

直接内存访问（Direct Memory Access，DMA），正如其名，在处理器没有介入的情况下，设备控制器能够与内存进行数据交换。

传输本身是由处理器进行初始化的，但当初始化完成，就由设备来完成传输操作。让我们来尝试理解DMA控制器中需要哪些技术。我们认为控制器连接的是一个异步高速设备，如硬盘。我们认为这种设备是流设备（streaming devices）：无论在哪个方向（传入或传出设备），一旦数据传输开始，数据就会持续不断地传入或传出设备，直到传输完成为止。

做一个类比，假设你在做饭。你需要往炉灶上的锅里倒水。你使用一个小杯子去厨房的水龙头接水，将水倒入锅中，然后再回去用杯子接水，重复这个过程直到锅中的水满为止。你知道如果水龙头不拧上，水就会一直流。所以你不断地打开或关掉水龙头，并用杯子作为水龙头和锅之间的缓冲区。

流设备就像水龙头。从设备中读数据，设备控制器打开设备，取出位流，将控制器关闭，把位传入内存，之后重复这些操作直到完成了整个数据传输为止。对于向设备写数据则过程相反。让我们来关注从设备读数据到内存的操作。向内存传输数据的过程中都包含了什么？控制器获取总线并每次向内存传输一字节（或者其他大小，不论总线传输的粒度）。总线是共享资源，总线上还有包括处理器在内的其他竞争者。所以，设备控制器和内存之间的数据传输是异步进行的。更糟糕的是，如果总线上还有其他优先级更高的请求，控制器可能会在很长时间内无法获得总线。为了缓解同步设备和异步总线间的差异，控制器需要一个硬件缓冲区（hardware buffer），类似于做饭例子中杯子的作用。我们直觉上推断缓冲区应该和设备与设备控制器之间异步传输单元的大小相同。例如，如果设备是摄像机，那么单帧图像（比如，100K像素）可能是设备和控制器之间最小的异步数据传输单元。因此，对于是摄像机的设备控制器情况，硬件缓冲区的大小至少应该是一个图像帧的大小。

429

为了初始化一个传输，处理器需要向设备控制器传输4种信息：命令、设备地址、内存缓冲区地址和数据传输量。注意数据传输是在设备空间和内存空间的连续区域之间。另外，控制器有一个状态寄存器，用来记录设备状态。如键盘的例子所示，我们可以在控制器中分配5个内存映射寄存器：命令、状态、设备地址、内存缓冲区地址和计数。所有这些寄存器有分配给它们的唯一内存地址。图10-5显示了流设备的DMA控制器的简化框图。

例如，从内存缓冲区中以地址M开始向地址为D的设备中传输N字节的数据，CPU执行如下指令：

步骤1：在计数寄存器中存储N。

步骤 2：在内存缓冲区地址寄存器中存储 M。
步骤 3：将 D 保存在设备地址寄存器中。
步骤 4：将写设备命令保存在命令寄存器中。
步骤 5：设置状态寄存器中的开始（Go）位。

430

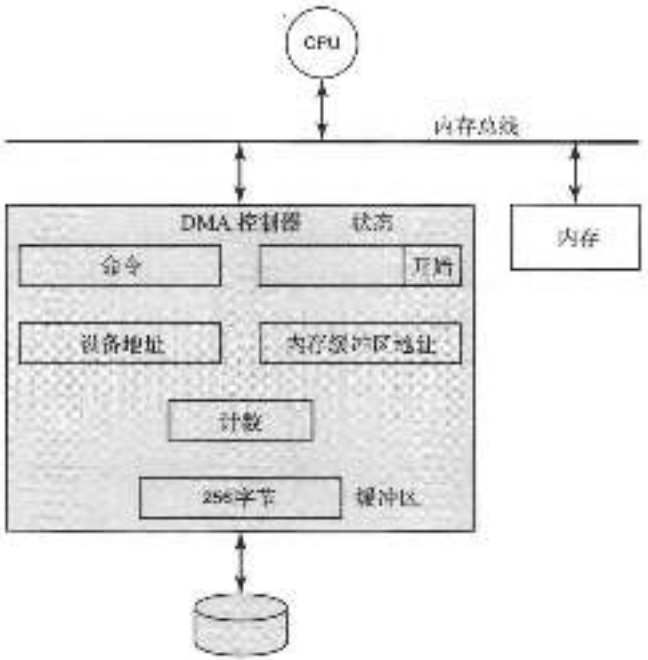


图 10-5 DMA 控制器。控制器中的所有寄存器都分配了唯一的内存地址，所以 CPU 可以通过存取（load / store）指令对它们进行操作

注意所有这些步骤都是简单的内存存储指令（到目前为止涉及 CPU 的情况），因为这些寄存器映射到了内存。

设备控制器和 CPU 类似。在设备控制器内部，有用于执行从处理器获取指令的数据通路及控制电路。例如，为了完成前面所说的传输（见图 10-6），控制器将反复访问内存总线，将 N 个连续的字节从内存地址 M 开始装入缓冲区中。一旦缓冲区就绪，控制器将以指定的设备地址将缓冲区中的内容初始化到设备中。如果处理器允许控制器执行中断，那么当传输完成时控制器将中断处理器。

设备控制器与处理器竞争内存总线周期，这种现象通常称为周期窃无（cycle stealing）。这是一个陈旧的术语，它的起因是因为处理器通常是总线的控制器。当处理器不需要它们时设备会窃用（steal）总线周期。从第 9 章关于分级存储体系的讨论中我们可以知道处理器在大多数情况下会使用缓存中的指令部分和数据部分。因为对于设计良好的结构会考虑内存总线的总带宽比连接到总线的所有设备（包括处理器）的需求总和大，所以设备窃用处理器的总线周期并不会产生问题。

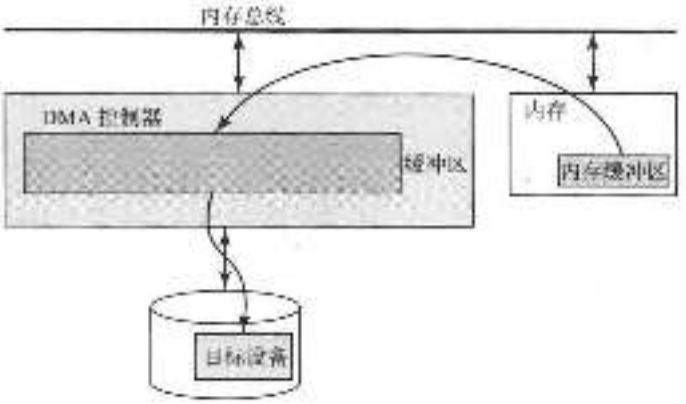


图 10-6 DMA 数据传输示例。一旦 CPU 利用所有需要的信息通过设置控制器中的寄存器对 DMA 进行初始化，控制器就会在不打扰 CPU 的情况下自动完成所要求的数据传输

431

10.4 总线

系统总线（或内存总线）是整个设计中的关键资源。从功能的角度讲，总线有如下部分：

- 地址线
- 数据线
- 命令线
- 中断线
- 中断响应线
- 总线仲裁线

电路上，高性能系统并行运行这些线构成了系统总线。例如，32 位寻址的处理器有 32 位地址线。总线中数据线的数目由总线支持的命令集决定。正如我们在第 9 章中讨论的那样，为了支持大的缓存块大小，可以想象数据总线的宽度比处理器中字宽度要宽。命令线会对内存系统的特定命令（读、写、块读取、快写入等）进行编码，所以命令线需要有足够的位数来对命令集进行二进制编码。我们在第 4 章讨论过处理器和 I/O 设备之间中断握手的细节。中断线（和中断响应线）的数目与所支持中断级别的数目相对应。在第 9 章中，我们强调了内存系统的重要性。在一定意义上，系统的性能主要由内存系统的性能决定，所以系统总线作为访问内存的窗口需要尽可能进行充分利用。通常，在当前总线周期中，设备竞争使用下一个总线周期。在当前周期结束前，需要选择在下一个总线周期中运行的设备。从集中式方案（处理器中）到更多分布式方案，有各种总线仲裁（bus arbitration）技术。关于总线仲裁方案更为详细讨论超出了本书的范围。有些较旧的书会对这个话题进行更多的细节讨论（例如，[Patterson，1998]）。但我们需要阅读一些更高级体系结构的书（例如，[Hennessy，2006]）来理解现代高速处理器如 Intel Pentium 处理器的系统总线是如何工作的。

在过去的几年，花费了很多努力对总线进行标准化。标准化的主要目的是允许第三方供应商与总线进行交互开发 I/O 设备。标准化对计算机生产商提出了一个很有意思的挑战。一方面，标准化帮助了计算机生产商，因为对于给定平台这增加了第三方供应商提供的可用外围设备的范围。但另一方面，为了保证所生产产品的竞争性，大多数计算机生产商都反对这

样的标准化。实际中我们看到了两者的折中选择。系统总线倾向于由制造商专有设计，并不依赖于任何发布的公开标准，但连接外围设备的 I/O 总线倾向于采用制定的标准。例如，外围组件互连（Peripheral Component Interchange，PCI）是 I/O 总线的公开标准。大多数计算机生产商会支持 PCI 总线，同时提供内部桥接器（bridge）将 PCI 总线与系统总线进行连接（见图 10-7）。

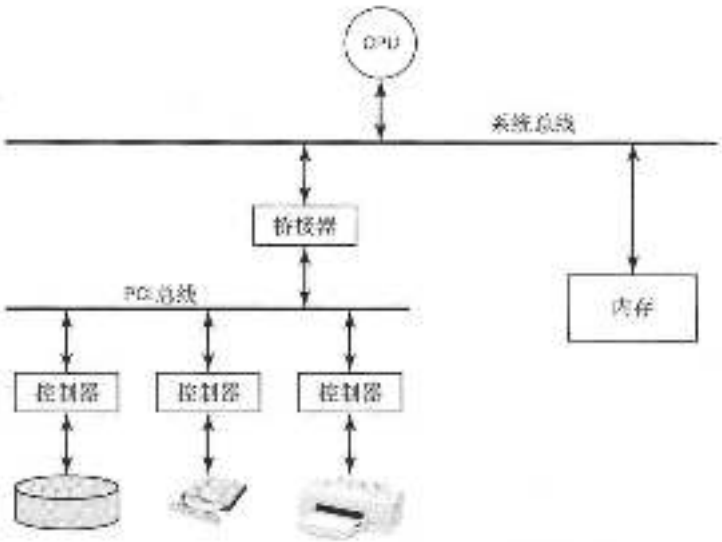


图 10-7 标准总线与系统总线共存。桥接器将 CPU 的内部系统总线与类似 PCI 的标准总线进行连接

有些总线设计将线路设计成多种功能复用的形式，例如 PCI 总线对于地址和数据使用同样的 32 条线路。通过命令线和协议细节来决定在某时刻这些线路中传递的是数据还是地址。

在总线设计中一个很重要的考虑因素就是控制机制。总线可以使用同步方式进行操作。在这种情况下，使用公共总线时钟线（和 CPU 时钟类似）在单个设备上对协议操作进行编排。总线也可以使用异步方式进行操作。总线控制器初始化总线操作；当主线受控器返回响应时表示总线操作完成。这种主从关系避免了对总线时钟的依赖。为了增加总线的利用率，高性能计算机系统使用分离传输总线（split transaction bus）。这个方案中，多个独立的信息交互可以同时进行。虽然这增加了总线的吞吐量，但同时在很大程度上增加了设计的复杂性。在计算机体系结构的很多高级课程中都会将类似的主题列在其中。

10.5 I/O 处理器

在企业级应用高性能的系统中，如网络服务器和数据库服务器，I/O 处理器的 I/O 任务与主处理器是分离的。I/O 处理器在没有打扰主处理器的情况下从处理器获取（或向处理器发送）一组设备的一系列命令，并执行这些命令。I/O 处理器减少了主处理器所经受的中断数。主处理器在共享内存中建立 I/O 程序（见图 10-8），并启动 I/O 处理器。I/O 处理器完成程序的执行，然后中断主处理器。

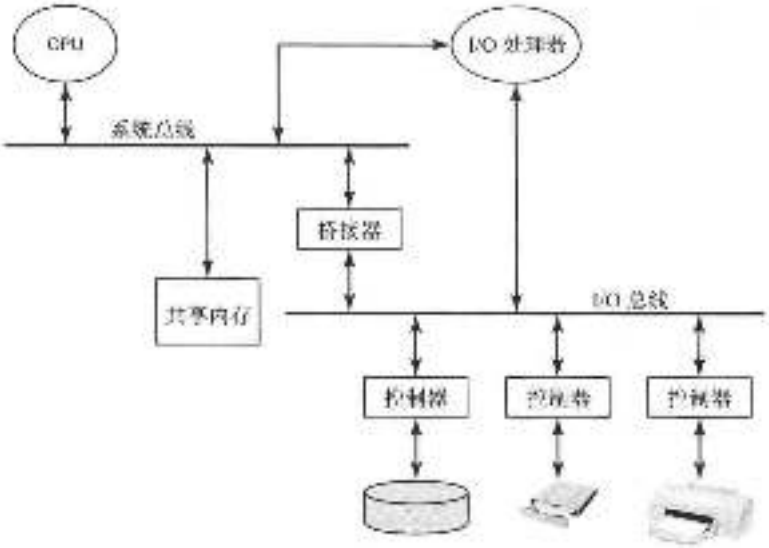


图 10-8 I/O 处理器负责与 I/O 相关的管理任务，尽量减少了主处理器计算任务中的中断次数

在大型机时代，IBM 推广 I/O 处理器。即使在今天，大型机在大的企业级服务器中仍很通用。IBM 将这些 I/O 处理器称为*信道*。多路复用信道（multiplexer channel）控制慢速面向字符的设备，如一组终端或显示设备。块多路信道控制多个中等速度的面向块的设备（面向流的设备，如本章中所提的设备），如磁带驱动器。选择器信道负责单独的、高速的面向流的设备，如磁盘。

I/O 处理器的功能与 DMA 控制器类似。然而，I/O 处理器的级别更高，因为它可以执行一系列的 CPU 命令（通过 I/O 程序）。另一方面，DMA 控制器以从属模式工作，一次处理一个来自 CPU 的命令。

10.6 设备驱动

设备驱动是操作系统的一部分，计算机系统中的每个设备都有设备驱动来进行控制。图 10-9 显示了系统软件的结构，特别是设备驱动和操作系统其余部分之间的关系，如 CPU 调度器、I/O 调度器和内存管理器。

434

设备驱动软件的细节由软件控制的设备特性所决定。例如，键盘驱动可以使用中断驱动的程控 I/O 在键盘控制器和 CPU 之间传输数据。另一方面，硬盘驱动器（磁盘）的设备驱动在磁盘控制器和内存之间创建 DMA 传输的描述符，并等待指示数据传输完成的中断。这里你可以感受到工作中抽象的优点，就相关的设备驱动而言，我们并不关心设备的具体细节。例如，设备可以是诸如键盘或鼠标这样的慢速设备。就设备驱动而言，执行的代码与 10.2 节中讨论的伪代码类似，即将数据从控制器中的设备寄存器转移到 CPU 中。类似地，高速设备可能是磁盘、扫描仪或摄像机。对于数据的传输，高速设备的设备驱动的执行情况与 10.3 节中伪代码的执行过程类似，即创建 DMA 传输的描述符，并令与设备相关的 DMA 控制器负责执行。当然，设备驱动需要兼顾特定于每个设备的控制功能。很明显，设备控制器与设备驱动联系很紧密。

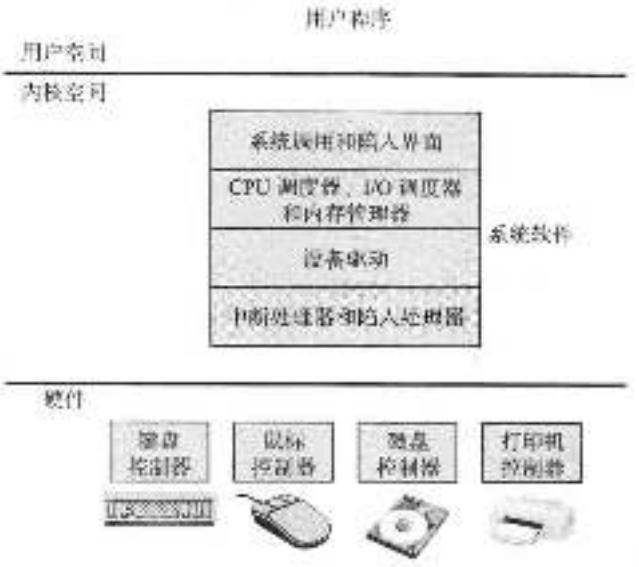


图 10-9 系统软件栈中设备驱动的位置。设备驱动作为操作系统的一部分与 I/O 调度器进行交互

435

10.6.1 例子

举一个更具体的例子，考虑云台全方位（上下左右）移动及镜头变倍、变焦控制（Pan-Tilt-Zoom，PTZ）摄像机的设备驱动。为了聚焦摄像机视图，设备控制器可以向 CPU 提供内存映射命令寄存器来指定控制功能，如缩放级别、移轴级别、摄像机前 x-y 坐标的空间。类似地，控制器提供摄像机开始和停止的命令。除了执行这些控制功能的命令外，控制器也为数据传输使用 DMA 设备。表 10-1 对 PTZ 摄像机的设备控制器功能进行了总结。

表 10-1 PTZ 照相机控制器的命令总结

命令	控制器操作
系摄（± □）	将摄像机摇摄角度 ± □
移轴（± □）	将摄像机的位置移动 ± □
缩放（± z）	将摄像机焦距缩放 ± z
开始	开始摄像
停止	停止摄像
内存缓冲区（M）	为传输到 M 的数据设置内存缓冲区地址
帧的数目（N）	设置需要拍摄和传入内存的帧的数目（N）
使能中断	使设备启用中断
禁用中断	使设备禁用终端
开始 DMA	在摄像机中开始 DMA 数据传输

这类设备的设备驱动可能包含如下模块，伪代码如下所示：

```
// device driver: camera
// The device driver performs several functions:
//    control_camera_position;
//    convey_DMA_parameters;
```

```
//    start/stop data transfer;
//    interrupt handler;
//    error handling and reporting;
// Control camera position
camera_position_control(angle pan_angle; angle tilt_angle; int z)
{
     pan(pan_angle);
     tilt(tilt_angle);
     zoom(z);
}
// Set up DMA parameters for data transfer
camera_DMA_parameters(address mem_buffer;int num_frames)
{
     memory_buffer(mem_buffer);
     capture_frames(num_frames);
}
// Start DMA transfer
camera_start_data_transfer()
{
     start_camera();
     start_DMA();
}
// Stop DMA transfer
camera_stop_data_transfer();
{
     // automatically aborts data transfer
     // if camera is stopped;
     stop_camera();
}
// Enable interrupts from the device
camera_enable_interrupt()
{
     enable_interrupt();
}

// Disable interrupts from the device
camera_disable_interrupt()
{
     disable_interrupt();
}
// Device interrupt handler
camera_interrupt_handler()
{
     // This will be coded similar to any
     // interrupt handler we have seen in
     // Chapter 6.
     //
     // The upshot of interrupt handling may be
     // to deliver "events" to the upper layers
     // of the system software (see Figure 10.9)
     // which may be one of the following:
     //       - normal I/O request completion
     //       - device errors for the I/O request
     //
}
```

给出上面关于设备驱动的简单实现是想给你信心，让你知道编写这样的软件和写其他的编程作业类似。我们应该指出现代设备可能要复杂得多。例如，现代的PTZ摄像机可能要与摄像机本身中的设备控制器交互，所以展现给计算机的接口级别更高。类似地，摄像机可以

直接接入局域网（我们会在第 13 章中进行讲解），所以通过使用网络协议栈与设备交互和在局域网中与对等计算机交互类似。

伪代码表达的主要思想是编写设备驱动的代码很容易。如果你有一定的编程经验，你肯定知道编写任何程序需要考虑特殊情况（例如，检查数组越界）和异常的处理（如在系统调用中检查返回的代码）。设备驱动的代码也是一样的。让设备驱动代码变得更有趣或者更有挑战性取决于你的态度，你可以考虑可能出现的与设备驱动代码逻辑不相关的各种情况。在设备驱动代码中需要考虑一些特殊情况，举例如下：

1）控制器命令中的参数是非法的（例如，摇摄、移轴、缩放和数据传输内存地址中的非法值）。

2）设备已经被其他程序使用。

3）因为某些原因设备没有做出响应（如设备电源没有打开或设备出现了故障等）。

由于人为因素，也会出现一些完全意想不到的情况，举例如下：

1）当数据正在传输时将设备从计算机上拔出。

2）当设备正在传输数据时将电源线从设备上拔出。

3）在传输数据时发生故障（比如，有人不小心将摄像机碰倒等）。

10.7 外围设备

历史上，将 I/O 设备划分为面向字符的设备[⊖]和面向块的设备。点阵打印机、阴极射线终端（CRT）和远程打印机都是前者的示例。这些设备一次输入/输出一个字符。因为这些设备的速度相对较慢，所以程控 I/O（PIO）是这些设备和计算机系统之间进行数据传输的可行方法。硬盘驱动器（磁盘）、CD-RW 和 MP3 播放器都是面向块的设备。正如名字所示，这些设备在设备和计算机系统之间以数据块为单位进行传输。例如，一旦激光打印机开始打印一页纸，它会不断地需要那一页中的数据，因为在打印页面的过程中没有办法暂停打印机。对于磁带驱动器也是同样的道理，一次从磁带中读或写一个数据块。这些设备的数据传输受 10.1.1 节提及的数据溢出的限制。所以，DMA 是处理这种设备和计算机系统之间有效数据传输的唯一有效方法。

426 ~ 438

表 10-2 对现代计算机系统中的典型设备进行了总结，包括数据传输速率（2008 年前后）和与 DMA 相比程控 I/O 的效率。第二列表示人为因素是否会影响设备的数据传输率。类似键盘和鼠标的设备与人的速度类似。例如，典型的打字员的打字速率是 300 ~ 600 字符/分钟，那么键盘的输入速率就是 5 ~ 10 字节/秒。类似地，移动鼠标的速率是 10 ~ 20 事件/秒，即输入速率是 80 ~ 200 字节/秒。处理器可以使用程控 I/O（轮询或中断）在没有数据损失的情况下对这些速率的设备进行处理。在大多数现代计算机系统中都有图形显示，图形显示的设备控制器中有设备驱动利用 DMA 传输进行更新的帧缓冲区。对于图像显示，1）屏幕分辨率为 1600 × 1200；2）屏幕刷新率为 60Hz；3）每个像素 24 位，数据传输率大于 300MB/s。一张播放 1 小时的 CPU 载有超过 600MB 的数据。一个播放 2 小时电影的 DVD 载有超过 4GB 的数据。这些技术都要能够从介质中以比实时速率更快的速率读取数据。例如，CD 以 50 倍于实时播放的速率进行读取，而 DVD 以 20 倍于实时播放的速率进行读取，数据传输率如表 10-2 所示。注意技术是不断变化的，所以这张表是 2008 年前后的技术指标，通过该表理解计算机外围设备如何与 CPU 进行交互。

⊖ 在 10.5 节，我们在没有给出正式定义的情况下引入了术语面向字符（character-oriented）和面向块（block-oriented）。

表 10-2 计算机外围设备的数据传输速率一览[1]

设备	输入 / 输出	人为影响	数据传输率（2008 年前后）	PIO	DMA
键盘	输入	是	5 ~ 10B/s	×	
鼠标	输入	是	80 ~ 200B/s	×	
图像显示	输出	否	200 ~ 350MB/s		×
磁盘（硬盘驱动）	输入 / 输出	否	100 ~ 200MB/s		×
网络（LAN）	输入 / 输出	否	1Gb/s		×
调制解调器[2]	输入 / 输出	否	1 ~ 8Mb/s		×
喷墨打印机[3]	输出	否	20 ~ 40KB/s	×[3]	×
激光打印机[4]	输出	否	200 ~ 400KB/s		×
声音（麦克风 / 扬声器）[5]	输入 / 输出	是	10B/s	×	
音频（音乐）	输出	否	4 ~ 500KB/s		×
闪存	输入 / 输出	否	10 ~ 50MB/s		×
CD-RW	输入 / 输出	否	10 ~ 20MB/s		×
DVD-R	输入	否	10 ~ 20MB/s		×

①以常见速的调制解调器的数据传输率是 2400b/s，这种速率对常中断的 PIO 或许是适用的。然而，对于支持大于 1Mb/s 的上行数据传输率和大于 8Mb/s 的下行数据传输率的现代电缆调制解调器而言，需要 DMA 传输来避免数据丢失。

②这意味着对于文本喷墨打印机的打印速率为 20 页 / 分钟（pages per min, ppm）；对于图像，为 2-4 ppm。

③喷墨打印技术允许在等待从计算机传入数据的过程中暂停打印。因为数据传输速率足够低，所以该设备适合使用 PIO。

④这意味着对于文本激光打印机的打印速率为 40 ppm；对于图像，为 4 ~ 8 ppm。

⑤典型情况下，演讲者的说话速率约为 120 词 / 分钟。

⑥感谢 UT-Austin 的 Yale Patt 和他的同事，感谢他们对外围设备速率所做的工作。

10.8 磁盘存储器

磁盘是很重要的外围设备，我们以磁盘作为具体的例子来进行学习。磁盘驱动器是科学技术发展的结果，最初是在有磁性的线上记录数据，之后发展为在带有磁性涂层的麦拉胶带上记录数据。磁带只允许顺序访问所记录的数据，为了增加数据传输率，也为了能够随机访问数据，逐渐转变为在转动的鼓状物上记录数据。之后进一步发展为在带有磁性的盘片上记录数据。

现代磁盘驱动通常由多个轻的非铁磁性的金属盘片构成，盘片的顶层和底层都涂有铁磁性的物质（见图 10-10），这样两个面就都可以用来记录数据。通过一个中央主轴将盘片连在一起，并以非常高的速率旋转（目前最先进的大容量驱动器的转速为 15 000 RPM 左右）。有一个读 / 写磁头（magnetic read/write heads）陈列，每面一个，磁头并不会触碰盘面。在磁头和盘面之间有一层细微的气隙（纳米级的气隙，比一粒烟尘还要小），允许磁头在盘片表面移动，不会触碰表面。如图 10-10 所示，每个磁头通过磁头臂与共用的固定轴相连。固定轴、磁头臂和附属的磁头共同构成了磁头组件（head assembly）。磁头臂机械地熔接在固定臂上，构成了单个对齐结构，这样所有的磁头可以做同样的移动，整齐（像摆动的门）地移入、移出磁盘。因此，所有磁头在相同的径向位置上的各自表面上同时处于同一位置。

控制磁头组件运动的执行器（actuator）的粒度和该技术允许的记录密度（recording density）决定了在表面上以磁道（track）的形式记录数据，每个磁道都和磁盘中心保持特定的径向距离。正如其名，磁道是盘片上由磁性记录材料构成的带状环形。而且，每个磁道都是

由扇区（sector）构成的。扇区是连续的多字节构成的记录信息，大小固定，构成了磁盘的基本记录单元（unit of recording）。换句话说，扇区是磁盘能够读或写的最小信息单元。磁盘外围的传感器对扇区进行划分。所有盘片对应的磁道形成了逻辑柱面（logical cylinder）（见图 10-11），柱面是由所有盘面对应的磁道累加构成的。我们将柱面作为逻辑实体，原因在稍后讨论磁盘访问所涉及的延迟和 I/O 操作时将会进行说明。因为即使最细微的灰尘落在盘片表面也会造成磁盘损坏，所以整个磁盘（盘片、磁头组件和传感器）是密封的。

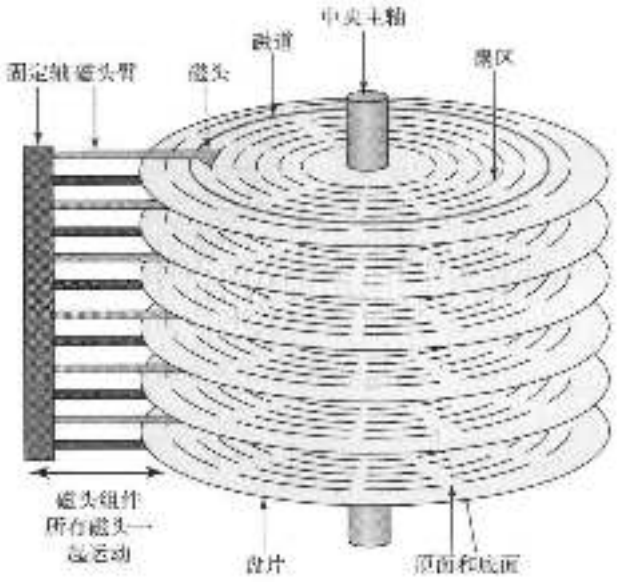


图 10-10 磁盘

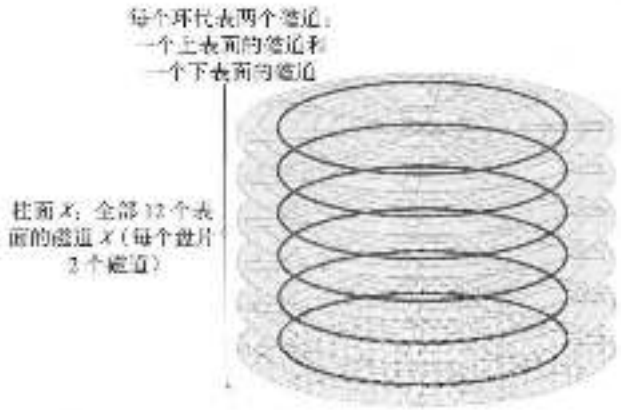
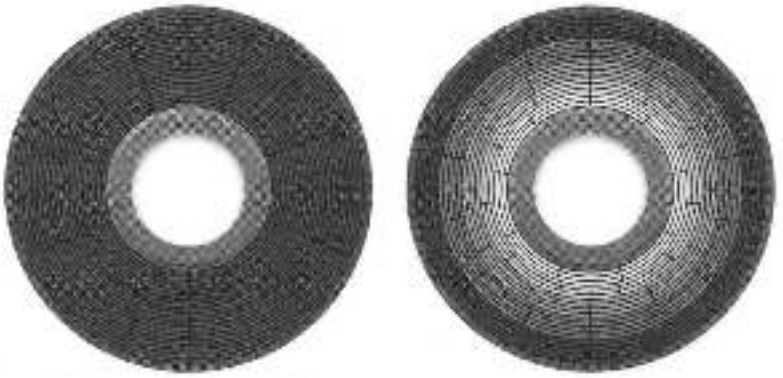


图 10-11 6 个盘片组成的磁盘的逻辑柱面

磁道是磁盘表面以盘心为中心的磁带。通常情况下，与内侧磁道相比，外侧的磁道圆周较大。所以外侧扇区和内侧扇区相比所占的面积也大（见图 10-12a）。正如前面所说，扇区有固定的大小来记录数据。为了对外侧磁道面积较大但扇区尺寸固定的情况进行缓解，早期的磁盘技术采用了降低外侧磁道记录密度的方法。这种方法不能将磁盘的有效空间完全发挥出来。

为了解决这个无法充分利用的问题，现代磁盘驱动器使用了称为区位记录（Zoned Bit Recording，ZBR）的技术。ZBR使表面每个扇区的面积大体相等。但外侧磁道的扇区比内侧磁道的扇区数目多（见图10-12b）。磁盘表面分为不同的区；不同区的磁道有不同数目的扇区，这样就能够更好地利用。



a）普通（非分区）记录方式　　b）分区记录方式

图10-12　非分区和分区记录方式的区别[⊖]

假设

p 是盘片的数目，

n 是每个盘片表面的数目（1或2），

t 是每个表面的磁道数，

s 是每个磁道的扇区数，

b 是每个扇区的字节数。

假设采取非分区记录结构，磁盘的总容量为

$$容量 = (p \times n \times t \times s \times b) 字节 \tag{10-1}$$

采用分区记录方式，

z 是分区的数目，

t_i 是分区 z_i 上的磁道数目，

s_n 是分区 z_i 上每个磁道的扇区数目。

采用分区记录方式，磁盘的总容量为

$$容量 = (p \times n \times (\sum (t_n \times s_n) \quad 1 \leqslant i \leqslant z) \times b) 字节 \tag{10-2}$$

例10-1 假设磁盘驱动器有如下配置：

- 每个扇区256字节
- 每个磁道12个扇区
- 每个表面有20个磁道
- 3个盘片

a. 若采用非分区的记录方式，每个驱动器的总容量是多少字节？

⊖ 图的来源：http://www.pcguide.com/ref/hdd/geom/tracksZBR-c.html. ©Charles M. Kozierok/ The PC Guide, PCGuide.com.

b. 另一种分区记录方式，参数如下：

- 分 3 个区
  - 区 3（最外侧）：8 个磁道，每个磁道 18 个扇区。
  - 区 2：7 个磁道，每个磁道 14 个扇区。
  - 区 1：5 个磁道，每个磁道 12 个扇区。

采取上面的分区记录方式，驱动器的总容量是多少？

答：

a. 总容量 = 盘片数目 × 面/盘片 × 磁道/面 × 扇区/磁道 × 字节/扇区

= 3 × 2 × 20 × 12 × 256 字节

= 360KB（K=1024）

439 / 443

b. 区 3 的容量 = 盘片数目 × 面/盘片 × 区 3 的磁道数目 × 扇区数/磁道 × 字节/扇区

= 3 × 2 × 8 × 18 × 256

= 216KB

区 2 的容量 = 盘片数目 × 面/盘片 × 区 2 的磁道数目 × 扇区数/磁道 × 字节/扇区

= 3 × 2 × 7 × 14 × 256

= 147KB

区 1 的容量 = 盘片数目 × 面/盘片 × 区 1 的磁道数目 × 扇区数/磁道 × 字节/扇区

= 3 × 2 × 5 × 12 × 256

= 90KB

总容量 = 所有分区的容量和 = 216 + 147 + 90

= 453KB（K=1024）

ZBR 的一个严重副作用是外侧与内侧磁道的数据传输率不同。与内侧磁道相比，外侧磁道有更多的扇区；磁盘的角速度和正在读的磁道无关，都是相同的。所以，与在内侧磁道相比，当磁头移动到外侧磁道时每次公转读出的扇区数目也更多。对磁盘进行空间分配时，倾向于首先使用外侧的磁道，然后再使用内侧的磁道。

磁盘上特定数据块的地址是三元组{柱面#，表面#，扇区}。向磁盘读或写数据需要如下几个步骤。第一，磁头组件移动到特定的柱面。完成这个移动的时间称为寻道时间（seek time）。我们可以发现寻找某一特定柱面和寻找柱面上的任何特定磁道的操作是相同的，因为柱面是与所有表面（见图 10-11）相关磁道的逻辑聚合。第二，磁盘需要通过转动将要求的扇区移动到磁头的下方。这一时间称为旋转延迟（rotational latency）。第三，当扇区在磁头下移动时，从选中的表面读取数据并传送给控制器，这一时间称为数据传输时间（data transfer time）。

这三部分构成了从磁盘读/写文件的总时间，其中寻道时间是最耗时的部分，其次是旋转延迟，最后是数据传输时间。典型的寻道时间和平均旋转延迟分别为 8ms 和 4ms。这些时间很长，是由磁盘子系统的机电特性所决定的。 444

下面介绍如何计算数据传输时间。注意磁盘在读/写时不会停止。就像在 VCR 中当磁头读取数据并将图像呈现在电视上的同时，磁带是保持不停转动的。当数据移动到磁头下面时磁盘保持不停地转动，磁头向表面读取（或写入）数据。数据传输时间可以从旋转延迟和介质的记录密度推导出来。你或许会想读/写介质本身会不会也需要时间。答案是需要；然而，这个时间是电磁时间，和磁盘旋转读取所需扇区的所有位造成的延迟相比是微不足道的。

数据传输率（data transfer rate）指当所需的扇区移动到读磁头的下方时，单位时间内传输的数据量。在 2008 年前后，数据传输率是 200 ~ 300MB/s。

设

r 是旋转速度，单位是每分钟所转的圈数（Revolutions Per Minute，RPM），

s 是每个磁道的扇区数目，

b 是每个扇区的字节数目，

转一圈的时间 $=60/r$ 秒，

每转一圈读取的数据量 $=s\times b$ 字节，

磁盘的数据传输率 = 磁道中的数据量 / 转一圈的时间 $=(s\times b)/(60/r)$。

$$\text{数据传输率}=(s\times b\times r)/60\text{B/s} \tag{10-3}$$

例 10-2 磁盘驱动器的参数如下：

- 每个扇区 512 字节
- 每个磁道 400 个扇区
- 每个表面 60000 个磁道
- 3 个盘片
- 旋转速度为 15 000RPM
- 使用记录方式

磁盘的传输率是多少？

445 答：

转一圈的时间 $=1/15\,000\text{m}=4\text{ms}$

磁道中的数据量 = 每个磁道的扇区数 × 每个扇区的字节数

$=400\times 512$

$=204\,800$ 字节

因为磁头每转一圈磁头读取一个磁道，所以

传输率 = 每个磁道中的数据 / 每次旋转的时间

$=(204\,800/4)\times 1000$ 字节 / 秒

$=51\,200\,000$ 字节 / 秒

对于特定请求的查询时间和旋转延迟取决于数据在磁盘上的具体位置。一旦磁头移到了需要的扇区上，读 / 写数据的时间就确定了，由磁盘旋转的速度决定。为了满足要求，在对磁盘驱动器的性能进行估计时很容易想起平均寻道时间（average seek time）和平均旋转延迟（average rotational latency）。假设请求均匀地分布在所有磁道上，则平均寻道时间是查找第一个磁道和最后一个磁道所用时间的平均值。同样，假设请求均匀地分布在磁道的所有扇区中，那么平均旋转延迟就是磁道中每个扇区的访问时间的平均值，正好是磁盘旋转延迟的一半。⊖

设

a 是以秒为单位的平均寻道时间，

r 是磁盘的旋转速度，每分钟转多少转（RPM），

s 是每个磁道的扇区数目。

$$\text{旋转延迟}=60/r\text{ 秒} \tag{10-4}$$

⊖ 在最好情况下，当寻道成功时所需的扇区正好在磁头下方；最坏情况下，磁头刚刚错过了所需的扇区，需要等待磁盘旋转一周。

$$平均旋转延迟 = (60/(r \times 2))\ 秒 \qquad (10\text{-}5)$$

一旦读磁头旋转到所要读的扇区上方，之后扇区的读取时间由磁盘的 RPM 决定。

扇区读取时间 = 旋转延迟 / 每个磁道的扇区数目

$$扇区的读取时间 = (60/(r \times s))\ 秒 \qquad (10\text{-}6)$$

为了随机读磁盘上的扇区，磁头需要查找到特定的扇区，之后在它读扇区前需要等待所需的扇区旋转到磁头下方。因此，随机读一个扇区由三部分构成：

- 寻道时间 = 平均寻道时间 = a 秒 446
- 查找扇区时间 = 平均旋转延迟 = $(60/(r \times 2))$ 秒
- 读一个扇区的时间 = 扇区读取时间 = $(60/(r \times s))$ 秒

读磁盘上的一个随机扇区的时间 = 寻道时间 + 将磁头移动到所需扇区上方的时间

$$+ 扇区读取时间 = a + (60/(r \times 2)) + (60/(r \times s))\ 秒 \qquad (10\text{-}7)$$

例 10-3 磁盘驱动有如下参数：

- 256 字节 / 扇区
- 12 扇区 / 磁道
- 20 磁道 / 面
- 3 盘片
- 平均寻道时间为 20ms
- 旋转速度为 3600RPM
- 采用普通的记录方式

a. 从同一磁道中读取 6 个连续的扇区的时间是多少？

b. 随机读取 6 个扇区的时间是多少？

答：

a. 平均寻道时间 =20ms

磁盘的旋转延迟 =1/3600m

=16.66ms

平均旋转延迟 = 旋转延迟 /2

=16.66/2

读一个扇区的时间 = 旋转延迟 / 每个磁道的扇区数目

=16.66/12ms 447

在同一磁道读 6 个连续的扇区，所需费的时间

= 平均寻道时间 + 平均旋转延迟 + 读 6 个扇区的时间

= 20ms+16.66/2+16.66/2

= 36.66ms

b. 对于第二种情况，我们需要分别计算每次的查找和读取时间。

所以，读取每个扇区花费的时间 = 平均寻道时间 + 平均旋转延迟 + 读一个扇区的时间

= 20+16.66/2+16.66/2

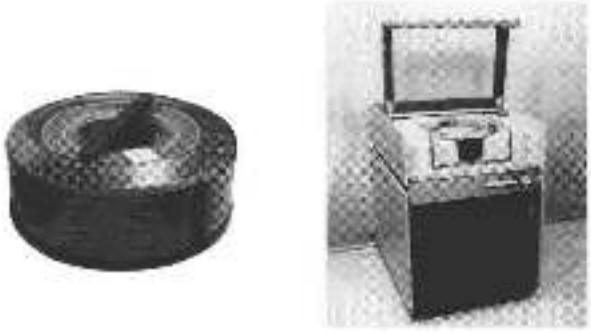
因此，读 6 个随机扇区的总时间

= 6 × (20+16.66/2+16.22/12)

= 178.31ms

10.8.1 磁盘技术的传奇故事

为了理解磁盘子系统的基本术语，我们一直讨论的是简单情况。在近二十年里，磁盘技术中的记录密度保持指数级的增长。例如，1980 年，20MB 认为是很大的磁盘存储容量。这时的磁盘有 10 个盘片，而且很笨重。驱动器本身看起来像个洗衣机（见图 10-13），并且介质可以从驱动器取出。



a) 可拆卸介质　　b) 磁盘驱动

图 10-13　磁性介质和磁盘驱动[⊖]

2008 年前后，台式机的存储容量达到了几百 GB 的水平。这样的驱动器将介质集成在里面。同样在 2008 年，台式机市场出现了小型硬盘（容量大概为 100GB ~ 1TB，直径为 3.5 英寸），有 2 ~ 4 个盘片，旋转速度约为 7200RPM，每个表面的磁道数为 5000 ~ 10 000，每个磁道有几百个扇区，每个扇区为 256 ~ 512 字节（见图 10-14）。



图 10-14　PC 的硬盘驱动器（2008 年前后）[⊜]

⊖ 图片来源：图 10-13a——©Barry Demchak；图 10-13b——美国明尼阿波利斯市，明尼苏达大学的查尔斯·巴贝奇研究所提供。

⊜ Western Digital 硬件驱动图片：可变的 RPM，1TB 容量，来源：http://www.wdc.com. 由 Western Digital 公司提供。

我们提供了计算磁盘数据传输率的简单方法和基于柱面-磁道-扇区概念的磁盘访问模型，这些模型已经不能满足现代技术的需要。原因很简单。本质上，磁盘与图 10-14 中的相同。当然，记录密度和 RPM 增加了；驱动器中盘片的尺寸和数目也显著增加了。但这些变化没有对磁盘访问和传输的基本模型造成影响。真正的变化在于对如下三方面进行了改进：驱动器电路、记录技术和接口。

第一个改进在于驱动器的内部电路。简单的模型假设磁盘以恒定的速率旋转。磁头移动到要求的磁道之后等待所需的扇区转动到磁头下方。这样很浪费能量。所以现代驱动器改变了 RPM，RPM 取决于需要读取的扇区，保证当磁头组件达到规定的磁道时所需的扇区正好在磁头下方。

另一个改进在于记录策略。图 10-15 显示了磁盘表面的横截图，说明了这一记录技术的优势。传统方式采用磁化水平平行的磁表面介质的方法来记录数据。这一技术有时称为纵向（longitudinal）记录（见图 10-15a）。最近发明了新的记录技术，垂直磁记录（Perpendicular Magnetic Recording, PMR），正如其名，采用磁化垂直磁表面介质的方法来记录数据（见图 10-15b）。关于这两种记录技术电子特性的研究已经超出了本书的范围。这里想让大家明白这一新技术极大地增加了磁盘表面单位面积所记录的数据密度。图 10-15 证明了这一点。可以发现 PMR 使记录密度变为原来的 2 倍，并且对于给定的磁盘规格，存储容量比纵向记录的容量要大。

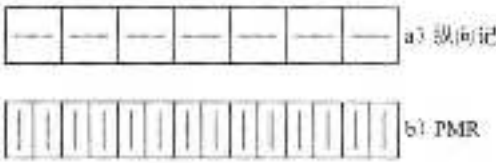


图 10-15 磁盘记录

现代驱动器的第二个变化在于将计算部分设计在了驱动器内部。今天，硬盘驱动器提供了与其他系统进行连接的更智能的接口。现在再也见不到磁盘控制器在驱动外面的结构了。今天的驱动器包含了控制器。你也许听说过 IDE（Integrated Drive Electronics，电子集成驱动器）、ATA（Advanced Technology Attachment，高级技术附件规格）、SATA（Serial Advanced Technology Attachment，串行高级技术附件）和 SCSI（Small Computer Systems Interface，小型计算机系统接口）等术语。这些都是现代智能接口的名字。

这些高级接口减少了 CPU 对磁盘的数据请求的工作量。驱动器的内部有一个微处理器负责对逻辑上连续的块（在早期也是物理上连续的）进行调度来获得对磁盘数据的最优访问时间。除了微处理器外，为了有效处理来自 CPU 的请求，驱动器中还有数据缓冲区，用于提前读入磁盘的扇区内容。微处理器还保存了来自 CPU 的内部请求队列，为了达到最优性能，会对请求队列重新进行排序。

磁盘驱动器的许多延迟是由读/写操作涉及的机械转动造成的。关于磁盘机电部分的讨论已经超出了本书的范围。我们关注于系统软件如何利用磁盘驱动器来对信息进行排序。磁盘分配策略应该尝试减少访问数据的寻道时间和旋转延迟。因为我们知道寻道时间是磁盘传输延迟中开销最大的部分，所以我们现在来详细阐述逻辑柱面的概念。如果我们需要保存一个可能要占多个磁道的大文件，我们应该将文件分布在同一面上相邻的磁道（adjacent tracks on the same surface），还是给定柱面相对应的磁道（corresponding (same) tracks of a given

448 ↓ 450

cylinder)？在现代磁盘技术中，很难回答这个问题。如果使用第一种方法，我们观察到对于给定文件会有多次寻道；对于第二种方法，观察到对于给定文件寻道一次可以查找文件的所有部分。很容易看出后者效果更好。这就是在磁盘子系统的上下文环境中识别基础抽象的原因。但由于磁盘中盘片的数目有限，通常为1或2，所以柱面概念的重要性也减弱了。

文件系统是下一章的内容，我们在下一章中介绍存储分配策略。

从系统吞吐量的角度，操作系统应该让磁盘以尽量降低旋转机械开销的方式对操作进行调度。在现代驱动器中，这种对请求进行重新排序的操作发生在驱动器内部。磁盘调度在下一节进行讨论。

10.9 磁盘调度算法

磁盘的设备驱动器嵌入了磁盘高效调度算法，对来自操作系统的请求进行调度。在第7章和第8章中我们看到，操作系统的内存管理器为了处理磁盘I/O的分页请求可能会发出它自己的命令。我们将在第11章中看到磁盘驱动器为终端用户保存文件系统。这样为了响应用户打开、关闭、读/写文件等请求，操作系统（通过系统调用）会发出磁盘I/O请求。因此，在任何时间点上，磁盘的设备驱动器都可能正在处理一些来自操作系统的I/O请求（见图10-16）。操作系统将这些请求按照生成时间进行排序。设备驱动器使用磁盘调度算法对这些请求进行调度，除其他事项外，每个请求都会指定在磁盘上保存指定数据的磁道。因为寻道时间是磁盘I/O操作中最耗时的部分，所以磁盘调度的主要目标是减少寻道时间。

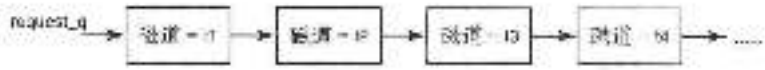


图10-16 按照到达时间排序的磁盘请求队列

[451] 我们首先假设单个磁盘接收到一系列的请求，并采取最有效的算法处理这些请求。之后假设只有一个磁头且寻道时间和遍历的磁道数目成比例。最后假设数据在磁盘上是随机分布的，且读和写花费的时间相同。

比较不同算法之间差异的典型度量是请求的平均等待时间（average waiting time）、等待时间的方差（variance in wait time）和总吞吐量（throughput）。平均等待时间和吞吐量是不言自明的术语，可以参见第6章中关于CPU调度的讨论。它们是以系统性能为中心的度量指标。从单个请求的角度看，等待时间的方差更有意义。这一度量指标告诉我们单个请求的等待时间与平均值之间的偏离的大小。与CPU调度类似，响应时间（response time）、或称为周转时间（turnaround time），从单个请求的角度看，是很有意义的度量指标。

在表10-3中，t_i、w_i和e_i分别表示对请求i的周转时间、等待时间和实际的I/O处理器时间。大部分指标和数学表达式都与第6章中关于CPU调度所讲的内容类似。

表10-3 对性能指标的总结

名称	表示公式	单位	描述
吞吐量	n/T	任务/秒	以系统为考虑中心，度量在时间T内处理n个I/O请求
平均周转时间（t_{avg}）	$(t_1+t_2+\cdots+t_n)/n$	秒	以系统为考虑中心，度量任务的平均完成时间
平均等待时间（w_{avg}）	$(0+e_1+(e_1+e_2)+\cdots+(e_1+\cdots+e_{n-1}))/n$或者$(w_1+w_2+\cdots+w_n)/n$	秒	以系统为考虑中心，度量I/O请求所经历的平均等待时间

（续）

名称	表示公式	单位	描述
响应时间/周转时间	t_r	秒	从用户为考虑中心，度量特定 I/O 请求 i 的周转时间
响应时间的方差	$E[(t_r-\bar{t}_r)^2]$	秒^2	从用户为考虑中心，度量 I/O 请求 i 的实际响应时间 (t_r) 与期望值 $(\bar{t}_r)$ 的统计方差
等待时间的方差	$E[(w_i-\bar{w}_i)^2]$	秒^2	从用户为考虑中心，度量 I/O 请求 i 的实际等待时间 (w_i) 与期望值 $(\bar{w}_i)$ 的统计方差
饥饿状态			从用户为考虑中心，当为有些 I/O 调度者的内在特性对特定 I/O 请求或一组 I/O 请求提供的服务

152

我们重新复习磁盘调度的 5 种不同算法。为了使讨论更简单、更具体，我们假设磁盘有 200 个磁道，标号为 0 ~ 199（0 是最外层的磁道，199 是最内层的磁道）。磁头完全收回的位置是磁道 0。当磁头组件处在磁道 199 时就达到了它的最远距离。

你会发现这些磁盘调度算法与第 6 章中的 CPU 调度算法类似。

10.9.1 先到先服务

正如名字所示，算法按照请求的到达时间来进行服务。从这一点上说，它和 CPU 调度策略的 FCFS 算法类似。这个算法有一个很好的特性，不论 I/O 请求的是哪个磁道，等待时间的方差最小。但这是唯一的优点。从系统的角度看，对于大多数工作负载，这个算法会导致较差的吞吐量。图 10-17 说明了磁头如何按照 FCFS 调度策略在盘面来回移动处理请求，特别是当 FCFS 的请求要访问距离很远的磁道时。

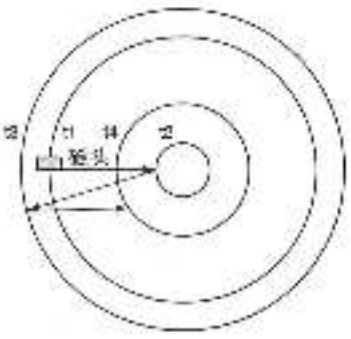


图 10-17 FCFS 的磁头的移动

10.9.2 最短寻道时间优先

这种调度策略和 SJF 处理器的调度策略类似。基本思想是优先处理那些离磁头位置最近磁道上的任务（见图 10-18）。正如 SJF 用于处理器调度一样，对于特定的一组请求，最短寻道时间优先（SSTF，Shortest Seek Time First）会保证有最小的平均等待时间并输出较好的结果。然而，和 SJF 类似，因为请求可能与正在处理的大量请求的距离较远，所以 SSTF 有可能出现饥饿请求。和 FCFS 相比，这个调度策略的方差较大。

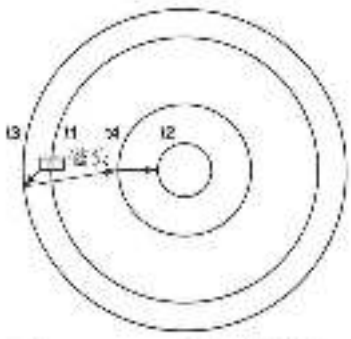


图 10-18 SSTF 策略磁头的移动

10.9.3 SCAN

这个算法和磁头组件的机电特性相符合。基本思想如下：磁头从当前位置（磁道 0）向最内层磁道（磁道 199）移动。随着磁头的移动，在从最外层磁道向最内层磁道移动的过程中，算法不考虑到达时间，处理途中遇到的请求。一旦磁头到达了最内层磁道，它就反向向最外层磁道移动，并处理途中遇到的请求。只要请求队列不为空，算法就会不断重复这个过程。图 10-19 展示了这个算法的思想。

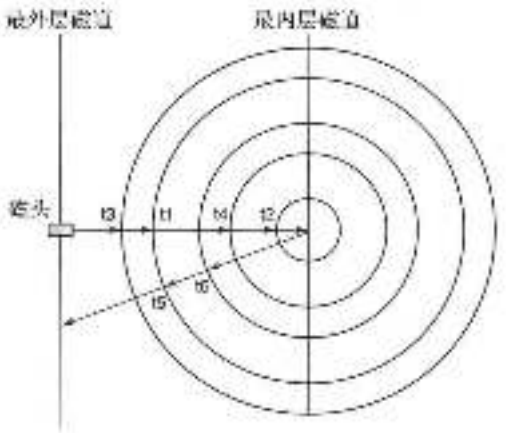


图 10-19 SCAN 算法磁头的移动

453
:
454

请求 t1、t2、t3 和 t4 在磁头向前移动时就已存在（如图 10-16 所示）。在磁头到达最内层磁道之后出现了请求 t5 和 t6（按顺序出现）。算法按照磁头反向遍历的顺序处理这些请求。在等电梯时应该知道都发生了什么，这个算法处理请求时就像等电梯，所以 SCAN 算法经常称为电梯（Elevator）算法。和 SSTF 相比，SCAN 算法等待时间的方差较小，平均等待时间与 SSTF 类似。与 SSTF 类似，SCAN 算法并不保存请求的到达顺序。然而，和 SSTF 相比，SCAN 算法有一个根本区别，SSTF 可能随机地让某个给定进程处于饥饿状态。另一方面，SCAN 算法违反先来先服务的公平特性是有上界的。上界是磁头从一端移动到另一端的移动时间。所以 SCAN 算法也避免了请求出现饥饿的状态。

10.9.4 C-SCAN

循环扫描（C-SCAN，Circular Scan）是 SCAN 算法的变形，算法将磁盘表面看成逻辑的圆。所以一旦磁头到达了最内层的磁道，算法就会将磁头组件移动到另一侧，并重新进行扫描。换句话说，算法在磁头反向移动过程中并不处理任何请求。如图 10-20 所示，对 SCAN 算法中出现的同样请求，形象地进行了说明。

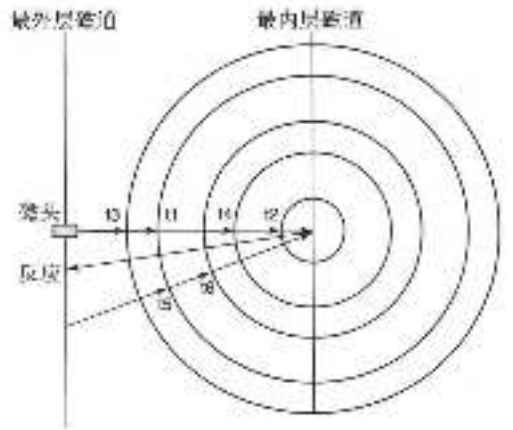


图 10-20 C-SCAN 算法磁头的移动

通过忽略反向运动所遇到的请求，C-SCAN 消除了对 SCAN 算法中磁盘中间磁道对大量请求进行处理的不公平方法。这个算法减少了处理请求时的不公平性（注意它是如何处理 t5 和 t6 的），并且和 SCAN 算法相比，它降低了等待时间的方差。

10.9.5 LOOK 和 C-LOOK

这两个策略与 SCAN 算法和 C-SCAN 算法类似，但当磁头移动方向上没有请求时磁头组件会立即改变移动方向。即磁头组件不会不必要地移动到最外侧或最内层。移动策略与电梯工作方式类似。因为避免了不必要的机械运动，所以这些算法比 SCAN 算法和 C-SCAN 算法要好。即使这样，历史上还是将 SCAN 算法称为电梯算法，LOOK 算法和大多数现代电梯系统的服务模式类似。图 10-21 显示了 LOOK 算法和 C-LOOK 算法处理 SCAN 算法和 C-SCAN 算法例子中出现的请求序列。注意磁头在处理后面的未完成序列时的位置变化。当没有需要处理的请求时，磁头会停在最后处理的请求位置。这就是 LOOK 算法与 SCAN 算法，C-LOOK 算法与 C-SCAN 算法的主要区别。 455

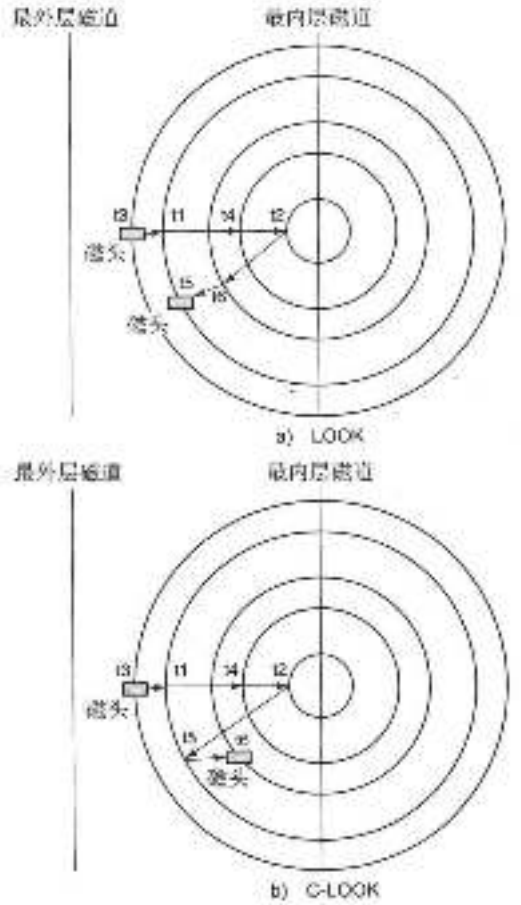


图 10-21 LOOK 算法和 C-LOOK 算法磁头的移动

10.9.6 磁盘调度总结

调度算法的选择取决于很多因素，包括预期的布局、存储分配策略和磁盘驱动器的机电

特性。通常情况下，磁盘调度使用 LOOK 算法和 C-LOOK 算法的变形。我们介绍其他的算法是为了更好地进行讨论，而不是在实际系统中将它们作为可选用的方案。

正如我们在 10.8.1 节中提到的，现代磁盘驱动器向 CPU 提供非常复杂的接口。这样，驱动器上块的内部布局对于磁盘设备驱动器甚至是不可见的，它是操作系统的一部分。假设接口允许出现来自设备驱动器的多个未完成的请求，那么控制器本身会对前述磁盘调度算法进行具体应用。

例 10-4 说明了各种调度算法之间的区别。

例 10-4 有如下情况：

磁盘上的柱面数目　　=200（编号为 0 ～ 199）

当前磁头位置　　=柱面 23

当前请求的到达序列　　=20,17,55,35,25,78,99

给出对于上述请求不同磁盘调度算法的调度结果。

答：

a. 对于给定请求使用 C-LOOK 算法的调度安排：

25,35,55,78,99,17,20

b. SSTF 算法的调度安排：

25,20,17,35,55,78,99

456
～
457

c. LOOK 算法的调度安排：

25,35,55,78,99,20,17

d. SCAN 算法的调度安排：

25,35,55,78,99,199,20,17,0

e. FCFS 算法的调度安排：

20,17,55,35,25,78,99

f. C-SCAN 算法的调度安排：

25,35,55,78,99,199,0,17,20

10.9.7 算法比较

我们还用例 10-4 中的请求序列来比较算法的不同。按照到达顺序，我们有 7 个请求：R1（柱面 20）、R2（柱面 17）、R3（柱面 55）、R4（柱面 35）、R5（柱面 25）、R6（柱面 78）和 R7（柱面 99）。

我们关注请求 R1。我们选择经历的磁道数作为比较分析的响应时间。因为上例中磁头开始的位置是 23，所以不同算法对 R1 的响应时间为

- T_1^{FCFS}=3（先处理 R1）。
- T_1^{SSTF}=7（先处理 R5，之后处理 R1）。
- T_1^{SCAN}=355（磁头运动轨迹按照例 10-4 中的（d）来进行计算）。
- $T_1^{C\text{-}SCAN}$=395（磁头运动轨迹按照例 10-4 中的（f）来进行计算）。
- T_1^{LOOK}=155（磁头运动轨迹按照例 10-4 中的（c）来进行计算）。
- $T_1^{C\text{-}LOOK}$=161（磁头运动轨迹按照例 10-4 中的（a）来进行计算）。

表 10-4 对例 10-4 的访问序列在不同磁盘调度算法（FCFS、SSTF 和 LOOK）下的响应时间进行了对比（以磁头的移动为单元）。

吞吐量定义为完成的请求数目与处理完所有请求所遍历的磁道数目的比值。

- FCFS=7/148=0.047 请求/磁道
- SSTF=7/92=0.076 请求/磁道
- LOOK=7/158=0.044 请求/磁道

从前面的分析中可以看出，SSTF 在平均响应时间和吞吐量方面表现得最好。但这是以公平性为代价的（在 SSTF 列中将 R5 的响应时间与早先的请求 R1 ~ R4 进行对比）。而且，SSTF 有可能出现饥饿状态。乍一看，LOOK 算法在表中的响应时间最坏。然而，还有几点需要注意。第一，响应时间对磁头的初始位置和请求的分布情况很敏感。第二，有可能选取的样例是病态的（或者说，是不合适的），正好适合某个特定算法。第三，在这个预先制定好的例子中，请求序列在处理过程中不发生变化。实际中，新的请求可能会加入队列中，并对吞吐量和响应时间造成影响（见练习题 13）。

表 10-4 例 10-4 调度算法的定量比较

请求	响应时间		
	FCFS	SSTF	LOOK
R1（柱面 20）	3	7	155
R2（柱面 17）	6	10	158
R3（柱面 33）	44	48	32
R4（柱面 35）	64	28	12
R5（柱面 25）	74	2	2
R6（柱面 78）	127	71	55
R7（柱面 99）	148	92	76
平均值	66.4	36	70

总体上看，如果请求均匀地分布在磁盘上，那么 LOOK 的平均响应时间接近 SSTF 的平均响应时间。更重要的是，表中没有反映出 FCFS 和 SSTF 内在地改变磁头组件方向所花费的时间和能量。这可能是将 LOOK 作为更好的磁盘调度选择的最重要的考虑因素。

利用表 10-3 中总结的公式，读者通过例 10-4 的练习可以很好地将所有磁盘调度算法进行比较。

经过多年发展，磁盘调度算法已经得到了广泛的研究。正如我们早先所观察到的（见 10.8.1 节），磁盘驱动技术的发展很快。正因为这些发展，所以对于每种新的磁盘，迫切需要对磁盘调度算法进行重新评估。[⊖]目前有些 LOOK 算法的变种证明它是所有选择中性能最好的。

10.10 固态硬盘

硬盘技术的一个基本限制是它的机电特性。经过多年发展，一些新的存储技术出现了替换磁盘作为存储的趋势，但目前还没有实现。主要原因是，和这些新技术相比，磁盘存储每字节的价格较为低廉。

458
~
459

威胁硬盘相对垄断地位的技术是固态硬盘（Solid State Drive，SSD）。这项技术的起源可以追溯到电可擦可编程只读存储器（Electrically Erasable Programmable Read-Only Memory，EEPROM）。在第 3 章中，我们引入了 ROM 作为一种内容为非易失性（nonvolatile）的固态内

⊖ 见例子，http://www.ece.cmu.edu/~ganger/papers/sigmetrics94.pdf.

存，即在供电周期中内容是保持不变的。将这个技术打个比方可能更好理解。你应该见过转辙机，如图 10-22 所示。一旦转辙机打开，传入的轨道（图的底部）仍然保持连接所选的分叉。ROM 也是以同样的方式工作。



图 10-22 转辙机

图 10-23 描述了一个简单的类似转辙机的电子部件。如果图中间的开关打开，那么输出就是 1；否则，输出就是 0。这就是 ROM 的基本构件。开关通过基本的逻辑门实现（NAND 或 NOR）。将这些开关封装在集成电路的内部就是 ROM。根据期望的输出，开关可编程输出 0 或 1。这就是电路称为可编程只读存储器（Programmable Read-Only Memory，PROM）的原因。

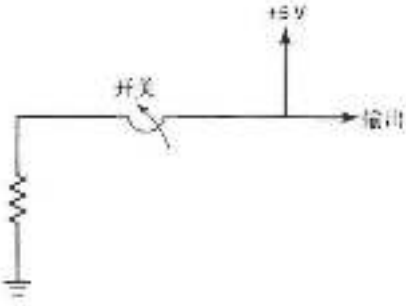


图 10-23 电子开关

这项技术之后又有了新的发展，将 ROM 中的位模式改为电可擦除和可编程的，这样就可以包含不同位的位模式。这就是 EEPROM 技术。虽然这项技术和 RAM 的特性非常相似，但它们有很大的区别。RAM 中的读/写粒度是可选的。由于 EEPROM 技术的电子特性，EEPROM 中的擦除每次会对一个块中所有的位进行擦除操作。与 RAM 中的读和写相比，这样的写要花费几个数量级以上的时间。所以 EEPROM 不能取代 DRAM 技术。然而，这项称为闪存（flash memory）的技术可以用在便携式记忆卡上，并且可以作为存储嵌入手机和 iPod 中。

还有一个问题可能会困扰你，你可能会想为什么闪存没有替代台式机和笔记本电脑中的磁盘作为永久性存储。毕竟作为完全的固定状态，这项技术没有磁盘技术的内在问题（由于磁盘的机电特性延缓了数据的访问时间）。

硬盘在三个方面仍然具有优势：较高的存储密度（导致每字节的价格较低）、较高的读/写带宽和更长的寿命。最后一点需要进行说明，SSD 技术本身也有内在限制：存储的每个给定区域只能重写特定的次数。这意味着对同一块进行频繁地写操作会导致不均匀的磨损，这样也缩短了存储作为整体的寿命。通常，SSD 的生产商采用损耗均衡（wear leveling）技术来避免这一问题。损耗均衡的目的是为了将频繁写入的块重新布局在存储的不同区域里。相对于普通存储系统的工作负载，这增加了额外的读/写周期。

SSD 技术仍然在不断发展，弥补这些不足。例如，2008 年前后，市场上出现了容量为 100GB、传输率为 100MB/s 的 SSD 设备。对于笔记本电脑中存储要求不高的领域，有些生产商也将 SSD 作为大容量存储。但是，2010 年前后，基于 SSD 的大容量存储的价格仍明显高于对应的基于磁盘的大容量存储的价格。

10.11 I/O 总线和设备驱动的演化

从 PC 出现的那一刻起就一直有将外围设备与计算机进行连接的需求。虽然表 10-4 显示了设备对 CPU 的影响，但很少出现设备直接与 CPU 连接的情况。这是因为外围设备是由第三方供应商制造的。[9]第三方供应商（比如，生产磁盘的 Seagate 和生产摄像机的 Axis）有别

460
461

⊖ 我们已经介绍了这个术语，但在 10.4 节没有定义它。

于IBM、Dell和Apple等计算机生产商，它们生产外围设备和与外围设备配套的设备驱动器。这些第三方供应商并不知道使用这些设备的计算机的内部信息，所以将设备设计成能够连接任何生产商生产的计算机。你可能听说过即插即用（plug and play）的概念，这意味着外围设备能够在不改变计算机任何内部构造的情况下与计算机系统进行连接。这一特性就是发展10.4节中PCI等标准的主要原因。

现代计算机爱好者一定都听过类似USB和火线（Firewire）的术语。让我来解释这些术语都是什么意思。正如我们在10.4节中所提，PCI总线是使用地址、数据和命令多路复用的32位并行总线。USB的意思是通用串行总线（Universal Serial Bus，USB），USB和火线是外围设备与计算机之间串行接口的两个有竞争性的标准。你可能会想，既然并行接口更快，为什么要使用串行接口与计算机连接。实际上，如果我们回顾历史，就会发现只有慢速的面向字符的设备（例如，阴极射线管，或称为CRT，通常称为哑终端）与计算机系统进行串行连接。

你知道信号在信号线上的传输最终受到光速限制。而实际的数据传输率，如表10-2所示，现在接近这个速率。由于线路的延迟是单一线路更快传输数据的主要限制因素，并行化有助于打破这一限制，通过在并行线路上传输数据提高总体的吞吐量。然而，随着技术的提高，线路的延迟也减少了，也可以有更高的信号产生频率。在这种情况下，串行接口的优势比并行接口大。首先，由于减少了线路和连接器的数目，所以会更小更便宜。其次，并行接口在速率较高时，如果没有对并行线路进行有效的防护，会发生串扰。另一方面，通过有效的波形整形和滤波技术，很容易在较高频率下操作串行信号。这也是现在串行接口实际上比并行接口速度更快的原因。

所以，将高速设备与计算机系统串行相连成了标准。这就是发展诸如USB和火线标准用于将外围设备与计算机系统进行连接的原因。因此，你可能注意到大多数现代笔记本电脑不支持任何并行接口。2007年，即使是并行打印机接口也从笔记本电脑上消失了。串行接口标准增强了现代外围设备的即插即用特性。

你可能会好奇对于串行接口为什么会有两个有竞争性的标准。这又是因为不同的计算机生产商都想占据更多的市场份额。Microsoft和Intel提倡USB，Apple公司提倡火线。今天，这两个串行接口都成为了用于连接慢速和快速外围设备的工业标准。USB 1.0能够达到1.5MB/s的数据传输率，通常用于对键盘和鼠标等慢速I/O设备进行连接。火线能够支持100MB/s的速率，通常用于类似电子摄像机等的多媒体高增值电子设备。USB 2.0支持的速率可达60MB/s，所以火线和USB之间的区别变得有些模糊了。

为了完成I/O总线的讨论，我们还需注意另外两项针对PC产业I/O体系结构的增强技术。高级图形接口（Advanced Graphics Port，AGP），是连接3D图形控制器和主板的专用通道。对于3D图形处理，在PCI总线上与其他设备共享带宽是不够的，特别是对于可交互的游戏。对3D图形处理更高带宽的需求促成了AGP通道的发展。最近，AGP很大程度上被PCI Express（PCI-e）取代，这是另一个新的提供主板和图形控制器之间连接的标准。这些标准更详细的电子特性和差异超出了本书的范围。 462

10.11.1 设备驱动的动态负载

设备是即插即用的，所以用设备驱动来控制它们。考虑数字摄像机的设备驱动。设备驱动不需要一直作为系统软件的一部分（见图10-9）。在Linux和Mircrosoft Vista等操作系统中，当相关设备上线时，设备驱动动态地链接到系统软件。当设备接入时操作系统通过设备

中断识别新设备（例如，当你将摄像机或快速记忆棒插入 USB 接口时）。操作系统会找设备列表并识别刚接入系统的设备（大多数情况下，设备供应商和开发设备驱动的操作系统供应商是有合作的）。之后为了控制设备将设备驱动动态地接并装入内存中。当然，如果接入的设备没有合适的驱动，那么操作系统只能让用户提供新接入系统设备的驱动。

10.11.2 信息汇总

随着计算机系统高级接口的出现，高速设备和低速设备变得离 CPU 本身的距离越来越远。所以程控 I/O 几乎成为过去。在 10.5 节中，我们提到了 IBM 在 20 世纪 60 年代和 70 年代提出了关于大型机 I/O 处理器的创新。现在这些概念已经体现在了你的计算机中。

主板（motherboard）是 PC 时代出现的术语，它意味着中央计算机系统电路的出现。它是单个印刷电路板，包含处理器、内存系统（包括内存控制器）和用于连接外围设备与 CPU 的 I/O 控制器。主板名字的由来是因为印刷电路板上有许多插槽，这些插槽（通常称为子卡）可以通过插入设备来对计算机系统进行扩充。例如，物理内存的扩充就是通过这种方式实现的。图 10-24 是现代主板的示意图。图中很清楚地表示了组件和它们的特性。你可以看到能够插入外围设备控制器子卡的插槽和插入 DIMMS（见第 9 章关于 DIMMS 的讨论）的插槽。

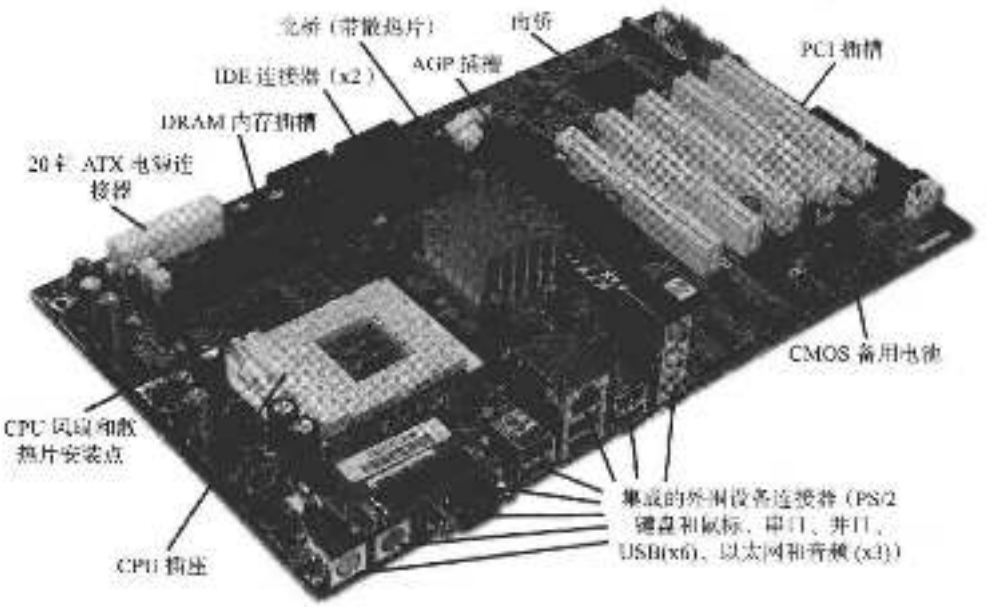


图 10-24 主板图片（ASRock K7VT4A Pro）[⊖]

图 10-25 显示了现代主板中重要电路单元的框图。理解这些单元很重要。每个计算机系统都需要在上电时自动执行一些低级代码。正如我们所知，处理器只是简单地执行指令。关键是将计算机系统引入操作系统对全部资源进行控制的状态。你可能听说过启动（booting up）操作系统这个术语。该术语是通过引导程序（bootstrapping）的简称，暗指通过自己的程序进行启动。当上电时，处理器自动执行只读存储器（Read-Only Memory，ROM）中已知固定位置的引导程序。代码会完成系统所有的初始化操作，包括在将控制转移至图 10-9 中的系

⊖ 图片来源：http://en.wikibooks.org/wike/File:ASRock_K7VT4A_Pro_Mainboard_Labeled_English .svg。

统软件的上层前识别外围设备。在 PC 中，引导程序代码称为 BIOS，是基本输入/输出系统（Basic Input/Output System 的缩称）。

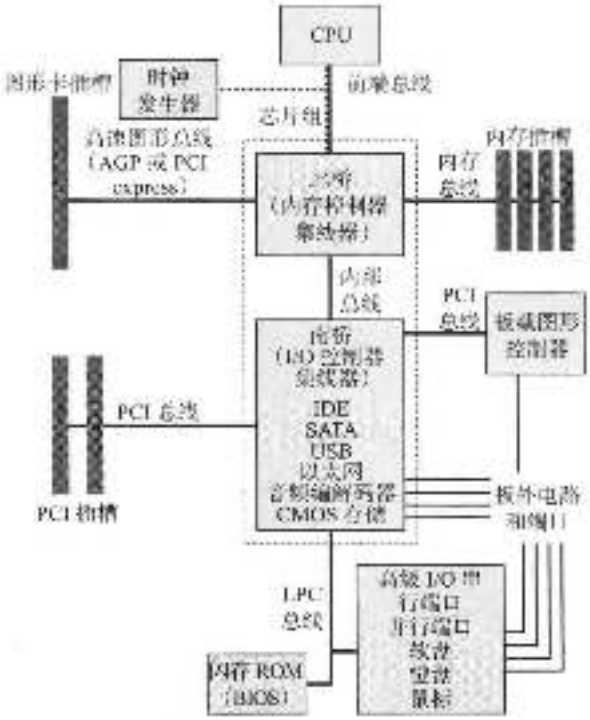


图 10-25 典型主板的框图

关于图 10-25，有以下几点值得注意：

- 标有北桥（Northbridge）的单元是用于编排 CPU 和内存系统包括 I/O 控制器之间通信的集线器（hub）芯片。
- 类似地，标有南桥（Southbridge）的单元是用作 I/O 控制器枢纽的芯片。它和本节讨论的标准 I/O 总线相连接，包括 PCI 和 USB，它对设备进行仲裁，包括总线、通过北桥的直接访问需求和通过 CPU 进行的中断服务。它嵌入了 10.5 节中关于 I/O 处理器讨论的许多功能。
- PCI Express 是另一个支持设备所需的高传输率和响应时间的总线标准，例如高分辨率的图形显示。
- LPC 代表低引脚数（Low Pin Count），LPC 总线是另一个低带宽设备（如键盘和鼠标）与 CPU 连接的标准。
- 标有高级 I/O 的单元是负责一些慢速设备 I/O 控制器的芯片，慢速设备包括键盘、鼠标和打印机。

正如我们所讨论的，计算机系统中的硬件是非常令人着迷的。虽然我们在本节将 PC 作为具体的例子，但在图 10-25 中单元的功能适用于任何计算机系统。曾经有一段时间根据机器的种类不同，从像 IBM PC 这样的个人计算机到像 Cray-1 这样的向量超级计算机，计算机系统内部构造有非常大的不同。随着单芯片微处理器技术的进步，如我们在第 5 章中所讨论的，

463, 465 以及根据摩尔定律所能达到的集成密度（见第 3 章），在用于构成计算机系统的构件上有了共识，范围从 PC 到台式机、到服务器、到超级计算机。

小结

在本章中，我们讨论了如下主题：

1）处理器和 I/O 设备之间的通信机制，包括程控 I/O 和 DMA。

2）设备控制器和设备驱动。

3）现代计算机系统中的常见总线，特别是 I/O 总线。

4）磁盘存储器和磁盘调度算法。

在下一章我们将学习文件系统，文件系统是构建在所有稳定性存储上，特别是硬盘上的软件子系统。

练习题

1. 请将程控 I/O 与直接内存访问（DMA）进行比较。
2. 假设磁盘驱动器有如下参数：
   - 表面的数目 =200
   - 每个表面的磁道数目 =100
   - 每个磁道的扇区数目 =50
   - 每个扇区的字节数目 =256
   - 速率 =2400 RPM

   磁盘的总容量是多少？

   平均旋转延迟是多少？
3. 磁盘有 20 个表面（即 10 个双面盘片），每个表面有 1000 个磁道，每个磁道有 128 个扇区，每个扇区有 64 字节，磁盘分配策略为每个文件分配连续多个柱面。

   每个柱面有多少字节？

   如果需要加载 5MB 的文件，那么需要多少个柱面？

   需要给这个 5MB 文件分配多大的空间？
4. 磁盘有如下参数：
   - 磁盘容量 310MB
   - 磁道大小：4096 字节
   - 扇区大小：64 字节

   程序员有 96 个对象，每个对象的大小为 50 字节。如果决定将每个对象保存为单独的文件，实际写入磁盘的总字节数是多少？
5. 描述 DMA 数据传输中操作序列的细节信息。

466 6. 在磁盘驱动器能够读数据前，需要做哪些机械操作？

7. 磁盘驱动器有 3 个双面的盘片，驱动器有 300 个柱面，每个表面有多少个磁道？
8. 假设磁盘驱动器有如下的参数：
   - 每个扇区 512 字节
   - 每个表面 30 个磁道
   - 2 个盘片
   - 区位记录：

- 有 3 个区

  区 3（最外圈）：12 个磁道，每个磁道 200 个扇区

  区 2：12 个磁道，每个磁道 150 个扇区

  区 1：6 个磁道，每个磁道 50 个扇区

  按照上述双位记录方法，驱动器的总容量是多少？

9. 假设磁盘驱动器有如下的参数：
   - 每个扇区 256 个字节
   - 每个磁道 200 个扇区
   - 每个表面 1000 个磁道
   - 2 个盘片
   - 旋转速度是 7500 RPM
   - 普通记录方式

   磁盘的传输率是多少？

10. 假设磁盘驱动器有如下参数：
    - 每个扇区 256 字节
    - 每个磁道 100 个扇区
    - 每个表面 1000 个磁道
    - 3 个盘片
    - 平均寻道时间是 8ms
    - 旋转速度为 15 000 RPM
    - 普通记录方式

    a. 从同一个磁道读取 10 个连续的扇区需要多长时间？

    b. 随机读取 10 个扇区需要多长时间？

11. 磁盘调度算法的目标是什么？

12. 用遍历的磁道数作为度量时间的标准，查看表 10-3 中总结的不同性能指标，比较对于例 10-4 中的请求模式，所有磁盘调度算法的性能表现。 467

13. 假设磁盘的细节信息与例 10-4 一样。请求队列不是保持不变而是随着新请求添加到队列中会发生变化。在任何时间，算法都根据当前的请求来决定处理哪一个请求。考虑下面的请求序列：
    - 初始时（在时间 0），队列包含柱面 99,3,25 的请求。
    - 在轮转算法决定接下来处理那个请求时，新的请求加入了队列：46。
    - 下一个决策点，新的请求加入了队列：75。
    - 下一个决策点，新的请求加入了队列：55。
    - 下一个决策点，新的请求加入了队列：85。
    - 下一个决策点，新的请求加入了队列：73。
    - 下一个决策点，新的请求加入了队列：50。

    假设磁头在时间 0 正在处理磁道 55 的请求。

    a. 给出 FCFS、SSTF、SCAN、C-SCAN、LOOK 和 C-LOOK 的调度。

    b. 对于每个前面的请求，计算响应时间（以磁头遍历的单元数为度量标准）。

    c. 每种算法的平均响应时间和吞吐量是多少？

参考文献注释和扩展阅读

为了获取最先进存储技术的信息，最好访问那些技术上领先公司的网页。IBM、MAXTOR 和

Seagate 都是技术先进的磁盘生产商。像 Samsung 和 Intel 在基于闪存技术的 SSD 市场上都处于领先地位，通过它们的网页可以获得最先进的SSD 存储技术的信息。为了更好地了解磁盘缓存设计的考虑因素，我们推荐读者阅读 Alan Jay Smith[Smith，1985]开创性的论文。有些书在不同程度上介绍了 I/O 的不同方面。Hennessy and Patterson[Hennessy，2006]在存储系统设计方面提出了先进的观点。Tanenbaum[Tanenbaum，2007]从在操作系统中管理它们（即对其编写程序）的角度表述了 I/O。Bryant and O'Hallaron[Bryant，2003]从用户级编程的角度表述了 I/O。Silberschatz et al.[Silberschatz，2008]对大容量存储设备的结构（包括磁盘调度算法）进行了很好的总结，包括操作系统中管理 I/O 设备的软件
468 设计方案。

文件系统

在本章，我们将讨论与大容量存储系统有关的一些话题。特别地，我们将讨论在设计文件系统时的一些可行的选择，以及文件系统在磁盘（或者用个人计算机领域的流行说法，硬盘）上的实现。要认识到，文件系统在本书“阐述整个计算机系统”的目标中所扮演的重要角色。为了深刻了解计算机系统所能展现出的能力，必须掌握在计算机内部信息是如何存储和操作的。因此，深入地讲解文件系统是如何工作的对于阐述计算机系统非常重要。

我们都已经对现实生活中存放了大量纸质材料于其中的文件柜和文件夹非常熟悉了。文件夹上的标签可以用来确定里面存放的内容，从而方便我们日后检索。通常，我们可能会在文件柜里放一个目录文件夹用于说明所有文件在柜子中的组织形式，如图 11-1 所示。

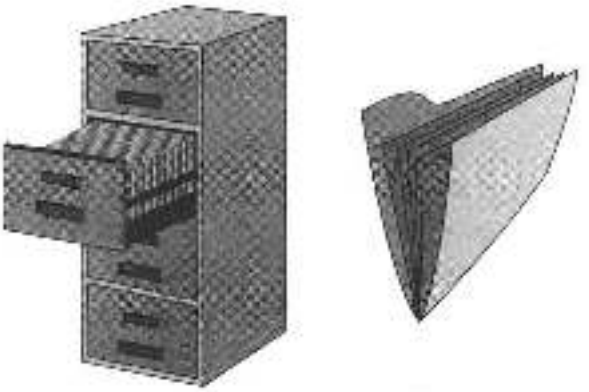


图 11-1　文件柜和文件夹

文件系统与我们现实生活中的文件柜很相似。每个文件（类似于一个纸质的文件夹）里有一些信息，并包含了一些与这些信息相关联的属性。进程是对处理器的一种软件抽象；数据结构则可看作内存的软件抽象。类似地，文件可以是输入/输出设备的软件抽象，因为设备可以作为信源或信宿进行服务。这种抽象使得用户的程序可以用一种与设备无关的方式与输入/输出进行交互。

首先，我们将讨论与文件相关联的属性，以及其中所蕴含的一些设计选项。然后我们将探讨在大容量存储设备上实现文件系统时的一些设计选择。

11.1　属性

与文件相关联的属性称作元数据。元数据代表空间开销，因此为了实用性考虑，我们需要对此进行细致的分析。

让我们先简单地了解一些可能要与文件相关的属性。

- **文件名**：该属性给文件内容一个逻辑标识。例如，如果存储音乐文件，我们可能希望给每张唱片一个唯一的名称。为了能够方便地查找，我们可能会保留一个单独的目录文件，里面包含了所有音乐唱片的名称。很容易看出这种用于早期的存储系统（例如，Univac Exec 8 计算机（20 世纪 70 年代））的单层命名方案太过局限。之后的一些系统（例如，DEC TOPS-10（20 世纪 80 年代初期））使用两层命名方案：顶层目录可以访问一个单独的用户或工程（例如，Billy Joel 的唱片）；第二层则指定了该用户或工程下的一个特定文件（例如，某一首特定的歌曲）。

然而，随着系统变大，很明显需要一种更具有层次的结构来对文件进行命名（比如，每个用户可能希望有他自己的音乐收藏集，其中包含不同艺术家的作品）。换句话说，我们可能需要一个如图 11-2 所示的多层目录。

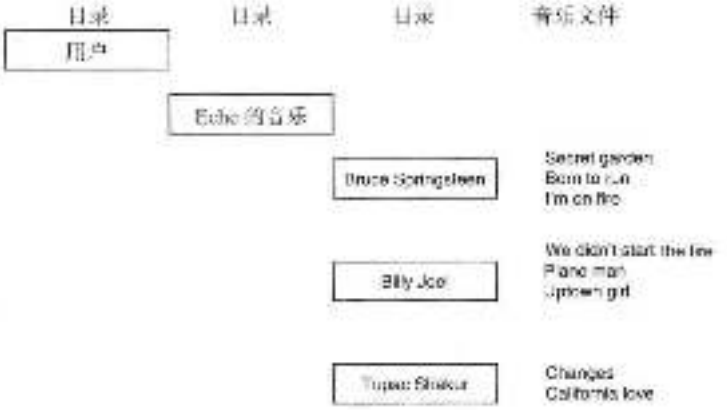


图 11-2 多层目录。分层结构是组织信息的自然方法

大多数现代操作系统，例如 Windows XP、UNIX 以及 MacOS，都实现了多层次的命名方案。文件名的每个部分仅对于其之前部分的名称是唯一的。这就提供了一种树形结构来组织文件系统中的文件（如图 11-3 所示）。树中的每个节点是一个对其父节点而言唯一的名字。目录也是文件。在图 11-3 中的树结构中，中间节点就是目录文件，而叶子节点就是数据文件。目录文件的内容，就是以该目录文件为根的下一层子树中的文件信息（例如，users 目录的内容是 {students, staff, faculty}；目录 faculty 的内容则是教师成员 rama 等）。

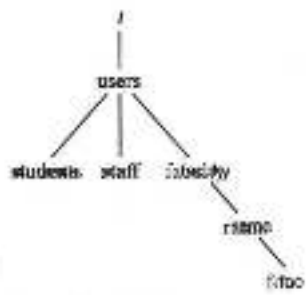


图 11-3 文件名 /users/faculty/rama/foo 的树形结构

有些操作系统在文件名后会强制性加入扩展名（以后缀的形式）。例如，在 DEC TOPS-10 操作系统中，文本文件会自动得到 .TXT 的后缀并附在用户给出的名称后面。在 UNIX 和 Windows 操作系统中，这样的文件扩展名是可选的。系统通过后缀来猜测文件的内容并启动合适的应用程序来处理这个文件（例如，C 编译器、文档编辑器、相片编辑软件等）。

有些操作系统允许给文件一个别名。别名可能在文件实际内容所处的层。或者，别名也可能只是简单在名称的层面不包括其实际内容。例如，UNIX 中的 ln 命令（表示链接）就可以给一个已存在的文件创建一个别名。命令

```
ln foo bar
```

的执行结果就是给一个已存在的名为 foo 的文件创建一个别名 bar。这样的别名，也称作硬链接，给予新名称 bar 与其原始名称 foo 同等的状态。即使我们删除了文件 foo，文件的内容依然可以通过名称 bar 访问到。

469 / 471

索引节点	访问权限	硬链接	大小	创建时间	名称
3193357	-rw-------	2 rama	80	Jan 23 18:30	bar
3193357	-rw-------	2 rama	80	Jan 23 18:30	foo

我们将在 11.3.1 节中解释索引节点（i-node）是什么。现在，我们只需了解它是表示文件的一个数据结构就足够了。注意，foo 和 bar 的内部表示完全一样，这是因为它们具有相同的索引节点。它们两者拥有相同的状态，无论它们之间的创建顺序是怎样的。这也是两者有相同大小、相同时间戳的原因，尽管 bar 比 foo 晚创建。

将前面的情形与接下来的 UNIX 命令相比较：

```
ln -s foo bar
```

这个命令同样会为 foo 创建一个别名 bar。

索引节点	访问权限	硬链接	大小	创建时间	名称
3193495	lrwxrwxrwx	1 rama	3	Jan 23 18:52	bar → foo
3193357	-rw-------	1 rama	80	Jan 23 18:30	foo

然而，这里的区别是：bar 与 foo 在名称上是等价的，但是却并不直接指向文件内容。注意两个名称的索引节点是不同的。因此，bar 的创建时间就是当 ln 命令被执行从而创建别名的时间。同样，它们的文件大小也不一样。foo 的大小是文件真实内容的大小，而 bar 只是 foo 这个字符串的大小（3 字节）。这样的别名也称作软链接。用这两个文件名都能够以相同的权限来操作文件内容。然而，在删除文件时就有区别了：删除 foo 会导致文件内容被移除。名称 bar 依然存在，但是其别名 foo 和里面的内容已经不存在了。如果这时试图访问 bar 的内容就会导致错误。

你可能想知道为什么操作系统希望支持两种不同的别名机制，即硬链接和软链接。这是为了权衡两者的效率和可用性。软链接可以直接告诉你原始文件名，而硬链接则隐藏了这个重要的细节。

所以，软链接增加了可用性。另一方面，文件系统每次遇到一个软链接，它都不得不通过遍历其内部数据结构（即 UNIX 中的索引节点）来获得其别名。稍后我们将看到在 UNIX 文件系统中这是如何完成的。硬链接直接指向原始文件名称的内部表示，因此就无需额外的时间开销来获得其别名，这样就可以提升文件系统的性能。

然而，一个目录的硬链接会导致循环链表，使得删除操作变得困难。基于这种原因，操作系统（如 UNIX）不允许对目录创建硬链接。

在大多数操作系统（UNIX、Windows）中，向一个已存在的文件写入内容会导致内容的覆盖。然而，在一个支持版本控制的文件系统中，这种写入可能会创建这个文件的另一个版本。 472

- **访问权限**：这个属性指定了谁可以访问某个特定文件以及每个被允许用户拥有什么权限。一个文件的权限通常包含读、写、执行、更改拥有者、改变权限。有些权限存在于个人用户级别（例如，文件的创建者或用户），另一些权限只对系统管理员（UNIX 中的 root 以及 Windows 中的 administrator）开放。例如，UNIX 中一个文件的拥有者可能会执行“更改文件的允许模式”命令：

```
chmod u+w foo      /* u stands for user;
                    * w stands for write;
                    * essentially this command
                    * says add write access
                    * to the user;
                    */
```

该命令会给文件拥有者写 foo 文件的权限。另一方面，只有系统管理可以执行“改变拥有者”命令

```
chown rama foo
```

它把文件 foo 的拥有者变为用户 rama。

一个对于文件系统设计者而言颇有意思的问题就是应该如何控制访问权限的粒度。理想情况下，我们可能希望给系统中的每个用户提供对每个文件独立的访问权限。这样会给每个文件增加 $O(n)$ 元数据空间开销，其中 n 是系统的用户数。操作系统试图通过各种设计方案限制这种空间开销。例如，UNIX 将用户分成 3 种：用户（user）、组（group）和所有的（all）。用户是系统的一个授权用户；组是系统的一组授权用户；所有的表示系统中的所有授权用户。系统管理员维护不同组的名字及其成员。例如，CS2200 课程中的学生可能都属于一个名为 cs2200 的组。UNIX 支持对任意文件设定个人用户的所有权以及组的所有权。文件的所有者可以通过下列命令来改变组的所有权：

```
chgrp cs2200 foo
```

把文件 foo 的组所有者改为 cs2200。

UNIX 对上述 3 种权限的每一个都提供了读、写、执行权限。因此，用 3 位即可表示对每一种的访问权限（对读、写、执行分别用 1 位表示）。执行权限允许将文件当作可执行程序来运行。例如，编译器编译并链接后的输出就是一个二进制可执行文件。下面的例子表示一个 UNIX 文件所有可见的元数据：

473

```
rwxrw-r-- 1 rama fac 2364 Apr 18 19:13 foo
```

文件 foo 被用户 ruma 和组 fac 拥所有。第一个字段提供了 3 种用户访问权限。前 3 位（rwx）表示给用户（rama）提供了读、写、执行权限；接下来的 3 位（rw-）表示给组（fac）提供了读、写权限（没有执行权限）；最后的 3 位（r--）表示所有用户都有读权限（没有写和执行的权限）。在访问权限之后的数字“1”表明指向该文件的硬链接个数。文件的大小是 2 364 字节，而文件内容的修改时间是 4 月 18 日 19 点 13 分。

Windows 操作系统以及某些 UNIX 操作系统通过访问控制列表（ACL）来对每个文件实现更细粒度的权限控制。这种灵活性会带来每个文件元数据大小增加的代价。

表 11-1 总结了常用的文件系统属性以及它们的含义。表 11-2 列举了大多数 UNIX 文件系统支持的一些常用命令。所有命令都是对于当前工作目录而言的（一个例外是，当命令指定了 UNIX 绝对路径时，例如 /users/r/rama）。

表 11-1 文件系统属性

属 性	含 义	详细描述
名称	文件名	在文件创建/重命名时设定的属性
别名	同一个底层文件的其他名称	当创建别名时设置的属性；像 UNIX 这样的系统提供了显式命令创建别名；UNIX 在两个不同层次上支持别名（物理的/硬链接，符号的/软链接）
所有者	创建该文件的创建者	在文件创建时设置的属性；像 UNIX 这样的系统提供了可以让超级用户修改文件所有者的机制
创建时间	文件第一次创建的时间	文件被创建或从某个其他地方复制过来的时间
上次写时间	最近一次文件写入的时间	文件写入/复制时设置的属性；在大多数文件系统中该属性与创建时间属性一致。注意把一个文件移到另一个位置会保留文件的创建时间

（续）

属　性	含　义	详细描述
权限 • 读 • 写 • 执行	对文件当前的权限，决定了哪个用户可以对其进行什么样的操作	文件创建时会为其设置成默认值；通常，文件系统提供了可以所有者修改权限的命令；现代的文件系统（如 NTFS）提供了访问控制列表（ACL），对不同用户提供不同级别的访问权限
大小	文件系统上占据的所有空间	每次文件修改后会进行更新的属性

表 11-2　常见的 UNIX 文件系统命令

UNIX 命令	语义	详细描述
touch <name>	创建一个名为 <name> 的文件	创建一个 0 字节的文件 <name>。创建时间为当前时间
mkdir <sub-dir>	创建一个子目录 <sub-dir>	用户必须拥有当前工作目录的写权限（如果 <sub-dir> 是相对路径名）以便能够成功执行该命令
rm <name>	移除（或删除）名为 <name> 的文件	只有文件的所有者（或超级用户）可以删除该文件
rmdir <sub-dir>	移除（或删除）名为 <sub-dir> 的子目录	只有 <sub-dir> 的所有者（或超级用户）可以删除该目录
ln –s <orig> <new>	创建一个名为 <new> 的软链接并指向 <orig>	两者只是在名称上有等价性；因此文件 <orig> 被删除，与 <orig> 相关的存储空间被收回，因此 <new> 将指向一个不存在的文件
ln <orig> <new>	创建一个名为 <new> 的硬链接并指向 <orig>	即使文件 <orig> 被删除了，依然可以通过 <new> 访问到物理文件
chmod <rights> <name>	将文件 <name> 的访问权限修改为 <rights>	只有文件的所有者（或超级用户）可以修改访问权限
chown <user> <name>	将文件 <name> 的所有者修改为 <user>	只有超级用户可以修改文件的所有者
chgrp <group> <name>	将文件 <name> 的所有组修改为 <group>	只有文件的所有者（或超级用户）可以修改与该文件相关的组
cp <orig> <new>	为文件 <orig> 创建一份名为 <new> 的拷贝	如果 <new> 是文件名，那么会在同一个目录下创建一个拷贝；如果 <new> 是目录名，则会在 <new> 目录下创建一个与 <orig> 文件名相同的文件
mv <orig> <new>	将文件 <orig> 重命名为 <new>	当 <new> 是文件名时，则为重命名操作；若 <new> 是目录名，则将文件 <orig> 移到 <new> 目录下
cat/more/less <name>	查看文件内容	

11.2　在磁盘子系统上实现文件系统的设计选择

我们的讨论先从把文件系统当成输入/输出设备的一种软件抽象开始。在程序执行的生命周期之外，对文件系统同等重要的是保存信息。文件可以作为满足这种需求的一个方便的抽象。永久性读/写存储器就是保存这类信息所需要的正确解决方案。文件系统是操作系统另一个重要的软件子系统。通过使用磁盘作为永久性存储器，我们将讨论实现文件系统时的一些设计选项。

就像我们在第 10 章中看到的，磁盘在物理上由盘片、盘道以及扇区组成。一个给定的磁

盘会有特定的固定硬件参数。逻辑上，相应的多个盘片组成了一个柱面。磁盘与 I/O 之间的延迟通常由 4 部分组成：

- 寻找指定柱面时间。
- 将磁盘的读 / 写头旋转到特定扇区的旋转延迟。
- 磁盘控制器缓冲区的传输时间。
- 控制器缓冲区与系统内存之间的 DMA 传输。

我们知道一个文件可以是任意大小的，程序也是一样。例如，包含了一些简单 ASCII 文本的文件可能只有几 KB 大小。另一方面，你下载到计算机上的一部电影可能会占据几百 MB 空间。文件系统需要从用户的角度将文件作为存储抽象和硬盘的物理细节之间的桥梁。一个文件（取决于其大小）可能占据多个扇区，多个磁道，甚至多个磁盘。

因此，文件系统中的一个基本设计问题就是文件在磁盘上的物理表示。设计上既需要考虑最终用户的需求，也要考虑系统的性能。我们来看一看这些问题。从用户的角度，可能有两个需求：首先，用户可能希望按顺序查看文件的内容（例如，UNIX 中的 more、less 和 cat 命令）；其次，用户可能希望在一个文件中搜索特定的内容（例如 UNIX 的 tail 命令）。前者表明物理表示需要能够支持有效的线性访问；而后者是随机访问。从系统性能的角度，文件系统的设计应当能够支持文件的按需扩展，以及当创建新文件或者扩展已有文件时在磁盘上能够进行空间的有效分配。

因此文件系统设计时的性能系数[⊖]是：

- 快速的顺序访问。
- 快速的随机访问。
- 扩展文件的能力。
- 简单的存储分配。
- 磁盘上的空间利用率。

在接下来的几节里，我们将比较几种在硬盘上的文件分配方案。对每种方案，我们将讨论文件系统中需要的数据结构，以及每种方案在性能系数上的特点。对于余下的讨论，我们将定义一些常用术语。磁盘上的地址用一个三元组 {柱面 #，表面 #，扇区 #} 表示。文件系统认为磁盘由多个磁盘块（文件系统的一个设计参数）构成。每个磁盘块是磁盘上的一段物理的连续区域（也就是一组扇区、磁道或柱面，取决于具体的分配方案），是文件系统管理磁盘空间时的最小单元。为了简化讨论，我们用磁盘块地址作为对应一个特定磁盘块的磁盘地址（即四元组 {柱面 #，表面 #，扇区 #，磁盘块大小}），并用整数来表示。

11.2.1 连续分配

磁盘分配方案与第 8 章中介绍过的基于固定的 / 可变大小的内存分配方案有相似之处。在创建文件时，文件系统给这个文件预分配固定大小的空间。分配的空间大小取决于文件的类型（例如，文本文件 / 媒体文件）。而且，分配的空间大小是文件能够达到的最大大小。图 11-4 展示了这种方案所需的数据结构。目录数据结构中的每项包含了一个文件名到磁盘块地址的映射，以及分配给该文件的磁盘块个数。

文件系统需要维护一个包含可用磁盘块的空闲链表（见图 11-5）。空闲链表使文件系统可以记录当前还未分配的磁盘块。在第 8 章中，我们讨论了在出现页错时内存管理器通过物理

476

⊖ 性能系数是用于评价系统性能的标准。

映的空闲链表来处理的方法。类似地，文件系统在需要创建新文件时通过磁盘块的空闲链表进行磁盘块分配。我们将看到，空闲链表的细节取决于文件系统所使用的具体分配策略。

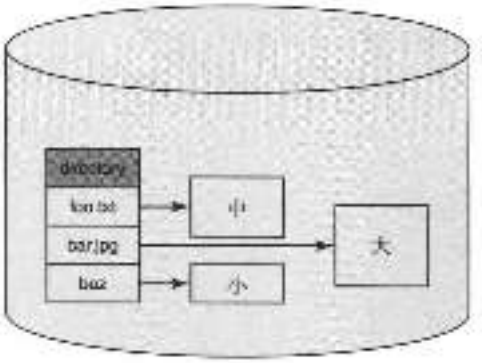


图 11-4 一个连续分配的例子：文件名到磁盘块地址的映射。连续分配的大小与文件大小相匹配

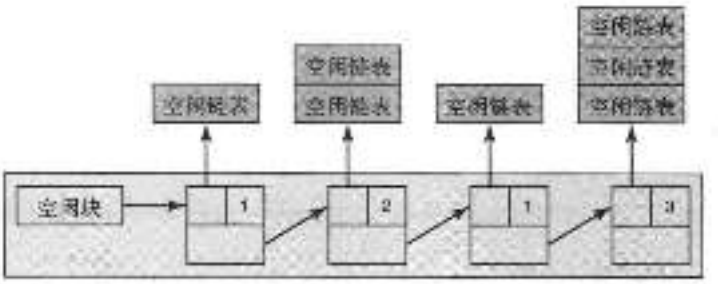


图 11-5 用于连续分配的空闲链表，每个节点包含了（起始磁盘块地址指针，块数）的信息

为了进行连续分配，空闲链表中的每个节点包含开始磁盘块地址和可用的块数。为一个新文件分配磁盘块时可以采用最先适配或最佳适配策略。在文件删除时，释放的磁盘块将回到空闲链表中。文件系统将相邻的节点进行合并形成更大的连续磁盘块区间。当然，考虑到其中的开销，文件系统不能经常进行这种压缩操作。当用户显式地进行请求时文件系统才进行这样的操作。这种描述与第 8 章中可变大小的内存划分有些类似，且这种磁盘分配策略也会有类似的外部碎片问题。而且，由于文件系统在创建文件时申请的是固定大小的磁盘块（为了满足文件最大的大小），所以这种方案也会有内部碎片（与第 8 章中的固定大小的内存划分方案类似）的问题。

文件系统既可以把这些数据结构放在内存中，也可以放在硬盘上。由于文件是永久性存储，所以这些数据结构也不得不放到永久性存储器中（也就是说，有些磁盘块用来实现这些数据结构）。因此，这些数据结构驻留在磁盘。然而，为了进行更快的分配，以及加速文件的访问，文件系统需要把这些数据结构缓存到内存中。

我们需要对这种方案的性能参数进行定量分析。分配的代价可能会很高，这取决于具体使用的算法（最先适配或最佳适配）。由于文件占据了一块固定的区间（磁盘上的一段连续区域），所以文件的线性访问和随机访问都非常迅速。当磁盘头位于特定文件的起始磁盘块地址，这种方案的特点使得我们只需要很少的一点额外时间就可以跳转到文件的不同部分。这

477 / 478

种分配方案有两个缺点：

1）当文件增长到创建时分配的大小以上后就无法进行扩展了。一种可能的解决方法是在空闲链表中寻找一个更大的区间，然后将文件复制到这个新分配的区间里。这样做的代价非常高；而且需要存在一个可用的更大区间才可以。

2）如我们之前所说，这种方案会因为内部碎片和外部碎片造成存在的空间浪费。

例11-1

注

磁盘上的柱面数 = 10 000

扇区数 = 10

每个盘片的面数 = 2

每个磁道的扇区数 = 128

每个扇区的字节数 = 256

磁盘分配策略 = 基于簇的位置

a. 一个3MB的文件需要分配多少个连续的簇？

b. 这种分配方式会产生多少内部碎片？

答：

a. 一个柱面上的磁盘簇数 = 盘片数 × 每个盘片的面数 = 10 × 2 = 20。

磁道的大小 = 磁道的扇区数 × 扇区大小 = $128 \times 256 = 2^{15}$ 字节。

一个柱面的容量 = 柱面的磁道数 × 磁道大小 = $20 \times 2^{15} = 10 \times 2^{16}$ 字节。

479 3MB文件的连续簇数 = CEIL($(3 \times 2^{20})/(10 \times 2^{16})$) = 5

b. 内部碎片大小 = 5个柱面的容量 − 3MB = 3 276 800 − 3 145 728 = 131 072字节

11.2.2 带有溢出区域的连续分配

这种策略与前一种几乎一样，不同之处是文件系统设置了一个溢出区域供无法放入初始分配的同定区间的大文件使用。溢出区域同样也是由物理上连续的区域组成，用以容纳大文件溢出的部分。因此文件系统需要一个额外的数据结构来管理溢出区域。这种方案与前一种几乎一样，除了性能参数上有一些不同处。这种方案的好处是，文件可以增长到溢出区域所允许的大小，却无需任何其他的高代价操作。其不利之处是，对于大文件的随机访问可能会相继受到一些影响（需要额外的寻道时间）。

尽管有一些限制，但连续分配依然由于其文件访问时间上的杰出性能优势，在IBM VM/CMS等系统上广泛使用。

11.2.3 链接分配

在这种方案中，文件系统在单个磁盘块的层面上处理分配。文件系统需要维护一个包含所有可用磁盘块的空闲链表。一个文件可以占据任意大小的磁盘块来存储到磁盘上。当文件大小增长时，文件系统就从空闲链表分配新的磁盘块。空闲链表实际上是一个磁盘块的链表，每一块都指向磁盘的下一个空闲块。文件系统将这个链表的头指针缓存到内存中以便快速地分配磁盘块来满足新的分配请求。在删除文件时，文件系统将磁盘块放回空闲链表。通常，通过磁盘块来维护这样的表在进行遍历时非常耗时。另一种方法是用一个位矢量来实现这个链表，每个磁盘块1位。如果对应的位是0，则表明该块处于空闲状态；为1，则表示处于忙碌状态。

注意随着时间的流逝空闲链表也会产生变化，例如应用程序产生和删除文件或者文件增大或减小时。因此，无法保证文件会占据连续的磁盘块。所以，如图 11-6 所示，文件在物理上存储于一个磁盘块的链表中。与前面的分配方案一样，一部分磁盘块用于保存文件系统的永久性数据结构（空闲链表和目录）。

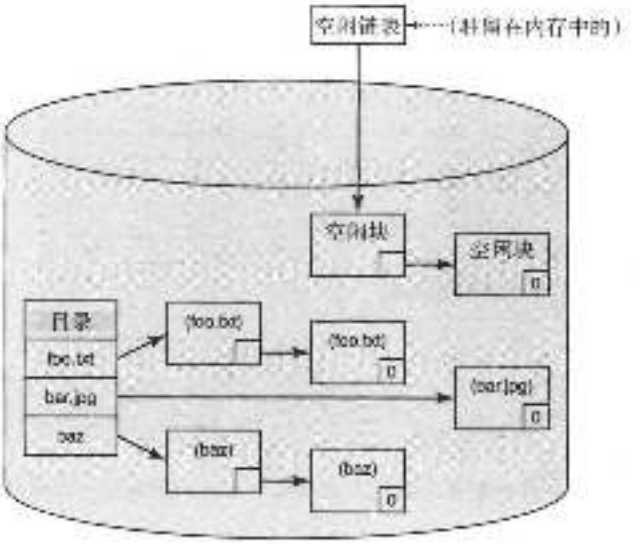


图 11-6 链接分配。磁盘块构成了一个链表的数据结构

这种方案的好处是分配速度很快，因为每次只分配一个磁盘块。而且文件的增长也变得更容易。由于这种按需分配的特点，所以不会存在外部碎片。因此也就无需磁盘压缩。从不利之处看，由于一个文件的磁盘块不一定是连续的，所以与连续分配相比文件访问的性能可能不太好，尤其是随机访问，还需要从磁盘块上获取下一块的指针。即使是顺序访问，也会因为从链表中定位到不同磁盘块的时间而导致效率不高。这种方案的易出错性也是另一个劣势：在链表维护时产生的任何程序错误都会导致文件系统的彻底损坏。

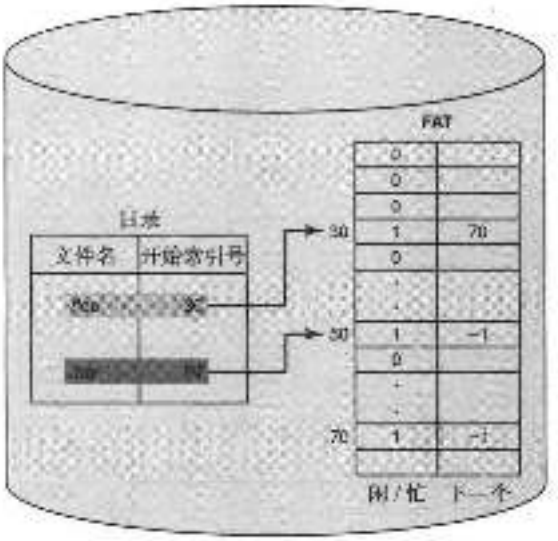
11.2.4 文件分配表

这是链接分配的一个变种。在磁盘上的文件分配表（FAT）包含了当前保存磁盘中文件的链表（见图 11-7）。这种方案从逻辑上将磁盘分成多个区。每个区有一个 FAT，其中的每项对应一个特定的磁盘块，而用一个字段表明这一块的可用性（0 表示空闲；1 表示忙）；下一个字段给出了表示文件的链表中的下一个磁盘块。特殊值（-1）表明这一项是该文件的最后一个磁盘块。对整个区，一个单独的目录包含了文件名到 FAT 索引的映射，如图 11-7 所示。与链接分配类似，文件系统根据需求给文件分配磁盘块。

比如，/foo 占据了两个磁盘块：30 和 70。项 30 的下一个字段的值是 70，即下一个磁盘块的地址。项 70 的下一个字段的值是 -1，表示这是 /foo 的最后一个磁盘块。类似地，/bar 占据了一个磁盘块（50）。如果 /foo 和 /bar 变大，我们会分配一个空闲的磁盘块并相应地修改 FAT。

让我们来分析这种方案的优劣。由于 FAT 以一种表格数据结构的方式组织磁盘的链表结构，所以与链接分配相比它发生错误的概率会更小。同时，通过将 FAT 缓存在内存中，还可

以获得更快的分配速度。在进行顺序文件访问时 FAT 与链接分配方法相似。对于随机访问，由于包含了文件下一块指针所以它可以获得更好的效率。


FAT
目录
文件名
开始索引号
30
50
70
1
70
1
-1
1
-1
0
0
0

11.2.5 索引分配

这种方案为每个文件分配一个索引磁盘块。文件的索引块是一个包含了文件所有数据块地址的固定大小的数据结构。这种方案将文件的数据块指针聚集到一起放到一个文件中，如图 11-8 所示。这张表称为索引节点（index-node，i-node），占一个磁盘块。目录（也存储在磁盘上）包含了每个文件的文件名到索引节点的映射。与链接分配类似，这种方案用磁盘块的一个位矢量来维护空闲链表（0 表示空闲，1 表示忙）。

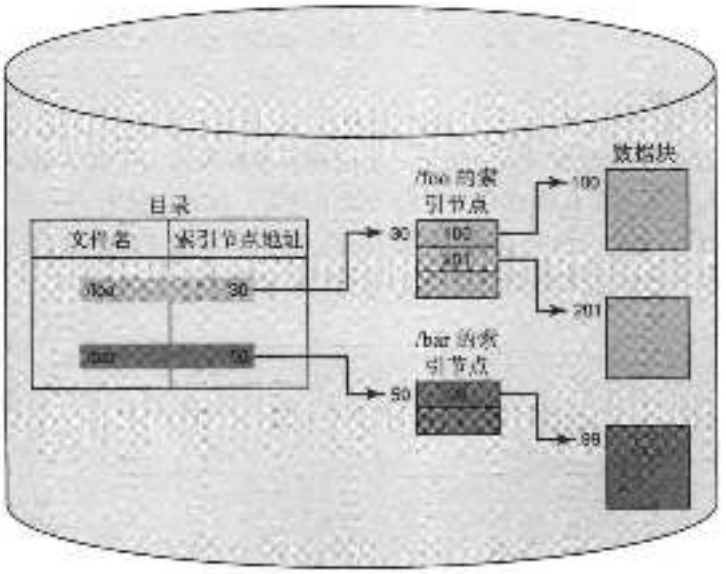


图 11-8 索引分配。与 FAT 所有文件用一个表来表示不同，索引分配为每个文件分配了一个单独的表（索引块）

与 FAT 相比，这种方案对于随机访问表现得更好，因为索引节点把所有磁盘块指针聚集到了一个简洁的数据结构中。不足之处是文件的最大大小有限制，因为每个文件的索引节点拥有固定的数据结构，其直接指向了各个数据块。索引节点中数据块指针的个数决定了文件可能的最大大小。

我们将在下面的各节中探索其他能够消除这种最大文件限制的方案。

例 11-3 考虑磁盘上的索引分配方案：

- 磁盘有 10 个盘片（每个盘片有 2 个面）。
- 每个面有 1000 个磁道。
- 每个磁道有 400 个扇区。
- 每个扇区有 512 字节。
- 每个索引节点是固定大小的数据结构，占据一个扇区。
- 一个数据块（即分配的单位）是连续的 2 个柱面。
- 指向磁盘数据块的指针用一个 8 字节的数据结构表示。

问：

a. 文件系统创建一个文件最小需要占用多少空间？

b. 根据上述分配方案，一个文件最大可以达到多少？

483
:
484

答：

一个磁道的大小＝每个磁道的扇区数 × 扇区大小＝400×512 字节＝200KB（K=1024）

一个柱面的磁道数＝盘片数 × 每个盘片的面数＝10×2＝20

一个柱面的大小＝柱面的磁道数 × 磁道大小＝20×200KB＝4000KB

分配单元（数据块）＝2 个柱面＝2×4000KB＝8000KB

索引节点的大小＝扇区大小＝512 字节

a. 一个文件的最小字节＝索引节点大小＋数据块大小＝512＋(8000×1024)=8 192 512 字节

索引节点中的数据块指针个数＝索引节点大小／数据块指针大小＝512／8＝64

b. 文件的最大大小＝索引节点的数据块指针个数 × 数据块大小＝64×8000KB＝5 24 288 000 字节

11.2.6 多级索引分配

这种方案通过把文件的索引节点变为一个间接表来修复索引分配中的限制。例如，通过一级间接索引，每个索引节点项指向一个指向数据块第一级表，如图 11-9 所示。在图中，foo 的索引节点包含了一个第一级间接索引块的指针。第一级间接表的entry与等于索引节点保存的指针数。这些第一级间接表存储了指向文件数据块的指针。

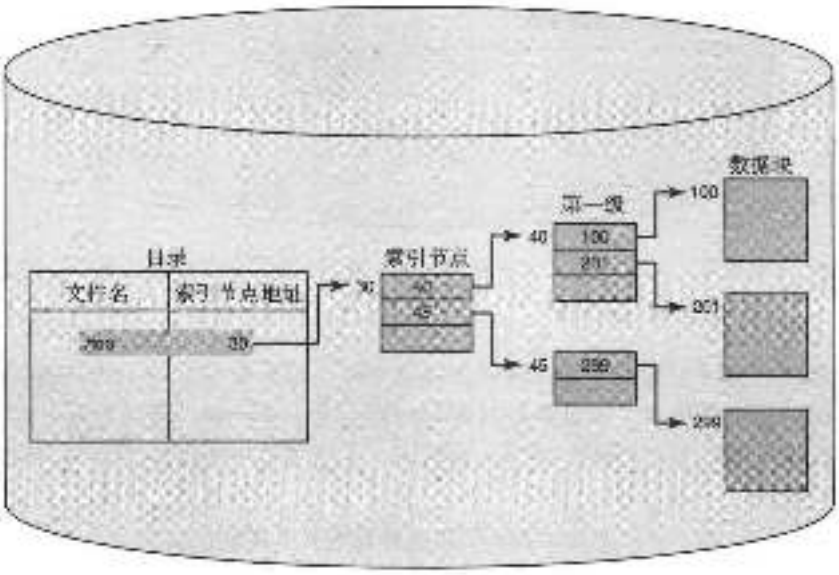


图 11-9 多级索引分配（一级间接索引）。与单级的索引分配相比，这种方案在应对文件增长时更灵活

这种方案还可以将索引节点继续扩展成二级（甚至更高级）的间接表。这取决于文件系统需要支持的文件大小。这么做的不利之处是，即使一个只需几个数据块的小文件也需要额外级的间接信息。

11.2.7 混合索引分配

这种方案结合了前两种的特点来形成一种新的混合方案。如图 11-10 所示，每个文件有一个索引节点。对于小文件，所有的数据块用直接指针存储表示。如果文件大小超过了直接数据块能表示的容量，那么将使用一个一级甚至更多级的间接索引来表示其他的数据块。图 11-10 中的 /foo 使用了直接、一级间接和二级间接指针。/foo 的索引节点是一个复杂的数据结

构：它有两个直接数据块的指针（100 和 201）；一个一级间接索引块的指针（40）；一个二级间接索引块的指针（45）；指向一级间接索引块的指针（60 和 70）多个指针；一个二级间接索引块的指针（当前还未分配）。这种方案在保留了前面两种方案优点的同时克服了两者的缺点。当创建文件时，只将索引节点分配给它，也就是说，初始时其一级、二级、三级间接指针都是空的。如果文件还没有超过直接块中的容量大小，就没必要创建额外的索引块。然而，当文件逐渐增长并超过了直接数据块的容量时，我们才会按需通过一级/二级/三级索引块的方式分配更多的空间。 485 / 486

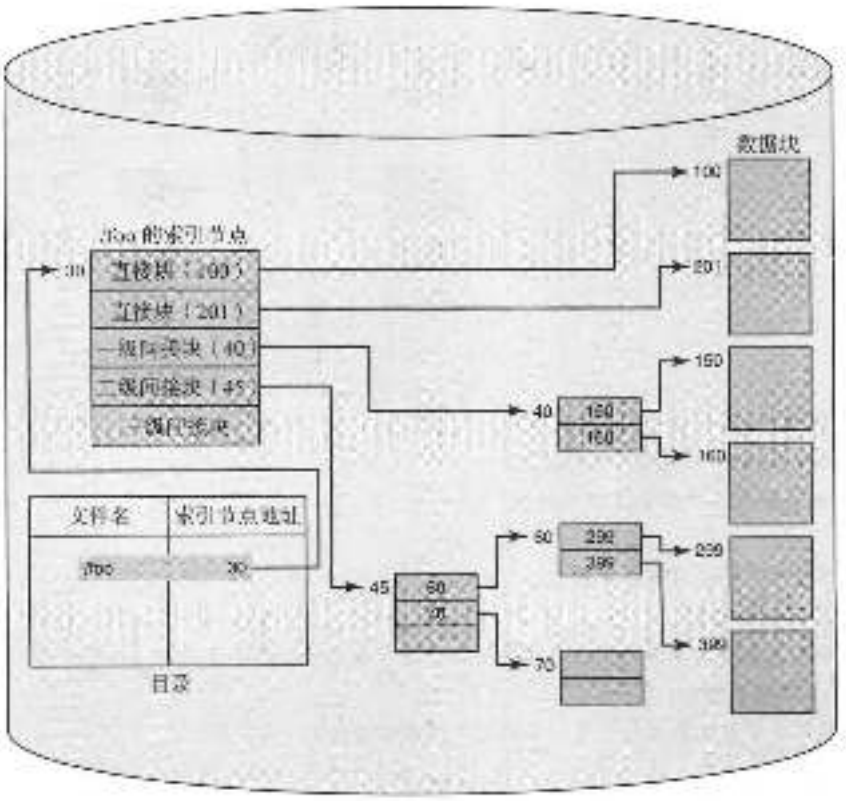


图 11-10 混合索引分配。这是一种单级与多级索引分配的折中方案，在保证快速访问小文件的同时也允许文件可以不断扩展

例 11-4

设

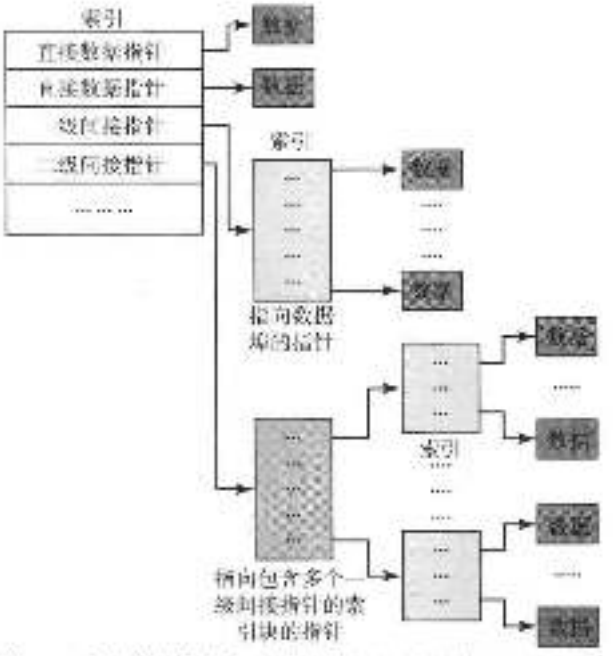
- 索引块大小 = 512 字节
- 数据块大小 = 2048 字节
- 指针大小 = 8 字节（指向索引块或数据块） 487

索引节点的组成为：

- 2 个直接数据块指针
- 1 个一级间接指针
- 1 个二级间接指针

一个索引块用于存储一个索引节点，也可以用来存储指向其他索引块或数据块指针的索引块。在下

面的图中画出了数据结构。注意，索引块和数据块都是按照分配的。



a. 在这个文件系统中，一个文件的最大大小可以是多少（字节）？

b. 一个 266KB 的文件需要多少数据块？

c. 一个 266KB 的文件需要多少索引块？

答：

488

a. 一级间接索引或二级间接索引指针在一个索引块中的个数 = 512 / 8 = 64。

直接数据块的个数 = 2。

一个索引节点包含了 1 个指向某个包含了多个指向数据块指针的索引块（这类索引块也称一级间接索引块）的一级间接指针。

通过一级间接索引的数据块个数 = 索引块中的数据块指针个数 = 64。

一个索引节点包含了 1 个二级间接指针，其指向一个包含了 64 个一级间接指针的索引块（这类索引块也称二级间接索引块）。每个这样的指针都指向一个一级间接索引块，其中的每个又指向 64 个数据块。如上图所示。

因此，通过二级间接索引的数据块个数 = 一个索引块中的一级间接指针个数 × 索引节点包含的数据块指针个数 = 64 × 64。

块中的最大文件大小 = 直接数据块个数 + 通过一级间接索引的数据块个数 + 通过二级间接索引的数据块个数 = 2 + 64 + 64 × 64 = 4162 数据块。

最大文件大小 = 块中的最大文件大小 × 数据块大小 = 4162 × 2048 字节 = 8 523 776 字节。

b. 需要的数据块个数 = 文件大小 / 数据块大小 = 226×2^{10} / 2048 = 133。

c. 为了索引 133 个数据块，我们需要：

1 个索引节点（包含 2 个直接数据块）

1 个一级间接索引块（索引 64 个数据块）

1 个二级间接索引块

二级简介索引块上的 2 个一级间接索引块（从而获得了剩下的 64+3 个数据块）

489

因此，我们共需要 **5 个索引块**。

11.2.8 不同分配策略的比较

表 11-3 总结了多种分配策略之间的优劣。

表 11-3 各种分配策略比较

分配策略	文件表示	空闲链表维护	线性访问	随机访问	文件增长	分配开销	空间效率
连续	连续块	复杂	非常好	非常好	困难	中到高	内部/外部碎片
带附加的连续	(小文件)连续块	复杂	(对小文件)非常好	(对小文件)非常好	简可	中到高	内部/外部碎片
链表	非连续块	位矢量	好，但是取决于寻道时间	不好	非常好	小到中	优秀
FAT	非连续块	FAT	好，但是取决于寻道时间	好，但是取决于寻道时间	非常好	小	优秀
索引	非连续块	位矢量	好，但是取决于寻道时间	好，但是取决于寻道时间	受限	小	优秀
多级索引	非连续块	位矢量	好，但是取决于寻道时间	好，但是取决于寻道时间	好	小	优秀
混合索引	非连续块	位矢量	好，但是取决于寻道时间	好，但是取决于寻道时间	好	小	优秀

490

为了让文件系统可以应对电力故障，文件系统的永久性数据结构（例如，索引节点、空闲链表、目录、FAT 等）需要驻留在磁盘上。保存这些数据结构所需的磁盘块数量代表了某种分配策略的空间开销。访问这些数据结构还有时间开销，因此文件系统还需要将这些关键的数据结构缓存到主存上以避免时间代价。

11.3 信息汇总

作为一个具体的例子，UNIX 操作系统通过层次化命名实现了混合分配方式。通过层次化命名，目录结构不再是集中的。每个索引节点表示层次名称中多个部分中的一部分。除了子叶节点上的数据文件外，所有的中间节点都是目录节点。索引节点中的类型字段表明该节点是目录还是数据文件。索引节点的数据结构包含了其他一些我们之前讨论过的文件属性，例如访问权限、时间戳、大小和所有者等。另外，为了支持别名，索引节点还有一个引用计数字段。

图 11-11 显示了文件 /users/faculty/rama/foo 的完整索引节点结构。为了简单起见，此处省略了数据块。

图 11-12 显示了用 bar 作为 foo 的硬链接的索引节点结构。两个文件共享相同的索引节点。因此，索引节点上的引用计数字段的值为 2。图 11-13 是第三个文件 baz 为 /users/faculty/rama/foo 的软链接时的索引节点结构。baz 的索引节点表明它是一个软链接并且包含了文件名 /users/faculty/rama/foo。文件系统用软链接提供的文件名开始遍历索引节点结构（例如，本例

中访问 baz 时，将从 / 的索引节点开始）。

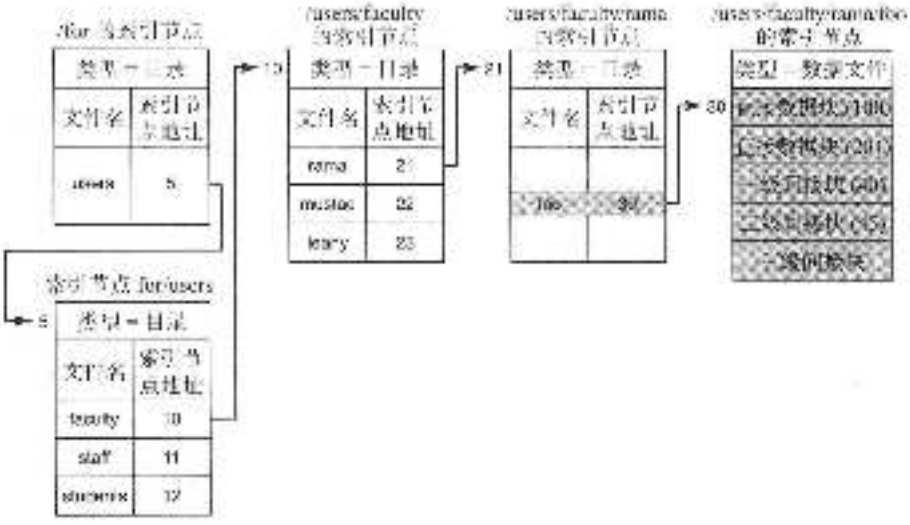


图 11-11 UNIX 中层次化命名的一个简化的索引节点结构（/users/faculty/rama/foo 文件通过一系列文件创建）。

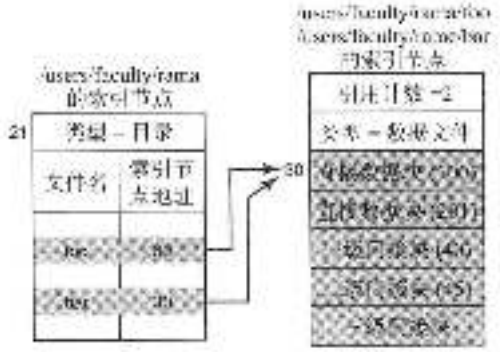


图 11-12 两个文件 foo 和 bar 共享一个索引节点（两者之间是硬链接，通过“ln foo bar”命令创建）

例 11-5 当前目录是 /tmp，/tmp 的索引节点是 20。

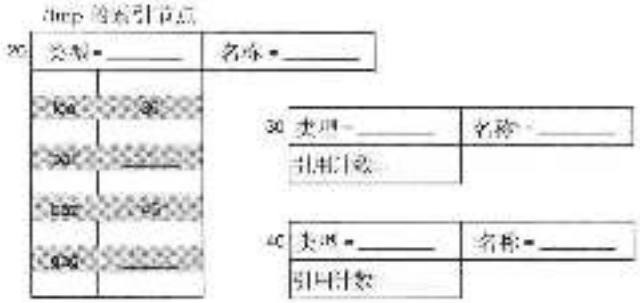
在当前目录下按顺序的 UNIX 命令：

```
touch foo              /* 在当前目录下创建一个零字节的文件 */
ln foo bar             /* 创建一个硬链接 */
ln -s /tmp/foo baz     /* 创建一个软链接 */
ln bar gag             /* 创建一个硬链接 */
```

注意：

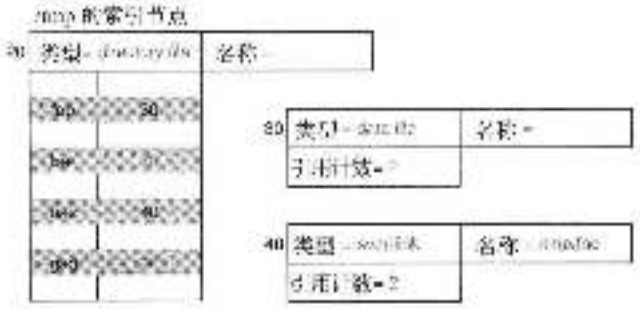
- 索引节点的类型可以是目录文件、数据文件、软链接之一。
- 如果就是要数链接，那么你不得不提供与数链接相关联的名称；否则，索引节点中的名称字表为空。
- 引用计数是一个字表的正整数。

将内容填入下图的空格来完成索引节点的信息。



491 ~ 493

答：



例 11.2

给定下列命令，写出索引节点的内容。为了简化问题，你可以自定索引节点的磁盘块地址。请给出索引节点的引用计数。

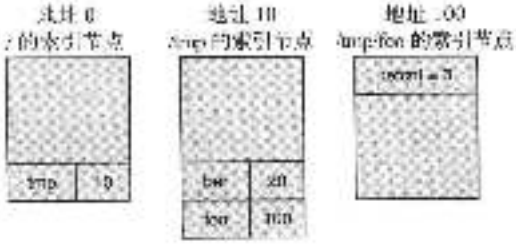
```
touch /tmp/foo
mkdir /tmp/bar
mkdir /tmp/bar/gag
ln /tmp/foo /tmp/bar/foo2
ln -s /tmp/foo /tmp/bar/foo
ln /tmp/foo /tmp/bar/gag/foo
ln -s /tmp/bar /tmp/bar/gag/bar
ln -s /tmp /tmp/bar/gag/tmp
```

注意：

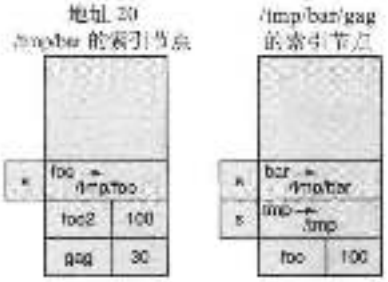
- "mkdir" 创建一个目录。
- "touch" 创建一个零字节的文件。
- "ln" 是链接命令（-s 表明是符号链接也称为软链接）。

假设首面的文件和目录是文件系统中仅有的文件和目录。

答：



494



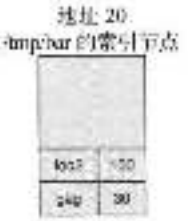
此后执行下述命令：

rm /tmp/bar/foo

请给出索引节点与新内容（仅指受影响的索引节点的内容写出即可）。

答：

只在 /tmp/bar 的索引节点发生了如下所述的变化。需要将符号链接项从索引节点中移除。



关于图 11-13，我们首先看一看，如果文件 bar 被删除了会对索引节点有什么影响。将 /user/faculty/rama 的索引节点项 bar 移除，/users/faculty/rama/foo 索引节点（块 30）的引用计数将字段减 1（即新的引用计数将等于 1）。

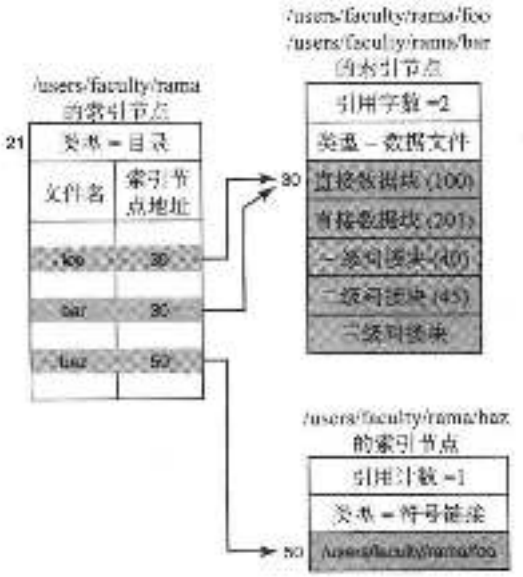


图 11-13 baz 处文件 /users/faculty/rama/foo 的软链接（通过命令 "ln -s /users/faculty/rama/foo baz" 创建）

然后，我们看看当 bar 和 foo 都被删除后索引节点结构又会有什么改变。图 11-14 描述了这种情形。在删除 bar 和 foo 后，由于块 30 的引用计数值变为 0，所以文件系统将把块 30 放回空闲链表中。注意 baz 依然存在于 /users/faculty/rama 下。这是因为文件系统只检查在创建符号软链接时建立的别名的有效性。但是，注意此时该名称的索引节点（此处的“foo”）信息已经不存在了。因此，文件系统没有办法在删除文件（此处的“foo”）时检查指向文件的符号链接是否存在。然而，任何尝试获取 baz 内容的操作都会发生错误，因为就文件系统而言，文件 /users/faculty/rama/foo 已经不存在了。这说明我们在使用符号链接时需要小心谨慎。

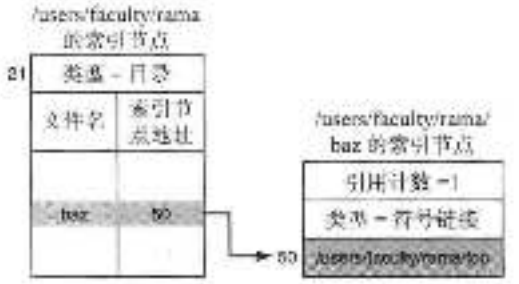


图 11-14 当 foo 和 bar 都被删除后索引节点结构的状态（此处显示了“rm foo bar”导致的影响）

例 11-2

考虑下列命令：

```
touch foo;              /* 创建一个零字节文件 */
ln foo bar;             /* 创建 foo 的一个硬链接 bar */
ln -s foo baz;          /* 创建 foo 的一个软链接 baz */
rm foo;                 /* 删除文件 foo */
```

495
:
496

当执行下列命令时会发生什么？

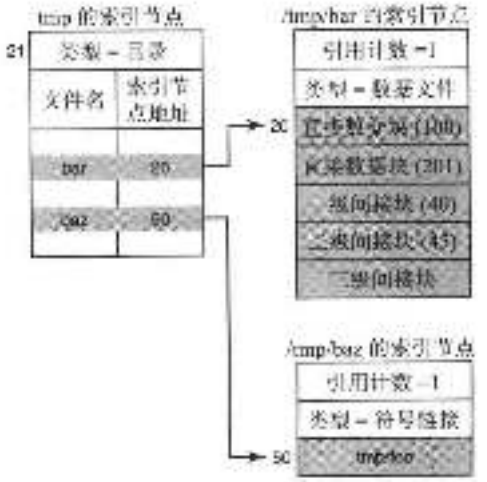
```
cat baz;                /* 查看并输出文件 baz 的内容 */
```

答：

假设所有之前操作发生的位置，即当前目录，为 tmp。在这个例子中，foo、bar、baz 都在 tmp 的索引节点中创建。

假设 foo 的索引节点为 20。当我们创建硬链接 bar 时，索引节点 20 的引用计数变为 2。当 foo 被删除后，索引节点的引用计数降到 1，但是索引节点并没有被删除，因为 bar 还有一个指向它的硬链接。但是，foo 从当前目录的索引节点中删除了。

至此，当我们尝试通过 cat 命令查看文件 baz 的内容时，文件系统会发生错误，因为文件 foo 已经不再存在于当前目录中了。下图说明了具体的情况。



11.3.1 索引节点

在UNIX中，每个文件有一个与之关联的唯一编号，称作索引节点编号。你也许会把
[492] 这个编号当作保存与文件相关的所有信息（所有者、大小、名称等）的表的索引。当UNIX系统发展后，与每个文件有关的信息的复杂度也随之扩展了（例如，文件名可以任意长等）。因此，在现代的UNIX系统中，每个文件用存储了一整块磁盘块的唯一的索引节点数据结构来表示。索引节点集构成了一张逻辑表，索引节点编号是这个磁盘块的地址，它可以用来作为这张逻辑表的索引，并包含了那个特定文件的信息。为了方便文件系统的实现，所有索引节点在文件系统中占据物理介质上空间相邻的位置。这隐式地限制了文件系统能够保存的最大文件数目。习惯上，通过存储分配算法维护（每个索引节点一位）的位矢量用来指定某个特定的索引节点是否被使用了。这就允许在处理文件创建请求时能够有效地分配索引节点。类似地，为了有效地进行存储分配，存储管理器在磁盘上用位矢量实现了数据块的空闲链表（每一位表示一个特定的数据块是否被使用或者不在介质上）。

11.4 文件系统的组件

尽管文件系统可以在用户层实现，但通常它并不属于操作系统。图11-15展示了磁盘上文件系统的层次结构。我们把管理文件系统的操作系统部分称作文件系统（FS）管理器。为了便于说明，我们把文件管理器分成以下几层：

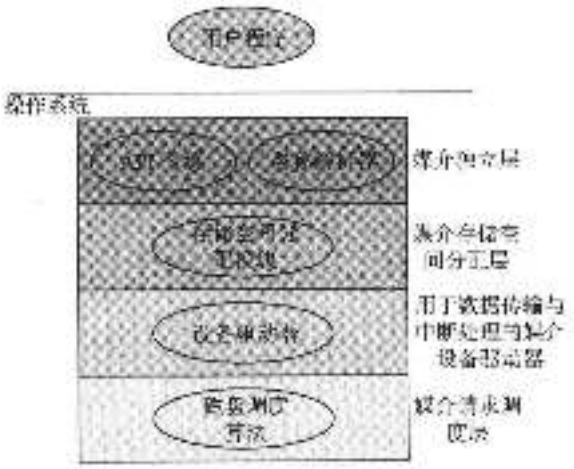


图11-15 磁盘文件系统管理器的层次结构图

- **媒介独立层**：这一层由用户接口组成——提供给用户的应用程序接口（API）。API模块向用户程序提供文件系统命令，其能够对文件进行打开、关闭、读、写等操作。这一层也包括了名称解析器模块，它将用户提供的名称转换为对文件系统而言有意义的内部表示。例如，它可能把用户文件名（例如，E:\myphotos）映射到文件指定设备上（例如，磁盘、CD、闪存盘等）。
- **媒介存储空间分配层**：这一层将（文件创建时的）空间分配、（文件删除时的）空间回收、空闲链表维护以及其他与管理物理设备空间有关的功能进行了具体化的实现。例如，如果在磁盘上存在文件系统，那么数据结构和算法就会使用11.2节讨论的分配方案。

- **设备驱动器**：这一部分处理与设备通信的命令，以及在设备和操作系统缓冲区之间发生的数据传输。设备驱动器的细节（第 10 章中包含了设备驱动器的一些信息）取决于文件系统位于的大容量存储设备。
- **媒介请求调度层**：这一层负责调度操作系统发来的请求，使其符合设备的物理性能。例如，对于磁盘，这一层将实现第 10 章介绍的磁盘调度算法。如我们在第 10 章见到的，即使是磁盘，调度算法可能实际上就在作为驱动器一部分的设备控制器中。调度算法可能会相当不同，这取决于该大容量存储设备的特点。

图 11-15 显示了软件栈底三层的例子，每个例子是给用户提供的文件系统接口的大容量存储设备。因此，文件是一种强有力的抽象，其隐藏了所属物理设备上的具体细节。

11.4.1 创建、写入文件的剖析

假设程序发出了一个 I/O 请求要在硬盘上创建一个文件。下列步骤跟踪了当这样的 I/O 请求从图 11-15 的软件层发出所经过的路径：

1）创建文件调用的 API 例程通过检查许可、访问权限和其他与该请求有关的信息来检验请求是否合法。检验通过后，它调用名称解析器。

2）名称解析器告诉存储分配模块为新文件分配一个索引节点。

3）存储分配模块从空闲链表中得到一个磁盘块并将其返回给名称解析器。存储分配模块通过与分配方案相符合的方式填充索引节点（见 11.2 节）。假设我们使用混合方案进行分配（见 11.2.7 节）。由于文件创建时还没有任何数据，所以文件不会分配任何数据块。

4）名称解析器创建一个目录项并将名称记录到索引节点中，将信息映射到目录的新文件中。

注意这些步骤并没有真正涉及设备，因为文件系统访问的数据结构都在内存中。

现在假设程序向刚创建的文件进行写操作。我们来跟踪这个操作在软件层中的执行过程。

1）与前面一样，写文件的 API 例程检查该请求的合法性。

2）名称解析器将内存缓冲区与文件的索引节点信息一起传送给存储分配模块。

3）存储分配模块从空闲链表中分配与写入信息大小相当的数据块，然后向磁盘发出写请求并将该请求交给设备驱动器。

4）设备驱动器将请求放入请求队列。设备驱动器与磁盘调度算法一起完成写文件到磁盘的操作。

5）文件写入完成后，设备驱动器得到一个来自文件系统返回给磁盘控制器产生的中断，然后与 CPU 调度器通信后继续从文件写的地方向下执行你的程序。

应当注意到，就操作系统而言，当请求提交到设备驱动器后写文件的调用就完成了。这个调用的成功或失败在之后控制器进行中断时我们就会知道。文件系统与 CPU 调度器的交互方式与内存管理器一样（见 8.2 节）。为了处理页错误，内存管理器发送一个与错误进程有关的 I/O 请求。当 I/O 请求完成后，就会通知内存管理器，告诉 CPU 调度器继续故障进程的执行。这也是文件系统的处理方式。

11.5 各种子系统的交互

探究目前我们已经涉及的操作系统的各种软件子系统会很有意思：CPU 调度器、虚拟内存（VM）管理器、文件系统（FS）管理器以及各种输入 / 输出设备的驱动器。

当然，所有的这些子系统都为用户程序服务。例如，VM 管理器处理用户程序的页错误，

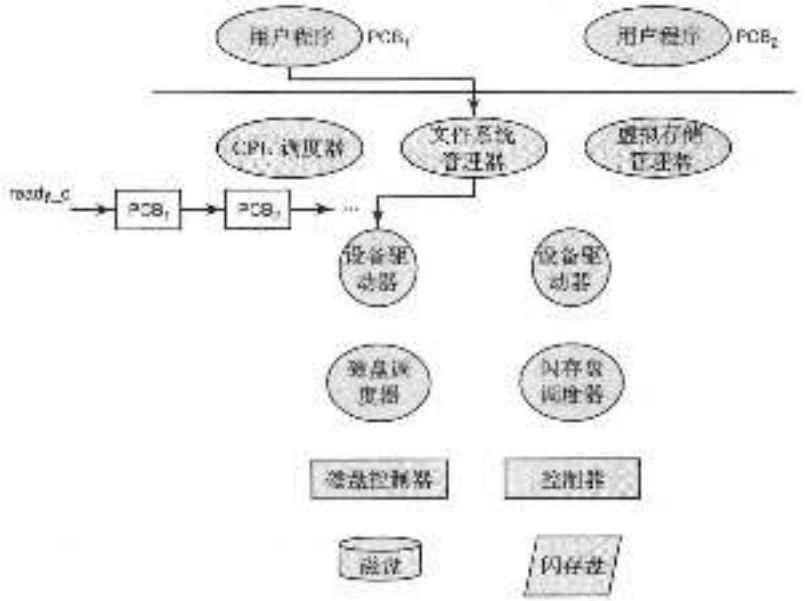
498 ↓ 499

500 通过隐式地调用与该故障进程有关的 I/O 请求。文件管理器显式地发出 I/O 请求来完成用户程序关于读/写文件的请求。在这两个例子中，I/O 操作的对象是某个大容量存储设备（硬盘、闪存、CD、DVD 等）。与特定 I/O 请求相关的设备驱动器执行 I/O 请求并将 I/O 操作的结果报告给请求者。

设备驱动器需要确定从设备控制器发来的 I/O 完成中断是谁的 I/O 请求。一个简单且统一的方法是使用 6.4 节介绍的 PCB 数据结构作为这些子系统之间通信的中介。

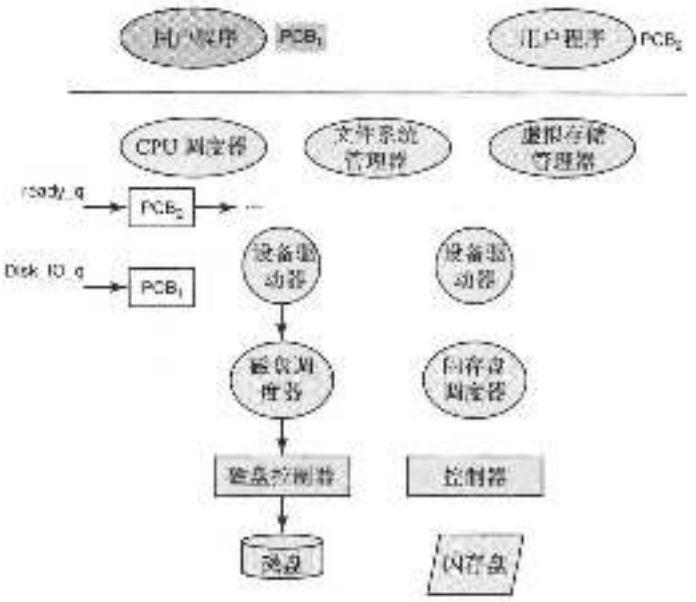
例如，文件系统管理器的顶层通过进程的 PCB 将文件 I/O 请求交给设备驱动器。这么做很直截了当，而且与过程调用很相似，除了过程调用穿过图 11-15 中软件栈的各个层外。信息流如图 11-16a 所示。PCB_i 表示发出文件系统调用的进程的 PCB。一旦调用提交给设备驱动器，进程就不再运行了，图 11-16b 中有相关描述。PCB_i 在由设备驱动器进行服务时位于 Disk_IO_q 中。

当 I/O 完成后，设备驱动器需要通知系统软件栈的上层。在第 6 章中（见 6.7.1 节），我们介绍了回调。这是系统软件栈中低层调用上层的机制。事实上，从图 11-6c 中可以看到，回调是连续的。首先，磁盘控制器通过中断的形式向设备控制器的中断处理器发出一个回调。中断处理器确切地知道这个 I/O 是从哪个与之相关的进程发起的（从 Disk_IO_q）。当然，设备驱动器中的中断处理器需要了解是非调用的。基于这个理由，每个系统栈中的上层需要给低层注册一个处理器来允许回调。通过这个处理器，设备驱动器向文件系统管理器发出回调来指明文件 I/O 请求的完成情况。我们可以看到这种回调机制与硬件中断（参见第 4 章）之间的相似性。两者在功能方面具有相似性（与系统进行异步的通信事件），在传达这种事件的机制方面也有相似之处。因此回调也通常称作软件中断。

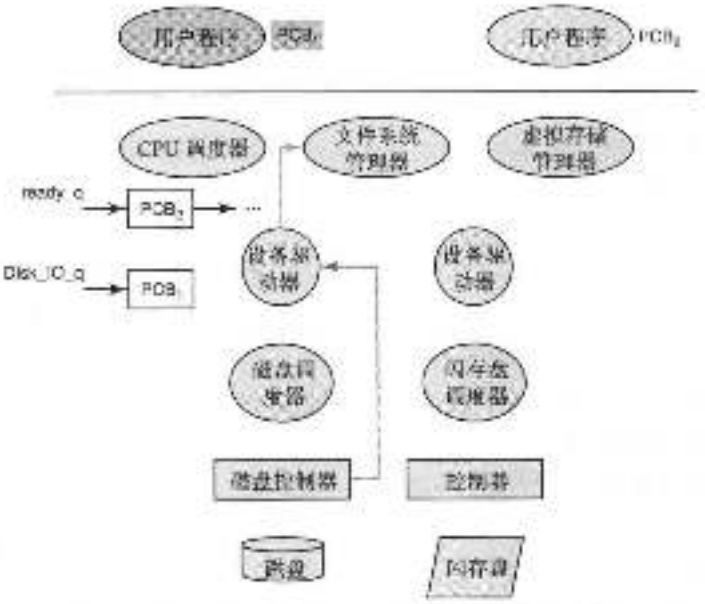


a）文件系统调用的信息流

图 11-16

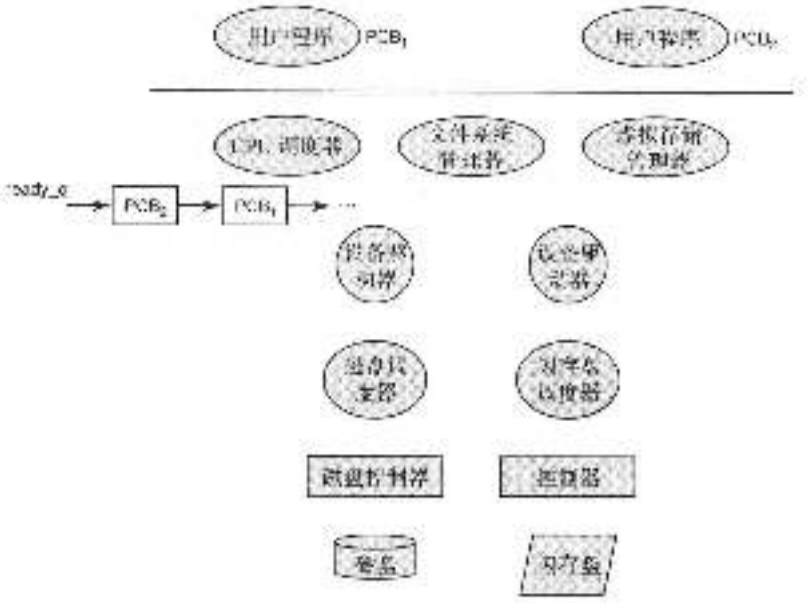


b) 磁盘驱动器处理来自 PCB_1 的文件系统调用



c) 文件 I/O 请求完成的一系列回调（磁盘控制器指向设备控制器的上箭头在硬件中；从设备控制器到文件系统管理器的上箭头在软件中）

图 11-16 （续）



d) 文件系统管理器将发送了I/O请求的进程（PCB1）放回CPU的ready_q中

图11-16 （续）

在收到回话后，文件系统管理器将请求进程的PCB（PCB_1）重新存入CPU调度器的ready_q，如图11-6d所示。

我们可以看到，操作系统组件之间的交互是通过用PCB的形式对执行程序进行抽象来平滑地实现的。这就是PCB抽象的威力。

类似序列事件会在一个进程出现页错误时发生，唯一的区别是虚拟存储管理器是图11-16a-d中软件栈中上下各个动作的发起者。

11.6 文件系统在物理媒介上的布局

我们来看一看操作系统如何在开机加电后控制系统中的资源。

在第10章中，我们探讨了操作系统启动后的基本思想。我们将解释BIOS如何进行设备的基础初始化并将控制交给系统软件的更高层。实际上，这会涉及操作系统启动过程。操作系统的映像放在大容量存储设备中。事实上，BIOS甚至不了解操作系统启动需要什么。因此，BIOS需要清楚地知道信息存储在哪里以及在大容量存储设备中怎样存放，这样它就可以读操作系统并将控制权交给它。换句话说，大容量存储设备上的信息布局成为BIOS和操作系统之间的协议，无论是Windows、Linux或是其他操作系统。

为了更具体地说明，我们假定大容量存储设备是一个磁盘。在磁盘存储空间的最开始位置，我们将其称作（柱面0，盘道0，扇区0），它是一个可激主引导记录（MBR）的特殊记录。当你通过编译过程创建一个可执行程序时，直到加载器将其加载到内存，这个程序一直放在磁盘上。同样，MBR只是一个放在磁盘上已知单元的程序。BIOS作为加载器将这个程序加载入内存，然后将控制权交给MBR。

MBR程序知道磁盘上剩余内容的布局，并且知道操作系统在磁盘上的确切位置。物理磁盘本身可能分成了多个分区。例如，在台式计算机或笔记本电脑上，你可能见过名称不同的

多个“驱动器”（微软的 Windows 将其用 C、D 等进行命名）。这些可能是不同的物理驱动器；但也可能只是逻辑驱动器，每个对应同一个物理驱动器上的某个独立分区。我们假设有一个单独的磁盘，但是能够双启动：可以让 Linux 和 Windows 成为你的操作系统。这只取决于你的选择。每个操作系统的文件系统当然也可以不同。这就是分区的作用。

图 11-17 显示了磁盘的一个概念布局。为了清晰地描述，我们将每个分区放在不同的磁盘盘面上。然而，需要强调的是，每个盘面可能有若多分区，这取决于磁盘容量。MBR 程序的关键数据结构是分区表。这张表（见表 11-4）给出了每个磁盘分区的设备开始和设备结束地址（例如，用三元组{盘面，磁道，扇区}的形式）。根据用户在启动时的选择，MBR 通过分区表决定激活哪个分区。当然，在有些系统上可能没有选择（例如，只有一个分区，或者只有一个分区有与其关联的操作系统）。

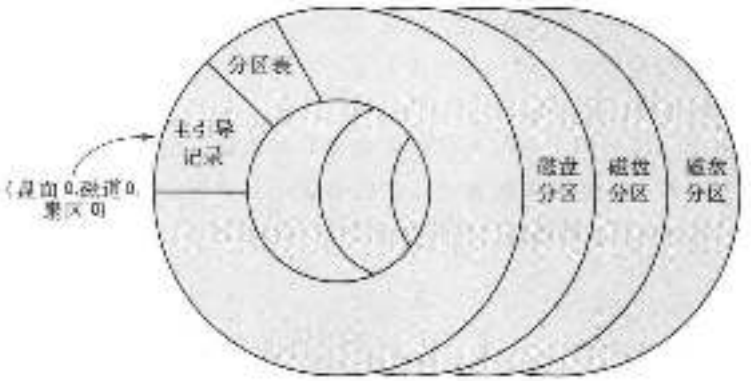


图 11-17 磁盘上信息的概念布局。在每个磁盘的盘面上可能有一个或多个分区

表 11-4 显示了多个分区。根据需要启动的操作系统，MBR 程序将激活相应的分区（在本例中，即分区 1 或 2）。注意分区 3 – 5 并没有与之关联的操作系统。它们只是给其中之一或其他操作系统使用的逻辑“驱动器”。

表 11-4 分区表数据结构

分区	开始地址{盘面，磁道，扇区}	结束地址{盘面，磁道，扇区}	操作系统
1	{1, 10, 0}	{1, 600, 0}	Linux
2	{1, 601, 0}	{1, 2000, 0}	MS Vista
3	{1, 2001, 0}	{1, 5000, 0}	无
4	{2, 10, 0}	{2, 2000, 0}	无
5	{2, 2001, 0}	{2, 3000, 0}	无

图 11-18 是每个分区的信息布局。除了第一项（启动块外），分区的实际信息布局在各个文件系统之间是不同的。然而，为了更具体一些，我们假定了一个特定的布局并描述了每项的功能。图 11-18 选择的信息布局与传统的 UNIX 文件系统非常相近。

我们来看一下分区中的每项：

- 启动块是每个分区中的第一项。MBR 从被激活分区的启动块读取数据。启动块是负责加载与该分区相关联的操作系统的程序（就像 MBR 一样）。为了一致性，每个分区都有一个启动块，即使在一个特定的分区中根本没有与之关联的操作系统（见表 11-4 中

的分区3～5）。

- 超级块包含了所有与该分区所含操作系统相关的信息。这是了解该分区其余部分布局的关键。启动程序（除了加载操作系统，或者代之外）把超级块读入内存。通常，它包含了一个代码，通常也称作魔数，它表明该分区中文件系统的类型。它还包含了磁盘块个数，以及其他与文件系统有关的管理信息。
- 分区中的下一项包含了该分区中用于存储管理的数据结构。这些数据结构与文件系统采用的特定分配策略有关（见11.2节）。例如，它可能包含表示所有可用磁盘数据块（比如，空闲链表）的位映射。作为另一个例子，这项中可能还包含我们之前提过的FAT数据结构（见11.2.4节）。
- 分区中的下一项对应于文件系统维护的每个文件信息。这个数据结构与文件系统的特性有关。例如，在11.3.1节的UNIX文件系统中，每个文件有一个与之关联的唯一编号，也称索引节点号。在这样的系统中，这项可能是文件系统中所有索引节点的集合。
- 现代文件系统都是层次化的。分区中的下一项指向了层次化树状文件系统的根目录。例如，在UNIX中，它对应于目录“/”。
- 分区中的最后一项是用于存储数据和目录文件的磁盘块集合。分区中存储管理项的数据结构负责分配/回收这些磁盘块。它们可能用于存储数据（例如，一张JPEG图像）或者包含了其他文件（即数据文件或其他子目录）的目录。

505
～
506

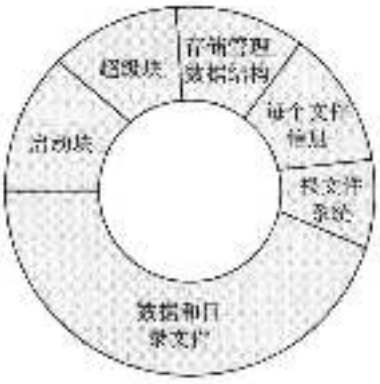


图11-18 每个分区的布局。分区中的项只是存储设备用于组织磁盘信息的数据结构

显而易见，超级块是文件系统的一个关键数据结构。如果由于某些原因将其损坏了，那么就很难恢复文件系统的内容。

11.6.1 内存中的数据结构

为了更具效率，文件系统会在启动时把关键的数据结构（超级块、索引节点的空闲链表以及数据块的空闲链表）从磁盘读取进来。当用户程序创建和删除文件时，文件系统就会处理这些存放在内存中的数据结构。事实上，文件系统甚至不会马上将创建的数据文件写入大容量存储设备。这种延后有两方面的原因。首先，大量的文件，尤其是在程序开发环境里，都具有较短的生命周期（小于30秒）。考虑那些在程序编译和链接过程中创建的中间文件，创建这些文件的程序（例如，编译器和链接器）在创建完执行文件后就会将它们删除。因此，过一会儿再执行向大容量存储设备的写入操作会有一定好处，因为这样延迟可以有助减少I/O

数据量。其次，是为了方便和效率。文件系统可以批量地进行 I/O 请求，从而既可以减少单独发出请求的开销，又可以较好地处理发自设备 I/O 的中断。

这种策略也有不足。内存中的数据结构与它们在磁盘上的对应部分有不一致之处。当尝试从计算机中拔掉内存条时，我们注意到计算机屏幕上显示消息“移除设备不安全”。这是因为操作系统还没有将这些内存中的数据结构（甚至于数据自身）的变化提交到大容量存储设备上。大多数操作系统都提供了强制执行的命令。例如，UNIX 中的 sync 命令可以强制把所有内存缓冲区中的数据写到大容量存储设备上。 507

11.7 处理系统崩溃

我们来看一看操作系统崩溃究竟是什么意思。我们经常说，操作系统是一个程序。它与你可能写过的其他部分软件一样容易发生故障。如果一个用户程序做了一些它不应当做的事（例如，尝试访问一部分超过其边界的内存）就会被操作系统终止，或者这个程序可能会因为需要等一个永远不会发生的事件而挂起。操作系统也可能产生这种错误，也可能突然终止执行。另外，电源故障会强制性地导致操作系统的执行被终止。这就是所谓的系统崩溃。

文件系统是计算机系统的一个关键部件。由于它存放了永久性数据，所以其稳定性对于任何企业的生产力而言都是最为重要的。因此，文件系统能够在系统崩溃后继续正常工作就非常重要了。操作系统非常小心地保证文件系统的完整性。如果系统意外地崩溃了，那么在崩溃时存储于内存中的数据结构与磁盘上的版本就会产生不一致性。

基于这种原因，作为一种最后手段，操作系统会在终止前（无论崩溃是由操作系统漏洞还是电源故障引起的）把内存的内容导出到大容量存储设备的某个已知位置上。系统启动时，启动程序最先做的一件事情就是查看是否有这样的崩溃映像。如果有，那么它就会尝试重建存储在内存中的数据结构并让磁盘上的版本与之吻合。台式计算机启动时会花一点儿时间的一个原因就是操作系统需要进行一致性检查来保证系统完整性。UNIX 操作系统自动在启动时进行文件系统一致性检查（fsck）。只有当系统通过了一致性检查后启动过程才会继续下去。企业都会进行磁盘往磁带上周期性的备份来避免故障。

11.8 其他物理媒介上的文件系统

到目前为止，我们都假设物理媒介是磁盘。我们知道文件系统可以存放在各种物理媒介上。如果大容量存储器的物理媒介不同，我们目前讨论的内容在什么程度上会发生改变呢？CD-ROM 和 CD-R（可刻录 CD）可能是最简单的。一旦刻录后，文件就不会在这种媒介中被清除，这就简化了文件系统的复杂度。例如，对于前者，CD 上就无需空闲链表；相比之下，后者的所有空闲空间都在 CD 的尾部，可以将文件加到后面。CD-RW（可重写 CD）的文件系统就稍微复杂一些，因为删除文件的空间需要被放回到媒介的空闲空间上。DVD 的文件系统也类似。

在第 10 章中提到，固态硬盘（SSD）当前正与磁盘在大容量存储媒介上进行竞争。SSD 可以随机访问，所以磁盘块的寻道时间（在基于磁盘的文件系统中一个主要担心的问题）在基于 SSD 的文件系统中就不是特别令人担心。因此，在分配策略上就有一定的机会进行简化。然而，对于 SSD 的文件系统实现也有一些特别需要考虑的地方。例如，SSD 的一个给定区域（通常指一块）只能进行有限次的写入，之后就会不可用。这是由 SSD 技术特点决定的。因 508

此，SSD 的文件系统采用损耗均衡的分配策略来保证所有的存储区域在 SSD 的生命周期中被均衡地使用。

11.9 现代文件系统一览

本节我们将研究一些现代文件系统演变中一些令人兴奋的新发展。第 6 章中，我们从历史的角度介绍了 UNIX 系统的发展以及它如何为 Linux 和 Mac OS X 铺平了道路。现在，Mac OS X、Linux 和 Windows 已经在各种应用领域占领了广泛的市场。这里，我们将我们的讨论集中在 Linux 和微软系列操作系统的文件系统上。

11.9.1 Linux

Linux 提供的文件系统接口（即 API）与早期 UNIX 系统相比并无太大改变。然而，内部的文件系统实现经历了巨大的变化。这与第 2、3 章中处理器设计中提到的架构与实现分级有其相似性。

大多数革命性的变化是为了适应多文件系统分区、更长文件名、更大文件以及隐藏本地媒介上的文件与网络上的文件的区别等而设计的。

一个重要的 UNIX 文件系统改变引入了虚拟文件系统（VFS）。这种抽象使得多个存在不同的文件系统可以“在表层下”共存。文件系统可以驻留在本地设备上，一个通过网络访问的外部设备，或者驻留在其他不同类型的媒介上。VFS 并不影响用户。你打开、读、写文件，与你在传统 UNIX 文件系统上所做的一样。VFS 的抽象十分清楚那个文件系统为你的文件负责，通过间接层（VFS 抽象层中存储的函数指针）将你的调用重定向到特定的可以服务你请求的文件系统上。

现在我们来快速地看一看 Linux 自身文件系统实现中的一些发展。在第 6 章中，我们提到了 MINIX 如何作为 Linux 的入口工作。Linux 的第一个文件系统使用了 MINIX 文件系统。它在文件名的长度和文件的个数上有限制。因此，它很快就能够避免这些限制的 ext（表示可扩展文件系统）替代了。然而，ext 在性能上有些不够高效，从而在其基础上出现 ext2（ext 509 的第二个版本）这种目前依旧在 Linux 社区中广泛使用的文件系统。

ext2 ext2 文件系统分区的整体布局与 11.6 节中的通常描述没有显著的区别。ext2 的一些较为实用的内容是：

- **目录文件**：第一个是目录文件索引节点结构的内容。一个目录文件由零数个连续的磁盘块组成，使得在一个 I/O 操作内可完成向磁盘写目录文件。目录中的每一项有一个文件或目录名。由于名称可以任意长，所以每一项的大小是可变的。图 11-19a 是目录项的格式。

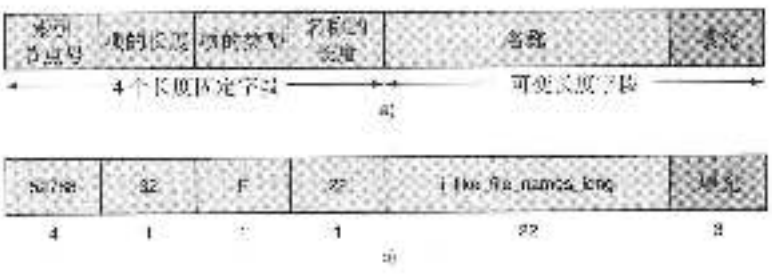


图 11-19 目录文件中每一项的格式

每一项由以下项目组成：

- **索引节点号**：一个定长的字段，给出了关联项的索引节点号。
- **项的长度**：一个定长的字段，以字节为单位表明了项的长度（即该项在磁盘上占据的空间大小）——它可以用来确定下一项在目录结构中的起始位置。
- **项的类型**：一个定长的字段，说明该项是数据文件（f）还是目录文件（d）。
- **名称的长度**：一个定长的字段，表明文件名的长度。
- **名称**：一个边可变长的字段，以 ASCII 给出文件名。
- **填充**：一个可选的可变长的字段，有时用于将一项的长度变成 2 的幂。我们很快会看到，这种填充空间可能会在文件删除时被创建或者扩展。

图 11-19b 是一个名为“i_like_my_filenames_long”的文件的示例项，其中已经填入了一些值。每个字段的大小（以字节为单位）也在图中标出来了。

目录中的项以文件创建顺序排列。例如，你创建了文件“datafile”、“another_datafile”以及“my_subdirectory”，前两个是数据文件，第二个是目录文件。目录项如图 11-20a 所示。假设你删除了其中的某个文件，例如“another_datafile”。此时，在图 11-20b 中你就会发现有一块空闲空间空出来了。被删除文件项的空间成为前一项的填充部分。在目录中创建新文件项时就可以再次利用这块空间了。



图 11-20 目录文件中多个项的布局

- **数据文件**：第二个有趣的地方就是数据文件的索引节点结构。按照之前的说法，ext2 将 MINIX 有最大文件数目的限制消除了。数据文件的索引节点结构就反映了这种改变。这里使用 11.2.7 节中的混合分配方案。数据文件的索引节点包含了下列字段：
  - **12 个数据块地址**：数据文件的前 12 个数据块的地址可以通过索引节点直接获得。对于小文件而言，这么做很方便。文件系统也尝试连续地分配这 12 个数据块以便性能可以更加好。
  - **1 个一级间接指针**：这个指针指向的磁盘块可以用作存放数据块指针的容器。例如，一个 512 字节的磁盘块，若一个数据块指针大小为 4 字节，那么第一级间接索引就可以把文件大小扩展 128 个数据块。
  - **1 个二级间接指针**：这个指针指向的磁盘块也是一个存放了指针的容器，其中每个指针指向包含了一级间接指针的磁盘块。接着前面的例子，二级间接索引可以将文件大小再扩展 128×128（即 2^{14}）个数据块。
  - **1 个三级间接指针**：这个指针在二级间接指针的基础上又加了一级索引，使得文件还可以扩展 $128 \times 128 \times 128$（即 2^{21}）个数据块。

图 11-21 给出了这样一个索引节点的所有填入字段的信息。

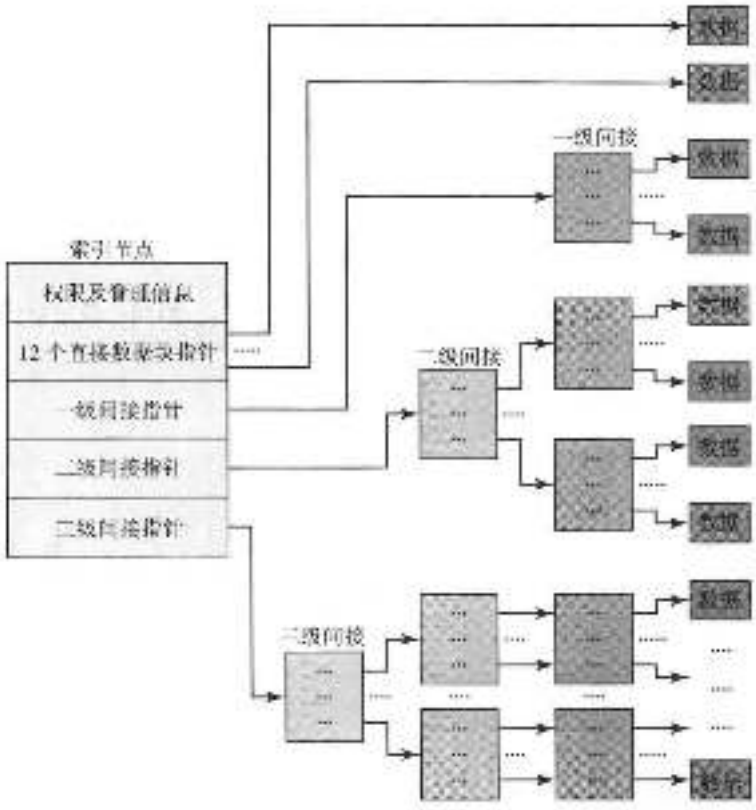


图 11-21 Linux ext2 中一个数据文件的索引节点结构

例 11-8 ext 系统中文件最大大小是多少（以字节为单位）？假设一个磁盘块为 1KB 且磁盘块指针需要 4 字节。

答：

直接数据块

[512]

$$\text{直接数据块的个数} = \text{索引节点中直接指针的个数} = 12 \tag{11-1}$$

通过一级间接指针得到的数据块个数

接下来，通过索引节点中一级间接指针计算可以得到的文件的可用数据块个数。一级间接指针指向一个包含了数据块指针的一级间接表。

$$\begin{aligned}&\text{通过索引节点一级间接指针可获得的数据块个数}\\&= \text{一级间接表中的指针数}\\&= \text{磁盘块中的指针数}\\&= \text{磁盘块大小} / \text{指针大小}\\&= 1\text{KB} / 4\text{ B}\\&= 256\end{aligned} \tag{11-2}$$

通过二级间接指针得到的数据块个数

下面，我们计算通过索引节点的二级间接指针可以获得的数据块个数。二级间接指针指向一个包含了指向多个一级间接指针表的多个指针。

通过索引节点的二级间接指针可表示的[illegible]最大块数

=磁盘块中的指针数

=磁盘块大小/指针大小

=1KB/4B (11-3)

=256

通过索引节点的一级间接指针可表示的数据块个数（见式（11-2））

=256 (11-4)

由式（11-3）、（11-4）可知，

通过索引节点的二级间接指针可以表示的数据块个数

=256×256 (11-5)

通过三级间接指针得到的数据块个数

通过类似的分析可知，通过索引节点的三级间接指针能够获得的数据块个数

=256×256×256 (11-6)

由式（11-1）、（11-2）、（11-5）、（11-6）可知， 513

一个文件能够获得的磁盘块总数

=12+256+256×256+256×256×256

=12+2^8+2^{16}+2^{24}

最大文件最大大小（磁盘块大小=1KB）

=(12+2^8+2^{16}+2^{24})KB

>16GB

日志文件系统 大多数现代的Linux文件系统都是日志文件系统。通常，你会觉得向文件写入就是写入文件数据块。逻辑上，这并没错。然而，实际中，会有各种问题出现。例如，如果文件太小了，那么就会既有空间开销，（更重要的是）还会有将这些小文件写到磁盘上的时间开销。当然，操作系统的缓冲区会把文件写到内存里然后过一会儿再写入磁盘。除了这种优化外，最终文件还是被写入磁盘。小的文件不仅浪费磁盘空间（由于内部碎片问题），也会导致时间开销（寻道时间、旋转延迟以及元数据管理）。

另一个原因是系统崩溃。如果系统在写入文件的过程中崩溃了，那么文件系统就会停留在一个不一致的状态上。我们已经略微提及过这个问题（见11.7节）。

为了解决写入小文件效率低下以及从系统崩溃恢复的问题，现代文件系统使用了日志方式。这个想法很简单直观，就好比记录一个个人的日志或日记一样。一个人的日记是以时间为序的一个人每天活动中重要事件的记录。日志对文件系统也起同样的作用。在写文件时，文件系统不是写文件和创建一个小文件，而是创建包含了与该文件写相关的信息（即对索引节点和超级节点的元数据修改，以及对文件数据块的修改）一个日志记录（类似于一个数据库记录）。因此，日志就是文件系统所有变化的一个以时间为序的记录。这种方式的好处是日志与文件系统本身分离，让操作系统对其实现进行优化来最好地满足操作系统的需求。例如，日志可能是用一个线性的数据结构实现的。这个线性数据结构中的每一项表示了一个特定的文件系统操作。

例如，假设你按照一定顺序修改了三个文件（X、Y和Z）中的某些数据块，这些数据块分散在磁盘的各个位置。与这些修改相关的日志项如图11-22所示。注意每条日志记录的大小可能不同，这取决于相应写操作一次修改的文件中的数据块个数。 514

日志数据结构由多个日志段组成。每个日志段有右界的大小（例如，1MB）。当一个日志

段满时，文件系统会把它写到磁盘上一个连续分区并为接下来的文件系统写操作开出一个新的日志段。注意，文件 X、Y、Z 并不反映对它们所做的改变。每过一会儿，文件系统会读取日志段（以它们生成的顺序），将文件的变化提交实际的文件，并将日志项应用到对应文件上。一旦提交了变化，该日志段就作废了。如果文件 X 在它的变化被应用前被读取，那么文件系统可以足够智能在读取前将日志段的变化应用到文件。



图 11-22 修改文件系统的日志项

显然，日志段将小的写操作（可以是小文件也可以是大文件中的一小部分）合并起来变成日志上较粗粒度的操作（即，向磁盘上连续部分进行较大的写操作）。

日志系统克服了小的写操作问题。作为一个额外功能，它还能帮助系统更好地应对崩溃问题。作为一种最后手段（见 11.7 节），操作系统在崩溃或电源故障时将所有内存中的信息写入磁盘。重启时，操作系统将恢复内存中的日志段。文件系统将识别出日志段的变化（此时磁盘上和内存中的日志段都从崩溃中恢复了）没有被成功提交。文件系统将简单地重新应用日志记录使得其一致性得到满足。

ext3 是 Linux 文件系统的下一个版本。与 ext2 相比，ext3 文件系统的主要扩展是支持日志。在这种意义上，用 ext2 建立的文件分区可通过 ext3 文件系统访问，因为两者的数据结构和内部抽象是一样的。由于创建一个所有文件系统操作的日志可能在空间和时间上有较高的代价，所以 ext3 可以只对元数据的变化（即索引节点、超级块等）做日志。这种优化在系统崩溃时可以提供更好的性能，但是无法在崩溃后保证数据的正确性。

还有一些新的 UNIX 文件系统。ReiserFS 是 Linux 系统只有元数据日志的另一个文件系统。jFS 是 IBM 的一个日志文件系统，用于 IBM 的 AIX（也是一个 UNIX 版本的操作系统）和 Linux。xFS 是一个主要为 SGI Irix 操作系统设计的日志文件系统。zFS 是一个高端企业级系统使用的高性能文件系统，其主要针对 Sun 的 Solaris 操作系统。这些文件系统的讨论超出了本书的范围。

11.9.2 Microsoft Windows

微软的文件系统有一段非常有趣的历史。如你所知，MS-DOS 一开始就作为面向 PC 的操作系统。由于在 PC 发展的早期，磁盘的容量都非常小（大约为 20 ~ 40MB 的级别），所以这些
515 计算机的文件系统也受到了限制。例如，FAT-16（11.2.4 节中讨论的分配方案中的一个特别的例子）使用 16 位磁盘地址，给个文件分区有 2GB 的大小限制。FAT-32，使用了 32 位磁盘地址，将文件分区的大小限制扩展到了 2TB。这两个文件在微软的 XP 和 Vista 系统上已经逐渐被淘汰了。2010 年前后，NT 文件系统（也称 NTFS，在微软的 NT 操作系统中首次引入），已经成为事实上的标准微软文件系统。FAT-16 可能在一些可移动媒介（例如，软盘）上仍在使用。FAT-32 也有用途，尤其是为了兼容较旧版本的 Windows（例如，Windows 95、Windows 98）。

NTFS 支持大多数我们在 UNIX 文件系统中讨论的特性。它使用了 64 位磁盘地址，因此可以支持非常大的磁盘分区。卷是该文件系统的基础单元。一个卷可能占据磁盘上的一部分、整个磁盘，甚至多个磁盘。

API和系统特性　NTFS与UNIX之间的一个基本区别是文件的视角。在NTFS中，文件是由一些具有类型的属性组成的面向对象，而不是UNIX中的字节流。这种文件视角在用户层面可以提供一些较为明显的灵活性。每个具有类型的属性是一个单独的字节流。这就使得我们可以在不影响文件其他部分的情况下创建、读、写、删除某个属性。有些属性类型是所有文件都有的（例如，名称、创建时间、访问控制等）。比如，你可以给一个图像文件具有多个属性：原始图像、缩略图等。一个文件的属性还可以任意创建或者删除。

NTFS支持长文件名（最长为255个字符），通过Unicode字符还可以用非英语文字对文件进行命名。它与UNIX类似，也是一个层次化的文件系统，尽管UNIX中的层次分隔符"/"在NTFS中被替换成了"\"。我们在11.1节中讨论过的文件别名的软链接和硬链接，在最近也被加入NTFS文件系统中。

NTFS的一些有趣特性还包括：当文件分别进行写和读时，实时压缩和解压缩；一个可选的加密特性；以及通过日志来支持小文件的写入和系统崩溃。

实现　与UNIX的索引节点表（见11.3.1节）类似，NTFS的主要数据结构是主文件表（MFT）。它也存储在卷上，并包含了文件系统其他部分的重要元数据信息。一个文件通过MFT中的一个或多个记录表示。这取决于属性的个数以及文件的大小。通过位映射来表示那些MFT记录是空闲的。表示一个文件的MFT记录集合在功能上与UNIX中的一个索引节点相似，但是，只是相似而已。一个文件的一个（或多个）主记录包含了下列属性：

- 文件名。
- 时间戳。
- 安全信息（用于文件的访问控制）。
- 包含了数据的数据或者间接块的指针。
- 一个可选的指向其他MFT记录的指针。如果文件太大无法用一个记录表示或者如果文件拥有多个属性，所有这些都无法放入一个MFT记录中。

每个NTFS文件有一个唯一的ID，即散列索引号，是64位。这个ID是MFT中该文件的索引号，它提供了与UNIX中索引节点编号类似的功能。 516

存储分配方案尽最大可能为文件的数据块分配连续的磁盘块。为此，磁盘分配方案维护以簇为单位的可用磁盘块，其中每簇表示一段连续的磁盘块。簇的大小由磁盘格式化时的一个参数进行指定。当然，我们无法每次都把连续的磁盘块分配给所有的数据块。比如，考虑一个拥有13个数据块的文件。假设簇大小为4块，空闲链表包含了起始于磁盘块地址为64、256、260和408的簇。在这个例子中，就会有3段不连续的分配，如图11-23所示。

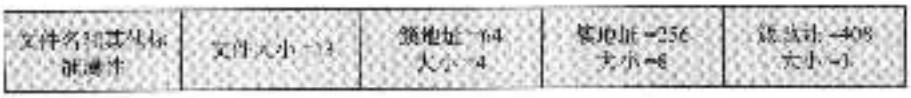


图11-23　一个由13个数据块构成的文件的MFT记录

注意这种分配方案可能导致内部碎片，因为在簇408中有3个剩余的磁盘块未被使用。

一个MFT记录通常有1～4KB的大小。因此，如果文件太小，文件的数据就会包含在MFT记录自身内，这也就解决了写小文件的问题。而且，基于簇的磁盘分配保证了较好的线性访问性能。

小结

在本章中，我们学习了可能是系统中最重要的一个部分，文件系统。本章包括如下内容：

- 与文件关联的属性。
- 磁盘存储管理使用的分配策略和相关的数据结构。
- 文件系统管理的元数据。
- 文件系统的实现细节。
- 操作系统各个子系统之间的交互。
- 磁盘上文件的布局。
- 文件系统的数据结构及其有效的管理。
- 系统崩溃的处理。
- 其他物理媒介上的文件系统。

我们也了解了一些目前（2010年前后）正在使用的现代文件系统。

517 文件系统在研究和开发方面还是一块肥沃的土地。在文件系统中，你可以看到在处理器架构与实现的类比。在文件系统API的大部分内容都在各个版本的操作系统（UNIX、Microsoft等）间保持不变的情况下，文件系统的实现却在不断地发展，其中的一些是由于磁盘技术上的进步，还有一些是由于应用中文件类型的变化。例如，现在我们经常处理很多多媒体文件（音频、图形、照片、视频等），而文件系统的实现就需要能够适应这些多种的文件类型。

本章只是涉及文件系统实现中的一些表面知识。我们希望可以激发你的兴趣，并继续学习操作系统方面更高级的课程。

练习题

1. 一个文件的属性数据存储在哪里？说出每种方案的优劣。
2. 观察下列UNIX中的目录项：

```
-rwxrwxrwx  1  name  0  Apr 27 21:01  foo
```

当执行下列命令后：

```
chmod u-w foo
chmod g-w foo
```

文件“foo”的访问权限是什么？

3. 选择所有正确的：

链式分配具有：

- 较好的线性访问
- 较好的随机访问
- 较容易扩展文件大小
- 较差的磁盘利用率
- 较好的磁盘利用率

4. 选择所有正确的

对磁盘空间进行固定连续分配具有：

- 较好的线性访问
- 较好的随机访问

- 较容易扩展文件大小
- 较差的磁盘利用率
- 较好的磁盘利用率

5. 假设：

磁盘柱面数	=6000
盘片数	=3
每个盘面的面数	=2
每个磁道的扇区数	=400
每个扇区的字节数	=512
磁盘分配策略	=连续柱面

518

a. 一个 1GB 大小的文件需要分配多少个柱面？

b. 这种分配导致的内部碎片有多大？

6. FAT 分配策略的问题是什么？

7. 比较链接分配策略与 FAT 分配策略。

8. 索引分配怎样解决了 FAT 和链接分配的问题？

9. 假设在磁盘上使用索引分配方案：

- 磁盘有 3 个盘面（每个盘面有 2 个面）；
- 每个面有 4000 个磁道；
- 每个磁道有 400 个扇区；
- 每个扇区有 512 字节；
- 每个索引节点是一个占据了一个扇区的大小的固定数据结构；
- 一个数据块（即分配单位）由连续的 4 个柱面组成；
- 磁盘数据块指针由一个 8 字节的数据结构表示。

a. 该系统上的一个文件最小占据多少空间？

b. 这种分配方案下，一个文件的最大大小是多少？

10. 考虑以下的混合分配方案：

- 索引块大小 = 256 字节
- 数据块大小 = 8KB
- 磁盘块指针大小 = 8 字节（无论是数据块还是索引块的指针）
- 一个索引节点包含 2 个直接块指针，1 个一级间接指针，1 个二级间接指针，1 个三级间接指针。

a. 该文件系统的一个文件最大可以有多少字节？

b. 存储一个 1GB 的文件需要多少数据块？

c. 存储一个 1GB 的文件需要多少索引块？

11. 硬链接和软链接的区别是什么？

519

12. 考虑下列命令：

```
touch f1                /* 创建文件 f1 */
ln -s f1 f2             /* 符号链接 */
ln -s f2 f3
ln f1 f4                /* 硬链接 */
ln f4 f5
```

a. 通过这些命令一共会创建多少个索引节点？

b. 每个创建的索引节点的引用计数分别为多少？

13. 对一个磁盘块大小为 8KB、磁盘块指针大小为 4 字节的 ext2 文件系统，能够存储的最大文件可以是多少？画出索引节点结构的草图并说明达到最大文件大小的计算过程（参见例 11-8）。

参考文献注释和扩展阅读

在系统方面，处理器的微架构、缓存存储器和文件系统大概是3个得到较好研究的话题。这3个话题的定位也许是最能够理解的，因为它们对应用程序的性能具有最大的影响。UNIX由贝尔实验室的Ritchie和Thompson发明，而大多数普及UNIX的工作，尤其是在学术界，大多数工作是由UC-Berkeley的研究人员所做的，特别是Bill Joy，Sun Microsystems的联合创始人。读者可以阅读由[McKusick，1984]发表的开创性论文，其中综述了Berkeley UNIX文件系统的很多设计原理。McKusick等人[McKusick，1996]对Berkeley UNIX文件系统的设计与实现进行了详细的描述。对于任何希望编写操作系统的学生而言，这是一份宝贵的资料。由Prabhakaran等人[Prabhakaran，2005]发表的论文也很重要，其中介绍了日志文件系统的发展进程，包括了Linux ext3、ReiserFS和Microsoft NTFS等。Linux，作为一个开源项目，在不断地进步。因此，学习Linux的最好来源就是大量的在线资源[⊖]。Love[Love，2003]和Bovet、Cesati[Bovet，2005]的关于Linux的书也是很好的学习Linux的补充资源。
520 读者可以通过阅读[Russinovich，2005]来了解Microsoft文件系统的一些细节。

⊖ www.linux.org。

多线程编程与多处理器

多线程是一项让程序可能并发地执行多任务的技术。从计算机发展的早期开始，开拓并行性就是计算机科学家的一个追求。在 20 世纪 70 年代早期，Concurrent（并发）Pascal 和 Ada 的程序语言就加入了表达程序级并发性的特性。人们一直是并行地思考和完成各种事情。比如，你在读书时可能还会听一些最喜欢的背景音乐。我们经常会在与某人讨论某个重要话题的同时用双手干一些其他事情，比如修汽车或者把洗好的衣服叠起来等。计算机扩展了人类计算的能力，我们自然也希望计算机能够代替我们并行地执行各种我们希望完成的任务。顺序编程让我们不得不以顺序的方式来表达我们的计算需求。很不幸，由于我们并行地思考却只能写顺序执行的程序表达我们的想法。举个例子，考虑一个视频监控的应用。我们希望计算机能从 10 个不同摄像头上连续地收集图像，单独地分析每个图像来找出任何可疑行为，并在发生情况时进行报警。在这种描述中并没有什么顺序执行的部分。然而情况恰恰相反。如果我们想用 C 语言编写一个计算机程序来执行这个任务，我们最终会用顺序的方式来编写代码。

本章目的是介绍开发多线程程序的概念，操作系统为了实现这些概念所需的支持，以及为了实现操作系统机制所需的架构支持。对学生们而言，并行编程及其相关问题都是非常必 521
要的内容，因为现在的单芯片处理器包含了多个处理器核。因此，并行处理在当前的计算系统中已经非常常见了，无论在低端或者高端的机器上。我们在本章中要传达的一个重点是多线程所需的线程和系统支持是非常简洁明了的。

12.1 为什么需要多线程

多线程让我们得以在算法中表达其内在的并行性。线程与进程非常类似，两者都代表了一个活动的处理单元。在本章的后面，我们将讨论线程和进程在语义上的区别。这里，我们只需了解一个用户级别的进程可能由多个线程组成。

我们首先了解一下多线程能在编程级帮助我们什么。第一，它允许用户程序用模块化的方式表达其中的并行行为，这就像过程抽象能帮助我们用模块化的方式组成顺序执行的程序一样。第二，它帮助程序中具有 I/O 操作的程序进行重叠计算。我们从第 10 章中了解到 DMA 让一个高速 I/O 设备可以直接与内存进行通信，而无需处理器的干预。图 12-1 是一个周期性进行 I/O 操作的程序，但是并不马上需要 I/O 的结果。将 I/O 活动表达成一个单独的线程可以帮助我们在通信时利用这些时间进行计算。



图 12-1 使用线程，在 I/O 时进行重叠计算。b) 中的计算线程可以继续执行与 I/O 线程执行的 I/O 活动无关的计算

然后，我们看一看多线程在系统级能够提供哪些帮助。现在，在一个计算机

甚至一个芯片里有多个处理器已经非常常见了。这是多线程很重要的另一个原因，因为任何在用户级并发的表达方式可以帮助利用存在于计算中的硬件并行性。想象一个视频监控的程序需要做的事情。图 12-2 展示了与一个单独视频流相关联的处理过程。程序的数字化部件不断地将视频转换成一系列的像素映。跟踪部件分析每一映中需要标注的内容。报警部件基于跟踪来采取具体行动。应用的流水线与处理器的流水线很相似，即使在一个更大的级中。因此，如果我们在计算机内部有多个可以自动工作的处理器，它们就可以并行地执行这些程序部件，从而提升应用程序的性能。

所以，从程序的模块性、更好地利用 I/O 与计算的重叠，以及并行处理所带来的性能提升的角度来看，多线程是非常吸引人的。

本章中，我们将以图 12-2 的应用程序为例，通过其运行的例子来建立多线程的编程概念。

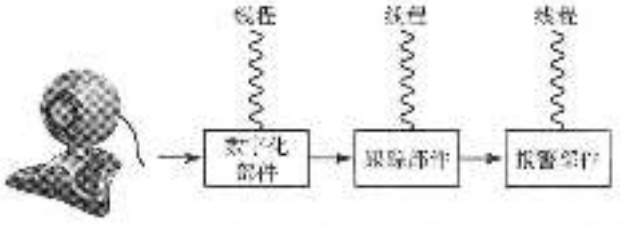


图 12-2 视频处理流水线。一个并领运行的应用程序，摄像头不断地捕捉一帧帧的图像并通过数字化部件将其数字化，由跟踪部件分析，并由报警模块触发控制动作

12.2 线程所需的编程支持

既然我们已经知道了线程可以作为表达并发性的工具，那么线程需要些什么来支持其作为一种编程抽象呢？我们希望能够动态地创建线程，终止线程，与其他线程进行通信，并在线程活动之间进行同步。

新像系统提供了一个包含程序员需要的常用函数的数学库，系统也提供了函数库来支持线程抽象。我们在接下来的几个小节中介绍能够提供这样一个库的技能。需要注意的是，数据类型和库函数使用的语法仅用于说明，真正的语法和支持的数据类型在线程库的不同实现中会有所不同。

522 / 523

12.2.1 线程创建和终止

一个线程执行一段程序。考虑程序和进程之间的关系，一个进程在程序入口处开始执行一个程序，在 C 语言程序中就是 main 过程。相比之下，我们可能希望用线程表达并发，可以启动在程序的任何点。也就是说，线程的入口处可以是任何用户定义的过程。我们把顶层过程定义为一个（以程序设计语言的可见性规则而言）可见的过程名称，无论该过程究竟在哪里被用作一个线程创建的对象。顶层过程可以有一些输入参数。

因此，一个典型的线程创建调用可能是如下所示：

```
thread_create(顶层过程，参数列表);
```

线程创建调用给出了顶层过程的名称以及传递的参数列表，使线程从那个过程开始执行。从用户程序的角度来看，这个调用会创建一个执行单元（即，线程），而它与发出这个调用的当前线程是并发的（图 12-3 中的前 / 后关系）。

因此，thread_create 函数实例化一个新的称为线程的有其自己生命周期的独立实体。

这与父母生下孩子类似。一旦孩子出生，学会走路，他就会不再依赖父母，开始在家的周围漫步做着各种自己的事情（当然需要在其父母的限制之内）。这就是实际上发生在线程上的事情。一个执行的程序是一个进程。在顺序程序中只有一个拥有控制权的线程，即进程。所谓拥有控制权的线程，指的是一个活动的实体，在程序的内存印记上漫游，执行程序员的意图。既然一个线程有其自己的生命周期，那么它就可以做它自己的事情（当然也是在其父进程的限制之内）。例如，在图 12-3 中，一旦创建成功，“子线程”就可以在其代码内进行过程调用，而不管“主线程”在它的代码体内正在干什么。



图 12-3 线程创建

我们来看一看对于线程的限制。还是以人类为例，父母会在靠近楼梯的地方放一扇门来防止孩子爬楼梯，安装对儿童安全的锁柜等。对于线程，也有类似的由程序设计语言和操作系统规定的限制。一个线程在顶层过程中开始执行。换句话说，一个多线程程序的入口也是一个普通的顺序程序。因此，程序设计语言的可见性和作用域规则对一个线程能够在生命周期中操作的数据结构也有同样的约束。当操作系统将程序实例化进程时，它会给每个程序创建一个唯一的内存空间，即地址空间。如我们在之前章节里看到的，地址空间包含了每个程序特定的代码、全局数据、栈和堆等部分。进程在这个“沙盒”里运行。该进程的子进程或子线程也只能在同样的沙盒里运行。 524

读者可能会疑惑，进程和线程之间有什么区别。我们会在 12.7 节关于操作系统对线程的支持部分详细阐述两者的区别。这里，我们只要知道与进程有关的状态数远远大于与线程相关的就可以了。另一方面，线程共享父进程的地址空间，但是总体上比一个进程相关联的状态信息少。这让一个进程比一个线程更加显得重要。然而，进程和线程都是在父进程的地址空间内拥有独立控制权的线程，而且有其自己的生命周期。

进程和线程的一个基本区别是内存的保护。操作系统把每个程序变成一个进程，每个进程都有其自身的地址空间作为相互之间的墙。但是，线程在一个单独的地址空间内执行。因此，它们相互之间并无保护。用人类的例子来说就，就是我们不会走入邻居的房子里，然后在墙上乱涂乱画。然而，孩子（如果没有好好看管）会很高兴在自己家里的墙上用蜡笔进行涂鸦。他们也许会相互打架。我们马上将看到如何在线程之间加入纪律性来保证它们在同一个地址空间内保持纪律地执行。

一个线程自动地在其退出开始进入的顶层过程后终止。另外，库可能还会提供一种显式的调用来结束同一进程的线程：

```
thread_terminate(tid);
```

这里，tid 是系统提供的希望结束线程的标识符。

例 12-1 写出图 12-2 所示程序中初始化数字化部件和跟踪部件的代码段。

答：

525 见图 12-4a。

```
digitizer()
{
    /* code for grabbing images from camera
     * and share the images with the tracker
     */
}

tracker()
{
    /* code for getting images produced by the digitizer
     * and analyzing an image
     */
}

main()
{
     /* thread ids */
     thread_type digitizer_tid, tracker_tid;

     /* create digitizer thread */
     digitizer_tid = thread_create(digitizer, NULL);

     /* create tracker thread */
     tracker_tid = thread_create(tracker, NULL);

     /* rest of the code of main including
      * termination conditions of the program
      */
}
```

图 12-4 a) 创建线程的代码段

注意图形部分就是创建所需结构的代码。

12.2.2 线程之间的通信

线程可能需要共享数据。例如，图 12-2 中的数字化部件与跟踪部件需要共享其创建的帧缓冲区。

我们来看一看系统如何实现这种共享。恰好这并不麻烦。我们已经注意到，一个多线程
526 程序是通过将一个顺序程序中的顶层过程转化成线程来实现的。因此，在原始程序作用域内的多个线程（即顶层过程）都可见的数据结构就成为线程之间可以共享的数据结构。具体地，在一个像 C 这样的程序设计语言中，全局数据结构就是线程之间可以共享的数据结构。

当然，如果计算机是多处理器的，在实现共享时就会（在操作系统和硬件级）存在一些需要处理的系统问题。我们会在 12.9 节再来看这些问题。

例 12-2 给出图 12-2 中数字化部件和跟踪部件的数据结构定义，以便实现图像共享。

答：

见图 12-4b。

```
#define MAX 100                    /* maximum number of images */

 image_type frame_buf[MAX];        /* data structure for
                                    * sharing images between
                                    * digitizer and tracker
                                    */

digitizer()
{
  loop {
    /* code for putting images into frame_buf */
  }
}

tracker()
{
  loop {
    /* code for getting images from frame_buf
     * and analyzing them
     */
  }
}
```

图 12-4 b) 在数字化部件和跟踪部件之间共享的数据结构

注意：（图 12-4b 的）阴影部分是全局创建的数据结构，在数字化部件和跟踪部件之间进行共享 527

12.2.3 读 / 写冲突、竞争条件及不确定性

在一个顺序执行的程序中，我们从来不担心数据结构的完整性，因为程序中没有并发的活动。然而，当多个线程在一个地址空间里并发执行时，线程之间没有错误地相互影响就很有必要。我们定义如下的情况为读写冲突：多个并发的线程同时尝试访问一个共享的变量，且至少有一个线程尝试写该变量。而竞争条件发生在当一个程序存在读 / 写冲突，却又没有消除这种冲突的同步操作时。程序的竞争条件可能是有意的或者无意的。比如，如果一个共享变量用于进程间的同步，那么就是一个有意的竞争条件。

考虑下面的代码段：

```
情景 #1:
int flag = 0;  /* shared variable initialized to zero */

线程 1:                          线程 2:
while (flag == 0) {              .
   /* do nothing */              .
}                                .
                                 if (flag == 0) flag = 1;
                                 .
                                 .
```

线程 1 和线程 2 都是同一个进程的一部分。由于各自的定义，线程 1 和线程 2 存在读写冲突。在一次循环里线程 1 不断地读取共享变量 flag，而线程 2 在其执行过程中写该变量。表面上，这种设置可能会出现问题，因为两个线程之间存在竞争。然而，这是一个有意竞争（有时也称作同步竞争），线程 1 等待 flag 的值被线程 2 改写。因此，竞争条件并总是意味着代码有问题。

接下来，我们将定义会导致错误程序行为的特殊竞争条件。数据竞争是指没有同步操作进行协调的读写冲突，即线程中被访问的变量不是同步变量。也就是说，数据竞争发生在并行程序中对任意访问变量进行非同步访问的时候。

考虑下列代码段：

```
变量 #1:
int count = 0; /* shared variable initialized to zero */
线程 1(T1)              线程 2(T2)              线程 3(T3)
.                       .                       .
.                       .                       .
count = count+1;        count = count+1;        count = count-1;
.                       .                       .
.                       .                       .

线程 4(T4)
.
.
printf("count = %d\n", count);
.
.
```

在这 4 个线程之间就存在数据冲突（针对变量 count）。线程 4 会输出什么值？线程 1、2、3 都对当前的 count 值加 1。然而，每个线程看到的当前 count 值是多少呢？根据递增语句的执行顺序（count=count+1），线程 4 的 printf 语句将输出不同的结果。

图 12-5 展示了 4 种不同的执行顺序来说明这个问题。

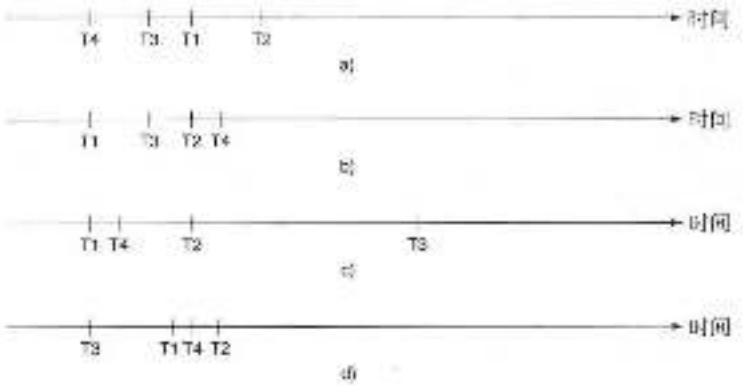


图 12-5 在一个单处理器上使用非抢占式线程调度的情况下，情景 #2 可能的执行情况示例。线程执行顺序的不可确定性会导致 T4 产生输出的不确定性

读者可能会想到的第一个问题是：为什么有这么多情景 2 中所示代码的可能执行方式？答案很简单。线程是并发的，与其他线程异步地执行。因此，一旦创建了这些线程，它们的执行顺序就由计算机中处理器的个数、线程之间的依赖关系以及操作系统使用的调度算法决定的。

图 12-5 所示的情景 #2 的执行时间轴假设使用单处理器来调度线程。且线程之间无依赖关系。也就是说，线程一旦创建了，线程就可能以任何顺序执行。而且，我们还假设调度是非抢占式的。下列的例子说明了这些线程的执行情况可能超过 4 种。

例 12-3 假设一个进程有 4 个线程，且线程在一个单处理器上调度。一旦创建了线程，每个线程就会给出其线程组，然后退出。假设使用非抢占式调度器，那么有多少种可能的执行结果？

答：

如果可知，线程之间是相互独立的。因此，操作系统可能以任何顺序调度它们。

4 个线程的可能执行个数 = 4!。

所以，并行程序的执行与在单处理器上一个程序的顺序执行有根本的不同。顺序程序的执行模型很简单：所有指令按照程序顺序依次执行[⊖]。我们定义程序顺序为对程序员来说程序指令的文本顺序与程序每次执行时指令的逻辑顺序的结合。逻辑顺序显然依赖于程序员期望程序去完成的语义。比如，如果你编写一个高级语言程序，源代码就提供你一个程序的文本顺序。根据输入数据和程序的实际逻辑（即条件语句、循环、过程调用等），程序的执行会按照源代码中的某条路径执行。换句话说，顺序程序的行为是确定性的，这意味着对于给定的输入，每次执行程序都会得到相同的输出结果。

528 ~ 530

理解由多个线程组成的并行程序的执行模型是很有帮助的。一个进程的每个独立的线程经历与顺序程序一样的执行过程。然而，无法保证同一进程的不同线程之间的执行顺序。即并行程序的行为是非确定性的。我们定义非确定性执行为：对于给定的输入，一次执行输出的结果可能与另一次有所不同。

我们回到情景 #2 来看一看图 12-5 中的这 4 个可能的非抢占式程序的执行结果。每种情况中，T4 可能输出的值会是什么呢？

首先来看图 12-5a。线程 T4 第一个完成执行。因此，它输出的 count 值是 0。在图 12-5b 中，线程 T4 最后执行。因此，T4 输出的值是 3（线程 T1、T2、T3 都将 count 值加 1）。图 12-5c 和图 12-5d 分别输出 1 和 2。

而且，如果调度器是抢占式的，程序的线程之间会有更多可能的交错情况。糟糕的是，语句

```
count = count+1
```

可能会编译为一串机器指令。例如，这条语句的编译会产生包含了从内存加载、累加、存储到内存的一系列指令。线程也许会在完成加载后，执行存储指令前被抢占。这对期望的程序行为和有数据竞争的实际程序执行产生严重的影响。

在第 4 章（4.3.4 节）中，我们介绍了原子操作的基本概念：不可分割的。我们在第 5 章中看到，在 ISA 中每条指令的执行都设计成原子的。处理器只有在完成一条指令的执行后才可以被外部中断打断以确保每条单独指令的原子性。在一个多线程程序中，指令的原子性不足以在出现数据竞争时保证程序行为达到预期目的。我们将在 12.2.6 节回顾这些问题，并论证在一个程序例子中，一组指令的原子性也是需要的。

图 12-6 说明由于并行程序模型中的非确定性，同一程序的各个线程的指令可能在实际运行于单处理器上时怎样被任意地交错。

图 12-6 有几点需要注意。同一线程的指令按照程序顺序执行（如 T1：I1，T2：I2，T3：I3，T4：I4，…）。然而，不同线程的指令可能被任意地交错在一起（见图 12-6b），而每个线程内部却是按顺序执行的。例如，如果你注意线程 2（T2）的指令，就会发现它们就是按照

⊖ 注意，我们在第 5 章（5.13.2 节）中提到，处理器的实现可能会选择重新安排指令的执行顺序；只要这种重新安排的顺序对程序员来说没有影响程序结果，那么就不会有问题。

531 T2 的程序顺序执行的。

线程 1 (T_1)	线程 2 (T_2)	线程 3 (T_3)
I_1	I_1	I_1
I_2	I_2	I_2
I_3	I_3	I_3
I_4	I_4	I_4
I_5	I_5	I_5
I_6	I_6	I_6
I_7	I_7	I_7
I_8	I_8	I_8
I_9	I_9	I_9

a) 一个具有 3 个线程的并行程序

$T_1:I_1$
$T_2:I_1$
$T_3:I_1$
$T_3:I_2$
$T_1:I_2$
$T_2:I_2$
$T_1:I_3$
$T_1:I_4$
$T_1:I_5$
$T_1:I_6$
$T_1:I_7$
$T_2:I_3$
$T_2:I_4$
$T_2:I_5$
$T_2:I_6$
$T_2:I_7$
$T_3:I_8$
$T_3:I_9$

b) 3 个线程指令交错执行

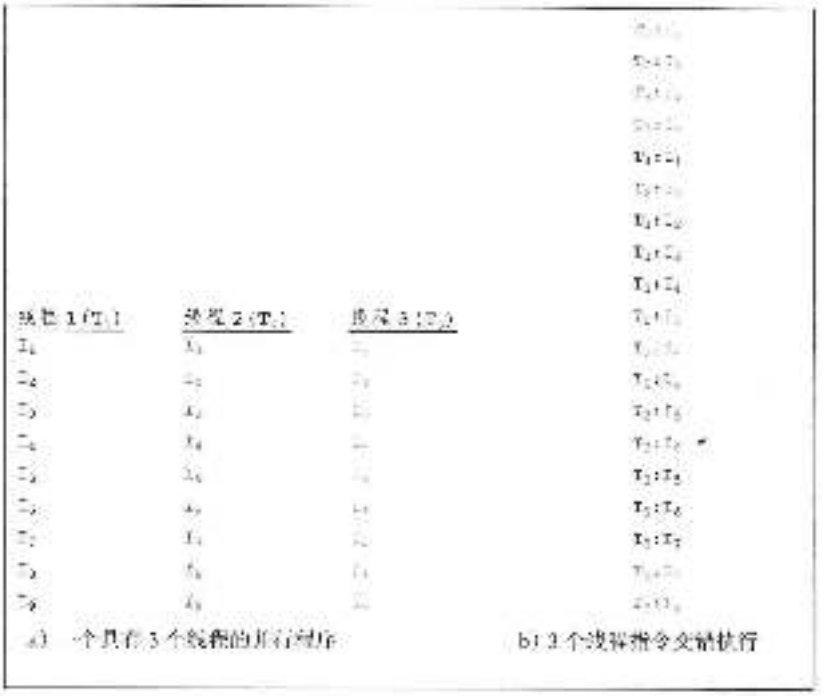


图 12-6 一个并行程序及其在单处理器上可能的执行顺序。3 个线程之间指令的交错取决于它们在单处理器上怎样调度运行

图 12-4 给定下列线程以及它们的执行顺序，共存单元 x 中的最终值是什么？假设每条指令的执行是原子的，且初始时 Mem[x]=0。

线程 1 (P1)	线程 2 (P2)
Time 0: R1 ← Mem[x]	Time 1: R2 ← Mem[x]
Time 2: R1 ← R1+2	Time 3: R2 ← R2+1
Time 4: Mem[x] ← R1	Time 5: Mem[x] ← R2

532

答：

线程 T1 和 T2 都加载到共存单元，加上一个值，并将其写回。由于数据竞争的出现以及可抢占式调度，不幸的是，x 的值将是最后一个有效操作后写入的值。

由于 T2 最后执行存储操作，所以 x 的最终值就是 1。

总之，非确定性是并行程序执行模型的核心。对于应用程序的开发者来说，理解这一点很重要，需要掌握这个概念来编写正确的并行程序。表 12-1 总结了顺序程序和并行程序的执行模型。

表 12-1 顺序程序和并行程序的执行模型

	执行模型
顺序程序	程序执行是确定性的，即指令按照程序顺序执行。只要不改变程序顺序，处理器的硬件实现可能会为了流水线执行的效率将一些指令的顺序打乱
并行程序	程序执行是非确定性的，即每个单独线程按照程序顺序执行。然而，同一个程序中的不同线程的指令可能是交错执行的

12.2.4 线程之间的同步

在已知并行程序的执行模型是不确定的情况下，程序员怎样才能让他的程序有一个确定性的行为呢？答案非常简单：通过线程之间的同步即可获得确定性的行为。我们将在下面详细讨论什么是线程之间的同步。特别地，我们将讨论两种同步方式：互斥与会合。

互斥　用一个类比来说明。我们在图 12-7 中看到一些孩子在玩耍。有一些活动他们可以在不妨碍他人的情况下单独且同时做（见图 12-7a）。然而，如果他们共享了某个玩具，我们就告诉他们需要轮流玩从而让每个孩子都有机会玩（图 12-7b）。



a）孩子们单独玩　　b）孩子们共享一个玩具[⊖]

图 12-7　孩子们在一个共享的“沙盒”里玩耍

类似地，如果有两个线程，一个是生产者，一个是消费者，当消费者在读取共享的缓冲区时生产者不修改缓冲区就很必要（见图 12-8）。我们将这种要求称作互斥。生产者和消费者同时运行，除了当其中的一个或者两者都要修改或检查共享的数据结构时。这种情况下，为了保证数据的完整性就有必要让它们顺序地执行。

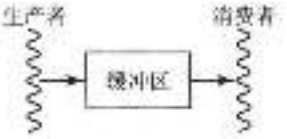


图 12-8　线程之间共享的缓冲区。生产者线程将东西放到缓冲区里，消费者从缓冲区中将其取出进行处理

库基于这个目的提供了互斥锁。锁是具有这里所展示语义的一种数据抽象。程序可以像任意其他类型定义的变量那样定义任意数量的这种锁。读者可以看到与现实生活中锁的相似之处。只有一个线程可以在一个时间持有一个特定的锁。一旦线程获取了一个锁，其他线程就不能再得到同一个锁，直到获得该锁的线程将其释放后才行。下列声明创建了一个锁类型的变量：

```
mutex_lock_type mylock;
```

下列调用允许一个线程获得 / 释放一个特定的锁：

```
thread_mutex_lock (mylock);
thread_mutex_unlock(mylock);
```

533 / 534

第一个函数成功返回则说明调用的线程成功地获得了该锁。如果另一个线程已经持有了这个锁，那么调用的线程就会阻塞，直到锁被释放且处于空闲状态。通常，我们把一个线程的阻塞态定义为某个线程无法继续执行，直到某个条件满足为止。第二个函数释放命名的锁。

有时，一个线程可能不希望被阻塞，而是能够选择在锁无法获得时干些其他事情。库也

⊖ 图片来源：http://www.liveandlearn.com/toys/tetherball.html。

提供了另一个种非阻塞式获取锁的调用：

```
(success, failure) ← thread_mutex_trylock (mylock);
```

这个调用会对上获取该锁的请求返回成功或者失败。

例 12-5 写出允许图 12-8 中生产者和消费者线程访问缓冲区的代码段，以一种互斥锁的方式，分别实现放入/取出一个项目的操作。

答：

```
item_type buffer;
mutex_lock_type buflock;
int producer()
{
  item_type item;

  /* code to produce item */
  . . . . . .

  thread_mutex_lock(buflock);
    buffer = item;
  thread_mutex_unlock(buflock);

  . . . . . .

  . . . . . .
}
```

```
int consumer()
{
  item_type item;

  . . . . . .

  thread_mutex_lock(buflock);
    item = buffer;
  thread_mutex_unlock(buflock);

  . . . . . .

  /* code to consume item */
}
```

注意：只有 buffer 和 buflock 是共享的数据结构。"item" 是每个线程中的局部变量。

在例 12-5 中，生产者和消费者在大部分时间内同时执行。当生产者或消费者分别执行阴影部分的代码时，另一个线程干什么呢？这个问题的答案取决于另一个线程这时执行到哪里。假设生产者在执行阴影部分代码。那么消费者的行为会有如下两种情况：

535

- 如果在阴影部分之外，那么消费者也在执行。
- 如果消费者试图进入阴影部分，那么它不得不等待直到生产者执行完阴影部分；类似地，如果消费者已经在阴影部分里了，那么生产者就将等待直到消费者执行完阴影部分为止。

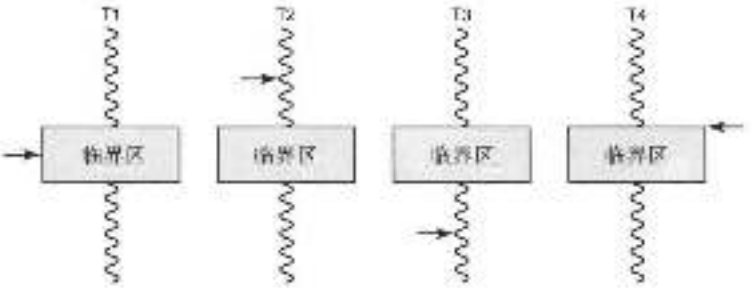
也就是说，生产者和消费者在各自的阴影部分里的执行是互斥的。这种以互斥方式执行的代码称为临界区。我们将临界区定义为一段程序，在其内部线程的执行是序列化的。即，一个时间只有一个线程能够执行在临界区内部的代码。如果多个线程同时到达临界区，那么它们之一可以成功地进入并执行临界区中的代码而其他线程只能在入口处等待。我们中的很多人都在一个忙碌的 ATM 机上经历过类似的情况，就是当我们不得不等到轮到我们进行提现或者存款时。

下面我们展示了更新一个共享的计数器代码作为临界区的例子：

```
mutex_lock_type lock;
int counter; /* shared counter */

int increment_counter()
{
        /* critical section for
         * updating a shared counter
         */
        thread_mutex_lock(lock);
            counter = counter + 1;
        thread_mutex_unlock(lock);
        return 0;
}
```

任意数量的线程可能同时调用 increment_counter。根据 thread_mutex_lock 的互斥性，只有一个线程可以在临界区内更新这个共享的计数器。

例 12-6 给出下图中线程（由箭头定义）的执行过程，指出哪些处于活动状态，哪些处于阻塞状态，并给出理由。假设临界区是互斥的（即，它们由一个锁管理）。T1 ～ T4 表示一个进程的线程。 536



答：

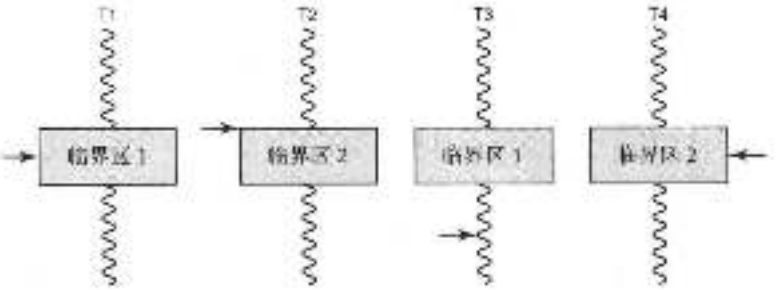
T1 处于活动状态，并在临界区内部执行；

T2 处于活动状态，并在临界区外部执行；

T3 处于活动状态，并在临界区外部执行；

T4 处于阻塞状态，并等待进入其临界区。（一旦被按 T1 释放，它就会进入临界区。）

例 12-7 给出下图中线程（由箭头定义）的执行过程，指出哪些处于活动状态，哪些处于阻塞状态，并给出理由。注意临界区 1 和临界区 2 是由不同的锁管理的。



答：

T1 处于活动状态，并在临界区 1 内部执行代码。

T2 处于阻塞状态，并等待进入临界区 2。（一旦被按 T4 释放，它就会进入临界区。） 537

T3 处于活动状态，并在临界区 1 外部执行代码。

T4 处于活动状态，并在临界区 2 内部执行代码。

会合 为了更好地理解第二种类型的同步方式，我们用另一种类比来说明。你和你的朋友决定去看电影。你先到了电影院。你等待你朋友来，这样你们就可以一起进入电影院了。你与你朋友进行了动作的同步。但这是一种不同的同步方式，我们称为会合。

与这种类比类似，一个线程可能需要等待同一进程中的另一个线程。这种机制的最常见用途就是父线程等待其创建的子线程。例如，主线程创建了一个子线程用于从磁盘读取文件，与此同时，它自己也并发地进行了相关活动。一旦主线程完成了自己的工作，它就不会继续执行，直到创建的子线程也完成了读操作。这是一个很好的主线程等待子线程终止的例子，

子线程终止说明文件读文件已经完成。

库通过函数调用方式提供了这样一种会合机制：

```
thread_join (peer_thread_id);
```

这个函数阻塞调用者直到相应 ID 的线程结束。那个线程结束后，调用线程才能继续执行。

更形式化地，我们把会合定义为同一程序的各个线程之间的集合点。会合要求至少有两个线程，不过可以包含给定程序的所有线程。参与会合的线程都到达会合点后就可继续执行。图 12-9 是线程之间会合的一个例子。

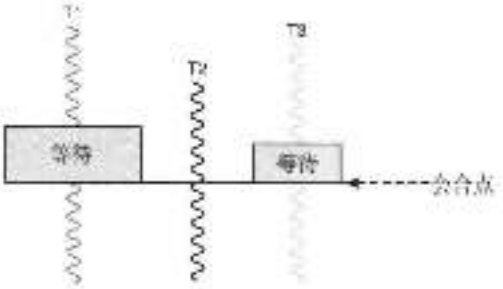


图 12-9 线程之间的会合。每个线程可能在不同的时间到达会合点，但是它们会一直等
538 直到所有需要到达的线程都到达会合点才继续执行

T1 第一个到达会合点，等着其他两个线程到达。T3 接着到达；最后，当 T2 到达后，会合完毕，3 个线程分别继续各自的执行。显然，会合是一种很方便的让并行程序中各个线程相互之间进行协调来防止不确定性的手段。会合机制的一种最常见的形式是障碍同步。该机制在科学计算的并行应用程序中尤其有用。需要参与会合的给定程序的所有线程执行障碍同步调用。一旦所有线程到达了障碍处，各个线程就可以继续各自的执行了。

thread_join 调用是适用会合机制的一个特例。它是一种单方面的会合。只有执行了这个调用的线程会等待（假设这时与之对等的线程还未结束执行）。对等线程完全不知道另一个线程已经在等待了。注意，这个调用让调用线程去等待另一个线程的结束。比如，如果主线程创建了一组子线程，直到它们都执行完毕才退出，那么它需要执行多个 thread_join 调用，一个接着一个，每个调用涉及其中一个子线程。在 12.2.8 节（见例 12-10），我们将看到两个线程之间通过条件变量实现的对称的会合方式。

还有一种现实生活中的例子，但在线程世界却不是那样。一个孩子在现实生活中通常活的比父母更久。然而在线程的环境中却不一定是这样的。尤其是，所有线程在同一个地址空间里执行，其结果是并不是所有的线程都有同样的状态。在父线程和子线程之间就有区别。在图 12-4a（见例 12-1）中，当进程初始化后，在地址空间里就只有一个称为“主线程”的线程。一旦“主线程”创建了“数字化线程”和“跟踪线程”，地址空间里就有 3 个活动的线程。当“主线程”退出后会发生什么？它是父线程，因此与进程自身是同义的。根据大多数操作系统实现的常见语义，当进程中的父线程结束后，整个进程也就结束了。然而，注意如果一个子线程又创建了两个子线程，那么它无法决定这两个子线程的生命周期；而是由与进程同义的主线程才能决定。这是 thread_join 调用比较方便的另一个原因，因为父线程可以在退出前等待子线程。

例 12-8 "主线程"创建了一个用户过程"foo"。请说明我们如何才能保证"主线程"没有过早地退出。 539

答：

```
int foo(int a)
{
  . . . . .
  return 0;
}

int main()
{
        int f;
        thread_type child_tid;
        . . . . .
        child_tid = thread_create (foo, &f);
        . . . . .
        thread_join(child_tid);
}
```

注意："主线程"通过执行 thread_join 等待线程 ID 为 child_tid 的子线程完成后才退出程序。

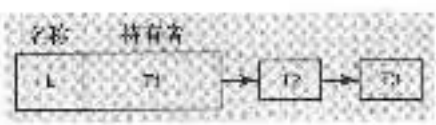
12.2.5 线程库中数据类型的内部表示

我们注意到一个线程在希望获取一个已经被其他线程占有的锁时会发生阻塞。我们了解这样的陈述是什么含义。现在我们已经清楚，与 C 语言这样的程序设计语言中的数据类型（例如，"int"和"float"）相比，线程库需要支持本章中介绍的那些数据类型。

thread_type 和 mutex_lock_type 是不透明数据类型，意味着用户无法直接访问这些数据类型的内部表示。在内部，线程库可能包含了与 thread_type 相关的一些信息。mutex_lock_type 非常有意思，也值得从程序员的角度更多地了解它。这种数据类型变量的内部表示至少有以下两样东西：

- 当前持有锁的线程（如果有的话）。
- 等待锁的等待请求队列（如果有的话）。

因此，如果我们有一个锁变量 L，当前，线程 T1 拥有该锁，还有两个线程 T2 和 T3 正在等待获取该锁，那么变量 L 在线程库中的内部表示可能看起来像下面这样： 540



当 T1 释放锁后，T2 获得该锁，因为 T2 是等待队列中的第一个线程。注意，每个锁变量有自己的等待队列。一个线程只可以在任何时刻位于一个等待队列中。

例 12-9 假设下列事件按所示顺序发生（T1 ～ T5 是同一个进程为线程）：

T1 执行 thread_mutex_lock(L1);

T2 执行 thread_mutex_lock(L1);

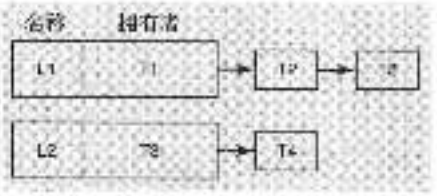
T3 执行 thread_mutex_lock(L2);

T4 执行 thread_mutex_lock(L2);

T5 执行 thread_mutex_lock(L1);

假设在此之前没有其他对线程序的调用，请给出在下列 5 个调用之后两个锁 L1 和 L2 的内部队列示意图。

答：



12.2.6 简单的编程示例

无同步的基本代码 首先，我们需要了解为什么需要同步。看一看下面的示例程序 #1，其包含了图 12-2 中数字化部件和跟踪部件之间的交互。这里，我们将慢慢地改进这个程序使其达到应用所需的语义。为了方便高级读者，样例程序 #5 给出了提供所需语义的程序（见 12.2.9 节）。

541

```
/*
 * Sample program #1:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf(MAX);

digitizer()
{
  image_type dig_image;
  int tail = 0;

  Loop { /* begin loop */

    if (bufavail > 0) {

      grab(dig_image);

      frame_buf[tail mod MAX] =
          dig_image;
      bufavail = bufavail - 1;

      tail = tail + 1;
    }
  } /* end loop */
}

tracker()
{
  image_type track_image;
  int head = 0;

  Loop { /* begin loop */

    if (bufavail < MAX) {

      track_image =
        frame_buf[head mod MAX];
      bufavail = bufavail + 1;

      head = head + 1;
      analyze(track_image);
    }
  } /* end loop */
}
```

在这个示例程序中，bufavail 和 frame_buf 是两个线程之间需要共享的数据结构。示例程序将 frame_buf 实现为一个循环队列，通过 head 和 tail 指针实现在尾部插入、头部删除（见图 12-10，阴影区域包含了 frame_buf 中的合法项目）。缓冲区的可用空间由 bufavail 变量表示。

head 和 tail 指针是自己线程内部的局部变量。数字化部件的代码不断地循环，从摄像头抓取一个图像，放入帧缓冲区中，将 tail 指针指向 frame_buf 中下一个空的位置。帧缓冲区中空间的可用性（bufavail>0）表明执行是在循环内部。类似地，跟踪部件的代码不断地循环，从帧缓冲区中获取一个图像（如果有的话），将 head 指针指向下一个 frame_buf 中的合法帧，并对帧进行相关分析。除了阴影部分之外两个线程其他都相互独立（见示程序 #1）。阴影部

分的代码对共享变量 frame_buf 和 bufavail 进行操作。

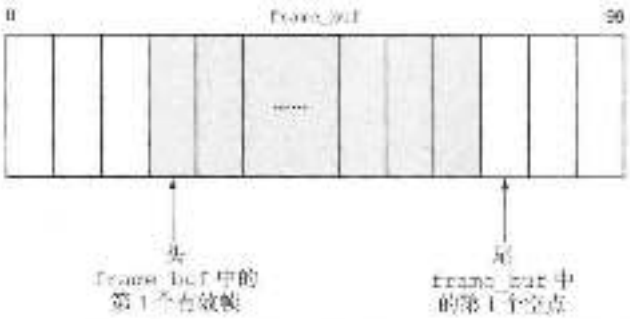


图 12-10 通过头和尾指针，像循环队列一样执行 frame_buf。当跟踪部还未处理时，阴影区域是新的项目

一组指令的原子性需求 前一段代码的问题在于数字化部件和跟踪部件的线程同时运行，当代码在不同处理器上运行时，就可能同时对共享的数据结构进行读和写。图 12-11 给出了这种情形，即两个线程同时试图修改 bufavail。

我们将更深地来看一看这种情况。语句

```
bufavail = bufavail - 1;                                    (1)
```

在处理器上作为一组指令执行（将 bufavail 加载到处理器的寄存器；执行递减操作；把寄存器的内容存回 bufavail）。

类似地，语句

```
bufavail = bufavail + 1;                                    (2)
```

在处理器上也作为一组指令执行（将 bufavail 载入处理器的寄存器；执行累加操作；把寄存器的内容存回 bufavail）。

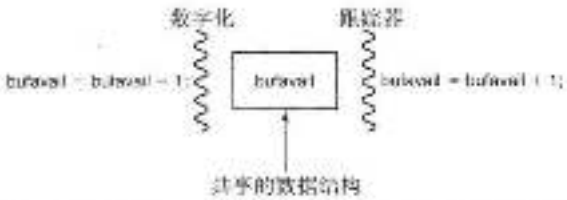


图 12-11 非同步访问共享数据产生的问题。数字化部件和跟踪器部件同时更新同一个内存单元（bufavail）

为了程序的正确执行，我们需要让指令（1）和指令（2）两条语句以原子的形式执行。即，要么指令（1）先执行，指令（2）后执行，或者反过来。指令（1）和指令（2）相互交错执行会导致程序出现错误和意外的行为。这是我们在 12.2.3 节提到的数据竞争。在 12.2.3 节中我们提到，即使是在一个单处理器上也可能出现这种指令交错的情形，这是由上下文切换引起的（见例 12-4）。处理器在指令集架构级保证了每条指令的原子性。系统软件（即操作系统）则需要保证一组指令的原子性。

因此，为了保证原子性，我们需要将对共享数据结构的访问封装到临界区中，这就保证

了执行的互斥。但是，我们一定要小心谨慎地选择什么时候以及怎样来使用同步。随便地滥用同步在保证原子性的同时可能会限制并发性，更严重地，会导致程序的不正确行为。

使用粗粒度临界区的代码改进 我们接下来将在同一个程序中使用同步结构 thread_mutex_lock 和 thread_mutex_unlock 来实现互斥的目的。

示例程序 #2 是图 12-2 中的另一个多线程样例程序。这个程序说明了互斥锁的使用方法。与示例程序 #1 的区别是，它在前影部分的代码加入了同步概念。在每个数字化和跟踪器部件线程的内部，lock 与 unlock 之间的代码就是每个线程访问共享数据结构要完成的工作。同步概念使得 lock 和 unlock 之间的零段代码的原子性得到了保证。程序可以得到“正确的”所需语义，但有严重的性能问题，我们接下来将详细阐述。

```
/*
 * Sample program #2:
 */
#define MAX 100
int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

digitizer()
{
  image_type dig_image;
  int tail = 0;
loop { /* begin loop */
  thread_mutex_lock(buflock);
  if (bufavail > 0) {
    grab(dig_image);
    frame_buf[tail mod MAX] =
      dig_image;
    tail = tail + 1;
    bufavail = bufavail - 1;
  }
  thread_mutex_unlock(buflock);
 } /* end loop */
}

tracker()
{
  image_type track_image;
  int head = 0;
loop { /* begin loop */
  thread_mutex_lock(buflock);
  if (bufavail < MAX) {
    track_image =
      frame_buf[head mod MAX];
    head = head + 1;
    bufavail = bufavail + 1;
    analyze(track_image);
  }
  thread_mutex_unlock(buflock);
 } /* end loop */
}
```

使用细粒度临界区的代码改进 仔细检查示例程序 #2 就会发现它没有同步问题，但是数字化和跟踪器线程之间没有并发性的执行部分。我们分析在该样例程序中需要什么样的互斥。对于数字化部件所需的抓取图像以及跟踪器对图像的分析没有互斥的必要。类似地，一旦线程通过检查 bufavail 确定了对 frame_buf 操作的合法性之后，插入或者删除项目也可以并发地进行。就是说，即使 frame_buf 是一个共享的数据结构，在程序中的使用中也只是序列化的访问。因此，我们将程序改成示例程序 #3 所示的样子来增加两个线程之间的并发性。我们把互斥限制为对 bufavail 的检查和修改上。不幸的是，这样的代码还是有严重的问题，我们将稍后做出解释。

```
/*
 * Sample program #3:
 */
#define MAX 100
int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
    grab(dig_image);
    thread_mutex_lock(buflock);
    while (bufavail == 0)
      do nothing;
    thread_mutex_unlock(buflock);
    frame_buf[tail mod MAX] =
      dig_image;
    tail = tail + 1;
    thread_mutex_lock(buflock);
    bufavail = bufavail - 1;
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop { /* begin loop */
    thread_mutex_lock(buflock);
    while (bufavail == MAX)
      do nothing;
    thread_mutex_unlock(buflock);
    track_image =
      frame_buf[head mod MAX];
    head = head + 1;
    thread_mutex_lock(buflock);
    bufavail = bufavail + 1;
    thread_mutex_unlock(buflock);
    analyze(track_image);
  } /* end loop */
}
```

12.2.7 死锁和活锁

我们来详细分析示例程序 #3 的问题。考虑数字化部件代码中的 while 语句。它通过检查 bufavail 来寻找 frame_buf 中的空闲位置。假设 frame_buf 是满的。此时，数字化部件会不断地执行 while 语句，等着 frame_buf 有新的空间释放出来。跟踪器需要通过移除 frame_buf 中的一个项目并累加 bufavail。然而，数字化部件持有 buflock，因此跟踪器会阻塞在获取 buflock 的地方。类似地，当 frame_buf 为空时也会有这种情况（跟踪器代码的中的 while 语句）。

我们刚才描述的问题称为死锁，是所有并行程序都不愿面对的棘手问题。死锁是指当一个线程正在等待一个永远不可能发生事件时的情形。例如，数字化部件在 while 循环中等待 bufavail 变为非零值，但这个事件却因为跟踪器无法获取锁而不会发生。之前描述的情形是死锁的一种特殊情形，通常称为活锁。涉及死锁的线程可以主动也可以被动地进行等待。活锁发生在当一个线程主动检查一个从不会发生的事件时。在这个例子中，我们看到，数字化线部件有 buflock 并期待 bufavail 变为非零值。这是活锁，因为它在浪费处理器的资源区等待一个不会发生的事件。另一方面，跟踪器等着数字化部件把锁释放。跟踪器的等待是被动的，因为它在到锁释放前会被操作系统阻塞。无论等待是主动的还是被动的，涉及死锁的线程就永远地被阻塞了。读者应该很明白死锁和活锁是并行程序执行中的另一个基本的非确定性特点。

542 - 546

我们回白前面代码中的 while 语句并不需要互斥，因为它们只是检查缓冲区的可用性。事实上，把 while 语句的互斥性移除就能消灭这里的死锁问题。示例程序 #4 采用了这个方法，其与示例程序 #3 的区别就是 while 语句周围关于锁的部分在两个线程中都删除了。

```
/*
 * Sample program #4:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {/* begin loop */
    grab(dig_image);
    while (bufavail == 0)
      do nothing;
    frame_buf[tail mod MAX] =
      dig_image;
    tail = tail + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail - 1;
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop {/* begin loop */
    while (bufavail == MAX)
      do nothing;
    track_image =
      frame_buf[head mod MAX];
    head = head + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail + 1;
    thread_mutex_unlock(buflock);
    analyze(track_image);
  } /* end loop */
}
```

547

这种解决方案是正确的，而且在两个线程之间具有并发性。然而，由于等待的原因这种方案的效率较低。我们把两个线程中的 while 等待称作忙等待。这种等待效率低下，因为处理器可以在这时给其他进程或线程做些更有用的事情。

12.2.8 条件变量

理想情况下，我们希望系统能够识别数字化线程正在等待的条件（bufavail > 0）没有得到满足，因此将其持有的锁释放，当该条件满足后再重新调度它。

这是另一种通常由库提供的数据抽象语义，称为条件变量。

下列声明创建了一个条件变量类型的变量：

```
cond_var_type but_not_empty;
```

库也提供了让线程通过条件变量等待和唤醒另一个线程的方式：

```
thread_cond_wait(but_not_empty, buflock);
thread_cond_signal(buf_not_empty);
```

第一个调用让线程（我们例子里的跟踪器）等待一个条件变量。等待一个条件变量相当于把调用的线程从调度列表中删除。这个调用中的第二个参数是一个互斥锁变量。在将调用线程从调度列表中删除之前，库隐含地先执行一个 unlock 的操作。第二个调用则把等待该条件变量的线程唤醒。唤醒，顾名思义，意味着等待的线程可以继续执行了。库了解与等待调用相关联的锁变量。因此，库会在调度等待的线程前先在该变量上调用一个 lock 操作。当然，如果没有等待的线程，对条件变量的唤醒操作就会被当做一个空（NOP）操作。如果有多个线程等待着同一个条件变量，那么库会选择其中一个（通常根据先来先服务原则）并发送唤醒信号。在通过等待和唤醒方式进行线程同步时要非常细心。图 12-2a 是一个正确使用的例子。而图 12-2b 是一个错误使用的例子，T1 在 T2 被唤醒后开始等待，导致了死锁。

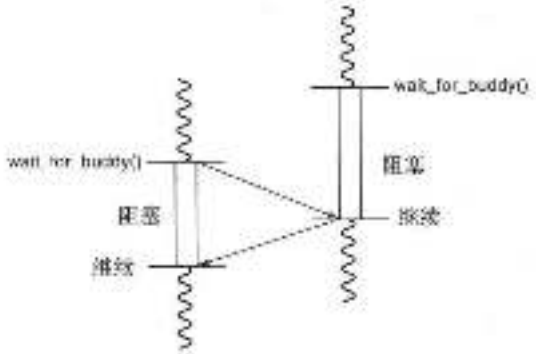
图 12-12b 说明了一个过早发送的信号会导致死锁的情况。例 12-10 说明如何设计一种会合机制，让两个线程无论谁先到达都能进行同步。



a) 在唤醒前等待　　b) 在唤醒后等待（T1 永远被阻塞）

图 12-12　使用条件变量的等待和唤醒："c" 为条件变量，"m" 为 cond_wait 调用中与 "c" 相关联的互斥锁。在 a 中，T1 收到了唤醒信号，因为它已经在等待了；而 b 中，信号丢失了，因为 T1 此时并未在等待

例 12-10　写出 wait_for_buddy() 的代码，使等待的 2 个线程能够相互会合。两个线程谁先到达都可以。注意这是一种通用的完成同一进程的组内线程之间会合的方式。



答：

解决方案使用了一个布尔变量（buddy_waiting）、一个互斥锁（mtx）和一个条件变量（cond）。基本思想如下：

- 先等到个线程先到达（下述代码的 "if" 部分），将 buddy_waiting 标识设为 true 并等待。
- 第二个到达的线程（下述代码的 "else" 部分）将 buddy_waiting 标识设为 false，唤醒第一个线程并等待。
- 第一个到达的线程从条件变量的等待中被唤醒并解锁，到此将第二个到达的线程唤醒，将互斥锁解开，然后离开该过程。
- 第二个到达的线程从条件变量的等待中被唤醒并解锁，将互斥锁解开，并离开该过程。
- 仔细观察 "if" 和 "else" 中等待和唤醒的顺序。如果不按照这种顺序，就会导致死锁。

548
)
549

```
boolean buddy_waiting = FALSE;
mutex_lock_type mtx; /* assume this has been initialized
                        properly */
cond_var_type cond;  /* assume this has been initialized
                        properly */
wait_for_buddy()
{
  /* both buddies execute the lock statement */
  thread_mutex_lock(mtx);

  if (buddy_waiting == FALSE) {
    /* first arriving thread executes this code block */
```

```
    buddy_waiting = TRUE;

    /* the following order is important */
    /* the first arriving thread will execute a wait statement */
    thread_cond_wait (cond, mtx);

    /* the first thread wakes up due to the signal from the second
     * thread, and immediately signals the second arriving thread
     */
    thread_cond_signal(cond);
  }
  else {
    /* second arriving thread executes this code block */
    buddy_waiting = FALSE;

    /* the following order is important */
    /* signal the first arriving thread and then execute a wait
     * statement awaiting a corresponding signal from the
     * first thread
     */
    thread_cond_signal (cond);
    thread_cond_wait (cond, mtx);
  }

  /* both buddies execute the unlock statement */
  thread_mutex_unlock (mtx);
}
```

550

条件变量数据类型的内部表示 从编程角度来理解线程库中的 cond_var_type 数据类型很直观。这种类型的变量，至少包含下列信息：

- 一个等待该变量唤醒信号的线程队列；
- 每个等待线程等待时关联的互斥锁。

调用 thread_cond_wait 的线程同时给出了一个互斥锁。线程库在将其放入等待队列前先解开这个锁。类似地，当收到该条件变量的唤醒信号后，线程从等待队列释放，线程库需要在将其重新执行该线程前重新获取这个锁。这就是线程库为什么要在等待队列中记住与每个线程相关联的锁。

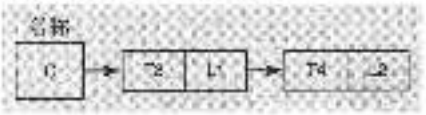
因此，假设两个线程 T3 和 T4 在条件变量 C 上执行条件等待调用时，令 T3 的调用是

```
thread_cond_wait(C, L1);
```

而 T4 的调用是

```
thread_cond_wait(C, L2);
```

C 在上述两个调用后的内部表示看起来就像这样：



注意一个给定条件变量的所有与等待操作有关的锁并不一定需要是同一个锁。

例12-11 假设事件按下述次序发生（T1 和 T2 是同一个线程的线程）：

T1 执行 thread_mutex_lock(L1);

551 T2 执行 thread_cond_wait(C1, L1);

T3 执行 thread_mutex_lock(L2);

T4 执行 thread_cond_wait(C2, L2);

T5 执行 thread_cond_wait(C1, L2);

a. 假设在此之前没有其他线程库的调用，请写出上述 5 个调用后线程库中为每队列的状态；

b. 现在，发生了下列事件：

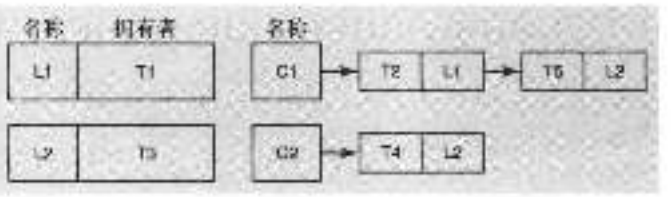
T6 执行 thread_cond_signal(C1);

T7 执行 thread_cond_signal(C2);

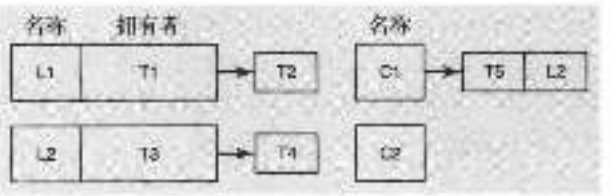
请写出上述两个调用后线程库中为每队列的状态。

答：

a.



b.



库在分别收到 C1 和 C2 为唤醒命令后，将 T2 移到 L1 的等待队列，将 T4 移到 L2 的等待队列。 552

12.2.9 视频处理示例的完整解决方案

我们现在回到图 12-2 的视频处理的例子。下面是一个使用了等待和唤醒语义的程序示例。注意每个线程在检查当前不为真的条件后进入等待，另一个线程使该条件变为真，最终保证一定没有死锁。注意每个线程在持有互斥锁时执行唤醒操作。然而，这并不是必需的，它不过是一个好的编程实践，并使并行程序中的错误更少。

```
/*
 * Sample program #5:  This solution delivers the expected
 *                     semantics for the video processing
 *                     pipeline shown in Figure 12.2, both
 *                     in terms of performance and
 *                     correctness for a single digitizer
 *                     feeding images to a single tracker.
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

cond_var_type buf_not_full;
cond_var_type buf_not_empty;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;
```

```
tracker()
{
  image_type track_image;
  int head = 0;
```

```
loop { /* begin loop */
  grab(dig_image);
(1) thread_mutex_lock(buflock);
    if (bufavail == 0)
      thread_cond_wait(buf_
        not_full, buflock);
  thread_mutex_unlock(buflock);
  frame_buf[tail mod MAX] =
    dig_image;
  tail = tail + 1;
(2) thread_mutex_lock(buflock);
    bufavail = bufavail + 1;
    thread_cond_signal
      (buf_not_empty);
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

```
loop { /* begin loop */
  thread_mutex_lock(buflock);
  if (bufavail == MAX)
    thread_cond_wait(buf_
      not_empty, buflock);
  thread_mutex_unlock(buflock);
(3) track_image = frame_buf
    [head mod MAX];
  head = head + 1;
(4) thread_mutex_lock(buflock);
  bufavail = bufavail + 1;
  thread_cond_signal
    (buf_not_full);
  thread_mutex_unlock(buflock);
  analyze(track_image);
} /* end loop */
}
```

这个程序示例中需要注意的关键是每个线程所维护的不变量。不变量是程序状态的某个无可争议的表示。在调用 thread cond wait 时，不变量表示调用者持有锁。库隐式地释放调用者的锁。当线程继续执行时，就需要重新建立不变量。在继续之前库会通过隐式地重新获取锁来重建代表阻塞线程的不变量。

12.2.10 解决方案的讨论

并发性 我们来分析示例程序 #5 中的解决方案，并证明其中并不缺乏并发性。

- 第一，注意代码段（1）和（3）持有锁只是为了检查 bufavail 的值。如果检查产生理想结果，那么释放锁并继续或者获一个图像或者取一个图像。如果检查产生不理想结果会怎么样呢？在这种情况下，代码段会在 buf not full 和 buf not empty 上执行条件等待。无论是哪个线程，两者都会立即释放关联的锁。
- 第二，注意代码段（2）和（4）持有锁只是为了更新 bufavail 变量，并发出唤醒信号来解锁另一个（如果正在等待的）线程。

给出以上两点，我们可以发现程序并不缺乏并发性，因为锁不会被某个线程在其他阶段获取。

553 ~ 554

测试题12 假设数字化部件在代码段（2）中忘记测试生产者程序#5中的 buf not empty 条件变量就发出信号。

说明下列答案是正确的还是错误的，并给出理由。

假设忘记在让生产者线程（2）发信号 buf_not_empty 的条件变量。

答：

错误的。虽然信号可以处于等待状态，但不是一直在等待。生产者程序（2）中的发信操作是有条件的。因此，我们无法了解 bufavail 的值是多少。假设忘记代码段（2）中会使图像的双方不相等 bufavail = MAX 时。我们知道它非常重要，因为缓冲区的大小是固定的，但是我们不知道它等于 MAX。

没有死锁 接下来，我们说明解决方案是正确的且不会产生死锁。首先，我们将形式化表示为：在任何时间点，两个线程不会都被阻塞，以不会产生死锁。

- 假设数字化部件在代码段（1）中等待唤醒信号。已知这一点，我们将说明跟踪器也不会阻塞从而导致死锁。由于数字化部件被阻塞了，所以我们知道下列为真的事实：
  - bufavail=0。
  - 数字化部件被阻塞了，等待 buf_not_full 的唤醒信号。
  - 为了数字化部件 buflock 被线程库隐式地释放了。

有 3 个可能导致跟踪器阻塞的地方：

- 代码段（3）的入口：由于数字化部件没有获取 buflock，所以跟踪器不会在入口处被阻塞。
- 代码段（4）的入口：由于数字化部件没有获取 buflock，所以跟踪器不会在入口处被阻塞。
- 代码段（3）中的条件等待语句：数字化部件被阻塞了，在代码段（1）内部等待唤醒信号。因此，bufavail = 0。所以，代码段（3）中的“if”语句将返回期望的结果，跟踪器不保证不会阻塞。

- 与前者的参数类似，我们可以证明当跟踪器在代码段（3）中等待唤醒信号时，数字化部件也不会阻塞从而导致死锁。

其次，我们说明如果一个线程被阻塞了，它最终一定会被另一个线程解除阻塞。

- 假设数字化部件被阻塞了，在代码段（1）中等待唤醒信号。如我们之前所说明的，跟踪器可以无阻塞地执行其代码。因此，最终它会进入代码段（4）中的唤醒语句。收到 555
信号后，数字化部件等待着重新获得（当前由跟踪器在代码段（4）中持有的）锁。注意线程库会为数字化部件隐式地获取该锁。跟踪器离开了代码段（4），释放了锁；而数字化部件重新获得锁并移出代码段（1），自己释放了锁。
- 用前面类似的方式，我们也可以证明当跟踪器在代码段（3）中等待唤醒信号时，数字化部件也会发送唤醒信号来解锁跟踪器。

因此，我们形式化证明了这个解决方案的正确性，并且不会缺乏并发性。

12.2.11 重新检查条件

示例程序 #5 对这个拥有一个数字化部件和一个跟踪器的例子能够正确地工作。然而，总的来说，条件变量的编程需要更加谨慎来避免同步错误。考虑下面使用共享资源的代码段。任意数量的线程可以执行过程 use_shared_resource。

```
/*
 * Sample program #6:
 */
enum state_t {BUSY, NOT_BUSY} res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

/* helper procedure for acquiring the resource */
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);                    T3 is here
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex);   T2 is here
    res_state = BUSY;
```

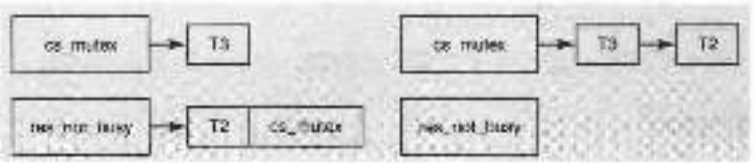
```
  thread_mutex_unlock(cs_mutex);
}

/* helper procedure for releasing the resource */
release_shared_resource()
{
  thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;                          T1 is here
    thread_cond_signal(res_not_busy);
  thread_mutex_unlock(cs_mutex);
}

/* top level procedure called by all the threads */
use_shared_resource()
{
  acquire_shared_resource();
    resource_specific_function();
  release_shared_resource();
}
```

可以看到：

- T1 刚刚使用完资源并且将 res_state 设置为 NOT_BUSY。
- T2 处于条件等待状态。
- T3 等待获取 cs_mutex。

图 12-13 是 cs_mutex 和 res_not_busy 的库中等待队列的状态：

- T2 在 res_not_busy 的队列中，而 T3 在 cs_mutex 的队列中（见图 12-13a）。
- T1 在条件变量 res_not_busy 上发出了唤醒信号，导致 T2 移动到 cs_mutex 队列中，因为库需要在继续 T2 执行前重新获取 cs_mutex（见图 12-13b）。



a) 在T1发出唤醒信号前的等待队列　　b) 在T1发出唤醒信号后的等待队列

图 12-13　等待队列的状态。一旦 T1 发出唤醒信号，T2 就离开条件等待状态。但是，cs_mutex 需要被 T2 重新获取来满足 12.2.9 节中的不变量。基于这个原因，T2 进入了 b 中 cs_mutex 的等待队列

556 / 557

当 T1 释放了 cs_mutex 后，会发生下列事件：

- 将锁给予 T3，即 cs_mutex 等待队列中的第一个线程。
- T3 发现 res_state 为 NOT_BUSY，释放 cs_mutex，并继续使用资源。
- T2 从 thread_cond_wait 继续执行（因为 cs_mutex 现在可用了），释放 cs_mutex，也继续使用资源。

现在，我们违反了使用共享资源的互斥条件。我们看看是如何导致这种情况的。T1 激活了 T2 在发出信号之前等待的条件，但是 T3 在 T2 继续执行前将其无效化了。因此，重新检查继续执行的条件（即程序需要满足的条件）是一种避免这种同步错误的编码手段。

下面的程序段通过改变与 thread_cond_wait 关联的语句从 if 变为 while 来解决前面的问

题。这种方法保证一个线程在继续执行时重新检查条件从而在 thread_cond_wait 被调用时又继续保持阻塞状态。

```
/*
 * Sample program #7:
 */
enum state_t (BUSY, NOT_BUSY) res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

acquire_shared_resource()
{
  thread_mutex_lock(cs_mutex);                    T3 is here
  while (res_state == BUSY)
    thread_cond_wait (res_not_busy, cs_mutex);    T2 is here
  res_state = BUSY;
  thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
  thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;                         T1 is here
    thread_cond_signal(res_not_busy);
  thread_mutex_unlock(cs_mutex);
}
use_shared_resource()
{
  acquire_shared_resource();
    resource_specific_function();
  release_shared_resource();
}
```

例 12-13 重写示例程序的要允许多个数字化线程和跟踪线程可以一起工作。本例留作读者的一个练习。

[提示：在线程从条件等待中被唤醒后重新检查条件十分重要。可且，目前数字化线程的实例共享了头指针，而跟踪线程的实例共享了尾指针。因此，对这些指针的修改需要在每一类的实例之间进行互斥。为了保证线程之间的并发性并减少不必要的竞争，请对头指针和尾指针使用不同的锁来实现互斥。]

12.3 线程函数调用和多线程编程概念总结

我们总结在前几节中介绍的一些多线程编程所需要的基本函数调用。注意这只是基本调用的一个说明性集合，并不意味着是全面的。在 12.6 节，我们给出 IEEE POSIX ⊖标准线程库提供的一个全面的函数调用集。

- thread_create (top-level procedure, args);

创建一个从 top-level procedure（顶层过程）开始执行的新线程，以 args 为过程所需的参数。

- thread_terminate (tid);

结束线程 ID 为 tid 的线程。

- thread_mutex_lock (mylock);

当线程返回时它拥有了锁 mylock。如果该锁当前已被其他线程占用，那么将阻塞调用的

⊖ IEEE 是一个国际性组织，代表 Institute of Electrical and Electronics Engineers, Inc.；POSIX 代表便携式操作系统接口（POSIX®）。

线程。

- `thread_mutex_trylock (mylock);`

558 ~ 559

该调用不会阻塞调用的线程。如果线程获得了 mylock，则返回 success；如果锁正被其他线程使用，则返回 failure。

- `thread_mutex_unlock(mylock);`

如果调用的线程目前已经有 mylock 了，则将其释放；否则，返回错误。

- `thread_join (peer_thread_tid);`

直到 ID 为 peer_thread_id 的线程结束，调用的线程才能继续执行。

- `thread_cond_wait(buf_not_empty, buflock);`

调用线程阻塞在条件变量 buf_not_empty 上；原隐式地释放锁 buflock。如果锁当前不被调用的线程拥有，则返回错误。

- `thread_cond_signal(buf_not_empty);`

一个（如果有的话）等待条件变量 buf_not_empty 的线程被唤醒。如果与被唤醒的线程相关联的（即 wait 调用中的）锁当前可用，则该线程准备继续执行；否则，线程从条件变量的队列移动到相应锁的队列中。

为了提供快速参考，表 12-2 总结了我们在多线程编程中介绍过的一些重要概念。

表 12-2 与线程有关的概念总结

概　念	定义及使用方法
顶层过程	并行程序的一个线程的开始执行位置
程序顺序	顺序程序的执行模型。它结合了程序的文本顺序和程序员定义的程序逻辑（条件语句、循环、过程等）
并行程序的执行模型	并行程序的执行模型在每个线程中保持程序的顺序，但是允许不同线程的指令之间相互交错
确定性执行	给定的程序，对于一组给定的输入，每次运行的输出结果相同。顺序程序表示的执行模型具有这个属性
不确定性执行	对于同一个程序的一组相同的输入，每次运行可能产生不同的输出结果。并行程序的执行模型具有这个属性
数据竞争	同一个程序的多个线程在没有同步的情况下同时访问同一个共享的变量，其中至少一个访问是对变量的写操作
互斥	同一个程序的各个线程顺序（即非并发地）执行的一种手段。当需要在并行程序中避免数据冲突时采用
临界区	程序中的一个区域，在其中线程的活动都是按顺序的。用于保证互斥性
阻塞	线程的某种状态，此时该线程在一个队列中等待，直到某个条件发生才可以继续变为可运行状态
忙等待	线程的某种状态，此时该线程在继续的下执行前不断地检查某个条件是否被满足
死锁	同一个程序的一个或多个线程被阻塞了，等待某个永远不可能被满足的条件
活锁	同一个程序的一个或多个线程处于繁忙状态，等待一个永远不可能被满足的条件
会合	一个并行程序的多个线程使用这种机制来协调它们的活动。最通用的会合是障碍同步。一个会合的例子是 thread_join 调用

12.4 线程编程的一些注意事项

以下是线程编程时需要牢记的几个重点：

1）在设计数据结构时尽可能采用能够加强线程并发性的方式。

2）最小化互斥时需要锁定数据结构的粒度以及持有锁所需要的时间。

3）避免忙等待，因为这很浪费处理器资源。

4）对程序中的每个临界区，都要仔细了解其中的不变量是真，以便确保在临界区中不变量的状态被保护。

5）让临界区的代码尽可能简单，使得手动检查是否有死锁或活锁变得更方便。

12.5 使用线程作为软件结构抽象

图 12-14 是将线程用作系统软件结构抽象的一些模型。

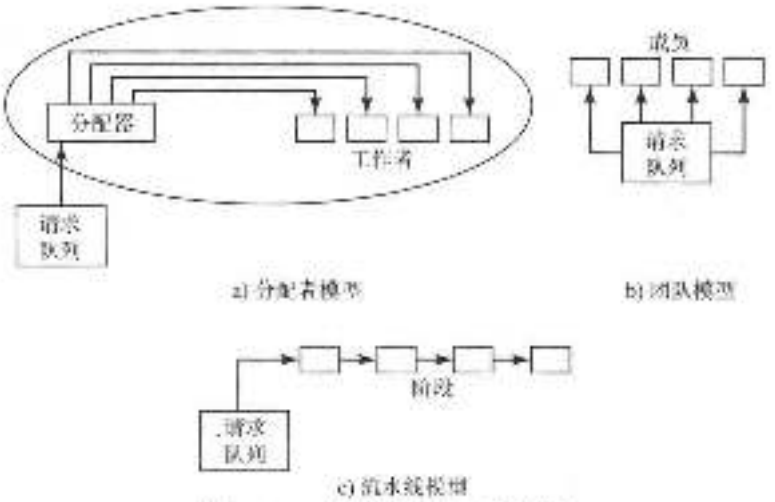


图 12-14 使用线程的架构服务器

诸如文件服务器、邮件服务器和 Web 服务器这样的软件实体，通常在多处理器上执行。图 12-14a 给出了这些服务器的一个分配器模型。当分配器线程收到请求时将其分配给工作者线程池中的某个线程。请求完成后，工作者线程回到空闲池中。当请求数量突破服务器能够处理的容量时，通过请求队列将并发的工作量稳定下来。调度者也提供工作负载管理器的功能，通过增大或减小工作者线程的数量来满足需要。图 12-14b 是团队模型，团队中的每个成员直接访问请求队列获取任务。图 12-14c 是流水线模型，更适合用于处理类似本章之前讨论的视频监视的连续工作的应用程序。流水线的每个阶段处理一个特定的任务（例如，数字化部件、跟踪器等）。

客户端程序也可以得益于多线程。线程增加了客户端程序的模块性和简洁性。例如，客户端程序可以使用线程来处理异常、信号，以及终端的输入/输出。

12.6 POSIX pthread 库调用总结

IEEE 通过 POSIX pthread 库对线程的应用程序接口（API）进行了标准化。每种 UNIX 操作系统都实现了这个标准。这种标准化方便了程序的移植。Microsoft Windows 并没有在其线

程库中使用 POSIX 标准[⊖]。下面总结了一些最常用的 pthread 库调用和它们简单的作用描述。若想了解更多信息，可以查看相关的文档源（例如，每类 UNIX 系统上的 man 页[⊜]）。

560 / 562

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutex_attr_t *mutexattr);
```

参数

mutex：要初始化的互斥锁变量的地址。

mutexattr：用于初始化互斥锁的属性变量的地址。查看 pthread_mutexattr_init 以了解更多信息。

语义：每个互斥锁变量必须必须被声明（pthread_mutex_t）并被初始化。

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

参数

cond：需要初始化的条件变量的地址。

cond_attr：用于初始化条件变量的属性变量的地址。未在 Linux 中使用。

语义：每个条件变量必须被声明（pthread_cond_t）并被初始化。

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

参数

thread：线程标示符（tid）的地址。

attr：应用到新线程的属性的地址。

start_routine：新线程开始执行的函数。

arg：传入 start_route 的第一个参数的地址。

语义：函数将创建一个新线程，建立执行地址（通过函数名称传递），并将参数在线程开始执行时传递进去。将新创建线程的线程 ID（tid）放到指向的位置。

```
int pthread_kill(pthread_t thread,
                 int signo);
```

参数

thread：发送唤醒信号的线程 ID。

signo：发送给线程的信号数。

语义：用于向某个 tid 已知的线程发送唤醒信号。

```
int pthread_join(pthread_t th,
                 void **thread_return);⊕
```

563

参数

⊖ 尽管 Microsoft 没有直接支持 POSIX 标准，但 WIN32 平台上用 C 开发多线程程序时线程库中，大部分 POSIX 的标准线程调用都有对应语义的函数调用接口。

⊜ 例如，可以浏览 http://linux.die.net/man/。

⊕ pthread 库也支持通用的在并行科学计算程序中极其实用的障碍同步。感兴趣的读者可以查看 UNIX 的参考版 http://linux.die.net/man/。

th：等待的线程的 tid。

thread_return：如果 thread_return 不为 NULL，则 th 的返回值存储在 thread_return 指向的位置。th 的返回值要么是提供给 pthread_exit(3) 的参数，要么是当 th 被取消后的 PTHREAD_CANCELED。

语义：pthread_join 将调用线程的执行暂时挂起，直到标示符为 th 的线程退出（调用 pthread_exit(3) 或者被取消）。

```
pthread_t pthread_self(void);
```

参数：无

语义：pthread_self 返回调用线程的线程标识符。

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

参数

mutex：需要锁定的互斥锁变量的地址。

语义：等待直到给定的互斥被被锁定，然后将其锁定后返回。

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

参数

mutex：需要解锁的互斥锁变量的地址。

语义：如果调用者是将互斥被锁定来的线程，那么将开放该锁。

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

参数

cond：等待的条件变量的地址。

mutex：与 cond 关联的互斥锁变量的地址。

语义：pthread_cond_wait 以原子化的方式锁住互斥锁 mutex，并唤醒等待条件变量 cond。线程执行被挂起，因此不会消耗 CPU 时间，直到条件变量被唤醒。在 pthread_cond_wait 的入口处，mutex 必须被调用的线程锁定。在回到调用线程前，pthread_cond_wait 需要重新获取 mutex。 564

```
int pthread_cond_signal(pthread_cond_t *cond);
```

参数

cond：条件变量的地址。

语义：将指定的条件变量的某个等待线程唤醒。

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

参数

cond：条件变量的地址。

语义：前一个调用的变种，即将指定的条件变量的所有等待线程唤醒。

```
void pthread_exit(void *retval);
```

参数

retval：线程返回值的地址。

语义：终止调用线程的执行。

12.7 操作系统对线程的支持

在 MS-DOS 这种 PC 上早期极其简单的操作系统中，用户程序和系统内核之间没有任何分离（见图 12-15）。因此，用户程序和内核之间的线是假想的。所以，在用户和线程空间之间进行切换的代价（在时间方面，等价于一个过程调用）非常小。这种结构的缺点是在用户程序之间没有内存保护，因此一个错误或者恶意程序可以轻易地破坏内核的内存空间。

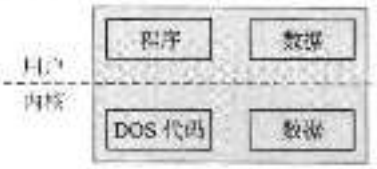


图 12-15 MS-DOS 的用户与内核边界。MS-DOS 是 PC 上的一个早期操作系统。阴影表示在用户和内核之间并无强制的分隔

现代操作系统，例如 MS Windows XP、Linux、Mac OS X，以及 UNIX 通过虚拟内存机制提供真正的内存保护，我们在之前的章节讨论过相关话题。图 12-16 是这些操作系统中用户进程和内存的内存空间示意图。

565

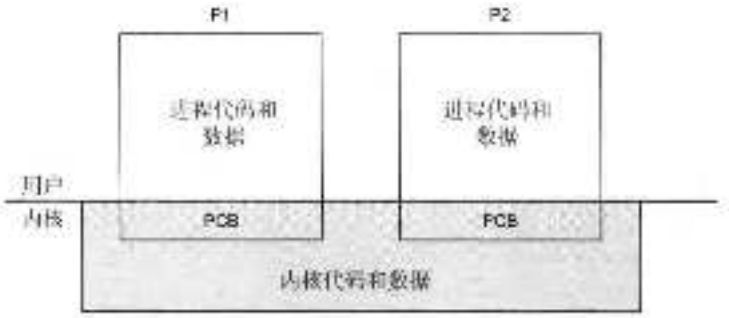


图 12-16 传统操作系统中的内存保护。在用户和内核之间有强制的分隔

每个进程在其自己的地址空间里。在用户空间和内核空间之间有一条清晰的分割线。系统调用会导致保护域的切换。既然我们对存储器的层次体系已经非常熟悉了，我们可以发现工作集（其影响了所有的存储器层次，从虚拟存储到处理器 cache）在每个这样的地址空间切换时发生改变。因此，频繁地在边界上切换会降低性能。一个进程控制块（PCB）定义了一个特定的进程。在前面的章节中，我们看到多个操作系统的组件（例如调度器、存储系统、I/O 子系统等）如何使用 PCB。在传统的操作系统中（即，不是多线程的），进程是单线程的。因此，PCB 包含的信息完全说明了这个单线程在处理器上的活动（当前的 PC 值、栈指针值、通用寄存器等）。如果一个进程发出了一个阻塞了当前进程（例如从磁盘读取一个文件）的系统调用，那么整个程序就不会再往下执行了。

大多数现代的操作系统（Windows XP，Sun Solaris，HP Tru64 等）都是多线程的。也就是说，操作系统将一个运行程序的状态识别为所有构成该程序的各个线程状态的组合。图 12-17 展示了现代操作系统的内存空间分布，它既能支持单线程程序也能支持多线程程序。一个给定进程的所有线程共享该进程的地址空间。

一个给定进程可能有多个线程，但是由于所有的线程共享同一个地址空间，所以它们在内存中共享同一个页表。下面详细阐述一个线程的计算状态。线程控制块（TCB）包含了一个线程相关的所有状态信息。然而，包含于 TCB 的信息与 PCB 相比是极小的。特别地，TCB 包含了 PC 值、栈指针值以及通用寄存器值。

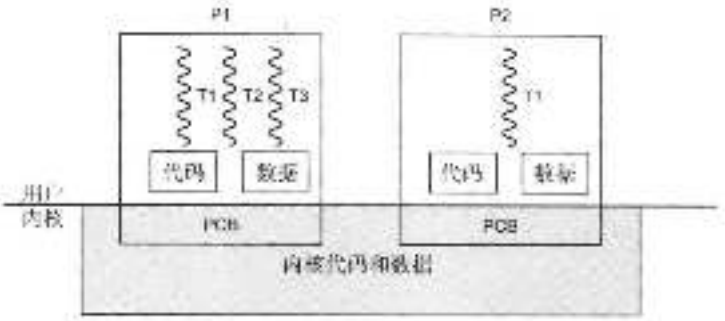


图 12-17 现代操作系统中的内存保护。一个进程可能有多个线程。同一个进程中的所有线程共享进程的地址空间

将多线程进程和单线程进程的内存布局进行比较就会发现一些有趣的地方（见图 12-18）。我们之前提到，一个给定进程的所有线程共享代码、全局数据、堆的空间，因此栈是内存中仅有的对特定线程而言是特有的部分。由于多线程进程栈的布局与仙人掌（见图 12-18c）在视觉上是相似的，所以我们将这种栈称作仙人掌栈。



a) 单线程进程 b) 多线程进程 c) 仙人掌

图 12-18 单线程进程和多线程进程的内存布局。多线程进程的每个线程有自己的栈

下面，我们来看一看实现一个线程库要做的事情——先看用户级，然后看内核级。

12.7.1 用户级线程

566 - 567

首先，我们考虑完全处于用户级的实现。也就是说，操作系统只知道进程的存在（即单线程的）。然而，我们依然可以在用户级实现线程。换言之，线程库作为操作系统上的一种功能存在，就像你拥有可以给任意程序使用的数学库那样。库提供了之前讨论过的线程创建调用，并支持 mutex 和 cond_var 等的数据类型及其相关操作。程序希望向线程库中的库链接发出调用，然后如图 12-19 所示的那样变成了程序的一部分。

操作系统维护传统的队列，即可调度的线程，这是在操作系统级的调度单元。线程库维护每个进程的一个准备运行的线程列表，通过包含了对应于每个线程信息的线程控制块（TCB）。TCB 包含了每个线程的最小信息（PC 值、SP 值，以及通用寄存器值）。用户级进程可以是单线程的（例如 P3），也可以是多线程的（P1、P2）。

读者可能会好奇，如果进程是操作系统的调度单元，那么用户级线程的功能会怎样。即使底下的平台是多处理器的，给定进程中的线程也无法并发地执行。我们记得：线程是构建软件的一种结构化机制。这是提供用户级线程的主要原因。它们以协程的形式运行，即当线程执行了线程同步调用并将其阻塞后，线程调度器就会选择同一个进程的其他线程运行。类

568

作系统并不知道线程调度器通过使用 TCB 进行的这种线程级的上下文切换。在用户级切换线程的代价较低，因为切换并不涉及操作系统。上下文切换的代价近似等于在程序中执行一个过程调用。因此，用户级线程提供了一种结构化的机制，而无需使用代价高昂的涉及操作系统的上下文切换。用户级线程进行上下文切换付出的直接和间接代价是最小的。（请参见 9.22 节有关直接和间接代价的一些细节。）而且，线程级调度器可以针对特定应用程序进行个性化设计。

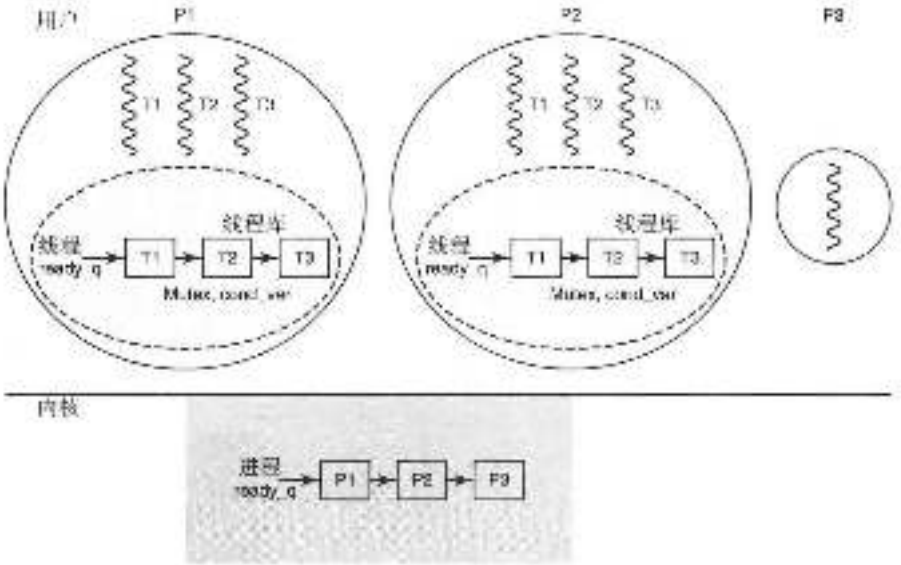


图 12-19 用户级线程。线程是应用程序进程地址空间中的一部分。在这种意义上，在应用程序逻辑和线程库功能之间并无强制的内存保护。另一方面，在用户和内核之间并无强制的分离

当多线程进程中的某个线程执行了一个阻塞系统调用时会发生什么呢？这种情况下，操作系统阻塞了整个进程，因为它不了解同一个进程的其他线程也可以运行了。这是一个用户级线程的基本问题。解决这个问题有几种不同的方法：

1）一种可能的方法是封装所有的操作系统调用（例如，fopen 变为 thread_fopen）来强制所有调用都通过线程库进行。然后，当某个进程（例如，图 12-19 中 P1 的 T1）执行这样的调用时，线程库意识到执行该系统调用会阻塞整个进程。因此，它会将该调用推迟到所有这个进程的线程都无法继续运行时再将这个阻塞的调用交给操作系统执行。

2）第二种方法是通过操作系统的向上调用机制（见图 12-20）警告线程调度器：该进程的一个线程将要执行一个阻塞的系统调用。这个警告让（库中的）线程调度器得以执行线程切换或者推迟该线程的阻塞调用到一个稍后的时间。当然，为了使操作系统支持这种向上调用方法需要对操作系统进行扩展。

在第 6 章，我们探究了不同的 CPU 调度策略。在这种背景下，我们看看线程调度器如何在用户级线程之间进行切换。显然，当一个线程执行一个线程库的同步调用时，而这时线程调度器就可以用来切换线程。类似地，线程库可能提供一个 thread_yield 调用让线程可以自愿放弃使用处理器并将执行的机会交给同一个进程的其他线程。我们在第 6 章中研究的一种调度方式是可抢占式调度。如果要在用户级实现可抢占式的线程调度器，线程调度器可以从线程请求一个时间中断，并以其作为执行线程之间可抢占式线程调度器的触发器。

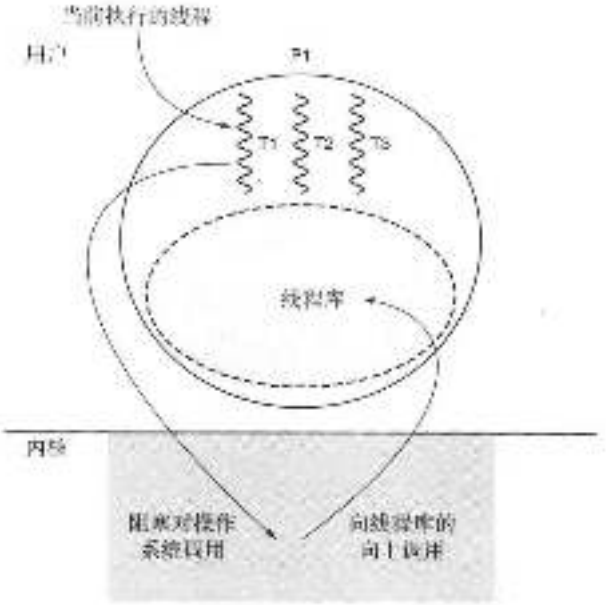


图 12-20 向上调用机制。线程库注册内核中的处理程序，阻塞对操作系统的调用。操作系统向处理程序发送一个向上调用，并向线程库告警由那个进程的线程发出的阻塞系统调用

12.7.2 内核级线程

我们知道，为了在内核级实现线程需要操作系统的相关支持：

1）一个进程的所有线程在同一个地址空间中生存。因此，操作系统需要保证这些线程共享进程的同一个页表。

2）每个线程需要有自己的栈，但也共享其余的内存信息。

3）操作系统需要支持之前提到的线程级同步。

569 i 570

首先，我们考虑对进程级调度器进行简单的扩展来支持内核中的线程。操作系统可能会实现一个如图 12-21 所示的二层调度器。进程级调度器管理进程中的由所有线程共享的 PCB（页表、统计信息等）。线程级调度器管理 TCB。进程和调度器分配进程的时间片，并在进程之间使用可抢占式调度。在一个时间片内，线程级调度器用循环方式或者协同模式调度该进程中的线程。对于后一种情形，线程自觉地放弃处理器，让同一个进程的其他线程可以被线程级调度器调度执行。由于操作系统知道线程的存在，所以在当前执行线程发出阻塞的系统调用（如进行 I/O 或线程同步）时，可以切换到其他线程来执行。

目前的计算机和芯片都是多处理器，如果一个进程的线程不能利用硬件并发性就会有极大的性能限制。图 12-21 中的结构让一个给定进程的线程能够重叠 I/O 和处理——这是在用户级线程上的一个进步。为了完全利用多处理器中的硬件并发性，线程需要成为操作系统的一个调度单元。接下来，我们讨论 Sun Solaris 线程作为内核级线程的一个具体例子。

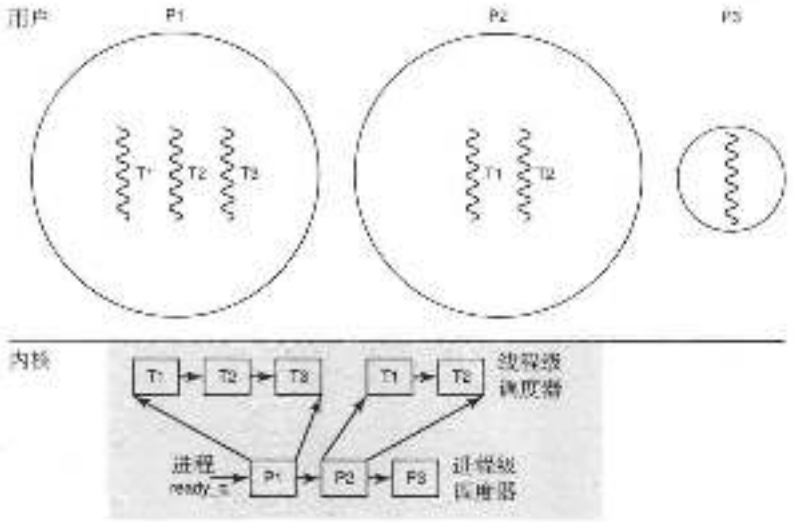


图 12-21 内核级线程。进程级调度器使用进程的 ready_q 队列，即使当前调度进程的一个线程运行了一个很长的系统例程，操作系统也可以通过线程级调度器从当前进程中选择一个准备好的线程使其在余下的时间片内运行

571

12.7.3 Solaris 线程：一个内核级线程例子

图 12-22 是 Sun Solaris 操作系统的线程结构。

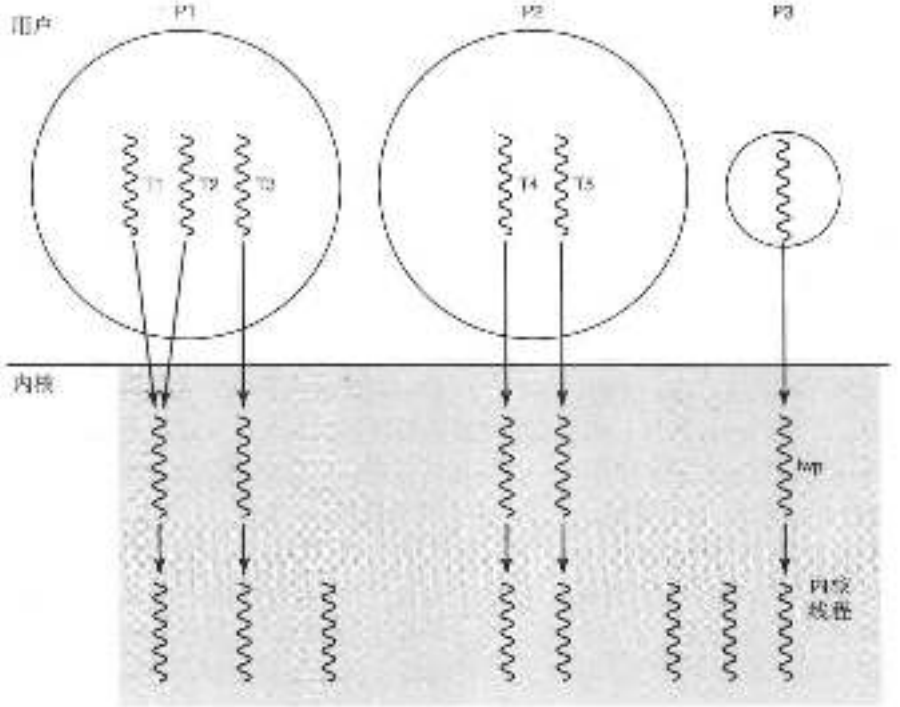


图 12-22 Sun Solaris 线程的结构。进程中的一个线程被绑定到一个轻量级进程（lwp）。同一个进程的多个线程可以被绑定到同一个 lwp。在一个 lwp 和一个内核线程之间有一个一对一映射。处理器调度的单元是内核线程

进程是表示执行程序的实体。一个进程可以创建任意数量的包含于该进程的线程。操作系统允许线程的创建者拥有在该进程中对这些线程调度语义的控制权。例如，线程可以以完全并发的方式或者以协同的方式运行。为了支持这些不同的语义，操作系统承认3种线程：用户，轻量级进程（lwp）以及内核。

1）**内核**：内核线程是一种调度单元。我们将看到它们与lwp和用户线程之间的关系。

2）**lwp**：lwp（轻量级进程）是进程在内核中的一种表示。每个进程在启动后，就关联到一个不同的lwp。在一个lwp和一个内核线程之间有一个如图12-22所示的一对一的关联。另一方面，一个内核线程可以从任意lwp上取消绑定。操作系统使用这种线程来实现需要独立于用户级进程的功能。例如，内核线程可以作为执行设备相关函数的载体。 [572]

3）**用户**：顾名思义，这些是用户级线程。

操作系统支持的线程创建调用可以创建用户级线程。线程创建调用指定了新创建的线程是否要被依附到某个已存在进程的lwp上或者分配一个新的lwp。例如，图12-22中进程P1的线程T1和T2。它们都以协同的方式执行，因为它们绑定到了同一个lwp。任意数量的用户级线程可以绑定到一个lwp。另一方面，进程P1的线程T3与T1或T2中的某一个并发地运行。P2的线程T4和T5也是并发地执行。

调度器的准备队列是准备运行的内核线程集合。它们中的一些，根据其与lwp的绑定关系，以用户线程或者进程的方式执行。如果一个内核线程被阻塞了，关联的lwp以及用户级线程也被阻塞了。由于调度单元是内核线程，所以如果底下的平台是多处理器的，那么操作系统可以在并行的处理器上并发地调度这些线程。

理解这种结构中线程切换的内在开销是一个有趣的事情。每个上下文切换都是从一个内核线程到另一个内核线程。然而，切换的代价变化极大，这取决于与内核线程绑定的是什么。上下文切换代价最小的形式是在绑定到同一个lwp的两个用户级线程之间（见图12-22中的T1和T2）。假定进程在用户级有一个线程库。因此，线程的开销完全处于用户级（类似之前讨论的用户级线程）。在同一个进程的lwp之间（见图12-22中的T1和T3）切换是代价稍高的一种上下文切换。这种情况下，直接开销（保存和加载TCB）是通过内核执行切换的时间。由于存储器的层次架构，这种切换没有隐藏的开销，因为线程处于同一个进程内。代价最高的上下文切换是在不同进程的两个lwp之间（见图12-22中的T3和T4）。这种情况下，直接和间接的代价都会涉及，因为线程处于不同进程中。

12.7.4 线程和库

与在用户级或者内核级实现线程的选择无关的是，保证多线程程序使用库的安全性是十分重要的。比如，一个进程的所有线程共享一个堆。因此，支持动态内存分配的库就需要认识到线程可能同时向堆请求内存。拥有这些库调用的线程安全封装以便保证原子性是极其实用的。图12-23是一个例子。其中的库调用隐式 [573]

```
/* original version */

void *malloc(size_t size)
{

  * * * * * *
  * * * * * *

  return(memory_pointer);
}
```

```
/* thread safe version */

mutex_lock_type cs_mutex;
void *malloc(size_t size)
{
  thread_mutex_lock(cs_mutex);

  * * * * * *
  * * * * * *

  thread_mutex_unlock(cs_mutex);

  return (memory_pointer);
}
```

图12-23 库调用的线程安全封装。整个函数封装在一个互斥锁的lock和unlock之间来保证原子性

此为了调用该方法的线程获取了一个互斥锁。

12.8 在单处理器上的多线程的硬件支持

我们来看一看为了支持多线程需要哪些硬件的支持。有 3 件需要考虑的事情：

1）线程创建和终止。

2）线程之间的通信。

3）线程之间的同步。

12.8.1 线程创建、终止以及线程间的通信

首先，我们考虑单处理器的情况。一个进程的多个线程共享同一个页表。在一个单处理器上，每个进程有一个独立的页表。在进行同一个进程内的线程上下文切换时，TLB 或者 cache 没有改变，因为所有的内存映射和 cache 的内容与新线程依然是相关的。因此，线程的创建和终止，或者线程之间的通信，都不需要任何特殊的硬件支持。

574

12.8.2 线程之间的同步

我们思考一下实现互斥锁需要什么。我们需要一个初始值为 0 的内存单元 mem_lock。其语义如下：如果 mem_lock 是 0，那么锁处于可用状态。如果 mem_lock 是 1，那么已经有某个线程获取了该锁。下面就是 lock 和 unlock 的算法：

```
lock:
        if (mem_lock == 0)
                mem_lock = 1;
        else
                block the thread;

unlock:
        mem_lock = 0;
```

lock 和 unlock 算法需要是原子的。我们看一看上述算法是否符合该条件。unlock 算法是处理器上的一个单独的内存存储指令。由于每条指令的执行是原子的，所以 unlock 也是原子的。

实现 lock 算法一定会经过的数据通路如下：

- 读取内存中某个位置。
- 判断该值是否为 0。
- 将内存中该位置的值设置为 1。

12.8.3 原子的 Test-and-Set 指令

我们知道 LC-2200（见第 2 章）ISA 没有提供任何单条指令可以原子地执行上述数据通路的操作。因此，为了让 lock 算法具有原子性，我们介绍一条新指令：

```
Test-And-Set memory-location
```

这条指令的语义如下所述：

- 将内存特定单元的当前值读入寄存器。
- 将该内存单元设置成 1。

这条指令的关键点是，如果线程执行了该条指令，那么前两个操作（获取内存中某个值并设置成新的值 1）是原子的。也就是说，在执行 Test-and-Set 期间不会有（其他线程的）指

令与其交错执行。 575

例 12-14 考虑下列多处 binary-semaphore 的过程：

```
static        int shared-lock = 0; /* global variable to
                                      both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int L)
{
 int X;

 X = test-and-set (L);

 /* X = 0 for successful return */
 return(X);
}
```

两个线程 T1 和 T2 同时执行下列语句：

```
MyX = binary_semaphore(shared-lock);
```

其中 MyX 是 T1 和 T2 各自的局部变量。

那么，T1 和 T2 可能得到的返回值是什么？

答：

注意指令 test-and-set 是原子的。因此，尽管 T1 和 T2 同时执行过程，但指令的语义保证其中一定有一个先执行这条指令。

所以，可能的结果是：

1）T1 先执行。

那么 T1 的 MyX=0；T2 的 MyX=1。

2）T2 先执行。

那么 T1 的 MyX=1；T2 的 MyX=0。

这意 T1 和 T2 不可能同时得到 0 或者 1 的返回值。

你也许听说过并且看到过广泛运用于铁路上的格雷信号系统。古时（甚至在某些发展中国家的今天），在一个高杆灯上的机械臂（见图 12-24）用来在火车接近共享的铁轨时提醒火车司机停下或者通过。

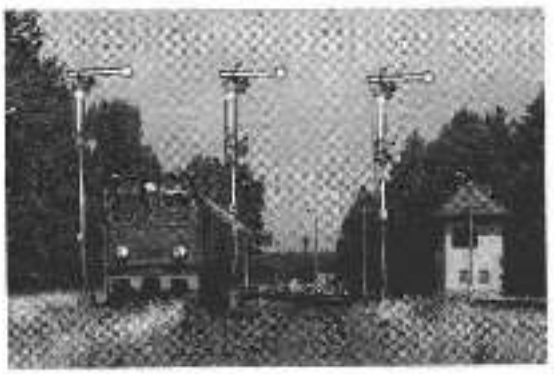


图 12-24 铁路信号量

计算机科学家借用了信号量这个术语。例 12-14 中的过程是一个二元信号量。即，它在许多线程中发出信号，告诉每个线程是否可以安全地进入一个临界区。

Edsger Dijkstra，著名的荷兰计算机科学家，首先提出使用信号量作为协调并发线程活动的一种同步机制。他提出了这种信号量的两个版本。二元信号量是我们刚才看到的一种，其中的信号量向一组相互竞争的线程提供或者拒绝对一个资源的访问。计数信号量是一个更通用的版本，用于当一个资源有 n 个实例时，信号量可以向竞争的线程提供或者拒绝对这 n 个资源的访问。在任意时间点，可以有至多 n 个线程同时共享这些资源。

12.8.4 使用 Test-and-Set 指令的 Lock 算法

介绍了原子的 test-and-set 指令，我们就可以来看一看互斥锁原语的实现，而这正是多线程应用程序编程支持中的核心部分。我们可以通过建立二元信号量来实现 lock-and-unlock 算法，如下所示。

```
#define SUCCESS 0
#define FAILURE 1

int lock(int L)
{
   int X;
   while ( (X = test-and-set (L)) == FAILURE ) {
            /* current value of L is 1
             * implying that the lock is
             * currently in use
             */
            block the thread;

            /* the threads library puts the
             * the thread in a queue; when
             * lock is released it allows
             * this thread to check the
             * availability of the lock again
             */
   }

   /* falling out of the while loop implies that
    * the lock attempt was successful
    */

   return(SUCCESS);

}

int unlock(int L)
{
   L = 0;
   return(SUCCESS);
}
```

当线程调用 lock 算法得到返回结果时，说明其已经获得该锁。使用这个基本的 lock-and-unlock 算法，我们可以建立 12.2 节中讨论过的同步机制（例如，互斥锁和条件变量），以及 12.6 节总结的 POSIX 线程库。

因此，多线程所需的最小硬件支持是一个原子的 test-and-set（简称 TAS）指令。这个指令的关键属性是它对内存单元进行原子化的读取、修改以及写入。还有其他实现了相同功能的指令。在现代的处理器架构中，都在其资源库中实现一条甚至多条拥有该属性的指令。

注意，如果操作系统直接处理线程，那么为了保存原子性，它可以在 Lock 算法执行时简单地关闭中断。TAS 指令允许在用户级执行 Lock。

12.9 多处理器

顾名思义，多处理器由一个计算机中的多个处理器组成，共享所有内存、总线和输入/输出设备等资源（见图 12-25）。这就是所谓的对称多处理机（SMP），因为系统资源对于所有处理器一视同仁。SMP 是增加系统性能的一种代价较高的方法，因为其增加了整个系统的开销。很多我们日常生活中使用的服务器（Web 服务器、文件服务器、邮件服务器）都在 4 路或者 8 路的 SMP 上运行。 576 ~ 578

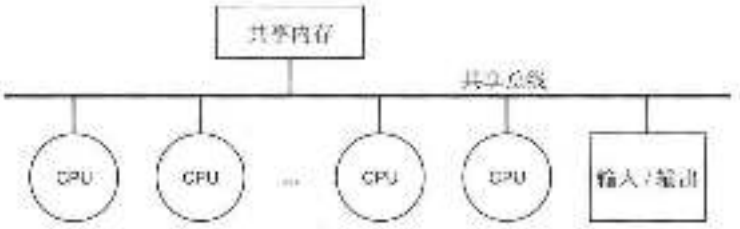


图 12-25 对称多处理器（SMP）。系统资源对所有处理器一视同仁

当程序运行在多处理器上时，在系统级上就会比较复杂了。这种情况下，一个给定程序的线程可以运行在不同的物理处理器上。因此，系统软件（即操作系统和运行时库）和硬件需要协同工作来提供在用户程序级上对线程共享的数据结构所期望得到的语义。

即使是在一个单处理器上，我们也可以看到几个与维持顺序程序的语义、硬件和软件实体相关联的复杂之处：TLB、页表、缓存和内存管理器等。读者可以想象当系统软件和硬件在保证多线程程序语义时的复杂性。我们将在本节讨论这些问题。

系统（硬件和操作系统一起）需要保证三件事情：

1）同一个进程的多个线程共享同一个页表。

2）即使在不同的物理处理器上，同一个进程的多个线程所看到的内存层次是一样的。

3）在并发执行时，线程可以保证进行同步操作时的原子性。

12.9.1 页表

处理器如图 12-25 所示的那样共享物理内存。因此，操作系统通过共享内存中的页表对给定进程的所有线程都是相同的来满足第一个需求。然而，有一些与多处理器上的操作系统有关的问题。理论上，每个处理器独立地执行同一个操作系统。然而，为了保证系统完整性它们需要对一些执行决策进行协调。它们执行特定协调好的操作来维持多线程程序的语义。这些包括：在不同处理器上同时调度同一进程的线程；页替换；维护每个 CPU 上的 TLB 项的一致性。这些问题超出了本书的讨论范围。但是这些问题值得深入探讨，我们也鼓励读者参加操作系统方面的高级课程来进行进一步研究。 579

12.9.2 分级存储体系

每个 CPU 有自己的 TLB 和缓存。我们之前说过，操作系统十分关心 TLB 的一致性，以便保证所有线程看到的共享进程地址空间是相同的。缓存由硬件管理。每个处理器的缓存当前可以缓存相同内存单元的内容。因此，硬件需要负责维护可能在每个处理器的缓存上缓存的共享内存的一致性（见图 12-26）。我们将其称为多处理器缓存一致性问题。

图12-27说明了缓存一致性问题。线程T1、T2和T3（属于同一个进程）分别在处理器P1、P2和P3上执行。三者都在此时将单元X的内容缓存在自己的缓存里（见图12-27a）。T1向X写入。此时，硬件有两种选择：

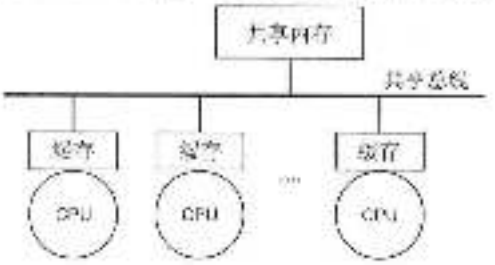


图12-26 每个处理器拥有自己缓存的SMP。硬件保证每个处理器上的缓存内容是一致的

- 将对等缓存中X的拷贝无效化，如图12-27b所示。这需要在共享总线上加入一条失效线。相应地，缓存通过在其上监听来监控总线上从对等缓存传来的无效化请求。根据总线上的这种请求，每个缓存检查该单元是否在本地缓存。如果是，则缓存无效化该单元。接下来同一单元的缺失要么通过拥有最新拷贝的缓存来满足（本例中的P1），要么通过内存来满足，取决于使用的写入策略。我们将这种解决方案称作写入无效化协议。
- 更新对等缓存中X的拷贝，如图12-27c所示。这可能表现为总线上的一个内存写入操作。对等缓存观察到了这个总线请求，更新它们X的拷贝（如果在缓存中）。我们将这种解决方案称作写入更新协议。

侦听缓存是一个流行的基于总线的缓存一致性协议的术语。在本节中，我们展示一个非常基本的、直观的应对多处理器缓存一致性问题的解决方案。如果处理器没有共享的总线（一个广播媒介），那么侦听缓存的方案就无法实现了。在12.10.2节，我们将讨论一种不同的方案，称为基于目录的方案，这种方案将不依赖共享总线来进行处理器之间的通信。在20世纪80年代的中后期，缓存一致性问题的可伸缩方案是一个研究话题，并因此产生了几篇博士论文。读者可以参加计算机系统结构方面的高级课程，从而在这方面进行更深入的学习。

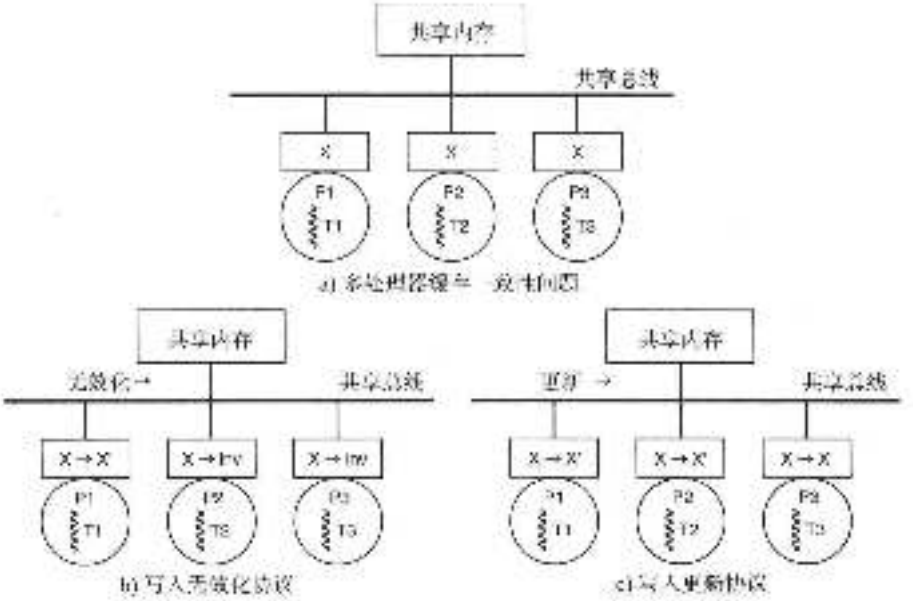


图12-27 多处理器缓存一致性问题及其解决方案

例 12-15 考虑下列 SMP（对称多处理器）的细节：

580
/
581

缓存一致性协议：写入无效化

缓存到内存的协议：回写

初始时，cache 为空；且内存正常。

A 包含 10，B 包含 5。

考虑下列不同处理器 P1、P2 和 P3 的内存访问的时间表：

时间（递增序）	处理器 P1	处理器 P2	处理器 P3
T1	Load A		
T2		Load A	
T3			Load A
T4		Store #40, A	
T5	Store #30, B		

根据这些信息，说出缓存的活动和其中存储的内容。

答：

（T 表示缓存位置是无效的，NP 表示不存在数据。）

时间	变量	P1 的缓存	P2 的缓存	P3 的缓存	内存
T1	A	10	NP	NP	10
T2	A	10	10	NP	10
T3	A	10	10	10	10
T4	A	T	40	T	10
T5	A	T	40	T	10
	B	30	NP	NP	5

12.9.3 保证原子性

在设计上，CPU 共享 SMP 内的内存。因此，（12.8 节中的）lock 和 unlock 算法在多处理器中能够很好地工作。关键的需求是在线程并发地在不同处理器上执行时保证这些算法的原子性。例如，像（12.8 节中的）TAS 指令自动这样原子地运行读取 - 修改 - 写入共享内存单元就可以很好地完成这个功能。

582

12.10 高级话题

我们将向读者介绍一些多处理器和多线程方面的高级话题。

12.10.1 操作系统话题

死锁 在 12.2.7 节中，我们介绍了死锁和活锁的概念。这里，我们将死锁的概念归纳并予以推广。我们可以简单且直观地定义死锁为，当一个线程等待一个永不可能发生的事件时的情况。死锁在计算机系统中发生，存在多个原因。一个原因是系统中有并发的活动，但是硬件和软件资源却又是有限的。例如，我们考虑一个运行多个应用程序的单处理器。如果调度器使用非抢占式算法，运行在处理器上的应用程序进入了一个无限循环，那么其他的所有程序都进入死锁状态。这种情况下，进程都在等待一个物理资源，即处理器。这种死锁通常称为资源死锁。导致死锁的条件有两方面：一个是访问共享资源（处理器）的互斥性需求，另

一个是缺乏抢占性导致资源迫离了当前用户。

当锁要管理一个复杂应用中的不同数据结构时也会发生类似的事情。我们考虑一个有趣的类比。Nick 及其同伴到娱乐中心玩壁球。那里只有一个场和两个球拍。有两个服务台分别用于登记场地和球拍。Nick 及其同伴先去登记领球拍再去登记场地。Alex 及其同伴也有类似的计划，只是他们打算先去登记场地再去登记领球拍。现在 Nick 和 Alex 就死锁了。这是一个资源死锁问题。除了之前说的导致死锁的两个原因外，这种情况下也有两个原因导致了死锁问题：循环等待（Alex 等着 Nick 放弃拍了，Nick 等着 Alex 放弃场地），以及他们每一边可以予部一件资源并等着另一个的事实。复杂的系统软件使用细粒度的锁来加强线程执行的并发性。例如，考虑工资单的处理。支票签发进程可能会锁定所有雇员的记录来生成薪水支票，而绩效加薪进程可能会扫描数据库，锁定所有雇员的记录并给员工加薪。每个进程的持有并等待和循环等待导致了死锁。

总之，计算机系统中涉及进程间资源死锁问题有下列必须同时满足的条件：

- **互斥**：一个资源只能在互斥方式下使用。
- **无抢占**：持有一个资源的进程自愿放弃。
- **持有并等待**：一个进程允许在等待其他资源时持有一个资源。 583
- **循环等待**：在等待资源的进程之间有循环依赖的关系（A 等待由 B 持有的一个资源；B 等待由 C 持有的一个资源；$C\cdots X$；X 等待由 A 持有的一个资源）。

这些是死锁的必要条件。有 3 种处理死锁的策略：死锁避免、预防和检测。对这些策略感兴趣的读者可以参阅操作系统方面的高级教材（例如，[Tanenbaum，2007；Silberschatz，2008]）。这里我们给出这些策略最基础直观的内容。死锁避免算法是极其保守的。它基本上假定资源的请求模式是先验知识。因此，算法可以做出拒绝有不会导致死锁的资源分配策略。例如，如果你手上有 $100，而且你清楚在最坏情形下你需要 $80 来度过这个月余下的日子，你就知道你可以给朋友借出 $20。如果你朋友需要 $30，你只能说不，因为那样你就有可能进入一种无法顺利度过这个月的情况了。然而，这个月你可能有几顿午饭或晚饭是免费吃的，那样就不需要 $80 了。所以，你做出的关于能借给朋友多少钱的策略是一个基于最坏情形下的保守策略。你可能猜到了，死锁避免会由于其内在的保守性导致资源利用率低下。

更加重要的是，死锁避免并不实用，因为它需要未来资源请求的先验知识。一个更好的策略是死锁预防，防止出现上述 4 个死锁的必要条件。其基本思路是破坏某个必要条件，从而防止系统死锁。还是用前面借钱的那个例子，你可以给你朋友借出 $30。但是，如果发现这个月你需要 $80，你就从借给朋友的 $30 中再要回 $10。这种策略破坏了“无抢占”的必要条件。当然，同样的预防策略可能无法适用于所有资源种类。例如，如果进程需要以互斥方式访问一个单独的物理共享资源，那么避免死锁的方式是有与请求者数量一样多的共享资源。这可能看起来有些疯狂，不过仔细想想，这就是一个部门打印机的共享方式。简单地说，我们对打印任务进行假脱机、即将任务缓存，然后等待物理打印机就绪。“假脱机”或者“缓存”是破坏必要条件“互斥”的一种方式。类似地，为了使条件“持有并等待”不成立，我们可以命令所有资源需要在开始运行前同时获取。回壁球的例子，Nick（或 Alex）需要同时登记场地和球拍，而不是先登记其中的一个。最后，关于条件“循环等待”，我们可以给资源确定顺序，要求所有请求必须按顺序进行。例如，我们要求你必须在请求球拍（资源 #2）前先去申请场地（资源 #1）。这就能保证不会出现循环等待。

584 死锁预防可以比死锁避免获得更好的资源利用率。然而，它依然是一个保守方案。例如，

在进程开始前要求其获得所有资源，这一定可以预防死锁的产生，但是该进程并不需要在整个这段时间内使用所有资源，那么资源就被浪费了。因此，一个更好的策略是死锁检测与恢复，这种方法让资源请求和授权可以更自由地进行。如果一个死锁发生了，我们可以通过一种机制发现它并予以恢复。用壁球的例子，当一个前台工作人员注意到死锁后，她会从 Nick 那儿拿来球拍，叫上负责球拍的同事，并让她把 Alex 叫到前台来解决死锁问题。

例 12-16 考虑一个包含了 3 种资源的系统：1 个显示器，1 个键盘，1 个打印机。

有 4 个进程：

P1 需要所有的 3 种资源。

P2 需要键盘。

P3 需要显示器。

P4 需要键盘和显示器。

解释如何使用死锁避免、预防和检测来分别满足 4 个进程的需求。

答：

我们考虑每种策略的解决方法。

避免： 在一个进程开始时将所有需要的资源捆在一起进行分配。此时，如果 P1 正在运行，那么 P2、P3、P4 就不会运行了；如果任何的某一个正在运行，那么 P1 就不能开始了。

预防： 我们人为地定义资源的顺序——键盘、显示器、打印机。所有进程总是按照上面的顺序请求资源，在完成时同时释放所分配的资源。这保证了不会出现循环等待（P4 无法持有显示器再请求键盘；P1 无法在已经持有显示器时再请求打印机，等等）。

检测： 允许资源独立地以任意顺序被请求。我们假设所有进程都可以立即开始。如果进程 P2 请求一个资源（比如，键盘），而它当前分配给了另一个进程 P4，如果 P4 正在等待另一个资源，那么就会强制让 P4 释放键盘，将键盘分给 P2，并重新开始执行 P4。

一个在资源分配中与死锁密切相关的话题是饥饿，即某个进程等待资源时无限期被阻塞的情形。例如，如果资源按照某种优先级进行了分配，如果有一串高优先级的进程不断请求同一个资源，那么低优先级的进程可能出现饥饿问题。回到壁球场的例子，假如教工的优先级高于学生，那么学生就会出现“饥饿”现象。我们在之后讨论同步的经典问题时会给出另一个饥饿的例子。 585

除了资源死锁外，计算机系统对其他形式的死锁也非常敏感。尤其是在本章前面部分讨论的死锁类型，即写出正确并行程序过程中的可能导致死锁或活锁的那种错误。这个问题在分布式系统上会加剧（见第 13 章），因为那里的消息可能由于各种原因在传输中丢失，从而导致了死锁。所有的这些情形都可以归结为通信死锁。

高级同步算法 在本章中，我们已经学习了基本的同步概念。这种概念已经体现在 IEEE 标准中，例如 POSIX 线程库。如我们之前观察到的，这些库及其变种包含于几乎所有的现代操作系统中。大多数并行系统的应用软件都是用这些多线程库构建起来的。

涉及互斥锁和条件变量的编程比较困难而且容易出错。主要原因是共享数据结构的同步访问逻辑贯穿于整个程序中，使得这种编程从软件工程角度来看就变得较为困难了。这种难度体现在大型复杂并行程序的设计、开发以及维护上。

我们将并发编程的需求归结为 3 件事情：

1）线程以互斥的方式（即，串行地）执行程序中的某些部分（我们在 12.2.4 节中提到的临界区）的能力。

2）线程在某个条件不满足时进行等待的能力。

3）线程通知另一个可能等待着某个条件满足的线程的能力。

管程是一种由 Brinch Hansen 和 Tony Hoare 在 20 世纪 70 年代提出来的编程概念，用于满足前面提到的需求。管程是一个抽象数据类型，包含了操作这些数据结构所需的数据结构和过程。用现代编程语言（例如，C++ 或 Java 等）中的术语来讲，我们可以把管程看作在句法上与之类似的对象。我们来看看一个 Java 对象和管程之间的区别。主要的区别是在任意时间点，一个管程内只能有恰好一个活动的线程。换言之，如果需要在程序中设立临界区，那么需要把那部分程序用管程实现。如果一个程序需要多个独立的临界区（比如，例 12-7 的例子），那么就需要用多个管程来构造，其中的每一个管程都对应一个临界区。管程中的一个线程可能会在需要一个资源时被阻塞。为此，管程提供了条件变量，并提供了两个操作，wait（等待）和 [586] notify（通知）。读者可以直接看到管程中的条件变量与 pthread 库中的条件变量之间的对比。管程概念满足了上述 3 个写并发程序时的需求。为了验证这是对的，我们来看一个例子。

例 12-17 对本章中提到处理内例子给出用管程实现的解决方案。

答：

数字化部件和跟踪器的代码已经写好了，并假设有一个名为 FrameBuffer 的管程。数字化部件和跟踪器中的 grab 和 analyze 过程处于管程外部。

```
digitizer()
{
  image_type dig_image;

  loop {
    grab(dig_image);
    FrameBuffer.insert(dig_image);
  }
}

tracker()
{
  image_type track_image;

  loop {
    FrameBuffer.remove_image(&track_image);
    analyze(track_image);
  }
}
```

```
monitor FrameBuffer
{

#define MAX 100

image_type frame_buf[MAX];
int bufavail = MAX;
int head = 0, tail = 0;

condition not_full, not_empty;
    void insert_image(image_type image)
    {
        if (bufavail == 0)
            wait(not_full);
        frame_buf[tail mod MAX] = image;
        tail = tail + 1;
        bufavail = bufavail - 1;
        if (bufavail == (MAX-1)) {
```

```
        /* tracker could be waiting */
        notify(not_empty);
      }
    }

    void remove_image(image_type *image)
    {
      if (bufavail == MAX)
          wait(not_empty);
      *image = frame_buf[head mod MAX];
      head = head + 1;
      bufavail = bufavail + 1;
      if (bufavail == 1) {
        /* digitizer could be waiting */
        notify(not_full);
      }
    }

} /* end monitor */
```

例 12-17 的解答中有几点需要注意的地方。最主要的一点是，pthread 版本里贯穿于数字化部件和跟踪器过程中的同步和缓冲区管理的细节在管程 FrameBuffer 中被巧妙地隐藏了。这就简化了数字化部件和跟踪器过程的代码，只需实现需要的功能即可。这样保证了最终的程序与使用少量同步概念的版本相比会更少出错些。这种解决方案的另一个优雅之处在于，应用程序中可以有任意数量的数字化部件和跟踪器线程。根据管程概念的语义（互斥），所有线程在管程内部的调用都会串行化。总的来说，管程概念明显能提升了并行程序软件工程的效率。

读者可能会好奇，如果管程是这样一个美妙的概念，为什么我们现在不使用它？主要的原因是，它是一个编程概念。在例 12-17 中，我们用 C 风格的语法编写了一个管程，与本章之前的视频处理解决方案相类似。但是，C 不支持管程概念。我们当然可以通过“模拟”的方式，使用操作系统已有的工具实现管程（例如 pthread 库；见练习 21）。

有些程序设计语言已经采纳了管程的想法。例如，Java 是一个面向对象的程序设计语言，支持用户级线程，允许方法（即过程）组合到一起形成所谓的类。通过在方法前加上“synchronized”关键字，Java 保证在运行时的任意时刻，给好只有一个用户级线程可以执行一个给定对象上的同步方法。换言之，当线程开始执行一个 synchronized 方法后，其他线程就不允许再执行同一对象中的同步方法。其他没有“synchronized”关键字的方法依然可以并发地执行，只需保证只有一个同步方法正在执行。Java 没有与管程条件变量类似的内置数据结构，但是它提供了 wait 和 notify 函数来允许在同步方法内阻塞或继续其中线程的执行（见练习 22）。

587
,
588

多处理器上的调度 在第 6 章中，我们介绍了多个处理器调度算法。它们也适用于并行系统。然而，在多个处理器上，调度器需要选择执行一个应用中的多个线程还是不同应用的多个线程。这提供了一些有趣的选项。

在不同处理器上调度线程的最简单方法是有一个由所有处理器上的调度器共享的单独的数据结构（一个运行队列）（见图 12-28）。这也同时保证了所有处理器同等地共享（运行线程的）计算负载。

这种方法有一些问题。第一个问题就是分级存储体系的污染。在现代的拥有多级缓存的处理器上，对需要进入远离处理器级的访问有明显的时间损失（更多细节参考第 9 章）。考虑在处理器 P2 上运行的线程 T1，它运行完自己的时间片后，需要切换出上下文环境。通过一

个中心队列，T1 可能被某个其他处理器，比如 P5，在下一个时间片选中。我们来看一看为什么这不是一个令人满意的情况。T1 占用的大部分内存也许依然在处理器 P2 较近的缓[589]存上。因此，如果 T1 在下一时间片依然运行在 P2 上，可能遇到的缓存缺失会更少。也就是说，T1 与 P2 有亲密度。那么，对于基本调度算法，一种在多处理器上的改进就是使用缓存亲密度，一项首先由 Vaswani 和 Zahorjan 提出的技术。每个处理器上的调度队列（见图 12-29）可以比一个共享队列更好地跟踪管理线程以及它们与特定处理器的关系。为了负载均衡，发现自己已经做完工作的处理器（队列为空）可能会从其他处理器的调度队列中进行任务窃取。另一种对调度算法的改进是对当前持有互斥锁的线程增加时间片长度。其原理来自于一个事实，即同一个程序的其他线程可能直到当前线程释放锁后才能开始真正运行。这项技术也是由 Zahorjan 提出的。图 12-29 是每个处理器内调度队列的概念图。

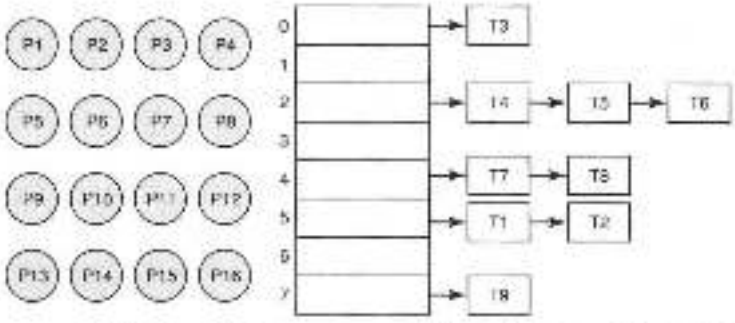


图 12-28 不同优先级的共享调度队列。左边的处理器共享了一个公共的就绪队列

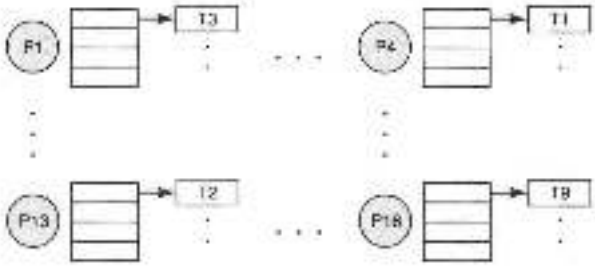


图 12-29 每个处理器上的调度队列。每个处理器有其自己的就绪队列，因此在每个处理器上的调度器访问自己的就绪队列时就不会有竞争关系

我们将向读者介绍两个让多处理器调度更有效的技术，对于多线程应用程序而言，它们尤为有效。第一个称为空间共享。其想法是给一个应用程序，在其生命周期内，共享一组处理器。在程序开始时，向应用程序分配与线程数一样多的处理器。调度器会等到有这么多可用的空闲处理器后才开始运行该程序。由于一个处理器分配给一个线程，所以就没有上下文切换的开销（而缓存紧密度也能得到维持）。如果一个线程由于同步或者 I/O 阻塞了，那么处理器周期就被简单地浪费了。这项技术向应用程序提供了优秀的服务，但是会有浪费资源的风险。对空间共享基本想法的一个修改是应用程序可以根据系统的负载扩大或缩小 CPU 的需求。例如，如果系统只能提供 10 个处理器，那么一个需要 20 个线程的 Web 服务器会把需要的线程数降到 10。之后，若系统有更多可用的处理器，Web 服务器可以声明并增加更多的线[590]程并运行在那些处理器上。

一个空间共享调度器会将系统中的所有处理器分成不同大小的分区（见图 12-30）且每次分配分区给应用程序，而不是分配给单个处理器。这减少了调度器需要维护的用于统计的数据结构数。读者应该会联想到我们在第 7 章中学习的固定大小分区的内存分配。与内存管理方案类似，空间共享可能导致内部碎片。比如，如果一个应用程序需要 6 个处理器，它会得到包含了 8 个处理器的分区，而其中的 2 个处理器则会在应用程序执行期间保持空闲状态。

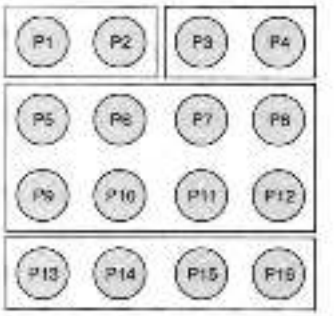


图 12-30 调度器创建了 4 个分区来共享空间。其中包括了两个 2 处理器分区；一个 4 处理器分区；以及一个 8 处理器分区

我们要介绍的最后一个技术是组调度。它对空间调度技术进行了补充。考虑如下情形：线程 T1 持有一个锁；T2 正在等待该锁。T1 释放了锁，但 T2 当前并未被调度，所以没有利用刚刚可用的这个锁的优势。应用程序可能被设计成使用细粒度的锁，让线程只持有一个锁很短的一段时间。在这样的情形下，该应用程序的工作效率将遭受极大损失，这是由于调度器并不了解该程序的线程之间存在的强耦合关系。组调度缓解了这种情况。一个应用程序相关的线程以组的形式进行调度，因此称作组调度。每个线程在不同的 CPU 上运行。然而，与空间共享相比，CPU 并不专属于某个线程，而是时间共享的。

组调度按下述方式工作：

- 时间分为固定大小的片。
- 所有 CPU 在每个时间片开始时调度。
- 调度器使用组的原则把处理器分配给一个给定应用程序的线程。
- 不同组可以在不同时间片内使用同一组处理器。 591
- 多个组可以同时被调度，取决于处理器的可用情况。
- 一旦被调度了，线程到处理器的关联会保持到下一个时间片开始前，即使线程被阻塞了（即处理器会处于空闲状态）。

组调度在其调度决策中考虑了缓存紧密度。图 12-31 说明了 3 个组如何使用组调度原则在空间和时间上共享 6 个 CPU。

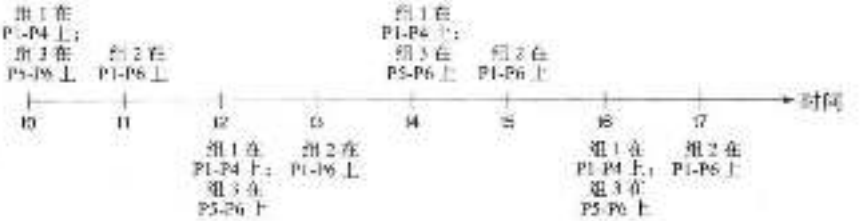


图 12-31 对 3 个不同的组进行组调度的时间线。组 1 需要 4 个处理器；组 2 需要 6 个处理器；组 3 需要 2 个处理器

总之，我们介绍了 4 种用来增加多处理器上 CPU 调度效率的技术：

- 缓存紧密度调度。
- 基于锁的时间片扩展。

- 空间共享。
- 组调度。

多处理器调度器也许会组合地使用上述技术来使效率最大化，这取决于系统要完成的工作量。最后要说明的是，这些技术在顶层，与每个处理器中使用的短期调度算法结合在一起使用（关于短期调度算法可以参考第 6 章）。

有关并发的经典问题 我们将在这里介绍一些有关并发的经典问题，这些问题在推进并行系统中的同步上做出了很大贡献。

1）**生产者－消费者问题**：这也称为有界缓冲区问题。本章中的视频处理应用程序就是一
592 个生产者－消费者问题的实例。生产者不停地向一个共享的数据结构中放入东西，消费者不断地从里面拿出东西。这是很多程序中都会发生的一个常见的通信范例。适用该通信范例的任何应用程序都可以建模为一个流水线。

2）**读者－写者问题**：我们假设你正在尝试获取一个球赛的门票。你进入了一个网站，例如 Ticketmaster.com 网站，选择特定日期球赛的门票，并检查是否还有座位。你可能看了其他选项，例如关于票价和座位方面，然后最终锁定在一组座位上然后购买了球票。当你在做这件事情时，可能有数百个其他的体育迷也在寻找并且购买同一天同一场的球票。直到你真的买好了想要的票，在此之前这些座位依然是可以被任何人选择购买的。数据库包含了所有关于不同日子的座位是否可用的信息。查看座位是否可用是数据库上的一个读操作。同时可能会有很多个读者浏览数据库检查座位的可用性。购买球票则是数据库上的一个写操作。这个操作要求对数据库的互斥访问（至少是对数据库一部分的互斥访问），即在执行写入期间没有其他读者。

上面的例子是经典的读者－写者问题的一个实例，它在 1971 年由 Courtois 等人提出。该问题的一个简单直接的解决方案是：当读者在数据库中时，允许新的读者进来，因为他们不需要互斥访问。当一个写者进来后，如果当前还有其他读者在数据库中，那么就让他暂时等待。一旦所有读者都退出了数据库，就让写者独占地进行操作。相反，如果写者在数据库中，直到写者退出数据库阻塞所有的读者。这种使用互斥锁的方案如图 12-32 所示。

仔细考虑这个简单的方案，我们立刻就能发现其中的缺点。由于读者无需互斥访问，所以当至少有一个读者在数据库中时，新的读者可以继续进入数据库。这导致了写者的饥饿问题。修复这个问题可以这样：当一个新的读者希望进入时，检查是否有等待的写者。如果有就让读者在写者后进入（见练习题 23 ～ 26）。

3）**哲学家就餐问题**：这是一个由 Dijkstra（1965）提出的著名的同步问题。自提出后，任何新的同步方法都会使用这个问题作为检验方法来看看提出的新方案能够多有效地解决这个问题。5 个哲学家围着一张圆桌而坐。他们在吃饭和思考之间转换状态。桌子中央有一碗意大利面。每个哲学家有自己单独的盘子。一共有 5 个叉子，每个叉子在两个哲学家之间，如图 12-33 所示。当一个哲学家想吃饭时，他会拿起两边挨着他的叉子，从中间的碗中拿一些意大利面到自己的盘子里吃。一旦吃完了，他会放下叉子并继续思考[⊖]。

⊖ 这些哲学家不在乎个人卫生，因此会在旁边的人用过后继续用他们可能用过的叉子！

```
mutex_lock_type readers_count_lock, database_lock;
int readers_count = 0;

void readers()
{
   lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
     if (readers_count == 0) {
       /* first reader in a new group,
        * obtain lock to the database
        */
       lock(database_lock);
       /* note only first reader does this,
        * so in effect this lock is shared by
        * all the readers in this current set
        */
     }
     readers_count = readers_count + 1;
   unlock(read_count_lock);
   read_database();
   lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
     readers_count = readers_count - 1;
     if (readers_count == 0) {
       /* last reader in current group,
        * release lock to the database
        */
       unlock(database_lock);
     }
   unlock(read_count_lock);
}

void writer()
{
   lock(database_lock);   /* get exclusive lock */
   write_database();
   unlock(database_lock); /* release exclusive lock */
}
```

图 12-32 读者-写者问题的解决方案，通过互斥锁实现

这个问题需要确保每个哲学家在不妨碍其他人的前提下吃饭和思考。换言之，每个哲学家是一个独立执行的线程，我们希望他们能够最大并发地做吃饭和思考这两件事。思考无需协调，因为它是独立的。另一方面，吃饭需要协调，因为每两个临近的哲学家之间有一个共享的叉子。

一种简单的解决方案是让每个哲学家事先同意在拿叉子时按照顺序拿起一个（比如先拿左边的），然后拿起右边的叉子，最后才开始吃。这种解决方案的问题是它并不能工作。如果每个哲学家同时拿起左边的叉子，每个人都等着取右边的叉子，而这个叉子已经被右边的邻居拿走了。这就导致出现了死锁的循环等待条件。

我们来勾勒出一个可能正确的方案。我们在思考和吃饭之间加入一个中间状态，饥

饿。在饥饿状态下，哲学家会尝试拿起两个叉子。如果成功了，他会继续进入吃饭状态。如果他没有同时拿到两个叉子，他会一直尝试拿直到成功拿到两个叉子为止。怎样允许一个哲学家同时拿起两个叉子呢？此时他的两个邻居必须都不在吃饭状态。而且，他希望在他尝试拿叉子时两个邻居没有改变状态。一旦吃完了，他将自己的状态变为思考，并简单地通知相邻的两个哲学家，这样如果他们饿了就可以尝试吃饭了。图 12-34 给出了哲学家就餐问题的使用管程的解决方案。注意当一个哲学家试图通过 take_forks 去拿叉子时，管程（保证任意时间点只有一个活动的线程在其内部）确保没有其他哲学家能够更改他的状态。



图 12-33 就餐的哲学家[9]：一个饿了的哲学家拿起靠近他两边的同只叉子开吃，并在吃完后将其放下

```
void philosopher(int i)
{
  loop {/* forever */
    do some_thinking();
    DiningPhilosophers.take_forks(i);
    eat();
    DiningPhilosophers.put_down_forks(i);
  }
}
```

图 12-34 a) 每个哲学家线程执行的代码

[9] 图片来源：http://commons.wikimedia.org/wiki/File:Dining_philosophers.png。

```
monitor DiningPhilosophers
{
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (i+N-1) mod N
#define RIGHT (i+1) mod N

condition phil_waiting[N]; /* one for each philosopher */
int phil_state[N]; /* state of each philosopher */

        void take_forks (int i)
        {
          phil_state[i] = HUNGRY;
          repeat
            if ( (phil_state[LEFT] != EATING) &&
                 (phil_state[RIGHT] != EATING) {
              phil_state[i] = EATING;
              break
            }
            wait(phil_waiting[i]);
          until (phil_state[i] == EATING);
        }

        void put_down_forks (int i)
        {
          phil_state[i] = THINKING;
          notify(phil_waiting[LEFT]); /* left neighbor
                                         notified */
          notify(phil_waiting[RIGHT]); /* right neighbor
                                          notified */
        }

        /* monitor initialization code */
        init;
        {
          int i;
          for (i = 1; i < N; i++) {
              phil_state[i] = THINKING;
          }
        }

} /* end monitor */
```

列 12-34 5 个哲学家就餐问题的管程

12.10.2 架构话题

在 12.9 节中，我们介绍了多处理器。我们将稍微深入讨论有关并行架构方面的高级话题。

硬件多线程 流水线处理器使用指令级并行（ILP）。然而，根据我们在第 5 章和第 9 章提到的，指令级并行由于各种因素有其限制，包括分支、有限的功能单元，以及处理器核内存之间日益加大的时间周期差距。比如，缓存缺失会导致流水线的停滞，严重性随发生缓存缺失的等级而增加。需要芯片外处理的缓存缺失导致 CPU 需要等待数十个时钟周期。在多线程程序中，有另一个有效使用处理器资源的方式，那就是执行有就绪的线程。这就是所谓的线程级并行（TLP）。通过 TLP，硬件级别的多线程可以减少由于 ILP 限制带来的流水线停滞的影响。

我们可以用一个例子来进行类比。想象一队人正在一个银行柜员前等待服务。一个顾客走向柜员，柜员发现该顾客需要在业务办理完成前先填完一张表。柜员非常聪明，她要求顾客在旁边填写这张表，并开始处理下一个顾客的业务。当第一个顾客填完表格后，柜员转向他并完成一开始的那个业务。柜员可能会让多个需要填表格的顾客在旁边填表，从而让队列的移动更为迅速。

硬件多线程与这个现实生活中的例子十分相似。单处理器就是银行柜员。顾客是独立的线程。可能导致处理器停滞的长延迟操作就是顾客的填表行为。当线程处于一个长延迟操作中时（例如，缓存缺失导致需要访问内存），处理器就会从另一个准备就绪的线程中获取下一条指令执行。因此，就像银行柜员一样，处理器有效地利用了资源，即使一个或多个就绪的线程卡在长延迟操作上。自然地，这也提出了几个问题。处理器如何知道有多个线程准备运行？操作系统如何知道可以同时在一个相同的物理处理器上调度多个线程运行？处理器怎样维持每个线程之间的状态（PC、寄存器等）？这种技术只是为了提高多线程应用程序的运行速度，还是对顺序程序也有效？我们将在接下来的几个段中解答这些问题。

硬件中的多线程是另一个硬件与系统软件之间合作的例子。处理器架构指定了硬件上能够处理多少个并发的线程。在Intel架构上，这称为逻辑处理器的个数，表示为保持线程状态之间的区别所需硬件资源的副本的等级。每个逻辑处理器有其自己的PC和寄存器堆。操作系统允许一个应用程序将一个线程绑定到一个逻辑处理器上。在12.7节中我们讨论了线程级调度所需的操作系统支持。通过处理多个逻辑处理器，操作系统能够同时调度多个就绪线程使其在处理器上执行。物理处理器通过寄存器堆、PC、页表等的副本集来维护每个逻辑处理器的独立角色。因此，当一个特定逻辑处理器上的一条指令进入流水线时，处理器就知道为了指令执行所需访问的与线程相关的硬件资源。

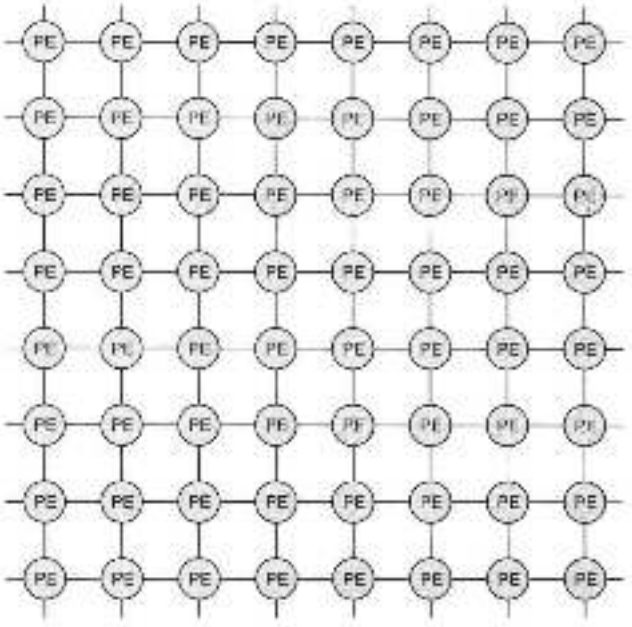
一个多线程应用程序通过硬件对多线程的支持来增加效率，因此即使一个线程由于ILP限制被阻塞了，同一个程序的其他线程也可以获得继续运行的机会。不幸的是，如果一个程序是单线程的，那么它就无法从硬件的多线程上获得好处从而加快执行速度。但是，硬件多线程依然能够帮助提升系统整体的吞吐量，这是因为硬件多线程对于进入流水线的线程是否属于独立的进程还是同一个进程的某个部分是不了解的。

上述讨论引发了另一个问题：使用超标量设计的处理器的ILP能够与TLP共存吗？答案
598 是可以共存，而这正是大多数现代处理器用来增加性能的做法。基本事实是现代的多部件处理器拥有比一个单独线程使用更多的功能单元。因此，使用多部件处理器的ILP和使用逻辑处理器的TLP是一个天衣无缝的结合。

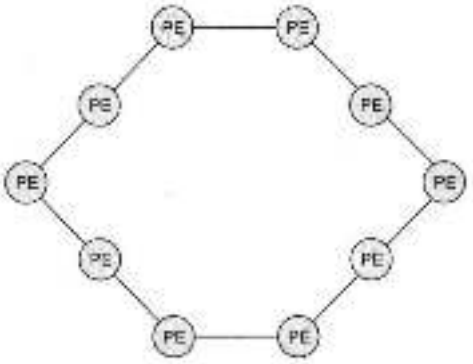
每个厂商给这种融合了ILP和TLP的方法取了不同的名字。Intel的超线程在多数Intel处理器上都是其中的一个标准特性；IBM称其为同步多线程（SMT）并在IBM Power5处理器予以运用。

互联网络 在我们看几种不同的并行架构前，对计算机系统中的元素如何进行内部互联有一个基础的认识将会非常有用。在一个单处理器中，我们已经在第4、9和10章中学习了使处理器、内存以及外部设备相互连接在一起的总线的概念。并行机器中互联网络的一个最简单的形式就是共享总线（见12.9节）。然而，大规模并行机器可能有数千个处理器。我们把这种并行机器中的每个节点称为一个处理单元，简称PE。共享总线很可能成为在如此大规模机器上的各个PE之间通信的瓶颈。因此，大规模机器需要使用更复杂的互联网络，例如网格（每个处理器与其东南西北的4个邻居相连）或树。这种复杂的互联网络使得不同PE之间可

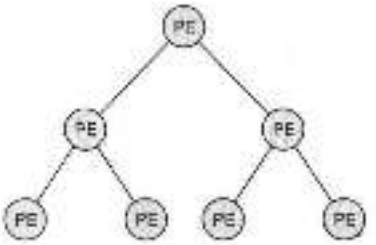
以同时通信。图 12-35 是这种复杂互联网络的例子。每个 PE 可能在本地有连接内存和其他外部设备的总线。



a) 一个网格结构的互联网络



b) 一个环状的互联网络



c) 一个树状的互联网络

图 12-35 几种不同的互联网络

并行架构的分类：1966 年，Flynn 基于并发处理的独立指令和数据流数量，提出了一种简单的对所有架构进行分类的方法。这种分类方法对于理解设计空间和架构选择很有帮助，也有助于我们理解每种架构风格最适合的应用程序种类。表 12-36 用图示的方法说明了这种分类方法。

单指令单数据流（SISD）：这是最经典的单处理器。一个单处理器上有并行成分吗？我们知道，在编程级，单处理器处理一个单独的指令流并且顺序地执行这些指令。然而，在实现级上，单处理器使用指令集并行（简称 ILP）。ILP 使流水化的、超标量的指令集架构实现成为可能（见第 5 章）。

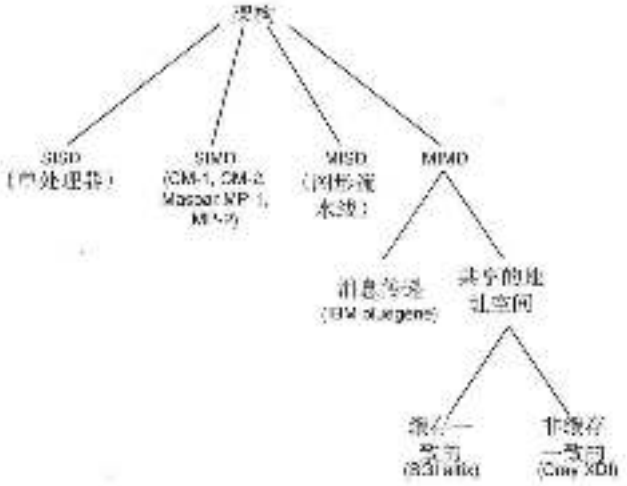


图 12-36 并行架构的分类

单指令多数据流（SIMD） 所有处理器以锁步方式在独立的数据流上执行同一条指令。在系手级的微处理器（20 世纪 90 年代推出的强劲单芯片微处理器）使这种风格的架构在商业上毫无竞争力之前，有些机器就是以这种架构构建的，例如 Thinking Machine Cporation 的 Connection Machine、CM-1 和 CM-2；Maspar MP-1 和 MP-2。这种风格的架构特别适用于图像处理应用（比如，对图像的每个像素应用同一个操作）。图 12-37 是一个 SIMD 机器的典型结构。SIMD 机器的每个处理器称为一个处理单元（PE），并且有自己的从不同数据源上预载的数据存储器。在控制单元中包含了一个单独的指令存储器，取出指令并将其分发给 PE 阵列。指令存储器也预载了需要在 PE 阵列上执行的程序。每个 PE 在来自不同数据存储器的数据流上执行指令。SIMD 模型促进了非常细粒度上的并行度。例如，一个 for 循环可以并行执行，让每个迭代运行在不同的 PE 上。我们把细粒度并行定义为在与其他处理器通信前先在各个处理器上执行一小部分指令（比如说，小于 10 条）。

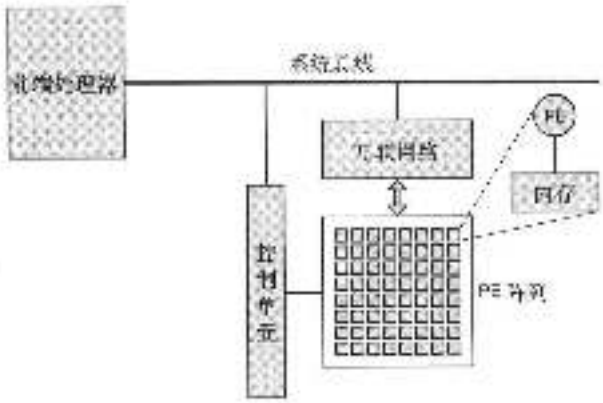


图 12-37 SIMD 机器的结构。前端处理器通常是一个运行类 UNIX 操作系统的性能强劲的工作站。PE 代表处理单元，是 SIMD 机器的基本构建模块

SIMD 机器是用来执行一些细粒度计算密集型任务的重负载机器。SIMD 机器的程序开发通常都是通过一个前端处理器（一台工作站级别的机器）完成的。前端还负责预加载 PE 中

的数据存储器和阵列控制单元的指令存储。所有 I/O 也是通过前端处理器协调执行的。PE 之间通过互联网络通信。由于 SIMD 机器有数千台处理器，所以它们使用前面提到的较为复杂的互连网络（例如，网格或者树状结构）。尽管目前市场上没有这种类型的商业机器，但 Intel MMX 指令就受到了这种架构风格利用的并行机制的启发。这种风格的计算最近出现了一定程度的复兴，典型的有 nVidia 图形卡（或称为图形处理单元，简称为 GPU）这样的流加速器。随着面向流的处理（音频、视频等）变得越来越常见，传统的处理器架构和流加速器正在融合。例如，Intel 的 Larrabee 通用图形处理器（GPGPU）架构代表了这样的一种集成架构，它们为未来的超级计算机提供了良好的平台。

多指令单数据流（MISD） 在算法级，我们可以看到这种架构风格的运用。假设我们要在同一个图像流上运行多个不同的脸部识别算法，每个算法代表了一个不同工作在同一个数据流（即图像流）上的指令流。MISD 是并行架构中的一种类别，但是这种风格的架构并没有什么迷人之处。目前大多数计算还是喜欢使用 MIMD 或者 SIMD 架构。因此，没有哪个架构完全匹配这种计算风格。脉动阵列[6]也许可以算作某种形式的 MISD 架构。脉动阵列的每个单元工作在数据流上并将转化的数据传递给阵列中的下一个单元。尽管这只是一个设计草案，但如果把每个指令看作“数据”和指令从一个阶段到下一个阶段移动的状况，那么就可以把它当做一种粒度非常细的代表 MISD 风格的指令处理流水线。

多指令多数据流（MIMD） 这是最通用的架构风格，而且大多数现代的并行架构都是这种风格。每个处理器有自己的指令流和数据流，相互之间异步地工作。处理器自身可以是现有的处理器。在每个处理器上运行的程序可以是完全独立的，或者是某个复杂应用程序的一部分。如果程序是后者，那么由于这种架构模型的内在异步性，在不同处理器上执行的同一个应用程序的线程之间需要进行同步。这种架构风格最适合支持显示出中等到较粗粒度并行性的应用程序。所谓中等粒度并行度是指在与其他处理器通信前大约执行 10 ~ 100 条指令的程序；粗粒度并行度则是指在处理器进行通信前执行几千条指令的情况。

早期的多处理器都是 SIMD 风格的。因此，对 SIMD 机器的每个处理器都需要进行针对机器做特别的设计。这在通用处理器通过分立电路组装的时代是没有问题的。然而，我们已经说过，系手级微处理器的到来使得个性化构建的处理器在市场中越来越难以生存。另一方面，MIMD 机器中的基本构建是通用处理器。因此，这种架构可以非常好地得益于现成商业处理器技术上的进步。而且，这种架构风格的并行机器可以运行多个独立的顺序应用程序，也可以运行一个并行程序的多个线程，还可以组合地运行上述的两类程序。应当注意，随着流加速器的到来（例如，nVidia GeForce 系列的 GPU），混合的并行机器模型正在形成。

消息传递与共享地址空间的多处理器 MIMD 机器可以分成两大类：消息传递型和共享地址空间型。图 12-38 是消息传递型多处理器的图示。每个处理器有自己私有的内存，进程之间通过互联网络上的消息发送进行通信。处理器之间没有共享的内存。因为这个原因，这种架构也称作分布式内存的多处理器。

IBM 的 Bluegene 系列是符合该模型的当代消息传递型机器。过去的消息传递型机器包括了 TMC 的 CM-5、Intel Paragon、IBM SP-2。上一代的并行机器依赖于特殊的互联网络技术来提供处理器之间的低延迟通信。然而，随着计算机网络的进步（见第 13 章），局域网技术提

599 / 602

[6] 由 Kung 和 Leiserson 在脉动阵列方面发表的文章 [Kung，1979] 是该领域的开创性工作。

供了在消息传递型多计算机系统中的低延迟高带宽通信。因此，一个集群并行机器，即使用吉比特以太网（见13.8节）的局域网进行内部互联的一组计算机，已经成了并行计算十分流行的一种平台。就像pthread库提供了在共享内存环境下进行并行计算的工具，消息传递接口（MPI）是消息传递计算环境中的编程库。

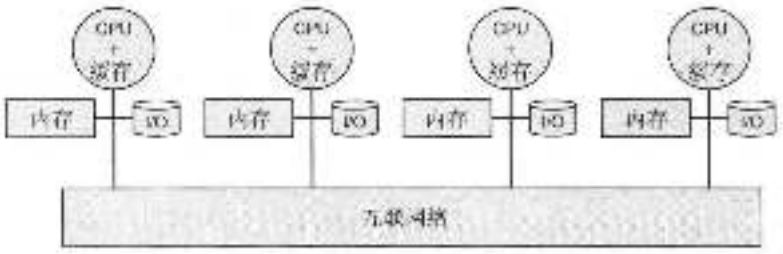


图12-38 消息传递型多处理器

共享地址空间型多处理器是MIMD机器的第二个类别，顾名思义，它提供了内存单元的等效性，即不考虑是哪个处理器访问该单元。换言之，一个给定的内存地址（例如，0x2000）在无论哪个处理器中指的都是同一个内存单元。我们知道，一个处理器有多级缓存。自然地，当访问内存单元时，单元中的数据将传入处理器缓存中。在12.9节中，我们知道这会带来多处理器中的缓存一致性问题。共享地址空间的机器可以进一步分成两大类，根据缓存一致性问题是在硬件还是软件中处理。非缓存一致的（NCC）多处理器提供了共享的地址空间，但在硬件上并无缓存一致性。这种机器包括以前的BBN Butterfly、Cray T3D和Cray T3E等。最近的一个例子是Cray XDI。

缓存一致的（CC）多处理器提供了共享的地址空间，也提供了硬件上的缓存一致性。这种类型的机器包括以前的包括KSR-1、Sequent Symmetry和SGI Origin 2000。现代的这种机器包括SGI Altix。而且，任何典性能集群系统（例如，IBM Bluegene）的独立节点通常都是缓存一致的多处理器。

在12.9节中，我们向读者介绍了基于总线的共享内存多处理器。大规模并行机器（无论它是消息传递型的还是共享地址空间型的机器）都需要比总线更复杂的互联网络。提供了共享地址空间的大规模并行机器通常称为分布式共享内存（DSM）的机器，因为物理内存是分布式的并且与每个单独的处理器相关联。

603
·
604

我们在12.9节中讨论的缓存一致性机制使用了广播媒介，即一条共享总线，因此使得对一个内存某单元的改变可以同时被所有处理器上的缓存看到。对于专用的互联网络，例如树状、环状或者网格（见图12-35），它就不再是广播媒介了。通信是点到点的。因此，需要一些其他的机制来实现缓存一致性。这种大规模共享内存的多处理器使用一种基于目录的方案来实现缓存一致性。想法非常简单。共享内存是物理上分散的，将共享内存的每一块与一个目录关联起来。每个内存单元有一个目录项与之相对应。这个项包含了当前缓存了该内存单元的处理器。缓存一致性算法可以是基于无效化或者基于更新的。目录保存了这些统计信息，用于发送无效化或更新信息。

图12-39是基于目录的DSM的一个结构图。单元X当前被P2、P3、P4缓存。如果P4希望向单元X写入，在P4被允许进行实际写入操作前，与单元X相关联的目录会向P2和P3发送一个无效化信息。总之，分布式目录完成来自不同处理器的内存访问请求，确保对这些

请求的响应返回给处理器的数据值是一致的。每个目录对所负责的那部分共享内存按照收到的顺序处理内存访问请求，从而保证返回给处理器的数据值的一致性。

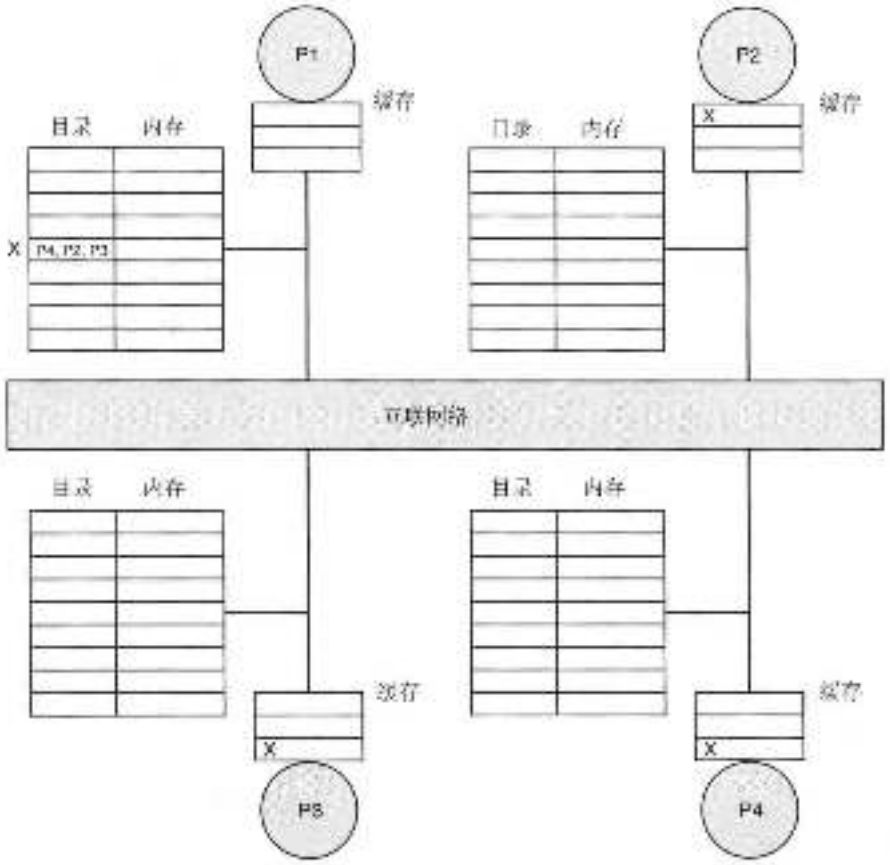


图 12-39 具有基于目录软件的 DSM 多处理器

例 12-18 在一个与图 12-39 类似的 4 处理器 DSM 上，内存单元 Y 在 P3 的物理内存上。当前，P1、P2 和 P4 在各自的缓存中有 Y 的拷贝。Y 的当前值是 101。P1 希望将 Y 改写为 108。说出在 Y 被改写为 108 前发生的一系列步骤。假设缓存的写策略是回写。

答：

a. 由于内存单元 Y 在 P3 中，所以 P1 就在互联网络上向 P3 发送一个请求并为 Y 申请写权限（见图 12-40a）。

b. P3 通过查找 Y 的目录项发现，P2 和 P4 在各自缓存中有 Y 的拷贝。因此向 P2 和 P4 发送将 Y 无效化的请求（见图 12-40b）。

c. P2 和 P4 将各自缓存中的 Y 无效化并通过互联网络向 P3 发回确认信息。P3 从 Y 的目录项中移除 P2 和 P4（见图 12-40c）。

d. P3 向 P1 发送写许可（见图 12-40d）。

e. P1 将 Y 的缓存块中写入 108（见图 12-40e）。

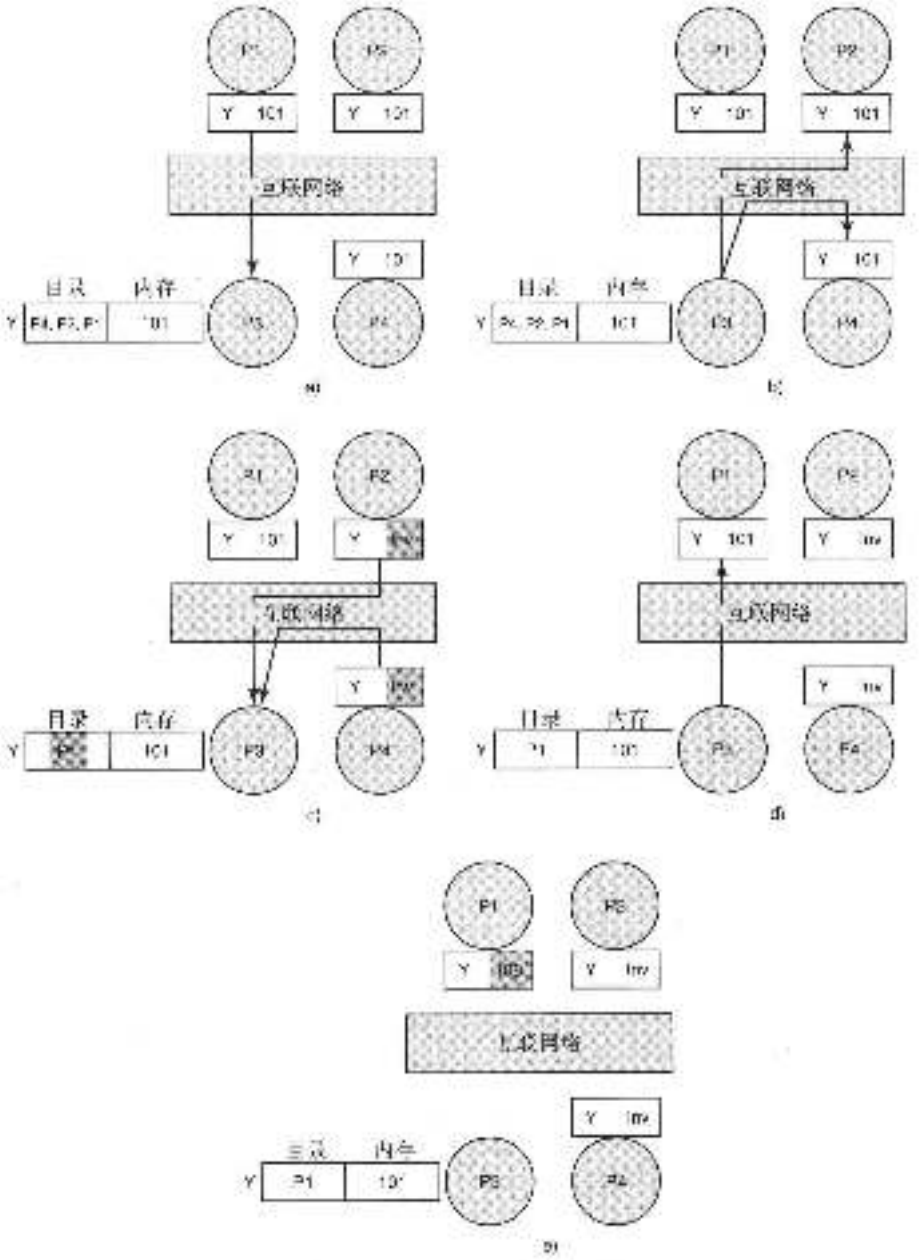


图 12-40 例 12-18 的各个步骤

内存一致性模型与缓存一致性：缓存一致的多处理器确保一旦内存单元（可能当前被缓存了）被修改了，这个生会被传递到所有缓存的拷贝和内存上。我们可以期望在所有缓存的拷贝更新了这个新值前会有一段延时。这就提出了一个问题，从程序角度看会是怎么样的呢？内存一致性模型就是为了在程序员和存储系统之间定义这个视角而产生的。我们在本书中不停地看到系统设计就是关于在硬件和软件之间建立契约。例如，ISA 是编译器编写者和处理器架构师之间的契约——一旦 ISA 设置好了，其实现就可以在 ISA 的硬件实现里使用他的选择。内存一致性模型是一个类似在程序员和存储系统架构师之间的契约。

605
~
607

我们用一个例子来说明。下面给出的是 SMP 的处理器 P1 和 P2 的一段执行记录。内存是共享的，但是处理器寄存器是私有的。

```
                Mem[X] = 0

                处理器 P1                    处理器 P2

Time 0:         R1 ← 1
Time 1:         Mem[X] ← R1                 R2 ← 0
Time 2:         . . . .                     . . . .
Time 3:         . . . .                     R2 ← Mem[X]
Time 4:         . . . .                     . . . .
```

在时间 4 P2 上 R2 的值会是什么呢？直觉的答案是 1。然而，这真的取决于存储系统的实现。我们假设上述执行过程中的每条指令对于该处理器都是原子的。因此，在时间 1，处理器 P1 向 Mem[X] 写入了值 1。这意味着，只要 P1 参与，在时间 1 后 P1 读取 Mem[X] 时，存储系统会保证它得到的值是 1。然而，当其他处理器访问相同内存单元的值时，存储系统关于得到的值能做出什么保证呢？如果在时间 3 时，P2 访问得到的返回值有可能是 0 吗？缓存一致性机制保证在 P1 和 P2 的缓存和内存上，在 Mem[X] 存储的值最终都会变为 1。但是它并不能指定什么时候这些值会变成一致的。而内存一致性模型回答了这个什么时候的问题。

一个直观的由 Leslie Lamport 提出的内存一致性模型称为顺序一致性（SC）。在这个模型中，内存读/写对于整体系统而言是原子的。因此，如果 P1 在时间 1 向 Mem[X] 写入，那么内存该位置的新值在这之后对所有处理器都是可见的。所以，当 P2 在时间 3 读取 Mem[X] 时，存储系统保证它读取到的值是 1。就程序员关心的内容而言，这就是所有关于存储系统中他在编写正确程序时需要了解的细节。带有缓存的存储系统的实现与缓存一致性机制的细节对程序员而言都是无关紧要的。这与我们在处理器 ISA 设计中看到的架构与实现的分离是类似的。

我们给出 SC 内存模型一个更精确的定义。来自某个处理器的内存访问的效果与没有其他处理器访问内存时的效果是一样的。即，每个内存访问（读或写）是原子的。单处理器是这样 [608]
的，因此很自然希望多处理器也是一样的。对于来自不同处理器的内存访问，最终的效果将是来自不同处理器的独立原子的内存访问的任意交叉组合。

一种可视化 SC 内存模型的方式是把来自不同处理器的内存访问看成卡牌游戏中一群玩家需要处理的牌。你可能看过下图中的情景，玩牌老手将两摞牌通过洗牌混合并在一起，并保持每一摞中的牌还是保持原来的顺序。SC 内存模型相当于对来自 n 个处理器的内存访问进行了一个 n 路的洗牌合并。



回到前面的例子，P2 对 Mem[X] 的访问发生在 P1 对 Mem[X] 的访问之后。处理器之间是异步的。因此，另一个相同的程序在执行时这两个访问的顺序可能就会正好反过来。这种

情况下，P2 对 Mem[X] 的访问会得到 0 的结果。换言之，对于前面的那个程序，SC 模型会让存储系统给 P2 对 Mem[X] 的访问返回 0 或者 1。

我们可以看到 SC 内存模型可能导致并行程序中的数据竞争（见 12.2.3 节关于数据竞争的描述）。幸运的是，这并不影响正确的并行程序开发，只要程序员使用前面讨论的同步机制来协调应用程序中线程之间的活动。事实上，就像我们在前几节中看到的，程序员完全不需要知道内存一致性模型，因为他是用诸如 pthread 这样的库进行并发的。就像 ISA 是编译器编写者和架构师之间的契约，内存一致性模型可以看做库的编写者和系统架构师之间的契约。

SC 内存模型对处理器如何使用同步原语来协调它们的活动没有涉及。我们在前几节说过，系统可能会为了协调线程之间的活动而在硬件或软件上提供同步原语。因此，可能包括
609 了这样的一种同步原语，它与正常的内存访问一起来指定硬件和软件之间的契约。这给系统架构师在优化存储系统性能时提供更多的实现选择。面向共享内存系统的内存一致性模型是 20 世纪 80 年代末期到 90 年代初期的一个热门研究领域，那段时间内涌现了不少相关的博士论文。这方面话题的更多细节超出了本书的讨论范围（我们在本章结尾提供了建议的扩展阅读材料）。

12.10.3 未来之路：多核与众核架构

并行计算的未来之路令人向往。新世纪已经是多核处理器的时代了。多核（multicore）的名称来自于一个简单的事实，即芯片由多个独立时钟控制的处理器核构成。Moore 定律预言，芯片密度随着时间的增长速度。到目前为止，这种在芯片密度的增长已经被充分利用来增加处理器性能。有些现代处理器，包括 AMD Phenom II、IBM Power5、Intel Pentium D、Xeon-MP 以及 Sun T1 都使用了多核技术。

然而，随着处理器性能的增长，单芯片处理器的能耗也在稳定增加（见 5.15.5 节）。能耗与处理器的时钟频率成正比。这个趋势迫使处理器架构师在增加芯片密度的同时把注意力集中到了降低能耗上。答案是在一块单独的芯片上放置多个处理器核，每个以较低的时钟频率运转。保持能耗的基本策略就是选择性地将芯片的一部分关掉。换言之，基于能耗上的考虑让我们不得不放弃更快的单处理器转而选择更高的并行度。通过目前包含了多个独立处理器核的单芯片处理器，并行计算不再只是一个选项，而是我们为了发展必须前进的方向。而单芯片处理器发展的下一步则是众核（many core）处理器，即在一块芯片上包含成百甚至上千个核。每个这种处理器核很可能是一个非常简单的处理器，与我们在第 5 章中提到的基本的流水线处理器可能并无很大区别。所以，即使在前面章节中我们没有深入探讨现代处理器微架构的复杂之处，我们提到的简单实现可能也会与未来众核处理器息息相关。

我们也许会这么想：多核架构与封装到一个单独芯片里的 SMP 没有什么区别。而众核架构，就是类似的一个拥有成百上千个 PE 的大规模多处理器的封装版。然而，这并不是平常的生意，尤其考虑到需要彻底重新思考的问题，其范围包括了系统级的电气工程、编程范式和资源管理。

在电气工程级，我们已经提到了减少能量浪费的需求。保持能耗是一件涉及实际的事情。很显然，对一个体积比 1 美分还小、能耗却达到数百瓦的处理器进行降温是一件不可能完成
610 的事情。另一个需要关心的事情是电信号的分布问题，尤其是芯片上的时钟。通常，我们会觉得在线上放置 1 或 0 就可以将这个信号传输给接收者。在有足够时间，即线路延迟（信号在线路上传输给接收者的时间）够长的情况下这是一个安全的假设。我们在第 3 章中看到，

线路延迟是电线上的电阻和电容的函数。当时钟速度增加后，线路延迟可能会接近时钟周期的时间，导致芯片内的线路变成一根根传输线，导致信号强度随距离而衰减。因此，随着在多核和众核时代芯片密度的增加，芯片设计者不得不重新评估关于数字信号的一些基本假设。这就导致在集成电路（IC）设计上出现了新趋势，即所谓的三维设计，也称3D IC。简单地说，为了减少线路长度，芯片被设计成在z维的多个平行面上包含有源器件（导体）的形式。因此，到目前为止，只是局限在二维上的线路将以三维空间的形式进行布局，从而减少有源器件之间的线路延迟。开发3D芯片与建造曙天大楼有些类似。主流的通用处理器将包含这种技术，其成为主流只是时间问题。

在架构级，SMP和多核处理器之间有几个区别。在SMP的硬件级，仅有的硬件共享资源是总线。另一方面，在一个多核处理器中，其中的一些硬件资源对所有核都是共享的。例如，我们在第9章展示了AMD Barcelona芯片的多级存储体系（见图9-45）。芯片上的L3级存储被芯片上的4个核共享。还有其他可被共享的资源，例如从芯片出来的I/O以及存储总线。有效地调度使用好这些共享资源是一个架构需要考虑的问题。

在编程级，在多核上运行并行程序与在SMP上有几个很重要的区别。一种设计并行程序的好方法是被保在计算和通信之间有很好的平衡。在SMP上，线程在需要与运行于其他处理器上的线程进行通信前先分配一定数量的工作。这就是说，计算需要是粗粒度的，从而抵消处理器之间的通信开销。由于一个多核处理器中各个PE之间的邻近关系，可能获得与在SMP上相比更细粒度的并行度。

在操作系统级，SMP和多核处理器之间有根本性的区别。在多核处理器上调度多线程程序的各个线程与SMP相比需要重新考虑更充分地利用核与核之间共享的硬件资源。在SMP上，每个处理器有独立的操作系统的镜像。在多核处理器上，核的优势是可以共享同一个操作系统的镜像。操作系统设计者需要重新思考怎样的操作系统数据结构可以在核之间共享，以及那些需要保持独立。

众核处理器则可能在所有这些级上产生一些全新问题，并加上一些诸如应对局部软/硬件故障等的新问题。 611

小结

在本章中，我们讲述的内容涵盖了使用线程进行并行程序设计的核心概念，操作系统对线程的支持，以及有关线程架构的辅助功能。我们也回顾了操作系统和并行架构方面的一些高级话题。

应用程序在编写多线程的并行程序时需要关心的3件事情是线程的创建/终止、线程之间的数据共享，以及线程之间的同步。12.3节给出了线程函数调用的一个总结，表12-2给出了读者在开发多线程程序时需要熟悉的几个问题。12.6节给出了pthread库所支持的一些重要的线程API调用的总结。

在讨论线程的实现时，我们讨论了在操作系统之上实现线程的可能性，即用户级库，它所需要的操作系统的支持是最少的。大多数现代操作系统，例如Linux、Microsoft XP和Vista，都支持把线程作为CPU调度的基本单元。在这种情况下，操作系统在编程级实现了所需的功能。我们在12.7.2节中介绍了内核级线程，并在12.7.3节中以Sun Solaris操作系统为例介绍了其如何管理线程。

线程所需要的基础架构上的辅助功能是一个原子的读取-修改-写入内存操作。我们在

12.8 节介绍了 test-and-set 指令，并说明了如何通过使用该指令来实现操作系统中更高层次的同步支持。12.9 节提到，为了支持处理器之间的数据共享，需要解决缓存一致性的问题。

在 12.10 节中，我们介绍了与多处理器有关的操作系统和架构方面的高级话题。特别地，我们向读者介绍了死锁的形式化处理方法，复杂的同步概念（例如，管程）、高级同步技术，以及与并发和同步有关的经典问题（见 12.10.1 节）。在 12.10.2 节中，我们介绍了并行结构的分类（SISD、SIMD、MISD 和 MIMD），并深入地讲述了消息传递型和共享内存型 MIMD 架构。

与多处理器和多线程程序相关的软/硬件问题是迷人而深邃的。我们向读者介绍了该领域中的一些令人兴奋的话题。然而，在本章中，我们仅仅是接触了这些问题的表面。我们希望能够激发读者的好奇心，并在更高级的课程上对这些问题进行更深入的研究。本章涉及的一些话题的更深入的探讨可以参考并行系统方面的高级课本，例如 [Almasi，1993；Culler，1999]。

历史回顾

自计算机科学发展早期并行计算和多处理器就一直是计算机科学家和电气工程师的研究焦点。

在第 5 章中，我们了解了流水线处理器设计利用了指令级并行（ILP）。相反，多处理器
612 利用了一种不同的并行性，即线程级并行（TLP）。利用 ILP 并不需要终端用户做些什么不同的事，用户继续写顺序程序即可。编译器，与架构师一起，就可以魔法般地在顺序程序中利用 ILP。然而，TLP 的利用需要做更多的工作。要么我们把程序写成使用多线程的显式并行程序（如本章中的一些例子），要么就需要把一个顺序程序自动转换成一个多线程的并行程序。对于后一种方式，需要对顺序程序设计语言（例如，Fortran）进行扩展，通过程序插入的指示符来进行自动的并行化。编译器利用这些指示符对原始的顺序程序进行并行化。这种方式在并行计算发展早期十分流行，但是最终用得越来越少，因为其应用性被限制到只能利用循环级并行，对函数级并行却有些无能为力。而循环级并行是通过发现程序中“for”循环的各次迭代相互独立，从而把每个迭代或一组迭代转变成一个并行的线程。函数级并行，或称任务级并行，则可以处理程序认为可以并行处理的工作（与本章用来说明线程编程概念的视频监控例子有些类似）。

我们可以轻易地知道利用并行资源来完成更多工作的吸引力。如果一个应用程序能够获得一个给定的单处理器的性能，那么理论上，并行化之后可以在性能上再提高 N 倍。然而，根据我们在第 5 章中了解到的 Amadahl 定律，程序中潜在的串行部分限制了这种性能上的线性增长（见练习 16）。而且，在单处理器和并行机器上组成的独立处理器之间存在性能滞后。这是因为构建一个并行机器并不是简单地当有更快处理器后直接把现有的替换掉就完事。Moore 定律（见 3.1 节）使得在过去的 30 年里，单处理器的性能在持续不断地增长。因此，当下一代更高性能的微处理器上市后并行机器也很快地过时了。例如，一个现在只需花费几千美元的笔记本电脑的性能比 20 世纪 70 年代到 80 年代耗资数百万美元的 Cray 机器更强。时效性是并行机器的软件并没有像单处理器的软件发展那么快的一个主要原因。

并行架构曾经是为需要高性能应用程序而保留的高端市场，主要来自科学和工程领域。主要的公司和它们推出的并行机器包括了 TMC（并行机器的连接机系列，从 CM-1 发展到 CM-5）、Mapsar（MP-1 和 MP-2）、Sequent（Symmetry）、BBN（Butterfly）、Kendall Square Research（KSR-1 和 KSR-2）、SGI（Origin 系列以及现在的 Altix）、IBM（SP 系列）。这些
613 机器的典型特征包括使用现成的处理器（例如，TMC 的 CM-5 使用了 Sun SPARC，SGI 的

Origin 系列使用了 MIPS）或者使用自己定制的处理器（例如，KSR-1、CM-1、CM-2、MP-1 和 MP-2）；一个专用的互联网络；以及与架构的风格相关的黏合逻辑。互联网络是这种架构的一个关键部分，因为处理器之间有效的数据分享和同步依赖于互联网络的性能。

20 世纪 90 年代，随着性能强劲的单芯片微处理器（也称为多线程处理器）的到来，这种技术突破大大撼动了高性能计算市场。由于单芯片的微处理器性能超过了定制的处理器，所以对于并行机器而言，构建这种机器在经济方面的生命力就变成了一个问题。其中的一些通过自身再造生存了下来，很多曾经叱咤风云的并行计算厂商已经消失了（比如 TMC 和 Maspar）。

除了单芯片微处理器的性能，局域网技术也飞速地发展。我们将在第 13 章介绍局域网（LAN）的发展，在此我们只需了解，随着交换吉比特以太网的到来（见 13.8 节），并行机器上专用互联网络的需求就变得越来越小了。LAN 技术的突破促生了一类新的并行机器——集群。集群是一些用现有的 LAN 技术连接起来的计算节点。从 20 世纪 90 年代中后期开始直到现在，集群就一直是高性能计算的主力架构。集群促进了消息传递型编程的发展，而就像我们前面说过的，MPI 通信库已经成为在集群上编程的事实标准。当然，随着技术的发展，计算节点中的内容在发生变化。比如，现在一个 n 路 SMP（n 可能是 2、4、8 或 16，取决于厂商）作为一个计算节点已经不是什么罕见的事情了。而且，每个 SMP 中的处理器可能是一个硬件多线程的多核处理器。这就产生了一种混合的并行程序设计模型：在节点内部使用共享内存型的程序设计，而在节点之间使用消息传递型的程序设计。

练习题

1. 比较进程和线程。
2. 线程从哪里开始执行？
3. 线程什么时候终止？
4. 同时能有多少个线程获得一个互斥锁？
5. 一个条件变量允许线程进行有条件还是无条件的等待？
6. 定义死锁。解释死锁如何发生以及如何防止。
7. 描述下列发生的问题：

```
if(state == BUSY)
  pthread_cond_wait(c, m);
state = BUSY;
```

8. 比较 PCB 和 TCB 的内容。
9. 在只调度进程而不是线程的系统上使用用户级线程是否有意义？性能会提高吗？
10. 选择下列中的某一项将句子补充完整：

    在一个单处理器中的用户级线程同步

    - 不需要特别的硬件支持，因为关闭中断就足够了。
    - 需要一些能够实现读取 - 修改 - 写入的指令。
    - 可以简答地通过 load/store 指令实现。

11. 选择下列中的某一项将句子补充完整：

    确保一个给定进程的所有线程共享 SMP 中的一个地址空间是

    - 不可能的。
    - 可以平凡实现的，因为页表在共享内存中。

- 可以通过小心地给每个线程复制操作系统页表来实现。
- 可以通过提供缓存一致性的硬件实现。

12. 选择下列中的某一项将句子补充完整：

保持 SMP 中的 TLB 一致

- 是用户程序的责任。
- 是硬件的责任。
- 是操作系统的责任。
- 是不可能的。

13. 选择所有对于同一地址空间创建的线程而言符合的选项：

- 它们共享代码。
- 它们共享全局数据。
- 它们共享栈。
- 它们共享堆。

14. 从下列关于线程的描述中，选择正确的：

- 对于没有提供线程支持的操作系统，如果一个进程中的某个用户级线程进行了一个阻塞系统的调用，那么操作系统就会阻塞整个进程。
- 在 pthread 中，执行了 pthread_cond_wait 的线程会一直阻塞。
- 在 pthread 中，执行了 pthread_mutex_lock 的线程会一直阻塞。
- 在 Solaris 中，同一个进程的所有用户级线程需要同等地竞争 CPU 资源。
- 在 Solaris 中，一个进程的所有线程共享相同的页表。

15. 说出 Sun Solaris 中内核线程的优势。

16. 考虑有 100 000 个核的一个多处理器。它用于模拟喷气式飞机的机翼。程序有 80% 是平行的。在这个具有 100 000 个核的多处理器上运行该程序能获得多少加速比？

614
:
615

17. 说明缓存一致性策略中写无效化和写更新之间的区别。

18. 考虑下列关于 SMP（对称多处理器）的细节：

缓存一致性协议：写无效化

缓存到内存的策略：回写

初始时，情况如下：

缓存为空。

内存位置：

C 包括 31

D 包括 42

考虑下列来自处理器 P1、P2 和 P3 的内存访问及其时间顺序：

时间（非实际顺序）	处理器 P1	处理器 P2	处理器 P3
T1		Load C	Store #50, D
T2	Load D	Load D	Load C
T3			
T4		Store #40, C	
T5	Store #55, D		

填充下面的表格，写出每一时刻缓存的内容。

我们已经给出了 T1 时刻的内容。

（I表示缓存单元的值是无效的。NP 表示这不存在值。）

时间	变量	P1 的缓存	P2 的缓存	P3 的缓存	内存
T1	C	NP	31	NP	31
	D	NP	NP	50	42
T2	C				
	D				
T3	C				
	D				
T4	C				
	D				
T5	C				
	D				

616

19. 为什么在仍然持有与相应 pthread_cond_wait 相关联的互斥锁时执行一个 pthread_cond_signal 调用被认为是一个好的编程实践？
20. 为什么从 pthread_cond_wait 调用处继续执行时再检查监视条件是一个好的编程实践？
21. 在 C 语言中用 pthread 库实现例 12-17 的使用管程的解决方案。
22. 在 Java 中实现例 12-17 的使用管程的解决方案。
23. 使用互斥锁写出写者优先的读者－写者问题的解决方案。
24. 使用互斥锁写出读者与写者优先级相同的读者－写者问题的解决方案。（提示：在解决方案中使用 FCFS 规则。）
25. 使用管程实现练习题 23 和 24。
26. 使用 Java 实现练习题 23 和 24。
27. 使用计数信号量写出读者－写者问题的解决方案，任意时刻允许至多 n 个同时的读者或者 1 个写者访问数据库。
28. 图 12-34 给出了哲学家就餐问题的管程解决方案。使用互斥锁重新实现该解决方案。

参考文献注释和扩展阅读

阅读一些关于构建并行机器早期尝试的文献会很有意思。Illiac IV（20 世纪 60 年代开始并花 10 年完成的一个 University of Illinois 项目）[Hord, 1982]、Cmmp（Carnegie Mellon，20 世纪 70 年代初）[Wulf, 1972]，以及 Cm*（Carnegie Mellon，20 世纪 70 年代末）[Swan, 1977] 就是这样一些早期的尝试。Illiac IV 是一台 SIMD 风格的并行机器，而 Cmmp 和 Cm* 都是 MIMD 风格的并行机器。阅读这些项目的相关论文和书籍会对学生大有帮助。

Michael Flynn 的开创性论文给出了并行机器的一种分类方法 [Flynn, 1966]。H. T. Kung [Kung, 1979] 将并行算法按照最适合执行它们并行机器的类型进行了分类。ISA 在单处理器上是软／硬件的一个定义明确的接口。Leslie Lamport 的开创性论文 [Lamport, 1979] 把顺序一致性定义为内存一致性模型，将其作为共享内存多处理器上软／硬件之间的一个契约。

20 世纪 80 年代和 90 年代初期，大量研究开展了如何构建面向共享内存型多处理器的多级存储器体系可大量研究。这使得一种新的缓存一致性协议被定义出来（参见 [Archibald, 1986]，它对基于总线监听的缓存一致性协议进行了比较性的研究）。研究者也致力于定义新的内存一致性模型，作为硬件与软件之间的契约（[Adve, 1996] 简单介绍了内存一致性模型）。市场上出现了多种基于总线的商业机器。617

20 世纪 80 年代早期至右大量关于建立大规模多处理器尝试的研究。那段时期的一些著名例子包括 IBM RP3[Pfister, 1985]、NYU Ultracomputer[Edler, 1985]、Illinois 大学的 Cedar 项目 [Gajski, 1983]、

以及 BBN Butterfly[BBN Butterfly，1986]。

在此之后的下一波并行机器在构建时扩展了多处理器的缓存一致性方案，使得对大规模共享内存机器也能较好地工作。[Lilja，1993]介绍了构建大规模缓存一致的共享内存型机器时会遇到的一些问题。120世纪90年代的著名例子包括Kendall Square Research KSR-1[Burkhardt，1992]、Stanford DASH[Lenoski，1992]，以及MIT Alewife[Agarwal，1995]。

最近，构建大规模并行机器的活动主要集中在工业界。及时跟进这方面前沿技术的最佳方式是通过访问网页的方式紧密跟踪并行超级计算机业界领导者（主要IBM和Cray）的最新进展。另一个很有用的资源是定期发布世界"前500强"超级计算机的网站，上面有它们的配置信息、物理位置、以及相对性能等内容[⊖]。

并行系统的系统软件主要朝两个方面发展。第一个方面是编译器、程序设计语言和相关库方面。研究者试图通过主要聚焦于编译器技术来弄明白如何构建高性能的并行软件[Kuck，1976；Padua，1980；Allen，1987]。这使得在顺序程序设计语言中出现了关于并行扩展的定义（例如，FORTRAN-D[Fox，1990]和IBM Parallel FORTRAN[Toomey，1988]），也导致了很多并行线程库的出现（例如，POSIX pthreads[Nichols，1996]、MPI[MPI，2009；Snir，1998]、OpenMP[OpenMP，2010；Chapman，2007]、PVM[Sunderam，1990]、CMU的Cthreads[Cooper，1988]、IVY[Li，1988]、Treadmarks[Keleher，1994]、Shasta[Scales，1996]、Cashmere[Kontothanassis，2005]，以及CRL[Johnson，1995]）。

最近的焦点开始转向如何通过提供更高级编程抽象的方式减少最终用户运行并行程序设计的复杂度（例如，Intel在并发集合上的网页[Intel CnC，2009]，以及Google的Map-Reduce编程[Dean，2004]）。另一个焦点是多核和众核架构上的高效并行编程运行时系统。对于这方面发展的最新动向，可以关注程序设计语言和编译器方面的顶级会议（例如，PpoPP[⊖]和PLDI[⊜]）。

第二个方面是并行机器上高效资源管理所需的操作系统机制。多处理器问题是一个已经研究得较为
618 透彻的话题，大多数操作系统书籍中都会讲到这个。Mellor-Crummey和Scott[Mellor-Crummey，1991]给出了一个关于多处理器上使用的不同同步算法的综述。在操作系统结构上利用多处理器中的局部性从过去一直到现在都是获得了很多关注的话题。Sun的Sprint内核[Mitchell，1994]和IBM的K42项目[Krieger，2006]就是这样的例子。

最近，学术界和工业界都开始大量关注虚拟化技术，从而对资源进行有效的利用，在不同用户之间提供隔离，并能很好地从软/硬件故障中快速恢复。资源虚拟化的基础内容，可以参考[Barham，2003]的很据代表性的论文。同样，访问VMware[⊕]公司，以及Xen[⊗]开源项目的网站也是在这个领域获取最新进展的好途径。这方面的顶级学术会议有SOSP[⊙]、OSDI[⊛]、以及Usenix Annual Technical
619 Conference[⊜]。

⊖ http://www.top500.org/。

⊜ Principles and Practice of Parallel Programming: http://polaris.cs.uiuc.edu/ppopp10/。

⊜ PLDI: http://cs.stanford.edu/pldi10/。

⊕ http://www.vmware.com/。

⊗ http://xen.org/。

⊙ Sympopsium on Operating Systems Principles: http://www.sigops.org/sosp/sosp09/。

⊛ Operating System Design and Implementation: http://www.usenix.org/event/osdi10/。

⊜ Usenix Annual Technical Conference: http://www.usenix.org/event/atc10/。

网络与网络协议基础知识

设想你的计算机没有连接到因特网。我们会认为这样的计算机功能齐全吗？多半不会。虽然我们现在把因特网和网络连接看作理所当然的，但并清楚我们如何从最开始到达这一步仍然会有启示。我们会在本章结尾回顾网络的发展史。首先，我们要了解网络连接计算机的基本要求，无论这些计算机是在同一栋楼里还是横跨了半个地球。

13.1 预备知识

正如我们在第 10 章中提到的，外围设备与计算机系统的其余部分通过以下两种方式之一相互连接：程控 I/O（Programmed I/O）或者直接内存访问（Direct Memory Access，DMA）。前者适用于低速设备，后者适用于高速设备。网卡是一种高速设备，使用 DMA 与系统相连。

到现在为止，我们从前面章节中学到的关于计算机的一切都在我们的控制之中，包括计算机中的硬件和处理硬件的操作系统抽象。虽然将计算机连接到网络只需要一块简单的 DMA 硬件，但这样的连接所带来的影响却是深远的。与使用仍然是计算机中的磁盘等外围设备不同，连接到网络就将使计算机暴露于整个世界之中。将计算机连接到网络使我们能够浏览网页，与世界任何地方的朋友聊天，同时作为网络中信息的使用者和贡献者。另一方面，我们 [620]
无法控制网络上所发生的一切，且无法预测这变幻莫测的网络会如何影响我们的计算机。这就像乘坐过山车——你想要寻求刺激感，但需要有安全的保障。

关于网络的讨论有很多主题，包括网络协议、网络安全、网络管理、网络服务等。可以肯定地说，这些主题中的每个都有专门的教科书，我们在参考书目中列出了一些好的参考书。

本章坚持本书的主题，即硬件和操作系统，因为它们属于你的计算机。伴随这个目标，第 13 章将从硬件和操作系统的角度，专注于网络的基本原理，为你带来一个特殊的计算机网络之旅。我们的目的不是深入细节，而是让你更全面地了解网络，它会激起你更深入地学习这个主题的兴趣。本章不涵盖与网络安全和网络管理相关的内容。

我们采取自上而下的方法来探索计算机网络这个有趣的领域。操作系统有 3 个部分会促进计算机与网络连接。就像 pthreads 库提供了一套并发多线程程序的 API 一样，socket 库提供了一套并发网络应用程序的 API。为操作系统定义并实现这套 API 是简化网络编程的第一步。我们很快就会看到，要使应用程序产生的消息到达它们的目的地，会涉及很多方面的问题。解决所有这些问题的抽象称作**协议栈**（protocol stack），它是操作系统简化网络编程的第二部分。计算机自身通过网卡（Network Interface Card，NIC）与网络相连。操作系统中与网卡交互的**网络设备驱动程序**（network device driver）是简化网络编程的第三部分。

13.2 基本术语

我们将通过学习一些基本术语来开始我们的旅程。在网络用语中，连接到网络的计算

机称作主机（host）。为了将计算机连接到网络，我们需要一个外设控制器，它称为网卡（Network Interface Card，NIC）。图 13-1 展示了连接到网络的多台主机。

图 13-1 使用了一朵巨大的云来表示网络，但图中的网络究竟是什么？我们很快就会看到，它其实是多个网络的集合，这个网络集合的名称是因特网（Internet）。

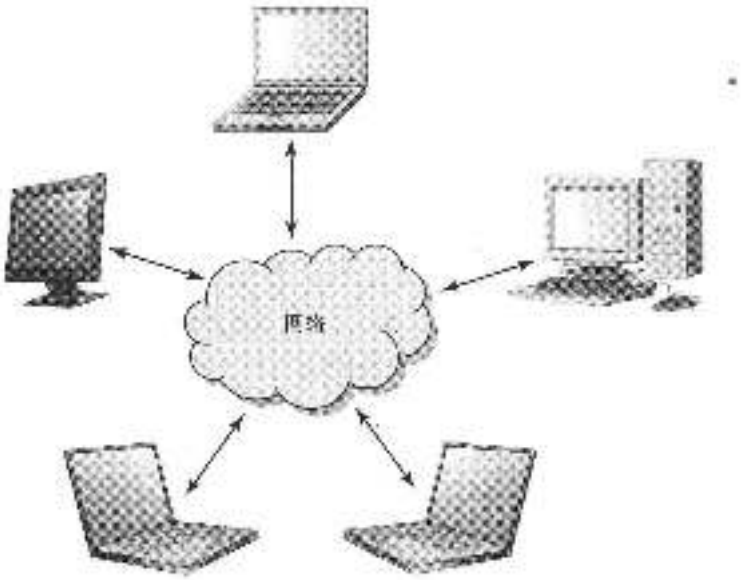


图 13-1 连接到网络的各种主机

什么是因特网？打开任何一本有关计算机网络的书，它都会以一种独特的方式来回答这个问题，而本书尝试以你的角度来解答这个问题。拿邮政服务来做类比。Vasanthi 希望从佐治亚州亚特兰大市给她在印度泰米尔纳德邦 Mailpatti 的奶奶寄一封信，她在信封上写上她奶奶的地址，再把信投入信箱（见图 13-2）。之后邮递员取走这封信。当然，取信的邮递员并不知道如何把这封信一路送到 Vasanthi 的奶奶那里，邮递员只了解他自己的路线——那些需要他投递与收取邮件的房屋。但他知道如果把邮件递交到当地邮局，那里的邮政工人负责这些邮件剩余的旅途。让我们进一步了解这个故事，因为它与因特网的工作方式类似。这些信件在邮局里被分选整理，送往印度泰米尔纳德邦的信件被整理放置在一个特定的箱子中。最终，这个箱子被空运到印度钦奈，信件从钦奈的主邮局被送往 Mailpatti 的邮局。因为没有通往奶奶家的公路，所以 Mailpatti 的邮局使用一辆牛车来运送这最后的一公里。奶奶收到信件非常高兴。

Vasanthi 信件的旅程与 Charlie 给他在加利福尼亚州尤巴市的母亲发送的电子邮件的旅程有着惊人的相似之处。Charlie 居住在亚特兰大，他家使用调制解调器通过电缆接入因特网，调制解调器将 Charlie 与互联网服务提供商（Internet Service Provider，ISP）相连。ISP 作为 Charlie 连入因特网的接入点，它代表了一个接入网（access network）。人们可以通过许多途径（如电缆、电话线、卫星等）来访问因特网。每个人依据自己的情况与喜好选择一个合适的 ISP。接入网就如同邮递员，它知道如何从指定计算机获取东西或者向给定计算机发送东西，但不知道如何让 Charlie 的电子邮件一路传送到加利福尼亚州的母亲。然而接入网知道如何把电子邮件传送给区域 ISP，区域 ISP 在功能上与本地邮局非常相似。区域 ISP 知道如何与其他在全国乃至世界各地的区域 ISP 通信。Charlie 的电子邮件通过亚特兰大地区的 ISP 发送到加利福尼亚海湾地区的

ISP。除了 Charlie 的电子邮件以外，还有更多消息在亚特兰大和海湾地区之间穿梭。因此，网络核心能够处理更大的流量，这类似于仅用一架邮政飞机，就能运送所有从美国寄到印度泰米尔纳德邦的邮件。通过海湾地区的区域 ISP，电子邮件到达尤巴市的接入网。Charlie 的母亲不太会使用计算机，她通过一个连接到她计算机的调制解调器使用电话线拨号连接到因特网。对 Charlie 的母亲来说，接入网是尤巴市当地的电话公司。将电子邮件传送到 Charlie 母亲计算机的最速拨号连接，类似于最终将 Vasanthi 的信件传送到她奶奶的牛车。

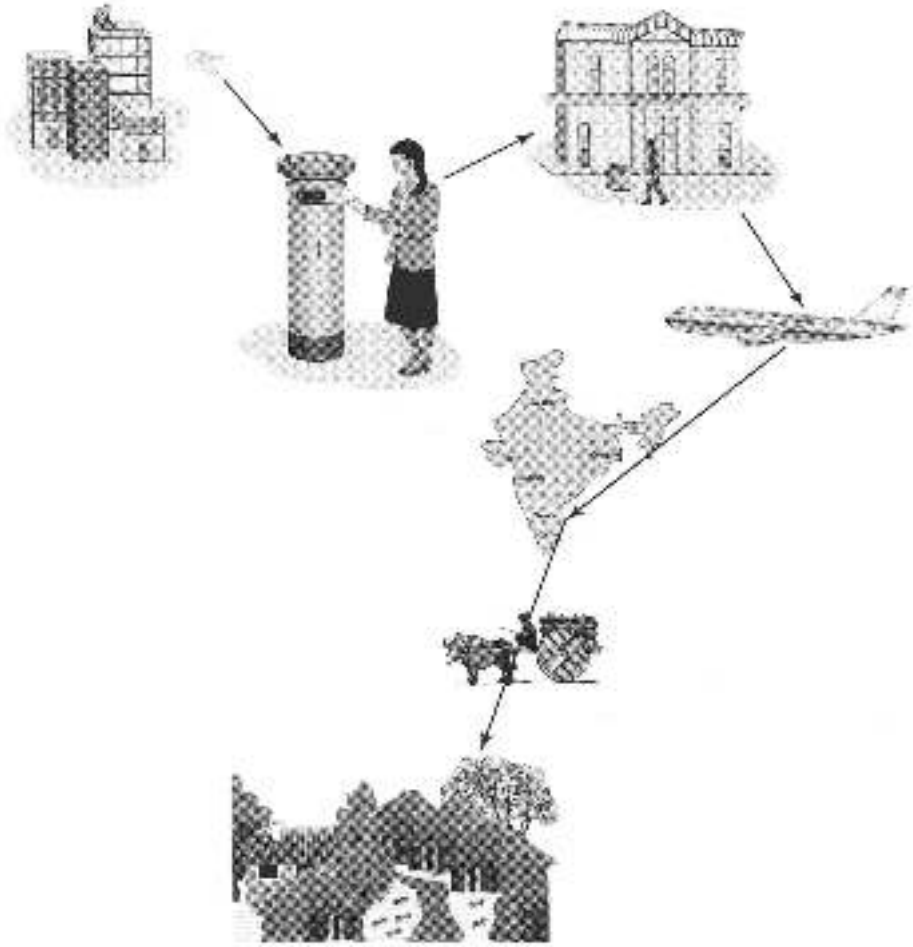


图 13-2 从美国佐治亚州亚特兰大市到印度泰米尔纳德邦 Mailparti 的邮递

图 13-3 中的每朵 ISP 云代表了相应 ISP 的计算机系统。这些 ISP 系统之间不直接相互连接。在网络核心中有称为路由器的设备，负责在全球各种 ISP 之间路由消息。区域 ISP 通过网络核心来知晓其他的 ISP 以及向其他的 ISP 路由消息。这咋一看就像是魔术一样，但事实并非如此。在佐治亚州亚特兰大市的地方邮局如何知道 Vasanthi 写给她奶奶的信件应该用飞机运往印度欣奈？因为美国邮政服务普遍采用了美国邮政编码（ZIP code）来标识全球范围内的每一个地区。因特网同样使用了一个通用的寻址系统来标识每一个可能连入因特网的设备，这就是大家所熟知的 IP 地址（Internet Protocol address，IP address）。Charlie 和他母亲的计算机都拥有一个全球唯一的 IP 地址。接入网、互联网服务提供商（ISP）和核心网络合在一起组

成了我们通常所说的因特网。总之，因特网是一个由网络组成的网络。

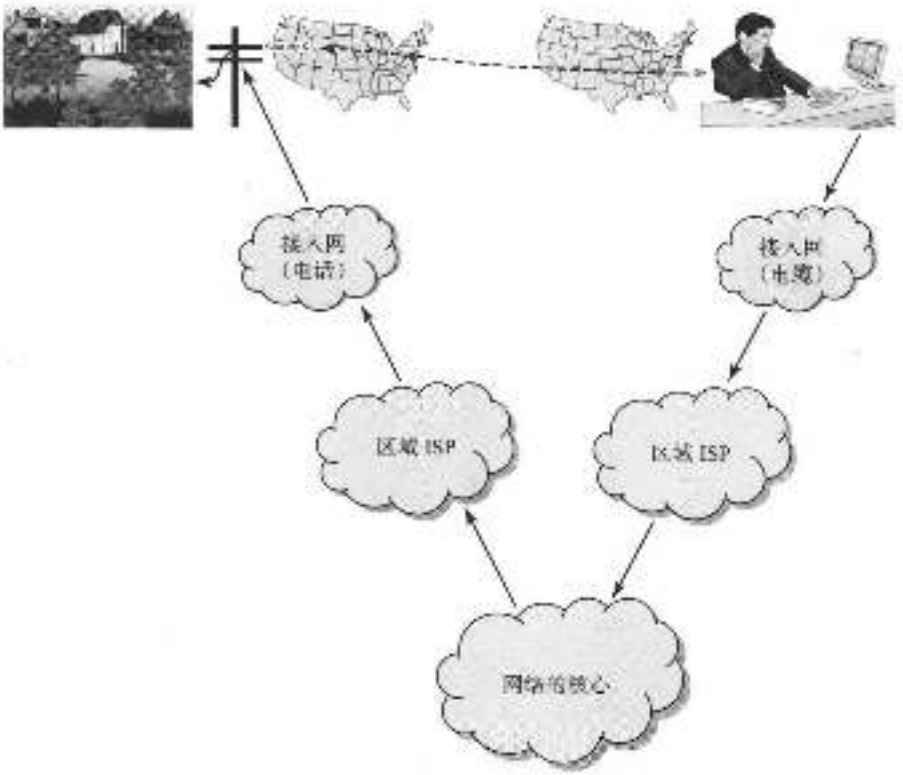


图 13.3 Charlie 的电子邮件从佐治亚理工大市到达加利福尼亚州尤巴市

要理解因特网中的数据包传输，最关键的一点是，一个从源发往目的的数据包会穿过途中一系列位于路由器输入端的队列。在没有任何竞争的情况下，这些队列都是空的，Charlie 的电子邮件将顺利通过这些路由器，到达他妈妈的计算机。然而，在存在其他流量的情况下（如网络拥塞，参见 13.6.3 节），数据包可能会在途中的路由器遭遇到排队延迟（queuing delay）。由于队列的大小有限，所以如果一个路由器的队列已满，数据包可能会丢弃，从而导致数据包丢失（packet loss）。因此，排队延迟和数据包丢失是因特网中数据包传输所拥有的内在问题。我们将在 13.3 节、13.6 节和 13.12 节更详细地讨论这些方面。

在抽象层次上，邮政服务将信件从人 *A* 投递到人 *B*。让我们来看看全球邮政服务的基础配套设施，它包括邮递员、自行车、汽车、船只，甚至牛车，还包括邮政火车、货车和飞机。同样，因特网由一大堆小设备所支撑。在本章的其余部分深入研究细节之前，让我们对网络的概况做一个鸟瞰。

我们将扩展之前对主机的定义，以涵盖我们在日常生活中所接触的各种计算机（笔记本电脑、掌上电脑以及 Web 服务器和邮件服务器等服务器）。这些计算机代表了网络边缘。我们还经常听到术语叫客户端和服务器，这些术语主要是指一个给定的主机所扮演的角色。例如，当我们用 Google 进行搜索时，我们的机器就是一个客户端，在另一端的搜索引擎中对应的机器就是服务器。网络边缘与网络核心也有区别，位于网络深处路由数据包的机器构成网络核心。

现在，如果你在学校、办公室或者宿舍，你会将你的计算机连接到一个局域网（Local Area Network，LAN）。即使是在家里，特别是在西方国家，许多人可能有一个局域网来连接家中所有的机器。正如我们已经提到的，你在家里会连接到一个ISP，这可能是一家有线电视公司、一家电话公司或者一家卫星公司。这些ISP之间有彼此通信的方法，因此无论网络边缘使用了哪一个ISP，在佐治亚州亚特兰大市的Charlie总是能够给他在加利福尼亚州尔湾市的母亲发送电子邮件。还有一些其他类型的硬件共同完善因特网的基础设施，包括传输位的物理介质自身和连接主机与物理介质的电子线路。这些电路包括了集线器/中继器、网桥、交换机和路由器。我们将在13.9节讨论这些硬件元素和它们的功能。接下来，我们将讨论网络软件。

621，625

13.3 网络软件

网络软件是任何现代操作系统的重要组成部分，我们通常把操作系统的这一部分称作协议栈（protocol stack）。让我们先来了解什么是网络协议，它是一种定义计算机之间彼此通信的消息的语法和语义的语言。读者可能已经从本章使用的术语猜到了，协议是为了使任意两个实体之间能进行交互而规定的公约。例如，即使在计算机内部，处理器和内存在内存总线（memory bus）上的交互行为也遵守相应的协议。我们在2.8节讨论的为寄存器保存/恢复所规定的公约也是一种在调用者与被调用者之间的协议。

然而，用于计算机之间通信的协议由于各种原因要得复杂。例如，我们在局域网中使用的一种协议会不断地检查以太网（13.8.1节）。以太网使用的数据包大小最大为1518字节，称为1帧，它包含了数据、目的地址以及其他保证数据传输完整性所必需的信息。对于给定的网络技术，它所使用的数据包的大小限制主要依据协议设计约束来决定，而在介质上的最大传输速率则是另一个问题。就像处理器的时钟速度是实际使用的逻辑延迟特性的函数，任何网络技术的最大传输速率限制也是由物理层信号传输的性质和驱动介质的逻辑延迟特性所决定的。

让我们来考虑网络的使用情况。你可能会通过网络从宿舍给你的家人发送在教室中拍摄的图像，图像可能有几兆大小。你立刻就可以看到问题，一个网络数据包无法容纳整个图像。因此，你的一条消息（任意大小）需要拆分成多个更小的数据包来满足网络介质的物理限制。将一条消息拆分成一组数据包带来了另一个问题。见图13-3，我们看到从Charlie的计算机发出的数据包可能要通过多种不同的网络才能到达他妈妈家里的计算机。并且除了Charlie的电子邮件外，网络上还有其他流量。无论是否有竞争的网络流量，我们都无法预测他的电子邮件会经历多少排队延迟，无法保证一条消息的一组数据包会按顺序到达目的地。考虑一个由3个数据包组成的消息，数据包编号为0、1、2。发送端按顺序发出数据包。但是在发送端和接收端之间可能有多条路径，网络可以沿不同的路径自由地路由数据包，接收端收到数据包的顺序可能是0、2、1。接收端必须正确地组合这些数据包以便构成原始的消息。因此，数据包的乱序到达是数据传输的第二个问题。第三个问题源自网络体系结构自身的性质——在这个完全分散的网络中，网络的各个部分各自决定其中的数据包如何缓冲、转发和忽略。因此，网络中的数据包可能会丢失。数据包通过网络时发生数据包丢失一般是由于资源不足（例如传输途中路由器的缓冲区容量不足）。第四个问题涉及传输过程中的瞬时故障，这可能会改变数据包的内容。即数据包在网络传输过程中可能会产生位错误。

626

我们总结了与消息传输相关的一系列问题：

1）任意的消息大小与网络数据包的物理限制；

2）数据包的乱序到达；

3）网络中的数据包丢失；

4）传输中产生的位错误；

5）传输途中的排队延迟。

当然，我们也可以让应用程序来解决所有的这些问题，但是如果这样做，任何网络应用程序都需要考虑这些问题，因此解决这些问题并作为操作系统的一部分是有意义的。操作系统解决这些问题的部分就是协议栈。我们很快会清楚为什么这部分软件称为协议栈。

就像系统软件（编译器和操作系统）让任意的应用程序数据结构存入结构严谨的物理内存一样，协议栈让一个任意大小的应用程序消息在网络中传输，并在目的地将消息完整重构。图13-4表示了在两个不同主机上的两个进程P1和P2之间的消息交换。

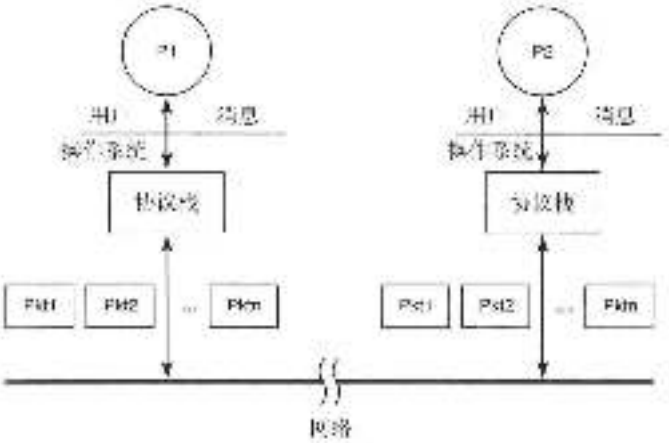


图13-4 两个网络连接的节点之间的消息交换。消息分解成数据包，并通过连接节点的物理介质发送

627

13.4 协议栈

我们应该如何构建协议栈？一种可能的方式是实现一个包含所有我们之前讨论的功能，以保证从源到目的地的消息可靠传输。这种方法的缺点是，它将物理网络的细节不必要地捆绑到了协议中。例如，如果物理网络发生变化，则整个协议会受到影响。

协议分层很好能解决这个问题。协议分层使用抽象的力量来分离这些关系，而不是把所有功能捆绑到一个协议中。这就是为什么我们把这一块操作系统称作协议栈。发出的消息会逐步从最顶层的应用程序到达最底层的物理介质。类似地，收到的消息会逐步从最底层的数据到达最顶层的应用程序。

13.4.1 因特网协议栈

接下来，我们需要了解协议栈必须要解决的细节。自从其起源于20世纪60年代末以来，这一直是网络研究的重点。Vinton Cerf和Robert Kahn等互联网杰出人物曾经展望，较小的网络总有可能需要通过网络连接在一起，形成更大的网络，由此创造了网络互联（Internetting）这个术语。当然，现在因特网已经是一个家喻户晓的词。到20世纪70年代末，当今无处不

生许多协议，如 TCP、UDP、IP，其概念已经在因特网协议架构中存在。

图 13-5 展示了 5 层的因特网协议栈，我们将对每一层的作用进行快速总结。在本章后面的章节中，我们将更深入地讨论传输层、网络层和链路层。



图 13-5 因特网协议栈这种结构自从 20 世纪 70 年代末确定以来，一直使用这 5 层协议栈

应用层（application layer） 顾名思义，这一层负责支持基于网络的应用，例如即时通信（IM）、多人视频游戏、P2P 音乐 / 视频共享、Web 浏览器、电子邮件和文件传输。在这一层中可以使用多种协议，包括用于 Web 应用程序的 HTTP 协议、用于电子邮件的 SMTP 协议，以及用于文件传输的 FTP 协议。本质上，这些协议的作用是为应用程序实体（客户端、服务器、对等节点）之间相互通信提供一种共同的语言。 628

传输层（transport layer） 这一层负责处理应用层的消息并在通信终端之间传输。当然，这一层需要担负我们之前讨论过的变幻莫测的网络（例如，将消息拆分成数据包、处理数据包的乱序到达）。现在因特网上主要使用的两个传输协议是 TCP 和 UDP。传输控制协议（Transmission Control Protocol，TCP）在两个端点之间为应用程序的数据提供一个可靠的和有序交付的基于字节流的传输。TCP 是面向连接的协议。也就是说，在实际的数据传输发生之前，在两个端点之间先建立逻辑连接[⊖]，很像一次电话呼叫。一旦会话结束，就会关闭连接，这在网络用语中通常称为拆除（teardown）连接。另一方面，用户数据报协议（User Datagram Protocol，UDP）类似于通过美国邮政发送一张明信片。它处理有严格界限的消息。也就是说，使用 UDP 连续发送的消息在协议层之间不存在任何关系。简单地说，TCP 提供了流语义的数据传输，而 UDP 提供了数据报语义的数据传输。在发送消息前 UPD 不会建立任何连接，在发送消息后也没有连接拆除。使用 UDP 时消息可能会乱序到达，因为该协议不保证按序传输。总之，这两个主要的网络传输协议之间最显著的差异是，TCP 为端到端提供有序可靠的数据传输，而 UDP 不提供这些。

网络层（network layer） 传输层不知道如何将数据包从源路由到目的地，路由是协议栈中网络层的责任。网络层的作用很简单：在发送端，网络层寻找一条路径将传输层给予的数据包传送到预期的目的地址。在接收端，网络层将数据包传送给传输层，再由传输层负责将数据包整理成传递给应用层的消息。从这个意义上说，网络层的作用与邮政服务非常类似，将信件投入邮箱，并希望信件能够到达目的地。当然，邮政服务由一个人来阅读信封上的地址，并确定如何最好地将信送到目的地。在网络层处理数据包时，我们需要一个确切的格式来描述数据包中的信息（地址、数据等）。在因特网用语中，这层协议的通用名称是 IP 协议（Internet Protocol），它包含了数据包格式和识别数据包向目的地传输的路由。

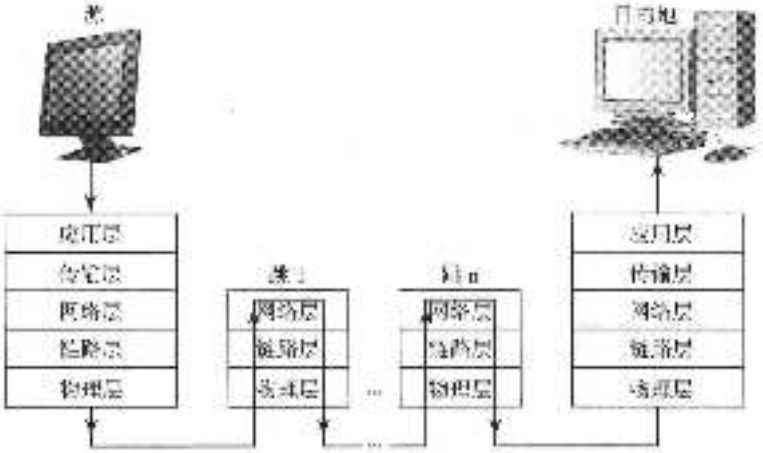
链路层（link layer） 回想邮政服务的类比，Vasanthi 的信件从东治亚州亚特兰大市被空运到印度钦奈，但是从 Mailpatti 邮局到奶奶家的最后的一公里却只能使用牛车运送。飞机和牛车作为邮政服务的不同渠道，在邮政系统运送 Vasanthi 的信件时负责不同的旅程。链路层承担类似的作用，在因特网的节点之间传送 IP 数据包，使数据包能够从源路由到目的地。以太网（Ethernet）、令牌环（Token Ring）和 IEEE 802.11 都是链路层协议。网络层根据数据包将要采取 629

⊖ 电话设备通过预先分配物理资源，在两端之间建立一个真正的连接，称作电路交换（见 13.7.3 节）。另一方面，TCP 是面向连接的，因为它为两端之间以提供是一个连续的外观，不需要预先分配物理资源（见 13.6.5 节）。

到下一跳将 IP 数据包发送到相应的链路层（如果需要，网络层会依据链路层的特性将 IP 数据包拆分成更小的片段）。链路层将这些片段传递到下一跳，那里它们被传回网络层。这个过程一直重复到数据包（可能被分段）到达目的地。可以看到，一个 IP 数据包在它的旅途中可能被多种链路层协议处理。在目的地，网络层重组这些片段来重构原始的 IP 数据包。

物理层（physical layer）这一层负责在物理上（电、光等）将数据包的位从一个节点传输到下一个节点。从这个意义上说，这一层与链路层密切相关。一个给定的链路层协议可能使用多种物理介质来传输位，而每一种物理介质都可能有独特的物理层协议。例如，以太网可能对于双绞线、同轴电缆、光纤等不同物理介质使用不同的物理层协议。

图 13-6 展示了一条消息从源出发经过多次网络跳到达目的地所经过的协议栈。



630 图 13-6 数据包通过网络的旅途。中间节点指为“目的”向目的地传输消息的数据包

在任意两层协议栈之间，有着明确的接口。分层模型使每层协议能够独立于其他层做决策。

分层是一个结构化的工具，避免了复杂的协议栈。它划分各层之间消息传输与接收的责任。模块化可以使新的模块集成在某个特定的层上，尽量不改变其他层。例如，在协议栈中，新出现的物理层会影响链路层，但原则上不会影响网络层和传输层。乍一看这似乎是一个潜在的不利因素，分层可能造成性能的损失，因为消息需要穿过多层协议。然而明智地定义层与层之间的接口可以防止效率降低。

对于协议栈的层数并没有明确的规定。实际上，这取决于协议栈所提供的功能。

13.4.2 OSI 模型

国际标准化组织（International Standards Organization，ISO）已经制定了一个 7 层模型的协议栈，即开放系统互连（Open Systems Interconnection，OSI）模型。OSI 7 层模型是一个抽象的参考模型，用于说明协议栈的功能并建议如何划分各层。图 13-7 展示了这个参考模型。



图 13-7 OSI 参考模型是由国际标准化组织（ISO）制定的一个抽象参考模型

比较图 13-5 和图 13-7，我们可以看到后者比前者多了

两层：表示层（presentation）和会话层（session）。顾名思义，会话层负责管理两个端点之间特定的通信会话。例如，假设你在网络上与朋友有几个同步的即时通信（IM）会话，会话层对每个这样的通信会话维护进程级的信息。传输层负责保证任何输到端的消息可靠传输。会话层抽象应用程序消息传输的细节，提出一个更高级的应用程序接口（例如，UNIX 套接字）。表示层涵盖了许多应用程序所共有的功能。例如，格式化显示窗口中的文本输出是独立于你可能会使用的特定 IM 客户端程序（AOL、MSN 等）的。因此，不依赖于应用程序内部细节的演示功能（如本地显示文本字符、格式化和字符转换）属于表示层。 631

13.4.3 分层的实际问题

OSI 模型作为一个有用的参考模型，确保每个协议栈的实现涵盖所有必要的功能。然而，实际实现的协议栈很少能够严格坚持模型所规定的分层。作为实际问题，因特网随着 20 世纪 80 年代所定义的 OSI 模型一同发展。正如我们之前提到的，对于负责网络通信的传输层和网络层，TCP/IP 协议是事实上的标准。主要的观察结果是，在因特网演变的过程中，如 TCP 和 IP 之类的标准协议导致 OSI 模型划分的层次折叠。例如 HTTP、FTP 和 SMTP 等协议包含了 OSI 模型中的第 7 ~ 5 层，TCP 和 UDP 在 OSI 模型的第 4 层，IP 包含了 OSI 模型中第 3 层的功能。网络接口（如计算机的以太网卡）假定在 OSI 模型中的第 2 层。

我们对于 5 层网络协议栈给出了一个高层次的描述，有许多优秀的教科书涵盖了这样层次的细节。我们从一开始就说过，本章的目的是让读者从系统体系结构和操作系统的角度更全面地了解作为一个重要 I/O 设备的网络。

我们将采取自上而下的方法来探索协议栈的各个层以及其中的设计思想。我们从应用层开始讨论，为其余的章节设定背景。我们会特别强调传输层，因为这是与操作系统接触最紧密的地方。然后我们继续沿栈向下讨论网络层，网络层同样也是操作系统的一部分。最后我们探讨最接近系统体系结构的链路层。

接下来说明本章其余部分的组织结构。我们在接下来的几节中对传输层、网络层和链路层进行一定深度的探索，是否需要这些细节取决于读者自身的看法。由于我们已经对这些层的功能给出了一个高层次的概述，所以如果读者觉得没必要详细了解传输层、网络层和链路层，则完全可以跳过接下来的这几节。13.9 节概述在现代计算机系统中所使用的网络硬件。13.10 节将讨论层与层之间的关系。然后继续探索在操作系统中实现协议栈的问题。

13.5 应用层

正如我们所知，因特网应用程序已经不计其数，从手机到高性能计算集群，接入网络的设备无处不在。这里，分清应用程序和应用层协议是很重要的。 632

一般来说，任何网络应用程序都包含两个部分：

- **客户端**：这部分是在掌上电脑、手机、笔记本电脑和台式机等终端设备上。
- **服务器**：这部分提供一些网络服务预期功能（例如，搜索引擎）。

网络应用程序的例子包括万维网（World Wide Web，WWW）、电子邮箱和网络文件系统（Network File Systems，NFS）。例如，WWW 的客户端是一个 Web 浏览器，如 Firefox 或 Internet Explorer。WWW 的服务器端称作 Web 服务器，包括 Apache[⊖]（一个开源的 Web 服务

⊖ 参见 http://www.apache.org/。

器平台，用来构建 Web 代理等通用服务）和 Google 与 Yahoo! 等向用户提供专业服务的门户网站。另一个例子，电子邮箱的客户端是 Microsoft Outlook 和 UNIX Pine 等程序，服务器端是 Microsoft Exchange 等邮件服务器。

网络应用程序远比应用层协议大很多。例如，Web 浏览器能够保留 URL 的访问历史，缓存下载的网页等，这些细节使得每个应用程序各不相同。另一方面，由于网络应用程序在客户端和服务器之间的消息交换已经被明确定义，所以它们被实现在应用层协议中。

应用层协议能够适应不同类型的网络应用程序。例如，Web 应用程序使用超文本传输协议（Hyper-Text Transfer Protocol，HTTP）来规定 Web 客户端和服务器之间的交互行为。同样，电子邮箱使用简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）来规定邮件客户端和服务器之间的交互行为。

WWW 和电子邮件等许多网络应用程序，经常超出计算机硬件体系结构和操作系统的限制。这就是为什么你能够从手机以及机场和网吧等的公共终端来阅读电子邮件或访问 CNN 或 BBC 等热门网站。因此，HTTP 和 SMTP 等应用层协议是独立于平台（定义为硬件体系结构和操作系统的结合）的标准，无论这些应用程序的客户端或服务器在什么平台上。

另一方面，操作系统也提供它们自己独有的网络服务。例如，你能够通过 UNIX 文件系统访问文件，或者通过 UNIX 终端使用网络打印机。这也是客户端-服务器应用程序增强操作系统功能的例子。为了支持这些应用程序的开发，操作系统提供了网络通信库。类似于 pthreads 库为一个地址空间内的线程之间的交互行为提供 API，这些库为网络中客户端与服务
633 器的交互行为提供 API。这样的通信库同样代表一个应用层协议。例如，UNIX 操作系统提供了套接字（socket）库来作为构建网络应用程序的 API。其他流行的操作系统，如微软的 Vista 和苹果的 Mac OS，也提供了一套类似的 API 来支持在它们的平台上开发网络应用程序。与实现一个多线程库类似（参见第 12 章），操作系统实现套接字库的 API 也会涉及一些具体问题，这些问题已经超出了本书的范围。有兴趣的读者可以参考讨论这些细节问题的其他书籍[Wright，1995；McKusick，2004]。我们将在 13.15 节讨论使用套接字 API 进行网络编程的基本问题。

13.6 传输层

我们假设传输层提供了一组应用程序编程接口（Application Program Interface，API）调用，以便使应用层能够在网络上发送和接收数据。

- send（目的地址，数据）
- receive（源地址，数据）

让我们列举协议栈中传输层的预期功能：

1）在应用层上支持任意的数据大小。

2）支持数据的按序交付。

3）保护应用程序不受数据丢失的影响。

4）保护应用程序不受传输过程中的位错误的影响。

传输层可以以字节流或消息的形式查看来自应用层的数据。相应地，传输可以是面向数据流或者面向连接的（例如，TCP 协议）。在这种情况下，应用程序数据被认为是连续的字节流。传输层将数据分成称为段的预定义单元，并将这些段发送到目的地的传输层。或者，传输层可以是面向消息或面向数据报的（例如，用户数据报（UDP）协议），在这种情况下，处

应用程序数据的方式类似于通过邮政系统发送明信片。为了方便讨论，我们只使用消息（message）来指代在传输层传输的单位内容。

为了满足网络硬件的限制，传输层在源就将数据拆分成数据包，在目的地的对等传输层再将数据包重新组合成原始的消息并传递给消息接收者。我们将这组功能称作分散（scatter）/汇集（gather）[⊖]。由于数据包可能不会按顺序到达目的地，所以在源的传输层对消息的每个 634
数据包都赋予了一个唯一的序号。无论数据包到达接收端的顺序如何，都可以按照序号来重组为原始的消息。因此，为每个数据包附加一个唯一的号可以解决网络通信中的前两个问题，即任意的消息大小和乱序到达。

由于数据包在传输途中可能会丢失或损坏，所以源需要确认目的地已经接收到了数据包。在何情况对传输层来说，结果都是数据包没有到达预定的目的地。数据包丢失或损坏并不是什么神秘的事情。大家都经历过双手都还拿不下全部购物袋的时候，这时你可能已经掉了一两件物品，但你很难在别人告诉你之前意识到这一点。我们需要考虑网络中的物理资源，当讨论网络层时我们会看到，这些资源在负载时可能会到达其容量限制，从而导致数据包丢失。同样，传输过程中的电磁干扰可能会导致位错误。应当提到的是，这样的位错误并不总会使数据包完全无法使用。我们会在本节结束时看到，当发生位错误时，可以对数据包进行纠错，这通常称为前向纠错（Forward Error Correction，FEC）。但是当错误超出 FEC 算法的修复能力时，仍然等价于数据包丢失。协议需要具备识别数据包丢失的能力，一种可能的方法是使用肯定确认（positive acknowledgement），如图 13-8 所示。或许你在发送邮件时已经使用过邮政服务的"挂号信"功能。邮政服务给寄了名的回执返回给发件人来证明邮件已经成功交给了收件人。邮政服务给每封挂号信一个唯一的 ID 用于追踪。传输层中的肯定确认就类似于此服务。需要注意的是，传输层有一个重要的参数，叫往返时间（Round Trip Time，RTT）。RTT 定义为一个小消息（如 0 字节）从发送端到达接收端，之后再回到发送端所需要的时间（参见图 13-8）。RTT 是发送端发送一个消息之后再收到确认消息所需要的估计时间。在 13.6.1 节我们将看到它用于选择重传超时值。RTT 取决于许多因素，包括发送端到接收端的距离、在网络途中的排队延迟，以及发送端与接收端处理消息的开销。

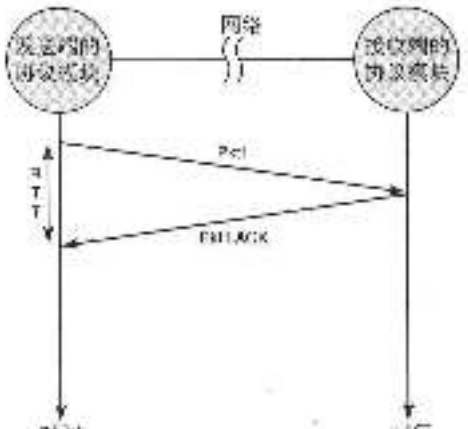


图 13-8 数据包级的肯定确认。当接收端接收到一个有效数据包时，返回一个 ACK 包

但是，网络与邮政服务仍有一些重要的区别。第一，邮政服务中的邮局会为发送人提供确认（以回执的形式）；而网络中的确认消息只会停留在与邮局相似的传输层，不会传递给应用程序。第二，邮政服务中的邮件会作为实体被传递，仅需要一个确认消息；而网络中的消息会被传输层拆分成许多数据包。传输层对于如何处理确认消息有许多种选择，并且这些选择产生了众多的传输协议。

13.6.1 停止并等待协议

一种简单的实现方法需要做到以下几点。

⊖ 按 TCP 中的说法，称为将数据流组合（segmenting）成数据包。

1）发送端每发送一个数据包，就等待一个肯定确认，通常简称为 ACK。

2）每当接收端收到一个数据包，就发送那个数据包的 ACK。ACK 需要包含数据包的信息以使发送端识别被确认的数据包。由于序号是每个数据包的唯一签名，所以 ACK 包中只需要包含已接收数据包的序号。

3）发送端在发送数据包之后会等待一段时间，这段时间称为超时（timeout）。如果发送端在这段时间内没有收到数据包的 ACK，就会重新发送该数据包，如图 13-9 所示。同样，如果接收端再次收到相同的数据包，也会重新发送 ACK（这表示接收端的 ACK 在途中丢失了，如图 13-10 所示）。

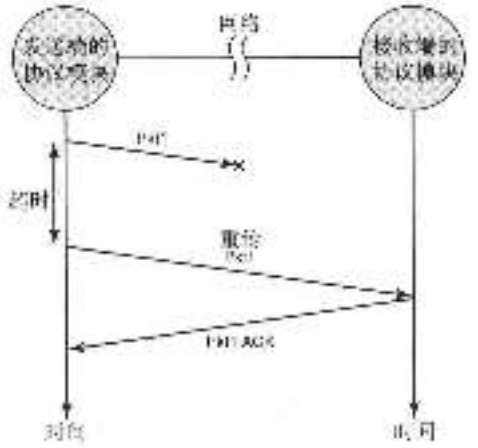


图 13-9 超时重传。如果发送端在超时内没有收到确认，就会重新发送数据包

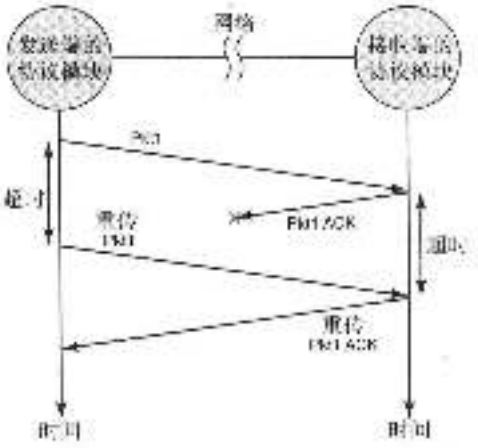


图 13-10 收到重复的数据包。如果接收端收到了相同的数据包，就会重新发送该数据包的确认

我们把这种协议称作停止并等待（stop-and-wait）协议，因为发送端在发送一个数据包后会停止传输，并在继续发送下一个数据包前等待一个 ACK。

例 13-1 用停止并等待协议方法进行可靠传输时，发送端需要在协议内缓存多少数据包？

答：

答案是 1。因为发送端的协议每一次只发送一个数据包并等待 ACK。

我们为什么一定需要序号呢？毕竟在收到当前数据包的 ACK 之前，发送端不会继续发送下一个数据包。原因非常简单和直观，数据包和 ACK 包都可能会丢失，因此发送端和接收端的协议都具备在超时后重传数据包的机制。发送端如何知道一个收到的 ACK 是确认当前数据包还是重复确认前一个数据包？这就是序号的作用。根据单调递增的序列号，发送端可以判断收到的 ACK 是当前数据包的 ACK 还是重复的 ACK。

让我们看看能不能简化这个协议中数据包的序号，根据协议，在任意时刻源和目的地之间都恰好只有一个数据包正在传输，我们真的需要为数据包分配一个单调递增的序号吗？确实不需要。因为依照这个协议，数据包会按顺序从源到达目的地传输，序号纯粹是为了消除重复。所以我们只需要使用 1 位来表示序号就足够了。协议以序号 0 发送数据包并等待序号为 0 的 ACK。当收到序号为 0 的 ACK 时，以序号 1 继续发送下一个数据包并等待序号为 1 的 ACK。因为这个原因，停止并等待协议也经常称为交替位协议

(alternating bit protocol)。

让我们看看如何选取超时时间。以我们以往的经验，我们知道去学校所花费的时间很可能与从学校回家的时间不一样。因为我们可能选择不同的路线、交通状况可能不一样等。这对消息传输来说也一样，消息从发送端到接收端所经过的路径很可能与反方向时不同，两个方向上的排队延迟也不一样，这些都可能导致网络中消息往返的测量时间不对称。由于这个原因，消息的往返时间（RTT）比单向传输时间更加有用。我们将在 13.12 节详细讨论消息传输时间，现在我们只需要注意超时时间必须比预期的 RTT 时间长。

图 13-11 展示了发送端和接收端按照停止并等待协议传输消息时，数据包与 ACK 包的传递过程。图中 RTT 是消息的往返时间，发送端必须在传输一个数据包之后等待一个 RTT 时间来接收 ACK，之后再准备发送下一个数据包，如此循环。

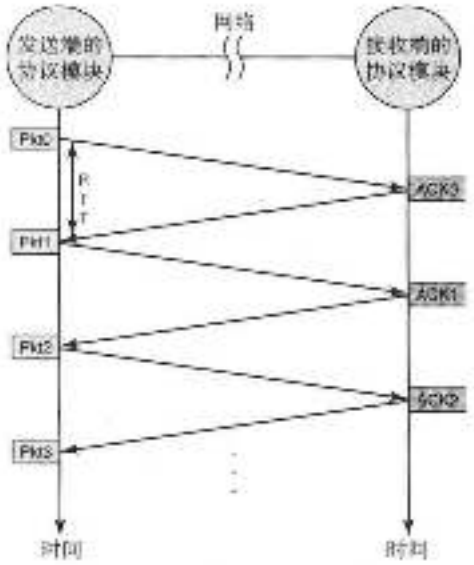


图 13-11 停止并等待协议发送数据包的时间轴。每个数据包都会被单独确认，发送端只有在收到前一个数据包的 ACK 后才会发送下一个数据包

例 13-2 一条消息有 10 个数据包，数据包的 RTT 时间是 2 毫秒（ms）。假设发送数据包和接收 ACK 的时间与介质中的传播时间相比可以忽略不计，且没有发生数据包丢失。那么停止并等待协议需要多少时间来完成数据传输？

答：

在本例中，RTT=2 ms。

因此，消息传输的总时间 =10×RTT

=10×2 ms

=20 ms

635 / 639

13.6.2 流水线协议

停止并等待协议的优点在于简单，但是如果你在因特网上下载电影，你一定不会乐意于使用这个协议。从图 13-11 可以看出，当发送端等待 ACK 到达时，网络有大量的时间处于空载状态。空载时间（dead time）定义为网络没有任何活动的时间。如果计算机有吉比特的网络连接，那么我们在网络空闲的每 1 秒内都可以发送 1 比特的数据。前面协议的缺陷是它们假设数据包丢失是一种常态，而不是一种例外，这个假定可能会导致网络带宽的利用率严重不足。例如，在例 13-2 中，RTT 是 2ms。换句话说，传输层每 2ms 发送一个数据包，得到传输层的吞吐量为 500 数据包 / 秒。如果每个数据包中有 1000 字节的有效数据，则传输层的吞吐量是 4 兆比特 / 秒。也就是说，我们只使用了可用网络带宽的 0.4%，吉比特网络连接的利用率严重不足。如果网络是可靠的（即没有数据包丢失），我们可以快速地连续发送消息的所有数据包，而不需要等待任何 ACK。

例如，如果网络是可靠的（没有数据包丢失），那么我们可以将数据包传输流水线化，不

需要等待 ACK，如图 13-12 所示。

这里区分开带宽（bandwidth）和传播时间（propagation time）是很重要的。带宽决定了主机将一个数据包传入线路所需要的时间。传播时间与端到端的延迟相关，它是数据传输途中的传播延迟与排队延迟的累积函数。我们将在 13.12 节重新考虑这个问题，并对这些术语给出更精确的定义。



图 13-12 没有 ACK 的流水线化数据包传输

例 13-2 一条链路为 10 个数据包，从源节点传送至目的地一个数据包的时间为 1ms。假设发送/接收数据包的时间与介质中的传播时间相比可以忽略不计，且没有发生数据包丢失。那么没有 ACK 的流水线协议需要多少时间来完成数据传输？

答：

这与前面的例子相似，但有一个区别：数据包是流水线的。如图 13-12 所示。产生数据包并将它们传入链路的时间可以忽略不计，因此所有数据包到达目的地所需的时间只是从源到目的地的端到端延迟。

640 传输总时间 =1 ms。

这个极端的例子虽然不切实际，但它显示了数据包流水线的重要性，特别是当从源到达目的地有巨大延迟时。图 13-13 形象地展示了停止并等待协议和流水线协议的区别。在图 13-13a 中，任意时刻都只有一个数据包处于传输过程中，而在图 13-13b 中，可以同时传输多个数据包。

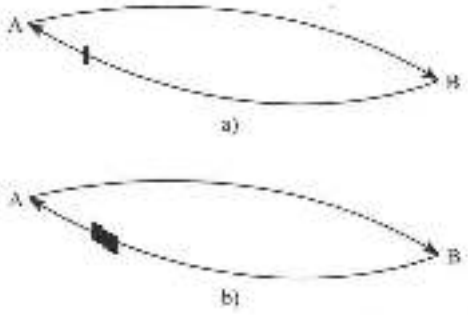


图 13-13 停止并等待传输和流水线传输的区别

图 13-12 引出了一个问题，我们真的需要 ACK 吗？这个问题的答案实际上取决于应用程序需要传输层提供的服务保障。例如，有些应用程序可能不需要可靠的传输。我们很快就会看到（在 13.6.5 节），UDP 是一种不使用 ACK 的传输协议，它就适用于这种应用。然而有些应用程序可能需要可靠的传输，就像使用邮政服务“挂号信”功能的人。对于这样的应用程
641 序，我们不能假定网络是一定可靠的，因此我们无法省去 ACK。

13.6.3 可靠的流水线协议

对极端的停止并等待协议和没有 ACK 的流水线协议的折中方法是将发送数据包和接收 ACK 流水线化。发送端在等待确认之前先发送一组数据包（称为一个窗口）。接收端与之前一样，对每个数据包单独进行确认。但好消息是，发送端不需要等到所有的数据包确认，就可以继续开始发送。图 13-14 形象地展示了这种情况。

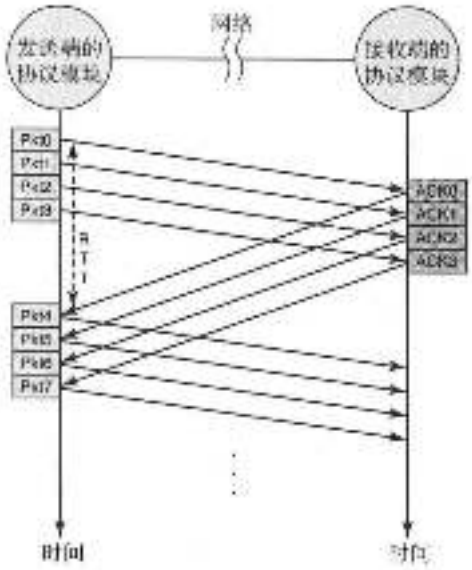


图 13-14 可靠的带 ACK 的流水线传输（窗口大小=4）。发送端在开始等待 ACK 之前会发送一个窗口的数据包

如图 13-14 所示，发送端（窗口大小为 4）发送 4 个数据包，并等待 ACK；当收到 ACK0 时，就能够发送 Pkt4。理想的情况下，没有数据包丢失，发送端一直重复这种循环。即发送端发送 Pkt4 ~ Pkt7 然后等待；当收到 ACK4 时，发送端会开始发送接下来的 4 个数据包 Pkt8 – Pkt11，以此类推，直到消息传输完毕。

当发送端收到一个数据包的 ACK 之后，那个数据包的传输就完成了。窗口的大小可以是由协议双方商定的参数，或者由发送端根据网络拥塞情况进行动态调整。

网络拥塞（network congestion） 让我们来了解究竟什么是网络拥塞，以及它为什么会发生。

打个简单的比方，考虑高速公路为什么会在每天的某些时段中堵车——例如，在高峰时期。通常会有许多支线公路在向高速公路传输流量，而在路途的另一端，人们会离开高速公路重新进入各条支线公路。此外，不同方向的高速公路可能会在城市中心合并。一种或多种原因会导致高速公路堵车。即使在高速公路饱和之后，也会不断有更多的汽车尝试从支线公路开上高速公路。在路途的另一端，希望下高速公路的汽车会因为支线公路容量有限（车道数量、速度限制、交通灯等）而无法离开。当两条高速公路合并时，车道的数量也会少于原有的总量。

网络拥塞也是由于几乎完全一样的原因。考虑下图的情况。



有 4 条 1Gbps（吉比特每秒）线路的网络流量进入可以支持高达 10Gbps 的线路。即使所有的 4 条线路都是满流量状态，10Gbps 的胖线路也能够满足它们的需求。但是，如果有 20 条这样的支路都进入这条胖线路，你立刻可以看到网络无法满足这样的速度需求。部分网络流量将开始堵塞。这就是网络拥塞的原因和表现形式。网络拥塞的结果是在路由器的数据包

队列堆积，使数据包在途中的路由器队列里等待，不能直接通过路由器。这最终将会导致不可预知的排队延迟，就像高速公路在高峰期堵车一样。之后如果路由器的硬件队列被填满，则路由器会因为缺乏缓冲区空间而简单地丢弃数据包，导致数据包丢失。

你可能会好奇为什么会设计出一个可能会拥塞的网络，答案是相当简单的。就像我们已经看到的，因特网是一个网络的网络。回到我们从亚特兰大向匹加罗尔发送消息的例子（见图 13-13），我们注意到这些消息会穿过许多不同网络的网络链路。在发送端你可能有一个吉比特网络连接，网络核心也可以满足许多吉比特链路的需求，但是位于班加罗尔的最终目的地可能只使用缓慢的拨号连接来连入因特网。这类似于较慢的支线公路与快速的高速公路。除非所有的端到端链路都拥有相同的带宽，否则我们无法避免网络阻塞。

642 ~ 643

在高速公路上遇到交通堵塞时，我们每个人都可能有独自的处理方式。从出口离开，喝一杯咖啡，做一会儿四肢伸展运动等。传输协议也会做类似的事情来应对网络拥塞。例如，它可能会根据观察到的网络拥塞状况，自我调节传送给网络层的数据量。随处可见的因特网传输协议 TCP，在这种情况下也会为了共同的利益而进行自我调节。其基本思想是，如果大家都行为良好，那么每个网络流都将根据带宽总量与当前的流量状况，公平地得到一份可用的网络带宽。换句话说，一个行为良好的传输协议会调节自己产生的网络负载，以确保在网络流量竞争中只使用了自己应有的那一份。

当然，这种方法也有其缺点。包含拥塞控制的协议不能保证数据能够尽快到达预期的接收端。例如，如果其他传输协议不遵循这样的自我调节机制，那么一个行为良好的协议作为一个好人，将会是最终的输家。你也看到过高速公路上那些自以为是的司机，一路上不给他人留余地，竭尽所能地超车。也就是说，使用行为良好的传输协议时，不存在明确的延迟上限，或者说不能保证有最低的传输速率。当你试图访问网络上的信息时，等待时间可能会有很大的差距。因为 Web 应用程序所使用的底层传输协议 TCP 是一个包含拥塞控制的行为良好的协议。由于这个原因，需要实时传输保障的网络应用程序（如视频和音频）可能会选择使用 UDP，并在其上自己提供可靠性。

滑动窗口（sliding window）包含拥塞控制的传输协议通过调节窗口大小来进行自律。窗口大小限制了数据的发送速率，进而减少路由器中堆积的队列，缓解网络拥塞。

我们已经知道，发送端将消息拆分成一组数据包，每个数据包都有唯一的序号。因此，对于一个给定的窗口大小，我们定义发送端不需要等待 ACK 就可以发送的这组数据包（与序号）为一个**活动窗口**（active window），如图 13-15 所示[⊖]。我们可能会问，什么决定了每个数据包的宽度（width）。宽度表示了发送端将一个数据包从计算机传入网络所需要的时间。我们可以简单地说，宽度是数据包大小与网络接口带宽比值的一阶近似。例如，如果你有一个吉比特/秒的全双工网络接口，数据包的大小是 1000 字节，则每个数据包的宽度是 8 微秒。宽度很重要，因为它能告诉我们在一个 RTT 时间内能发送多少数据包。换句话说，数据包的宽度为我们提供了窗口大小的上限，即在给定的 RTT 时间内所能发送的数据包数量。例如，如果 RTT 是 2 毫秒，则最大的窗口大小是 250 个数据包（每个数据包有 1000 字节），假设 ACK 包的宽度可以忽略不计。因为以下几种原因，实际选择的窗口大小可能会小于这个上限。这些原因包括网络拥塞（我们之前在本节讨论过）、发送端和接收端的缓冲区大小、包头中用于表示数据包序号的字段的大小。我们将在 13.12 节更详细地讨论消息传输时间。

644

⊖ 这张经过许可的图片改编自一张相似的图片，来自 Kurose 和 Ross 的书，《Computer Networking: A Top Down Approach Featuring the Internet》，Addison-Wesley [Kurose，2006]。

在图 13-15 中，只要收到活动窗口中的第一个红色数据包的 ACK，活动窗口就向右移动一步（第一个白色数据包变成蓝色数据包）。随着时间的推移，活动窗口会（从左至右）划过整个序号空间，因此我们将此称为滑动窗口协议（sliding window protocol）。讨论中的序号是单调递增的，但在实际实现中序号空间是循环的，会从 0 重新开始。

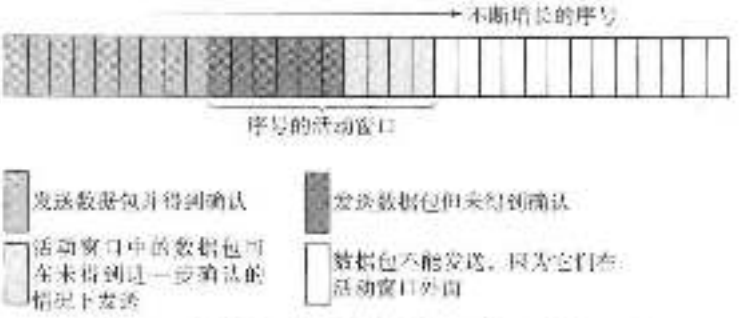


图 13-15 滑动窗口协议的活动窗口（窗口大小=10）

例 13-4 一条消息有 10 个数据包，已知 RTT 时间是 2 毫秒。假设发送数据包和接收 ACK 的时间与介质中的传播时间相比可以忽略不计，且没有发生数据包丢失。那么窗口大小为 5 的滑动窗口协议需要多少时间来完成数据传输？ 645

答：

下图展示了完成传输的时间轴：



在本例中，RTT=2 ms。发送端先发送一个 5 个数据包的窗口，然后开始等待 ACK。发送端会在发送第一个数据包的 2 毫秒后收到第一个 ACK，即在 2 毫秒的周期（RTT）内，发送端已经成功完成了这 5 个数据包的传输（由于我们忽略除介质中的传播时间以外的其他所有时间）。所以完成数据传输只需要两个这样的周期。消息传输的总时间 $=2\times RTT=2\times 2$ ms$=4$ ms。

我们可以看到，所选的窗口大小指定了任意时刻处于传输中的数据包的最大数量。例如，如果在例 13-4 中我们需要发送 12 个数据包，而不是 10 个，则需要 3 个周期来传输数据。最后 2 个数据包将在第 3 个周期中传送，完成消息传输。（我们会在习题中看到例 13-4 的变体。）

一种传输协议中经常使用的减少 ACK 包的优化是，累积数据包的确认，并发送累积 ACK。这个想法非常简单和直观。发送端在等待 ACK 之前发送出一个窗口的数据包，因此接收端会收到 n 个序号连续的数据包。我们可以只发送第 n 个数据包的 ACK，而不是为每个数据包发送一个 ACK。协议的语义允许这样的优化，因为收到第 n 个数据包的 ACK 就表示之前的 $n-1$ 个数据包已经成功送达。TCP 使用这样的累积 ACK，以减少网络传输的开销。

就像我们之前提到的，（数据或 ACK）数据包可能会丢失。在这种情况下，发送端和接收端都分别准备好重传丢失的数据包或 ACK 包。传输协议使用超时机制来发现丢失的数据包。

646 基本的思想是每端发送一个数据包，就设置一个计时器。例如，如果发送端在超时期限内没有收到数据包的 ACK，就将重传这个数据包。发送端自然需要缓冲还没有收到确认的数据包（图 13-15 中的红色部分）。

在设计滑动窗口协议时，有许多细节问题需要解决。我们已经提到了缓冲和超时重传。其他的细节问题包括：

- 选择合适的超时时间。
- 选择合适的窗口大小。
- 处理乱序到达的数据包。
- 决定什么时间对数据包进行确认，包括发送一组数据包的累积确认并移动活动窗口（参见本章末尾的练习 23）。

这些细节在我们的讨论范围之外，留给更深入的计算机网络课程。[1]

例 13-5 假设网络中平均每 5 个数据包会有 1 个丢失，对于一条包含 125 个数据包的消息，计算发送均要完成消息传输所需要发送的数据包总数。

答：

我们可以预计 125 个数据包会丢失 20%（1/5），即 25 个数据包。当我们重新发送这 25 个数据包时，预计会丢失 5 个数据包，以此类推。

发送的数据包	丢失	成功
125	25	100
25	5	20
5	1	4
1	0	1
156		125

完成消息传输所发送的数据包总数 =156。

13.6.4 处理传输错误

647 首先，接收端要能够检查数据包在传输过程中是否出错。因此发送端会依据数据包的实际内容计算一个校验和（checksum），并将其附加到数据包的末尾，以使接收端能够识别出错的数据包。校验和的计算既可以非常简单也可以非常复杂。例如，因特网中的校验和一般都是简单地计算数据字节的和（视为 16 位整数）。接收端对数据做同样的计算，并对比计算结果与数据包中的校验和，以识别出错的数据包。还可以使用纠错码（Error Correcting Codes，ECC）来检测数据包，并修复出错的数据包。这些讨论同样也留给更深入的计算机网络课程。无论如何，如果一个数据包损坏到无法修复，仍然等价于数据包丢失。传输层有识别错误并采取纠正措施的责任。前向纠错（Forward Error Correction，FEC）具有修复数据包的能力，但它并不能够保证修复成功。因此，传输协议必须要依靠超时重传来处理传输错误。

有趣的是，在本节讨论传输协议时，数据包的大小是唯一被使用的网络特定信息。换句话说，传输协议规范抽象出网络本身的详细信息。因此，协议栈中的传输层提供了在本节中说明的所有功能。

[1] 关于这些问题的详细讨论，参见 Kurose 和 Ross 的书，《Computer Networking: A Top Down Approach Featuring the Internet 3》，Addison-Wesley [Kurose，2006]。

13.6.5 因特网上的传输协议

因特网上使用的传输协议可以分为两大类：面向连接的（connection-oriented）和无连接的（connection-less）。TCP 是前者的例子，而 UDP 是后者的例子。

TCP TCP 需要首先建立端到端的连接。在连接建立后，两端之间的实际数据流是字节流（stream），即 TCP 处理字节流而不是消息。例如，假如你使用 Web 浏览器向 CNN 请求一个网页，Web 浏览器会与 CNN 的（或其代理）Web 服务器创建一个 TCP 连接。一旦连接成功，客户端会先发送一系列请求，服务器在响应后会回送网页中的一系列对象。对 TCP 传输来说客户端和服务器之间的请求和响应就是字节流。当整个网页传输完后，双方将拆除连接。

TCP 连接是一种全双工（full duplex）连接，即在连接建立后，两端可以同时发送和接收数据。虽然连接是由一端发起建立的，但建立的连接是对称的。TCP 主要提供了以下几种功能来保证两端之间的信息流传输：

- **建立连接**：在这个阶段，两端使用三次握手来协商传输的初始序号。
  - 客户端向服务器发送连接请求消息（有一个特殊字段表明它是连接请求），其中包含客户端计划在发送数据包时使用的初始序号。
  - 服务器发送对连接请求的确认消息（同样有一个特殊字段表明它是建立连接的三次握手的一部分），其中包含服务器计划在发送数据包时使用的初始序号。
  - 客户端分配资源（数据包窗口缓冲区、重传定时器等），并发送确认消息（这是三次握手中的最后一步）。当服务器收到此次确认后，就会为这次连接分配资源（数据包窗口缓冲区、重传定时器等）。

  此时，新建立的 TCP 连接的客户端与服务器已经准备好交换数据了。
- **可靠的数据传输**：在这个阶段，两端都可以发送和接收数据。协议保证从上层传递下来的数据会被如实地按序传递给接收端，没有任何的数据丢失或损坏。
- **拥塞控制**：在数据传输阶段，发送端也会通过观察网络拥塞状况来自行动态调整窗口大小以避免路由器队列堆积（从而缓解网络拥塞，详见 13.6.3 节）。因为这个原因，TCP 数据流在网络拥塞时可能会遭遇无上限的延迟。这会给需要保障的实时传输带来问题。尽管有这个内在问题，但由于它的普遍性和可靠性，TCP 也用于许多实时的数据流传输。
- **拆除连接**：在这个阶段，两端将按下列步骤断开连接。
  - 客户端向服务器发送拆除连接的请求（有一个特殊字段表明它是拆除连接请求）。服务器回复一个 ACK。
  - 服务器向客户端发送它自己的拆除连接请求（有一个特殊字段表明它是拆除连接请求）。客户端回复一个 ACK，并释放与此连接相关的客户端资源。当收到 ACK 后，服务器释放与此连接相关的服务器资源。连接正式关闭。

虽然在讨论中假定由客户端发起拆除，但无论客户端还是服务器都能够发起连接拆除。在连接拆除过程中，此连接不会传输新的数据。但是在连接关闭前，所有之前传输的数据确保已经被可靠交付。

UDP UDP 位于 IP 之上，为应用程序提供不可靠的数据报服务。正如我们之前所看到的，TCP 是面向流的，在实际的数据通信开始之前需要在两端之间使用复杂的握手来建立连接。同样，在通信完成之后也需要使用复杂的握手来关闭连接。此外，TCP 还拥有一些高级功能（如确认、滑动窗口和拥塞控制）来确保在广域网中的可靠传输，并坚持公平原则与其他用户共享可用带宽。这些高级功能给通信带来了开销。因此，可以在弱保障下充分发挥作

用的应用程序（如寄一张明信片）会使用 UDP，因为它更简单、速度更快。例如，对 IP 电话（Voice over IP，VoIP）来说，数据包的到达延时远比丢失几个数据包更为重要（由于实时限制，没有时间用于恢复丢失或损坏的数据包）。因此 UDP 越来越广泛地使用。当前估计 20% 的因特网流量都使用 UDP 协议。

当然，UDP 也有一些缺点：消息可能会乱序到达；消息可能会丢失；没有自我调节机制。所以 UDP 流量很可能是网络拥塞增长的源头。类似于 TCP，UDP 不提供任何保障（例如，延迟上限或传输速率下限）。表 13-1 总结了 TCP 和 UDP 的优点与缺点。

表 13-1 TCP 和 UDP 的比较

传输协议	特 点	优 点	缺 点
TCP	面向连接的；共享控制；基于数据流；支持可靠和 ACK	可靠；消息按顺序到达；行为良好，主动避免拥塞	建立和维护连接复杂；劣势发生变化；没有延迟或传输速率保障
UDP	无连接的；无拥塞控制；基于数据报；无可靠和 ACK	简单；快速；特别适合不易丢失的环境和容忍丢失的应用程序	不可靠；乱序到达；可能导致网络拥塞；没有延迟或传输速率保障

之前的讨论听起来是说，需要实时保障的网络应用程序既不能使用 UDP，也不能使用 TCP。事实上，这并不完全正确。实时应用程序的开发人员会考虑这些协议所拥有（或缺乏）的功能，以使用户体验不会受到负面影响。例如，音频或视频服务的应用程序会在开始播放之前就缓冲几分钟的内容。此外，它们还能够动态地增加缓冲区容量，以应对网络拥塞和满足应用程序指定的服务质量保障。表 13-2 列出了部分网络应用程序以及它们所使用的传输协议。

650

表 13-2 网络应用程序和传输协议

应用程序	关键需求	传输协议
Web 浏览器	可靠的消息传输；消息顺序到达	TCP
即时通信	可靠的消息传输；消息顺序到达	TCP
IP 电话	低延时	通常是 UDP
电子邮件	可靠的消息传输	TCP
文件传输	可靠的消息传输；消息顺序到达	TCP
网络视频	低延时	通常是 UDP；可能是 TCP
P2P 网络上的文件下载	可靠的消息传输；消息顺序到达	TCP
局域网网络文件服务	可靠的消息传输；消息顺序到达	TCP；或基于 UDP 的可靠传输
远程终端访问	可靠的消息传输；消息顺序到达	TCP

13.6.6 传输层总结

一般情况下，操作系统支持多个协议族，以满足不同应用程序的通信需求。在 20 世纪 80 年代早期，ISO 标准组织提出了一套新的传输协议，称作 OSI 传输层协议（ISO Transport Protocol，从 TP0 到 TP4）。作为网络通信的可能标准。然而，因为无处不在的 TCP，TP0 – TP4 从来没有被真正启用过。尽管当前有网络社区中有人担心 TCP 不再适合因特网传输，但已经有太多的网络应用程序使用 TCP 协议，它也很难被其他协议所替换。

但是，谁也不知道明天究竟会是什么样。例如，GENI（Global Environment for Network

Innovation，全球网络创新环境）和 PlanetLab[⊖]等全球网络基础研究项目可能会作为变革的推动者，为网络上不同类型的应用程序带来更新、更好的网络协议。

我们对于传输协议这个迷人的领域只给出了一份简要的介绍，你能够在更深入的计算机网络课程中学习到与这些协议相关的更多知识。

13.7 网络层

乍一看，网络层的作用似乎简单而直接——将从传输层传送来的数据包发送到目的地，并将从目的地传送来的数据包传送给传输层（上层协议栈）。这个功能也可以捆绑到传输层中，但这不会是一个好主意。第一，考虑你的笔记本电脑或家用计算机上有多少不同的网络连接。至少你可能会有一个有线连接和一个无线连接。因此，通常不同的目标主机需要通过不同的网络连接来访问。第二，看图 13-6，源与目的地可能不是直接相连的，因此数据包可能需要通过多跳才能到达目的地。这些网络的中间跳不需要提供传输层的功能，因为中间节点只是简单地向目的地转发数据包。第三，由于我们无法控制网络的变化，所以数据包从源到达目的主实际所经过的路线，也称作路由（route），是不固定的。结论是，应该由协议栈中一个不同的协议层来决策如何最好地将一个数据包传送到目标主机，我们把实现了这个功能的协议层称作网络层。这样的责任分离使传输层能够独立于任何网络连接添加/删除操作。传输层和网络层之间的接口确定目的地址和数据包大小等参数。网络层负责依据给定的目标地址来路由数据包，为此它会维护一张包含从源到任意期望的目标主机的路由或路径的表（称为路由表）。[⊖]当网络层从线路中收到一个数据包时，它向最终目的地转发该数据包，或者如果此节点是数据包的目的地，就将其传送给传输层。

下面是网络层所需要的功能：

- **路由算法**（routing algorithm）：网络层需要决定从源向目的地传输数据包的路由。用于确定路由的算法称为路由算法，这是网络层的主要功能。在本节中，我们将向读者介绍一些在因特网中广泛使用的著名路由算法。
- **服务模式**（service model）：网络层需要以现有的路由信息为基础，将从输入链路到达的数据包转发给合适的输出链路。这通常也称作网络层的交换（switching）功能。这个功能非常依赖于协议栈上层向网络层提供的服务模式。在本节中，我们将讨论网络中，尤其是因特网中，著名的交换策略和服务模式。

执行网络层功能的设备称为路由器。对于本节所讨论的路由算法来说，路由器与终端主机没有特殊的区别，唯一的区别是路由器知道它不是数据包的终端主机。路由器中的协议栈只包含了物理层、链路层和网络层。

13.7.1 路由算法

网络是主机和路由器的集合体，其中每台设备都有其独特的身份标识。在因特网世界中，身份标识是一个唯一的 IP 地址。如果网络是全连接的，假定在任意两个节点之间传输一个数据包的开销是完全一致的，则任意源的数据包都可以经由一跳路由到达任意目的地。然而，

⊖ PlanetLab 是一个实验性的网络测试床，拥有世界各地数百的网络节点。为了方便在广域网中进行受控实验而建立。参见 http://www.planet-lab.org/。

⊖ 在 13.7.4 节中，我们将看到网络层中还有另一张转发表（forwarding table），它包含了数据包向目标主机路由所必须经过的下一跳。

在现实中，(a) 网络不是全连接的；(b) 在任意两个节点之间传输一个数据包的开销可能是不一致的。让我们先了解这里所说的开销（cost）究竟是什么意思。归根结底，我们的目标是以最小的延迟将数据包从 A 点传输到 B 点。开销可以认为是一个汇总了两点之间的数据包传输延迟（这取决于连接的带宽）与网络流量的量化指标。拿开车上下班做比喻，上下班的开销（即行程时间）取决于途中的车速限制与交通流量。特别是，在上下班的高峰期开销会更高。有时选择一条较长的路线会更加划算，能够在较短的时间内到达目的地。

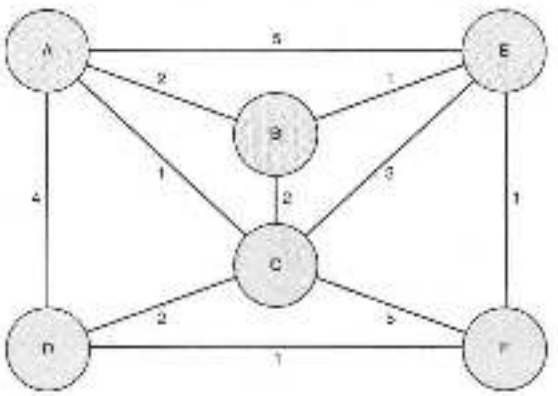


图 13-16 一个示例网络。顶点代表主机，边代表链路，边上数字表示主机之间的传输开销（延迟）

Dijkstra 链路状态路由算法 图 13-16 使用图来表示网络，图中每个顶点代表一台主机，而边代表主机之间有可用的物理链路。我们定义一个节点的链路状态为它到相邻节点的传输开销。例如，在图 13-16 中，节点 A 的链路状态为 {B:2，C:1，D:4，E:5}。即，从 A 到 B 有 2 个单位的传输开销，从 A 到 C 有 1 个单位的开销，从 A 到 D 有 4 个单位的开销，从 A 到 E 有 5 个单位的开销。同样，B 的链路状态为 {A:2，C:2，E:1}，以此类推。

（Dijkstra）链路状态（Link State，LS）路由算法是使用全局信息的分布算法。即网络中的所有节点都拥有完整的网络状态信息（例如，连通性和链路开销）。每个节点可以在本地运行此算法，根据全局信息来决定向目的地发送数据包的最佳路由。网络中的每个节点从它的相邻节点获取信息，因此每个节点周期性地将它的链路状态广播给相邻节点。

这个算法通过迭代寻找一个节点到网络中其他节点的最短路径。算法的每次迭代确定到一个节点的最短路由。因此，如果网络中共有 n 个节点，此算法需要进行 $n-1$ 次迭代来确定到所有其他节点的最短路由。

这个算法的原理非常简单。开始时，你知道从 A 到其相邻节点（B、C、D、E）的开销，其中 A–C 是开销最小的路由（1 个单位）。A–D 的直接链路有 4 个单位的开销，但是如果我们先通过开销最小的链路 A–C，则 A 到 D 只需要 3 个单位的开销（A–C，再 C–D）。

在算法开始执行时，我们知道 A 到其相邻节点的开销，每次迭代我们将增加一条从 A 到其他新节点的最短路由。例如，在第一次迭代中，我们依据 A 的链路状态确定了一条最短路是 A–C。在第二次迭代中，我们依据 A 与已经发现的最短路由来确定到达新节点的一条新的最短路由。算法一直进行迭代直到确定所有的最短路由。示例中有 6 个节点，因此算法需要执行 5 次迭代。

表 13-3 概述了图 13-16 中的示例执行算法所得到的结果。在表的每行中，第二列高亮的节点是此次迭代中确定的最短路由的终点。例如，在迭代 1 中最短路由为 A–C。粗体路由为由于新发现的最短路由而被更新的部分。例如，到 D、E、F 的路由会在迭代 1 更新，而到 B 的路由保持不变。

表 13-3 Dijkstra 算法对于图 13-16 中图的计算操作。在每次迭代中，我们根据开销来选择最好的下一跳[⊖]

迭代次数	已知最短路由的终点	B 开销 / 路由	C 开销 / 路由	D 开销 / 路由	E 开销 / 路由	F 开销 / 路由
开始	A	2/AB	1/AC	4/AD	5/AE	∞
1	AC	2/AB	1/AC √	3/ACD	4/ACE	6/ACF
2	ACB	2/AB √	√	3/ACD	3/ABE	6/ACF
3	ACBD	√	√	3/ACD √	3/ABE	4/ACDF
4	ACBDE	√	√	√	3/ABE √	4/ABEF
5	ACBDEF	√	√	√	√	4/ABEF √

后面是 Dijkstra 的 LS 路由算法的伪代码，我们使用图 13-16 中的信息来进行非形式化的描述。在此算法中我们使用了下列符号：

- R 表示从 A 开始的最短路由的已知节点集合。
- link-state（X: value）表示从 *A* 到 *X* 的直接链路开销。
- cost（*A*->*X*）表示 *A* 到 *X* 的路由开销。
- route（*A*->*X*）表示从 *A* 到 *X* 的路由，可能会经过算法已经发现的中间节点。

```
Init:
R = {A} // set of nodes for which route from A known
cost(A → X) = link-state(X: value) for all X adjacent to A
cost(A → X) = ∞ for all X not adjacent to A
// let n be the number of nodes in the network
for (i = 1 to n - 1) {
     choose node X not in R whose cost(A → X) is a minimum;
     add X to R;
     set route(A → X) as the least-cost route from A to X;
     update routes for nodes adjacent to X:
          for each Y not in R and adjacent to X {
               cost(A → Y) = MIN(original cost(A → Y),
                                 cost(A → X) + cost(X → Y));
               set route(A → Y); // only if new route
                                 // through X is lower cost
          }
}
```

链路状态算法也称为 Dijkstra 最短路径（shortest-path）算法。这个算法除了每个节点都需要拥有全局信息外，另一个问题是该算法假设网络中的所有节点能够同步执行算法，这样每个节点才能计算出相同的最短路径。在实际中，这个要求很难实现，各个路由器与各个主机会在不同的时间执行算法。这可能会导致路由决策中的某些不一致。尽管会出现这种暂时的不一致性，但有扩展算法能够保证网络稳定。这些扩展超出了本书的讨论范围。 653 / 655

距离矢量算法（distance vector algorithm）另一种在因特网中广泛使用的路由算法是距离矢量（DV）算法。这个算法是一个异步算法，且只需要网络链路状态的部分知识。由于因特网不断演变的性质，这两个属性使得距离矢量算法非常适合于因特网。

这个算法的基本思想很简单。无论最终目标是什么，任何节点都只需要决定它将数据包传输给哪一个有物理链路直接连接的相邻节点。对于相邻节点的选择非常简单，每次选择当前最短路由的后续节点。例如，参考图 13-16，如果 *E* 希望向 *D* 发送一个数据包，它会在它的所有邻居（*A*，*B*，*C*，*F*）之中选择 *F* 作为下一跳。为什么呢？因为如果我们直接看图 13-

⊖ 此表有下关键的错误 5/ADF-4/ACDF。——译者注

16，我们能够立刻发现 F 拥有到 D 的最短路由。值得注意的是，E 做出决定所需要的并不是到达 D 的实际路由，它只需要知道它的相邻节点到 D 所需要的开销。

每个节点维护一张路由表，称为距离矢量表。这张表记录了通过每一个物理连接的相邻节点，能够到达每个目的地的最短路由。距离矢量这个名字源于每个节点都有一个通过相邻节点到达目的地的开销矢量。表 13-4 是图 13-16 的示例网络中节点 E 的 DV 表。每行表示通过相邻节点到达一个特定目的地所需要的最短路由开销。DV 表中只记录开销，但为了方便展示，我们也在括号中补充了实际的路由。我们应该明确选择下一跳时并不需要实际的路由信息。到每个目的地的最短路由选择显示为灰色。

表 13-4 节点 E 的 DV 表。每行表示通过相邻节点到达一个特定目的地所需要的开销[⊖]

	通过相邻节点到达目的地的开销			
目的地	A	B	C	F
A	5 (EA)	3 (EBA)	4 (ECA)	5 (EFDCA)
B	7 (EAB)	1 (EB)	5 (ECB)	3 (EFEB)
C	6 (EAC)	3 (EBC)	2 (EC)	4 (EFDC)
D	8 (EACD)	4 (EBEFD)	5 (ECD)	2 (EFD)
F	9 (EABFF)	3 (EBEF)	6 (ECDF)	1 (EF)

我们对构建这张表的每个节点的 DV 算法只给出非形式化说明。每个节点将它到每个目的地的最短路由开销发送给其相邻节点，每个节点使用这个信息来更新它的 DV 表。当满足下列两个条件之一时，一个节点会重新计算它的 DV 表项。

656

1）节点观察到相邻节点的链路状态发生改变（例如，由于网络拥塞，从 E 到 B 的链路状态从 1 变成 5）。

2）节点从相邻节点收到了最短路由开销的更新。

在重新计算表项之后，如果有任何的变化，则该节点将这个改变告诉其相邻节点。DV 的数据结构和算法都非常简单明了。我们把编写 DV 算法的伪代码作为一道习题留给读者。

与 Dijkstra 链路状态算法相比，我们立刻就能够看到这个算法的异步特性，以及只使用局部信息的性质。网络的状态会持续地改变（例如，当前网络流量的传输模式、新的网络流量、增加 / 删除节点与路由器等）。我们可能会怀疑是否能够依据不断改变的当前网络状态来计算路由。幸运的是，这些算法具有良好的收敛性，能够确保计算路由的速度比网络状态的变化速度更快。

随着时间的推移，因特网上出现了更多的高效路由算法，但是 LS 算法和 DV 算法仍然在因特网路由中占据主导地位。

分层路由（hierarchical routing） 读者也许想知道 LS 算法或 DV 算法如何在因特网庞大的规模（超过百万个节点）与覆盖范围内进行路由。这两个算法都将因特网上的所有节点看成是对等节点（peer）。也就是说，网络这个单一实体中的所有节点都是平等的。这种扁平结构不能扩展到数百万的节点上。当任何组织的规模超过一定阈值后，就会使用分层结构来管理控制可能会出现的混乱，因特网也使用相同的原则。对于因特网庞大的规模，需要进行管理控制是使用分层结构令人信服的理由。特别是，在当今这个垃圾邮件不断增加的时代，部

⊖ 原表有 4 处不够准确结果的错误，已修正。3(BA) 应为 3(EBA)；6(EFDCB) 应为 3(EFEB)；2(EBEF) 应为 3(EBEF)；7(ECBEF) 应为 6(ECDF)。——译者注

分组织希望能够控制内部网络中的流量出入。将因特网中的路由器按区域划分为自治系统（Autonomous System，AS）能够很好地解决规模和管理控制这两个问题。在一个 AS 内的路由器可以使用 LS 或 DV 协议来为 AS 内的主机进行路由，其中的一个或多个路由器需要能够与 AS 外的目标通信，这些路由器称为网关路由器（gateway router）。不同 AS 的网关路由器使用边界网关协议（Border Gateway Protocol，BGP）进行通信。一个 AS 内的节点不会关心但不能影响到其他 AS 内节点的演变 / 增删 / 扩展。



图 13-17 网关节点中网络层的详细信息。它至少需要支持两种协议：一个用于 AS 内的路由，另一个用于跨 AS 的路由

因此，如图 13-17 所示，网关节点中的网络层至少需要支持两种协议：一个用于 AS 内通信，一个用于 AS 间通信。

参考图 13-18，设想连接到路由器 *C.1* 的主机需要与连接到路由器 *C.2* 的主机进行通信。在这种情况下，通信是通过自治系统 *C* 所使用的 AS 内协议（LS、DV 或其他变种）进行的。再设想连接到路由器 *A.4* 的主机需要向连接到路由器 *B.3* 的主机发送数据包。路由器 *A.4* 依据路由表查出要将需要送出自治系统 *A* 的数据包发送给 *A.G*。网关路由器 *A.G* 根据路由表知道，要到达自治系统 *B* 中的任何目的地，需要将数据包送到 *B* 的网关节点 *B.G*。网关路由器 *A.G* 使用 BGP 协议与 *B.G* 通信，再由 *B.G* 在自治系统 *B* 中使用 AS 内协议将数据包有效地路由到节点 *B.3*。如图 13-17 所示，路由器 *A.G*、*B.G*、*C.G1* 以及 *C.G2* 各自都有一个协议栈，使它们能够与各自的自治系统内的节点以及使用 BGP 协议的其他自治系统的网关节点通信。

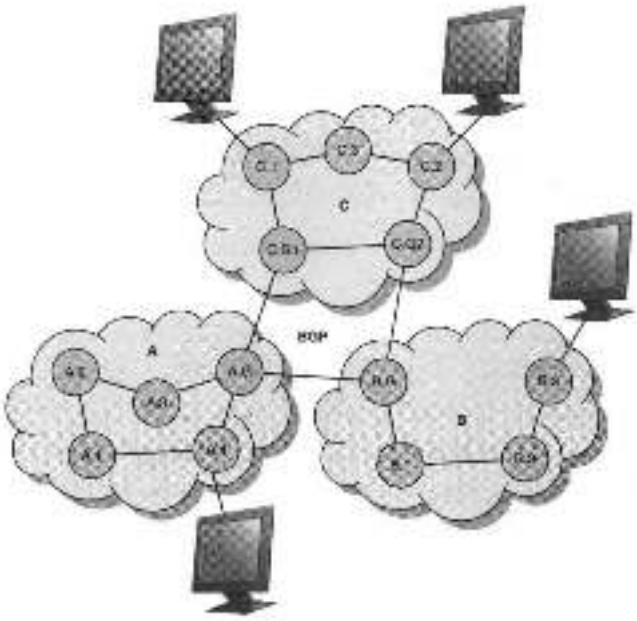


图 13-18 3 个不同的 AS 依靠 AS 内和 AS 间协议共存

注意，节点 *A.4* 的网络层完全不需要知道自治系统 *B* 的内部结构，甚至 *A* 的网关节点 *A.G* 也不知道自治系统 *B* 内部的 AS 内回设。

BGP 协议自身的细节超出了本书的范围。[8]

13.7.2 因特网寻址

到目前为止，我们使用节点这个术语来表示主机或者路由器。但是主机和路由器从根本上完全不一样。正如我们之前所提到的，主机是在网络边缘的终端设备，通常只通过网卡（NIC）连接到网络。而路由器（见图 13-18）允许与多个主机连接，它作为中继设备将每个连接发送来的消息路由到通往预期目的地的合适连接。通常，路由器的网卡（NIC）数量与它所能支持的连接数一样。处于网络边缘的主机同时是消息的生产者和消费者。因此，主机上的协议栈包含了 13.4.1 节所讨论的所有 5 个层，而路由器只包含了协议栈的底三层，因为它的预期功能是作为网络层级的数据包路由器。

857 ~ 859

让我们更深入地了解互特网寻址，以便理解主机如何获取网络地址，以及每个路由器所处理的网络地址数量。你已经听说过 IP 地址，我们很快就会看到，IP 寻址是一个为任何网络设备寻址的相当合乎逻辑的方式。考虑美国地面通信的电话号码。通常，一个城市的所有区域都拥有相同的区域代码（电话号码的前 3 个数字。例如，亚特兰大地区分配到的区域代码为 404、678 和 770）。随后的 3 个数字代表特定的区域或交互（例如，佐治亚理工学院分配到的交换代码是 894 和 385）。最后的 4 个数字表示特定的终端设备。

因此，电话号码是一个多部分地址（参见图 13-19）。因特网的地址也是这样。但是我们很快就会看到，不同于电话号码，IP 地址不具有地理意义。IP 地址的多部分性质实质上是一种支持网络分层寻址的机制。IP 地址（IPv4[9]）有 32 位。连接到因特网的每个接口都需要有一个全球唯一的 IP 地址。因此，如果你的笔记本电脑同时有因特网的无线连接和有线连接，它们会有各自独立的 IP 地址。同样，连接各部分独立网络的路由器的每个网络接口都有唯一的 IP 地址。32 位的 IP 地址由 4 部分组成，通常使用点分十进制（dotted decimal）表示法来表示，例如，*p.q.r.s*，其中 *p*、*q*、*r*、*s* 都是 8 位的数值。考虑 IP 地址 128.61.23.216，其中的每个数值都是相应的 8 位信息的十进制等效值。这个 IP 地址的 32 位二进制表示为

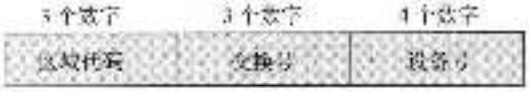


图 13-19 美国的电话号码

$$\underbrace{10000000}_{128}\ \underbrace{00111101}_{61}\ \underbrace{00010111}_{23}\ \underbrace{11011000}_{216}{}_2$$

32 位结构的 IP 地址主要用于因特网路由。IP 地址的最高几位构成了 IP 网络的代码。例如，地址的前 24 位可能用于指定一个 IP 网络，而后 8 位唯一地标识了该网络中的特定设备。习惯上，使用 *x.y.z.0/n* 来表示一个 IP 网络，其中 *n* 是 IP 地址中用于标识网络的位数。在此例中，IP 网络是 128.61.23.0/24，IP 地址的前 24 位构成了网络标识。图 13-20 展示了一个通过路由器连接到因特网的局域网（LAN）中的一些主机。在该图中有多少个 IP 网络？

660

[8] 有兴趣的读者可以参考《BGP4: Interdomain Routing in the Internet》，by J. Stewart, Addison-Wesley [Stewart, 1998]。

[9] 我们的讨论仅限于 IPv4。IPv6 使用 64 位寻址，用于解决 IPv4 的 32 位寻址局限性。但目前 IPv4 已经被广泛应用，IPv6 需要一段时间才能取代 IPv4 的位置。



图 13-20 IP 网络

答案是 2 个。局域网中的主机和与局域网连接的路由器中的接口都属于相同的 IP 网络，地址为 128.61.23.0/24。另一方面，路由器连接到因特网的另一个接口的地址为 128.61.22.4，属于不同的 IP 网络，地址为 128.61.22.0/24。

例 13-3 在图 13-21 中有多少个 IP 网络？假设 32 位 IP 地址中的前 24 位表示 IP 网络。

答：

图中共有 3 个 IP 网络：第一个 IP 网络连接了图中底部局域网中的 4 台主机和下方路由器（网络地址：128.61.23.0/24）。第二个 IP 网络将两个路由器连接到一起（网络地址：128.61.21.0/24）。第三个 IP 网络连接了顶部局域网中的 3 台主机和上方路由器（网络地址：128.61.22.0/24）。 661

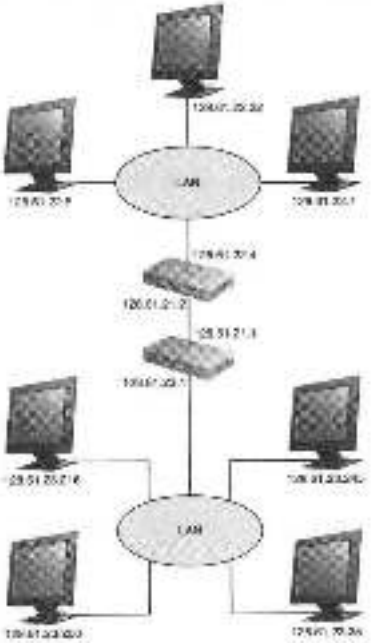


图 13-21 多个 IP 网段

因特网由上百万个IP网段组成，要知道IP地址中标识网络的位数不需要总是24位。假设你要创办一家公司，需要将1000台计算机连接到因特网。你需要向互联网服务提供商
662 （ISP）申请一个前22位固定的IP地址区间，IP地址的后10位能够允许1024（2^{10}）台计算机接入因特网。这样一个网络的网络地址有22位，点分十进制的形式为x.y.z.0/22。组织中所有主机的IP地址都有相同的前22位，只有后10位有所区别。类似于前面的例子，网络管理员可能会进一步地划分IP地址的后10位以创建子网。

13.7.3 网络服务模式

现在应该能够明确地知道，在大型网络中的数据包要经过多次中间跳才能从源到达最终目的地。例如，在图13-21中，考虑从左下角的主机（接口地址：128.61.23.200）向顶部的主机（接口地址：128.61.22.22）发送数据包。数据包在到达目的地之前需要进行3次网络跳（128.61.23.0/24、128.61.21.0/24和128.61.22.0/24）。

电路交换（circuit switching） 网络如何在终端主机之间快速地传输数据包？这个问题由网络服务模式回答。在我们开始讨论网络服务模式之前，我们应该先了解一些网络中的基础术语。让我们从电话网络开始。在Alexander Graham Bell（电话的发明归功于他）之后，我们已经走过了很长的路。通话不再是使用两个终端之间单一的物理线路，而是在呼叫过程中在两端之间进行一大堆电路切换。虽然技术已经发生了巨大的变化，但从早期的电话开始，其原理一直是相同的：逻辑上，当电话呼叫时在两端之间建立了一条专用线路，这称作电路交换（circuit switching）。这是至今为止电话中所使用的主要技术。设想Vasanthi要从美国佐治亚州亚特兰大市到印度泰米尔纳德邦Mailpatti去探望她的奶奶。作为一个傻执的人，她预订了整个的旅程，从亚特兰大机场的接送大巴到最后到达奶奶家的牛车。如果她没有在这个旅程中的某处出现，则那段已经被预订的位置就会被闲置。电路交换也正是这种情况。

电话呼叫的两端之间有许多交换机，在交换机之间有物理链路连接。图13-22展示了两个不同的电路（虚线）并存同一组物理链路（实线）和交换机上。一旦呼叫建立，在通话过程中这些网络资源（物理链路的带宽）将被保留。因此，电路交换能够保障服务质量，但也存在浪费网络资源的风险（例如，当电话里没人说话的沉默时间）。交换机之间的承载电话会话的物理链路可以同时支持多种信道（channel）或电路（circuit）。如频分复用（Frequency Division Multiplexing，FDM）和时分复用（Time Division Multiplexing，TDM）等技术允许多个并发连接共享物理链路。这些技术能够划分给定链路上的总可用带宽，建立专用信道来支持个人会话。如果会话数量达到了最大限制，那么在现有的会话完成之前，不会受理新的会话。这就是为什么你有时会听到录音，"非常抱歉，目前线路忙，请稍后再拨。"这些技术细节超出了
663 本书的讨论范围，但可以肯定地说，这些技术类似于高速公路中的多条车道。

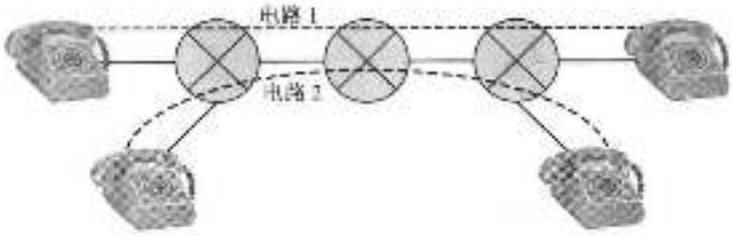


图13-22 电路交换。两个不同的电路（虚线）并存于同一组物理链路（实线）和交换机上

分组交换（packet switching） 电路交换的一种替代方法是分组（数据包）交换。设想Vasanthi第二次去她奶奶的村庄，这次她没有预订任何行程，而是在每个中转站让票务代理公司根据当前情况为她决定最好的行程。当她的下一段旅途没有空位时，她也可能不得不等待一会儿。分组交换的基本思想也与这种情况一样，不会在物理链路上预留带宽，当一个分组到达交换机时，交换机检查此分组的目的地，并将它发送到合适的链路上。图13-23展示了分组交换的概念图（在13.7节开始时介绍的路由器也是分组交换的一个例子）。它具有输入缓冲区和输出缓冲区，很快我们就将看到这些缓冲区的目的。

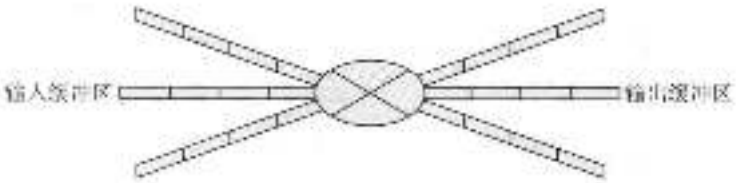


图13-23 分组交换的概念图。交换机的输入链路和输出链路都有相应的缓冲区，以应对突发的网络流量和物理链路竞争

分组交换网络也称为存储转发（store and forward）网络。在整个分组到达交换机之前，交换机无法向外发送这个分组。在分组交换网络中，这称为存储转发延迟（store and forward delay）。一个交换机可以有多条物理链路，每条输入或输出物理链路都有其相应的缓冲区。输入缓冲区用于接收分组到达的位。一旦整个分组都已经到达，交换机就准备好把分组发送到输出链路。分组交换机检查目的地址，并根据路由表中的路由信息，将分组发送到合适的输出链路上。但是输出链路可能仍然忙于传输前一个分组，在这种情况下，交换机就会将分组放入那条链路所对应的输出缓冲区中。因此，在实际将分组发送到输出链路之前可能会有一些延迟。这在分组交换网络中称作排队延迟（queuing delay）。你能够想象，这种延迟取决于网络拥塞的状况，是可以变化的。由于输入/输出可用的缓冲区的大小是固定的，所以当新的分组到达时，缓冲区（无论是输入还是输出）可能已经用完了。在这种情况下，交换机可能不得不丢弃一个分组（队列中的一个分组或者刚到达的分组，具体取决于网络的服务模式）。这就是我们在之前（参见13.3节）所提到的数据包丢失（packet loss）的原因。图13-24展示了分组交换网络中一条消息的分组流动。该图应该能让读者联想起第5章中所讨论的流水线指令执行。当消息的分组充满这条流水线时，每个交换机都在同时传输着不同的分组。

664

报文交换（message switching） 分组交换网络假设协议栈的高层（例如，传输层，参见13.6节）负责将消息划分成分组并收集分组构成原始消息，以及处理分组的乱序到达问题。但把这个功能开入网络自身也是可能的。这样的网络称作报文交换网络。如图13-25所示，在这种情况下，交换机对一条完整报文进行存储和转发，而不是单个分组。应该看到，分组交换网络对于单条报文能够有更短的延迟（通过流水线化的分组传输；比较图13-24和图13-25）。而且，如果在传输过程中出现位错误（这可能由于携带数据的物理链路中的各种电气和机电原因），其影响仅局限于单个分组，而不会影响整条报文。读者可能已经推测到，与报文交换相比，分组交换会累积更高的头部开销，因为每个分组都需要被单独寻址、路由和完整性检查。应当指出，报文交换网络也可以进行流水线传输。不过是对报文进行流水线化，而不是一条消息中的分组。

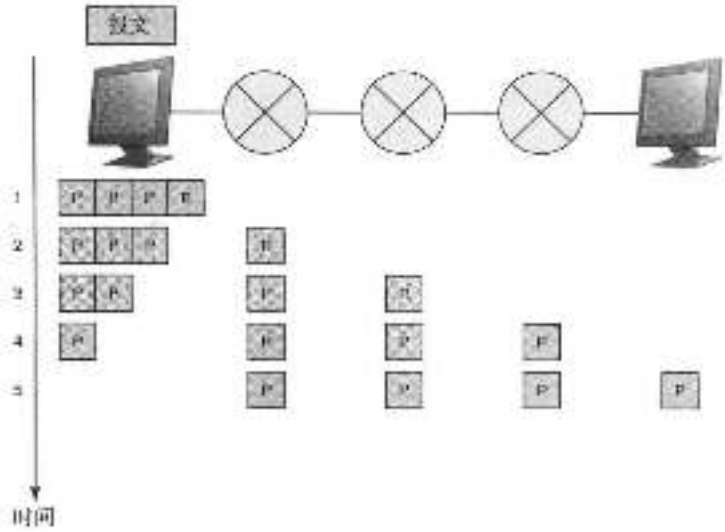
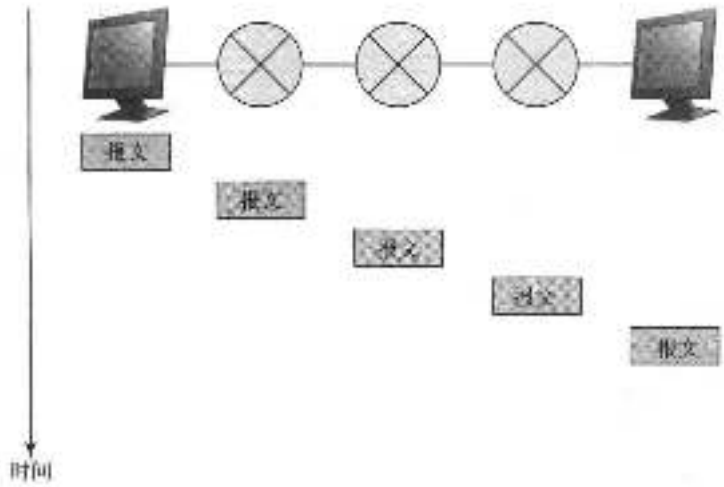


图 13-24 一个分组交换网络。在第一个时间点，发送端共有 4 个分组需要发送。在第 4 个时间点，每个中继交换机都在处理不同的分组。在第 5 个时间点，终点的第一个分组到达了目的地。假设没有数据包丢失，之后消息的后续分组将在连续的时间点到达



665
666

图 13-25 报文交换网络。每个交换机都需要等到报文完整到达后才能将报文发送给下一跳

分组交换网络的服务模式 分组交换提供了以下几个优点来避免竞争，尤其是对于计算机网络：

- 分组交换网络能够允分利用物理链路的可用带宽，而不像电路交换那样提前预留带宽（使用 FDM 或者 TDM）。这就是为什么电话网络中昂贵的国际电话也开始采用分组交换了。
- 因为预留机制，电路交换能够保障两端之间的传输时间，这对于语音流量是特别重要的。但是，这对于当前在因特网中占据主导地位的数据流量不是那么重要。随着网络技术的进步，甚至需要服务质量保障的应用程序（例如，通过因特网提供电话服务的

VoIP）也能在分组交换网络中很好地工作。

- 分组交换网络比电路交换网络更容易设计和运行。
- 正如我们之前观察到的，报文交换中的报文比分组交换中的分组粒度大，可能无法像分组交换那样有效地使用网络。此外，在分组级来处理传输中的位错误可以使网络的整体效率更高。

由于之前提到的所有这些原因，因特网在网络层使用分组交换，IP 是因特网中无处不在的网络层协议。

在分组交换网络中[⊖]，有两种网络服务模式：数据报（datagram）和虚电路（virtual circuit）。数据报服务模式类似于邮政服务，每个数据包（分组）都会包含目标地址，途中的路由器通过查看地址来决定合理的路由。路由器使用我们之前讨论的路由算法来构建路由表并不断更新，并依据路由表来决策。总体来说，因特网支持数据报服务模式。

虚电路类似于电话呼叫。我们已经了解了电路交换，其中为每个独立的会话分配专用的物理资源（链路带宽）。虚电路也是相似的，但不会预留物理资源。其想法是在呼叫建立（call setup）阶段建立一条从源到目的地的路由（称为虚电路）。源得到一个用于在会话过程中发送数据包的虚电路号。途中的路由器会维护一张表（称为 VC 表），其中包含了路由器当前处理的与虚电路有关的信息，即输入链路（incoming link）和输出链路（outgoing link）。因此，交换机决定路由的算法非常简单，检查输入数据包中的 VC 号，查询 VC 表，并将数据包放入相应输出链路的输出缓冲区中。最后当消息传输完成后，源的网络层执行呼叫拆除（call teardown），删除途中所有交换机中的相应表项。我们故意简化了对虚电路的讨论，以使读者不会混淆。实际上，在连接建立阶段，每个交换机都能够对新的连接选择一个局部 VC 号（为了简化网络管理），所以 VC 表变得有一点儿复杂：输入数据包的 VC 号（由前一个交换机选择）对应输入链路，输出数据包的 VC 号（由此交换机选择）对应输出链路。网络层协议中支持虚电路的例子包括 ATM 和 X.25。因特网 IP 协议只支持数据报服务模式。

667

13.7.4 网络路由与转发

我们要明确区分路由和转发，做一个简单的比喻将会有所帮助。考虑每天开车去工作，路由类似于决定从家到工作的路线，而转发类似于按照选择的路线开车。我们只会偶尔进行一次路线选择，但每天都会开车。[⊖]

除了网络层提供的服务模式处，路由和转发也都是网络层的功能。我们不希望读者产生网络层每次从传输层收到一个数据包都需要计算一次路由的印象。正如我们在 13.7 节开始时所提到的，当传输层发送来一个数据包时，网络层决定这个数据包所要经过的下一跳，这是网络层的转发功能。网络层维护一张转发表（forwarding table）来实现这个功能，给定目的 IP 地址，就能计算下一跳的 IP 地址。转发表位于因特网协议栈中的网络层。

上述讨论引出了新的问题，“路由表在哪里以及如何计算路由？”在第 8 章中，我们讨论了操作系统中执行某些特定功能的后台守护进程。例如，分页守护进程执行页面替换算法以确保虚拟内存系统在遇到页面错误后有可用的空闲页面帧池。基本目的是确保操作系统中的这些簿记（bookkeeping）操作不在程序执行的关键路径中。

⊖ 在网络服务模式的讨论中，我们将不区分报文交换和分组交换。因为报文交换也可以认为是一种特殊的分组交换，其中的分组就是整条报文。

⊖ 感谢我的同事 Constantine Dovrolis 提供了这个非常有启发性的例子。

计算网络路由也是操作系统的后台活动。在 UNIX 操作系统中，你可能会看到一个名叫 routed 的守护进程（路由守护进程）。这个守护进程在后台定期执行，使用类似于本节中所讨论的路由算法来计算路由，并创建路由表。也就是说，计算路由不是通过网络传输数据的关键路径。当网络发生变化时，路由表被广播给网络中的其他节点，以便更新相应节点的路由表。路由守护进程使用新发现的路由来更新协议栈网络层中的转发表。

13.7.5 网络层总结

668 表 13-5 总结了所有到目前为止我们已经讨论过的与网络层功能相关的关键术语。

表 13-5 术语网络关键概要

网络术语	定义 / 使用
电路交换	电话中使用的网络层技术。在通话期间保留网络资源（两端之间所有链路的链路带宽），没有开认延迟或存储转发延迟
TDM	时分复用，电话中用于支持在一条物理链路中划分多个信道的技术
FDM	频分复用，也是电话中用于支持在一条物理链路中划分多个信道的技术
分组交换	在因特网中广泛使用的一种网络层技术。它支持尽力而为的分组传输，且不会在途中预留任何网络资源
报文交换	与分组交换类似，但是数据为整条报文（在传输层）而不是分组
交换机 / 路由器	一种支持网络层功能的设备。可能就是一台有数个网络接口和足够内存的计算机
输入缓冲区	交换机中各个输入链路的缓冲区，用于收集输入的数据包
输出缓冲区	交换机中各个输出链路的缓冲区，以应对链路繁忙的情况
路由表	交换机中根据输入数据包的目的地址给出下一个跳的表。网络层使用路由算法来计算表的初始值并周期性地更新
延迟	分组交换网络中的分组被经历的各种延迟
存储转发	数据包在完全到达交换机之前，在输入缓冲区中的等待时间
排队	数据包在发送到输出链路之前，在队列中的等待时间
数据包丢失	交换机因为输入或输出缓冲区已满而不得不丢弃数据包。同时也表明了特定路由的网络拥塞
服务模式	网络层与协议栈上层的约定。分组交换网络所使用的数据报和虚电路模式都提供了尽力而为的数据传输
虚电路（VC）	这种模式在源与目的地之间建立一条虚电路，数据包只需要使用虚电路号而不用包含目的地址。这能够简化交换机对输入数据包的路由决策
数据报	这种模式不需要建立或拆除连接。每个数据包都是独立的，交换机依据路由表中的信息提供尽力而为的服务模式

669

让我们以网络层与传输层的关系来结束网络层的讨论。值得注意的是，网络层完全对传输层隐藏了它所涉及的复杂性，这就是抽象的力量。与网络服务模式无关，传输层本身可以是无连接的（如 UDP），或者面向连接的（如 TCP）。然而在一般情况下，如果网络层支持虚电路，传输层也将是面向连接的。

13.8 链路层和局域网

到目前为止我们已经讨论了协议栈中对操作系统有影响的两层协议，让我们将注意力转移到对硬件有影响的链路层上。我们采用了自上而下的方法来说明协议栈，但在网络发展初期，事实上，是链路层使因特网成为家喻户晓的名字。

本质上，链路层负责获取用于传输的物理介质，并通过物理介质向目的主机发送数据包。但是，一点区别是，链路层处理帧（frames），而不是数据包。依据链路层的技术细节，一个网络层的数据包（例如，IP 协议所产生的）在链路层可能被划分为多个帧，以便传输到目的地。

根据控制物理介质的访问机制，链路层协议可以分为两大类[⊖]：随机访问（random access）和轮流访问（taking turns）。以太网（Ethernet）是前者的一个例子，而令牌环（token ring）是后者的一个例子。如今，以太网是链路层中最普遍的技术。首先将终端设备（主机）连接到局域网，然后再将局域网连接到广域网。

链路层协议中访问物理介质的这部分通常称为介质访问控制（Media Access and Control，MAC）层。

13.8.1 以太网

历史上，以太网是将计算机连接到一起的电缆，如图 13-26 所示。我们使用术语节点（node）来指代连接到网络的计算机或者主板。

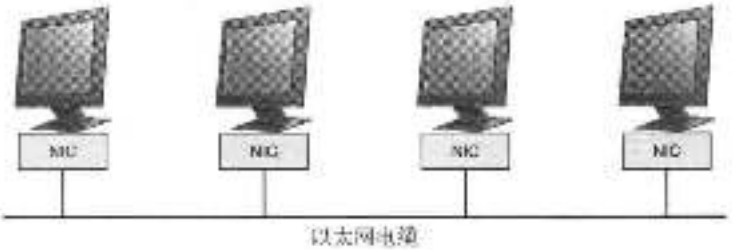


图 13-26 计算机通过以太网连成网络，这类似于连接计算机系统内部组件的总线

让我们来了解一下以太网协议。以太网电缆类似于一条总线。[⊖]然而与我们在前面章节中看到的不同，这条总线不局限于一台计算机，而是很可能贯穿整个建筑物。我们在第 10 章中提到了总线仲裁（bus arbitration），一个决定在竞争中的组件谁能够使用总线的方案。在一条连接计算机内部组件（处理器、内存、I/O 设备）的总线中，仲裁逻辑（arbitration logic）决定在需要同时访问总线的组件中谁能够控制总线。仲裁逻辑是实现总线仲裁方案的实际硬件。由于计算机内部的组件数量是有限且固定的，所以设计这样的仲裁逻辑是可行的。此外，在一台计算机内部，信号只需要传播很短的距离（最多几英尺）。另一方面，以太网要将一个几百米的办公环境中任意数量的设备连接到一起。因此以太网的设计者不得不考虑一些其他的仲裁方法来应对许多设备同时使用介质时所产生的竞争，并且同时要处理远距离传输和任意数量设备这两个问题。

13.8.2 CSMA/CD

一种随机访问通信协议称为载波监听多路访问 / 冲突检测（Carrier Sense Multiple Access/Collision Detect，CSMA/CD），它负责以太网等广播介质的基本仲裁。我们将继续简单而浅显的讨论，不会太多地深入数据传输背后通信原理的细节。以太网设计者采用了这个协议来应

⊖ 我们借用了这些术语，源自 Kurose 和 Ross 的书，《Computer Networking: A Top Down Approach Featuring the Internet》，Addison-Wesley [Kurose，2006]。

⊖ 我们将在 13.9 节中看到，现代以太网使用交换机以星型拓扑结构。将以太网比作总线这种观点对于理解以太网协议的细节是非常有用的。

对远距离传输和任意数量设备这两个问题。CSMA/CD 的基本思想来自我们在会议桌上礼貌地与同事谈话的方式。我们会先确保周围没有人正在说话后，开始说话。如果两个以上的人同时想说点什么，我们会先闭嘴，等待没有其他人说话时再次尝试说话。

CSMA/CD 协议与这种谈话情形没有太大的不同。图 13-27 展示了计算机使用 CSMA/CD 发送一帧时的状态转移过程。

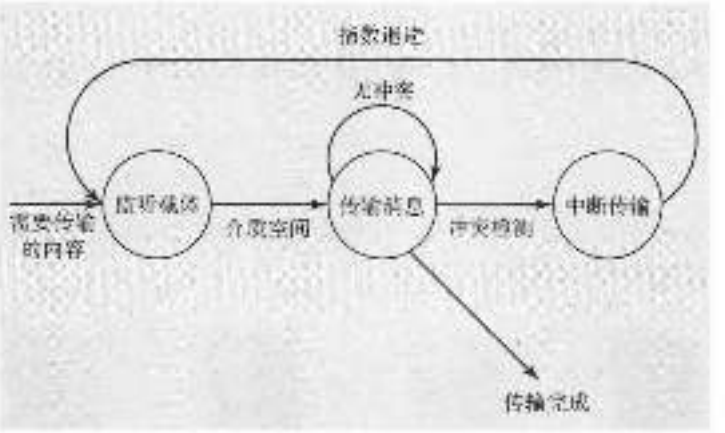


图 13-27 CSMA/CD 的状态转移过程

让我们更深入地了解 CSMA/CD 的协议名称和基本思想。

1）如果一个站点（station）（如计算机）想要通过介质（如电缆）传输数据，它会先监听是否有帧正在传输。如果有，该站点会等到介质空闲后再开始传输。如果介质空闲，则它可以立即开始传输帧。介质上没有任何电路活动就意味着空闲。

2）多个站点可能会同时监听到电缆空闲，它们都认为介质是空闲的，因此可能会同时开始进行帧传输，协议中的多路访问这个术语便源于此。这将造成问题，我们很快就会看到协议如何处理这个问题。

3）每个站点在开始传输帧后，会监听一次冲突。每个站点都能知道介质的当前电路活动情况。如果它观察到的（通过监听）与自己的行为不一致，它就会知道有其他站点也假定介质空闲。我们把协议中的这部分称为冲突检测（collision detection）。站点会立刻中止传输并发出一段噪声脉冲（noise burst）。（你可以认为这是有时能在收音机中听到的噪声，或者，继续以谈话做比喻，当多个人同时开始说话时，有人会礼貌地中断并说“对不起”。）噪声脉冲警告其他站点冲突已经发生，之后站点会等待一个随机的时间量，然后再次重复监听、传输、检测冲突这个循环，直到它成功地完成了传输。该算法使用随机数来决定站点在尝试重新传输之前的等待时间，随机范围随着冲突次数指数增长。因此，算法的这部分通常称为指数退避（exponential backoff）。

670
↓
672

我们定义冲突域（collision domain）为能够监听到彼此传输的一组计算机。[1]

让我们来了解帧究竟是如何传输的，以及如何检测介质是否空闲。协议使用基带信号（base band signaling），即在介质（如电缆）上直接用 0 和 1 的数字信号来传输帧。

[1] 以经典以太网比作总线的观点来看，连接到以太网的每台主机都是冲突域的一部分。但是，我们之后将在 13.9 节中看到，现代的网络设备将冲突域限制在连接到相同集线器（或者一组相互连接的集线器）的一对主机上。

宽带（broadband）是指在相同的介质上同时进行多个消息的模拟（analog）传输，不同的服务使用不同的频率同时发送它们的消息内容。例如，你家里的电缆服务可能就是宽带，或许在这一条线中同时承载了电视信号、电话服务以及因特网连接。

13.8.3 IEEE 802.3

以太网使用由 IEEE 制定的 IEEE 802.3 标准，采用 CSMA/CD 协议。在这个标准中，数字帧传输使用曼彻斯特码（Manchester code），即一种特定类型的位编码技术（见图 13-28）。这种编码，从低跳到高表示 0，从高跳到低表示 1。在 IEEE 802.3 标准中，低电平是 -0.85V，高电平是 +0.85V，空闲时是 0V。每位的传输都占用一个固定的时间量，曼彻斯特编码技术确保在每位传输的中间都有一个电压跳变，从而使发送站点和接收站点能够同步。因此，当线路中有帧传输时，就总会存在电路活动。如果在一位的传输时间中没有电压跳变，站点就会假定介质空闲。由于以太网采用基带信号技术，所以介质上每次只能有 1 帧在传输。

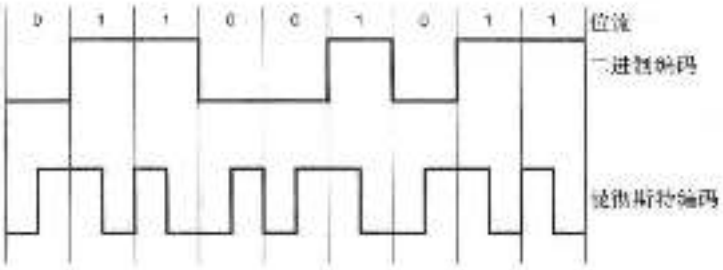


图 13-28 曼彻斯特编码。从低跳到高表示 0，从高跳到低表示 1。也就是说，当线路中有数据传输时总存在着电压波动

应当提到的是，早期的以太网使用 CSMA/CD，而现在所使用的大多数以太网 LAN 都是没有冲突的交换式以太网（参见 13.9 节）。同时也注意到，有趣的是 10 吉比特（10-gigabit）以太网甚至不支持 CSMA/CD，因为它假定主机通过交换链路连接到网络。然而，以太网作为链路层协议，为了兼容性会继续使用相同的传输格式（如帧头）。 673

练习 13-7 下面经过曼彻斯特编码的数据流所代表的位流是什么？



答：

0 1 0 1 1 0 0 1 1

13.8.4 无线局域网与 IEEE 802.11

与有线网络相比，无线网络又带来了新的挑战。为了区分，我们应该提到一个用于无线局域网的 CSMA 协议的变种，即 CSMA/CA，其中 CA 表示冲突避免（Collision Avoidance）。这个 CSMA 变种用于站点不能确定介质上是否有冲突的情形。有两个原因可能会导致冲突检

提出到问题，第一个原因非常简单，就是执行效率。冲突检测假定站点能够同时进行发送和接收，这样才能验证传输是否被其他站点干扰。对于有线介质，在网络接口中实现这样的检测能力在经济上是可行的，但对于无线介质却不可行的。

第二个原因更加有趣，即隐藏终端（hidden terminal）问题。设想有 3 个人沿着一条长走廊站着（图 13-29 中的 Joe、Cindy 和 Bala）。Cindy 能听到 Joe 和 Bala 谈话，但 Joe 和 Bala 都只能听到 Cindy 说话。Joe 和 Bala 可能会同时尝试与 Cindy 交谈，因此在 Cindy 处会有冲突，但是 Bala 和 Joe 都不会认识到这一点。这就是隐藏终端问题：对于 Bala 来说，Joe 是隐藏的，反之亦然。



图 13-29 隐藏终端问题。Joe 和 Bala 相互听不见，但当他们同时与 Cindy 说话时，Cindy 会听到混乱的字句

避免冲突的一种方法是，发送端通过向目的地发送一条很短的请求发送（Request To Send，RTS）帧，明确地与目的地请求发送许可。目的地（假设这个 RTS 帧没有受到干扰成功到达目的地）回复一个允许发送（Clear To Send，CTS）帧。当源收到 CTS 后，就向目的地发送数据帧。当然，不同节点的 RTS 帧之间可能会发生冲突，但幸运的是，这些都是很短的数据包，因此不会造成很大的伤害。想要传输的节点也能够很快地得到传输许可。在局域网中的所有节点都能够监听到 RTS/CTS，因此它们在数据传输完成之前不会尝试发送 RTS，从而确保不会发生冲突。

RTS-CTS 握手解决了隐藏终端问题并避免了冲突。

IEEE 802.11 RTS-CTS 标准是在无线局域网中的使用 RTS-CTS 的 CSMA/CA 协议的实现规范。

13.8.5 令牌环

正如我们在 13.8 节开头所提到的，以太网等随机访问协议的替代方案是各个站点轮流传输帧。一个简单的想法就是轮询（polling）。主节点按照预定的顺序轮询各个站点，看它是否需要传输。当从主节点获得允许后，从节点就开始传输消息。

令牌环提供了一种分散的轮流方式，而不是集中管理。如图 13-30 所示，其基本思想是将网络中的节点连成成环状。

令牌（token）是一个持续在线路中循环的特殊位模式。想要发送帧的节点会等待并获取令牌，然后把它要发送的帧放入线路中，最后将令牌放回线

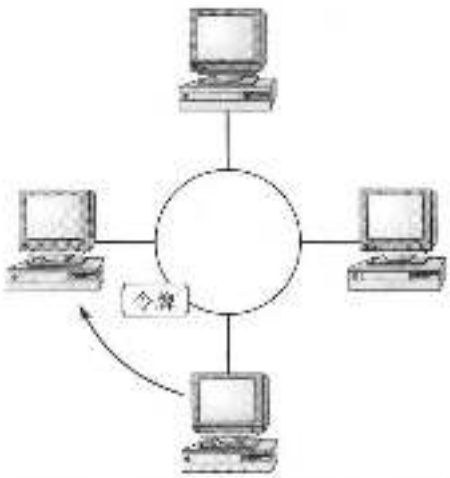
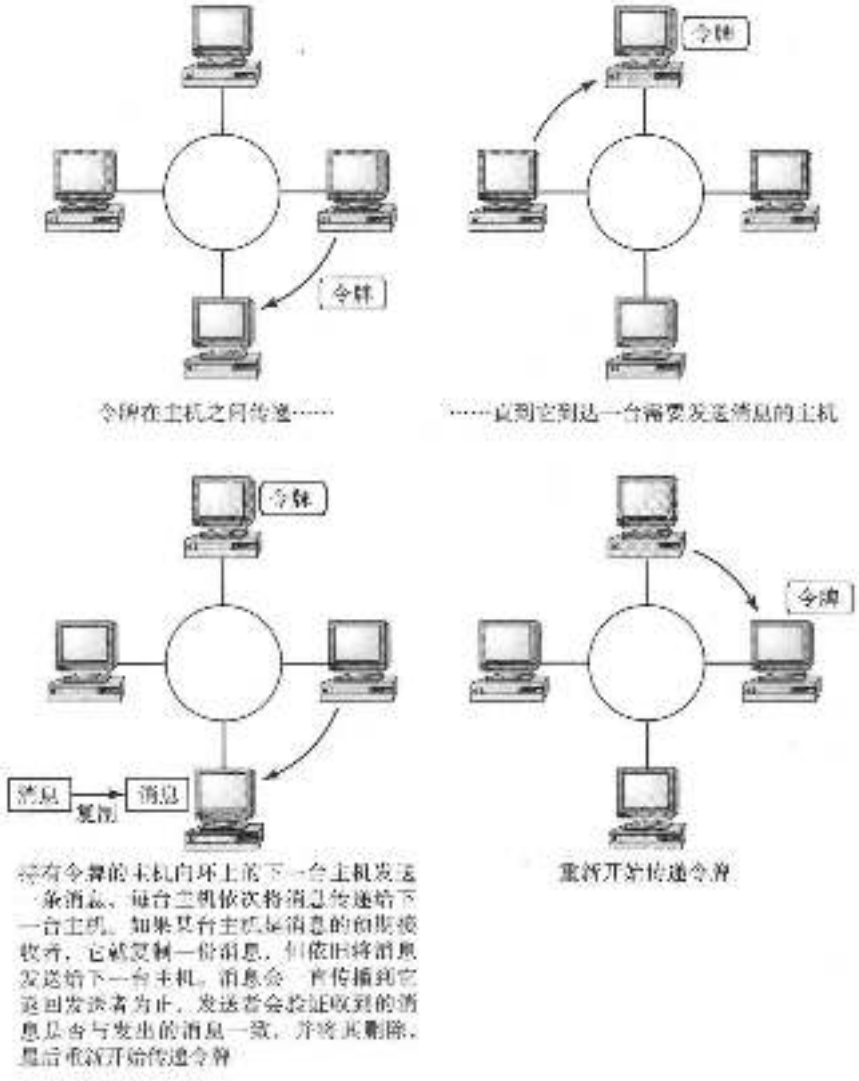


图 13-30 令牌环网络。令牌会在主机之间不断地被传递。只有获取令牌的主机才能进行传输

路中。每个节点都会检查帧的帧头，如果发现它就是该帧的目的地，就获取该帧。需要有人移除帧并重新生成令牌，通常情况下，由该帧的发送者负责从线路中移除数据帧并重新生成令牌。大体上令牌环也与广播介质类似，因为帧会经过环上的所有节点。但是，如果令牌环网络跨越了辽阔的区域，则由目的地节点负责移除数据帧并将令牌放回线路更为合理。

图 13-31 展示了在令牌环中传输帧的步骤。令牌环在设计上不会发生冲突，但也有自身的缺陷。一个缺点是，节点必须要等到令牌才能发送帧，如果在有大量节点的局域网中，这可能会导致相当大的帧传输延迟。另一个缺点是，如果有一个节点没有响应，局域网的环就断开了。同样，如果环上的令牌由于某种原因丢失或损坏，局域网就无法正常工作了。当然，我们能够灵活地解决所有这些问题，但是帧传输的延迟依旧是这种技术的严重局限。同样，令牌环也有它的优点。在高负载下以太网就会达到饱和，但由于高负载时存在过多的冲突，利用率永远不会达到 100%。而令牌环在重负载状态下工作良好。表 13-6 给出了两种局域网协议的比较。



令牌在主机之间传递……

……直到它到达一台需要发送消息的主机

持有令牌的主机向环上的下一台主机发送一条消息。每台主机依次将消息传递给下一台主机。如果某台主机复制消息的地址较对，它就复制一份消息，但依旧将消息发送给下一台主机。消息会一直传播到它返回发送者为止，发送者会校验收到的消息是否与发出的消息一致，并将其删除，最后重新开始传递令牌

重新开始传递令牌

图 13-31 令牌环发送一帧的步骤

表 13-5 以太网和令牌环的比较

链路层协议	特 点	优 点	缺 点
以太网	随机访问；使用有随机性的 CSMA/CD；指数退避	管理简单；在负载轻时工作良好	在高负载下冲突过多
令牌环	轮流访问；需要传输令牌	公平访问；在重负载下工作良好	在轻负载时获取令牌会有多余的延迟

13.8.6 其他链路层协议

让我们从局域网技术的角度来结束本节。以太网和令牌环都是出现于 20 世纪 80 年代末与 90 年代初的链路层技术。然而，由于各种原因，以太网已经成为局域网技术中的赢家。虽然我们已经学习了随机访问协议中的以太网和轮流访问协议中的令牌环，但值得一提的是，还有一些其他的链路层协议。它们中的一些有着优于以太网的性能，并在首次推出时做出了惊人的承诺。光纤分布式数据接口（Fiber Distributed Data Interface，FDDI）和异步传输模式（Asynchronous Transfer Mode，ATM）就是两个这样的例子。FDDI 最初设想用于光纤物理介质，被认为特别适合在大学校园或大公司等大型组织中作为一个高带宽的骨干网络，将一些基于以太网的局域网岛屿连接在一起。它类似于令牌环，也是轮流访问协议的一个变种。ATM 通过预留链路带宽和进行接入控制，能够提供服务质量保障并防止网络拥塞。它是一个提供了许多网络层功能的面向连接的链路层协议，因此 ATM 也简化了直接在它之上的传输层协议的实现。ATM 存在于一些由电信服务提供商所使用的城域网（MAN）或广域网（WAN）中。但是与以太网相比，ATM 过于复杂，不适合在局域网中使用。

当今另一个被广泛采用的链路层技术是点对点协议（Point to Point Protocol，PPP）。PPP 是拨号连接所使用的链路层协议，它的广泛应用源自大量用户群体所使用的拨号连接。

有趣的是，以往每次有成熟存在时，以太网都能找到了一种方法改头换面并占据上风。事实上，以太网已经不仅是局域网所选择的网络技术。吉比特以太网出现之后，就在大型组织中从 FDDI 处抢占了连接局域网岛屿的风头。10 吉比特（10-gigabit）以太网已经出现并成熟，且开始在城域网中使用，这使得 ATM 已经快要灭绝。

13.9 网络硬件

674，678

这里我们不会去深入了解物理层的详细信息，如电路、无线电以及光纤的光学性质。有兴趣的读者可以从其他来源获取这些信息（例如，[Kurose，2006；Tanenbaum，2002]）。我们将了解当前在基于以太网的局域网中普遍使用的网络设备。

1. 中继器

电信号的信号强度会随距离衰减。这是由于承载信号的物理线路中的电阻和电容有热损耗所导致。中继器（repeater）是能够将输入连接中的位信号放大，再传输给输出连接的电路设备。由于局域网能够跨越相当大的地理距离（例如，一栋大厦或一整个校园），因此需要定期增强信号强度。中继器通常用于局域网和广域网，以解决信号衰减问题。

2. 集线器

我们在 13.8.1 节中提到，以太网在逻辑上是一条总线（见图 13-26）。集线器（hub）本质上就是一个盒子里的以太网：从图 13-32 可以看出，图 13-26 中的以太网电缆被集中到一个集线器里。集线器将从一台主机收到的位传播给其他设备（构成一条逻辑总线的计算机和其他

集线器）。如图 13-33 所示，集线器使复杂局域网的构建变得相当简单，由相互连接的集线器所连接的逻辑总线上的所有主机都在同一个冲突域中。冲突域是指能够相互监听传输的一组计算机（参见 13.8.1 节）。因此，连接到集线器的计算机需要进行冲突检测，并按照以太网协议进行指数退避，以消除冲突并完成消息传输。集线器就是多端口的中继器，因此这两个术语经常被互换使用。

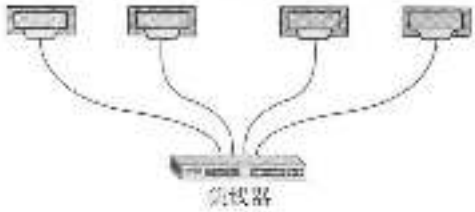


图 13-32 一个四端口集线器连接 4 台计算机的简单示例。在电路中集线器只不过是一条总线

3. 网桥和交换机

20 世纪 90 年代后期，局域网的发展过程中出现了另一个里程碑，即交换式以太网（switched Ethernet），使用网桥（bridge）和交换机（switch）将冲突域彼此分隔。例如，图 13-34 展示了使用网桥分隔两个冲突域。主机 1 和主机 2 构成了一个冲突域，主机 3 和主机 4 构成了另一个冲突域。主机 1 向主机 2 发送数据包不需要经过网桥，而如果目的地是主机 3 或主机 4 则需要经过网桥。这种流量的隔离允许主机 1 和主机 2 之间的通信与主机 3 和主机 4 之间的通信同时存在。

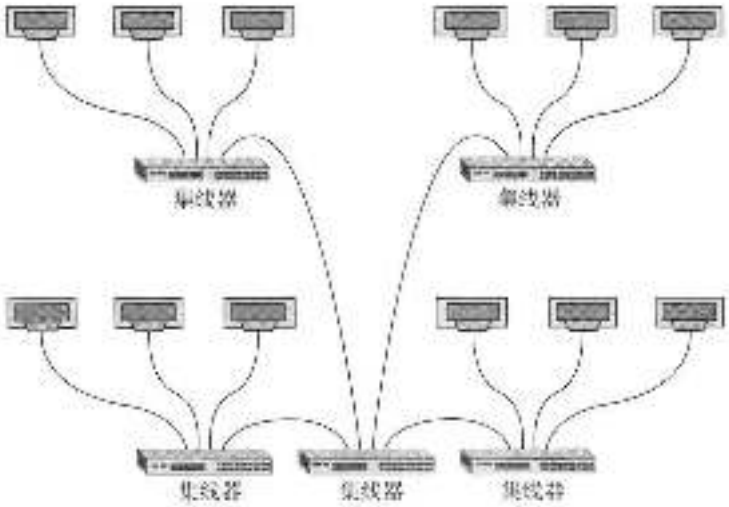


图 13-33 使用集线器来构建更复杂的网络。图中所有计算机都在相同的冲突域中

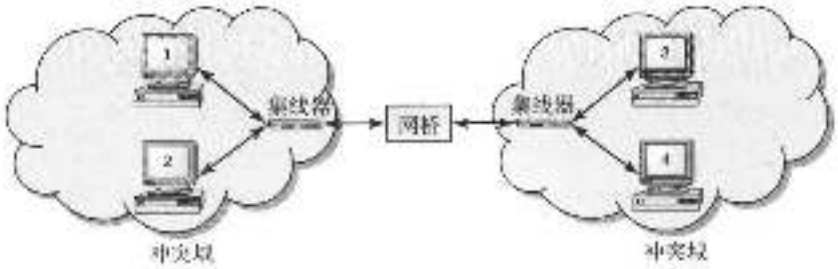


图 13-34 网桥将主机 1 和主机 2 的冲突域与主机 3 和主机 4 的冲突域分隔

当主机 1 与主机 4 同时想与主机 3 与主机 2 通信时会发生什么？网桥会发现冲突并让两边的流量依次通过（例如，先让 1 到 3 的流量通过，再在下一个网络周期中让 4 到 2 的流量

通过）。为此，网桥需要拥有足够的缓冲能力用于在冲突出现时保存数据包。因此，不在同一个冲突域的主机之间永远不会出现冲突。网桥的末端称为端口（port），它根据需要将数据包发送到其他端口。

虽然网桥和交换机经常在文献中交替出现，但有些作者将网桥定义为连接数量有限的（通常是2个）冲突域的设备，如图13-34所示。交换机在功能上是一种更广泛的网桥，支持任意数量的冲突域。图13-35展示了一个连接4个冲突域的交换机（A、B、C、D各代表了一个独立的冲突域）。

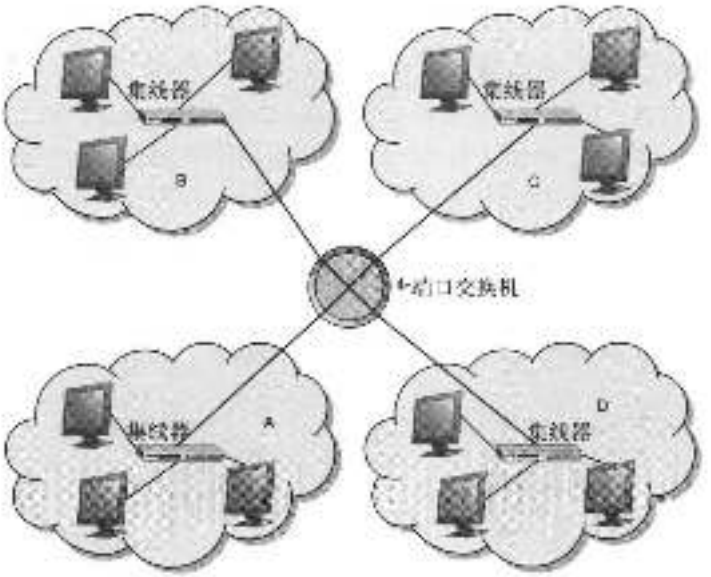


图13-35 四端口交换机的概念图。冲突域A中的主机能够在冲突域B与冲突域D有通信时，同时与冲突域C中的主机通信

如果我们将图13-33中的所有集线器换成交换机，我们就得到了一个没有冲突的交换式以太网。 681

4. 虚拟局域网

虚拟局域网（Virtual LAN，VLAN）是交换式以太网之后自然的下一步，利用如图13-36中的交换机实现。假设图13-36中的节点1和节点5要求处于相同的VLAN中；同样，节点2、节点6和节点7要求处于相同的VLAN中。如果节点2发送一条广播消息，则节点6和节点7就会接收到此消息，而其他节点不会收到这条消息。因此，利用分层交换机，不同地理位置（因此在不同的交换机上）的节点仍然能够构成一个广播域。

5. 网卡

网卡（Network Interface Card，NIC），又称网络适配器或网络接口卡，使计算机可以连接到网络，也能使主机连接到集线器、网桥或交换机。当连接到集线器时，网卡使用半双工模式进行通信（即在同一时间内，它可以发送或接收数据包，但两者不能同时进行）。智能网卡能够识别它是否与网桥连接，当它与网桥连接时可以使用全双工模式或者半双工模式进行通信。每块网卡都有一个介质访问控制（MAC）地址，网桥使用MAC地址来进行数据包转发。网桥自动获取连接到它网卡的MAC地址。

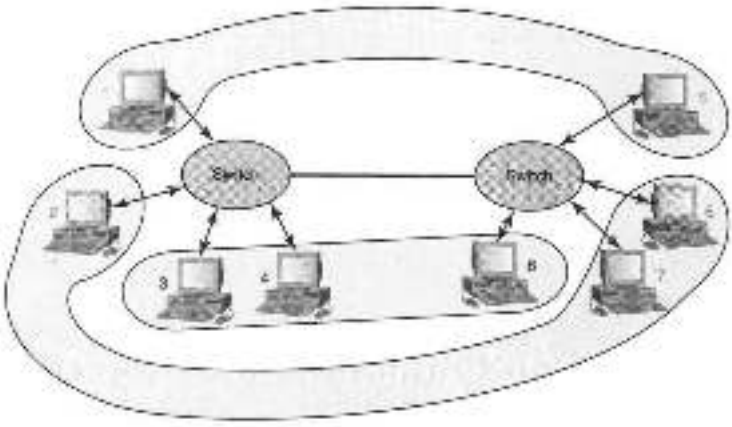


图 13-36 虚拟局域网（VLAN）：{1，5}、{2，6，7} 和 {3，4，8} 构成了 3 个 VLAN

网桥和交换机都只了解 MAC 地址，局域网中的数据流量完全以 MAC 地址为基础。数据包的结构如图 13-37 所示，有效载荷（payload）是指数据包中实际消息的那部分。包头（header）包含了目的节点的 MAC 地址。



图 13-37 发往局域网上节点的数据包。包头只含有目的 MAC 地址

若要向局域网外（即因特网）发送数据包，节点需要使用 IP 协议。正如我们之前在 13.7.2 节所看到的，发往因特网上节点的数据包含有 IP 地址，一个能够唯一标识目标节点的 32 位二进制数。

6. 路由器

我们在 13.7.1 节介绍了路由器这一概念，它是局域网中了解 IP 地址的主机。局域网中希望向因特网发送消息的主机会先构建一个数据包，如图 13-38 所示，并使用路由器的 MAC 地址将它发送给路由器。发送给路由器的有效载荷中包含了实际消息的 IP 头，其中含有目的节点的 IP 地址。正如我们之前在 13.7.1 节中所看到的，路由器使用路由表来帮助它路由实际的消息到达 IP 地址所标识的目的节点。



图 13-38 发往因特网上节点的数据包。目的 IP 地址是发送给路由器节点的有效载荷的一部分

表 13-7 总结了当前在搭建计算机网络时普遍使用的术语和设备。我们尝试将每个硬件设备对应到 OSI 模型（等价于因特网协议栈）的相应层。

表 13-7 网络组件概要

组件名称	定义 / 功能
主机	网络中的一台计算机；在计算机网络用语中也称为节点或站点
网卡	将计算机接入局域网的接口设备；对应于 OSI 模型中的第 2 层（数据链路层）
端口	中继器 / 集线器 / 交换机中用于连接计算机的末端；对应于 OSI 模型中的第 1 层（物理层）

（续）

组件名称	定义 / 功能
冲突域	用于表示在消息传输期间相互干扰的一组计算机
中继器	增强输入端口的信号强度并在输出端口加实地重新生成位流的设备；用于局域网和广域网；对应于 OSI 模型中的第 1 层（物理层）
集线器	作为多端口的中继器，将多台计算机连接到一起形成单一的冲突域；对应于 OSI 模型中的第 1 层（物理层）
网桥	连接并分隔独立的冲突域；通常有 2 ~ 4 个端口；使用 MAC 地址；对应于 OSI 模型中的第 2 层（数据链路层）
交换机	功能类似于网桥，但支持多个端口（通常 4 ~ 32）；对连接到交换式网络的计算机提供虚拟局域网的动态配置和分组等扩展功能；对应于 OSI 模型中的第 2 层（数据链路层）
路由器	本质上是一个交换机，但能从局域网向固特网路由消息；对应于 OSI 模型中的第 3 层（网络层）
虚拟局域网	现代交换机允许将物理上分散且连接到不同交换机的计算机进行分组，以便构建一个局域网；VLAN 独立于计算机的物理位置，提供更高级的网络服务，例如在网络网子网中进行广播和多播；对应于 OSI 模型中的第 2 层（数据链路层）

13.10 协议栈各层之间的关系

值得注意的是，传输层、网络层和链路层会在不同的层上处理数据的完整性。例如，无处不在的传输协议 TCP 和因特网上的网络协议事实标准 IP 都在它们的规范中包含了数据包的错误检查。咋一看这似乎是多余的。但是，虽然由于因特网的普及性，我们在现实中经常会一口气说“TCP/IP”，但是 TCP 不一定要运行在 IP 之上，它也可以使用 ATM 等其他的网络协议，TCP 也不是唯一能运行在 IP 之上的传输协议。而且，中间路由器（只负责网络层处理）同样也不需要检查数据包的完整性。因此，IP 规范需要考虑数据包的完整性，不能假设传输层一定会做检查。

682 ~ 684

网络层不把数据完整性检查交给链路层也是同样的道理。不同的链路层提供不同级别的数据完整性。因为网络层可能会运行在不同的链路层协议之上，所以网络层需要自己进行端到端的数据完整性保障。让我们回到 13.2 节的例子（见图 13-3）。Charlie 的哥亲使用尤巴市家里的计算机回复了 Charlie 的电子邮件，她的机器使用 PPP 协议向服务提供商传输 IP 数据包，服务提供商之间使用一个称作为中继（frame relay）的协议进行通信，在校园骨干网中使用 FDDI 协议，最后使用以太网协议到达 Charlie 的计算机。

13.11 用于数据包传输的数据结构

我们在前面的章节中逐渐经过了协议栈的各层，让我们先花一分钟喘口气，再慢慢回到协议栈的顶端。让我们来考察实现数据包传输所需要的最小数据结构。

传输层将应用层的一条消息或消息流划分成数据包后再传递给网络层，网络层再将每个数据包分别路由到目的地。因此，每个数据包都需要包含目标地址。此外，每个数据包还要包含一个序号以实现传输层的分散 / 收集功能。我们将这些数据包中不同于实际数据的元数据（metadata），称作包头（packet header）。除了目的地址和序号外，包头还可能包含源地址和校验和（用于让目的主机验证数据包的完整性）等信息。

传输层从应用层接收消息，将其划分为与网络特性相符的数据包，并在每个数据包前面加上包头。图 13-39 和图 13-40 使用与 C 语音类似的语法分别展示了一个简单的包头和一个

数据包的数据结构。字段 num_packets 使目的地的传输层知道它是否接收到了所有的数据包，以便组成一条完整的消息，并传递给应用层。目的地为了组成完整的消息，必须要发送端传输 num_packets 信息；但是为什么每个数据包的包头都含有这个字段？我们需要提醒自己注意网络是变化莫测的，每个数据包都重复携带这个信息的原因是数据包可能会乱序到达。当传输层接收到第一个新消息的数据包时需要知道要为这条消息分配多少缓冲区空间才能够完整地组装该消息。

```
struct header_t {
  int destination_address;    /* destination address */
  int source_address;         /* source address */
  int num_packets;            /* total number of packets in
                                 the message */
  int sequence_number;        /* sequence number of this
                                 packet */
  int packet_size;            /* size of data contained in
                                 the packet */
  int checksum;               /* for integrity check of this
                                 packet */
};
```

图 13-39 一个传输层数据包的包头示例

```
struct packet_t {
  struct header_t header;     /* packet header */
  char *data;                 /* pointer to the memory buffer
                                 containing the data of size
                                 packet_size */
};
```

图 13-40 传输层数据包的数据结构示例

例 13-5 一个数据包的包头由以下字段组成：

destination_address（目的地址）

source_address（源地址）

num_packets（数据包数）

sequence_number（序号）

packet_size（数据包大小）

checksum（校验和）

假设这些字段的每个占 4 字节，数据包最大为 1500 字节，计算数据包的有效载荷。

答：

数据包的包头大小 = 目的地址大小 + 源地址大小 + 数据包个数大小 + 序号大小 + 数据包大小 + 校验和

$= 6 \times 4B = 24B$

数据包的总大小 = 数据包的包头大小 + 数据包的有效载荷

数据包的有效载荷 = 数据包的总大小 − 数据包的包头大小 = 1500 − 24 = 1476B

685 ~ 686

13.11.1 TCP/IP 包头

需要强调的是，每层（传输层、网络层和链路层）包头的实际结构取决于该层的协议细节。例如，我们之前提到过，TCP 是面向字节流的协议，它将字节流划分成段（segment）作为传输单元（我们在之前的传输层讨论中一直把它称作数据包）。序号占据段中第一个字节的

位置。它表示这个段在字节流中的位置。由于 TCP 是面向连接的，所以包头中也含有端口号字段。连接的两端分别称为源端口（source port）和目的端口（destination port）。由于 TCP 连接中的数据流是双向的，所以在发送新数据时包头还可以捎带已接收数据的确认信息。包头中的确认序号（acknowledgement number）表示预期会从此连接中接收到的下一个数据包的序号。由于 TCP 协议有内置的拥塞控制，所以包头还包含了窗口大小（window size）字段，这个字段的作用即有深义且重要。两端都可以依据延迟时间和丢包后的重传次数来监视网络拥塞状况。发送端根据这些指标，利用在包头中的窗口大小字段来宣布它愿意从另一端接收的数据量。同样，发送端也是根据这些指标来选择段的长度。除此以外，段中还有一些其他的特殊字段：

- SYN：它标志新字节流的开始，用于在两端之间同步传输的起始序号。
- FIN：它标志字节流传输的结束。
- ACK：它标志包头中捎带有 ACK，因此确认序号字段是有意义的。
- URG：它标志这个数据段中有“紧急”数据。例如，如果你按下 Ctrl-C 终止了一个网络程序，那么应用层协议会将其转换为一个紧急消息交给 TCP。

源和目的 IP 地址都是它们自己从网络层那里隐式获取的。包头中包含这些 IP 地址的部分称为伪包头（pseudo header），在段传输及接收过程中，它会在因特网协议栈中的 TCP 和 IP 之间来回传输。

IP 数据包的格式非常简单。IP 将传输层的消息（或 TCP 的段）划分成多个 IP 数据包，并在目的地将这些数据包重组成原始的消息。因此，除了源和目的 IP 地址外，IP 包头还包含该 687 IP 包的长度和段偏移量（即这个数据包在传输层消息中的位置）。

13.12 消息传输时间

现在我们已经知道了消息如何在网络中传输，接下来让我们更详细地了解消息传输所需的时间。在 13.6 节中，为了使传输协议的讨论保持简单，我们忽略了除介质的传播时间外的其他所有时间开销。现在让我们来了解网络的连通性会如何影响端到端的传输时间。为此，首先考虑在两端之间传输消息可能会涉及的时间元素。图 13-41 展示了这些要素。

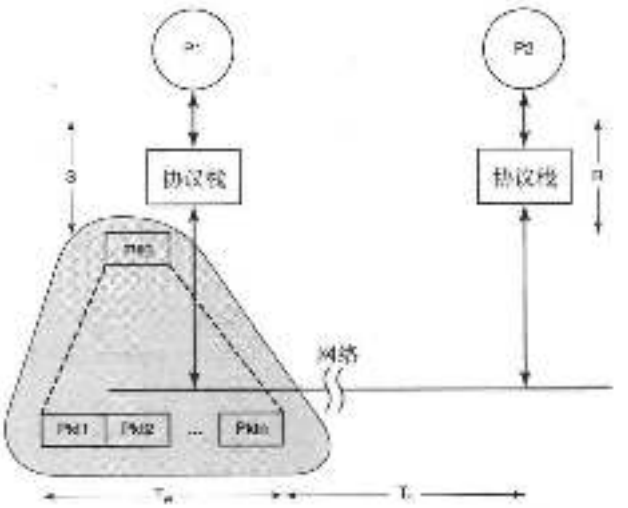


图 13-41 消息传输时间可以分为 4 个部分：发送端延迟（S）、传输延迟（T_x）、传播时间（T_p）、以及接收端延迟（R）

我们将简单地了解消息传输时间，消息传输的总时间由以下几部分组成：

1）**发送端的处理延迟（S）**：这是在发送端协议栈各层中累积消耗的时间，其中包括：

- 传输层功能消耗的时间，包括将消息划分成数据包，向数据包添加包头、计算数据包的校验和。
- 网络层功能消耗的时间，例如给数据包查找路由。
- 链路层功能消耗的时间，如介质访问控制和数据成帧。
- 物理层中与特定介质相关的功能所消耗的时间。 688

这部分时间取决于协议栈的软件架构细节和实际实现的效率。

2）**传输延迟（T_w）**：这是发送端将位传入线路所需要的时间，也就是，从你的计算机传入物理介质的时间。例如，对于一条给定的消息，古比特链路的传输延迟 T_w 会远小于拨号连接。

例 13-9 对于一条 21MB 大小的消息，(a) 计算使用 56Kbits/s 拨号连接时的传输延迟；(b) 计算使用古比特网络连接时的传输延迟。

答：

a. 传输延迟 = 消息大小 / 网络带宽 = $(21\times2^{20}\times8\ \text{b})/(56\times2^{10}\ \text{b/秒}) = 3\times2^{10}$ 秒 = 3072 秒

b. 传输延迟 = 消息大小 / 网络带宽 = $(21\times2^{20}\times8\ \text{b})/(10^{8}\ \text{b/秒}) = 0.176$ 秒

3）**传播时间（T_l）**：这是从发送端将消息传入线路到消息到达接收端的网络接口所需要的时间。也就是说，我们把消息在传输途中所经历的延迟都混合到了一起。消息所经历的延迟有以下两种：

- **传播延迟**：这是位信息从点 A 沿线路传播到点 B 所需要的时间，这个时间取决于许多因素。第一个因素涉及两点之间的距离与光速。例如，两点离得越远，位信息在线路中传输的距离就越长，因此也会消耗更多的时间。这就是我们从亚特兰大市访问 CNN 的网站会比从印度班加罗尔快的原因。除了距离外，还有其他因素会影响 T_l。T_w 已经计算了从计算机连接到物理网络的时间。但在实际中，除了计算机与网络的连接外，传输的消息在到达目的地之前可能还需要穿过多条（带宽不同的）物理链路。因此我们还需要对途中的每条物理链路分别计算传播延迟，并求和以得到总的传播延迟。为了简单起见，我们把各段延迟的和称作端到端的传播延迟。 689
- **排队延迟**：我们之前已经提到过，广域网其实是一个网络的网络（见图 13-3）。消息在经过沿途的交换机时会遇到排队延迟。此外，途中还有一些中间网络协议负责把消息传递给目的地，这些协议也会增加延迟时间。最终，接收端从线路中接收数据包并存入网络接口中的一个缓冲区。

为了简单起见，我们把上面的这些延迟全部混合在一起，称作传播时间（time of flight）。

4）**接收端的处理延迟（R）**：这部分时间对应于发送端的处理延迟，同样具有物理层、链路层、网络层以及传输层的延迟。

$$\text{消息传输的总时间} = S+T_w+T_l+R \tag{13-1}$$

式（13-1）也表示了消息的端到端延迟。我们很容易从这个式（13-1）中看到，计算机与网络之间的接口带宽不足以说明网络通信的延迟时间。我们可以依据消息传输时间来计算吞吐量（throughput），它定义为网络的实际传输速率。

$$\text{吞吐量} = \text{消息大小} / \text{端到端延迟} \tag{13-2}$$

我们已经知道，一条消息需要经过多跳才能从源到达目的地。因此，每条消息在任意两个网络跳之间都需要经历以下延迟：发送端的处理延迟、传输延迟、介质上的传播延迟、排队延迟，以及接收端的处理延迟。我们为了简化本节中消息传输时间的讨论，只让读者对端到端的延迟有所感受，因此将端到端的所有延迟（除了发送端的处理延迟、传输延迟和接收端的处理延迟外）都算入了传播时间。

例 13-10 考虑下列条件：

发送端的处理延迟 =1 ms

消息大小 =1000 b

690 线路带宽 =1 000 000 b/s

传播时间 =7 ms

接收端的处理延迟 =1 ms

请计算吞吐量。

答：

消息传输的总时间 $=S+T_x+T_t+R$，其中

S（发送端的处理延迟）=1 ms

T_x（传输延迟）= 消息大小 / 线路带宽 = 1000/1 000 000 s = 1 ms

T_t（传播时间）=7 ms

R（接收端的处理延迟）=1 ms

因此，传输一条 1000 位消息的时间 = 1+1+7+1 ms = 10 ms.

吞吐量 = 消息大小 / 传输时间 =（1000 b）/（10 毫秒）= 100 000 b/ 秒

下面的例子说明了，当消息传输过程中存在数据包丢失时所需要传输的数据包数量。

例 13-11 考虑下列条件：

消息大小 =1900 Kb

包头大小 =1000 b

数据包大小 =20 Kb

假设传输中有 10% 的数据包会中途（ACK 包不会出错，也没有数据包丢失），且每个数据包得以被正确接收的，则发送端要完成消息传输总共需要发送多少个数据包？

答：

数据包大小 = 包头大小 + 有效载荷

20 000=1000+ 有效载荷

数据包中的有效载荷 =19 000 b

发送消息所需要的数据包数量 =1 900 000/19 000=100

691 由于 10% 的数据包可能失败，所以数据包的总数 =100+10+1=111

发送的数据包	丢失	成功
100	10	90
10	1	9
1	0	1
111	11	

当我们使用滑动窗口协议时，计算消息的传输开销会变得麻烦。

下面的例子说明了当存在滑窗协议时的消息传输开销。

例 13-12 考虑下列条件：

消息大小 =1900 Kb

包头大小 =1000 b

数据包大小 =20 Kb

线路带宽 =400 000 b/s

传播时间 =2 s

窗口大小 =10

发送端的处理延迟 =0

接收端的处理延迟 =0

ACK 大小 = 忽略不计（视为 0）

假设网络中没有错误并且数据包按序传输，那么要完成消息传输总共需要多少时间？

答：

数据包大小 = 包头大小 + 有效载荷

因此，数据包中的有效载荷 = 数据包大小 − 包头大小 =20000−1000 =19000 b

发送消息所需要的数据包数量 =1 900 000/19 000=100

数据包的传输延迟 = 数据包大小 / 线路带宽 =20000/400000 s =0.05 s

数据包在发送端的时延 = $S+T_p$ = 发送端的处理延迟 + 传输延迟 = 0+0.05 s =0.05 s

692

由于窗口大小为 10，所以发送端将 10 个数据包传入线路，并开始等待 ACK。发送端的时序图如下所示。



接收端在第一个数据包发出 2 秒后收到这个数据包。

数据包从端到端延迟 = $S+T_p+T_t+R$ =0+0.05+2+0 s =2.05 s

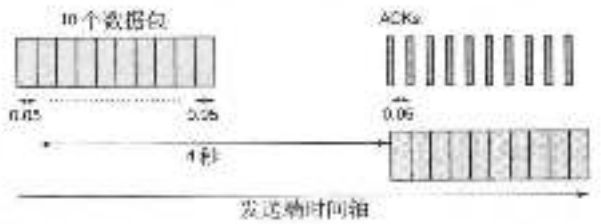
当接收端收到数据包时，无时开始准备 ACK 包：

接收端生成 ACK 包的开销 = $(S+T_p)$（在接收端）=0+0（因为 ACK 包的大小可以忽略不计）=0

ACK 包的端到端延迟 = $S+T_p+T_t+R$ =0+0+2+0 s =2 s

因此，如上图所示，在第一个数据包传入线路的 4 秒后，发送端收到第一个 ACK。更普遍的是，每个 ACK 都是在对应数据包传入线路的 4 秒后被接收（假设没有数据包丢失）。

在这样的场景中，10 个数据包所对应的 10 个 ACK 会以 0.05 秒的间隔时延到达，如下图所示。



693

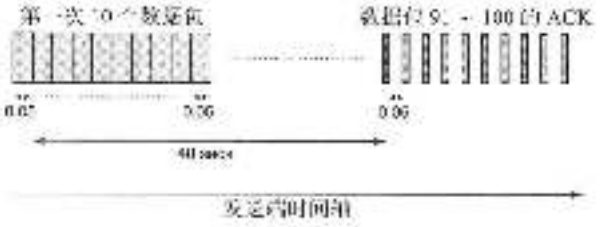
当发送端收到第一个 ACK 后，就可以继续发送下一个数据包。（滑动窗口的大小是 10）接下来的 10 个数据包会一个接一个地发送，如上图所示。

因此，在 4.05 秒的周期内，发送端发出了 10 个数据包并收到一个 ACK。之后每 4.05 秒就会重复一次这个周期，并发送 10 个新的数据包。

为了发送 100 个数据包，我们需要经历 10 个选择的周期。

因此，10 个周期所需的时间 =4.05 × 10=40.5 s。

经过这些时间之后，100 个数据包都已经发出，并已经收到了 91 个 ACK，如下图所示。从图中可以看出，剩余为 9 个 ACK（图中没有阴影的数据包）会以 0.05 秒的间隔相继到达。



接收剩余的 9 个 ACK 所需的时间 = 9 × 0.05 s = 0.45 s

完成消息传输的总时间 =10 个周期所需的时间 + 接收剩余的 ACK 所需的时间 = 40.5+0.45 s = 40.95 s

在前面的例子中，发送端和接收端的处理延迟都是零。如果它们中有非零值，则会增加端到端的总延迟（见式（13-1））。数据包的流水线也与前面的例子类似。（更多的例子请参见本章末尾的习题。）

13.13 协议层功能总结

五层网络协议栈的功能总结如下：

- 应用层包括了 HTTP、SMTP 以及 FTP 等协议，以便支持包括 Web 浏览器、电子邮件、文件上传下载、即时通信以及多媒体会议与协作等特定类型的应用程序。操作系统中特定的网络通信库，如套接字和远程过程调用（Remote Procedure Call，RPC；参见 13.16 节），也属于这一层。
- 传输层为应用程序在通信两端之间提供消息传输。我们已经知道这一层的功能取决于应用程序对于服务质量的要求。TCP 在两端之间提供可靠的面向字节流的按序传输，并包含拥塞控制，而 UDP 提供不可靠且没有保障的乱序到达数据报服务。
- 网络层将传输层传来的数据传递到目的地。这层的功能包括依据链路层协议进行分组 / 重组、路由、转发，以及为传输层提供服务模式。正如我们之前所看到的，作为守护进程运行的路由算法程序负责确定到达目的地的路由并维护路由表和转发表。
- 数据链路层向网络层提供访问物理介质的接口。这层的功能包括 MAC 协议、依据物理层的线路信息将数据包划分成帧、错误检测（例如，在以太网中检测数据包冲突，或者在令牌环网络中检测令牌丢失）、错误恢复（例如，在以太网中发生冲突之后的退避算法以及数据包重传，或者在令牌环网络中重新生成令牌）。
- 物理层涉及用于传输的物理介质的介质类型（铜导线、光纤、无线电等）中的机械电气细节、介质中信号的性质，以及用于数据链路层中 MAC 协议实现的信号交换的具体方式。

13.14 网络软件与操作系统

正如我们之前在 13.1 节中所提到的，操作系统和网络软件之间有 3 个逻辑的接触点。

13.14.1 套接字库

这是操作系统为网络协议栈所提供的接口。TCP/IP 是我们日常生活中所依赖的所有的与因特网相关服务的根基，包括浏览网页、写博客、即时通信等。因此，多了解一些 TCP/IP 为分布式应用程序编程所提供的接口是有意义的。

当你编写分布式应用程序时，你并不需要直接处理 TCP/IP 中的复杂情况。我们会在接下来的段落中解释这是为什么。

如果你观察因特网协议栈（见图 13-5），你就会意识到应用程序位于图中协议栈的上方。操作系统为分布式应用程序编程提供了定义良好的接口。例如，UNIX 操作系统提供套接字（socket）作为进程之间通信的抽象（见图 13-42）。

695

图 13-42 表明套接字抽象与两个进程所在的位置无关。例如，这两个进程可以都在同一个处理器上执行，或者在共享内存的多处理器的不同处理器上（如在第 12 章中讨论的 SMP）执行，或者是在通过网络连接的两台不同的计算机上执行。也就是说，套接字抽象与实际如何将位从进程 P1 移动到进程 P2 无关。

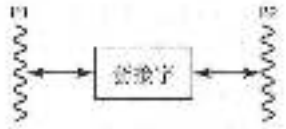


图 13-42 使用套接字进行进程间通信

TCP/IP 等协议是实现套接字抽象的工具。先让我们了解为什么套接字抽象要支持多种下层协议。从进程 P1 和进程 P2 的观点来看，这是无关紧要的。但是从效率的角度来看，根据通信两端的位置使用不同的协议是更好的。例如，考虑两个进程位于同一个处理器上或者位于一个 SMP 中的不同处理器上。在这种情况下，通信永远不会经过计算机外的外部线路，许多低层问题（如数据包丢失、数据包乱序到达以及传输错误）都不会存在。即使 P1 和 P2 在同一局域网（例如，家庭网络）中的不同机器上，数据包丢失的概率也可以忽略不计，因此基本不需要担心这些低层问题。另一方面，也无需要使用复杂的协议来解决这些低层问题的情况。例如，进程 P1 在你宿舍的计算机上执行，而进程 P2 在校园网中的工作站上执行。

如图 13-42 所示，通信所使用的通信类型也是在建立通信信道时要考虑的问题。邮政服务做比喻。你寄出一张贴了邮票的明信片，邮局并不能保证它能够到达目的地。如果你需要知道收件人是否实际收到了你的来信，你得付更多的钱以获取信件送到的回执。类似于这个明信片的例子，P1 和 P2 可能只需要偶尔交换简单且固定大小的（通常很小）数据报（datagram），并且不需要确保可靠。另一方面，图 13-42 也可以表示从朋友那里下载电影。在这种情况下，两端之间的通信需要传输连续的流（stream），并且需要可靠性保障。通信信道所需要的通信类型与协议的选取无关。应用程序的属性决定了通信类型，而通信两端的物理位置以及物理连接情况决定了使用的协议。

在 UNIX 中创建套接字对，可以指定所需的套接字属性，包括要使用的特定通信类型（如数据报、面向流等）与协议族（如 UNIX 内部协议族、网络协议族等）。

696

操作系统级的套接字 API 实现细节有点类似于我们在第 12 章中所讨论的多线程库的实现细节。这些细节已经超出了本书的范围。有兴趣的读者可以参考讨论这些问题的其他书。[⊖]

其他被广泛使用的操作系统也提供了套接字 API，例如 Microsoft Windows（XP，Vista 和 Win7 等多个版本）和 Mac OS X。

⊖ 《*The Design and Implementation of the FreeBSD Operating System*》，Marshall Kirk McKusick, George V. Neville-Neil 著 [McKusick, 2004]。

13.14.2 在操作系统中实现协议栈

在当今时代，无论操作系统为网络编程提供了什么样的API，它还需要为网络连接提供协议栈中不同层的具体实现。协议栈中的传输层和网络层通常在操作系统中以软件形式实现，而协议栈的链路层（即第2层）通常在硬件中实现。例如，你的笔记本电脑多半会有一块以太网卡，它提供协议栈第2层的功能。而在移动设备中，无线网卡已经变得非常常见，这些无线网卡实现了它们自己的链路层协议，这些协议是以太网等随机访问协议的变种。事实上，IEEE的无线局域网协议族标准都由自包含了以太网的IEEE 802族。当我们在13.8.4节中讨论CSMA/CA时，已经简要了解了无线局域网协议。我们把无线局域网协议的细节讨论留给更高级的课程。[⊖]

我们已经看到，TCP/IP是在当前因特网中占据主导地位的传输层/网络层协议组合。相应地，标准的操作系统（如UNIX、Mac OS以及Microsoft Windows）也都包含了TCP/IP协议栈的高效实现。而且，大多数关于操作系统规范与调优的新的研究工作也会使用协议栈作为有代表性的例子，以验证关于操作系统机制的新的研究结果。

协议栈是非常复杂的软件，通常是需要专业程序员开发若干人年，具有成千上万行代码的软件。有兴趣的读者可以参考完全致力于这个主题的教科书。[⊖]

13.14.3 网络设备驱动程序

如果不简单了解一下网络设备驱动程序，那么关于操作系统对于网络支持的讨论将是不
697 完整的。网卡（NIC）使主机能够连接到网络，一台主机也能有多个网络接口，这取决于它所连接到的网络的数量。主机中每块可用的网卡也都在操作系统中有相对应的设备驱动程序。正如我们之前提到过的（参见13.9节），网卡通常包含协议栈中链路层的一些硬件功能。而网卡的设备驱动程序对于这些硬件功能进行了补充，是网卡能够完成链路层的各种项事。我们在第10章中讨论了用于高速I/O的DMA控制器，网卡采用了DMA引擎，直接在主机内存与网络之间输入/输出数据。但是网络I/O与磁盘I/O之间有一点根本的不同，磁盘I/O在两个方向上的数据移动（即到/从磁盘）都是由操作系统发起的，用于响应一些用户级或系统级的需求（例如，打开文件或者处理页错误）。再考虑网络，向网络发送数据包肯定是由用户级或系统级的需求发起的，但是操作系统不能够控制网络数据包的到达。因此，操作系统必须要在任何时候都准备好处理这种可能事件，这就是网卡的设备驱动程序的主要任务，它实现了一组衔接操作系统与网卡的功能。

例如，对于一块连接主机与以太网网卡的网卡，相应的设备驱动程序包含以下功能：

- 在主机内存中分配/释放用于发送和接收数据包的网络缓冲区（network buffer）。
- 当有数据包需要发送到线路中时，建立网络传输缓冲区并通过网卡发起DMA操作。
- 在操作系统中注册网络中断处理程序。
- 为网卡建立网络接收缓冲区，并通过DMA将收到的网络数据包存入主机内存中。
- 获取网卡的硬件中断，在必要时上行调用（参见第11章中关于上行调用的讨论）至协议栈的上层（例如，将数据包到达事件告知上层）。

⊖ 参见Kurose和Ross的教科书，*Computer Networking: A Top Down Approach Featuring the Internet*，Addison-Wesley [Kurose, 2005]。这本书很好地覆盖了无线局域网的基本技术。

⊖ 《*TCP/IP Illustrated, Volume 2: The Implementation*（*Addison Wesley Professional Computing Series*）》，Gary R. Wright, W. Richard Stevens著[Wright, 1995]。

图 13-43 展示了处理网络数据包到达时硬件与软件的行为。因为网卡知道由设备驱动程序为接收数据包申请的网络缓冲区，所以它可以立即使用 DMA 将输入的数据包存入主机内存中这些预先分配的缓冲区。之后网卡使用处理器的硬件中断机制（第 4 章和第 10 章）将数据包到达事件告诉设备驱动程序。设备驱动程序使用我们在第 11 章中所讨论的上行调用机制将输入数据包告知协议栈上层。对于更多的详细内容，有兴趣的读者可以参考与网络设备驱动程序开发相关的文献。⊖

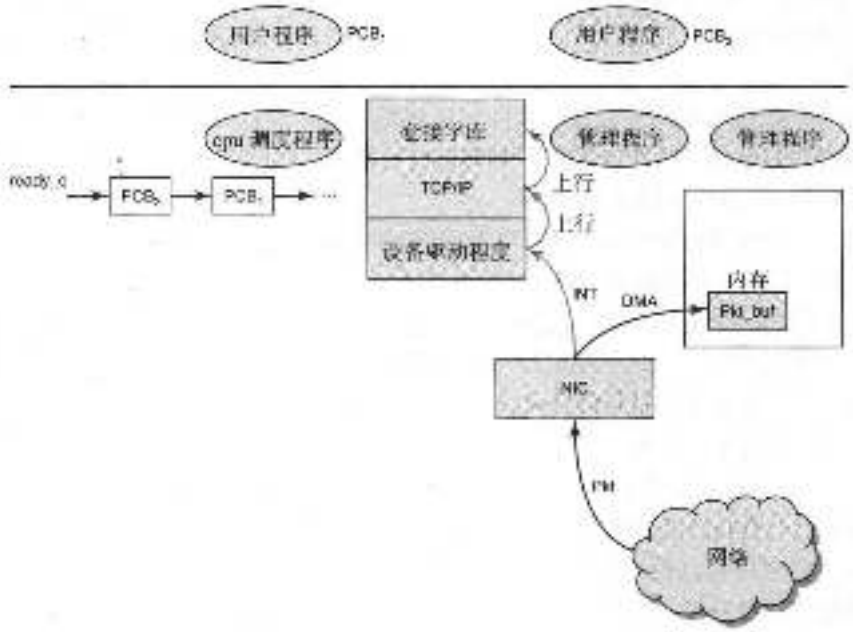


图 13-43 网络数据包到达。我们已经从前面的章节中熟悉了操作系统的其他功能，例如 CPU 调度、虚拟内存管理以及文件系统管理。图 13-43 中展示了用于支持网络编程的操作系统中协议栈各层以及当有网络数据包到达时硬件（NIC）与软件所执行的操作

13.15 使用 UNIX 套接字进行网络编程

为了使关于网络编程的讨论更加具体，让我们从应用程序的角度来仔细看看 UNIX 套接字如何工作。

在 UNIX 上套接字创建函数需要 3 个参数：域（domain）、类型（type）和协议（protocol）。

- 域（domain）：这个参数选取通信所使用的协议族。例如，如果通信进程位于因特网中，则会选择 IP 协议族；如果进程都在同一台机器或同一个局域网中，则会选择 UNIX 内部协议族。
- 类型（type）：这个参数指定应用程序所需的属性，如数据报或流。
- 协议（protocol）：这个参数在协议族（由域参数决定）满足所需属性（由类型参数决定）的协议中指定所使用的协议。

⊖ Network Device Driver Programming Guide, http://developer.apple.com/documentation/DeviceDrivers/Conceptual/NetworkDriver/NetDoc.pdf。

套接字创建函数提供这些参数选择是为了通用性。例如，在特定的协议族（如 UNIX 内部协议族）中可能会有多个协议满足通信类型的需求，实际上也可能有且仅有一个协议满足需求。例如，如果是 IP 协议族和流类型，那么 TCP 可能是唯一可选的传输协议。尽管如此，显著分离的选项体现了抽象的力量，这是我们在这本书中一直强调的。

在图 13-42 中，两个端点 P1 和 P2 是对称的。但这并不是所有的情况——至少通信信道不是使用这种方式建立的。让我们在本节中更深入地了解一些这方面的问题。

使用套接字的进程间通信遵循客户端/服务器（client/server）模式。使用套接字的通信可以设置为面向连接的或者无连接的。正如之前所提到的，类型（type）参数会在套接字创建时明确这些属性。

让我们来考虑使用 UNIX 套接字的面向连接的通信。可以把客户端与服务器之间建立连接的过程比作拨打电话。呼叫方需要知道要拨打的号码，接听方可以接受来自任何人的呼叫。客户端就是呼叫方，服务器就是接听方。

让我们进一步考虑这个比喻。服务器对于进程间通信会做以下准备：

1）创建一个用于通信的套接字，如图 13-44 所示。注意这使服务器创建一个通信终端，类似于安装了一台还没有与外界建立起连接的电话。为了使电话可用，我们需要先获得一个电话号码（这也涉及从服务提供商获取线路连接）并将号码与电话关联起来，这是下一个步骤。



图 13-44 服务器端的套接字调用

2）与比喻中的电话号码相对应的是一个名称（也称为地址）。一个与套接字相关联的唯一标识符。将名称与套接字关联起来的系统调用称作绑定（bind）。如果要使用 IP 协议通过因特网进行通信，则名称包括两个部分<主机地址，端口号>。主机地址是运行服务器端主机的 IP 地址，端口号是一个 16 位的无符号数。在执行绑定操作时，操作系统会检查服务器指定的端口号是否已经被占用（见图 13-45）。

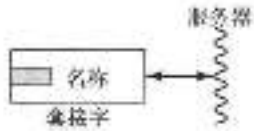


图 13-45 绑定后的服务器套接字

当完成绑定之后，可以把名称传给想与此服务器通信的其他客户端。如果是电话号码，你可能会将它发表在一本电话目录中，以使其他人查看。在客户端/服务器通信的情况下，类似的事情是创建一个名称服务器（name server）。名称服务器是一台众所周知的机器，潜在的客户端通过访问名称服务器来获取服务器的名称。

3）要使电话能够接听到来电，当然需要用电话线将电话连接到墙上的电话插座（来自服务提供商的线路连接）。首先，你需要告诉电话公司你想要接听来电以及是否想要“呼叫等待”功能——也就是，你希望通过电话线路同时接受多少个呼叫。这相当于服务器端执行监听（listen）系统调用。它告诉操作系统在这个套接字上可以排多少个呼叫。

4）服务器端为了完成输入连接请求的接收，需要执行一次接受（accept）系统调用。这相当于在电话公司开通电话服务之后，通过电话话筒收听来电，之后再将话筒放回原位，以等待新的来电。见图 13-46。



图 13-46 监听并接受呼叫的服务器套接字

现在，让我们看看客户端要与服务器建立连接需要做些什么。

1）创建一个客户端套接字，如图 13-47 所示。这相当于为呼叫方获取一个电话。

2）为了接通电话，呼叫方需要拨打电话号码。客户端使用连接（connect）系统调用来完成相同的事情。它连接客户端套接字与名称服务器发布的名称，如图 13-48 所示。



图 13-47 客户端的套接字调用

客户端和服务器都还没有完全准备好通信。连接（connect）系统调用类似于拨打电话号码，但直到接听方拿起话筒，呼叫过程才算完成。服务器端的接受（accept）系统调用在这种客户端 - 服务器情况中完成类似的事情。在电话呼叫中，需要拿起话筒才能建立起连接。在套接字中，服务器已经表示愿意接受从连接中传来的呼叫。因此，如图 13-49 所示，操作系统负责在客户端和服务器之间完成连接建立操作。注意如果你的电话使用了呼叫等待功能，你能够在与别人说话时知道有新的来电。操作系统也为套接字提供了相同的功能，它会为新建立的连接创建一个新数据套接字（new data socket）。当客户端呼叫已经处于接受状态的套接字时，操作系统会隐式地将连接请求排队。

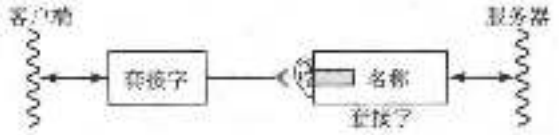


图 13-48 客户端执行连接（connect）系统调用之后的客户端 - 服务器关系

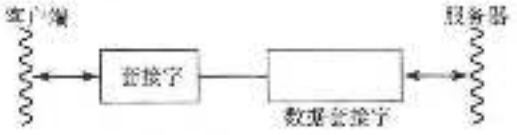


图 13-49 连接建立之后的客户端 - 服务器关系

此时，客户端和服务器都已经准备好以对称的方式交换消息。因此，虽然建立连接时两端并不对称（就像是电话呼叫的情况），但实际的通信确实是对称的。

你可能想知道为什么客户端和服务器都需要执行套接字（socket）系统调用。我们已经从前面的章节中知道，每个进程都在它自己的地址空间中执行，每个进程的数据结构也是通过地址空间中的系统调用创建的。因此，在每个通信进程的地址空间中都需要有套接字抽象的表示（见图 13-50）。所有其他的系统调用（绑定、监听、接受、连接）通过操作系统和网络协议栈使两个套接字连接在一起，以实现图 13-42 中的情形。

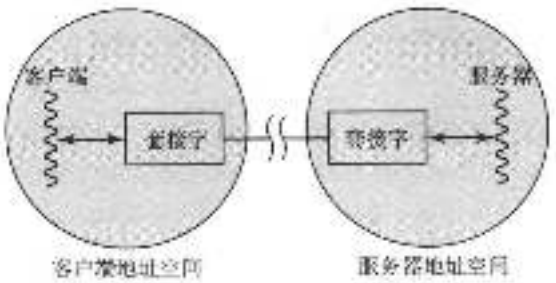


图 13-50 客户端 - 服务器地址空间中的套接字

之前的讨论主要集中在需要建立连接的流类型的套接字，如果套接字的类型是数据报，

则通信会变得非常简单。在这种情况下，服务器既不需要执行监听，也不需要执行接受。从客户端发来的数据报被简单地存入了由服务器创建的数据报套接字中。同样，如果客户端创建了数据报套接字，则它可以在此套接字上简单地使用服务器端的套接字地址（即主机地址和端口号）进行数据的发送与接收。

总之，无论进程在何处执行，UNIX 为进程间通信提供了以下几种基本的系统调用：

- socket（套接字）：创建一个通信终端。
- bind（绑定）：将一个名称或者地址绑定到套接字上。
- listen（监听）：在套接字上监听输入的连接请求。
- accept（接受）：在套接字上接受输入的连接请求。
- connect（连接）：使用名称（或地址）向一个远程套接字发送连接请求；如果服务器已经执行了接受（accept）系统调用，则通过为连接创建一个数据套接字来建立连接。
- recv（接收）：通过套接字从远程的对等节点接收输入数据。
- send（发送）：通过套接字向远程的对等节点发送数据。

图 13-51 展示了客户端和服务器之间用于建立面向流的套接字通信的协议。监听系统调用让操作系统预先知道流类型的套接字需要同时支持多少个连接。每次建立新的连接时（客户端的连接调用与服务器的接受调用相匹配），操作系统都会创建一个新的数据套接字。这个新的数据套接字的生命周期取决于连接的生命周期。一旦客户端关闭连接，这个新创建的数据套接字也会被关闭。

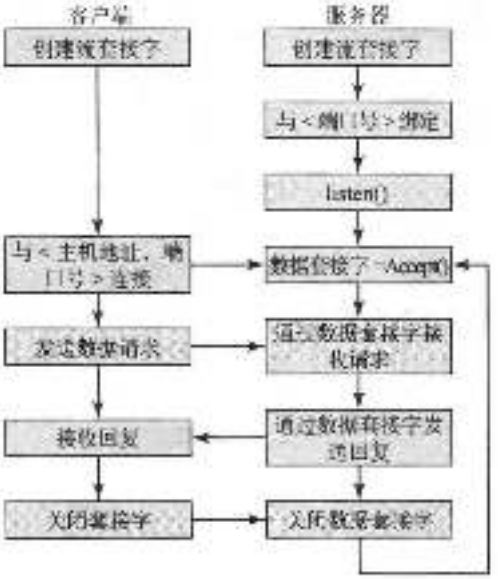


图 13-51 流套接字上的数据通信。服务器创建一个套接字，并绑定一个端口号，执行接受（accept）系统调用进行阻塞，以表明它愿意接受连接请求。客户端也创建一个套接字，并使用<主机地址，端口号>连接到服务器。操作系统为新建立的连接创建一个新的数据套接字，客户端与服务器在此之上交换数据流。每个连接都有其专用的数据套接字。当客户端关闭连接时，相应的数据套接字也会关闭。此时，服务器可以返回到原始的流套接字并等待新的连接请求

监听（listen）调用是实现多线程服务器的通有方机制，当创建并绑定套接字之后，多个线程可以在同一个套接字上执行接受（accept）调用。这使得在同一个套接字上能够同时容纳多个客户端—服务器连接（每个连接都有自己独立的数据套接字），其数量只受限于监听（listen）调用所指定的界限。

图 13-52 展示了客户端和服务器之间用于建立面向数据报的套接字通信的协议。注意当服务器绑定了端口号之后，就已经准备好在数据报套接字上进行发送/接收。同样，客户端不需要显式地执行连接调用，就可以在数据报套接字上开始进行发送和接收。

701
?
704

使用 UNIX 套接字的客户端—服务器程序的示例请参见附录。

还有一些问题需要引起使用 UNIX 套接字来实现分布式程序的程序员的关注，我们列举了其中的一些问题：

- 如果两个进程的套接字属性（域和类型）不匹配，那么是否还能进行通信？
- 服务器如何选取端口号？
- 在 UNIX 域套接字中，名称会转变成什么？
- 如果客户端想要连接的名称不存在，会发生什么？
- 如果由于某种原因客户端与服务器之间的物理链路中断了，会发生什么？
- 多个客户端可以向同一个名称成功地执行连接（connect）调用吗？

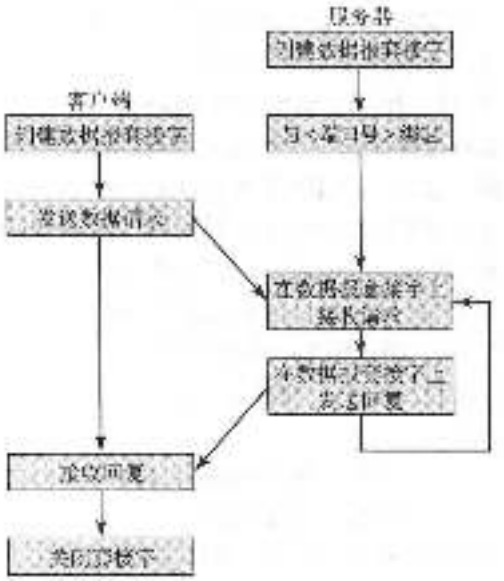


图 13-52 数据报套接字上的通信。当服务器创建了一个数据报套接字并绑定一个端口号之后，就可以开始在套接字上进行数据传输。客户端不需要执行连接调用，就可以与服务器进行数据交换

705

这些问题的答案取决于调用套接字库时的确切语义，以及实现套接字库的开发者所做的选择。提出这些问题的目的仅仅是为了激发读者的兴趣。通过仔细阅读 UNIX 系统上的套接字库的相关文档[⊖]，就能够很容易地找到这些问题的答案。

操作系统使用合适的传输协议（适合所需的套接字语义）来实现套接字（socket）系统调用。套接字库的实现细节超出了本书的讨论范围，对这些细节有兴趣的读者可以参考更高级的网络与操作系统书籍。

当然，我们略去了许多细节，为了让讨论保持简单。UNIX 操作系统提供了许多实用程序来协助使用套接字进行网络编程。

随着网络服务器（例如，文件服务器和 Web 服务器）中 UNIX 操作系统的普及与万维网（WWW）的出现，使用套接字进行网络编程已经变得越来越重要。我们鼓励有兴趣的读者去学习更高级的操作系统课程，以获取使用套接字进行分布式编程的实践经验并学习如何在操作系统中构建套接字抽象。

⊖ 参见在线请访问文档 http://www.freebsd.org/cgi/man.cgi。

13.16 网络服务与高层协议

让我们以一个具体的例子来了解网络应用程序，如文件传输协议（File Transfer Protocol，FTP）。这些服务分为客户端和服务器两部分。客户端上的主机通过网络连接到远程主机，并使用传输协议（例如，TCP/IP）与远程主机上的 FTP 服务器进行通信。当文件传输完成后，客户端和服务器会关闭它们之间的网络连接。用于广域网（WAN）中的其他网络应用程序（如 Web 浏览器和电子邮件等）也执行类似的操作。

了解为何有些局域网服务与广域网服务执行的操作完全不同是非常有趣的。例如，我们经常在日常计算中使用文件服务器。当我们打开一个文件时（例如，使用 UNIX 中的 fopen），实际上我们可能是通过网络从一个远程文件服务器访问该文件。然而，这样的局域网服务不会使用传统的网络协议栈。在 UNIX 操作系统中它们使用一种称为远程过程调用（Remote Procedure Call，RPC）的简单协议。图 13-53 展示了 RPC 的基本思想。进程 P1 正常调用过程 foo，而过程 foo 通过网络在另一台远程机器上的进程 P2 中执行。事实上，RPC 的用户看不见远程过程执行。

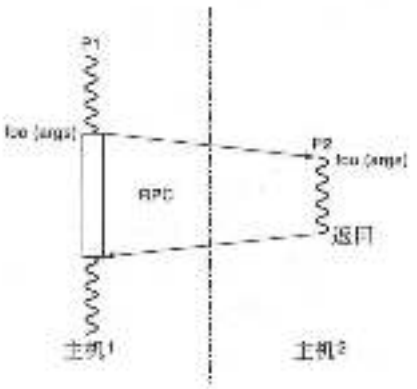


图 13-53 UNIX 中的远程过程调用（RPC）

网络文件系统（Network File System，NFS）存在于 RPC 机制之上。图 13-54 从高层展示了 NFS 等网络工具的交互过程。用户的 fopen 调用转变成对 NFS 服务器 RPC 调用，并在服务器上对相应文件执行文件打开系统命令。

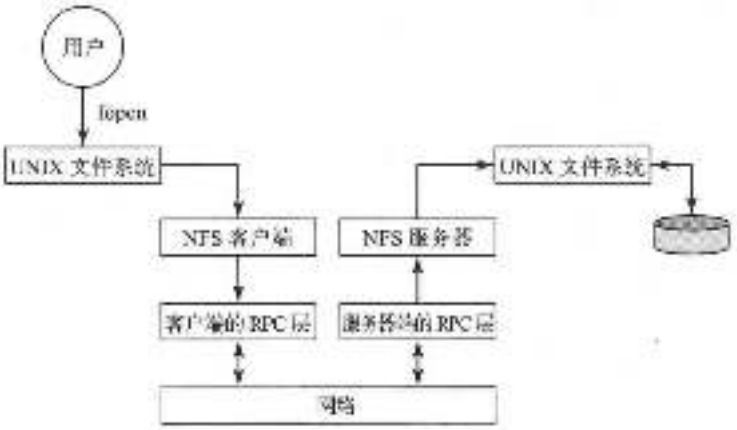


图 13-54 UNIX 网络文件系统（NFS）

本节中对于网络服务与高层协议的简要讨论引出了一些有趣的问题：

706 / 707

1）我们如何设计 RPC 系统？

2）与本地过程调用相比，远程过程调用的语义是什么？

3）RPC 如何处理网络传输中的故障？

4）我们在第 11 章中已经看到了文件系统的实现，在实现网络文件系统时有什么样的语

又差别？

本节对于分布式系统这个迷人的领域给出了简要的介绍，我们希望读者能够学习更深入的课程，以便更深入地了解分布式系统。

小结

在本章中我们涵盖了计算机网络的基础知识。我们在 13.3 节中讨论了网络通信所需要的支撑软件，并在 13.4.1 节中介绍了五层网络协议栈。网络协议栈的核心是传输层，我们在 13.6 节中对其做了详细介绍。传输层负责为应用程序提供抽象且独立于网络的细节，其主要功能是确保消息按序传输、支持任意大小的消息以及对应用程序屏蔽消息传输中的数据丢失。我们说明了许多传输层协议，它们在公平性、可靠性和能信信道利用率方面各有所长。

我们的传输协议讨论从简单的停止并等待协议开始（13.6.1 节），结束于当今因特网所使用的传输协议（13.6.5 节）。我们在 13.7 节详细讨论了网络层的功能。网络层负责为网络中的节点提供合理的寻址方案（13.7.2 节），其中包含从源到目的地为消息的数据包计算路径的路由算法（13.7.1 节），并为数据包传输提供服务模式（13.7.3 节）。在 13.7.1 节中所讨论的路由算法包括 Dijkstra 最短路径算法、距离矢量算法以及分层路由。我们在 13.7.3 节中涵盖了电路交换、分组交换以及报文交换这 3 种使用网络资源（如源与目的地之间的中间节点和路由器）的方式。分组交换已经被因特网广泛使用，我们讨论了分组交换中的两种服务模式，虚电路和数据报。我们在 13.8 节中讨论了链路层技术，并特别关注以太网（13.8.1 节），以太网已经成为了事实上的局域网标准。在这节中还涵盖了其他的链路层技术，如令牌环（13.8.5 节）、FDDI 和 ATM（13.8.6 节）。连接主机与物理层的网络硬件的讨论在 13.9 节。协议分层是一个用于构建系统软件的模块化方法，我们在 13.10 节中讨论了网络协议栈中不同层之间的关系。在之后的两节中我们讨论了用于数据包传输的数据结构（13.11 节）与网络中消息传输时间的组成成分（13.12 节）。我们在 13.13 节中总结了五层因特网协议栈的功能。

我们在 13.14 节中考虑了如何在操作系统中实现网络协议栈，包括套接字库以及设备驱动程序的讨论。在常见的操作系统中它们都与 TCP/IP 协议相关。我们在 13.15 节中直观地感受了 708
使用 UNIX 套接字的网络编程。在 13.16 节中简要地介绍了高层网络服务（如网络文件系统）。

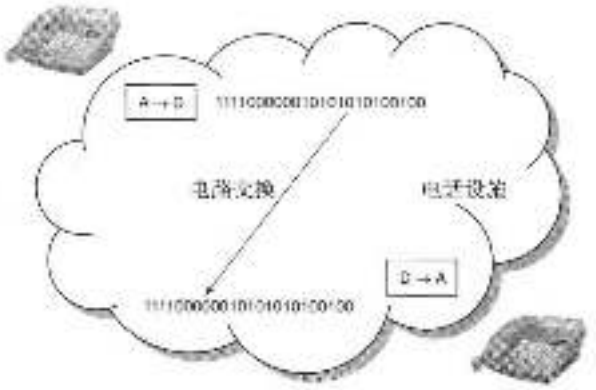
本章最后以计算机网络的历史回顾结尾。

历史回顾

我们以早期的计算作为网络演变之旅的开始。

从电话到计算机网络

首先让我们回顾电话的演变历史，因为计算机网络继承了电话的大量优点。在 20 世纪 60 年代之前，电话设施完全是模拟的。也就是说，当你拿起电话呼叫某人时，连接两台设备的线路携带着实际的语音信号。原则上只需要将一对耳机连接进电话线路就可以窃听私人谈话。在 20 世纪 60 年代，电话技术从模拟切换到了数字。在这种系统中，电话设施将模拟语音信号转换为数字数据，并通过线路发送比特。现在的电话网络仍然使用电路交换（参见 13.7.3 节），但是音频信号以 0 和 1 的形式发送。在接收端的电话设施将数字数据转换回原来的模拟语音信号，并通过电话将语音信号提供给终端用户（见图 13-55）。



[709] 图 13-55 使用电路交换的数字电话

在计算机的演变历史中，20 世纪 60 年代是大型机（mainframes）的时代。这些机器运行在面向批处理的多道程序设计环境中，并使用穿孔卡片（punched card）作为输入/输出的介质。之后，基于阴极射线管（CRT-based）的显示设备和键盘取代了穿孔卡片，成为用户与计算机进行交互的输入/输出介质。这开启了大型机的交互（interactive）计算与分时（time-shared）操作系统的时代（见图 13-56）。

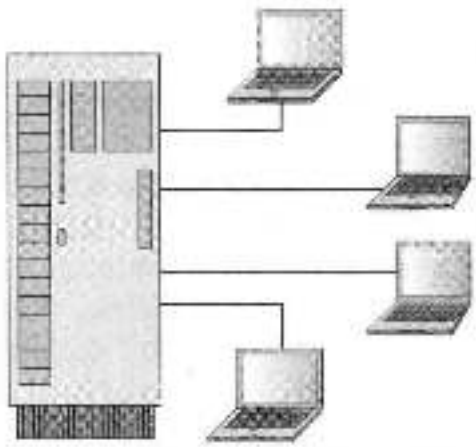


图 13-56 连接到一台大型机的多个终端

伴随着电话的发展，阴极射线管的出现开辟了一种新的可能性——显示设备不需要靠近大型计算机，它可以位于一个远程位置，如用户的家中。毕竟电话设施也携带数字数据，且它并不真正关心线路上的比特到底是语音还是数据。但不幸的是，电话设施假定输入/输出是模拟的（因为它本来用于传输语音），即使其内部通信也全部使用数字。因此，在远程显示设备与电话设施之间存在一个缺失环节，这也同样存在于电话设施与大型机之间（见图 13-57）。这个缺失环节就是调制解调器（modem），它可以在发送端将数字数据转换为模拟信号，并在接收端将模拟信号转换为数字数据。当然，数据传输的两个方向都需要这个数字→模拟→数字转换过程。

1962 年，AT&T 贝尔实验室（AT&T Bell Labs）推出了第一个商业化的全双工（full-duplex）调制解调器，它在每一端都能同时进行调制和解调（见图 13-58）。这标志着电信

（telecommunication）的诞生。1977年，Dennis Hayes发明了PC调制解调器，将终端和调制解调器之间的通信语言标准化。这为在线与因特网行业的萌芽和成长奠定了基础，并确定了调制解调器的工业标准。

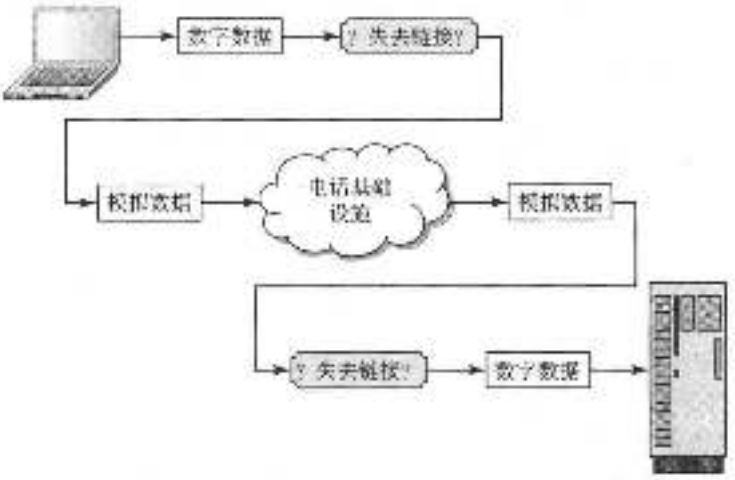


图13-57 即使电话设施内部也使用数字数据，但电话设施的输入/输出是模拟的（语音）

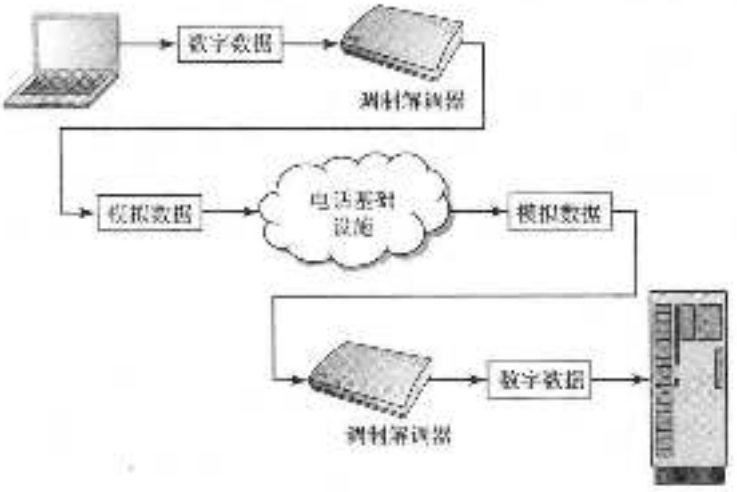


图13-58 调制解调器将终端连接到大型机

因特网的演变

许多计算机先驱都热衷于创造计算机远距离通信技术，军方出于战略原因也对实现这个技术很感兴趣。从20世纪60年代初期开始，伴随着使用电话线连接终端与大型机的调制解调器的发展，学术界与工业界中都有许多敏锐的头脑思考如何远距离连接计算机。1968年，一个美国联邦资助的机构美国国防部高级计划署（Advanced Research Projects Agency，ARPA）在这些学术界与工业界的敏锐头脑的帮助下，起草了一份关于第一个计算机网络的计划。第一个称为高级计划署网络（Advanced Research Projects Agency Network，ARPAnet）的分组交换计算机网络连接了4台计算机，第一台在加利福尼亚大学洛杉矶分校（UCLA），第二台在

斯坦福大学（Stanford），第三台在加利福尼亚大学圣巴巴拉分校（UC-Santa Barbara），第四台在犹他大学（University of Utah）。这个网络中的“路由器”实际上是接口报文处理机（Interface Message Processor，IMP），它由 BBN 公司（Bolt，Beranak，and Newman，Inc.）制造。IMP 系统的体系结构要求硬件与软件要能细致平衡，并在这些计算机中作为存储转发的分组交换机使用。IMP 之间使用调制解调器以及租用的电话线相互连接。1969 年，ARPAnet 进行了首次测试，网络大师 Leonard Kleinrock 从 UCLA 向斯坦福大学成功发送了第一条网络消息。

当然，经过这次最初的测试之后，ARPAnet 很快就蓬勃发展成了当今的因特网。其中一个主要的发展是可靠通信协议 TCP/IP，它由网络先驱 Vinton Cerf 和 Robert Kahn 发明。他们在 2004 年被授予计算机科学的最高奖项，图灵奖（Turing Award），以表彰他们在网络通信协议发展中的贡献。我们需要注意，从 20 世纪 70 年代中期到 70 年代末，本章所讨论的基本网络思想，包括传输协议与存储转发分组交换路由，都完整地用于因特网。20 世纪 80 年代早期，TCP/IP 协议栈得益于加利福尼亚大学伯克利分校（University of California，Berkeley）的软件分发工作，进入了 UNIX 操作系统。随后由于 UNIX 的普及，网络协议栈先后出现在台式机市场和服务器市场，并很快成为各种版本 UNIX 中的一个标准功能。即使是 IBM 也在 20 世纪 80 年代初期决定与威斯康星大学（University of Wisconsin-Madison）合作，将网络协议栈纳入它的 VM 操作系统。[⊖]因此，几乎所有主要的计算机供应商都在他们的计算机中采用了网络协议栈作为标准。但是，因特网在 20 世纪 90 年代后期才真正成为一个家喻户晓的名字，有两个原因导致了这段时间延迟。第一，当时的计算机并没有普及到个人，直到 PC 被发明并在商业上获得成功。第二，当时在因特网上没有杀手级应用，直到万维网的诞生。今天，即使是我们的奶奶都可能会告诉我们一台没有连接到因特网的计算机实际上没有什么用处。因特网的爆炸性增长也催生了许多公司，如专注于生产专业路由器的思科（CISCO）。

710
~
712

当然，因特网的演变历史完全足够写成一整本书，但是我们的目的仅仅是概述目前为止我们所经历的旅程。我们只是简单地从网络的演变来体会贯穿本书的 3 个主题——体系结构，操作系统以及网络。

个人计算机与局域网的出现

1972 年，一家名为施乐的文案管理公司（Xerox Corporation）设计出了世界上第一台个人计算机（Personal Computer，PC），名叫 Alto，它以 Palo Alto 研究中心（PARC）命名。

20 世纪 70 年代中期，在施乐公司 PARC 工作的 Metcalfe 和 Boggs 发明了以太网，以便使一栋建筑内的计算机能够相互通信。这标志着局域网（LAN）的诞生。但具有讽刺意味的是，施乐公司从来没有销售 PC 或以太网技术。其他公司，如 Apple 和 IBM，捡起了 PC 的想法，并改变了历史。1979 年，Metcalfe 创办了 3Com 公司以发展局域网市场，并成功地说服了计算机行业采用以太网作为局域网标准。

局域网的演变

粗缆网络（thicknet） 以太网的物理介质是同轴电缆（coaxial cable）（见图 13-59），最内层的厚铜线承载信号，外层的同轴导体接地（由白色绝缘层分隔）。最早期的以太网使用了粗同轴电缆（因此称为粗缆网络）和刺穿式搭接器（vampire taps）（见图 13-60）来连接每台计算机。同轴电缆贯穿整个复杂的办公室，把所有的计算机连接在一起。连接办公室计算机与以

⊖ 这本书的第一作者在威斯康星大学读研究生时，在这个 IBM 项目中实现了邮件传输协议 SMTP。

太网的是连接单元接口（Attachment Unit Interface，AUI）电缆，其长度至多可达到 50 米。我们习惯使用 xBASEy 表示法来表示以太网连接的类型。例如，10BASE5 是指以太网支持 10 兆比特 / 秒的数据传输速率，使用基带信号，任意两台计算机之间的最大距离为 500 米。使用刺穿式搭接器的粗缆网络在 1979 年～1985 年使用。

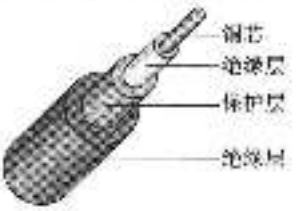


图 13-59 同轴电缆

细缆网络（thinnet）顾名思义，这种以太网使用细同轴电缆作为以太网介质，并使用 BNC 接头[⊖]将计算机连接到同轴电缆。由于电缆相对较细（这意味着更大的电阻，因此信号随距离的衰减也更严重），细缆网络中任意两台计算机之间的最大距离为 200 米。如图 13-61 所示，逻辑总线将这些电缆连接成菊花链（daisy chain）。细缆网络（数据传输速率为 10 兆比特 / 秒的细缆网络表示为 10BASE2）盛行于 1985 ～ 1993 年之间。

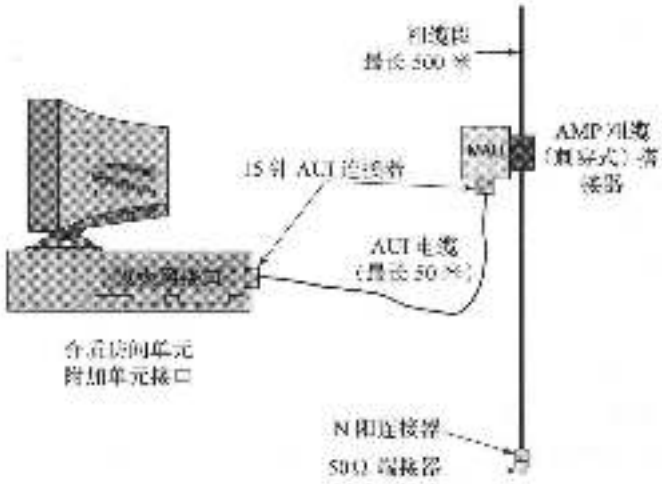


图 13-60 计算机通过刺穿式搭接器连接到以太网同轴电缆。通过同轴电缆连接的任意两个节点之间的最大距离为 500 米

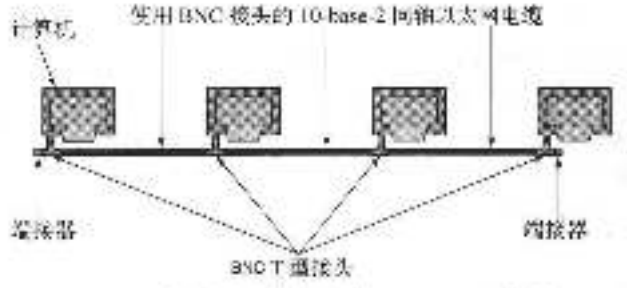


图 13-61 菊花链形式的细缆 10BASE2 以太网

快速以太网（fast Ethernet）采用菊花链形式的局域网会出现维护问题。例如，即使只有一个 BNC 接头松落，也会破坏整个局域网。电路设计的发展以及超大规模集成电路（VLSI）改变了这一局面。在 20 世纪 90 年代初期，电气工程师设计出了集线器，它是一个多路复

⊖ BNC 接头是一种很常见的视频连接方法，其缩写表示 Z 口锁（Bayonet mount locking）以及发明者的名称（Neill and Concelman）。

用器/收发器。集线器在逻辑上等价于一条总线，它有多个端口，每台计算机都连接到其中一个端口。随着电话和模块化插口（称为RJ45接头）的发展，以太网改为使用类似于电话线的双绞线（twisted pair of wires），线路两端使用RJ45接头连接计算机与集线器（见图13-32）。这显著地改变了局域网的样貌。部署以太网LAN只需要简单地将计算机连接到集线器，并将集线器连接在一起（见图13-33）。集线器在逻辑上将所有的计算机连接到了一条总线上。电缆的长度也变得无关紧要，因为计算机只需要连接到短距离（几十米）内的集线器。100BASE-T（T表示双绞线）通常称为快速以太网，是指拥有100兆比特/秒的数据传输速率，使用双绞线连接计算机与集线器的以太网。

1GBase-T与10GBase-T 吉比特以太网自2009年起就成为局域网互连的标准。支持高带宽网络连接的网卡使用双绞线连接主机与交换机。双绞线的最大长度为100米。

练习题

1. 分别从通信的两个方向描述调制解调器的功能。
2. 对比以太网和令牌环网络。
3. 描述令牌环网络的基本功能。
4. CSMA/CD这个缩写是什么意思，描述这个协议。
5. 区分无线以太网协议和有线以太网协议。
6. CSMA/CA中的冲突避免是什么，它是如何实现的？
7. 对比以下设备：网卡、集线器、中继器、网桥、交换机、路由器。
8. 对比网络协议栈与OSI七层模型。
9. 我们为什么需要网络协议？
10. 区分电路交换和虚电路。
11. 一个知识渊博的计算机工程师设计了一种使用网桥制造的交换机（如本章所讨论的），但为了节省成本，换使用了中继器来替代网桥。你会买这样的交换机吗？为什么会或者为什么不会？
12. 滑动窗口的目的是什么？
13. TCP/IP等协议如何处理乱序到达、数据包丢失，以及重复的数据包等问题？
14. 为什么会发生网络堵塞，TCP怎么处理网络拥塞？
15. IP网络指的是什么？假设你要创办一家公司，需要将2000台计算机连接到因特网，你如何向ISP申请IP地址？你公司的网络地址如何以点分十进制表示法表示？
16. 校验和如何使用，它为什么是必要的？
17. 一条消息有13个数据包，从源向目的地发送一个数据包的时间为2毫秒。假设发送数据包和接收ACK的时间与介质中的传播时间相比可以忽略不计，且没有发生数据包丢失，那么窗口大小为5的滑动窗口协议需要多少时间来完成数据传输？
18. 考虑下列条件：

    消息大小=1900 Kb
    包头大小=1000 b
    数据包大小=20 Kb
    线路带宽=400 000 b/s
    传播时间=2 s
    窗口大小=8
    发送端的处理延迟=0
    接收端的处理延迟=0

713 ↓ 715

ACK 大小 = 忽略不计（视为 0）。

假设网络中没有错误并且数据包按序到达，那么完成消息传输总共需要多少时间？

[提示：注意窗口大小决定了在任意时刻处于传输状态中的数据包的最大数量。当所有数据包都已经发送出去之后，发送端就会开始等待剩余的 ACK 到达以完成消息传输。] 716

19. 考虑下列传输层和网络层条件：

数据包大小 =20 000 b

ACK 大小 = 忽略不计（视为 0）

（单独确认每个数据包）

传输窗口大小 =20

发送端的处理延迟 =0.025 s 每个数据包（不包括 ACK）

接收端的处理延迟 =0.025 s 每个数据包（不包括 ACK）

数据包丢失几率 =0%

数据包出错几率 =0%

线路带宽 =400 000 b/s

传播时间 =4 s

这个传输层要完成 400 个数据包的传输（包括接收 ACK 包）总共需要多少时间？

[提示：注意从发送端的工作周期来看，所有的端到端延迟成分（如 13.12 节中的式（13-1）所示）都可以混合在一起。]

20. 下面是一个数据包包头的各字段大小：

目的地址	8B
源地址	8B
消息中数据包数	4B
序号	4B
实际数据包大小	4B
校验和	4B

假设数据包的最大大小为 1100 字节，计算数据包的最大有效载荷。

21. 考虑下列条件：

发送端开销 =1 ms

消息大小 =200 000 b

线路带宽 =100 000 000 b/s

传播时间 =2 ms

接收端开销 =1 ms

计算观测到的带宽。消息传输时间由发送端开销、传输延迟、传播时间以及接收端开销组成。忽略 ACK。 717

22. 考虑下列条件：

消息大小 =100 000 B

包头大小 =100 B

数据包大小 =1100 B

假设有 10% 的数据包会丢失，要完成消息传输总共需要发送多少个数据包？忽略小数部分的丢包概率。忽略 ACK。

23. 考虑一个使用累积 ACK 的可靠的流水线传输协议。其窗口大小为 10。接收端根据以下规则发送 ACK：

• 如果按序接收到 10 个连续的数据包，就发送一个 ACK，从而使发送端能够向前推进滑动窗口。

• 如果接收到的数据包的序号与顺序不符，则发送一个当前已经收到的最高序号的数据包的 ACK。

a. 发送端发送了 100 个数据包，每 10 个数据包中会有 1 个在传输中丢失。假设没有 ACK 包丢失。要完成消息传输实际需要发送多少个 ACK 包？

b. 如果不使用累积 ACK，需要发送多少个 ACK 包？即假设协议对每个数据包进行单独确认；如果收到一个乱序的数据包，则重新发送之前的最后一个 ACK 包。

参考文献注释和扩展阅读

1989 年，Tim Berners-Lee 发明了万维网 [Berners-Lee，1989]。然而连接网址的点和由来已久。1945 年，一篇题为 "As We May Think" 的文章描述了 Vannevar Bush 的远见，他预言了一个由设备（称为 memex）构成的复杂的信息交流网络以扩展人大脑的未来 [Bush，1945]。20 世纪 60 年代，3 个不同的研究小组（MIT，Rand Institute 以及 National Physical Laboratory in England）分别独立地发明了分组交换 [Baran，1964；Kleinrock，1961；Kleinrock，1964]。它是因特网数据通信技术的核心。1969 年，BBN 科技制造了第一代分组交换机，称为接口报文处理机（Interface Message Processor，IMP）。它使计算机能够相互通信。1969 年年底，ARPAnet[⊖]中的 4 个节点成为通过 IMP 实现连接！20 世纪 70 年代初期，除了 ARPAnet，还出现了其他几个分组交换网络，如 ALOHAnet[Abramson，1970] 以及 BBN 公司推出的 ARPAnet 的商业版本 Telenet。在此期间，Robert Metcalfe 在他 1973 年的博士论文中提出了以太网的原理[⊜]，这是此后以太网发展的起点。此时，连接所有这些独立的网络岛屿的时机已经成熟。Vincent Cerf 与 Robert Kahn 完成了这个开拓性的工作 [Cerf，1974]，在其中他们提出了网络互连的体系结构，于是我们今天所知道的因特网诞生了。这个体系结构指定了 TCP 作为主要的传输协议，它推动因特网发展至今。[⊝]所有的这些发展，再加上计算机与网络的硬件与软件的进步，促成了 Tim Berners-Lee 发明的万维网。

Berkeley UNIX 中的 TCP/IP 协议栈实现细节可以在 [McKusick，1996] 中找到。[Stevens，1994；Wright，1995] 这些书是细致学习 TCP/IP 协议及其实现的极好资源。对于渴望了解更多的同学，可以参考计算机网络方面的许多优秀教科书 [Kurose，2006；Tanenbaum，2002]。对网络编程感兴趣的同学会发现 [Comer，2000；Stevens，2003；Bryant，2003] 这些教科书是极好的资源。

718
，
719

⊖ ARPA 全称为 Advanced Research Projects Agency，而 ARPAnet 可以认为是当今因特网的始祖。

⊜ Metcalfe 和 Boggs 在 1976 年发表了描述以太网协议的论文 [Metcalfe，1976]。

⊝ 想要了解因特网横跨 1962—1992 的 30 年历史，请访问 http://www.computerhistory.org/internet_history/。

尾声：旅途回顾

本书已经走完“计算机内部”（inside the box）之旅，应该回顾一下我们所涵盖的主题，并看清系统软件与硬件之间的密切关系。

14.1 处理器设计

我们已经看到，高级语言结构在指令集设计与处理器体系结构的功能设计中起到了关键作用。表 14-1 总结了影响处理器体系结构决策的高级语言结构与功能。

表 14-1 高级语言与体系结构功能

高级语言结构 / 功能	体系结构功能
表达式	算术 / 逻辑指令，寻址模式
数据类型	不同的操作数粒度，多种精度的算术 / 逻辑指令
条件语句，循环	有条件和无条件的分支指令
过程调用 / 返回	栈，链接寄存器
系统调用，错误条件	陷入，异常
高级语言程序的执行效率（表达式以及参数传递等）	通用寄存器

我们已经看到，当体系结构被完全指定后，仍然会有好几种实现选择，如简单的设计与流水线设计。公平地说，我们只触及了微体系结构这个有趣领域的表面。希望我们已经充分地激发了读者的兴趣，使他们能够自行深入挖掘这个迷人的区域。

14.2 进程

进程这一概念作为一种方便的抽象，用于记忆所有与正在运行程序的相关细节。我们知道处理器在同一时间只能执行一个程序。操作系统提供了每个进程执行在各自处理器上的假象。为了方便操作系统制造这种假象，体系结构提供了陷入与中断机制来从当前执行的程序取回控制权。操作系统通过这些机制来获取处理器的控制权，以做出调度决策来决定接下来执行的用户程序。我们已经看到许多操作系统可能会采用的计算调度决策算法，以及操作系统为了实现这些算法所需要的数据结构。陷入机制使用户级程序能够通过系统调用来使用系统所提供的功能（如文件系统和打印功能），以扩展程序自身的功能。

在 Linux 或者 Windows XP 等产业化操作系统中的处理器调度组件要比本书中所介绍的复杂得多。本书的目的是为读者提供足够的视野，以便了解如何构建操作系统的调度子系统。

14.3 虚拟内存系统和内存管理

内存系统对计算机系统的性能起着至关重要的作用。因此，在本书中需要特别注意操作系统的内存管理组件与体系结构对内存管理的协助之间的相互影响。为了支持内存分配、保护、隔离、共享以及高效的内存利用等内存管理功能，我们介绍了一些体系结构技术，如硬

栏寄存器、界限寄存器、基址和限长寄存器，以及分页。我们也已经看到了操作系统对于在体系结构中支持特权模式（内核模式）的需求，以使在处理器进行程序调度之前准备好所需要的内存区域。我们讨论了许多关于操作系统的问题（如页面替换策略和工作集的维护），这些问题对于应用程序的性能至关重要。

我们从处理器管理（通过调度算法）与内存管理（通过内存管理策略）中学习到的主要经验是操作系统需要尽可能快地做出决策。操作系统应该迅速提供程序所需要的资源，然后迅速闪开！

720
~
721

这些讨论中的一个有趣的启示是，即使是最复杂的内存管理方案（如分页），以及非常高效的页面替换算法与工作集维护算法，也几乎不需要硬件提供额外的支持。这也再次强调了理解系统软件与硬件之间的合作关系的重要性。

14.4 分级存储体系

分级存储体系是内存系统的重要主题。程序的局部性（自然会导致程序的工作集概念）是让程序在现代计算机系统的大量内存中实现良好性能的关键特性。体系结构中利用了局部性的功能是分级存储体系。高速缓存的概念贯穿于系统设计的各个方面（从 Web 缓存到处理器缓存）。我们已经看到如何利用程序局部性在硬件中为指令和数据设计缓存。我们也已经看到如何在页表级别利用程序访问的局部性设计地址缓存（TLB）。

14.5 并行系统

并行是人类的基础思想，为此也是我们开发算法和程序的方式。从早期计算开始，系统软件和计算机设计师都在追求并行性。随着集成水平的不断提高，计算机现在可以容纳多个处理器。事实上，现在流行的单芯片处理器都是多核的，即在一块硅芯片上有多个处理器。

考虑到这些发展趋势，计算机科学家都需要了解并行编程所需要的系统软件支持与硬件协助。我们从操作系统的角度探讨了一些主题，如在单一的地址空间中支持多线程管理，并在这些线程之间进行同步与数据共享。我们也研究了体系结构中的增强功能，包括在顺序处理器中的原子读出 - 修改 - 写入原语，以及在对称多处理器中的高速缓存一致性。

这个主题非常丰富，可以想象这是值得深入探讨的。希望我们已经激发了读者进一步探索的兴趣。

14.6 输入 / 输出系统

不能与外界进行交互的计算机系统实际上没有什么用处。我们已经从硬件的角度看到了如何使用程控 I/O 和 DMA 等技术将（简单或复杂的）设备接入计算机系统。硬件的中断机制是获取处理器注意的关键。我们也已经看到内存映射 I/O 技术如何无缝地集成 I/O 子系统，且不需要扩充处理器的指令集架构。设备控制器镜像将设备连接到处理器上，操作系统中称为
722 设备驱动程序和中断处理程序的软件模块负责操控连接到系统的输入 / 输出设备。

需要特别关注的是两个特殊的 I/O 子系统，即磁盘和网络。前者用于在计算机中进行信息的永久性存储，后者用于与外界交流。

14.7 永久性存储

文件系统是操作系统的一个重要组成部分。毫不夸张地说，它很可能是产业化操作系统

中代码行数最多的一个子系统。由于其固有的简单性，对于任何与程序进行交互的输入/输出设备，文件都是一种方便的表示形式。特别是，存储在磁盘等允许信息永久保存的介质上的文件可以超出程序的生命周期存在。我们探讨了在设计文件系统时所做出的选择，包括文件的命名以及属性。我们重点探索了磁盘这一介质的空间分配策略、磁盘调度策略，以及在磁盘上的文件组织（包括操作系统中的数据结构）。

14.8 网络

随着因特网与万维网的出现与发展，当今将计算机系统连接到外部世界已经是理所当然的事情。因此，我们投入了相当多的时间分别从系统软件与硬件的角度来了解网络中的问题。在硬件方面，我们学习了构成当今网络环境的网络设备，如网卡、集线器、交换机以及路由器。从系统软件的角度，我们理解了对于协议栈的需求，其中包括传输层、网络层以及数据链路层。我们还学习了其中的各种机制，如校验和、滑动窗口以及序号。例如，我们使用纠错码来应对当线路数据包出错时所造成的数据丢失。为了使应用程序的有效载荷能够符合固定数据包大小等硬件限制，并处理数据包的乱序到达，协议栈纳入了数据包的分散/收集功能。协议使用端到端的确认来应对传输途中的数据包丢失。

网络和网络协议是有待进一步研究的迷人区域。因此，我们希望本书所涵盖的网络内容能够激励读者的兴趣以寻求更多的知识。

结束语

总之，系统体系结构这一领域是硬件和软件之间的交汇点，它是一块有趣且迷人的区域。这块区域在很长的一段时间内都不会下降，因为日常生活中的计算应用正在跨越式地发展。为了跟上这种需求的增长，系统架构师需要不断创新并制造新的计算机，孜孜不倦地追求速度更快、成本更低、功耗更低，并能够提供更多服务的计算机。我们希望本书能够成为未来的系统架构师的起点。

723

使用 UNIX 套接字进行网络编程

A.1 问题描述

使用 UNIX 套接字为两个进程（客户端与服务器）编写一个简单的客户端－服务器通信程序。客户端执行在一台因特网名称是 beehive.cc.gatech.edu 的机器上，服务器执行在一台因特网名称是 mit.edu 的机器上，服务器套接字所使用的端口号是 2999。套接字使用 TCP/IP 作为通信的底层协议。客户端在连接完成后发送字符串“Hello World! Client is Alive!”。当收到这条消息后，服务器回复字符串“Got it! Server is Alive!”。客户端和服务器都会把它们成功发送以及接收的消息输出到标准输出（stdout），然后关闭连接并结束。

A.2 源文件提供

后面各节给出了客户端代码、服务器代码以及 Makefile。对于你的目标机器，确保使用正确的二进制输出文件。为了简单起见，客户端和服务器使用相同的目标机器。例如，在 x86_64 架构上编译的二进制文件无法在 i686 架构上运行，除非你使用了“-m32”兼容选项（尝试使用“uname -m”来查看体系结构信息）。附录中提供的 Makefile 文件使用“-m32”作为默认值。

如果你不了解示例中的任何系统调用，使用 UNIX 帮助文档来查看详细解释。如果你不了解“man”命令本身，那么输入“man man”。例如，如果你想在任何 UNIX 机器上了解套接字系统调用，那么输入“man socket”。

A.3 Makefile

```
#########################################################
##
## Makefile: CS2200 Client/Server Example
##                 Using UNIX Sockets
## Author:    Junsuk Shin
##
#########################################################
CFLAGS  = -Wall -pedantic -m32
LFABCS  = -m32
CC      = gcc
RM      = /bin/rm -rf

SERVER  = server
CLIENT  = client

SERVER_SRC  = server.c
CLIENT_SRC  = client.c
SRCS    = ${SERVER_SRC} ${CLIENT_SRC}

SERVER_OBJ  = ${patsubst %.c,%.o,${SERVER_SRC}}
CLIENT_OBJ  = ${patsubst %.c,%.o,${CLIENT_SRC}}
```

```
OBJS    = $(SERVER_OBJ) $(CLIENT_OBJ)

# pattern rule for object files
%.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@

#-----------------------------------------------------------
all: $(SERVER) $(CLIENT)

$(SERVER): $(SERVER_OBJ)
        $(CC) $(LFAGS) -o $@ $<

$(CLIENT): $(CLIENT_OBJ)
        $(CC) $(LFAGS) -o $@ $<

clean:
        $(RM) $(OBJS) $(SERVER) $(CLIENT) core*

.PHONY: depend
depend:
        makedepend -Y -- $(CFLAGS) -- $(SRCS) 2>/dev/null

# DO NOT DELETE

server.o: example.h
client.o: example.h
```

A.4 共用的头文件

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
#define SERVER_PORT 2999
#define SERVER_MSG  "Got it!  Server is Alive!"
#define CLIENT_MSG  "Hello World!  Client is Alive!"

#define MAXPENDING  5
#define BUFF_SIZE   128

#endif
```

A.5 客户端源代码

```
#include "example.h"
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>

const char usage[] = "Usage: client [-h] [-p <server port>]
[-s <server address>]\n";

char *server_addr = NULL;
int server_port = SERVER_PORT;

void print_usage(void) {
    printf("%s",usage);
    exit(EXIT_FAILURE);
}

void read_options(int argc, char *argv[]) {
```

```
    int c;

    /* read command line options */
    while ( (c=getopt(argc,argv,"hp:s:")) != -1 ) {
        switch(c) {
        case 'p':
            server_port = atoi(optarg);
            if ( server_port <= 0 ) {
                print_usage();
            }
            break;
        case 's':
            server_addr = optarg;
            break;
        case 'h':
        default:
            print_usage();
        }
    }
    if ( server_addr == NULL ) {
        print_usage();
    }
}

int connect_to(char *host, int port) {
    int sock;
    struct addrinfo hint;
    struct addrinfo *addr;
    char port_str[8];

    /*
     * making connection from a client side is simpler than
     * a server side
     * It follows 2 steps:
     *      1) make socket (socket)
     *      2) make connection (connect)
     */
    memset(&hint, 0, sizeof(hint));
    hint.ai_family = PF_INET;
    hint.ai_socktype = SOCK_STREAM;
    snprintf(port_str,8,"%d",port);

    /* First, need to find out network address with a given
     * host name.  host can be any form such as
     * host name -- tokyo.cc.gatech.edu or
     * dotted decimal notation -- 192.168.1.1.
     * When a descriptive name is needed, gethostbyname()
     * or gethostbyname_r() can be used.  The behavior of
     * gethostbyname() when it is passed a numeric address
     * string as a parameter is unspecified.
     * For the dotted decimal notation,
     * inet_addr() can be used.
     * Since the return value from the above two
     * system calls (gethostbyname and inet_addr) are
     * different, the return value
     * should be handled differently.
     * In this example, getaddrinfo() is used, and it
     * simply handles both cases.
     */
    /* The getaddrinfo() function shall translate the name
     * of a service location (for example, a host name)
     * and/or a service name and shall return
     * a set of socket addresses and associated information
     * to be used in creating a socket with which to
```

```
     * addresses the specified service.
     */
    getaddrinfo(host, port_str, &hint, &addr);

    /* Second, make socket
     * (use addrinfo set by getaddrinfo() )
     * it can be simply
     * socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) for a
     * tcp/ip connection.
     */
    if((sock = socket(addr->ai_family,
                      addr->ai_socktype,
                      addr->ai_protocol)) == -1) {
        perror("socket");
        return -1;
    }

    /* Third, connect to a socket
     * Since, getaddrinfo() sets the relevant fields of
     * addr variable, we can simply use them for this call.
     * For example, if gethostbyname() is used to resolve
     * the network address, you might code the connect
     * call differently.
     */
    if(connect(sock,
               addr->ai_addr,
               addr->ai_addrlen) == -1 ) {
        perror("connect");
        return -1;
    }

    free(addr);

    return sock;
}

int main(int argc, char *argv[]) {
    int     socket;
    int     sent_size, recv_size;
    char    buffer[BUFF_SIZE];
    read_options(argc,argv);

    socket = connect_to(server_addr, server_port);
    if ( socket <= 0 ) {
        return EXIT_FAILURE;
    }

    /* send a message on a socket */
    /* Usually, recv/send doesn't fail, but always need to
     * check (good programming habit).
     */
    sent_size = send(socket,
                CLIENT_MSG,
                strlen(CLIENT_MSG)+1,0);

    if ( sent_size == -1 ) {
        perror("send");
        exit(EXIT_FAILURE);
    }
    printf("Message sent to %s\n\t%s\n",
                                server_addr,CLIENT_MSG);

    /* Receive a message from a connected socket */
```

```
    /*
     * buffer for the message needs to be provided.
     * It returns the length of the message written to
     * the buffer unless there is an error.
     * By default, it's blocking call.  If interested,
     * check out fcntl(), O_NONBLOCK, and EAGAIN return
     * value.
     */
    recv_size = recv(socket,buffer,BUFF_SIZE,0);
    if ( recv_size == -1 ) {
        perror("recv");
        exit(EXIT_FAILURE);
    }
    printf("Message received from %s\n\t%s\n",
                                    server_addr,buffer);

    /* Close sockets */

    /*
     * Not really necessary, since the program terminates,
     * but it's a good habit.
     * Same for file descriptors and
     * sockets.  Also, there's a limit for such
     * descriptors that a user can open.  Simply,
     * you cannot
     * open file/make socket beyond the limit.
     */
    close(socket);
    return EXIT_SUCCESS;
}
```

A.6 服务器源代码

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "example.h"

union sock {
    struct sockaddr s;
    struct sockaddr_in i;
};

const char usage[] =
                "Usage: server [-h] [-p <port number>]\n";
int port = SERVER_PORT;
char client_ip[16];

void print_usage(void) {
    printf("%s",usage);
    exit(EXIT_FAILURE);
}

void get_options(int argc, char *argv[]) {
    int c;

    /* read command line options      */
    /* "hp:" means -h and -p <string> */
    while ( (c=getopt(argc,argv,"hp:")) != -1 ) {
```

```
        switch(c) {
        case 'p':
            port = atoi(optarg);
            if ( port <= 0 ) {
                print_usage();
            }
            break;
        case 'h':
        default:
            print_usage();
        }
    }
}

int create_server_socket(int port) {
    struct sockaddr_in serv_addr;
    int serv_sock;
    int opt = 1;

    /* Creation of server socket usually follows these 3
     * steps:
     *     1) make socket / create endpoint of
     *        communication,
     *     2) bind a name (address) to a socket, and
     *     3) listen for incoming socket connections.
     */
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = PF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(port);

    /* make TCP socket */
    /* PF_INET : IP protocol family
     *           For more options, check out
     *                          /usr/include/bits/socket.h
     *            (e.g.,  PF_UNIX for unix domain socket)
     * SOCK_STREAM : sequenced, reliable connection
     *            (e.g.,  SOCK_DGRAM for connectionless,
     *                    unreliable connection)
     * IPPROTO_TCP : Transmission Control Protocol/TCP
     *            (e.g.,  IPPROTO_UDP, IPPROTO_RSVP, etc.)
     *                check out /usr/include/netinet/in.h
     *                          /usr/include/linux/in.h
     */
    if ((serv_sock=socket(PF_INET,
                          SOCK_STREAM,
                          IPPROTO_TCP)) == -1 ) {
        perror("socket");
        return -1;
    }

    /* Set port as reusable */

    /*
     * Port may not be usable if it's not closed properly
     * (e.g.,  segfault, kill process) Specifies that the
     * rules used in validating addresses supplied to
     * bind() should allow reuse of local addresses.
     */
    if (setsockopt(serv_sock,
                   SOL_SOCKET,
                   SO_REUSEADDR,
                   &opt,
                   sizeof(opt)) == -1 )
```

```
    {
        perror("setsockopt");
        return -1;
    }

    /* Bind a name (server_addr) to a socket (serv_sock).
     * When a socket is created with socket(), it exists in
     * a name space (address family), but has no name
     * assigned.
     *
     * It normally is necessary to assign a
     * local address by
     * using bind before a SOCK_STREAM socket may receive
     * connections (accept()).
     */
    if (bind(serv_sock,
            (struct sockaddr *)&serv_addr,
            sizeof(serv_addr)) == -1 ) {
        perror("bind");
        return -1;
    }

    /*
     * Listen for socket connections and limit the queue of
     * incoming.
     */
    if (listen(serv_sock,MAXPENDING) == -1 ) {
        perror("listen");
        return -1;
    }

    return serv_sock;
}

void read_client_ip(int sock) {
    union sock client;
    int client_len;

    client_len = sizeof(struct sockaddr);
    /* get the name of the peer socket */
    getpeername(sock,
                &(client.s),
                (socklen_t *)&client_len);

    /* inet_ntoa() convert the Internet host address to a
     * string in the Internet standard dot notation.
     */
    strncpy(client_ip,inet_ntoa(client.i.sin_addr),16);
}

int main(int argc, char *argv[]) {
    int                 server_sock, client_sock;
    int                 recv_size,sent_size;
    struct sockaddr_in  c_addr;
    unsigned int        c_len = sizeof(c_addr);
    char                buffer[BUFF_SIZE];

    get_options(argc,argv);

    server_sock = create_server_socket(port);

    /* Waiting on a client connection */

    /*
```

```
 * The accept function is used with connection-
 * based socket types:
 * (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM).
 * It extracts the first connection request on the
 * queue of pending connections, creates a new
 * connected socket with mostly the same properties as
 *  s, and allocates a new file descriptor for the
 * socket, which is returned.
 */
if ( (client_sock=accept(server_sock,
                         (struct sockaddr *)&c_addr,
                         &c_len)) == -1 ) {
    perror("accept");
    return EXIT_FAILURE;
}

read_client_ip(client_sock);

/* Receive a message from a connected socket */

/*
 * Buffer for the message needs to be provided.
 * It returns the length of the message written to
 * the buffer unless there is an error.
 * By default, it's blocking call.  If interested,
 * check out fcntl(), O_NONBLOCK, and EAGAIN return
 * value.
 */
recv_size = recv(client_sock,buffer,BUFF_SIZE,0);
if ( recv_size == -1 ) {
    perror("recv");
    exit(EXIT_FAILURE);
}
printf("Message received from %s\n\t%s\n",
                                    client_ip,buffer);

/* Send a message on a socket */
/* Usually, recv/send doesn't fail, but always need
 * to check (good programming habit!).
 */
sent_size = send(client_sock,
                 SERVER_MSG,
                 strlen(SERVER_MSG)+1, 0);
if ( sent_size == -1 ) {
    perror("send");
    exit(EXIT_FAILURE);
}
printf("Message sent to %s\n\t%s\n",
                                client_ip,SERVER_MSG);

/* Close sockets */

/*
 * Not really necessary, since the program terminates,
 * but it's good habit.  Same for file descriptors and
 * sockets.  Also, there's a limit for such
 * descriptors that a user can open.  Simply,
 * you cannot
 * open file/make socket beyond the limit.
 */
close(client_sock);
close(server_sock);
```

```
    return EXIT_SUCCESS;
}
```

A.7 客户端－服务器程序执行示例

示例程序在服务器端使用以下默认端口：

• 服务器端口：2999

这个选项可以通过命令行参数改变。

为了执行该程序，你需要运行客户端程序和服务器程序：

• 服务器

运行服务器程序（如果需要，可以使用不同的参数）

例如

```
./server（使用默认类型运行）
./server -p 3333（将服务器的端口号设置为3333）
```

• 客户端

运行客户端程序（如果需要，可以使用不同的参数）

例如

```
./client -s < host ip | host name >
./client -p 3333 -s helsinki.cc.gatech.edu.
```

725 / 735

参考文献

[Abramson, 1970] N. Abramson, The ALOHA system—Another alternative for computer communications, *FJCC 1970 Fall Joint Comp. Conf.*, AFIPS Press, Vol. 37, pp. 281–285.

[Adve, 1996] Sarita V. Adve and Kourosh Gharachorloo, Shared memory consistency models: A tutorial, *Computer*, December 1996, pp. 66–76.

[Agarwal, 1995] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung, The MIT Alewife Machine: Architecture and performance, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 22–24, 1995.

[Allen, 1987] Randy Allen and Ken Kennedy, Automatic translation of FORTRAN programs to vector form, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 9, No. 4, October 1987, pp. 491–542.

[Almasi, 1993] George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Series in Computer Science and Engineering, San Francisco.

[Archibald, 1986] J. Archibald and J. Baer, Cache coherence protocols: Evaluation using a multiprocessor simulation model, *ACM Trans. Comput. Syst.*, Vol. 4, No. 4, September 1986.

[ARM, 1990] *RISC Acorn RISC Machine Family Data Manual*, CORPORATE VLSI Technology Inc., San Jose, CA, Prentice Hall, Upper Saddle River, NJ.

[Backus, 1954] J. W. Backus, Preliminary Report, *Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*, Programming Research Group, Applied Science Division, International Business Machines Corporation, New York, November 10, 1954. Available at http://archive.computerhistory.org/resources/text/Fortran/102679231.05.01.acc.pdf.

[Backus, 1957] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. J. Stern, I. Ziller, R. A. Hughes, and R. Nutt, The FORTRAN automatic coding system, in *Proceedings Western Joint Computer Conference*, Los Angeles, February 1957, pp. 188–198.

[Baran, 1964] P. Baran, On distributed communications networks, *IEEE Transactions on Communications Systems*, Vol. 12, No. 1, March 1964, pp. 1–9.

[Barham, 2003] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 19–22, 2003. SOSP, ACM, New York, pp. 164–177.

[BBN Butterfly, 1986] BBN, *Butterfly Parallel Processor Overview*, Technical Report 6148, BBN Laboratories Incorporated, Boston, March 1986.

[Belady, 1966] Laszlo A. Belady, A study of replacement algorithms for virtual-storage computer, *IBM Systems Journal*, Vol. 5, No. 2, 1966, pp. 78–101.

[Belady, 1969] Laszlo A. Belady, Robert A. Nelson, and Gerald S. Shedler, An anomaly in space–time characteristics of certain programs running in a paging machine, *CACM*, Vol. 12, No. 6, June 1969, pp. 349–353.

[Bell, 1978] C. G. Bell, R. Cady, H. McFarland, B. Delagi, J. O'Laughlin, R. Noonan, and W. Wulf, A new architecture for mini-computers—The DEC PDP-11, *Proceedings of the Spring Joint Computer Conference*, AFIPS Press, Atlantic City, NJ, May 5–7, 1970, pp. 657–675.

[Bell Webpage, 2010] *Gordon Bell's CyberMuseum for Digital Equipment Corp (DEC)*: Documents, Photo Albums, Tales, and Videotapes about Computing History, 2010. Available at http://research.microsoft.com/en-us/um/people/gbell/Digital/DECMuseum.htm.

[Berners-Lee, 1989] Tim Berners-Lee, *Information Management: A Proposal*, CERN, Geneva, Switzerland, March 1989. Available at http://www.w3.org/History/1989/proposal.html.

[Blackberry OS, 2010] http://www.blackberry.com/.

[Bobrow, 1972] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, TENEX, A paged time sharing system for the PDP-10, *CACM*, Vol. 15, No. 3, 1972.

[Bovet, 2005] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, 3rd edition, O'Reilly, Cambridge, MA.

[Bryant, 2003] R. E. Bryant and David O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice Hall, Upper Saddle River, NJ.

[Burkhardt, 1992] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie, *Overview of the KSR 1 Computer System*, Tech. Rep. KSR-TR-9202001, Kendall Square Res., Boston, February 1992.

[Burks, 1981] Arthur W. Burks and Alice R. Burks, The ENIAC: The first general-purpose electronic computer, *IEEE Annals of the History of Computing*, Vol. 3, No. 4, 1981, pp. 310–389, commentary on pp. 389–399.

[Bush, 1945] Vannevar Bush, As we may think, *The*

Alliance, July 1999. Available at http://www.sheetlarks.com/devel/194507/brass.

[Carr, 1991] R. W. Carr and J. L. Hennessy, WSCLOCK—A simple and effective algorithm for virtual memory management. *SIGOPS Oper. Syst. Rev.* Vol. 15, No. 5, December 1981, pp. 87–95.

[Cerf, 1974] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, Vol. Com-22, No. 5, May 1974, pp. 637–648.

[Chapman, 2007] B. Chapman, G. Jost, and R. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. MIT Press, Cambridge, MA.

[Cocke, 2000] John Cocke and V. Markstein. The evolution of RISC technology at IBM, *IBM journal of R&D*, Vol. 44, Nos. 1–2, p. 35–55, January 2000.

[Comer, 2000] Douglas E. Comer and David L. Stevens, *Internetworking with TCP/IP Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version*. Prentice Hall, Upper Saddle River, NJ.

[Cooper, 1988] E. C. Cooper and R. P. Draves, *C Threads*. CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, June 1988.

[Culler, 1998] David Culler, J.P. Singh, and Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA.

[Dean, 2004] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6*, San Francisco, December 6–8, 2004. Available at http://www.usenix.org/events/osdi04/tech/full_papers/dean/dean_html/.

[Denning, 1968] Peter J. Denning. The working set model for program behavior, *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 323–333.

[Eckert, 1946] J. Presper Eckert and John Mauchly. *Outline of Plans for Development of Electronic Computers*. Document submitted to the U.S. Army. Available at http://www.computerhistory.org/collections/accession/102650910.

[Adler, 1986] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Teller, and James Wilson. Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 17–19, 1985, Boston, pp. 126–135.

[Flynn, 1966] M. J. Flynn, Very high-speed computing systems, *Proceedings of the IEEE*, Vol. 54, No. 12, December 1966, pp. 1901–1909.

[Foster, 2003] Ian Foster and Carl Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. The Elsevier Series in Grid Computing, 2nd edition, Morgan Kaufmann, San Francisco.

[Fox, 1990] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D Language specification*, Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, Houston, TX, December 1990.

[Kuskin, 1994] David Ojcius, David J. Kuck, Duncan J. Lawrie, and Ahmed H. Sameh. Cedar: A large scale multiprocessor, in *Proceedings of the International Conference on Parallel Processing*, 1983, pp. 524–529, Columbus, OH.

[Hennecher, 2001] Carl Hamacher, Zvonko Vranesic, and Safwat Zaky, *Computer Organization*, Computer Science Series, McGraw-Hill, Columbus, OH.

[Hennessy, 1981] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture, in Proceedings, *CMU Conference on VLSI Systems and Computations*, pp. 337–346, Computer Science Press, October 1981.

[Hennessy, 2002] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Francisco.

[Hord, 1982] R. Michael Hord. *The Illiac IV: The First Supercomputer*, Computer Science Press, Rockville, MD.

[IBM System/360, 1964] *IBM System/360 Principles of Operation*, IBM Press, Armonk, NY.

[IBM System/370, 1978] Architecture of the IBM System/370, *Communications of the ACM*, Vol. 21, No. 1, special issue on computer architecture, January 1978, pp. 73–96.

[Intel C++, 2008] Intel Corporation, Concurrent Collections. Available at http://software.intel.com/en-us/articles/intel-concurrent-collections-for-c/.

[Intel Instruction set, 2008] Intel Corporation, *The Intel® 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference A–M*, Order Number 253666, *Instruction Set Reference N–Z*, Order Number 253667, November 2008. Also available online at http://www.intel.com/products/processor/manuals/.

[Intel System programming guide, IA, 2008] Intel Corporation, *The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, Order Number 253668, November 2008. Also available online at http://www.intel.com/products/processor/manuals/.

[iPhone OS X, 2010] http://www.apple.com/iphone/.

[Johnson, 1995] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, CRL: High-performance all-software distributed shared memory, in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 3–6, 1995. ACM, New York, pp. 213–228.

[Jones, 1996] Richard Jones and Rafael D. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, Hoboken, NJ.

[Jul, 1988] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, Fine-grained mobility in the Emerald system, *ACM Transactions on Computer Systems (TOCS)*, Vol. 6, No. 1, February 1988, pp. 109–133.

[Katz, 2004] Randy H. Katz and Gaetano Borriello, *Contemporary Logic Design*, 2nd edition, Prentice Hall, Upper Saddle River, NJ.

[Keleher, 1994] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel, TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *USENIX*, Winter 1994, pp. 115–132.

[Kernighan, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 1st edition, Prentice-Hall, Englewood Cliffs, NJ, February 1978.

[Kleinrock, 1961] Leonard Kleinrock, *Information Flow in Large Communication Nets*, RLE Quarterly Progress Report, Massachusetts Institute of Technology, Cambridge, MA, July 1961. Available at http://www.cs.ucla.edu/~lk/REPORT/RLEreport.1961.html.

[Kleinrock, 1964] Leonard Kleinrock, *Communication Nets: Stochastic Message Flow and Design*, McGraw-Hill, NY.

[Kontothanassis, 2005] L. Kontothanassis, R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, and M. L. Scott, Shared memory computing on clusters with symmetric multiprocessors and system area networks, *ACM Transactions on Computer Systems*, Vol. 23, No. 3, August 2005. Available at http://www.cs.rochester.edu/research/cashmere/.

[Krieger, 2006] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig, K42: Building a Complete Operating System, *EuroSys 2006*, Leuven, Belgium, April 2006. Available at http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html.

[Kuck, 1976] David J. Kuck, Parallel processing of ordinary programs, *Advances in Computers*, Vol. 15, Academic Press, New York, pp. 119–179. Available online at http://books.google.com.

[Kung, 1978] H. T. Kung and C. E. Leiserson, Systolic arrays (for VLSI), in *Proc. SIAM Sparse Matrix Symp.*, Knoxville, TN, 1978, pp. 256–282.

[Kung, 1979] H. T. Kung, The structure of parallel algorithms, Carnegie Mellon University Technical Report: CMU-CS-79-143, *Advances in Computers*, Vol. 19, Academic Press, New York. Available online at http://books.google.com.

[Kurose, 2006] James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, Boston.

[Lamport, 1979] Leslie Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Transactions on Computers*, Vol. C-28, No. 9, September 1979.

[Lenoski, 1992] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam, The Stanford Dash Multiprocessor, *Computer*, Vol. 25, No. 3, March 1992, pp. 63–79.

[Li, 1988] K. Li, IVY: A shared virtual memory system for parallel computing, in *Proceedings of the 1988 ICPP*, Vol. II, University Park, PA, pp. 94–101.

[Lilja, 1993] David J. Lilja, Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons, *ACM Computing Surveys*, Vol. 25, No. 3, September 1993, pp. 303–338.

[Love, 2003] Robert Love, *Linux Kernel Development*, SAMS publishing, (A subsidiary of Prentice Hall), Upper Saddle River, NJ).

[Mac OS X, 2010] http://www.apple.com/macosx/.

[Mahon, 1986] M. Mahon, R. Lee, T. Miller, J. Huck, and W. Bryg, Hewlett-Packard precision architecture: The processor, *Hewlett-Packard Journal*, Vol. 37, No. 8, August 1986, pp. 4–21.

[Mano, 2007] M. Morris Mano and Charles Kime, *Logic and Computer Design Fundamentals*, 4th edition, Prentice Hall, Upper Saddle River, NJ.

[McKusick, 1984] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, A fast file system for UNIX, *ACM Trans. Comput. Syst.*, Vol. 2, No. 3, August 1984, pp. 181–197.

[McKusick, 1996] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, Boston.

[McKusick, 2004] Marshall Kirk McKusick and George V. Neville-Neil, *The Design and Implementation of the Free BSD Operating System*, Addison-Wesley, Boston.

[Mellor-Crummey, 1991] J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Syst.* Vol. 9, No. 1, February 1991, pp. 21–65.

[Metcalfe, 1976] Robert M. Metcalfe and David R. Boggs, Ethernet: Distributed packet switching for local computer networks, *Communications of the ACM*, Vol. 19, No. 7, July 1976, pp. 395–404.

[Mitchell, 1994] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia, An overview of the Spring system, *IEEE COMPCON '94*. Available at http://research.sun.com/features/tenyears/volref/papers/mitchell.ham.

[Moore, 1965] Gordon E. Moore, Cramming more components onto integrated circuits, *Electronics*

Magazine, Vol. 38, No. 8, April 19, 1965. Available at ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[MPI, 2009] Message Passing Interface Forum, http://www.mpi-forum.org/

[Nichols, 1996] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly, Cambridge, MA.

[Oliphint, 1987] C. Oliphint, Operating system for the B 5000, *IEEE Ann. Hist. Comput.* Vol. 9, No. 1, January 1987, pp. 23–28.

[OpenMP, 2010] http://openmp.org/wp/.

[Organick, 1972] Elliott I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA.

[Padua, 1986] David A. Padua, David J. Kuck, and Duncan H. Lawrie, High speed multiprocessors and compilation techniques, *IEEE Trans Computers* Vol. 29, No. 9, pp. 763–776.

[Patt, 2004] Y. N. Patt and S. J. Patel, Introduction to Computing Systems: from bits & gates to C & beyond, 2nd Edition, McGraw Hill, New York.

[Patterson, 1980] David A. Patterson and David R. Ditzel, The case for the reduced instruction set computer, *ACM SIGARCH Computer Architecture News*, Vol. 8, No. 6, October 1980, pp. 25–33.

[Patterson, 1981] David A. Patterson and Carlo H. Sequin, RISC I: A reduced instruction set VLSI computer, Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, pp. 443–457.

[Patterson, 1998] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann, San Francisco.

[Patterson, 2008] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th edition, Morgan Kaufmann, San Francisco.

[PDP-8, 1973] Digital Equipment Corp. *PDP-8/E, PDP-8/M & PDP-8/F Small Computer Handbook*, PDP-8 Handbook Series, Digital Equipment Corp., Maynard, MA.

[Pfister, 1985] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture, in Proceedings of the *International Conference on Parallel Processing*, University Park, PA, August 1985, pp. 764–771.

[Prabhakaran, 2005] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, Analysis and evolution of journaling file systems, Proceedings of the *USENIX Annual Technical Conference*, April 10–15, 2005, Anaheim, CA.

[Radin, 1982] G. Radin, The 801 minicomputer, Proc. *Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39–47.

[Reid, 2001] T. R. Reid, *The Chip: How Two Americans Invented the Microchip and Launched a Revolution*, Random House, New York.

[Ritchie, 1974] D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *CACM*, Vol. 17, No. 7, July 1974, pp. 365–375.

[Rubini, 2001] Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers*, O'Reilly, Cambridge, MA.

[Russinovich, 2005] Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th edition, Microsoft Press, Redmond, Washington.

[Saltzer, 2009] Jerome H. Saltzer and M. Frans Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, San Francisco.

[Scales, 1996] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath, Shasta: A low overhead, software-only approach for supporting fine-grain shared memory, *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, MA, October 1996, pp. 174–185.

[Schroeder, 1971] M. D. Schroeder, Performance of the GE 645 associative memory while Multics is in operation, in Proceedings of the SIGOPS *Workshop on System Performance Evaluation*, ACM, New York, pp. 227–245.

[Silberschatz, 2008] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating System Concepts*, 8th edition, Wiley, Hoboken, NJ.

[Sites, 1992] Richard L. Sites, Alpha AXP architecture, special issue, 1992 Alpha AXP architecture and systems, *Digital Technical Journal*, Vol. 4 No. 4.

[Smith, 1982] A. J. Smith, A. J. Cache memories, *ACM Comput. Surv.*, Vol. 14, No. 3, September 1982, pp. 473–530.

[Smith, 1985] A. J. Smith, Disk cache—Miss ratio analysis and design considerations, *ACM Trans. Computer Systems*, Vol. 3, No. 3, August 1985, pp. 161–203.

[Snir, 1998] Marc Snir and William Gropp, *MPI: The Complete Reference*, MIT Press, Cambridge, MA.

[SPARC Architecture, 2010] SPARC International, Inc., 2010. http://www.sparc.com/specificationsDocuments.html.

[Stallings, 2010] William Stallings, *Computer Organization and Architecture: Designing for Performance*, 8th edition, Prentice Hall, Upper Saddle River, NJ.

[Stevens, 1994] W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Boston.

[Stevens, 2003] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API*, 3rd edition, Addison-Wesley, Boston.

[Stewart, 1998] John W. Stewart III, *BGP4: Inter-domain Routing in the Internet*, Addison-Wesley, Boston.

[Strecker, 1978] W. D. Strecker, VAX-11/780: A virtual address extension to the DEC PDP-11 family, *Proceedings of the National Computer Conference*, AFIPS Press, Montvale, NJ, 1978, pp. 967–980.

[Sunderam, 1990] V. S. Sunderam, PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience*, Vol. 3, No. 4, December 1990, pp. 315–339.

[Swan, 1977] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, Cm*: A modular, multi-microprocessor, in *Proceedings of the June 13–16, 1977, National Computer Conference*, Dallas, TX, AFIPS 1977, ACM, New York, pp. 637–644.

[Symbian OS, 2010] http://www.symbian.org/

[Tanenbaum, 1987] Andrew S. Tanenbaum and Albert S. Woodhull, *Operating Systems Design and Implementation*, 3d edition, Prentice Hall, Upper Saddle River, NJ.

[Tanenbaum, 2002] Andrew S. Tanenbaum, *Computer Networks*, 5th edition, Prentice Hall, Upper Saddle River, NJ.

[Tanenbaum, 2005] Andrew S. Tanenbaum, *Structured Computer Organization*, 5th edition, Prentice Hall, Upper Saddle River, NJ.

[Tanenbaum, 2006] Andrew S. Tanenbaum, Albert S. Woodhull, *Operating Systems Design and Implementation*, 3d edition, Prentice Hall, Upper Saddle River, NJ.

[Tanenbaum, 2007] Andrew S. Tannenbaum, *Modern Operating Systems*, Prentice Hall, Upper Saddle River, NJ.

[Thornton, 1964] James E. Thornton, Parallel operation in the control data 6600, *Proc. AFIPS Fall Joint Computer Conferences, Part II*, October 27–29, 1964, pp. 33–40.

[Tomasulo, 1967] R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development*, Vol. 11, No. 1, p. 25.

[Toomey, 1988] L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, IBM Parallel FORTRAN, *IBM Systems Journal*, Vol. 27, No. 4, 1988, pp. 416–435.

[Torvalds, 1991] Linus Torvalds: posting to the comp.os.minix. Available at http://www.linus.org/people/linus_post.html.

[Ward, 1989] Stephen A. Ward and Robert H. Halstead, *Computation Structures*, MIT Press, Cambridge, MA.

[Wilkes, 1951] M. V. Wilkes, D. J. Wheeler, and S. Gill, *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Boston.

[Wilkes, 1971] M. V. Wilkes, Slave memories and dynamic storage allocation, *IEEE Trans. Computers*, Vol. C-20, No. 6, June 1971, pp. 674–673.

[Windows CE, 2010] http://www.microsoft.com/windowsembedded/en-us/default.mspx

[Windows Version 7, 2010] http://www.microsoft.com/windows/windows-7/

[Wright, 1995] Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, Boston.

[Wulf, 1972] W. A. Wulf and C. G. Bell, C.mmp: A multi-mini processor, in *Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, Part II*, Anaheim, CA, AFIPS 1972, Fall, part II, ACM, New York, pp. 765–777.

[Wulf, 1974] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, HYDRA: The kernel of a multiprocessor operating system, *Communications of the ACM*, Vol. 17, No. 6, June 1974, pp. 337–345.

[Yeh, 1992] Tse-Yu Yeh and Yale N. Patt, Alternative implementations of two-level adaptive branch prediction, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp. 124–134.

索　引

索引中所标页码为英文版原书页码，与页边栏中的页码一致。

A

Abstraction levels（抽象级），2-5
Access network（接入网），622
Access rights（访问权限），348-349，473-474
Access time（访问时间），94，119，353
Access violation trap（访问违例陷入），349
Accumulator（累加器），60，122
Acknowledgment number（确认序号），687
Activation record（活动记录），54
Active window（活动窗口），644，645
Actuator（执行器），440
ADD instruction（ADD 指令），103-106
ADDI instructions（ADDI 指令），111
Address，disk（地址，磁盘），477
Address computations（地址计算），307
Address offset（地址偏移量），38
Address space（地址空间），237，525
Address translation（地址转换）
  in Intel Pentium（在 Intel 奔腾处理器），313
  in MULTICS（多路信息和计算机服务），311
  with segmentation（采用分段机制），304
  supporting architecture and（支持体系结构），310
  with translation lookaside buffer（使用旁路转换缓存），344-346
  from virtual address to physical address（从虚拟地址到物理地址），293-294
Addressability of operands（操作数可寻址性），22-23，28
Addressing modes（寻址模式），24，35-37，58-59
Advanced graphics port（AGP）（高级图形接口），463
Advanced Research Projects Agency Network（ARPANET），712
Advanced synchronization algorithms（高级同步算法），586-589
Advanced technology attachment（ATA）（高级技术附件规格），450
Algol programming language（Algol 编程语言），18
Aliases，file system attributes and（别名，文件系统属性），471，474
Alignment of word operands（字操作数的对齐），32-34
Alignment restrictions（对齐限制），34
Allocation table（分配表）
  after P2 releases memory（在 P2 释放内存后），289
  after P1's completion（在 P1 完成之后），288
  after P1's request satisfied（在 P1 的请求被满足后），286-287
  after series of memory requests（在处理了一系列内存请求之后），288
  for fixed-size partitions（固定尺寸分区），286
  before P2 releases memory（在 P2 释放内存前），288
Alpha architecture（Alpha 体系结构），66
Alternate control unit design style（控制单元设计的另一种选择），119-121
Amdahl，Gene，169，226
Amdahl's law（阿姆达尔定律），169-170，226，613
Apache, web server（Apache,Web 服务器），633
Apple Mac OS X，275
Application domains（应用领域），266-267
Application hardware-operating system Interactions（应用硬件-操作系统交互），5-8，7f
Application，instruction-set design and（应用，指令集设计），18-19，67-68
Application layer（应用层），628，632-634
Application program interface(API)（应用程序接口），498，634
Applications，launching of（应用，启动），10-11
Arbitrary message size（任意消息大小），627
Arbitration logic（仲裁逻辑），670
Architectural enhancements，program discontinuities and（体系结构增强，程序不连

续表), 135-143
Architecture styles (体系结构风格), 59
Arithmetic logic unit (ALU) (算术逻辑单元), 22, 79, 175
Arithmetic mean (AM) (算术平均数), 162-163, 170
ARM architecture (ARM 体系结构), 123
Arrays (数组), 35-37
Arrival time, process (到达时间, 进程), 247
ASCII format (ASCII 码格式), 424
Association for Computer Machinery (ACM), 16
Associative memory (相关存储器), 345
Asynchronous events (异步事件), 130-131
Asynchronous logic circuits (异步逻辑电路), 91
Asynchronous mode transfer (ATM) (异步模式传输), 678
Asynchronously produced data (异步生成数据), 428
Athlon, AMD processor (Athlon, AMD 处理器), 210, 222
Atomic instruction (原子指令), 142
Atomic test-and-set instruction (原子的 Test-and-Set 指令), 575-577
Atomicity (原子性), 268, 543-544, 582
Attachment unit interface (AUI) cable (连接单元接口电缆), 713
Attenuation, electrical signals and (电信号衰减), 679
Audio data rates (音频数据传输速率), 439
Autoincrement (自动递增模式), 111
Autonomous system (AS) (自治系统), 657, 659
Average rotational latency (平均旋转延迟), 446
Average seek time (平均寻道时间), 446
Average turnaround time (平均周转时间), 245
Average waiting time (平均等待时间), 245, 452

B

Babbage, Charles, 16, 273
Back end, microarchitecture (后端, 微体系结构), 223-224
Backplanes (背板), 77
Backus, John, 16
Backward compatability (向后兼容), 66
Bandwidth (带 宽), 393, 429, 431, 461-462, 465, 604, 640, 644-645, 650, 653, 663-664, 676, 678, 690
Barcelona, AMD processor (Barcelona, AMD 处理器), 419, 611
Bardeen, John, 12
Barrier synchronization (障碍同步), 539
Base and limit registers, memory management And (基址 - 限长寄存器, 内存管理), 284, 285
Base band signaling technique (基带信号技术), 673
Base+offset addressing mode (基址 + 偏移量寻址模式), 27, 35-37
Basic blocks (基本块), 214-215
Basic computer organization (基本的电脑组成), 22
Basic input/output system (BIOS) (基本输入 / 输出系统), 464, 503-504
Batch-oriented operating systems (批处理操作系统), 13, 273
Bayonet Neill-Concelman (BNC), 714-715
Belady, Laszlo, 327
Belady's Min algorithm (Belady 的 Min 算法), 326-327
Bell Laboratories (贝尔实验室), 12
Benchmarks (基准测试程序), 161-165
BEQ instruction (BEQ 指令), 112-116, 187
Berkeley RISC, 74, 123, 232
Best fit (最佳适应), 289
Big endian (大端), 31-32
Binary instructions (双操作数指令), 22
Binary semaphore (二元信号量), 576-577
Blackberry mobile phones (黑莓手机), 17
Blackberry OS (黑莓操作系统), 13, 17
Block multiplexer channel (组块多路复用信道), 434
Block-offset (块偏移量), 378
Block-oriented devices (面向块的设备), 438-440
Block size (块大小)
  cache organization and (缓存组织结构), 369, 378
  miss rate behavior and (缺失率行为), 383
  multiword (多字), 379-381
  performance implications of (性能影响), 383-384
Blocked state (阻塞状态), 535
Blue Gene, IBM, 226, 601, 604
Bolt Beranak and Newman (BBN), 604, 613, 618, 712, 718

Boolean logic functions（布尔逻辑函数）, 119-120
Boot block, file system（启动块，文件系统）, 506
Booting up（开启）, 464
Bootstrapping（引导程序）, 464
Border gateway protocol（BGP）（边界网关协议）, 657
Bottom-half handlers, Linux（Bottom-half 处理过程，Linux）, 154
Bounds registers, memory management and（界限寄存器，内存管理）, 282-284
Branch handling techniques（分支处理技术）, 200-211
Branch instruction（分支指令）, 38-39
Branch prediction（分支预测）, 205-208, 210
Branch target buffer（分支目标缓存）, 208
Brattain, Walter H., 12, 15
Bridges, networking hardware（网桥，网络硬件）, 679-681
Bubble in pipeline（流水线中被引入的气泡）, 187
Bubbles, RAW data hazards and（气泡，写后读数据冒险）, 196, 199
Buffering（缓冲）, 178
Burroughs Corporation（Burroughs 公司）, 310
Burst time, CPU（突发时间，CPU）, 252
Bus arbitration（总线仲裁）, 670
Bus-based design（基于总线的设计）
  single bus design（单总线设计）, 86-87
  two bus design（双总线设计）, 87-89
Buses（总线）
  bus arbitration（总线仲裁）, 432
  bus clock line（总线时钟线）, 433
  bus to match cache block size（与缓存块大小相匹配的总线）, 406
  cycle time and（时钟周期）, 393
  I/O bus evolution（I/O 总线的发展）, 461-463
  master-slave relationship and（主从关系）, 433
  peripheral component interchange（PCI）and（外围组件互连）, 432
  split transaction（分离传输）, 433
  standard bus and system bus coexistence（标准总线与系统总线共存）, 432-433
Busy waiting, thread synchronization and（忙等待，线程同步）, 548, 560
Byte offset, caches and（字节偏移量，缓存）, 367

C

C-LOOK algorithm, disk scheduling and（C-LOOK 算法，磁盘调度）, 455-457
C programming language（C 语言）, 28-29
C-scan（circular scan）, disk scheduling and（C 扫描（循环扫描），磁盘调度）, 455
Cable modem（电缆调制解调器）, 622
Caches. *See also* Memory hierarchy（缓存，参见多级存储体系）
  cache blocks（缓存块）, 366, 388
  cache coherent multiprocessors（缓存一致的多处理器）, 604
  cache line（缓存行）, 366-369
  cache lookup（缓存查找）, 363-365
  cache memories（缓存）, 158
  concept of（概念）, 354-355
  controller（控制器）, 400
  data caches（D-cache）（数据缓存，D 缓存）, 392-393
  directory-based schemes and（基于目录的方案）, 605
  entry fields（缓存项的字段）, 365
  fully associative（全相关）, 385-387
  indices（索引）, 363, 365-366
  instruction caches（I-cache）（指令缓存，I 缓存）, 392
  integrating TLB and（TLB 整合）, 399-400
  main memory organization and block size（主存结构与块大小）, 406
  memory consistency models and（内存一致性模型）, 607-610
  memory location copy in（拷贝内存内容）, 180
  memory stalls and（内存延迟）, 376-377
  misses in processor pipeline（处理器流水线的缺失）, 375-377
  organization and（组织）, 360-369
  performance, spatial locality to improve（提升性能，空间局部性）, 377-384
  read access to cache from CPU（CPU 对缓存的

读访问), 370
read/write algorithms (读/写算法), 370-375
replacement policy and (替换策略), 394-396
set associative (组相关), 387
snoopy caches (侦听缓存), 580-581
virtually indexed physically tagged (虚拟索引物理标记的), 401-402
virtually tagged (虚拟标记的), 403
write access to cache from CPU (CPU 对缓存的写访问), 370-375
write miss in MEM stage (在 MEM 阶段没有写命中), 375-376
Cactus stack (仙人掌栈), 567
Call/return procedure (过程调用/返回), 48-53
Call teardown (呼叫拆除), 667
Callee (被调用者), 42, 46-47
Caller (调用者), 42
Capacity misses (容量缺失), 385, 396-397
Carrier sense multiple access/collision detect (CSMA/CD)(载波监听多路访问/冲突检测), 671-673
Cascaded interrupts (级联中断), 138-141
Cathode-ray-terminal (CRT) -based display Devices (基于 CRT 的显示设备), 710
CD-R, 508-509
CD-ROM, 508-509
CD-RW, 439-440
CDC 6600, 219
Cell, DRAM storage (单元, DRAM 存储), 408
Cell processor (Cell 处理器), 68
Central processing unit (CPU)(中央处理单元)
bursts (突发), 242-243, 260-261
burst time (突发时间), 252
cache lookup and (缓存查找), 363-365
communication with I/O devices (与 I/O 设备通信), 423-427
datapath design and (数据通路设计), 91
handling write-miss and (处理写缺失), 380
interpreting CPU-generated address (解释 CPU 生成的地址), 306-307
main memory connection to (主存连接到), 393
memory access path and (内存访问路径), 400
memory address generation and (内存地址生成), 378-379
multiword direct-mapped cache organization and (多字直接映射缓存组织结构), 379-380
power consumption and (功耗), 221-222
read access to cache from (对缓存的读访问), 370
scheduling and system performance (调度和系统性能), 416
simple main memory and (简单的主存), 405
time quantum and (时间量子), 416
utilization and (利用率), 245
write access to cache from (对缓存的写操作), 370-375
Cerf, Vinton, 712
Channels (信道), 434
Character-oriented devices (面向字符的设备), 438-440
Checksum (校验和), 685
chgrp, UNIX command (chgrp, UNIX 命令), 473, 475
Chip multiprocessors (芯片多处理器), 221-222
chmod, UNIX command (chmod, UNIX 命令), 473, 475, 518
Circuit switching (电路交换), 663-664, 667, 669
Circuits, combinational and sequential logic and (电路, 组合和时序逻辑), 78
Circular wait (循环等待), 583-584
Clean and dirty page distinction (干净页和脏页的区分), 320
Clear to send (CTS)(允许发送), 674
Client (客户端), 3
Clock algorithm (时钟算法), 335
Clock cycle (时钟周期), 81, 84-86
Clock cycle times (时钟周期时间), 159, 165, 216, 217
Clock pulse width (时钟脉冲宽度), 94
Clock signal, register and (时钟信号, 寄存器), 80
Clocks per instruction(CPI)(每指令的时钟周期数), 159, 166, 186
Cloud computing (云计算), 14, 255
Cluster parallel machine (集群并行机器), 604
Clusters (集群), 226, 517, 615
CMOS, 12, 123, 465

Co-routines (协程), 569
Coarse-grain critical sections (粗粒度临界区), 544
Coarse-grained parallelism (粗粒度并行度), 603
Coaxial cable (同轴电缆), 630, 713-714
Cocke, John, 158
Code fragment for thread creation (线程创建的代码片段), 526
Code refinement (代码优化)
  with coarse-grain critical sections (粗粒度临界区), 544
  with fine-grain critical sections (细粒度临界区), 545-546
Code with no synchronization (无同步的代码), 541-543
Cold misses (冷缺失), 396
Collision avoidance (冲突避免), 674
Collision detection (冲突检测), 672
Collision domain (冲突域), 672, 679-680
Column access strobe (CAS), DRAM and (列访问选通信号, DRAM), 409-410
Combinational logic, circuits and (组合逻辑, 电路), 78
Combinational logic delays (组合逻辑延迟), 180
Commit (提交), 514
Common data bus (CDB) (公共数据总线), 220
Communication among threads (通过线程通信), 526-527
Communication deadlocks (通信死锁), 586
Compaction (缩并), 289-290
Complete interrupt handler (完整的中断处理过程), 142
Completed execution, microarchitecture and (执行完毕, 微体系结构), 218
Completion order, microarchitecture and (完成顺序, 微体系结构), 217-218
Complex instruction set computers (CISC) (复杂指令集计算机), 66, 123, 157
Compulsory miss (强制缺失), 385, 396-397, 462
Compute bound workload (计算受限的负载), 266
Computer hardware evolution (计算机硬件的演化), 11-12
Computer peripherals, data rates of (计算机外围设备, 数据传输速率), 439
Computer systems (计算机系统)
  abstraction levels of (抽象层次), 2-5
  application launching and (启动应用程序), 10-11
  hardware organization and (硬件组织), 8-10, 9f
  I/O and system bus and (I/O 和系统总线), 11
  instantiations of (实现形式), 8, 8f
Concurrency (并发), 554-555
Concurrency problems (并发问题), 592-596
Concurrent processes, sharing of memory by (并发进程, 共享内存), 280
Condition variable data type (条件变量数据类型), 551-552
Condition variables (条件变量), 548-552, 586
Conditional branch engineering in microprogram (设计微程序中的条件分支), 116-117
Conditional statements and loops
  as high-level language feature (高级语言功能集中的条件语句和循环), 21
  if-then-else statements (if-then-else 语句), 38-40
  loop statements (循环语句), 41-42
  switch statements (switch 语句), 40-41
Conflict miss (冲突缺失), 363, 396-397
Content addressable memory (CAM) (内容寻址存储器), 345
Context switching (上下文切换), 259, 268, 316
Contiguous allocation (连续分配), 477-480
Contiguous words (连续的字), 378
Control pipeline hazards (控制流水线冒险)
  branch prediction and (分支预测), 205-208
  branch prediction with branch target buffer (带分支目标缓存的分支预测), 208
  delayed branch and (延迟分支), 203-205
  handling branches and (处理分支), 200-203
Control program monitor (控制程序监控器), 274
Control signal table (控制信号表), 97
Control unit design (控制单元设计), 95-96
Convoy effect (护送效应), 247, 249
Core memory (芯存储器), 122
Counting semaphore (计数信号量), 577
CPU brust time (CPU 突发时间), 252, 255
Cray, Seymour, 226

Cray computer（Cray 计算机），465，613
Cray series（Cray 系列），226
Creation time，file system attributes and（创建时间，文件系统属性），471，474
Critical section（临界区），536-537，560
Cycle count（时钟周期数），159
Cycle stealing（周期窃用），431
Cycle time（时钟周期时间），353

D

D-MEM，176，179
Daemon processes（守护进程），278
Daisy chaining（菊花链），144-145
DARPA（Defense Advanced Research Projects Agency）（美国国防部先进研究项目局），226
Data（数据）
caches（缓存），392-393
direct memory access(DMA)and（直接内存访问），429-431
directory files and（目录文件），507
forwarding（转发），192-197
high-level abstractions and（高级抽象），21
overrun flag（溢出标志位），425
precision（精确度），28
race（竞争），528，560
register（寄存器），424
storage management and（存储管理），506-507
threads library and（线程库），540-541
transfer time（传输时间），444-445
Data Hazards（数据冒险）
read after write hazard（写后读冒险），187，190-199
write after read hazard（读后写冒险），189-190，199-200
write after write hazard（写后写冒险），189-190，199-200
Dataflow architectures（数据流体系结构），122-123
Datagram（数据报），667，669，705
Datapath（数据通路）
design（设计），91-93
element connections（元件连接），82-86
enhancements for handling interrupts（处理中断的增强），143-146
instruction pipeline and（指令流水线），178-180
instruction-set architecture and（指令集体系结构），93-94
organization leading to lower CPI（减小 CPI 的组织），166
processor（处理器），79
Daughter cards（子卡），463
DBUF buffer，pipeline register（DBUF 缓存，流水线寄存器），182-183
Deadline scheduling（截止时间调度），269
Deadlocks（死锁）
absence of（缺乏），555-556
avoidance（避免），584
concept of（概念），583-586
definition of（定义），560
detection（检测），584-585
multithreaded programming and（多线程编程），546-548
prevention（预防），584
Debugging，processor design and（调试，处理器设计），69
DEC Alpha architecture（DEC Alpha 体系结构），31，58，123
DEC VAX architecture（DEC VAX 体系结构），58
DECODE macro state（DECODE 宏状态），91，102-103，116-119
Decode units（解码单元），215
Deep pipelining（深流水线），216-218
Degree of associatively（相关程度），387
Degree of multiprogramming（多道程序度），241
Delay slot（延迟槽），203-205
Delayed branch（延迟分支），203-205，210
Delays，packet-switched networks and（延迟，分组交换网络），669
Dell，61，461
Demand paging（按需分页）
data structures for（数据结构），318-319
disk map and（磁盘映射），319
frame table and（页帧表），319
free-list of page frames and（空闲页帧表），318-319
hardware for（硬件），317-318
hardware for instruction restart（为指令重启的

硬件），318
page fault anatomy（页错误解剖），320-324
page fault handler and（页错误处理程序），318
page table entry and（页表项），317
processor pipeline and（处理器流水线），317
Desktop computers（桌面计算机），266
Destination port（目标端口），687
Device drivers（设备驱动）
characteristics of（特征），434-438
dynamic loading of（动态负载），463
evolution of（演化），461-464
file systems and（文件系统），499
Difference engine（差分机），16
Digital Equipment Corporation（DEC），66
Digitizer thread，data structure and（数字化器件线程，数据结构），527
Dijkstra，Edsger，576-577
Dijkstra's link state routing algorithm（Dijkstra 链路状态路由算法），653-655
Dining philosophers，classic concurrency Problems（哲学家就餐，著名同步问题），593，595-597，617
Direct access，ISA and（直接访问，ISA），358
Direct-mapped cache organization（直接映射缓存结构），360-369
cache entry fields（缓存项中的字段），365
cache lookup（缓存查找），363-365
hardware for direct-mapped cache（直接映射缓存的硬件），366-369
with write-back policy（回写策略），373-374
Direct memory access（DMA）（直接内存访问），429-431
Directory-based schemes（基于目录的方案），581，605
Dirty bit，caches and（脏位，缓存），373
Discontinuities in program execution（程序执行中的不连续性），130-132
Disk block address（磁盘块地址），477
Disk conceptual layout（磁盘的概念性布局），505
Disk drive（磁盘驱动），449
Disk map（磁盘映射），319-320
Disk recording（磁盘记录），450
Disk request queue（磁盘请求队列），451
Disk scheduling algorithms（磁盘调度算法）
algorithm comparison（算法比较），458-459
C-scan（circular scan）（循环扫描），455
disk request queue and（磁盘请求队列），451
first-come first-served（先到先服务），453
LOOK and C-LOOK，455-457
performance metrics and（性能度量），452
SCAN（elevator algorithm）（电梯算法），453-455
shortest seek time first（最短寻道时间优先），453
Disk storage（磁盘存储），440-451
Disk technology saga（磁盘技术的传奇故事），448-451
Disk transfer latency（磁盘传输延迟），451
Dispatch（分发），243
Dispatcher（分发器），241，263
Distance vector algorithm（距离向量算法），656-657
Distributed-shared memory（DSM）（分布式共享内存），604
Dotted decimal notation（点分十进制），660
Drivers，buses and（驱动器，总线），87
DSP（digital signal processing）applications（数字信号处理应用），124
Dual in-line memory modules（DIMM）（双列直插式存储模块），412-414
Dual-ported register file（DPRF）（双端口寄存器堆），176-177，179
Dumb terminals（哑终端），462
Duplicate stack pointer（复制栈指针），147-148
DVD-R data rates（DVD-R 数据传输速率），439-440
Dynamic address translation（动态地址转换），310
Dynamic instruction frequency（动态指令频率），160-161，170
Dynamic memory allocation（动态内存分配），68
Dynamic random access memory（DRAM）（动态随机访问存储）
1 G-bit DRAM chip（1 G-bit DRAM 芯片），411，413
cache block size and（缓存块尺寸），406
dual in-line memory modules and（双列直插式存储模块），412
interleaved memory and（交错式内存），407-408

modern memory systems and (现代内存系统), 408-415
page mode DRAM (页式 DRAM), 412-413
simple memory system and (简单内存系统), 405
Dynamic relocation, memory management And (动态重定位, 内存管理), 284-285

E

EBUF buffer, pipeline register (EBUF 缓冲区, 流水线寄存器), 182-183
Eckert, J. Presper, 15, 273
Edge-triggered logic (边沿触发逻辑), 79-82
Effective memory access time (EMAT)(有效内存访问时间), 356
Electronically erasable programmable readonly memory (EEPROM)(电可擦可编程只读存储器), 460
Embedded computing (嵌入式计算), 266, 270
Endianness (字节序), 30-32
Engineering a conditional branch in microprogram (设计微程序中的条件分支), 116-117
ENIAC (Electronic Numerical Integrator and Computer)(电子数字积分计算机), 11, 11f
Enterprise computing (企业计算), 266
Error correcting codes (ECC)(纠错码), 648
Ethernet (以太网)
evolution of (历史回顾), 713-715
fast (快速), 714-715
invention of (发明), 713
protocol (协议), 670-671
EX stage, pipelining (EX 阶段, 流水线), 177-180
Exception/trap register (ETR) (异常/陷入寄存器), 134, 149
Exceptions (异常), 131-132. *See also* Program discontinuities (也参见"程序的不连续性")
Execution models (执行模型), 533
EXECUTE macro state (EXECUTE 宏状态)
ADD instruction (ADD 指令), 103-106
BEQ instruction (BEQ 指令), 112-116
JALR instruction (JALR 指令), 106-108
LW instruction (LW 指令), 108-111
NAND instruction (NAND 指令), 106
SW and ADDI instructions (SW 和 ADDI 指令), 111
Executed instructions, reduction of (执行的指令, 减少), 166
Execution core (执行核心), 223-224
Execution model for parallel program (并行程序的执行模型), 560
Execution time (执行时间), 157, 159, 170, 174-175
Expected I/O requirements (期望 I/O 需求), 238
Expected memory usage (期望内存使用), 238
Expected running time (期望运行时间), 238
Exponential backoff (指数退避), 672
Expressions and assignment statements (表达式和赋值语句)
adding registers inside processor (处理器中的加入寄存器), 22-23
alignment of word operands (字操作数的对齐), 32-34
endianness (字节序), 30-32
memory address specification (内存寻址规范), 26-27
operand location and (操作数位置), 22-26
operand packing and (操作数打包), 32-34
operand width and (操作数宽度), 27-30
Extended (ext) file system, Linux (可扩展文件系统, Linux), 509
External fragmentation (外部碎片), 287, 305, 478
External interrupts (外部中断), 219

F

Fast Ethernet (快速以太网), 742-743
Fast page mode (FPM), DRAM (快速页模式, DRAM), 413-414
Fault intolerance, processor design and (容错, 处理器设计), 69
Faulting page, loading (页错误, 加载), 320
Faulting process. *See* Page fault (错误处理, 参见页错误)
FBUF buffer, pipeline register (FBUF 缓冲区, 流水线寄存器), 182-183
Feedback line, pipelined processor (反馈线路, 流水线处理器), 187
Fence register, memory management and (栅栏寄

存器，内存管理），281
Fetch and execute stages（取指令和执行阶段），175
FETCH macro state（FETCH 宏状态），91，99–102
Fiber distributed data interface（FDDI）（光纤分布式数据接口），678
Field programmable gate arrays（FPGAs）（现场可编程门阵列），120–121
Figures of merit（性能系数），476–477
File allocation table（FAT）（文件分配表），481–483
File systems（文件系统）
allocation strategy comparison（分配策略的比较），490
attributes of（属性），469–475
CD-ROM and CD-R（CD-ROM 和 CD-R），508–509
components of（组件），498–503
consistency check（一致性检查），508
contiguous allocation and（连续分配），477–480
creating and writing files（创建和写文件），499–500
device driver and（设备驱动程序），499
disk driver handling and（磁盘驱动器处理），501–502
figures of merit for（性能系数），476–477
file allocation table（FAT）（文件分配表），481–483
hybrid indexed allocation and（混合索引分配），486–491
indexed allocation and（索引分配），483–485
information flow and（信息流），501
journaling（日志），514–515
layout on the physical media（物理介质上的布局），503–507
linked allocation and（链式分配），480–481
Linux，509–515
media independent layer（媒介独立层），498–499
media specific requests scheduling layer（媒介请求调度层），499
media specific storage space allocation layer（媒介存储空间分配层），499
in memory data structures and（存储于内存中的数据结构），507
Microsoft Windows，515–517
multilevel indexed allocation and（多层索引分配），485–486
physical media and（物理媒介），508–509
subsystem interactions and（子系统交互），500–503
system crashes and（系统崩溃），508
File transfer protocol（FTP）（文件传输协议），706
Fine-grain critical sections（细粒度临界区），545–546
Fine-grained parallelism（细粒度并行性），601
Finite state machine（FSM）（有限状态机）
control unit as（控制单元），89–91
for CPU datapath（CPU 数据通路），98–99
with disabled interrupts added（添加了关闭中断指令），139
instruction execution and（指令执行），173
modifications to for handling interrupts（为处理中断而修改），136–137
sequential logic circuits and（时序逻辑电路），90–91
Firewire（火线），462
First-come first-served（FCFS）（先到先服务），247–252，453
First fit（首次适应），289
First in first out（FIFO）page replacement（先进先出页替换策略），327–329，333–335
First-level caches（一级缓存），357
Fixed-size partitions（固定分区），286–287
Flash memory（闪存），439，461
Flexible placement（灵活替换）
fully associative cache and（全相关高速缓存），385–387
reducing the miss penalty and（降低缺失损失），393–394
set associative cache（设置组相关缓存），387
set associativity extremes and（组相关的极端情况），387–392
Flip-flop memory element（触发器记忆元件），78
Floating-point instructions（浮点指令），67
Floating-point pipelines（浮点流水线），217
Flush control lines，pipelined processor（冲刷控制线路，流水线处理器），205
Flushing（冲刷），212，320
Fortran monitoring system（FMS）（FORTRAN 监控系统），13，273
Fortran programming language（Fortran 编程语言），16，273

Forward error correction (FEC)(前向纠错), 635, 648
Forwarding table (转发表), 668
Four-port switch (四端口交换机), 681
Four-way set associative cache organization (四路组相关缓存结构), 389
Fragmentation (片段), 286–287
Frame pointer (帧指针), 55–58
Frame relay (帧延迟), 685
Frame table (帧页表), 319
free BSD, UNIX operating system (free BSD, UNIX 操作系统), 274, 697
Free-list (空闲表), 318–320, 336–338, 478
Free page frames pool (空闲页帧池), 336–338
Frequency division multiplexing (FDM)(频分复用), 631, 669
Front end, microarchitecture (前端, 微体系结构), 223–234
Full-duplex modem (全双工调制解调器), 710
Full flag, FIFO page replacement (满标志, FIFO 页替换策略), 327
Fully associative cache (全相关高速缓存), 385–387
Function calls (函数调用)
  activation records and (活动记录), 54
  frame pointer and (帧指针), 55–58
  procedure calling chores (过程调用剩余的工作), 46–47
  recursion (递归), 54–55
  software convention and (软件惯例), 47–53
  state of the caller (调用者状态), 43–45

G

Gang scheduling (组调度), 591–592
Garbage collection (垃圾回收), 68
Gateway routers (网关路由器), 657
General Electric, 310
General purpose graphic processing unit (GPGPU) (通用图形处理器), 602
GENI (Global Environment for Network Innovation), 651
Geometric mean (GM) (几何平均数), 162–163, 170
gethostbyname, UNIX command, (gethostbyname, UNIX 命令) 727–728
Global descriptor table (GDT)(全局描述符表), 312
Global victim selection, page replacement and (全局被替换, 页替换策略), 326
Gnu software foundation (GNU 软件基金会), 274
Google Earth, 2, 3f
Grand challenge problems (大挑战问题), 226
Granularity, operand width (粒度, 操作数宽度), 27–30
Graphical user interface (GUI)(图形用户界面), 2, 13, 275
Graphics display data rates (图形显示数据传输速率), 439
Graphics processing unit (GPU) (图形处理单元), 68, 602
Grid computing (网格计算), 13–14, 264–266

H

Handler procedure, interrupts (处理过程, 中断), 133
Hansen, Brinch, 586
Hard link, file systems (硬链接, 文件系统), 471
Hardware-based speculation, pipelined processor (基于硬件的投机执行, 流水线处理器), 218
Hardware buffer, DMA controller (硬件缓存, DMA 控制器), 429
Hardware for handling discontinuities (处理程序不连续性的硬件), 143–149
Hardware multithreading (硬件多线程), 596–599
Hardware organization, in desktop computer system (硬件组织, 在桌面计算机系统中), 8–10, 9f
Hardware resource organization, pipelined processor (硬件资源组织, 流水线处理器), 180
Hardware/software interface (硬件/软件接口), 4, 4f
Hardwired control (硬连线控制), 119–121
Harmonic mean (HM) (调和平均数), 162–163, 170
Head assembly, disk (磁头组件, 磁盘), 440
Hennessy, John, 158
Hidden terminal problem (隐藏终端问题), 674–675
Hierarchical name in UNIX, file systems (UNIX 中

的层次化名称，文件系统），491
Hierarchical routing（分层路由），657-658
High-level data abstractions（高级数据抽象），35-37
High-level languages（高级语言），3，20-21，158
High-performance computational resources（高性能计算资源），265
High-performance computing（HPC）（高性能计算），266
High water mark，thrashing control（上限，颠簸控制），342
Hit，caches（命中，缓存），356
Hit rate，caches（命中率，缓存），356
Hoare，Tony，586
Hold and wait，deadlocks（持有并等待），383
Hold time，register clocking（保持时间，寄存器时钟周期），94
Holes，memory allocation（洞，内存分配），288
Horizontal microcode（水平微码），119
Host，networking terminologies（主机，网络术语），621
HTTP（hyper-text transfer protocol）（超文本传输协议），633
Hubs，networking hardware（集线器，网络硬件），679-680
Hybrid architecture style（混合体系结构风格），59
Hybrid indexed allocation，file systems（混合索引分配，文件系统），486-491
Hyperthreading（超线程），599

I

I-MEM（instruction memory）（指令内存），176
IBM，31，59，273，309-310，434，599，604
IBSYS（IBM 7094 operating system）（IBM 7094 操作系统），13
ID/RR stage，pipelining（ID/RR 阶段，流水线），176-180
IEEE 802.3，673-674
IEEE 802.11，674
IF stage，pipelining（IF 阶段，流水线），176-180
If-then-else statement（If-then-else 语句），38-40
Immediate values（立即值），23-24
Implementation，scheduler evaluation and（实现，调度器评估），266-267
Implicit parallelism（隐式并行），214
Imprecise exception（非精确异常），219
In memory data structures，file systems（内存的数据结构，文件系统），507
Index node（i-node）（索引节点），483，491-492，497-498，512
Indexed allocation，file systems and（索引分配，文件系统），483-485
Indirect access，ISA and（间接访问，ISA），358
Indirect addressing modes（间接寻址模式），58
Inkjet printer data rates（喷墨打印机数据传输速率），439
Input buffers，switches and（输入缓冲区，交换机），669
Input/output（I/O）（输入/输出）
  bound workload and（受限负载），266
  burst and（突发），242，243，260-261
  bus evolution and（总线演化），461-463
  completion interrupt handler and（I/O 完成中断处理程序），263
  CPU communication and（CPU 通信），423-427
  device controller and（设备控制器），424-425
  memory mapped（内存映射），425-427
  processors and（处理器），433-434
  programmed（程序），427-429
  queue and（队列），243，244
  request trap and（请求陷入），263
  super I/O（超级 I/O），465
Instant messaging（IM）（即时消息），631
Instruction and data caches（指令和数据缓存），392-393
Instruction formats（指令格式）
  all instructions same length，（所有指令，相同长度）61
  architectural choices and（体系结构选择），59-62
  instruction format variable lengths（指令长度可变），62
  one-operand instructions（单操作数指令），60
  three-operand instructions（三操作数指令），60
  two-operand instructions（双操作数指令），60
  zero-operand instructions（零操作数指令），59-60

Instruction frequency（指令频率），160–161
Instruction issue（指令发射），315
Instruction-level parallelism（ILP）（指令级并行），214，596，598–600
Instruction pipeline（指令流水线）
  buffering and（缓冲），178
  datapath elements for（数据通路元素），178–180
  fixing problems with（修正问题），176–178
  passage of instructions through（指令经过），177
Instruction-processing assembly line（指令处理的流水线），172–175
Instruction register（IR）（指令寄存器），79
Instruction restart，hardware for（指令重启，硬件），318
Instruction-set architecture（指令集架构）
  additional addressing modes and（其他寻址模式），58–59
  additional instructions and（其他指令），58
  architecture styles（架构风格），59
  datapath（数据通路），93–94
instruction formats（指令格式），59–62
Instruction set design（指令集设计）
  applications and（应用），18–19，67–68
  issues and（问题），66–67
  process of（过程），20–21
  programming languages and（编程语言），18–19
Instructions per clock cycle（IPC）（每时钟周期的指令数），171
INT macro state（INT 宏状态），136–137，147–149
Integrated drive electronics（IDE）（电子集成驱动器），450
Intel 8080/8085 single-chip processors（Intel 8080/8085 单芯片处理器），274
Intel Core microarchitecture（Intel Core 微架构），222–225
Intel Pentium，312–313
Intel x86，31，66
Inter-thread synchronization（线程间同步），575
Interactive environments（交互式环境），240
Interactive video games，ISA and（交互式视频游戏，ISA），68
Interconnection networks（互联网络），599–600
Interface message processor（IMP）（接口报文处理机），712
Interleaved memory（交错式内存），407–408
Internal fragmentation（内部碎片），286，305，478
Internal representation of condition variable data type（条件变量数据类型的内部表示），551–552
Internet
  addressing（寻址），658–663
  characteristics of（特点），621–622
  evolution of（演变），712–713
  transmission control protocol and（传输控制协议），648–649
  transport protocols and（传输协议），648–651
  user datagram protocol and（用户数据报协议），648–650
Internet protocol stack（Internet 协议栈）
  application layer（应用层），628
  IP address（IP 地址），559–563，625，687
  link layer（链路层），629–630
  network layer（网络层），629
  networks（网络），660–662
  physical layer（物理层），630–631
  transport layer（传输层），629
Internet service provider（ISP）（互联网服务提供商），622，625，662
Interrupt controller（中断控制器），145
Interrupt enable（中断允许位），424
Interrupt flag（中断标志位），424
Interrupt handling（中断处理），150–152
Interrupt mechanisms at work（工作中的中断机制），150–152
Interrupt mode（中断模式），154
Interrupt vector（中断向量），146–147
Interrupt vector table（IVT）（中断向量表），134，149
Interrupts（中断），131–132. *See also* Program discontinuities（参见程序不连续性）
Invalid pages，page fault handling and（无效页，页错误处理），320
Inverted page tables（反向页表），349
iPhone OS X，17
iPod，423
ISO-Transport Protocol（ISO-传输协议），651

Issue order, pipelining and (发射顺序，流水线), 217–218

J

JALR instruction (JALR 指令), 106–108
Java programming language (Java 编程语言), 586–589
Job control language (JCL) (作业控制语言), 240, 273
Journaling file systems (日志文件系统), 514–515
Jump table, switch statement implementation And (跳转表，switch 语句实现), 40–41

K

Kahn, Robert, 742
Kernel-level threads (内核级线程), 570–573
Kernel mode (内核模式), 148, 149
Kernel space (内核空间), 281–282
Kernighan, Brian, 17
Keyboard (键盘), 424–425, 439–440
Kilby, Jack, 16
Killer micros (杀手级微处理器), 123, 226, 614
Kleinrock, Leonard, 712
KSR-1, multiprocessor from Kendall Square Research (KSR 数处理器), 604, 618

L

L1 cache (L1 缓存), 416
L2 cache (L2 缓存), 416
Lamport, Leslie, 608
Larrabee, Intel processor (Larrabee, Intel 处理器), 602
Laser printer data rates (激光打印机数据传输速率), 439
Last-in-first-out (LIFO) property (后进先出特性), 44
Last write time, file systems attributes and (上次写入时间，文件系统属性), 471, 474
LC-2200
  control unit (控制单元), 118
  hazards in (冒险), 211
  instruction format (指令格式), 63–65
  J-type instructions (jair) (J 类型指令), 63
  I-type instructions (addi, lw, sw, beq) (I 类型指令), 63
  O-type instructions (halt) (O 类型指令), 65
  R-type instructions (add, nand) (R 类型指令), 63
  register convention and (寄存器惯例), 65
Least recently used (LRU) algorithm (最少使用算法), 329–333
  cache replacement and (缓存替换), 394–396
  push down stack for (下推栈), 330
  reference bit per page frame and (每个页帧引用位), 332–333
  small hardware stack and (小的硬件栈), 331–332
Least significant byte (LSB) (最低有效字节), 29, 30
Legacy software (遗留软件), 77
Level logic (电平逻辑), 80
Lightweight process (lwp) (轻量级进程), 573
Link layer, Internet (链路层，Internet), 629–630
Link-layer protocols (链路层协议)
  asynchronous mode transfer (异步模式传输), 678
  CSMA/CD, 671–673
  Ethernet and (以太网), 670–671
  fiber distributed data interface (FDDI) (光纤分布式数据接口), 678
  IEEE 802.3, 673–674
  IEEE 802.11, 674
  point to point protocol (PPP) (点对点协议), 678
  token ring (令牌环), 675–677
  wireless LAN (无线局域网), 674–675
Link state routing algorithm (链路状态路由协议), 653–655
Linked allocation, file systems and (链式分配，文件系统), 480–481
Linker (链接器), 235
Linux Ext3, 515
Linux file system (Linux 文件系统), 509–515
Linux scheduler (Linux 调度器), 270–273
Little endian (小端), 31–32
Livelocks (活锁), 546–548, 560
Load and store instructions (加载和存储指令), 24–25, 197–199
Load time, static relocation (加载时间，静态重定位), 283
Loader (加载器), 235, 241

Local area networks (LAN)(局域网)
birth of (产生), 713
evolution of (演变), 713–715
fast Ethernet and (快速以太网), 714–715
at home (在家里), 625
technology breakthroughs and (技术突破), 615
thicknet and (粗缆网络), 713
thinnet and (细缆网络), 713–714
Local descriptor table (LDT), Intel Pentium (局部描述符表, Intel Pentium), 312
Local victim selection, page replacement and (局部被替换选择, 页替换), 326
Locality principle (局部性原则), 339, 355
Lock algorithm with test-and-set instruction (使用 Test-and-Set 指令的 Lock 算法), 577–578
Log segments, journaling file systems and (日志段, 日志文件系统), 514
Logic gates (逻辑门), 14
Logical cylinder for a disk (磁盘逻辑柱面), 440–441, 451
Logical page, virtual memory and (逻辑页, 虚拟内存), 291
Long-term scheduler (长期调度程序), 241
Longitudinal recording, disk and (水平记录, 磁盘), 450
LOOK algorithm, disk scheduling and (LOOK 算法, 磁盘调度算法), 455–457
Loop-level parallelism (循环级别并行性), 613
Lovelace, Ada, 16
Low pin count (LPC) bus (LPC 总线), 65
Low processor utilization, thrashing and (低处理器利用率, 颠簸), 338
Low water mark, thrashing control (缺页下限, 颠簸控制), 342
Lower bound clock cycle time (时钟周期的下限), 94
LW (load word) instruction (LW 指令), 108–111

M

MAC Address, link layer and (MAC 地址, 链路层), 682–684
Mac computers (MAC 计算机), 275
Mac iPhone OS, 13
Mac OS X, 17
Macro-fusion, microarchitecture and (宏合并, 微架构), 224
Macro states, control unit and (宏状态, 控制单元), 96, 172
Magnetic core memory (磁芯存储器), 12, 13f
Magnetic disk schematic (磁盘), 441
Magnetic media (磁性介质), 449
Main memory design considerations (主存的设计因素), 405–408
Mainframe computers (大型机), 226, 273, 710
malloc, C library call (malloc, C 库调用), 574
Manchester code (曼彻斯特码), 673
Many-core architecture (多核架构), 69, 610–611
Mapping, virtual memory and (映射, 虚拟内存), 291
Maskable interrupts (可屏蔽中断), 153
Master boot record (主引导记录), 504
Master file table (MFT)(主文件表), 516–517
Master-slave relationship, bus action and (主–从关系, 总线操作), 433
Math libraries (数学库), 19
Mauchly, John, 15, 273
Maximum precision, arithmetic operations (最大精度, 数学操作), 28
MBUF buffer, pipeline register (MBUF 缓冲器, 流水线寄存器), 182–183
Media access control (MAC) (介质访问控制), 670, 682
Media independent layer (媒介独立层), 498–499
Media specific requests scheduling layer (媒介请求调度层), 499
Media specific storage space allocation layer (媒介存储空间分配层), 498–499
Medium-grained parallelism (中粒度并行性), 603
Medium-term scheduler (高级调度器), 241
MEM stage, pipelining (MEM 阶段, 流水线), 177–180
Memory (内存)
access time and (访问时间), 180
address in an instruction (在指令中寻址), 26–27
addressability problem and (可寻址性问题), 23

bandwidth and (带宽), 392
basic concepts and (基本概念), 81–82
bounds and (限制), 282–284
cache coherence and (缓存一致性), 607–610
costs and (开销), 158
density and (密度), 68
footprints and (内存印记), 32, 157, 170
in single threaded and multithreaded processes (单线程进程和多线程进程), 567
interleaving and (交错), 486
Interpretation for set associative cache (组相关缓存的翻译), 389
LC-2200 requirements and (LC-2200 需求), 92
memory address register (内存地址寄存器), 92
memory oriented architecture style (面向内存的体系结构), 59
memory wall (内存墙), 68
operand addressability and (操作数可寻址性), 28
operating system and (操作系统), 237
page tables in (页表), 291–295
pressure and (压力), 341
processor design and (处理器设计), 68–69
protection (保护), 279, 566
register loading from (载入寄存器), 25–26
relative sizes of (相对大小), 296–297
role of (作用), 79
space separation and (空间的分离), 268
stalls and (延迟), 369, 376–377
store instructions and (存储指令), 431
user program and (用户程序), 237
Memory allocation schemes (内存分配方案)
best fit and (最佳适应), 289
compaction and (紧缩), 289–290
first fit and (首次适应), 289
fixed-size partitions and (固定长度分区), 286–287
variable-size partitions and (可变长度分区), 287–289
Memory hierarchy (内存层次)
basic terminologies and (基本术语), 355–356
block size performance implications (块大小对性能的影响), 383–384
cache concept and (缓存的概念), 354–355
cache controller and (缓存控制器), 400
cache design considerations and (缓存设计的考虑), 404
cache misses in processor pipeline and (处理器流水线中的缓存缺失), 375–377
cache organization and (缓存组织), 360
cache read/write algorithms and (缓存读/写算法), 370–375
cache replacement policy and (缓存替换策略), 394–396
direct-mapped cache organization and (直接映射缓存的组织结构), 360–339
flexible placement and (灵活放置), 384–392
instruction and data caches (指令和数据缓存), 392–393
locality principle and (局部性原则), 355
main memory design considerations (主存的设计因素), 405–408
modern main memory system elements and (现代主存系统分析), 408–412
modern processor example (现代处理器实例), 418–419
multilevel memory hierarchy (多级存储体系), 351–360
multiprocessors and (多处理器), 580–582
page mode DRAM and (页模式 DRAM), 412–415
performance implications of (性能影响), 415–416
pipelined processor design and (流水线处理器设计), 369
relative sizes and latencies of (相对大小和延迟), 416
spatial locality to improve cache performance (利用空间局部性提高缓存性能), 377–384
virtually indexed physically tagged cache (虚拟索引物理标记的缓存), 401–402
Memory management (内存管理)
concurrent processes and (并发进程), 280
controlling thrashing and (控制颠簸), 311–312
dynamic relocation and (动态重定位), 284–285
functionalities provided by (提供的功能), 278–280
improved resource utilization and (提高资源利用率), 278

interactions with process scheduler（进程到度器的交互），324–325
optimization and（优化），336–342
overall goals of（总体目标），280
overlapping I/O with processing（处理过程中的I/O重叠），337
paged virtual memory（分页虚拟内存），290–297
paging vs. segmentation（分页和分段），303–308
pool of free page frames and（空闲页帧池），336–338
program locality and（程序的局部性），721–722
resource limitations and（资源限制），279
reverse mapping to page tables and（页表的反向映射），338
segmented virtual memory and（分段虚拟内存），297–303
simple schemes for（简单方案），280–285
static relocation and（静态重定位），282–284
thrashing and（颠簸），338–340
user and kernel separation and（用户和内核分隔），281–282
working set and（工作集），340–341
Memory mapped I/O（内存映射I/O），425–427
Mesh interconnection network（网络结构的互联网络），599
Message-passing interface（MPI）（消息传递接口），604
Message-passing multiprocessors（消息传递型多处理器），603–606
Message switching（报文交换），666，669
Message transmission time（报文传输时间），688–690
Meta-schedulers（元调度器），264–265
Metadata（元数据），367，373，469，685
Microchip development（微芯片的发展），12
Microcode ROM access（访问微指令ROM），216
Microprograms（微程序），119
Microsoft，14，274–275
Microsoft Windows，13，515–517
Microstates，control unit and（微状态，控制单元），96，159
Minicomputers（小型计算机），226
MINIX file system（MINIX文件系统），509
MINIX operating system（MINIX操作系统），17，274
MIPS technologies（MIPS技术），31，58
Miss penalty（缺失损失），356，377，393–394
Miss rate（缺失率），356，377
mkdir,UNIX command（mkdir,UNIX命令），475，494
MMX instructions（MMX指令），67
Modeling，scheduler evaluation and（建模，调度器评价），266–267
Modems（调制解调器），439，710–712
Modern language support，processor design And（现代语言支持，处理器设计），68
Modern memory systems（现代内存系统），408–415
Modified interrupt handler（修改后的中断处理过程），139
Monitor programming construct（管程编程结构），586
Moore，Gordon，1，16，16f
Moore's law（摩尔定律），613
Most significant byte（MSB)(最高有效字节），29，30
Mouse data rates（鼠标的数据传输速率），439–440
Motherboard（主板），77，357，463–464
MS-DOS，274，565
Multi-level page tables（多级页表），346–348
Multi-port repeater，networking hardware（多端口的中继器，网络硬件），679
Multicast（组播），684
Multicore architectures（多核架构），69，610–611
Multicore chips（多核芯片），123–124
Multicore processor design（多核处理器设计），221–222
Multics（MULTICS），274，310–313
Multilevel directory，file systems and（多级目录，文件系统），470
Multilevel indexed allocation，file systems and（多层索引分配，文件系统），485–486
Multilevel memory hierarchy（多级存储体系），357–360
Multiple access，Ethernet and（多路访问，Ethernet），672
Multiple decode units（多个解码单元），215
Multiple functional units，microarchitecture and（多个功能单元，微架构），215，216–217
Multiple instructions multiple data（MIMD)(多指令多数据流），603

Multiple instructions single data (MISD) (多指令单数据流), 602–603
Multiple issue processors (多发射处理器), 124, 215–216
Multiplexing (多路选择器), 103, 434
Multiprocessors (多处理器), 578–582. *See also* Parallel architecture (参见并行体系结构)
  cache coherence and (缓存一致性), 580–581, 604
  characteristics of (特点), 222
  ensuring atomicity and (保证原子性), 582
  gang scheduling and (组调度), 591–592
  memory consistency and (内存一致性), 607–610
  memory hierarchy and (内存层次), 580–582
  message passing versus (消息传递), 603;606
  noncache coherent (非缓存一致), 604
  page tables and (页表), 579
  scheduling and (调度), 589–592
  space sharing and (空间共享), 589–592
Multiprogrammed scheduling environments (多程序调度环境), 239–240
Multithreading (多线程)
  advanced synchronization algorithms and (高级同步算法), 586–589
  atomic test-and-set instruction and (原子的 Test-and-Set 指令), 575–577
  atomicity for group of instructions (一组指令的原子性需求), 543–544
  basic code with no synchronization (无同步的基本代码), 541–543
  code refinement with coarse-grain critical sections (使用粗粒度临界区的代码改进), 544
  code refinement with fine-grain critical sections (使用细粒度临界区的代码改进), 545–546
  communication among threads (线程间通信), 526–527, 574–575
  concurrency problems and (并发问题), 592–596
  condition variables and (条件变量), 548–552
  deadlocks and livelocks and (死锁和活锁), 546–548, 583–586
  hardware multithreading (硬件多线程), 596–599
  hardware support for in uniprocessor (在单处理机上进行多线程的硬件支持), 574–575
  important points in programming threads (线程编程的一些注意事项), 561
  inter-thread synchronization and (线程间同步), 575
  internal representation of data types (数据类型的内部表示), 540–541
  kernel-level threads (内核级线程), 570–573
  light-weight process (lwp)(轻量级进程), 573
  lock algorithm with test-and-set instruction (使用 Test-and-Set 指令的 Lock 算法), 577–578
  multiprocessors and (多处理器), 578–582, 589–592
  nondeterminism and (不确定性), 528–533
  OS support for threads(操作系统对线程的支持), 565–574
  overlapping computation with I/O using (使用线程, 在 I/O 时进行计算), 522–523
  POSIX pthreads library and (POSIX pthread 库), 562–565
  race condition and (竞争条件), 528–533
  read-write conflict and (读写冲突), 528–533
  reasons for (原因), 522–523
  simple programming examples (简单的编程示例), 541–546
  Solaris threads and (Solaris 线程), 572–573
  thread creation and termination (线程创建与终止), 525–525, 574–575
  thread function call summary (线程函数调用总结), 559–561
  thread synchronization and (线程同步), 533–540
  threads library safety (线程库安全性), 573–574
  user level threads (用户级线程), 567–570, 573
  using threads as software structuring abstraction (使用线程作为软件结构抽象), 561–562
  video processing example and (视频处理示例), 553–559
Multiword block size (多个字缓存块大小), 379–381
Multiword direct-mapped organization (多字直接映射缓存组织结构), 380
MULTICS (multiplexed information and computing service (多路信息和计算机服务), 274

mutex，mutual exclusion（互斥），534–536，544–545，547–548，553，558–560，574，594，617
Mutual exclusion（互斥），533–537，560，583

N

Name equivalence，file systems and（在名称上等价，文件系统），472
Name resolver，file systems and（名称解析器，文件系统），499
NAND instruction（NAND 指令），106
Negative-edge-triggered logic（负边沿触发逻辑），80
Nesting，procedure calls and（嵌套，过程调用），44
Network file system（NFS）（网络文件系统），706–707
Network interface card（NIC）（网卡），621，682–683，697–698
Network layer（网络层），652–670
  Internet addressing and（因特网寻址），658–663
  network service model（网络服务模型），663–668
  routing algorithms and（路由算法），652–658
  routing vs. forwarding（路由与转发），668
  service model and（服务模型），652
Network service model（网络服务模型）
  circuit switching and（电路交换），663–664
  message switching and（报文交换），666
  packet switching and（分组交换），664–668
Networking（网络），参见 specific layers
  application layer and（应用层），632–634
  applications and their transport protocols（网络应用程序和传输协议），651
  basic terminologies and（基本术语），621–626
  data rates and（数据传输速率），439
  data structures for packet transmission（用于数据包传输的数据结构），685–686
  device drivers（网络设备驱动程序），697–699
  higher-level protocols and（高层协议），706–708
  Internet evolution and（因特网演变），712–713
  key networking terminologies（网络关键术语），669
  link layer and local area networks（链路层和局域网），670–678
  message transmission time and（消息传输时间），688–694
  network buffers（网络缓冲区），698
  network congestion（网络拥塞），625，643–644
  network layer and（网络层），652–670
  network protocol（网络协议），626
  network services（网络服务），706–708
  networked video games（联网视频游戏），5–8，6f
  networking gear summary（网络组件概要），684
  networking hardware（网络硬件），678–683
  networking software（网络软件），626–628
  1GBase-T and 10GBase-T and（1GBase-T 和 10GBase-T），715
  PC and arrival of the LAN（个人计算机与局域网的出现），713
  programming using UNIX sockets（使用 UNIX 套接字进行网络编程），699–706
  protocol stack and（协议栈），628–632
  protocol stack layer relationships（协议栈各层之间的关系），683–685
  sample network（示例网络），653
  software and operating system（软件和操作系统），695–699
  TCP/IP header（TCP/IP 包头），687
  from telephony to（从电话到计算机网络），709–712
  transport layer and（传输层），634–651
Networking hardware（网络硬件）
  bridges and switches（网桥和交换机），679–680
  hubs（集线器），679
  network interface card（NIC）（网卡），682–683
  repeater（中继器），679
  routers（路由器），683
  virtual local area networks（虚拟局域网），682
Networking software（网络软件）
nework device driver（网络设备驱动程序），697–699
protocol stack implementation and（协议栈实现），697
socket library（socket 库），695–697
Next-state field，control unit and（下一状态字段，控制单元），116，117
No-write allocate，write-miss handling（非写分配，写缺失处理），373
Noise burst，Ethernet and（噪声脉冲，Ethernet），672
Noncache coherent（NCC）multiprocessors（非缓存一致多处理器），604

Nondeterminism, multithreading and (不确定性, 多线程), 528–533
Nondeterministic execution, 531, 560 (非确定性执行)
Nonmaskable interrupts (不可屏蔽中断), 153
Nonpreemptive scheduling algorithms (非抢先调度算法), 245–256
  first-come first-serve (先到先服务), 247–252
  priority and (优先级), 255–256
  shortest job first (最短作业优先), 252–255
Nonrelocatable processes (不可重定位进程), 283
Nonzoned recording, magnetic disk and (非分区记录, 磁盘), 441–442
NOP (no-operation) instructions (NOP 指令), 186–187, 203–204
Not retired instructions, microarchitecture and (非退役指令, 微架构), 218
Noyce, Robert, 16
NT file system (NTFS) (NT 文件系统), 516–517
Nvidia, GPU vendor (Nvidia, GPU 供应商), 602–603

O

Object reference (对象引用), 517
Occupied bit (被占位), 286
Offset, page (偏移量, 页), 293
One-operand instructions (单操作数指令), 60
Open source operating systems (开源操作系统), 270
Open systems interconnection (OSI) suite (开放系统互连套件), 631–632
Operands (操作数)
  addressability of (可寻址性), 22–23
  addressing mode and (寻址模式), 24
  immediate values and (立即值), 23–24
  load and store instructions and (加载和存储指令), 24–25
  packing of (打包), 32–34
  register addressing and (寄存器寻址), 24
  using registers and (使用寄存器), 22–23
  width of (宽度), 27–30
Operating systems (OS) (操作系统)
  advanced synchronization algorithms and (高级同步算法), 586–589
  deadlocks and (死锁), 583–584
  evolution of (演化), 13–14
  instruction set design and (指令集设计), 19
  memory protection in (内存保护), 566
  networking software and (网络软件), 695–699
  processor design and (处理器设计), 68
  protocol stack implementation and (协议栈实现), 697
  thread support and (线程支持), 13–14
  video games and (视频游戏), 5–8
Optimization, memory management and (优化, 内存管理), 336–342
Out-of-order delivery of packets (数据包的乱序到达), 627
Out-of-order execution (乱序执行), 217–218, 224
Out-of-order processing (乱序处理), 218–219
Output buffers (输出缓冲区), 669
Overflow region, file systems and (溢出区域, 文件系统), 480
Overlapping I/O with processing (处理过程中的 I/O 重叠), 337
Owners, file system attributes and (拥有, 文件系统属性), 471, 474

P

Packet arrival from network (网络数据包到达), 699
Packet header (数据包头), 685
Packet loss (数据包丢失), 625, 627, 643, 665, 669
Packet queues (数据包排队), 643
Packet-switched networks (分组交换网络), 667–668
Packet switching (分组交换), 664–666, 669
Packet transmission, data structures for (数据包传输, 数据结构), 685–686
Packing of operands (操作数打包), 32–34
Padding, data structures and (填充, 数据结构), 510–511
Page based memory management (基于页的内存管理)
  access rights as part of page table entry (局部页表项的访问权限), 348–349
  demand paging and (按需分页), 316–324
  inverted page tables and (反向页表), 349
  multi-level page tables and (多级页表), 346–348

optimizing memory management (优化内存管理), 336–342
page replacement policies (页替换策略), 326–336
process scheduler and memory manager (进程调度器和内存管理器), 324–325
translation lookaside buffer and (旁路转换缓存), 343–346
Page coloring (页面着色), 403
Page fault (页错误)
anatomy of (剖析), 320–324
exceptions and (异常), 317
faulting page loading (加载缺失页), 320
finding free page frame and (发现一个空闲页框), 320
handler (处理程序), 318, 320–324
page table update and (更新页表), 321
picking victim page and (选择被替换页), 320
rate reduction and (降低页错误率), 326
restarting faulting process and (重启缺页异常进程), 321
thrashing control and (颠簸控制), 341–342
Page frames, virtual memory and (页帧, 虚拟内存), 290
Page mode DRAM (页式 DRAM), 412–415
Page replacement policies (页替换策略)
algorithm comparison and (算法比较), 337
Belady's Min and, 326–327
first in first out and (先进先出), 327–329
global victim selection and (全局被替换选择), 326
least recently used and (最近最少使用策略), 329–333
local victim selection and (局部被替换选择), 326
random replacement and (随机替换), 327
second chance page replacement algorithm (第二次机会页替换算法), 333–336
Page table base register (PTBR), virtual memory and (页表基址寄存器, 虚拟内存), 294–296, 351
Page table entry (PTE) (页表项), 295
Page tables (页表), 291–295
access rights and (访问权限), 348–349
demand paging and (按需分页), 317
faulting process and (缺页异常进程), 321
inverted (反向), 349
multi-level (多级), 346–348
multiprocessors and (多处理器), 579
Paged memory system, address computations in (分页内存系统, 地址计算), 307
Paged segmentation (页式分段), 306
Paged virtual memory (分页虚拟内存)
hardware for paging (支持分页的硬件), 295–296
page tables and (页表), 291–296
relative sizes and (相对大小), 296–297
Paging daemon (分页守护进程), 333
Paging hardware (支持分页的硬件), 295–296
Paging vs. segmentation (分页与分段), 303–308
Pan-tilt-zoom (PTZ) camera (云台变焦摄像机), 436–438
Parallel architecture (并行体系结构). *See also* Multiprocessors (参见多处理器)
multicore and many-core (多核与众核), 610–611
multiple instructions multiple data (MIMD) (多指令多数据流), 603
multiple instructions single data (MISD) (多指令单数据流), 602–603
single instruction multiple data (SIMD) (单指令多数据流), 601–602
single instruction single data (SISD) (单指令单数据流), 600–601
taxonomy, 600–601
Parallel programs, execution models of (并行程序, 执行模型), 533
Parallel tag matching hardware (并行标记匹配硬件), 386
Parallelism, processor design and (并行性, 处理器设计), 69
Parameter passing, procedure call and (参数传递, 过程调用), 46
Partitions (分区)
fixed size (固定大小), 286–287
layout and (布局), 506
table data structure and (表数据结构), 505–506
variable size (变量大小), 287–288
Payload (有效载荷), 683

PDP-11 (Digital Equipment Corp.), 59
Pentium processors (Pentium 处理器), 67, 312–313
Per-file information (每个文件信息), 507
Performance metrics (性能度量), 170, 245–247, 248
Peripheral component interchange (PCI) (外围组件互连), 432
Peripheral devices (外围设备), 438–440
Perpendicular magnetic recording (PMR) (垂直磁记录), magnetic disk (磁盘), 450
Personal computer (PC) (个人计算机), 274, 713
Personal computer hard drive (circa 2008) (电脑的硬盘驱动 (2008 年左右)), 449
Personal computer modem (个人电脑调制解调器), 710–712
Personal digital assistants (PDAs) (个人数字助理), 366
Pervasive computing domain (普适计算领域), 266
Physical address extension (PAE) feature (物理地址扩展特性), 297
Physical frame number (PFN) (物理帧号), 293–296, 402–403
Physical frames, virtual memory and (物理帧, 虚拟内存), 290–291
Physical layer, Internet (物理层, Internet), 630–631
Physical media, file system layout and (物理介质, 文件系统布局), 503–507
Physical memory (物理内存). *See* Memory (参见内存)
Pin a page, memory management and (锁住物理内存中的页面, 内存管理), 343
Pipeline-conscious architecture and Implementation (针对流水线的体系结构与实现)
  anatomy of instruction passage (指令穿过流水线的过程详解), 181–183
  easy-decodable instruction format and (容易解码的指令格式), 180–181
  equal amounts of work in each stage and (确保每个阶段的工作量相同), 181
  pipeline register design and (流水线寄存器的设计), 184
  pipeline stage implementation and (流水线各个阶段的实现), 185
Pipeline depth, microarchitecture and (流水线深度, 微架构), 215–218
Pipeline hazards (流水线冒险)
  clocks per instruction and (每指令的时钟周期数), 186
  control hazards (控制冒险), 200–209
  data hazards (数据冒险), 187–200
  pipeline efficiency and (流水线的效率), 186
  structural hazards (结构化冒险), 186–187
Pipeline register (buffers) (流水线寄存器 (缓冲区))
  contents and (内容), 183
  design and (设计), 184
  generic layout of (通用布局), 185
  with unique names (专有名称), 182
Pipelined processor (流水线处理器)
  design and (设计), 369
  program discontinuities in (程序不连续性), 211–214
Pipelined protocols (流水式协议), 640–642
  bandwidth and (带宽), 640
  packet transmission with no ACKs (无确认的包传输), 540
  propagation time and (传播时间), 640
Pipelining (流水线), 171–172
PlanetLab, 651
Platter, disk and (盘片, 磁盘), 440–444, 448, 479, 484–485, 504–505, 519
Plug and play (即插即用), 462
Point of call, procedure call and (调用点, 过程调用), 47
Point to point protocol (PPP) (点对点协议), 678
Pool of free page frames (空闲页帧池), 336–338
Positive acknowledgment, packet-level (肯定确认, 数据包级), 635–636
Positive-edge-triggered logic (负边沿触发逻辑), 80
POSIX pthreads library (POSIX pthreads 库), 562–565
Power consumption (功耗), 77, 221–222
Power PC, processor (Power PC, 处理器), 19, 123
Precision of operands (操作数的精度), 27–30
Preemptive scheduling algorithms (抢占式调度算法), 242–243, 256–264
Prepaging, memory management and (预分页, 内存管理), 343
Price-performance tradeoff, processor implementation And (性价比权衡, 处理器实现), 77

Principle of locality（局部性原则），339，355
Priority，processor scheduling and（优先级，处理器调度），255–255
Priority-interrupt schemes（优先级中断机制），144–145
Privileged instructions（特权指令），148，281
Privileged mode（特权模式），148
Privileges，file system attributes and（权限，文件系统属性），471，474
Procedure calls（过程调用）
compiling（编译），42–43
high-level language and（高级语言），21
parameter passing and（参数传递），46
return address and（返回地址），46
return to point of call and（返回调用点），47
return values and（返回值），47
shadow registers and（影子寄存器），43–44
software convention and（软件惯例），45
space for callee's local variables and（被调用者局部变量的空间），47
transferring control to callee（将控制权移交给被调用者），46
Process concept（进程的概念），720–721
Process control block（PCB）（进程控制块），243–244，566
Process independence，memory manager and（进程独立性，内存管理器），279
Process scheduling（进程调度器），242–245，324–325
Process termination trap handler（进程终止陷入处理器），263
Processes，characteristics of（进程，特点），237–239
Processing delay（处理延迟）
at the receiver（接收端的处理延迟），690
at the sender（发送端的处理延迟），688–689
Processor architecture and design（处理器体系结构与设计）
applications and instruction set design（应用程序与指令集设计），67–68
common high-level language feature set（常见的高级语言功能集），21
compiling function calls（编译函数调用），42–58
conditional statements and loops（条件语句和循环），37–42
deeper pipelines and（更深的流水线），215–218
design issues and（设计问题），68–69
expressions and assignment statements and（表达式和赋值语句），21–34
high-level data abstractions（高级数据抽象），35–37
impact of scheduling on（调度对处理器体系结构的影响），267–268
instruction-level parallelism and（指令集并行性），214
instruction-set architectural choices（指令集体系结构选择），58–62
instruction set design and（指令集设计），20–21
instruction set issues（指令集问题），66–67
Intel Core microarchitecture（Intel Core 微架构），222–225
issues influencing（影响处理器设计的问题），66–69
LC-2200 instruction set and（LC-2200 指令集），62–65
managing shared resources and（管理共享资源），219–221
multicore processor design（多核处理器设计），221–222
out-of-order processing and（乱序执行），218–219
power consumption and（功耗），228–222
program discontinuities and（程序的不连续性），218–219
Processor implementation（处理器实现）
alternate control unit design style（控制单元设计的另一种选择），119–121
architecture vs. implementation（体系结构与实现），76–77
bus-based design and（基于总线的设计），86–89
circuits and（电路），78
clock pulse width and（时钟脉冲宽度），94
comparison of control regimes（控制风格比较），121
control unit design and（控制单元设计），95–96
datapath design and（数据通路设计），91–93
datapath element connections and（连接数据通路

的元件), 82–86
DECODE macro state (DECODE 宏状态), 102–103, 116–119
edge-triggered logic and (边沿触发逻辑), 79–82
engineering a conditional branch in microprogram (设计微程序中的条件分支), 116
EXECUTE macro state and (EXECUTE 宏状态), 103–116
factors involved in (处理器实现涉及的因素), 77–78
FETCH macro state and (FETCH 宏状态), 99–102
finite state machine and (有限状态机), 89–91
hardware resources of the datapath (数据通路的硬件资源), 79
hardwired control and (硬连线控制), 119–121
historical perspective and (历史回顾), 122–124
ISA and datapath width (ISA 和数据通路宽度), 93–94
key hardware concepts (重要的硬件概念), 78–91
microprogrammed control and (微程序控制), 119
ROM plus state register and (ROM 加状态寄存器), 96–99
Processor performance (处理器性能)
benchmarks and (基准测试程序), 161–165
datapath elements for instruction pipeline (指令流水线的数据通路元素), 178–180
datapath organization leading to lower CPI (改进数据通路组织以减小 CPI), 166
decreasing clock cycle time and (减少时钟周期时间), 165
executed instruction reduction and (减少执行的指令条数), 166
hazards and (冒险), 185–211
instruction frequency and (指令频率), 160–161
instruction pipeline problems and (指令流水线问题), 176–178
instruction-processing assembly line and (近间处理指令的流水线), 172–175
pipeline-conscious architecture and implementation (有利流水线的体系结构与实现), 180–185
pipelining and (流水线), 171–172
simple-minded instruction pipeline and (简单指令流水线), 175–176
space and time metrics and (时间和空间性能指标), 156–160
speedup and (加速比), 167–171
throughput increase and (提升处理器的吞吐量), 171
Processor scheduling (处理器调度)
combining priority and preemption (结合优先级和独占), 264
evaluation and (评价), 265–267
historical perspective and (历史回顾), 273–275
meta-schedulers (元调度器), 264–265
nonpreemptive scheduling algorithms (非抢占式调度算法), 247–256
performance metrics and (性能度量), 245–247
preemptive scheduling algorithms (抢占式调度算法), 256–264
processor architecture impact and (调度对处理器体系结构的影响), 267–268
programs and processes and (程序和进程), 235–349
scheduling algorithm comparison (调度算法的比较), 269
scheduling environments and (调度环境), 239–241
Processors (处理器)
block multiplexer channel (组卷多路复用信道), 434
cache and (缓存), 345–346, 375–377
demand paging and (按需分页), 317
input/output and (输入/输出), 433–434
multiplexer channel and (多路复用信道), 434
processor status word (PSW) (处理器状态字), 163
receiving interrupt vector and (接收中断向量), 146–147
selector channel and (选择器信道), 434
sharing and (共享), 260
speed and (速度), 77
stream-oriented devices (面向流的设备), 434
Producer-consumer, classic concurrency problem (生产者–消费者, 经典并发性问题), 592

Producer-consumer problem(生产者-消费者问题), 592–593
Program counter (PC)(程序计数器), 37, 46, 79
Program counter relative addressing (程序计数器相对寻址), 39
Program discontinuities (程序不连续性)
architectural enhancements and (体系结构改进), 135–143
complete interrupt handler and (完整的中断处理过程), 142
datapath enhancements for interrupts and (针对中断处理的数据通路改进), 143–146
dealing with (处理), 132–135
handling cascaded interrupts (处理级联中断), 138–141
hardware details for handling (处理程序不连续性的硬件细节), 143–149
interrupt mechanism at work (工作中的中断机制), 150–152
modifications to FSM and (修改 FSM), 136–137
modified interrupt handler and (修改后的中断处理过程), 140
processor design and (处理器设计), 218–219
receiving address of handler and (获得处理过程地址), 146–147
returning from the handler and (从处理过程中返回), 141–142
simple interrupt handler and (简单的中断处理过程), 137–138
stack for saving/restoring and (保存/恢复时的栈), 147–149
types of (类型), 130–132
Program life cycle (程序生命周期), 235–236
Program memory footprint (内存印迹), 235
Program order (程序顺序), 214, 224, 530, 560
Programmable interrupt controller (PIC) (可编程中断控制器), 163
Programmable logic arrays (PLAs) (可编程逻辑阵列), 120–121
Programmable read-only memory (PROM) (可编程的只读存储器), 461
Programmed data transfer (程序数据传输), 428–429
Programmed I/O (PIO)(程序 I/O), 427–429
Programming languages, instruction set design and (编程语言, 指令集设计), 18–19
Propagation delay (传播延迟), 94, 689
Propagation time (传播时间), 640
Protocol stack (协议栈)
implementation and (实现), 697
Internet protocol stack (因特网协议栈), 628–631
layer relationships and (各层之间的关系), 683–685
layering issues and (分层的实际问题), 632
OSI model (OSI 模型), 631–632
Pseudo-direct addressing, instruction-set design and (伪直接寻址, 指令集设计), 58–59
Pseudo header, TCP/IP and (伪包头, TCP/IP), 687
Punched cards (打孔卡片), 273
Push/pop, stack operations (压栈/弹出, 栈操作), 47
Push down stack, LRU page replacement and (LRU 替换策略的下推线), 330

Q

Qualitative metrics, processor scheduling and (定性指标, 进程调度), 246
Queuing (排队)
definition of (定义), 669
delays and (延迟), 625–627, 643, 665, 690
disk request queue (磁盘请求队列), 451
instruction queue (指令队列), 224
packet queues (数据包队列), 643
ready queues (就绪队列), 243–244, 261–262
timestamping and (时间戳), 327

R

Race conditions, multithreading and (竞争条件, 多线程), 528–533
Random access (随机访问), 476, 670
Random page replacement (随机页替换), 327
Read access to cache (对缓存的读访问), 370
Read after write (RAW) data hazards (写后读数据冒险), 187, 190–199
bubbles created by (写后读冒险产生的气泡), 196, 199

data forwarding and (数据前递), 192–197
load instructions and (读取内存的指令), 197–199
shared resources and (分享资源), 220
Read-modify-write operation (读取–修改–写入操作), 582
Read-only memory (ROM) (只读存储器), 97–98, 116–119
Read stall, pipelining and caches (读存延迟, 流水线和缓存), 369
Read-write conflict, multithreading and (读写冲突, 多线程), 528–532
Read/write heads, disk and (读写磁头, 磁盘), 440
Readers-writers, classic concurrency problems (读者–写者, 经典并发性问题), 593–594, 617
Ready bit, device controller and (就绪位, 设备控制器), 424
Ready queue (就绪队列), 243–244, 261–262
Real-time systems (实时系统), 269
Recording density, disk and (记录密度, 磁盘), 440
Recursion (递归), 54–55
Reduced instruction set computers (RISC) (精简指令集计算机), 66, 158
Reference bit per page frame (每个页帧引用位), 332–333
Reference count fields, i-node and (引用计数的字段, i-node), 491–493
Reference counters, approximate LRU and (引用计数器, 近似 LRU), 332–333
Register addressing, instruction-set design and (寄存器寻址, 指令集设计), 24
Register contents, pipelining and (寄存器内容, 流水线), 176
Register file (寄存器堆), 79, 83, 194–195
Register oriented architecture style (面向寄存器的体系结构), 59
Register renaming, microarchitecture and (寄存器重命名, 微体系结构), 218
Register windows (寄存器窗口), 44, 268
Registers, CPU and (寄存器, CPU), 22–23
Reliable pipelined protocol (可靠的流水式协议)
with ACKs (可靠的带 ACK 的流水式传输), 642
network congestion and (网络拥塞), 643–644
sliding window and (滑动窗口), 644–647
Remote procedure call (RPC) (远程过程调用), 706–707
Rendezvous, 538–540, 560
Reorder buffer, microarchitecture and (重排缓存, 微结构), 217–218, 220
Repeater, networking hardware (中继器, 网络硬件), 679
Replacement policy, cache (替换策略, 缓存), 394–396
Request to send (RTS) (请求发送), 674
Reservation station, microarchitecture and (保留站, 微结构), 225
Resource deadlock (资源死锁), 583
Resource limitations, memory manager and (资源限制, 内存管理器), 279
Resource utilization, memory manager and (资源利用, 内存管理器), 278
Response time (响应时间), 245, 452
RETI instruction (RETI 指令), 148–149
Retransmissions, transport layer (重传, 传输层), 647–648, 687
Return address, procedure call and (返回地址, 过程调用), 46
Return values, procedure call and (返回值, 过程调用), 47
Reverse mapping to page tables (页表的反向映射), 338
Ring interconnection network (环状的互联网络), 600
Ritchie, Dennis, 17, 274
rmdir, UNIX command (rmdir, UNIX 命令), 475
Root directory, file systems and (根目录, 文件系统), 507
Rotational latency, disk and (旋转延迟, 磁盘), 444
Round robin scheduler (轮转调度器)
algorithm for (算法), 263
details of (细节), 263–264
different system layers and (系统的不同层面), 262–263
processor sharing and (处理器分享), 260
putting to work (让轮转调度器工作), 260–261

time-shared environments and（分时环境），259
Round trip time（RTT）（行程时间），636
Route daemon（routed），network layer（路由守护进程，网络层），668
Routers，networking hardware（路由器，网络硬件），683
Routing algorithms（路由算法）
Dijkstra's link state routing algorithm（Dijkstra 链路状态路由算法），653–655
distance vector algorithm（距离矢量算法），656–657
hierarchical routing（分层路由），657–658
Routing tables（路由表），668–669，683
Row access strobe（RAS），DRAM and（行访问选通信号，DRAM），409–410

S

SCAN（elevator algorithm），disk scheduling and（电梯算法，磁盘调度），453–455
scanf，C library call（scanf，C 库调用），235
Scatter/gather functionality（分散/收集功能），634
Scheduler（调度器），237–238，241
Scheduling（调度）
algorithms（算法），247
basics of（基础），242–245
environments and（环境），239–241
multiprocessors and（多处理器），589–592
terminologies and（术语），245
Scoreboard，CDC 6600 and（记分板，CDC 6600），249
Second chance page replacement algorithm（第二次机会页替换算法），333–336
Second-level caches（二级缓存），357
Sector，disk and（扇区，磁盘），441，443–448，450，466–467，476–477，479，484–485，504–505
Security，processor design and（安全性，处理器设计），69
Seek time，disk and（寻道时间，磁盘），444
Segmentation（分段），299–301
address translation with（地址转换），304
characteristics and（特性），299
hardware and，303
versus paging（与分页），303–308
segment table base register（STBR）（段表基址寄存器），303
segment tables（段表），300
segmented address（分段地址），299
segmented memory systems（分段内存系统），307
segmented virtual memory（分段虚拟内存），297–303
Selector channel（选择器信道），434
Semaphore signaling system（信号量信号系统），576
Semiconductor memory（半导体内存），12
Sequence number（序列号），634–635，687
Sequential access，file systems and（顺序访问，文件系统），476
Sequential consistency（SC）（顺序一致性），608–610
Sequential logic，circuits and（时序逻辑），78
Sequential programming model（串行编程模型），214
Sequential programs，execution models of（串行编程，执行模型），533
Serial advanced technology attachment（SATA）（串行高级技术附件），450
Servers（服务器），226，266，562
Service model（服务模型），652，669
Set associative cache（组相关缓存），387
Set associativity extremes（组相关缓存的极端情况），387–392
Set up time，register clocking（建立时间，寄存器计时），91
SGI Altix，601，604
Shadow register set（影子寄存器组），43
Shared address space multiprocessors（共享地址空间多处理器），603–606
Shared resources，managing（共享资源，管理），219–221
Short-term scheduler（低级调度器），241
Shortest job first（SJF），processor scheduling and（最短作业优先，处理器调度），252–255
Shortest-path algorithm（最短路径算法），655
Shortest remaining time first（SRTF），processor scheduling and（最短剩余时间优先，处理器调度），256
Shortest seek time first（SSTF），disk scheduling and（最短寻道时间有限，磁盘调度），453

Sign-extend hardware（符号扩展硬件），108
Signals，multithreading and（信号，多线程），548
Simple interrupt handler（简单中断处理器），137–138
Simple main memory（简单的主存），405
Simple-minded instruction pipeline（简单指令流水线），175–176
Simulation，scheduler evaluation and（模拟，调度器发展），266–267
Simultaneous multithreading（同步多线程），599
Single bus design（单总线设计），86–87
Single-chip microprocessors（单芯片微处理器），123，614
Single instruction，loading processor registers and（单指令，加载处理器寄存器），268
Single instruction multiple data（SIMD）（单指令多数据），601–602
Single instruction single data（SISD）（单指令单数据），600–601
Sliding window（滑动窗口），644–647
Small computer systems interface（SCSI）（小型计算机系统接口），450
SMTP（simple mail transfer protocol）（使用简单邮件传输协议），633
Snoopy caches（侦听缓存），580–581
Socket library（Socket 库），634，695–697
Soft real-time applications（软实时应用），270
Software conventions（软件管理），45
Software foundation（软件基金会），274
Software interrupts（软件中断），131，502
Software structuring abstractions（软件结构抽象），561–562
Solaris threads（Solaris 线程），572–573
Solid logic technology（SLT）（固态逻辑技术），122
Solid state drives（SSD）（固态硬盘），459–461，508–509
Space，microprogrammed control and（空间，微程序控制），119
Space and time metrics（空间和时间度量），156–160
Space overhead（空间开销），367，469
Space sharing scheduler（空间共享调度器），590–591
Spatial locality（空间局部性），355，377–384
Speedup，processor（加速，处理器），167–171
Split transaction buses（分离传输总线），433
Stack for saving/restoring（保存/恢复时的栈），147–149
Stack oriented architecture style（面向栈的体系结构），59
Stack pointer（栈指针），44，55，147–148
Stack switching（栈切换），147–148
Stalled pipeline（被拖延的指令），187，210
Standard Performance Evaluation Corporation（SPEC），163
Starvation（饥饿），247，585–586
State of the caller，procedure call and（调用者状态，过程调用），43–45
State register，control unit and（状态寄存器，控制单元），96–99
Static address translation（静态地址转换），310
Static instruction frequency（指令频率），160–161，170
Static properties，processor scheduling and（静态特性，处理器调度），238
Static random access memory（SRAM）（静态随机访问存储），354
Static relocation，memory management and（静态重定位，内存管理），282–284
Status register，device controller and（状态寄存器，设备控制器），424
Stop-and-wait protocols（停止并等待协议），637–639
Storage（存储）. *See* Disk storage（参见磁盘存储）
Store and forward（存储与转发），664，669
Stored program computer（存储程序计算机），15，122
Stream-oriented devices（面向流的设备），434
Streaming data（流数据），67
Streaming devices（流式设备），429
Structural pipeline hazards（结构性流水线冒险），175，186–188
Structured data types（结构化数据类型），35
Subsystem interactions，filing systems and（子系统交互，文件系统），500–503
Successful lookup，caches and（成功查找，缓存），356
Sun SPARC，31，44

Super input/output (Super I/O)(超级 I/O), 465
Superblock, file systems and (超级块, 文件系统), 506
Superscalar processors (超标量处理器), 123
Swapping out, memory management and (替换走, 内存管理), 283
Switch/router, networking hardware (交换机/路由器, 网络硬件), 669
Switch statement (switch 语句), 40–41
Switched Ethernet, 673, 679
Switches, networking hardware (交换机, 网络硬件), 679–681
Symmetric multiprocessor (SMP) (对称多处理机), 578–579
Synchronization (同步)
  among threads (线程间同步), 533–540
  synchronization race (同步竞争), 528
  synchronous events (同步事件), 130
  synchronous logic circuits (同步逻辑电路), 91
  synchronous transfer unit (同步传输单元), 429–430
  synchronously produced data (同步产生数据), 428–429
System/360 series, IBM (System/360 系列, IBM), 309–310
System/370 series, IBM (System/370 系列, IBM), 310
System bus (系统总线), 11, 432–433
System calls (系统调用), 131, 158
System centric metrics (系统中心的度量), 245
System crashes (系统崩溃), 508
System stack (系统栈), 147
Systolic architectures (收缩体系结构), 122–123

T

Tag field, caches and (标记字段, 缓存), 363–364
Taking turns (轮流访问), 670
Tanenbaum, Andrew, 17, 274
Tasks (任务), 238–239
TCP/IP header (TCP/IP 包头), 687
Teardown, connection (拆除, 连接), 629
Telecommunications (电信), 710–711
Temporal locality (空间局部性), 355
Test-and-set instruction (Test-and-Set 指令), 575–578
Thicknet (粗缆网络), 712
Thinnet (细缆网络), 712–714
Third-level cache (3 级缓存), 357
Third party vendors (第三方供应商), 461–462
Thompson, Ken, 17, 274
Thrashing (颠簸), 241, 243, 338–342
Threads (线程), 238–239. *See* Multithreading (参见多线程)
  creation and termination (创建和终止), 523–525, 574–575
  thread communication (线程通信), 574–575
  thread control blocks (线程控制块), 567–568
  thread level parallelism (线程级并行), 596–599
  thread of control (控制线程), 524
  thread-safe wrappers (库调用线程安全的封装), 574
Threads library (线程库)
  kernel-level threads (内核级线程), 570–573
  safety and (安全性), 573–574
  Solaris threads and (Solaris 线程), 572–573
  user level threads and (用户级线程), 567–570, 573
Three-operand instructions (三操作数指令), 22, 60
Throughput (吞吐量), 171, 245, 452
Time-division multiplexing (TDM)(时分多路复用), 663, 669
Time of flight (传播时间), 689
Time quantum (时间段), 416
Time-shared operating systems (分时操作系统), 710
Time-sharing environments (分时环境), 240
Timeline of scheduling processes (调度进程的时间轴), 246
Timer device (定时器装置), 268
Timer interrupt handler (时间中断处理器), 263
Timesharing (分时), 13, 274
Timeslice (时间片), 259
Token ring (令牌环), 675–677
Tomasulo, Robert, 219
Tomasulo algorithm (Tomasulo 算法), 219–220
Top-half handlers (top-half 处理过程), Linux, 154
Top-level procedure, threads and (顶层过程, 线程), 560
Torvalds, Linus, 17, 274

Total execution time (总执行时间), 162
Tracker threads, data structure and (跟踪器线程, 数据结构), 527
Tracks, disk storage and (磁道, 磁盘存储), 440–441
Translation lookaside buffer (TLB) (变址转化缓存), 296, 343–346, 386–387, 399–400
Transmission control protocol (TCP) (传输控制协议), 629, 648–649
Transmission delay (传输延迟), 94, 689
Transmission errors (传输错误), 647–648
Transport layer (传输层)
  expected functionality of (期望功能), 634–646
  Internet protocols and (因特网协议), 629, 648–651
  pipelined protocols and (流水式协议), 640–642
  reliable pipelined protocol (可靠的流水式协议), 642–647
  stop-and-wait protocols and (停止并等待协议), 636–639
  transmission errors and (传输错误), 647–648
Traps (陷入), 131–132, 135, 282, 317. *See also* Program discontinuities (程序不连续性)
Tree interconnection network (树状的互联网), 600
Tree structure, file systems (树结构, 文件系统), 471
Tristate buffers (三态缓冲器), 87
Truth table (真值表), 119
Turing, Alan, 15–16
Turing Award (图灵奖), 712
Turing machine (图灵机), 15–16
Turnaround time (周转时间), 452
Two bus design (双总线设计), 87–89
Two-level page table (二级页表), 347
Two-operand instructions (双操作数指令), 60

U

Unconditional jump instruction (无条件跳转指令), 39–40
Unit of execution, threads and (执行单元, 线程), 524
Unit of memory access and transfer (内存访问和转换单元), 377
Univac, 470
Universal serial bus (USB) (通用串行总线), 462
UNIX
  file system commands (文件系统命令), 475
  history of (历史), 274
  hybrid allocation approach and (混合分配方式), 491
  i-node data structure and (索引节点数据结构), 497–498
  i-node tables and (索引节点表), 516
  network file system (网络文件系统), 706–707
  operating system (操作系统), 17
  programming and (编程), 699–706
  remote procedure call in (远程过程调用), 706–707
UNIX sockets (UNIX 套接字)
  client-server relationship and (客户端–服务器的关系), 702–703
  client side socket call (客户端套接字调用), 702
  creation of (创建), 699–700
  datagram socket communication (数据报套接字通信), 705
  establishing server communication and (建立服务器通信), 701–703
  interprocess communication and (进程间通信), 700–701
  server side socket call (服务器端套接字调用), 700
  server socket after bind call (已经绑定的服务器套接字), 701
  server socket after listen and accept calls (监听并接受呼叫的服务器套接字), 701
  socket library and (套接字库), 706
  stream socket data communication (数据流类型的套接字), 704
Upcalls (上行调用), 263, 502–503, 570
User centric metrics (用户中心的指标), 245
User datagram protocol (UDP) (用户数据报协议), 629, 634, 648–650
User/kernel mode (用户/内核模式), 268
User level threads (用户级线程), 567–570, 573
User mode (用户模式), 148, 149

User stacks (用户栈), 147, 149

V

Vacuum tubes (真空管), 12, 12f
Variable size partitions (可变区分区), 287–289
Variance in response time (响应时间的变化), 246
Variance in wait time (等待时间的变化), 452
VAX 11 architecture (VAX 11 体系结构), 59
Vector supercomputers (向量超级计算机), 226
Vector tables, interrupt (向量表, 中断), 147
Vectors (向量), 35, 134
Vertical microcode (垂直微码), 119
Very large scale integration (VLSI) (超大规模集成电路), 12
Victim page, picking (被替换页, 挑选), 320
Victim selection, page replacement and (被替换页选择, 页替换), 326
Video games (视频游戏)
  application hardware-operating system (应用硬件-操作系统)
  interactions (交互), 5–8, 7f
  audio-visual content and (音频/视频的内容), 4
  networked (联网), 5, 6f
  operating system role in (操作系统的作用), 5–8
  software architecture for (软件架构), 3, 3f
Video processing pipeline (视频处理流水线), 523
Virtual address (VA) (虚拟地址), 291, 399, 402
Virtual circuit (VC) (虚电路), 667, 669
Virtual file system (VFS) (虚拟文件系统), 509
Virtual local area networks (VLAN),
  networking hardware (网络硬件), 682
Virtual memory (虚拟内存)
  paged (分页虚拟内存), 290–297
  relative sizes of (相对大小), 296–297
  segmented (分段), 297–303
  size and latency and (对时间容量和延迟), 416
Virtual page number (VPN) (虚拟页编号), 291, 295–296, 399, 402–403
Virtual-to-physical address translation (虚拟地址到物理地址的转换), 400
Virtualization, processor design and (虚拟化, 处理器设计), 69
Virtually indexed physically tagged cache (虚拟索引物理标记的缓存), 401–402
Virtually tagged caches (虚拟标记缓存), 403
VLIW (very long instruction word) processors (超长指令字体系结构处理器), 123–124
von Neumann, John, 15
von Neumann architecture (冯·诺依曼体系结构), 122

W

Wear leveling (损耗均衡), 461, 509
Weighted arithmetic mean (WAM) (加权算术平均数), 162, 170
Windows 95, 275
Windows 2000, 275
Windows CE, 13
Windows NT, 275
Windows NT 4.0, 275
Windows Version 35, 7, 17, 275
Windows Vista, 275
Windows XP, 275
Wire delay (线延迟), 94, 611
Wireless LAN (无线 LAN), 674–675
Word operands (字操作数), 32–34
Word precision (字精度), 28
Working set (工作集), 340–341, 383
Workload (工作负载), 266
World Wide Web (WWW) (万维网), 632. *See also* Internet (参见因特网)
Worst case delay for signal propagation (最坏情况下信号传播延迟), 85
Write access to cache from CPU (CPU 对缓存的读访问), 370–375
  no write allocate (非写分配), 373
  write allocate (写分配), 372–373
  write-back policy (回写策略), 373–374
  write-through policy (直写写策略), 370–372
Write after read (WAR) data hazard (读后写数据冒险), 189–190, 199–200, 220
Write after write (WAW) data hazard (写后写数据冒险), 189–190, 199–200, 220–221
Write allocate, write-miss handling (写分配, 写缺失处理), 372–373

Write-back policy（回写策略），373–374
Write buffer（写缓冲区），371，393
Write merging（写合并），374
Write stall，pipelining and caches（写拖延，流水线和缓存），369，371
Write-through policy（直接写策略），370–372
Write-update protocol（写入更新协议），580

X

Xbox，269
Xeon，Intel，76，121，222，610
Xerox，713
Xerox PARC，713
XScale processors（XScale 处理器），123

Y

Yahoo!，8，633

Z

Z bit，control unit（Z 位，控制单元），116
Z register，control unit（Z 寄存器，控制单元），92–93
Zero-operand instructions（零操作数指令），59–60
Zoned bit recording（ZBR），magnetic disk and（划位记录，磁盘），441–444