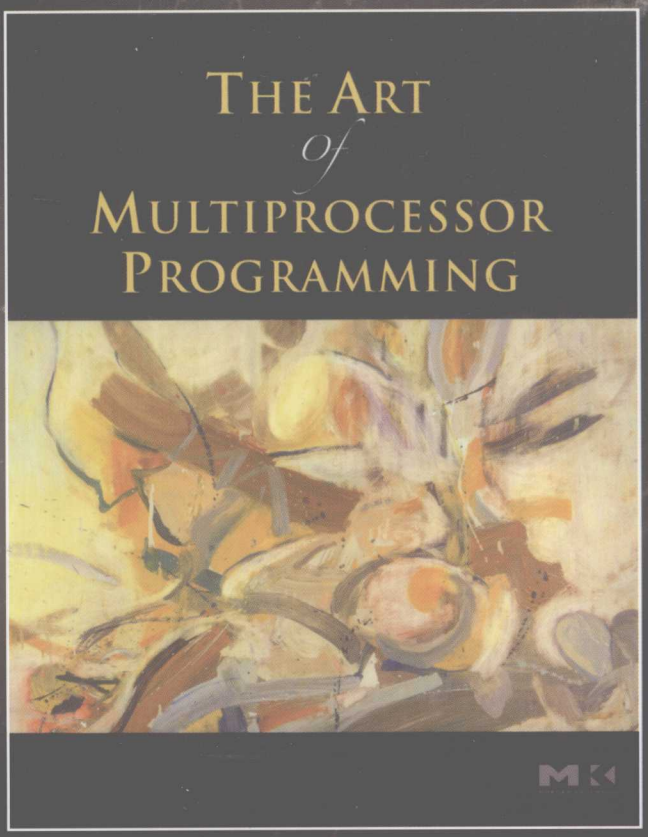
HZ BOOKS 华章教育
ELSEVIER
计 算 机 科 学 丛 书

多处理器编程的艺术

(美) Maurice Herlihy 布朗大学 (以) Nir Shavit 特拉维夫大学 著
金海 胡侃 华中科技大学 译



The Art of Multiprocessor Programming

机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

多处理器编程的艺术

(美) Maurice Herlihy 布朗大学 (以) Nir Shavit 特拉维夫大学 著 金海 胡侃 华中科技大学 译

THE ART of MULTIPROCESSOR PROGRAMMING

The Art of Multiprocessor Programming

机械工业出版社
China Machine Press

本书从原理和实践两个方面全面阐述了多处理器编程的指导原则，包含编制高效的多处理器程序所必备的算法技术。此外，附录提供了采用其他程序设计语言包（如C#、C及C++的PThreads库）进行编程的相关背景知识以及硬件基础知识。

本书适合作为高等院校计算机及相关专业高年级本科生及研究生的教材，同时也可作为相关技术人员的参考书。

本书版权登记号：图字：01-2009-1340

图书在版编目（CIP）数据

多处理器编程的艺术 /（美）荷里希（Herlihy, M.），（以）谢菲特（Shavit, N.）著；金海，胡侃译. —北京：机械工业出版社，2009

（计算机科学丛书）

书名原文：The Art of Multiprocessor Programming

ISBN 978-7-111-26805-5

Ⅰ. 多…　Ⅱ. ① 荷…　② 谢…　③ 金…　④ 胡…　Ⅲ. 微处理器—程序设计　Ⅳ. TP332

中国版本图书馆CIP数据核字（2009）第099149号

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：迟振春

三河市明辉印装有限公司印刷

2009年8月第1版第1次印刷

184mm × 260mm · 23.25印张

标准书号：ISBN 978-7-111-26805-5

定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，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，不仅擘划了研究的范畴，还揭示了学术的源变，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分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

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

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社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

译 者 序

每逢我们在多处理器平台上进行编程时，往往会有这么一种感觉：即使已熟练掌握了系统提供的各种同步原语，但所编制的并行程序的实际性能似乎总有些差强人意，并不十分理想。究其原因，问题的根结在于多处理器编程应是一门科学和艺术完美结合的学科。若要在多处理器系统结构上编制出性能良好的并行程序，要求设计者不仅要精通多处理器系统结构、并行算法以及一些系统构建工具，还应能基于一种设计理念，充分发挥个人的想象空间，合理搭配这些知识和资源，从而和谐地构建完整的系统，使设计者能比底层硬件和操作系统“做得更好”。也就是说，在编写多处理器程序时，要能同时从宏观和微观两种角度分析问题，并能在这两种角度之间灵活地转换。

自20世纪中叶第一台通用电子计算机研制成功以来，程序的编制大多是基于顺序计算模型的，程序的执行过程是操作的有序序列。由于顺序计算机能够用图灵机精确地描述，因此顺序计算的编程能在一组易于理解且完备定义的抽象之上进行，而不需要了解底层的细节。近年来，尽管单处理器仍在发展，但由于指令级并行的开发空间正在减少，再加上散热等问题限制了时钟频率的继续提高，所以单处理器发展的速度正在减缓，这最终导致了起源于在单独一个晶片上设计多个内核的多处理器系统结构的出现。多处理器系统结构允许多个处理器执行同一个程序，共享同一程序的代码和地址空间，并利用并行技术来提高计算效率。在这种计算模型中，并发程序的执行可以看做是多个并发线程对一组共享对象的操作序列，为了在这种异步并发环境中获得更好的性能，底层系统结构的细节需要呈现给设计者。

作为一名优秀的程序设计员，在编写多处理器程序之前首先应弄清楚多处理器计算机的能力和限制是什么；在异步并发计算模型中什么问题是可解决的，什么问题是不可解决的；是什么使得某些问题很难计算，而又使另一些问题容易计算。这要求设计者具备一定的多核并行计算理论基础知识，掌握多处理器系统结构上并发计算模型的可计算性理论及复杂性理论。其次，应掌握基本的多核平台上的并行程序设计技术，包括并行算法、同步原语以及各种多核系统结构。

Maurice Herlihy 教授和 Nir Shavit教授在并发程序设计领域具有很深的造诣，并拥有40年以上一起从事并发程序设计教学的合作经验。他们对多处理器并行程序设计技术做出了巨大的贡献，并因此而成为2004年ACM/EATCS Gödel奖的共同获得者。这本由他们合著的专著致力于解决如何采用更好的并行算法来克服多核并发程序并行度低的问题。

Amdahl定律早已明确地告诉我们，从程序本身可获得的并行度是有限的，加速比的提高主要取决于程序中必须增加的串行执行部分，而这部分又往往包含着具有相对较高开销的通信和协作。因此，在多处理器系统结构上，如何提高程序中必须串行部分的并行度，以及降低并行处理器中远程访问的时延是我们目前面临的两大技术挑战。对这些问题的有效解决，必须依靠软件技术和硬件技术的改进和发展。本书则侧重于对前一个挑战的研究。

作者先从宏观的抽象角度出发，在一个理想化的共享存储器系统结构中研究各种并行算法的可计算性及正确性。通过对这些经典算法的推理分析，向读者揭示了现代协作范例和并

发数据结构中所隐藏的核心思想，使读者学会如何分析饥饿和死锁等微妙的活性问题，深层次地研究现代硬件同步原语所应具有的能力及其特性。随后，从微观的实际角度出发，针对当今主流的多处理器系统结构，设计了一系列完美高效的并行算法及并发数据结构，并对各种算法的效率及其机理进行了分析。所有的设计全部采用Java程序设计语言详细地描述，可以非常容易地将它们扩展到实际应用中。

本书的前6章讲述了多处理器程序设计的原理部分，着重于异步并发环境中的可计算性问题，借助于一个理想化的计算模型来阐述如何描述和证明并行程序的实际执行行为。由于其自身的特点，多处理器程序的正确性要比顺序执行程序的正确性复杂得多，书中为我们展现了一系列不同的辅助论证工具，令人有耳目一新之感。随后的11章阐述了多处理器程序设计的实践部分。由于在多处理器环境中编写程序时，底层系统结构的细节并不像编写顺序程序那样被完全隐藏在一种编程抽象中，因此，本书在附录B介绍了多处理器硬件的基础知识。最后的第18章介绍了当今并发问题研究中最先进的事务方法，可以预言这种方法在今后的研究中将会越来越重要。

感谢王振飞博士在本书第13～18章翻译中所做的工作，感谢胡丽婷、袁旻昊、耿玮、蔡硕、张亮同学在本书翻译的初始资料整理中所给予的帮助。

译 者

2009年4月

前　言

本书可以作为高年级本科生的教材，也可以作为相关技术人员的参考书。

读者应具备一定的离散数学基础知识，能够理解“大O”符号的含义，以及它在NP完全问题中所起的作用；熟悉计算机系统的基本组成部件，如处理器、线程、高速缓存等；为了能够理解书中的实例，还需要具备初步的Java知识。（在使用这些高级程序设计语言之前，本书阐述了语言的相关功能特征。）书中提供两个附录以供读者参考：附录A包含程序设计语言的相关知识，附录B给出了多处理器硬件系统结构的相关内容。

本书前三分之一涵盖并发程序设计的基本原理，阐述并发程序设计的编程思想。就像掌握汽车驾驶技术、烹饪食物和品尝鱼子酱一样，并发思维也需要培养，需要适当的努力才能学好。希望立刻动手编程的读者可以跳过这部分的大多数内容，但仍需阅读第2章及第3章的内容，这两章包含了理解本书其他部分所必不可少的基本知识。

在原理部分中，首先讨论了经典的互斥问题（第2章），包括诸如公平性和死锁这样的基本概念，这对于理解并发程序设计的难点尤为重要。然后，结合并发执行和并发设计中可能出现的各种情形和开发环境，给出了并发程序正确性的定义（第3章）。接着，研究了对并发计算至关重要的共享存储器的性质（第4章）。最后，介绍了几种为实现高并发性数据结构而使用的同步原语（第5、6章）。

对于每一位渴望真正掌握多处理器编程艺术的程序设计人员来说，花上一定的时间去解决本书第一部分所提及的问题是很有必要的。虽然这些问题都是理想化的，但它们为编写高效的多处理器程序提供了非常有益的编程思想。尤为重要的是，通过在问题解决中获取的思维方式，能够避免出现那些初次编写并发程序的设计人员普遍易犯的错误。

接下来的三分之二讲述并发程序设计的具体实践。每章都有一个相应主题，阐明一种特定的程序设计模式或者算法技巧。第7章从系统和语言这两个不同的抽象层面，讨论争用及自旋锁的概念，强调了底层系统结构的重要性，指出对于自旋锁性能的理解必须建立在对多处理器层次存储结构充分理解的基础上。第8章涉及等待及管程锁的概念，这是一种常用（特别是在Java中）的同步用语。第16章包括并行性及工作窃取问题，第17章则介绍了障碍技术，这种技术往往在具有并发结构的应用中得以广泛的使用。

其他章节讲述各种类型的并发数据结构。它们均以第9章的概念为基础，因此建议读者在阅读其他章节之前首先阅读第9章的内容。该章采用链表结构来阐明各种类型的同步模式，包括粗粒度锁、细粒度锁及无锁结构。第10章则借助于FIFO队列说明在使用同步原语时可能出现的ABA问题，第11章通过栈描述了一种重要的同步模式——消除，第13章通过哈希映射阐述如何利用固有并行进行算法设计。高效率的并行查找技术则借助于跳表来阐述（第14章），而如何通过降低正确性标准来获得更高的性能则通过优先级队列进行了阐述（第15章）。

最后，第18章介绍了在并发问题的研究中新出现的事务方法，可以确信这种方法在不远的将来会变得越来越重要。

并发性的重要性还没有得到人们的广泛认可。在此，引用《纽约时报》1989年关于IBM PC中新型操作系统的一段评论：

真正的并发（当你唤醒并使用另一个程序时原来的程序仍继续运行）是非常令人振奋的，但对于普通使用者来说用处却很小。您能有几个程序在执行时需要花费数秒甚至更多的时间呢？

致谢

感谢Doug Lea、Michael Scott、Ron Rivest、Tom Corman、Michael Sipser、Radia Pearlman、George Varghese 和Michael Sipser在本书出版过程中所做的努力和给予的帮助。

感谢在本书的起草和修订过程中，向我们提供了大量宝贵意见的所有学生、同事和朋友：Yehuda Afek，Shai Ber，Martin Buchholz，Vladimir Budovsky，Christian Cachin，Cliff Click，Yoav Cohen，Dave Dice，Alexandra Fedorova，Pascal Felber，Christof Fetzer，Shafi Goldwasser，Rachid Guerraoui，Tim Harris，Danny Hendler，Maor Hizkiev，Eric Koskinen，Christos Kozyrakis，Edya Ladan，Doug Lea，Oren Lederman，Pierre Leone，Yossi Lev，Wei Lu，Victor Luchangco，Virendra Marathe，John Mellor-Crummey，Mark Moir，Dan Nussbaum，Kiran Pamnany，Ben Pere，Torvald Riegel，Vijay Saraswat，Bill Scherer，Warren Schudy，Michael Scott，Ori Shalev，Marc Shapiro，Yotam Soen，Ralf Suckow，Seth Syberg，Alex Weiss和Zhenyuan Zhao。同时，也向在这里没有提及的许多朋友表示道歉和感谢。

感谢Mark Moir、Steve Heller和Sun公司的Scalable Synchronization小组成员，他们为本书的撰写提供了巨大的支持和帮助。

目　录

第1章 引 言

计算机产业正在经历着一场重大的结构重组和巨变，在没有其他变革之前，这个过程无疑将会继续进行。主要的芯片制造商，至少是现在，都纷纷放弃尝试研制速度更快的处理器。虽然摩尔定律仍旧适用：每年集成在同样大小空间中的晶体管数越来越多，然而，由于散热问题难于解决，它们的时钟速度无法继续得到提高。取而代之的做法是，制造商开始转向“多核”系统结构的研制，由多个处理器（多核）共享硬件高速缓存直接进行通信。通过将多个处理器同时分配给单一任务以获得更高的并行性，从而提高计算的效率。

多处理器系统结构的普及对计算机软件的发展带来了深刻的影响。直至今日以前，技术的进步意味着时钟速度的提升，时钟本身的加速导致了软件执行效率的提高。今天，这种搭便车的现象已不复存在，技术的进步不再指时钟速度的提升而是指并行度的提升，并行问题已经成为现代计算机科学的主要挑战。

本书着重讲述共享存储器通信方式下的多处理器编程技术。通常称这样的系统为*共享存储器的多处理器*，现在称之为多核。在各种规模的多处理器系统中都存在着不同的编程挑战——对于小规模的系统来说，需要协调单个芯片内的多个处理器对同一个共享存储单元的访问；对于大规模系统来说，需要协调一台超级计算机中各个处理器之间的数据路由。其次，现代计算机系统所固有的异步特征也给多处理器编程带来了挑战：在没有任何警示的情形下，系统的活动可以被各种不同的事件中止或延迟，例如中断、抢占、cache缺失和系统故障等。这种延迟现象本身是无法预测的，时延的长短也是不确定的：cache缺失可以造成不到十条指令执行时间的时延，页故障可能造成几百万条指令执行时间的时延，而操作系统抢占则会导致多达上亿条指令执行时间的时延。

本书从原理和实践这两个互补的方面阐述多处理器的程序设计。*原理*部分着重于*可计算性理论*：理解异步并发环境中的可计算问题。借助于一个理想化的计算模型，对异步并发环境中什么是可解的这一问题进行了深入研究。在这个模型中，多个并发线程对一组共享对象进行操作，这些并发线程的对象操作序列被称为*并发程序*或*并发算法*。该模型实质上也正是Java™、C#及C++线程库中所采用的计算模型。

令人不可思议的是，的确存在一些易于说明的共享对象，我们无法采用任何并发算法来实现。因此，在编写多处理器程序之前弄清楚什么问题不能用计算机解决是十分重要的。大多数困扰多处理器程序员的问题都源自于计算模型自身的限制，所以，对并发共享存储器模型中可计算性理论的理解是学习多处理器编程必不可少的一个环节。书中与原理相关的章节向读者展现了各种各样的可计算问题，以帮助读者尽快地了解异步可计算性理论，同时也讲述了如何通过硬件和软件机制来解决这些问题。

理解可计算性的关键在于描述和证明特定程序的实际执行行为，更准确地说，即*程序正确性*问题。由于其自身的特点，多处理器程序的正确性比顺序程序的正确性更为复杂，因此，需要一系列不同的辅助论证工具来证明并发程序的正确性，甚至有可能只是为了“非形式化地证明”程序正确性（实际上程序员往往这么做）。顺序程序的正确性主要关心程序的各种安全特性。*安全性*说明了“不好的事件”绝不会发生。例如，即使在断电时，交通指示灯也决

不给任何方向显示绿灯。同样，并发程序的正确性也需要考虑安全性，但要考虑如何确保多个并发线程在各种可能的交互情形下的安全性，这使问题的解决变得难上加难。另外，还要考虑一个同样重要的因素，并发程序的正确性包括了各种各样的活性特性，而这种特性是顺序程序执行中所不会出现的。所谓活性，是指一个特定的“好的事件”一定会发生。例如，红色指示灯最终一定会变为绿灯。本书原理部分的最终目标就是要引入一系列分析推理并发程序的衡量标准和方法，为接下来研究现实对象和程序的正确性奠定基础。

本书的第二部分阐述多处理器程序设计的具体实践，着重于并发程序性能的分析。多处理器并发算法的性能分析与顺序算法的性能分析在风格上完全不同。顺序程序设计是基于一组易于理解且完备定义的抽象来进行的。编写顺序程序时，不需要了解底层的详细细节，例如，在硬盘和内存之间如何交换页面，在层次结构的高速缓存中如何移进/移出那些较小的内存单位。这种复杂的存储器层次结构实质上对程序员是不可见的，它被隐藏在一种完全的编程抽象之中。

然而在多处理器环境下，这种编程抽象被打破了，至少从性能角度来讲应该这样做。为了获得足够好的性能，程序设计人员有时需要比底层存储器系统“做得更好”，他们编写的程序代码可能让那些不熟悉多处理器系统结构的人感到莫名其妙。或许某一天，并发系统结构会像今天的顺序系统结构一样支持完全的抽象，然而到那时，程序设计人员早已了解这种新的系统结构了。

本书的原理部分讲述了一组共享对象和编程工具，着眼于每种对象和工具自身的能力，并借助于它们引出一些高层次的问题：用自旋锁来说明争用，用链表阐述数据结构设计中锁的作用等。这些问题对程序的性能都有着重要的影响，希望读者能充分理解它们，并能将所理解的内容应用于日后具体的多处理器系统设计中。最后，通过讨论诸如事务内存这种目前最先进的技术，作为本书的结束。

下面简要说明本书的写作风格。尽管有很多合适的语言可供选择，但本书仍采用了Java程序设计语言。当然，有大量的理由可以解释为何要做出这种选择，然而这样的话题还是更适于在闲暇时讨论！附录解释了Java所支持的一些概念在其他的常用语言或库中是如何表示的，同时也介绍了关于多处理器硬件的一些基础知识。纵观全书，我们尽量避免列出关于程序和算法性能的具体数据，而是从一般情形来考虑。这样做的理由是：多处理器系统之间差异很大，在一台机器上工作良好的并发程序并不代表在另一台机器上也有同样的表现，紧密结合一般情形能够保证本书所陈述的结论具有更加长久的有效性。

在每一章的末尾都提供了相关文献的引用，读者可以根据参考文献目录找到相应内容以便进一步阅读。此外，每章都提供了一些习题，读者可以据此检查自己对知识的理解程度。

1.1 共享对象和同步

在开始新工作的第一天，老板要求在一台能够支持10个并发线程的并行机上编写出查找$1\sim10^{10}$之间素数的程序（不要考虑为什么这样做）。机器是按照分钟租用的，程序运行时间越长，花费也就越大。如果想给老板留下一个不错的印象，应该怎样去做？

在最初的尝试中，可能会为每个线程分配一个大小相等的输入域。各个线程分别找出10^9个数字内的素数，如图1-1所示。这种方法可能会由于一个简单但很重要的原因而最终导致失败，那就是相同大小的输入范围并不意味着相同的工作量。素数的分布是不均匀的：在$1\sim10^9$之间有很多素数，但分布在$9*10^9\sim10^{10}$之间的素数却非常少。更为糟糕的是，整个范围内

不同素数的计算时间也是不相同的：判断一个较大的数是否为素数通常要比判断较小的数所花费的时间更长。简而言之，没有理由认为这种方式能够使得整个工作是由所有的线程平均承担完成的，甚至也不清楚哪个线程承担的工作最多。

```
1  void primePrint {
2    int i = ThreadID.get();  // thread IDs are in {0..9}
3    int block = power(10, 9);
4    for (int j = (i * block) + 1; j <= (i + 1) block; j++) {
5      if (isPrime(j))
6        print(j);
7    }
8  }
```

图1-1 通过等分输入域达到负载平衡。对{0..9}中的每个线程分配同样大小的输入子集

在线程间划分工作的另一种可行方案就是为每个线程一次分配一个整数（图1-2）。当一个线程结束对该整数的判断后，再次请求分配另一个整数。为此，需要引入一个*共享计数器*对象。该对象将一个整型值封装起来，线程通过调用getAndIncrement()方法对该整型值进行自增操作，并返回未被增加前的先前值。

```
1  Counter counter = new Counter(1);  // shared by all threads
2  void primePrint {
3    long i = 0;
4    long limit = power(10, 10);
5    while (i < limit) {               // loop until all numbers taken
6      i = counter.getAndIncrement(); // take next untaken number
7      if (isPrime(i))
8        print(i);
9    }
10 }
```

图1-2 通过共享计数器达到负载平衡。每个线程对动态确定的数字进行判断

图1-3是Counter对象的Java实现。该计数器由单线程调用时效果很好，但由多线程共享使用时却会出现错误。其问题的根本就在于语句

```
return value++;
```

实质上是下面几行代码的缩写方式：

```
long temp = value;
value = temp + 1;
return temp;
```

在这段代码中，value是对象Counter的一个域，它被所有的线程所共享。但是，每个线程都有一个它自己的本地拷贝temp，该拷贝是线程的局部变量。

假设线程*A*和线程*B*同时调用Counter的getAndIncrement()方法。那么，*A*和*B*有可能同时从value中读入1，然后分别将各自的局部变量temp设置为1，再将value设置为2，最终，*A*和*B*都返回了value的先前值1。显然，这种情形并不是我们所期望的：对计数器getAndIncrement()方法的并发调用返回了同一个值。我们希望不同的线程返回不同的值。事实上，还有可能出现更坏的情形：一个线程从value中读入了1，在它将value置为2之前，另一个线程执行了多次增量循环，读入1设置为2，接着读入2设置为3。当第一个线程最终完

成其增量操作并将value设置为2时，它实际上是将value的值从3改回到2。

```
1  public class Counter {
2    private long value;    // counter starts at one
3    public Counter(int i) {    // constructor initializes counter
4      value = i;
5    }
6  public long getAndIncrement() { // increment, returning prior value
7      return value++;
8    }
9  }
```

图1-3 共享计数器的实现

出现上述问题的根本原因在于对计数器值的增加需要在共享变量上执行两种不同类型的操作：将value的值读入temp变量，并将temp的值写入Counter对象。

类似的情形在现实生活中同样也会出现。当在走廊中行走时，需要躲过向你迎面走来的人。你可能发现那一瞬间自己一会儿向右，一会儿向左，这么来回好几次，因为另一个人也在试图这么做以避免与你碰撞。有时成功避开了，但有时还是撞上了。实际上，正如在后面的章节中将要讲述的那样，这种碰撞在很多时候是无法避免的。[⊖]直观上来看，你和向你迎面走来的人都在做两件事：观察（“读”）对方当前的位置，然后移向（“写”）另一边。然而问题是，当你在读对方的位置时，无法知道他下一秒是继续待着还是移动躲闪。你和对面的这个陌生人必须决定谁从左边走谁从右边走。同样，共享计数器的各个线程也需要决定谁先使用谁后使用。

在第5章将会看到，现代的多处理器硬件通常都提供了特殊的读-改-写指令，允许线程以原子的硬件操作来读、写或修改存储器的值。对于上述的Counter对象，可以采用这种硬件方式原子地完成计数器值的自增。

同样，通过在软件（只使用读、写指令）中保证一个时刻只有一个线程在执行读/写操作序列，也可以获得这种原子的行为效果。这种确保一个时刻只允许一个线程执行特定代码段的问题称为互斥问题，它是多处理器程序设计中经典的协作问题之一。

在实际编程中并不需要由自己来设计互斥算法（而是调用库）。但是，深入地理解互斥算法的实现是从全局上掌握并发计算的基础。同样，对于死锁、有界公平性、不同于非阻塞方式的阻塞同步等这些普遍但很重要问题的具体解决方法，也需要进行深入的研究。

1.2 生活实例

并发协作问题（如互斥）应该当作实际的具体问题来处理，而不应看作是一种编程训练。本节通过一些现实生活实例阐述基本的并发问题。像大多数讲述这些实例的其他作者一样，我们也是在原有的情节上重述这些故事（见本章末尾的注释）。

Alice和Bob是一对邻居，他们共用一个院子。Alice养了一只猫而Bob养了一只小狗。这两只小宠物都喜欢在院子里跑来跑去，然而（自然地）它们总是不能融洽地相处。在发生了一些不愉快的事情后，Alice和Bob决定采取措施，让两只小宠物不同时出现在院子里。显然，应该排除不许任一只动物待在院子里的做法。

应该怎样去做呢？Alice和Bob先要约定一种相互都可以接受的过程以决定他们该做什么。

⊖ 无法使用类似“总是靠右行”这种预防性的措施，因为对方有可能是英国人。

这种约定称为协作协议（简称协议）。

由于院子很大，Alice无法从窗户观察到Bob的小狗是不是在院子里。当然，她可以到Bob家敲门进去看看，但这太浪费时间，况且下雨怎么办呢？Alice也可以站在窗户边冲着Bob的屋子大声喊叫："Bob！我的猫可以出来了吗？"然而，问题是Bob有可能听不见，他或许在看电视，或许在招待他的女朋友，也可能出去买狗食了。他们两人同样也可以尝试通电话，但也会出现问题，例如Bob正在洗澡或给电话换电池等。

于是，Alice想出了一个好主意。她在Bob家的窗台上放了几个空啤酒罐（如图1-4所示），用绳子将它们一个个地绑起来，并将绳子的一端系在自己的屋子里。Bob也按照同样的方法在Alice家的窗台上安放啤酒罐。当Alice想给Bob发出信号时，则猛拉绳子打翻其中一个啤酒罐。若Bob发现一个啤酒罐被打翻，则重新摆好它。

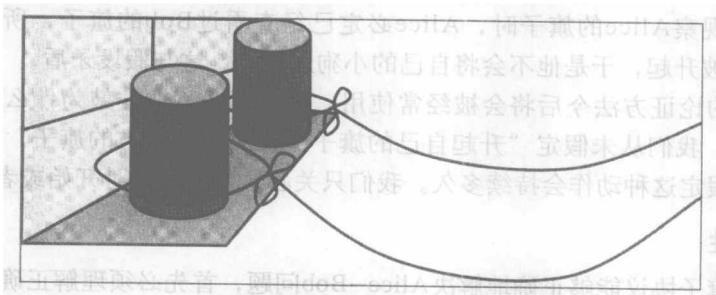


图1-4 使用啤酒罐进行通信

远程啤酒罐控制看似不错，但仍存在很大的缺陷。问题在于Alice只能在Bob的窗台上摆放有限个数的啤酒罐，迟早有一天，她会打翻所有的啤酒罐。即使Bob总是及时地扶正被打翻的啤酒罐，然而，一旦他去坎昆度假该怎么办呢？只要是指望由Bob来扶正啤酒罐，那么Alice早晚都会用完所有的啤酒罐。

Alice和Bob于是又尝试使用另外一种方法。他俩各自在对方很容易看到的地方竖起一个旗杆。如果Alice想让她的猫去院子里活动，则按以下步骤进行处理：

1. 升起她自己的旗子。
2. 若Bob的旗子是降下来的，则将她的猫放出去。
3. 当猫返回屋子时，将她自己的旗子降下。

Bob的操作则相对复杂一些。

1. 升起他自己的旗子。
2. 若Alice的旗子处于升起状态，则重复执行下列操作：

a) 降下他自己的旗子。

b) 等待直到Alice的旗子被降下。

c) 重新升起他自己的旗子。

3. 只要Bob升起了自己的旗子并且发现Alice的旗子是降下来的，就可以将自己的小狗放出去。
4. 当小狗返回屋子里时，Bob则降下他自己的旗子。

下面进一步地研究这个用于解决Alice-Bob问题的协议。从直观上来看，该协议之所以能够正常地工作是由于下面的旗子原则。如果Alice和Bob都

1. 升起自己的旗子，然后