PROGRAMMER TO PROGRAMMER™

Kongfz.com 孔夫子旧书网

Professional Multicore Programming

Design and Implementation for C++ Developers

C++多核高级编程

(美) Cameron Hughes
Tracey Hughes 著
齐 宁 译

清华大学出版社

Cameron Hughes，Tracey Hughes
Professional Multicore Programming: Design and Implementation for C++ Developers
EISBN：978-0-470-28962-4
Copyright © 2008 by Wiley Publishing, Inc.
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc.授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号　图字：01-2009-2815

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

C++多核高级编程/(美)休斯(Hughes，C.)，(美)休斯(Hughes，T.)　著；齐宁　译.
—北京：清华大学出版社，2010.3
书名原文：Professional Multicore Programming: Design and Implementation for C++ Developers
ISBN 978-7-302-22274-3

I. C…　II. ①休…②休…③齐…　III. C 语言—程序设计　IV. TP312

中国版本图书馆 CIP 数据核字(2010)第 046015 号

责任编辑：王　军　于　平
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：何　芊
出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印　刷　者：北京鑫丰华彩印有限公司
装　订　者：三河市新茂装订有限公司
经　销：全国新华书店
开　本：185×260　**印　张**：36.25　**字　数**：882 千字
版　次：2010 年 3 月第 1 版　**印　次**：2010 年 3 月第1 次印刷
印　数：1～4000
定　价：69.80 元

产品编号：030640-01

译 者 序

随着多核时代的到来，原本只属于高端应用的并行化编程也随之变得越来越普及。可以说，在处理器平台多核的大潮中，单纯的芯片制造工艺和技术已经不足以体现和发挥出多核所带来的更高处理能力以及性能优势，具备在多核环境中多线程工作的软件将会成为发挥多核高效率的保证。

多核技术在单个封装内集成了更多的核，为实现真正并行提供了物质基础。那么究竟如何设计和编写并行应用程序才能充分发挥多核架构的资源优势？这正是软件开发人员所要解决的问题。本书从这一需求出发，期望为面向多核架构设计和编写并行应用程序的开发人员提供一些力所能及的帮助。

在翻译本书的过程中，我们的切身体会就是本书并不仅仅是简单地介绍如何编写多核程序，更重要的是为程序开发人员提供了如何顺利从命令式顺序编程迁移到声明式并行编程的方法。作者以多个应用实例贯穿本书，一步步引导读者领会如何从问题陈述逐步精化，最终实现并行程序。

本书由齐宁和董泽惠翻译完成，Be Flying 工作室(http://blog.csdn.net/be_flying)负责人肖国尊负责翻译质量和进度的控制管理。书中文字和内容力求忠实原著，但限于译者水平和时间紧迫，翻译过程中难免出现不妥之处和错误，敬请广大读者批评指正。

关 于 作 者

Cameron Hughes 是一名专业的软件开发人员。他是 CTEST 实验室的软件工程师，同时还是 Youngstown 州立大学的编程人员/分析师。Cameron Hughes 有着超过 15 年的软件开发经验，参与过各种规模的软件开发工作，从商业和工业应用到航空设计和开发项目。Cameron 是 Cognopaedia 的设计者，目前是运行在 CTEST 实验室的 Pantheon 上的 GRIOT 项目的领导者。Pantheon 是具有 24 个节点的多核集群，用于多线程搜索引擎和文本提取程序的开发。

Tracey Hughes 是 CTEST 实验室的高级图像程序员，负责开发知识和信息的可视化软件。Tracey Hughes 是利用 CTEST 实验室的知识可视化的 M.I.N.D、C.R.A.I.G、NOFAQS 等项目的主要设计人员。她经常致力于 Linux 开发软件。她还是 GRIOT 项目的小组成员。

Cameron 和 Tracey Hughes 还是关于软件开发、多线程和并行编程方面的 6 本著作的作者，这 6 本著作是：*Parallel and Distributed Programming Using C++*、*Linux Rapid Application Development*、*Mastering the Standard C++ Classes*、*Object - Oriented Multithreading Using C++*、*Collection and Container Classes in C++*和 *Object-Oriented I/O Using C++ Iostreams*。

前 言

多核革命即将到来。并行处理不再是超级计算机或集群的专属领域，入门级服务器乃至基本的开发工作站都拥有硬件级和软件级并行处理的能力。问题是这对于软件开发人员意味着什么？对软件开发过程会有怎样的影响？在谁拥有速度最快的计算机的竞争中，芯片生产商更倾向于在单独的芯片上放置多个处理器，而不是提高处理器的速度。迄今为止，软件开发人员尚能依赖于新的处理器，在不对软件做出任何实际改进的情况下提高软件的速度，但是这样的情况将成为过去。为了提高总体系统性能，计算机供应商已经决定增加更多的处理器，而不是提高时钟频率。这意味着如果软件开发人员希望应用程序从下一个新的处理器受益，就必须对应用程序进行修改以利用多处理器计算机。

尽管顺序编程和单核应用程序开发仍会有一席之地，但软件开发将向多线程和多进程转变。曾经仅被理论计算机科学家和大学学术界所关注的并行编程技术，现在正处于为大多数人所接受的过程中。多核应用程序设计和开发的思想如今成为人们关注的主流。

学习多核编程

本书使用一般软件开发人员能够理解的术语介绍多核编程的基本知识。为读者介绍了为多处理器和多线程体系结构进行编程的基础知识，对并行处理和软件并发的概念进行了实用的介绍。本书介绍的是深奥的、不易理解的并行编程技术，但将使用一种简单、可理解的方式来介绍它们。我们介绍了并发编程和同步的缺陷与陷阱以及应对之策，对多处理和多线程模型进行了直截了当的讨论。本书提供了大量的编程实例，示范了如何实现成功的多核编程。本书还包含了调试及测试多核编程的方法与技术。最后，我们示范了如何使用跨平台技术来利用处理器的具体特性。

不同的视角

本书的内容是为对多核编程和应用程序开发有着不同切入点的广大读者设计的。本书的读者包含但不限于：

- 库及工具制作人员
- 操作系统程序员
- 内核开发人员
- 数据库服务器及应用服务器的设计人员及实现人员
- 科学应用程序员以及使用计算密集型应用程序的用户
- 应用程序开发人员

PDG

● 系统程序员

每组人员都会从不同的视角来了解多核计算机。有些人关心的是使用自底向上的方法，需要开发利用特定硬件和特定供应商的特性的软件。对于上述人员而言，他们希望更加详尽地介绍多线程处理的知识。其他人员可能对自顶向下的方法感兴趣，他们不希望为并发任务同步或线程安全的细节费心，而是倾向于使用高级库和工具来完成工作。还有一些人需要混合使用自底向上和自顶向下的方法。本书提供了对多核编程的多种视角的介绍，涵盖了自底向上和自顶向下的方法。

解决方案是多范型方法

首先，我们承认不是每个软件解决方案都需要多处理或多线程。有些软件解决方案通过使用顺序编程技术能够更好地实现(即使目标平台是多核的)。我们的方法是以解决方案和模型作为驱动。首先，针对问题开发出模型或解决方案。如果解决方案要求某些指令、过程或任务并发地执行，那么就决定了最好使用哪组技术。这个方法同强迫解决方案或模型去适合一些预先选择的库或开发工具的方法相反。技术应当遵从解决方案。尽管本书讨论库和开发工具，但并不偏向任何具体生产商库或工具集。尽管书中包含了利用特定硬件平台的实例，但我们依赖跨平台方法，使用 POSIX 标准操作系统调用和库，并且仅使用国际化 C++标准所支持的 C++特性。

面对多处理和多线程中的挑战和障碍，我们倡导组件方法。主要的目的是利用框架类作为并发的构建块。框架类被面向对象互斥量(mutex)、信号量(semaphore)、管道(pipe)、队列(queue)和套接字(socket)所支持。通过使用接口类，显著降低了任务同步和通信的复杂度。在我们的多线程和多处理应用程序中，控制机制主要是 agent 驱动。这意味着在本书中，您将看到应用程序架构支持软件开发的多范型(multiple-paradigm)方法。

我们对组件实现使用面向对象编程技术，对控制机制主要使用面向 agent 编程技术。面向 agent 编程的思想有时被逻辑编程技术支持。随着处理器上可用内核数目的增加，软件开发模型将会越发地依赖面向 agent 编程和逻辑编程。本书包含了对软件开发的这种多范型方法的简介。

为何使用 C++

事实上，每个平台和操作环境中均有可用的 C++编译器。ANSI(American National Standards Institute，美国国家标准协会)和 ISO(International Organization for Standardization，国际标准化组织)已经为 C++语言和它的库定义了标准。C++语言具有可靠的开源实现以及商业实现，并且已经被全世界的研究人员、设计人员和专业开发人员所广泛接受。C++语言被用于解决各种各样的问题，从设备驱动到大规模的工业应用。C++语言支持软件开发的多范型方法。在 C++中，我们可以无缝地实现面向对象设计、逻辑编程设计和面向 agent

设计。我们还可以在必要时使用结构化编程技术或低级编程技术。这种灵活性正是新的多核技术所需要加以利用的。此外，C++编译器为软件开发人员提供了多核处理器的新特性的直接接口。

UML 图

本书中的很多图使用了 UML(Unified Modeling Language，统一建模语言)标准。特别是使用活动图、部署图、类图和状态图来描述重要的并发架构和类关系。尽管 UML 的知识不是必需的，但熟悉它会对工作和学习有所帮助。

支持的开发环境

本书中的实例均使用 ISO 标准 C/C++开发。这意味着实例和程序能够在所有主要环境中进行编译。完整程序中仅使用 POSIX 兼容的操作系统调用或库，因此，这些程序能够移植到所有兼容 POSIX 的操作系统环境中。本书中的实例和程序在配有 UltraSparc T1 multiprocessor、Intel Core 2 Duo、IBM Cell Broadband Engine 和 AMD Dual Core Opteron 处理器的 SunFire 2000 上都进行了测试。

程序概要

本书中的多数完整程序均伴有一个程序概要(program profile)。概要包含实现细节，如必需的头文件、必需的库、编译指令和链接指令。概要还包括一个注释部分，包含执行程序时需要注意的任何特殊考虑。所有的代码仅用于说明目的。

测试及代码可靠性

尽管本书中的所有实例和应用程序均为了确保正确性而经过了测试，但我们并不保证书中包含的程序没有缺陷或错误，或者同任何特定标准或适销性相一致，或满足您的任何特定应用的要求。您不应依赖于它们来解决这样的问题，即问题的不正确的解决方案可能会导致人员伤害或财产损失。作者和出版商不对使用本书中的实例、程序或应用程序所导致的直接或间接损害承担任何责任。

约定

为了帮助您更好地了解本书的内容，我们在书中使用了如下的约定。

对于正文中的样式：

- 我们对新的术语和重要的词在引入它们时进行了强调。
- 我们采用类似如下的方式显示组合键：Ctrl+A。
- 我们以两种不同的方式显示代码：

```
We use a monofont type with no highlighting for most code examples.
```

```
We use gray highlighting to emphasize code that's particularly important
in the present context.
```

本书中既包含程序清单(code listing)，又包含代码示例(code example)。

- 程序清单是完整的、可执行的程序。如前所述，多数情况下，它们会伴有一个程序概要，它会告诉您程序编写时的环境，并给出编译指令和链接指令的描述，等等。
- 代码示例是一些片断，不加修改是不能够运行的。它们用来集中展示某些内容如何被调用或使用。

源代码

在您完成本书中的例子时，您可以选择手工输入所有代码或使用伴随本书的源代码文件。本书中所使用的所有源代码可以从 http://www.tupwk.com.cn/downpage 下载，也可以从 http://www.wrox.com 下载。访问到该网站之后，只要找到本书的书名(使用 Search 输入框或使用一个书名列表)，然后在本书的详情页面上单击 Download Code 链接来得到本书的所有源代码。

注意：

由于很多书的书名很类似，最简单的查找方式是通过 ISBN，本书的 ISBN 为 978-0-470-28962-4。

一旦下载了代码，只需要使用您最喜欢的压缩工具对它进行解压。或者，您可以转到 Wrox 的代码下载主页面，网址为 http://www.wrox.com/dynamic/books/download.aspx，查看本书及所有 Wrox 书籍的可用代码。

勘误表

我们尽全力确保正文或代码中没有错误。然而人无完人，错误总会发生。如果您在我们的书中发现了错误，例如拼写错误或错误的代码段，我们将会非常感激您的反馈。通过递交勘误，您可能会帮助另一名读者避免数小时的受挫，同时，也能帮我们提供质量更高的书籍。

为了找到本书的勘误页面，请访问 http://www.wrox.com 并通过 Search 输入框或根据书名列表找到本书的书名。然后，在本书的详情页面上，单击 Book Errata 链接。在这个页面上您可以看到所有已经为本书提交的并由 Wrox 编辑发布的勘误。

如果在 Book Errata 页上没有找到您所发现的错误，请将错误发送至 wkservice@vip.163.com。我们将会查看信息，如果情况属实，则会发布消息到本书的勘误页，并在本书的后续版本中做出修订。

p2p.wrox.com

如果您希望同作者和本书其他读者进行讨论，可以加入到 http://p2p.wrox.com 上的 P2P 论坛。该论坛是一个基于 Web 的系统，您可以在论坛中发布同 Wrox 书籍及相关技术有关的消息，并且可以同其他读者和技术用户交流。论坛提供了订阅特性，可以将论坛上发布的您所感兴趣的话题通过 E-mail 发送给您。论坛中有 Wrox 作者、编辑、其他行业专家以及读者。

在 http://p2p.wrox.com 上，您不仅可以找到许多帮助您阅读本书的不同的论坛，而且还可以开发自己的应用程序。要想加入论坛，应执行下列步骤：

(1) 进入 http://p2p.wrox.com 并单击 Register 链接。

(2) 阅读使用条款并单击 Agree 按钮。

(3) 填写加入论坛所要求的信息以及您希望提供的其他可选信息，然后单击 Submit 按钮。

(4) 您将会收到一封电子邮件，其中的内容描述了如何验证您的账号并完成加入过程。

注意：

即使不加入 P2P，您也可以阅读论坛中的消息，但是如果您希望发布自己的消息，则必须加入 P2P。

一旦加入 P2P 之后，您可以发布新的消息并回复其他用户发布的消息。您可以在任何时候阅读 Web 上的消息。如果您希望特定论坛的新的消息以电子邮件的形式发送给您，可单击论坛列表中论坛名字旁边的 Subscribe to this Forum 按钮。

要想知道更多关于如何使用 Wrox P2P 的信息，可以阅读 P2P FAQ 中关于论坛软件如何工作以及很多关于 P2P 和 Wrox 书籍的其他常见问题的答案。要想阅读 FAQ，可单击 P2P 页面中的 FAQ 链接。

目　　录

第1章　新的体系结构 ……………………… 1
1.1　什么是多核 ……………………… 1
1.2　多核体系结构 …………………… 2
1.3　软件开发人员眼中的多核体系结构 ……………………… 3
1.3.1　基本的处理器体系结构 ……… 4
1.3.2　CPU(指令集) …………………… 6
1.3.3　内存是关键 ……………………… 8
1.3.4　寄存器 …………………………… 10
1.3.5　cache ……………………………… 11
1.3.6　主存 ……………………………… 12
1.4　总线连接 ………………………… 13
1.5　从单核到多核 …………………… 13
1.5.1　多道程序设计和多处理 ……… 14
1.5.2　并行编程 ………………………… 14
1.5.3　多核应用程序的设计与实现 … 15
1.6　小结 ……………………………… 15
第2章　4种有影响的多核设计 ……… 17
2.1　AMD Multicore Opteron ………… 19
2.1.1　Opteron 的直连和 HyperTransport ……………… 19
2.1.2　系统请求接口和交叉开关 …… 20
2.1.3　Opteron 使用 NUMA 结构 …… 21
2.1.4　cache 以及多处理器 Opteron … 22
2.2　Sun UltraSparc T1 多处理器 …… 22
2.2.1　UltraSparc T1 内核 …………… 24
2.2.2　Cross Talk 与 Crossbar ……… 25
2.2.3　DDRAM 控制器和 L2 cache … 25
2.2.4　UltraSparc T1、Sun 和 GNU gcc 编译器 ………………… 25
2.3　IBM Cell Broadband Engine …… 25
2.3.1　CBE 与 Linux ………………… 26
2.3.2　CBE 内存模型 ………………… 27
2.3.3　对操作系统隐藏 ……………… 27
2.3.4　协处理器部件 ………………… 28
2.4　Intel Core 2 Duo 处理器 ……… 28
2.4.1　北桥和南桥 …………………… 29
2.4.2　Intel 的 PCI Express ………… 29
2.4.3　Core 2 Duo 的指令集 ………… 29
2.5　小结 ……………………………… 30
第3章　多核编程的挑战 ……………… 33
3.1　什么是顺序模型 ………………… 33
3.2　什么是并发 ……………………… 34
3.3　软件开发 ………………………… 35
3.3.1　挑战1：软件分解 …………… 38
3.3.2　挑战2：任务间通信 ………… 43
3.3.3　挑战3：多个任务或 agent 对数据或资源的并发访问 …… 47
3.3.4　挑战4：识别并发执行的任务之间的关系 ………………… 51
3.3.5　挑战5：控制任务之间的资源争夺 ……………………… 53
3.3.6　挑战6：需要多少个进程或线程 ……………………… 53
3.3.7　挑战7和挑战8：寻找可靠的、可重现的调试和测试 …… 54
3.3.8　挑战9：与拥有多进程组件的设计的相关人员进行沟通 …… 55
3.3.9　挑战10：在 C++中实现多处理和多线程 ……………… 56
3.4　C++开发人员必须学习新的库 … 56
3.5　处理器架构的挑战 ……………… 57

3.6 小结 ······57

第4章 操作系统的任务 ······59

4.1 操作系统扮演什么角色 ······59

4.1.1 提供一致的接口 ······59

4.1.2 管理硬件资源和其他应用软件 ······60

4.1.3 开发人员与操作系统的交互 ······60

4.1.4 操作系统的核心服务 ······63

4.1.5 应用程序员的接口 ······66

程序概要 4-1 ······70

程序概要 4-2 ······74

4.2 分解以及操作系统的任务 ······75

4.3 隐藏操作系统的任务 ······77

4.3.1 利用 C++抽象和封装的能力 ······77

4.3.2 POSIX API 的接口类 ······78

4.4 小结 ······85

第5章 进程、C++接口类和谓词 ······87

5.1 多核是指多处理器 ······87

5.2 什么是进程 ······88

5.3 为什么是进程而不是线程 ······88

5.4 使用 posix_spawn() ······90

5.4.1 file_actions 参数 ······91

5.4.2 attrp 参数 ······92

5.4.3 简单的 posix_spawn()示例 ······94

5.4.4 使用 posix_spawn 的 guess_it ······95

5.5 哪个是父进程，哪个是子进程 ······99

5.6 对进程的详细讨论 ······99

5.6.1 进程控制块 ······100

5.6.2 进程的剖析 ······101

5.6.3 进程状态 ······103

5.6.4 进程是如何被调度的 ······105

5.7 使用 ps 实用工具监视进程 ······107

5.8 设置和获得进程优先级 ······110

5.9 什么是上下文切换 ······112

5.10 进程创建中的活动 ······112

5.10.1 使用 fork()函数调用 ······113

5.10.2 使用 exec()系统调用系列 ······113

5.11 进程环境变量的使用 ······116

5.12 使用 system()生成新的进程 ······117

5.13 删除进程 ······118

5.13.1 调用 exit()和 abort() ······118

5.13.2 kill()函数 ······119

5.14 进程资源 ······119

5.14.1 资源的类型 ······120

5.14.2 设置资源限制的 POSIX 函数 ······121

5.15 异步进程和同步进程 ······124

5.16 wait()函数调用 ······125

5.17 谓词、进程和接口类 ······127

5.18 小结 ······131

第6章 多线程 ······133

6.1 什么是线程 ······133

6.1.1 用户级线程和内核级线程 ······134

6.1.2 线程上下文 ······136

6.1.3 硬件线程和软件线程 ······138

6.1.4 线程资源 ······138

6.2 线程和进程的比较 ······139

6.2.1 上下文切换 ······139

6.2.2 吞吐量 ······139

6.2.3 实体间的通信 ······139

6.2.4 破坏进程数据 ······140

6.2.5 删除整个进程 ······140

6.2.6 被其他程序重用 ······140

6.2.7 线程与进程的关键类似和差别 ······140

6.3 设置线程属性 ······142

6.4 线程的结构 ······143

6.4.1 线程状态 ······144

6.4.2 调度和线程竞争范围 ······145

6.4.3 调度策略和优先级 ······147

6.4.4 调度分配域 ······148

6.5 简单的线程程序 ······148

6.6　创建线程……150
6.6.1　向线程传递参数……151
6.6.2　结合线程……153
6.6.3　获得线程 id……154
6.6.4　使用 pthread 属性对象……155
6.7　管理线程……159
6.7.1　终止线程……159
6.7.2　管理线程的栈……168
6.7.3　设置线程调度和优先级……171
6.7.4　设置线程的竞争范围……175
6.7.5　使用 sysconf( )……175
6.7.6　线程安全和库……177
6.8　扩展线程接口类……179
6.9　小结……187
第 7 章　并发任务的通信和同步……189
7.1　通信和同步……189
7.1.1　依赖关系……190
7.1.2　对任务依赖进行计数……193
7.1.3　什么是进程间通信……195
7.1.4　什么是线程间通信……215
7.2　对并发进行同步……223
7.2.1　同步的类型……224
7.2.2　同步对数据的访问……224
7.2.3　同步机制……230
7.3　线程策略方法……250
7.3.1　委托模型……251
7.3.2　对等模型……253
7.3.3　生产者-消费者模型……254
7.3.4　流水线模型……255
7.3.5　用于线程的 SPMD 和 MPMD……256
7.4　工作的分解和封装……258
7.4.1　问题陈述……258
7.4.2　策略……258
7.4.3　观察……259
7.4.4　问题和解决方案……259
7.4.5　流水线的简单 agent 模型实例……260
7.5　小结……264
第 8 章　PADL 和 PBS：应用程序设计方法……265
8.1　为大规模多核处理器设计应用程序……265
8.2　什么是 PADL……268
8.2.1　第 5 层：应用程序架构选择……271
8.2.2　第 4 层：PADL 中的并发模型……281
8.2.3　第 3 层：PADL 的实现模型……284
8.3　谓词分解结构……306
8.3.1　示例：Guess-My-Code 游戏的 PBS……307
8.3.2　将 PBS、PADL 和 SDLC 联系起来……307
8.3.3　对 PBS 进行编码……308
8.4　小结……308
第 9 章　对要求并发的软件系统进行建模……311
9.1　统一建模语言……311
9.2　对系统的结构进行建模……313
9.2.1　类模型……313
9.2.2　类的可视化……315
9.2.3　对属性和服务进行排序……320
9.2.4　类的实例的可视化……322
9.2.5　模板类的可视化……324
9.2.6　显示类与对象的关系……325
9.2.7　接口类的可视化……329
9.2.8　交互式对象的组织……331
9.3　UML 与并发行为……332
9.3.1　协作对象……332
9.3.2　使用进程与线程的多任务与多线程……334
9.3.3　对象间的消息序列……335
9.3.4　对象的活动……337
9.3.5　状态机……339

9.4 整个系统的可视化······344
9.5 小结······345

第 10 章 并行程序的测试和逻辑容错······347

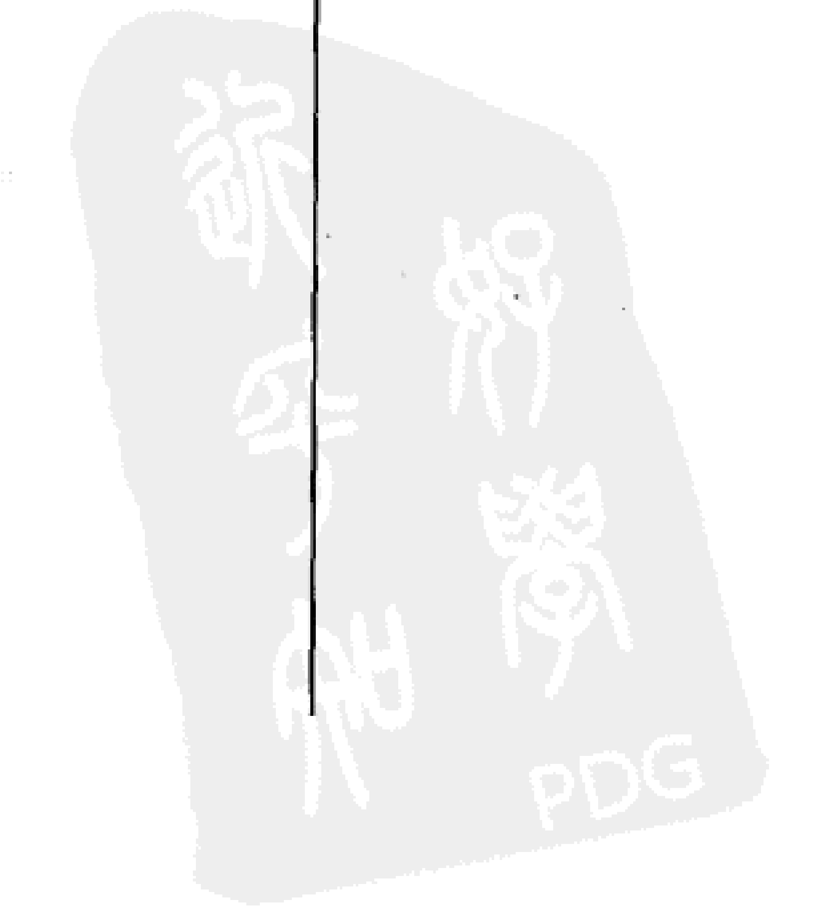
10.1 能否跳过测试······347
10.2 测试中必须检查的 5 个并发挑战······348
10.3 失效：缺陷与故障导致的结果······350
10.3.1 基本的测试类型······350
10.3.2 缺陷排除与缺陷存活······351
10.4 如何对并行程序实现缺陷排除······351
10.4.1 问题陈述······352
10.4.2 简单策略和粗解决方案模型······352
10.4.3 使用 PADL 第 5 层的修正的解决方案模型······352
10.4.4 agent 解决方案模型的 PBS······353
10.5 什么是标准软件工程测试······357
10.5.1 软件验证与确认······357
10.5.2 代码不能正常工作该怎么办······358
10.5.3 什么是逻辑容错······362
10.5.4 谓词异常和可能世界······367
10.5.5 什么是模型检测······368
10.6 小结······368

附录 A 并发设计使用的 UML······371

附录 B 并发模型······379

附录 C 线程管理的 POSIX 标准······393

附录 D 进程管理的 POSIX 标准······535



第1章 新的体系结构

在台式计算机的微处理器设计方面的最新进展包括将多个处理器放置到一个计算机芯片上。这些多核设计完全取代了作为台式计算机的基础的单核设计。IBM、Sun、Intel和 AMD 都已经将他们的芯片流水线从生产单核处理器变为生产多核处理器。这已经促使计算机供应商将他们的重心转移到销售拥有多核的台式计算机，如 Dell、HP 和 Apple。在这个新的领域的市场份额的竞争中，每个计算机芯片生产商都在挑战着能够经济地放置到一个芯片上的内核数目的极限。所有这些竞争使得消费者拥有了比以前更多的计算能力。主要的问题在于常规的台式计算机软件没有被设计为利用新的多核架构。实际上，为了能够从新的多核体系结构中得到任何真正的加速，将必须重新设计台式计算机软件。

设计和实现利用多核处理器的应用软件的技术，和在单核开发中使用的技术有着根本的区别。软件设计和开发的焦点将必须从顺序编程技术转变为并行和多线程编程技术。

现在，一般开发人员的工作站以及入门级的服务器都采用具有硬件级多线程、多处理和并行处理能力的多处理器。尽管顺序编程和单核应用开发仍将保留有一席之地，但是多核应用程序设计和开发的思想现在已经成为主流。

本章将会带您了解多核编程，介绍的内容有：

- 什么是多核？
- 有哪些多核体系结构以及它们之间有什么区别？
- 作为软件的设计人员和开发人员，当从顺序编程和单核应用开发转移到多核编程时，需要知道哪些知识？

1.1 什么是多核

多核是一种将多个处理器放置到一个计算机芯片上的架构设计。每个处理器被称作一个核。随着芯片容量的增加，在一个芯片上放置多个处理器变得可行。这些设计被称为芯片多处理器(Chip Multiprocessor，CMP)，是因为它们允许单芯片进行多处理。多核是 CMP 或单芯片多处理器的流行的名字。单芯片多处理的概念并不是新的，早在 20 世纪 90 年代

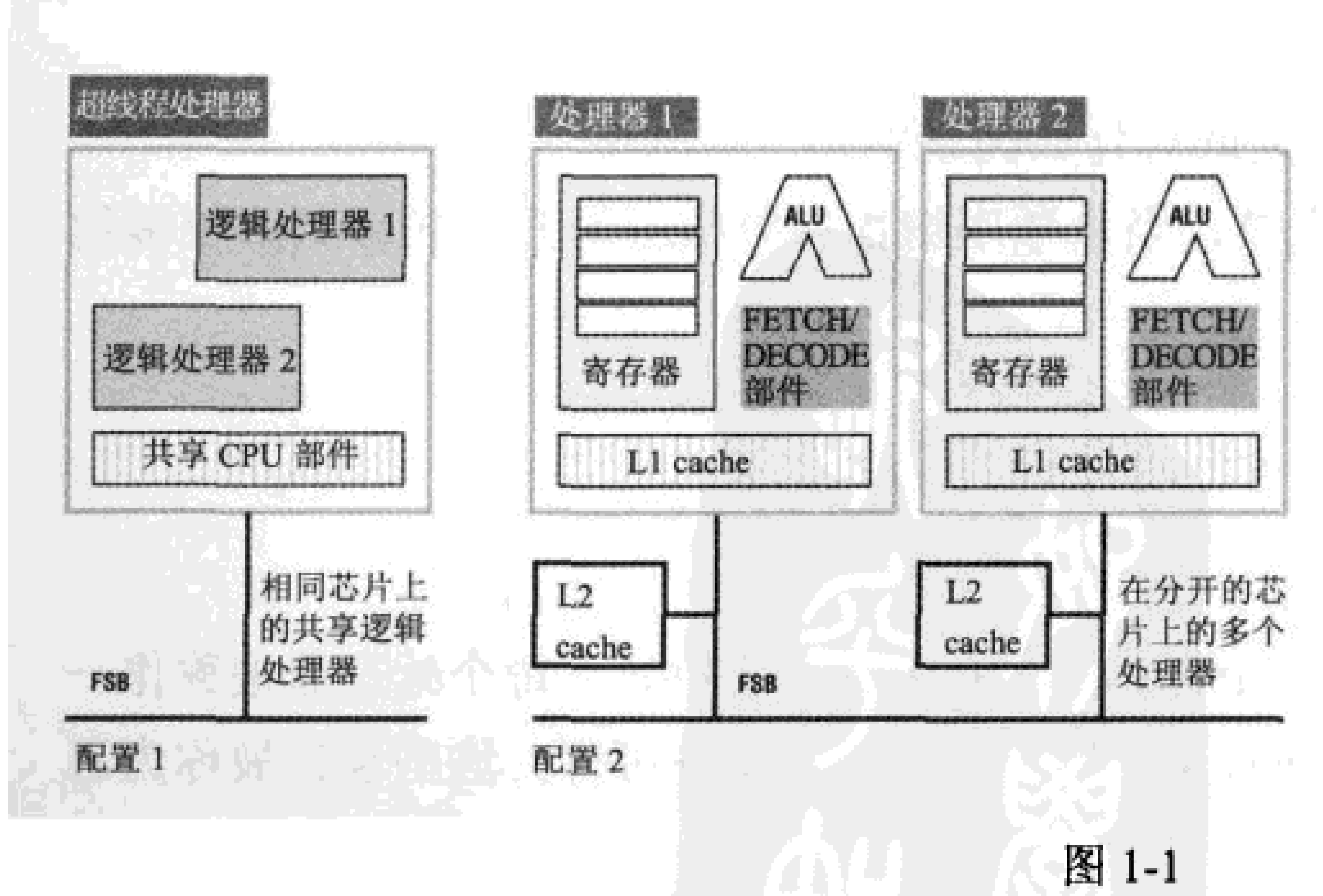
初期，芯片生产商就已经开始探索在单芯片上放置多个内核的想法。最近，CMP 已经成为改进总体系统性能的首选方法。这种方法与通过增加时钟频率或处理器速度来获得总体系统性能收益是截然不同的。增加时钟频率的方法已经开始在成本效益方面达到其极限了。较高的频率要求更多的能耗，使得系统制冷变得困难且代价高昂。这还会影响确定尺寸和封装(sizing and packaging)方面的考虑。所以，现在的做法是增加更多的处理器，而不是令处理器运行得更快以获得性能上的提升。由于这种方法实现简单，因此是更好的方法，所以该方法驱动了多核革命。当前，多核体系结构在增强总体系统性能方面成为了焦点。

对于熟悉多处理的软件开发人员，对多核开发应当也不陌生。从逻辑的观点，对分开封装的多个处理器进行编程与对包含在单独芯片上的单个封装上的多个处理器进行编程，两者之间并没有很大的区别。当然可能会存在性能差异，因为新的 CMP 利用了总线体系结构以及处理器之间连接等方面的进步。在某些环境下，这可能会使得原本为多个处理器编写的程序能够在 CMP 上更快地执行。除了潜在的性能收益，设计和实现都是非常类似的。在本书中，我们将讨论其中存在的微小差别。对于只熟悉顺序编程和单核开发的开发人员，多核方法提供了很多新的软件开发范型。

1.2 多核体系结构

CMP 有多种形式：两个处理器(双核)、四个处理器(四核)和八个处理器(八核)结构。有些结构是多线程，有些结构不是。在新的 CMP 中，高速缓冲存储器(cache)和内存的处理方式有着几种变体，在不同的实现中，处理器与处理器之间的通信方法也不同。来自各大主要芯片生产商的 CMP 实现中，在处理 I/O 总线和前端总线(Front Side Bus，FSB)上均不相同。

如果严格地从逻辑视图上查看被设计为利用多核体系结构的应用程序时，并看不出大多数区别。图 1-1 示范了支持多处理的 3 种常见配置。



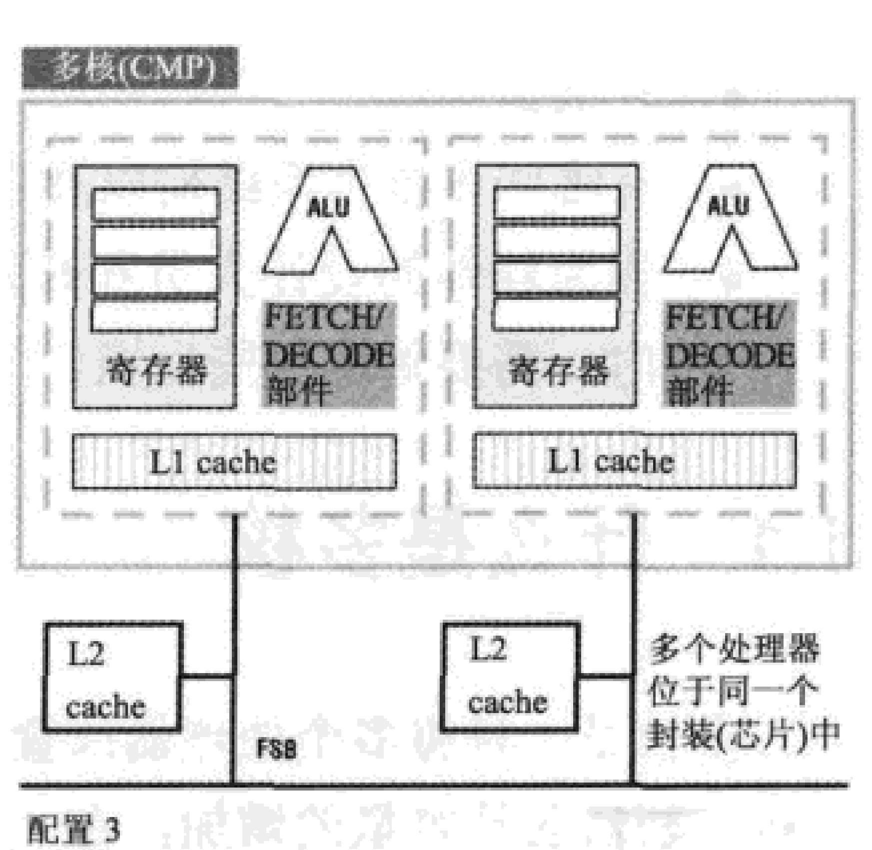


图 1-1

- 图 1-1 中的配置 1 使用超线程(hyperthreading)。与 CMP 类似，使用超线程的处理器允许在一个芯片上执行两个或多个线程。然而，在超线程的封装中，多个处理器指的是逻辑上的，而不是物理上的。这些封装中有着两套硬件，但是不足以构成一个单独的物理处理器。这样，超线程允许处理器在将自身呈现给操作系统时，好像是完备的多处理器，但实际上只是在一个处理器上运行多个线程。
- 图 1-1 中的配置 2 是经典的多处理器。在配置 2 中，每个处理器位于一个独立的芯片上，而且有着自己的硬件。
- 配置 3 代表了当前多处理器的发展趋势，它在一个芯片上提供完整的多个处理器。

如同您将在第 2 章所看到的，一些多核设计在核的内部支持超线程。例如，一个使用了超线程技术的双核处理器可以将自己作为四核处理器呈现给操作系统。

混合型多核体系结构

混合型多核体系结构(hybrid multicore architecture)在一个封装中混合了多种处理器类型和/或线程模式。这样能够通过将多种独特的性能合并到一个功能内核中，提供有效的代码优化和特化(specialization)的方法。混合型多核架构的最常见示例是 IBM 的 Cell broadband engine(Cell)。我们将在下一章中介绍 Cell 的体系结构。

需要注意的是每种配置是作为由两个或更多个能够并发执行多个任务的一组逻辑处理器来呈现给开发人员的。对系统程序员、内核程序员和应用程序开发人员的挑战是需要了解何时以及如何利用这一点。

1.3　软件开发人员眼中的多核体系结构

CMP 的低成本和广泛可用性，使得一般的软件开发人员能够进行各种级别的并行处理。并行处理不再是超级计算机或集群的专属领域。基本的开发工作站和入门级服务器现在都具有软件级和硬件级的并行处理能力。这意味着程序员和软件开发人员可以无需牺牲设计或性能，即可根据需要部署利用多处理和多线程的应用。然而，需要注意的是，并非每个软件应用都需要多处理或多线程。实际上，一些软件解决方案和计算机算法最好使用顺序编程技术来实现。在某些情况下，在软件中引入并行编程技术的开销会使软件性能降级。并行性和多处理是需要一定成本的。如果软件中顺序地解决问题需要的工作量少于创建额外线程和进程的开销，或者少于协调并发执行的任务之间通信的工作，则应选择顺序的方法。

有时可以较容易地确定何时及何地应当使用并行性，因为软件解决方案本身可能会要求支持并行性。例如在很多客户端—服务器配置中，很显然是需要并行性的。可能有一个服务器，例如数据库，还有很多可以同时对数据库发起请求的客户端。在多数情况下，您不希望一个客户端被要求等待，直到另外一个客户端的请求被满足。可接受的解决方案允许软件并发地处理客户端的请求。另一方面，有时候可能会在不需要并行性时面对并行性

的诱惑。例如，您可能会倾向于相信在文本中进行并行关键字搜索理所当然地比顺序搜索快，但是这依赖于需要搜索的文本的规模，同时还依赖于启动多个并行搜索 agent 所需要的时间和开销数量。设计决策者若赞成使用并发的解决方案，则必须考虑盈亏临界点和问题规模。在多数情况下，软件设计和软件实现是分开进行的，而且很多时候是由不同的小组来执行的。但是当主要的系统需求是软件加速或性能优化时，软件设计小组必须至少清楚软件实现的选择，而软件实现选择必须知道潜在的目标平台。

在本书中，目标平台是多核平台。为了充分利用多核平台，您需要理解做些什么工作才能获得 CMP 的性能。您需要理解 CMP 中的哪些部分是可以控制的。您将看到可以通过编译器、操作系统调用/库、语言特性、应用程序级库来访问 CMP。但首先，为了理解如何处理 CMP 访问，需要对处理器体系结构有基本的理解。

1.3.1 基本的处理器体系结构

您可以访问和影响的部件包括寄存器、主存储器、虚拟内存、指令集使用以及目标代码优化。在试图与多处理器体系结构打交道之前，理解在单处理器架构中可以影响哪些部件非常重要。图 1-2 给出了简化的处理器架构和内存部件的逻辑概览。

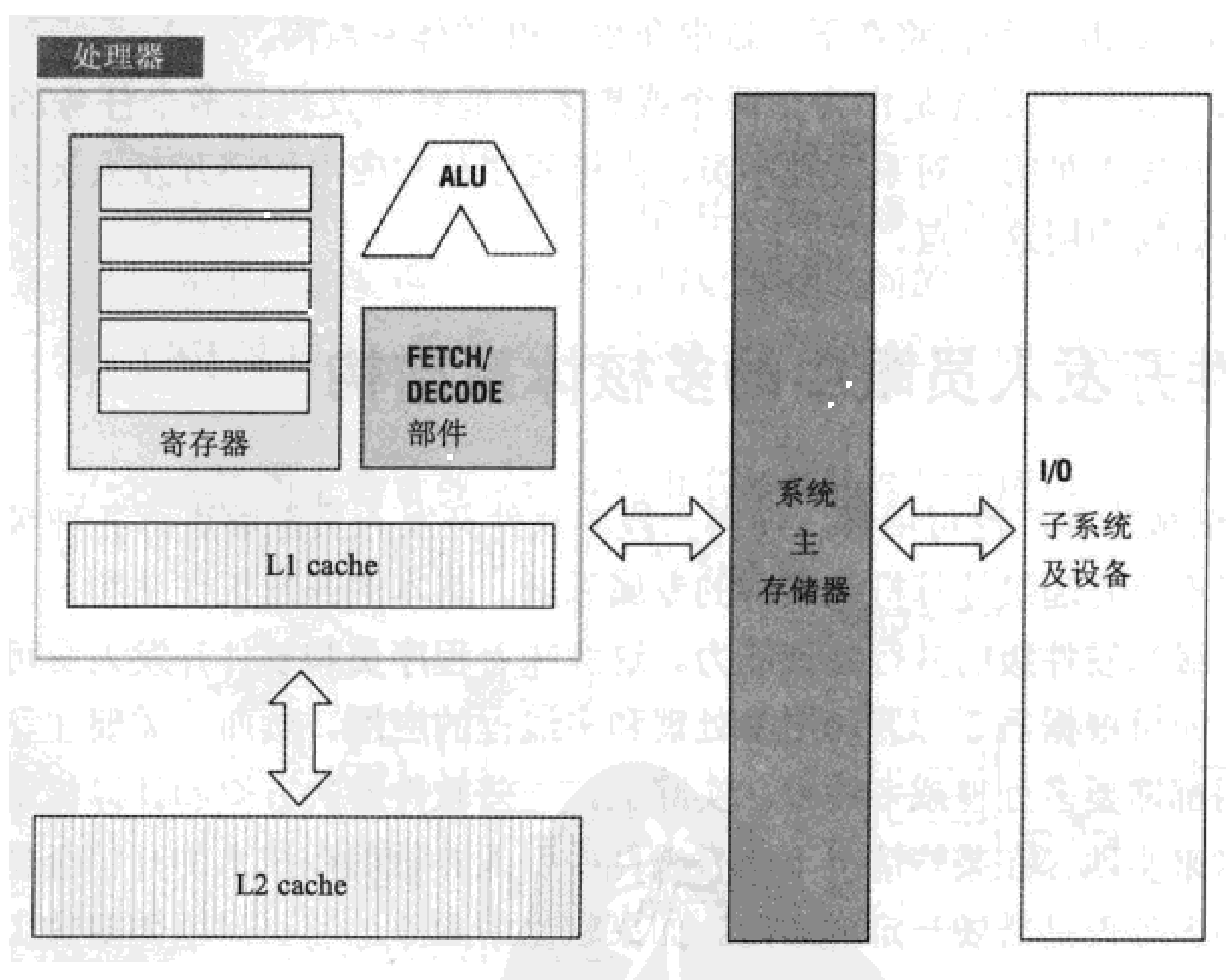


图 1-2

处理器架构有很多变体，图 1-2 只是一个逻辑概览。它说明了您可以使用的主要处理器部件。尽管这个级别的细节和这些部件对特定类型的应用程序开发经常是透明的，但是它们在自底向上多核编程和以加速和性能优化为主要目的的软件开发中都发挥着核心作用。与处理器的主要接口是编译器。操作系统是二级接口。

注意：

在本书中，我们将使用 C++编译器来生成目标代码。并行编程可用于使用多种方法的所有类型的应用程序，从低级到高级，从面向对象到结构化应用程序。C++支持多范型编程方法，因为其拥有较强灵活性，所以我们会选择使用它。

表 1-1 给出了编译器与 CPU 和指令集交互的类别清单。包括浮点类别、寄存器操纵类别和内存模型类别。

表 1-1

编译器开关选项	描　　述	使用的示例
Vectorization	这个选项将激活 vectorizer，它是编译器中的一个组件，自动在 MMX 寄存器中使用单指令多数据 (Single Instruction Multiple Data，SIMD)指令以及所有 SSE 指令集	**-x**　　**-ax** 激活 vectorizer
Auto parallelization	这个选项识别包含并行性的循环结构，然后(如果可能)安全生成并行执行的多线程等价体	**-parallel** 触发自动并行化

Parallelization with OpenMP	使用这个选项，编译器基于程序员在源码中加入的 OpenMP 指示来生成多线程代码	**#pragma omp parallel**
		{
		　　#pragma omp for
		　　//your code
		}

Fast	这个选项用于检测不兼容的处理器，在执行中生成错误消息	**-O1** 为代码规模和代码局部性进行优化，并禁用循环展开、软件流水和全局代码调度 **-O2** 默认值，将软件流水置为 ON
Floating point	允许编译器影响对浮点指令的选择和使用的一组开关	**-fschedule-insns** 告诉编译器可以发送其他指令，直到要求一个浮点指令的结果为止 **-float-store** 告诉编译器在生成目标代码时不使用在寄存器中存放浮点变量的指令

(续表)

编译器开关选项	描　述	使用实例
Loop unrolling	这个选项用于激活循环展开；它只应用于编译器被确定为应当被展开的循环；如果将 n 省略，则由编译器决定是否进行展开	**-unroll<n>** 激活循环展开，<n>设置循环展开的最大次数 **n=0** 禁用循环展开，仅为 64 位架构下的容许值
Memory bandwidth	这个选项用于激活或禁用对处理器使用的内存带宽的控制；如果禁用，则带宽会在多个线程间完全共享；可以和 auto parallelization 选项一同使用；这个选项仅用于 64 位架构	**-opt-mem-bandwidth<n>** **n=2** 为并行代码(如 pthreads 和 MPI 代码)激活编译器优化 **n=1** 为编译器生成的多线程代码激活编译器优化
Code generation	使用这个选项代码，为特定架构或处理器进行代码优化；如果有性能收益，编译器生成多条、处理器特定的代码路径；用于 32 位及 64 位架构	**-ax<processor>** 为指定处理器生成优化代码 **-axS** 使用 SIMD Extensions 4(SSE4)向量编译器和媒体加速器指令生成专门的代码路径
Thread checking	这个选项激活使用线程的应用程序或程序中的线程分析，只能和 Intel 的 Thread Checker 工具一同使用	**-tcheck** 激活使用线程的应用程序或程序的分析
Thread library	这个选项使得编译器包含来自 Thread Library 的代码；程序员需要在源代码中包含 API 调用	**-pthread** 针对多线程支持使用 pthread 库

1.3.2　CPU(指令集)

CPU 有着它识别并执行的原生指令集(native instruction set)。C++编译器的工作是将 C++程序代码转换到目标平台的原生指令集。编译器对 C++进行转换并生成一个由目标处理器的原生指令组成的目标文件。图 1-3 显示了基本编译过程的缩略图。

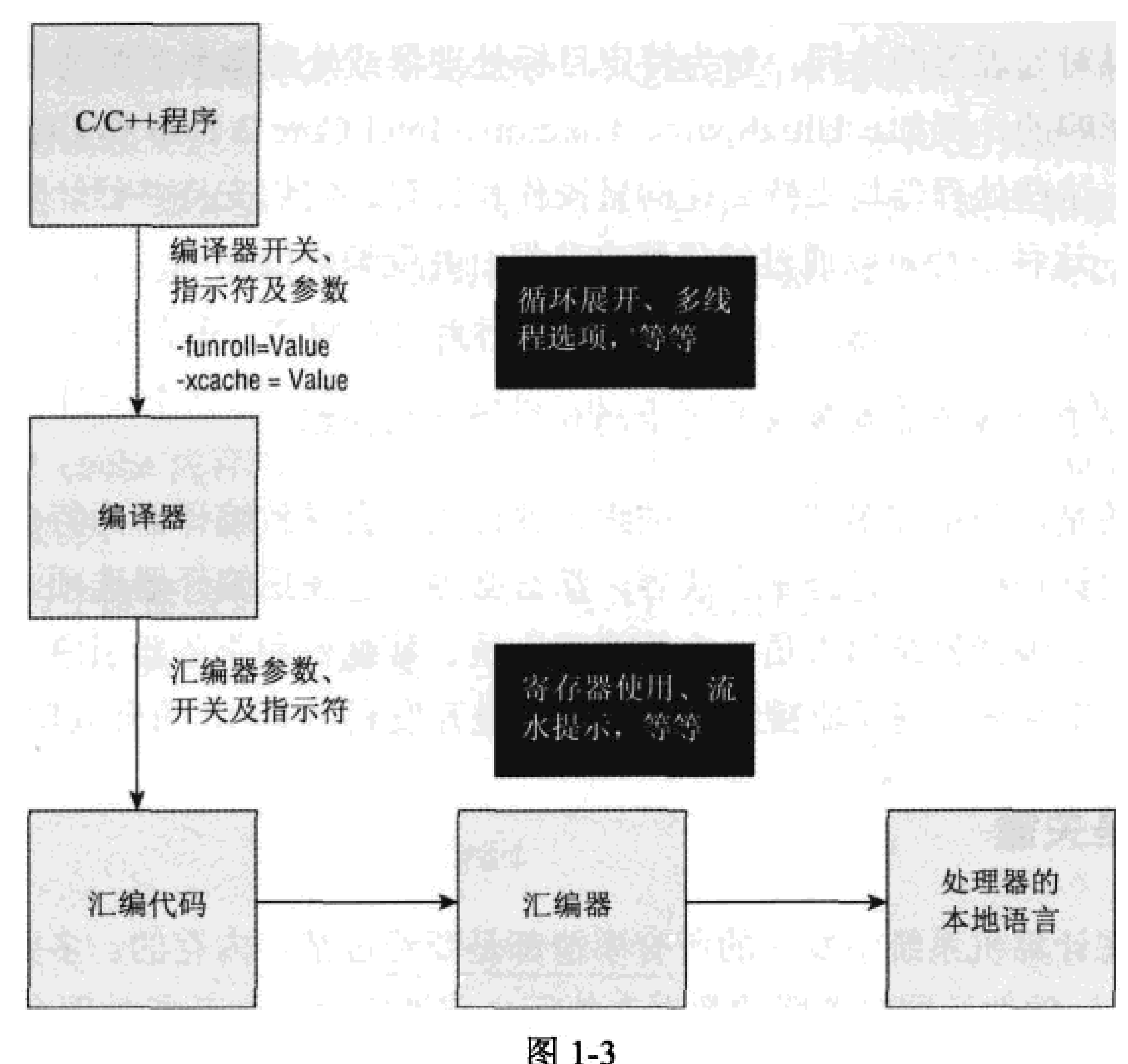


图 1-3

在将 C++代码转换为目标 CPU 本地语言的过程中，编译器可以选择如何生成目标代码。编译器可用来帮助确定寄存器如何使用或是否执行循环展开。可以通过对编译器选项的设置来决定是否生成 16 位、32 位或 64 位目标代码。编译器可用于选择内存模型。编译器可以提供代码提示来声明提供多少 L1(level 1) cache 或 L2(level 2) cache。注意在表 1-1 中的浮点操作类别中，该类别中的开关允许编译器影响对浮点指令的选择。例如，GNU gcc 编译器有- -float-store 开关。这个开关告诉编译器在生成目标代码时，不应当使用将在寄存器中存储浮点变量的指令。Sun C++编译器有-fma 开关，这个开关允许自动生成浮点和 multi-add 指令。-fma=none 禁用了这些指令的生成。-fma=fused 开关允许编译器通过使用浮点、fused 和 multiply=add 指令来尝试改进代码性能。在上述两种情况下，开关都作为选项提供给编译器：

```
gcc -ffloat-store my_program.cc
```

或

```
CC -fma=used my_program.cc
```

其他开关影响 cache 使用。例如 Sun C++编译器有-xcache=c，定义了被优化器使用的 cache 属性。GNU gcc 编译器有-Funroll -loops，指定循环如何展开。GNU gcc 编译器的 -pthread 开关开启了对使用 pthread 的多线程的支持。编译器甚至还有选项可用来设置典型内存引用间隔，使用的是-mmemory-latency=time 开关。实际上，对于图 1-2 中的任何部件，均有编译器选项和开关可影响它们的使用。

编译器提供对处理器的访问，对为特定目标处理器或处理器系列编写多核应用程序的开发人员是有影响的。例如，UltraSparc、Opteron、Intel Core 2 Duo 和 Cell 处理器都是常用的多核配置。这些处理器均支持高速向量操作和计算。它们支持并行计算的单指令多数据(SIMD)模型。这种支持可以通过编译器来获得，并受编译器影响。

注意：
第 4 章包含了对编译器在多核开发中的作用的详细介绍。

值得注意的是，使用过多这种类型的编译器选项，会导致编译器为特定处理器进行代码优化。如果设计目标之一是跨平台兼容，那么必须小心使用编译器选项。对于系统程序员、库制作人员、编译器编写人员、内核开发人员、数据库和服务器引擎开发人员，对基本处理器架构、指令集和编译器接口的基本理解是开发利用 CMP 的有效软件的先决条件。

1.3.3 内存是关键

事实上，在计算机系统中发生的所有事情都是要经过某种内存的。多数事情需要经过很多的内存级别。软件及同它关联的数据在执行之前通常存储在某些外部介质中(通常为硬盘、CD-ROM、DVD 等)。例如，假定您有一个非常重要且非常长的数字列表存放在一个光盘中，而且需要将这些数加到一起。同时假定用来对这个非常长的数字列表进行累加的程序也保存在光盘中。图 1-4 说明了程序和数据如何流向处理器。

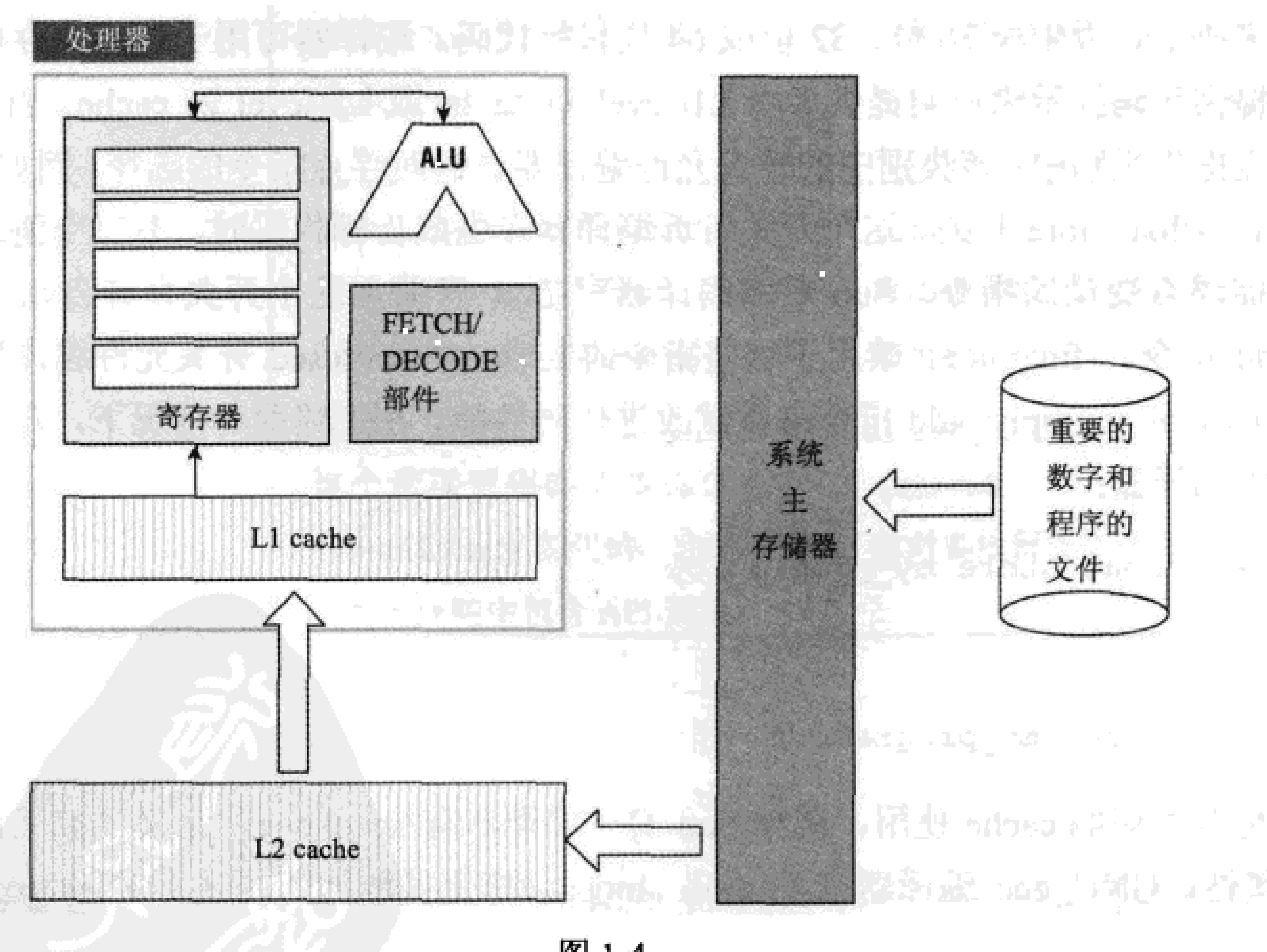


图 1-4

在不同类型的内存中，您必须记住的是典型的 CPU 仅对保存在其寄存器中的数据进行操作，它没有直接访问存储在其他位置的数据或程序的能力。图 1-4 显示了 ALU 对寄存器

的读取和写入，这是正常的状况。指令集命令(处理器的本地语言)被设计为主要针对 CPU 寄存器中的数据或指令进行工作。为了将您的重要的数字列表和程序取到处理器，必须从光盘中提取软件和数据并加载到主存储器中。从主存储器开始，软件和数据被传递到 L2 cache，然后到 L1 cache，接下来传递到指令和数据寄存器，这样 CPU 可以进行它的工作。值得注意的是在每个阶段，存储器的执行速度是不同的。二级存储器，如 CD-ROM、DVD 和硬盘的速度要低于主随机存取内存(random access memory，RAM)。RAM 的速度慢于 L2 cache 内存。L2 cache 内存慢于 L1 cache 内存。处理器中的寄存器是您能够直接打交道的速度最快的存储器。

各种类型的存储器除了在速度方面有区别外，规模也是一个因素。图 1-5 给出了分级存储器体系(memory hierarchy)的概览。

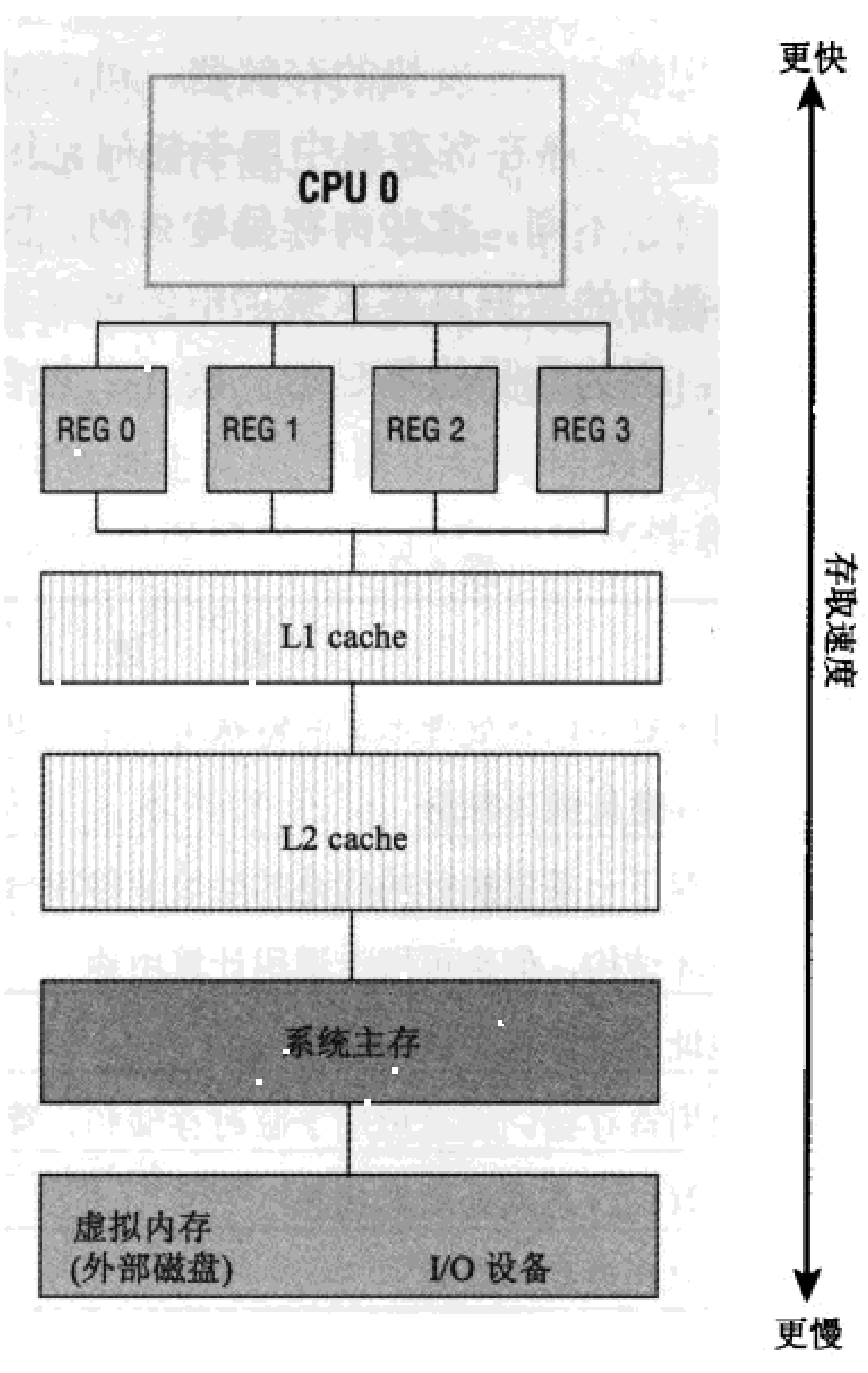


图 1-5

寄存器的速度最快，但是容量最小。例如，一台 64 位计算机通常有一组寄存器，每个寄存器能够保存最多 64 比特。在某些实例中，寄存器可以成对使用，允许保存 128 比特。在容量方面，紧随寄存器之后的是 L1 cache 和 L2 cache。L2 cache 目前是以 MB 为单位进行度量的。从 L2 cache 到系统主存，在最大容量方面有一个巨大的提升，系统主存的容量当前是以 GB 为单位进行度量的。除了各种类型的存储器的速度以及容量之外，还有一个因素是各类存储器之间的连接。这些连接被证明对总体系统性能有着重要的影响。在二级存储器中保存的数据和指令必须通过 I/O 通道或总线才能到达 RAM。一旦到达 RAM，数

据或指令通常经由系统总线到达 L1 cache。I/O 总线以及系统总线的速度和容量在多处理器环境中可能会成为瓶颈。随着芯片上内核的数目的增加，总线架构以及数据通路的性能带来的影响就更加明显。

本章稍后部分将讨论总线连接，首先我们来了解分级存储器体系以及它在多核应用程序开发中扮演的角色。要记住，就像您可以使用编译器对指令集选择的影响那样，您也可以使用编译器来操纵寄存器使用和 RAM 对象布局、提供 cache 规模提示，等等。您可以使用更多的 C++语言元素来指定寄存器使用、RAM 和 I/O。这样，在您详细了解多处理和多线程之前，必须对处理器处理的分级存储器体系有基本的了解。

1.3.4 寄存器

寄存器是用于特殊目的的、规模小但速度快的存储器，可以被内核直接存取。寄存器是易失的(volatile)。当程序退出时，程序在寄存器中用于任何意图和目的的任何数据或指令都会消失。与交换内存、虚拟内存不同，这些内存是持久的，因为保存在某种二级存储器中，而寄存器是暂时的。寄存器中的数据只在系统加电或程序运行期间保持。在通用计算机中，寄存器位于处理器内部，因为几乎为零延迟。表 1-2 包含了多数通用处理器中的寄存器的常见类型。

表 1-2

寄　存　器	描　　述
Index	在通用计算中以及处理地址的特殊用法中使用
Segment	用于保存地址的段部分
IP	用于保存下一条要执行的指令的地址偏移部分
Counter	用于循环结构，但也可用于通用计算用途
Base	用于地址的计算和存放
Data	用作通用寄存器，而且可用于临时存储和计算
Flag	显示计算机状态或处理器状态
Floating point	用于计算和移动浮点数

多数 C/C++编译器有着可以影响寄存器使用的开关。除了能够用于影响寄存器使用的编译器选项外，C++还有 asm{ }指示符，它允许在 C++的过程或函数中写入汇编语言，例如：

```
void my_fast_calculation(void)
{
  ...
    asm{
            ...
            mov 2 , %r3
```

```
            inc(%r3)
            ...
      }
      ...
}
```

my_fast_calculation( )将 2 加载到 UltraSparc 处理器的%r3 通用寄存器中。尽管对于 C++，cache 不是轻易可见的，但是寄存器和 RAM 是可见的。根据开发的多处理器软件的类型，无论是通过编译器还是通过 C++ asm{ }指示符，对寄存器的操纵是必要的。

1.3.5 cache

cache 是位于处理器和主系统内存(RAM)之间的存储器。尽管 cache 不像寄存器那样快，但是仍要快于 RAM。它的容量大于寄存器，但是小于主存。cache 增加了有效的存储器传递速度，因此提高了处理器总体性能。cache 用于保存处理器最近使用的数据或指令的副本。会从主存中获取小的内存块并保存到 cache 中，期望处理器会需要它们。程序具有时间局部性(temporal locality)和空间局部性(spatia! locality)的倾向。

- 时间局部性是重用最近访问的指令或数据的倾向。
- 空间局部性是访问物理上接近于最近访问项的指令或数据的倾向。

cache 的一项主要功能是利用程序的这种空间局部性和时间局部性的特性。cache 通常会被分为两个级别，即 level 1 和 level 2。

注意：

cache 的完整讨论超出了本书的范围。若需要了解关于 cache 的详细讨论，可参考[Hennessy,Patterson,2007]。

1. level 1 cache

level 1 cache 的规模较小，有时候只有 16KB。L1 cache 通常位于处理器内部，用于截获最近使用的指令或数据的字节。

2. level 2 cache

同 L1 cache 相比，level 2 cache 要大一些，但速度要慢些。当前，它位于主板上(处理器外部)，但是这一点正在逐渐变化。当前，L2 cache 通常以 MB 为度量单位。L2 cache 可以保存更大块的最近使用的指令、数据和邻近 L1 cache 保存内容的项。由于 L1 和 L2 快于通用 RAM，因此对程序接下来要做的事情猜测得越正确，则总体系统性能越好，因为正确的数据块将位于 L1 cache 或 L2 cache 中。这样就可以避免对 RAM 或虚拟内存乃至最坏情况下对外部存储器的访问。

3. 用于 cache 的编译器开关

除非是在进行内核开发、编译器开发或其他类型的底层系统编程，否则多数进行多核

应用程序开发的人员不会关心手工管理 cache。然而，在当前使用的多数主流编译器中，的确给出了编译器选项来提示可用的 L1 cache 或 L2 cache 的数量，或提示关于 L1 cache 或 L2 cache 的特性。例如，Sun C++编译器具有 xcache 开关。该开关的参考指南显示了它的语法和用途。

-xcache=c 定义了优化器可以使用的 cache 属性。它并不保证使用任何特定 cache 属性。尽管这个选项可以单独使用，但它通常会是-xtarget 选项的扩展部分，它的主要用途是用来覆盖-xtarget 选项提供的一个值。

-xcache=16/32/4:1024/32/1 指定了如下内容：

level 1 cache 有：	level 2 cache 有：
16KB	1024KB
32B 通路大小	32B 通路大小
4 路相联	直接映射

开发真正利用 CMP 的软件要求对目标处理器或处理器系列的指令集以及内存使用进行详细的考虑。这包括了解优化的机会，例如循环展开、高速向量处理、SIMD 处理、MP 编译器指示符和为某些值提供编译器提示，如 L1 cache 或 L2 cache 的大小。

1.3.6 主存

图 1-2 显示了寄存器、cache、ALU 和主存之间的相对关系。除了外部存储器(例如硬盘、CD-ROM、DVD 等)以外，RAM 是开发人员在工作时面对的最慢的内存。同时 RAM 物理上位于处理器的外部，数据通过总线传送到处理器，使得其速度更慢。另一方面，RAM 是对于多线程或多处理应用程序的软件开发人员而言最明显的部分。多数情况下处理器和任务间共享的数据保存在 RAM 中，每个处理器必须执行的指令在运行时将保存在 RAM 中，必须在多个处理器间同步的临界区(critical section)也主要保存在 RAM 中。如果有任务或处理器被锁住，通常是因为内存管理问题。几乎在任何情况下，处理器和任务或多个 agent 之间的通信，将通过在运行时驻留在 RAM 中的变量、消息队列、容器和互斥量等产生。在软件开发者的多核应用程序编程视图中，内存访问和管理是一个重要的元素。如同已经讨论过的图 1-2 中显示的其他逻辑部件那样，您有权使用影响应用程序如何处理内存的编译器开关。您所选择的内存模型非常重要。在 C++中通过 new( )操作符创建的对象会位于空闲存储区(堆)或虚拟内存(如果数据对象足够大)中。空闲存储区逻辑上位于 RAM 中，而虚拟内存是外部存储器的映射。

注意：

我们将在第 5 章中详细讨论进程或线程如何使用 RAM。

1.4　总线连接

通常，计算机中的子系统通过总线进行通信。总线是子系统之间的共享通信连接[Hennessy，Patterson，1996]。总线是计算机中的部件之间的通道或通路。传统上，总线分为 CPU-内存总线或 I/O 总线。基本的系统配置由两条主要的总线构成，即系统总线(也被称为前端总线，FSB)和 I/O 总线。如果系统有 cache，通常还会有后端总线(Back Side Bus，BSB)，用于连接处理器和 cache。图 1-6 显示了一个简化的处理器-总线的配置。

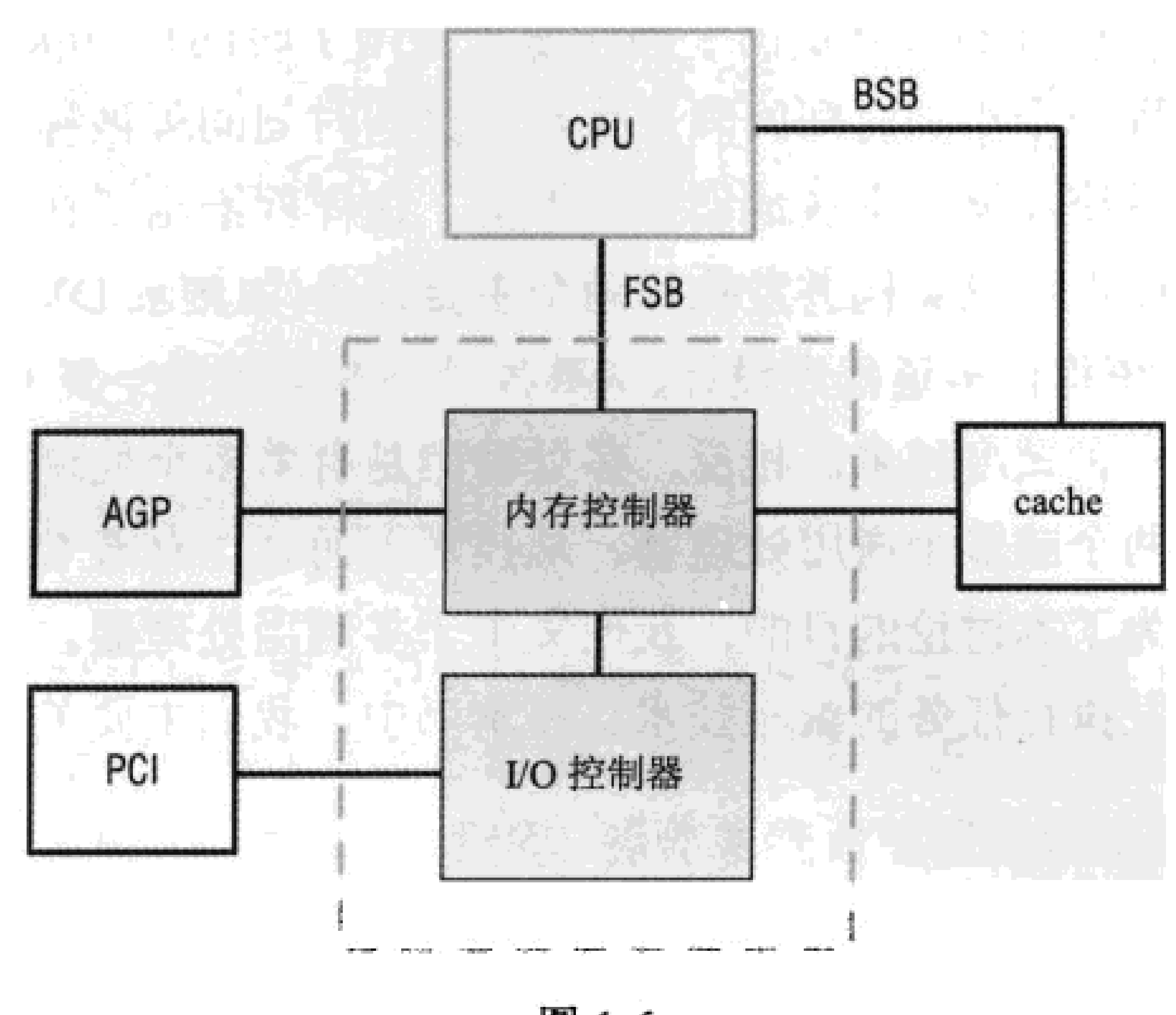


图 1-6

在图 1-6 中，FSB 用于 CPU 和内存之间的数据传输。FSB 是一条 CPU-内存总线。I/O 总线通常用于向其他外设发送信息。注意在图 1-6 中，BSB 用于在 CPU、cache 和主存之间移动数据。外围设备互连(Peripheral Component Interconnect，PCI)是 I/O 总线的例子。PCI 提供了到它所连接的设备的直接连接。然而，PCI 通常通过某种类型的桥接技术连接到 FSB。由于总线提供了 CPU、内存控制器、I/O 控制器、cache 和外设之间的通信通路，因此存在潜在的吞吐量瓶颈(throughput bottleneck)。多处理器配置会给 FSB 增加压力。目前的趋势是在一个芯片上增加更多的处理器，这会对基于总线的架构提出更高的通信要求。系统的性能受到 CPU、内存和其他系统外设之间所使用的总线的最大吞吐量的制约。如果总线的速度慢于 CPU 或内存，或者总线没有适当的容量、时序或同步，则总线会成为瓶颈，阻碍总体系统性能。

1.5　从单核到多核

在单核结构中，尽管很多当前的单核结构中包含特殊的图形处理部件、多媒体处理部件以及一些特殊的算术协同处理器，但您只需要关注一个(通用)处理器。但是即使使用单

核或单处理器计算机，多线程、并行编程、软件流水以及多道程序设计也都是可能的。所以本节将澄清一些基本事实，帮助您从单核编程迁移到多核编程。

1.5.1 多道程序设计和多处理

多道程序设计(multiprogramming)通常是在讨论操作系统时被提及，而不是在讨论应用程序时被提及。多道程序设计是一种调度技术，使得在任何时候都允许多个任务处于执行状态。在多道程序设计系统中，任务(或进程)共享系统资源，如主系统内存和处理器。在单核系统中，进程同时执行的假象是由于操作系统使用了时间片(time slice)技术。在时间片模式中，给每个进程一个小的时间间隔来执行。这些时间间隔被称作时间片，这些时间片非常短，而且操作系统能够非常快地切换上下文，这样就给出了在任意时刻执行多个进程或任务的假象。因此在单核架构并发执行两个主要任务(如刻录 DVD 的同时渲染计算机图像)的场景中，将系统称作多道程序设计。

多道程序设计是一种调度技术。相反，多处理器是有着多个处理器的计算机。在这种情况下，是特指有着两个或多个通用处理器。从技术上而言，有着 CPU 和 GPU 的计算机也是多处理器。但是为了本讨论的目的，我们集中于多通用处理器。这样，多处理是一种使用多个处理器来并发执行任务的编程技术。在本书中，我们主要关心属于并行编程范畴的技术。

1.5.2 并行编程

并行编程是使用并发执行的指令或任务来实现算法、计算机程序或计算机应用的科学技术。图 1-7 说明了每种类型的组成以及哪些是并行执行的。

图 1-7

图 1-7 中的并行算法可以并行执行一组指令。instruction 1 和 instruction 2 可以并发执

行。instruction 5 和 instruction 6 可以并发执行。在算法中，并行性发生在两条指令间。这和图 1-7 中的并行计算机程序不同，在那里，工作单元是过程、函数或线程。过程 A 和过程 B 可以同时执行。除了过程 A 和过程 B 之间的并发外，它们各自内部也可能存在并发。过程 A 的函数可能并行执行。因此对于包含并行性的计算机程序，工作单元要大于算法中的工作单元。

图 1-7 中的使用并行部件的计算机应用程序中具有最大的工作单元。任务 A 和任务 B 可能由很多过程、函数、对象等组成。当在应用程序级来查看并行编程时，所谈论的是更大的工作单元。除了任务，应用程序可能包含子系统，如后台网络组件或多媒体组件，这些组件相对于用户可执行的任务而言是同时在后台运行的。这里的关键点是图 1-7 中的每种结构都使用了并行编程，区别在于工作单元的大小，有时候被称作粒度。

注意：

我们将在第 4 章进一步讨论并行性的级别。

1.5.3　多核应用程序的设计与实现

多核应用程序的设计与实现使用并行编程技术来设计可以利用 CMP 的软件。设计过程将一些任务的工作指定为两个或多个线程、两个或多个进程，或线程与进程的组合。然后，该设计可以使用模板库、类库、线程库、操作系统调用或低级编程技术(例如软件流水、向量化等)来实现。本书将介绍多线程、多处理、进程间通信、线程间通信、同步、线程库、多线程类库和模板库的基础知识。低成本的 CMP 实现使得一般的开发人员也可以进行并行编程和多线程编程。本书主要介绍使用可以跨操作系统环境移植的多处理和多线程技术来开发多核应用。我们将仅使用符合操作系统 POSIX 标准的库以及符合 ISO 标准的 C++ 特性。

1.6　小结

本章涵盖了考虑开发多核应用所需要理解的关键概念。本章介绍的重要内容为：

- 多核芯片是有着两个或多个处理器的芯片。这种处理器配置称为 CMP。CMP 当前范围是从双核到八核。
- 混合型多核处理器可以包含不同类型的处理器。Cell broadband engine 是混合型多核处理器的实例。
- 根据开发者是系统程序员、内核程序员、库开发人员、服务器开发人员或应用程序开发人员，多核开发可以自底向上或自顶向下地实现。每组开发人员都会面对类似的问题，但是会从不同的角度来观察内核。
- 所有计划编写利用多处理器配置的软件的开发人员都应当熟悉目标平台的处理器架构。到多核处理器特定特性的主要接口为 C/C++编译器。为了更好地利用目标

处理器或目标处理器系列，开发者应当熟悉编译器、编译器的汇编器子部件和连接器的选项。二级接口包括操作系统调用、操作系统同步与通信组件。

- 并行编程是使用被设计为并发执行的指令或任务集来实现算法、计算机程序或计算机应用的科学技术。多核应用程序开发和设计是关于使用并行编程技术和工具来开发可以利用 CMP 架构的软件。

既然您已经了解了关于多核编程的基本思想和问题，第 2 章将介绍来自计算机工业领先位置的芯片生产商的 4 种多核设计，这些生产商为 AMD、Intel、IBM 和 Sun。我们将了解 Dual Core Opteron、Core 2 Duo、Cell Broadband Engine 架构和 UltraSparc T1 多处理器内核是如何实现 CMP 的。

4 种有影响的多核设计

在本章中，我们将仔细介绍来自计算机工业领先地位的芯片生产商的 4 种多核设计：

- AMD Multicore Opteron
- Sun UltraSparc T1
- IBM Cell Broadband Engine(CBE)
- Intel Core 2 Duo

这些生产商在实现 CMP 时采用的方法各不相同。他们的多核设计方法均得到了有效实现，而且每种设计同其他设计相比，各自有其优缺点。本书中的所有例子均将使用这些设计，这些程序实例已经在一个或多个上述多核处理器设计上进行了编译和执行。在本章中，我们将为您介绍每种设计，并在全书的适当位置介绍适合多核应用程序开发和设计的必要细节。

在很多大批量销售的软件应用程序中，硬件实现之间的区别已经被抽象技术掩盖了，因为软件的主要设计目标之一是尽可能使软件与尽可能多的硬件平台兼容，因此会有意识地避免特定平台的特性。在这些场景中，软件设计人员和开发人员适当地依赖于操作系统来隐藏应用程序可能遇到的平台差别。开发人员可以轻松愉快地进行开发，不需要担心硬件问题，这是一件好事。操作系统的主要工作之一就是隐藏和管理硬件细节。这种方法适用于全部的大规模销售应用程序或广大纵向市场应用程序。

然而，不是所有类别的软件开发人员都如此幸运。例如，那些开发高级事务处理数据库服务器、网络服务器、应用服务器、硬件密集型游戏引擎、编译器、操作系统内核、设备驱动、高性能科学建模和可视化软件的人员，都经常被迫利用平台特性以使他们的应用程序能够被终端用户所接受。对于这一类的开发人员，有效的软件开发的前提是熟悉特定处理器或处理器系列。表 2-1 列出了可能要求特定平台优化的应用程序的类型。

在表 2-1 中，我们还列出了涉及这些类型的应用的一些开发人员类型。希望对软件进行性能优化的系统程序员、图形程序员、应用程序开发人员和软件工程师都需要清楚目标平台的能力。当主要考虑跨平台兼容时，需要小心使用针对特定平台的优化。如果不需要考虑跨平台兼容，而且目的是在目标平台上获得最佳性能，则开发者对目标处理器或处理器系列了解得越多越好。

表 2-1

软件类型	开发者类型
高级事务处理软件服务器 ● 数据库 ● 财务事务处理服务器 ● 应用服务器等	● 软件架构师 ● 软件供货商 ● 软件生产商
内核	● 系统程序员
游戏引擎	● 系统程序员 ● 软件设计人员 ● 游戏开发人员 ● 图形程序员
设备驱动	● 系统程序员
大规模矩阵及向量计算	● 科学程序员 ● 数学家 ● 科学应用开发人员
编译器	● 系统程序员
数据库引擎	● 软件供货商 ● 数据库架构师
高清晰度计算机动画	● 图形程序员 ● 游戏开发人员
科学可视化建模	● 科学程序员

本书将介绍多处理器应用程序设计和实现的自顶向下和自底向上方法。为了利用自底向上的多处理器编程方法，要求对 CMP 架构、操作系统对多线程和多处理的支持、目标平台的 C/C++编译器有基本的了解。在第 4 章中，我们将详细介绍多核开发的操作系统支持和编译器支持。在本章中，我们将深入研究本章开始部分提到的 4 种有影响的多核设计。表 2-2 比较了 Opteron、UltraSparc T1、CBE 和 Core 2 Duo 处理器。

表 2-2

处理器名	超线程/SMT	使用 FSB	共享内存	Cache 2 位置	内核数
Opteron	否	否	否	主板	2
UltraSparc T1	是	否	否	die	8
CBE	是	否	是	die	9
Core 2 Duo	否	是	是	die	2

2.1 AMD Multicore Opteron

双核 Opteron 是 AMD 多核处理器产品线中的入门级产品。双核 Opteron 是最基本的配置，它采用了 AMD 实现多核架构的基本方法。Opteron 与 Intel 处理器系列在源码级和二进制代码级均兼容，也就是说，为 Intel 处理器编写的应用程序可以在 Opteron 上编译并执行。图 2-1 显示了双核 Opteron 的简化框图。

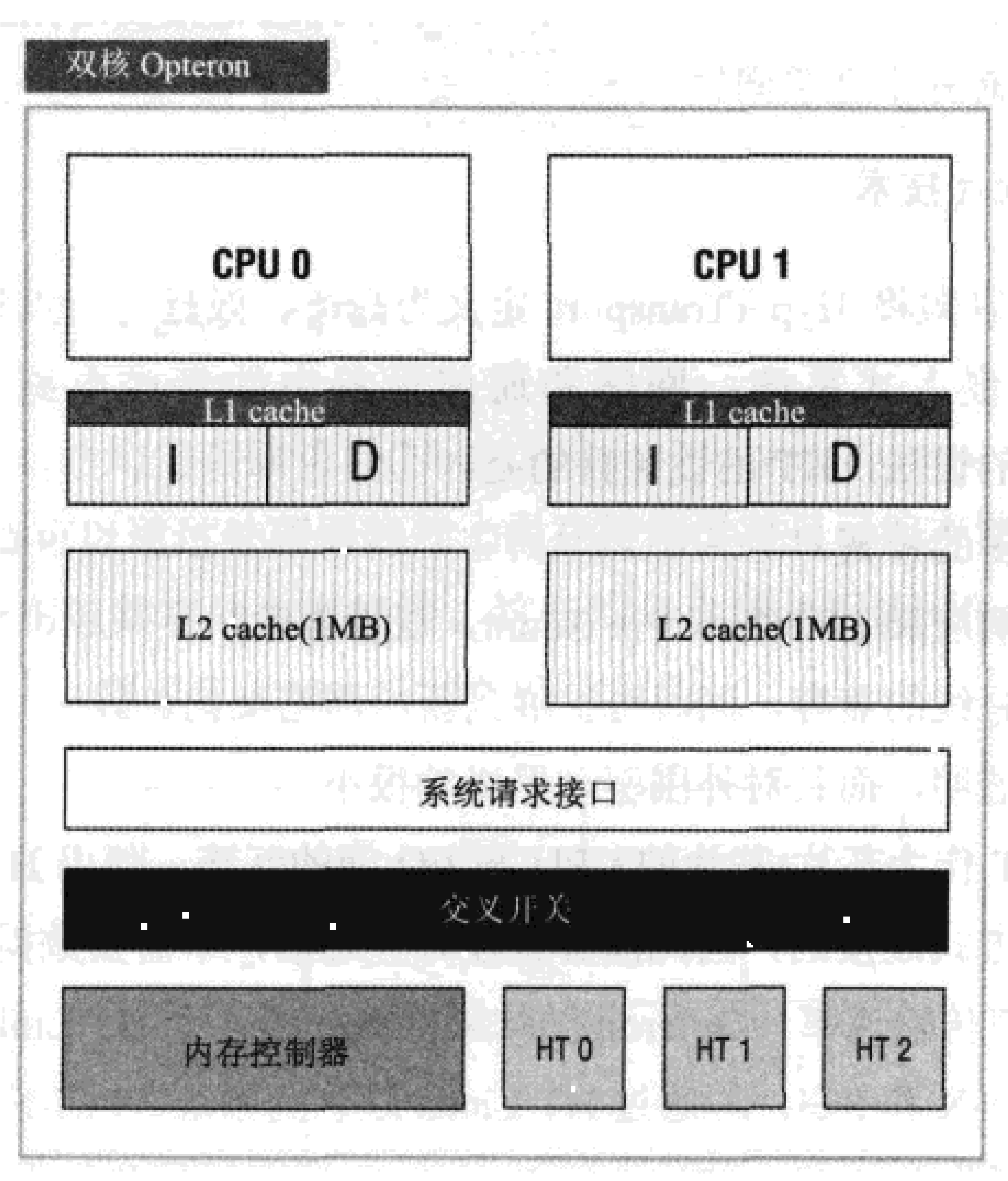


图 2-1

双核 Opteron 由两个 AMD 64 处理器、两组 L1 cache、两组 L2 cache、系统请求接口(System Request Interface，SRI)、交叉开关、内存控制器和 HyperTransport 技术组成。Opteron 同其他设计在架构上的关键区别之一就在于 AMD 中使用 HyperTransport 技术的直连架构(Direct Connect Architecture，DCA)。直连架构决定了 CPU 如何同内存以及其他 I/O 设备进行通信。

注意：

要想了解 AMD 子系统通信方法的价值，重要的是要了解总线技术在处理器架构中的地位。参见第 1.4 节以了解关于总线技术的更多内容。

2.1.1 Opteron 的直连和 HyperTransport

Opteron 处理器摆脱了基于总线的架构，它使用直连架构结合 HyperTransport(HT)技术

来避免基本的前端总线(FSB)、后端总线(BSB)和外围设备互联(PCI)结构带来的一些性能瓶颈。

1. 直连架构

DCA 是一种点到点(point-to-point)的连接模式，它不使用 FSB，而是将处理器、内存控制器和 I/O 直接连接到 CPU。这种专用连接方法避免了 CPU 和内存控制器间基于总线通信的潜在性能问题。同时由于连接是专用的，即每个内核都直接连接到自己的内存控制器和 I/O 控制器，因此避开了竞争问题。

2. HyperTransport 技术

HyperTransport 联盟将 HyperTransport 定义为高速、低延迟、点到点连接，用于增加计算机、服务器、嵌入式系统、网络和通信设备中的集成电路的通信速度。依照 HyperTransport 联盟的说法，HT 的设计目的是：

- 提供显著增多的带宽
- 使用低延迟响应和 LPC(Low Pin Count，低管脚数)
- 保留与老式总线的兼容，同时可扩展到新的网络架构总线
- 对操作系统透明，而且对外围驱动器影响很小

Opteron 使用 HT 作为芯片-芯片的 CPU 和 I/O 间的互连。使用 HT 连接的部件是以端到端(peer-to-peer)的方式连接的，因此能够直接相互通信，不需要数据总线。HT 的每个连接提供 12.8GB/s 峰值传输速率。Opteron 结构最多可配置 4 个 HT Links。I/O 设备和总线，如 PCI-E、AGP、PCI-X 和 PCI，均通过 HT Links 连接到系统。PCI-E、PCI-X、PCI 是 I/O 总线，而 AGP 是直接图形连接。PCI、PCI-E 和 AGP 用于系统和外围设备之间的连接。除了改进了处理器和 I/O 间的连接，HT 还可以用于方便 Opteron 上的处理器间的直接连接。通过使用 HT，Opteron 的多核通信得到了增强。

2.1.2 系统请求接口和交叉开关

系统请求接口(SRI)包含系统地址映射表并将内存范围映射到节点。如果内存访问是到本地内存的，则 SRI 中的映射表查找将它送到适当处理器的内存控制器。如果内存访问不是本地的(芯片外)，则通过一个路由表查找将它送到 HT 端口。更多内容请参考[Hughes，Conway，2007 IEEE]。图 2-2 显示了交叉开关的逻辑布局。

交叉开关有 5 个端口：内存控制器、SRI 和 3 个 HT。交叉开关处理在逻辑上分为命令报头包处理和数据报头包处理。逻辑上，部分交叉开关专用于命令包路由，其他部分专用于数据包路由。

PDG

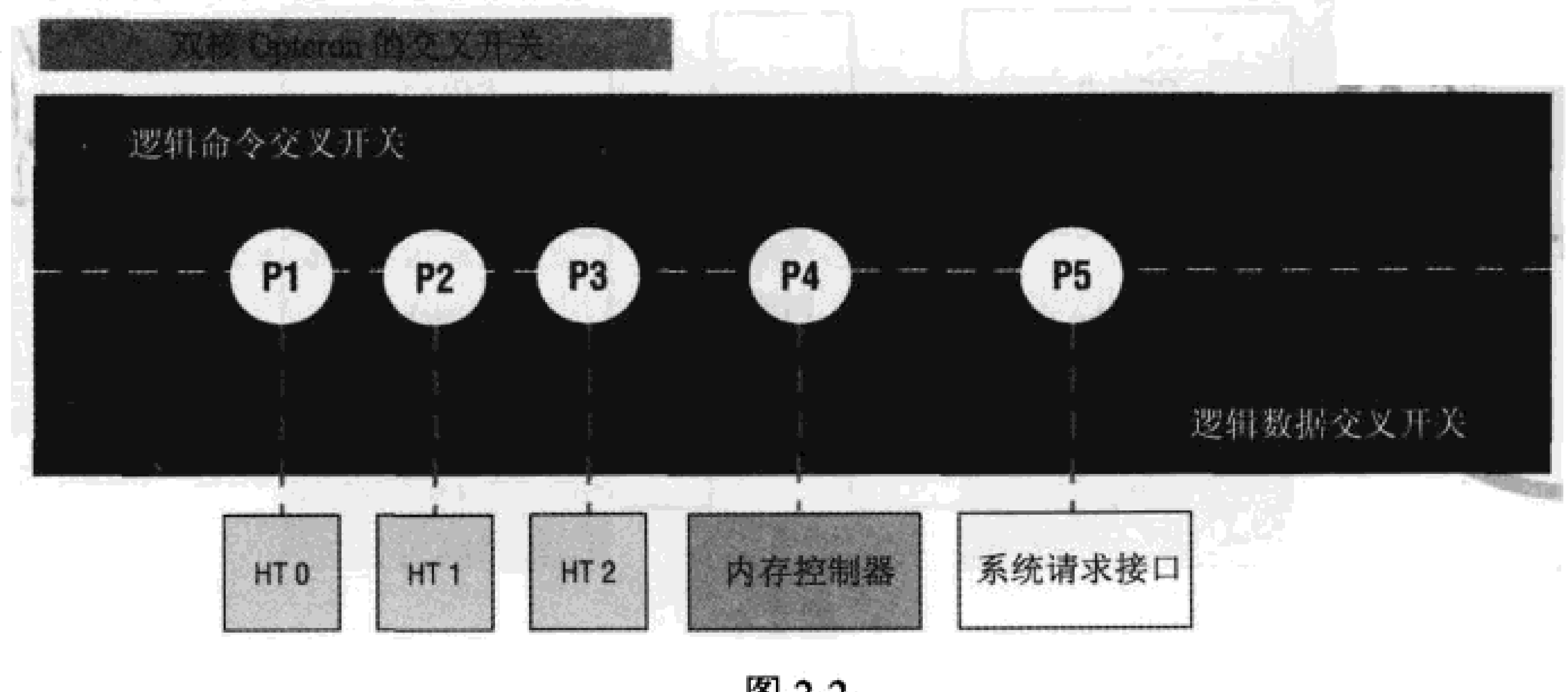


图 2-2

2.1.3　Opteron 使用 NUMA 结构

Opteron 使用的是非均匀存储访问(Non-Uniform Memory Access，NUMA)结构。在这种结构中，每个处理器可以通过处理器的片上内存控制器对自己的本地内存进行快速访问。NUMA 结构使用分布但共享的内存结构。这与均匀存储访问(Uniform Memory Access，UMA)结构相反。图 2-3 显示了 UMA 结构的简化框图。

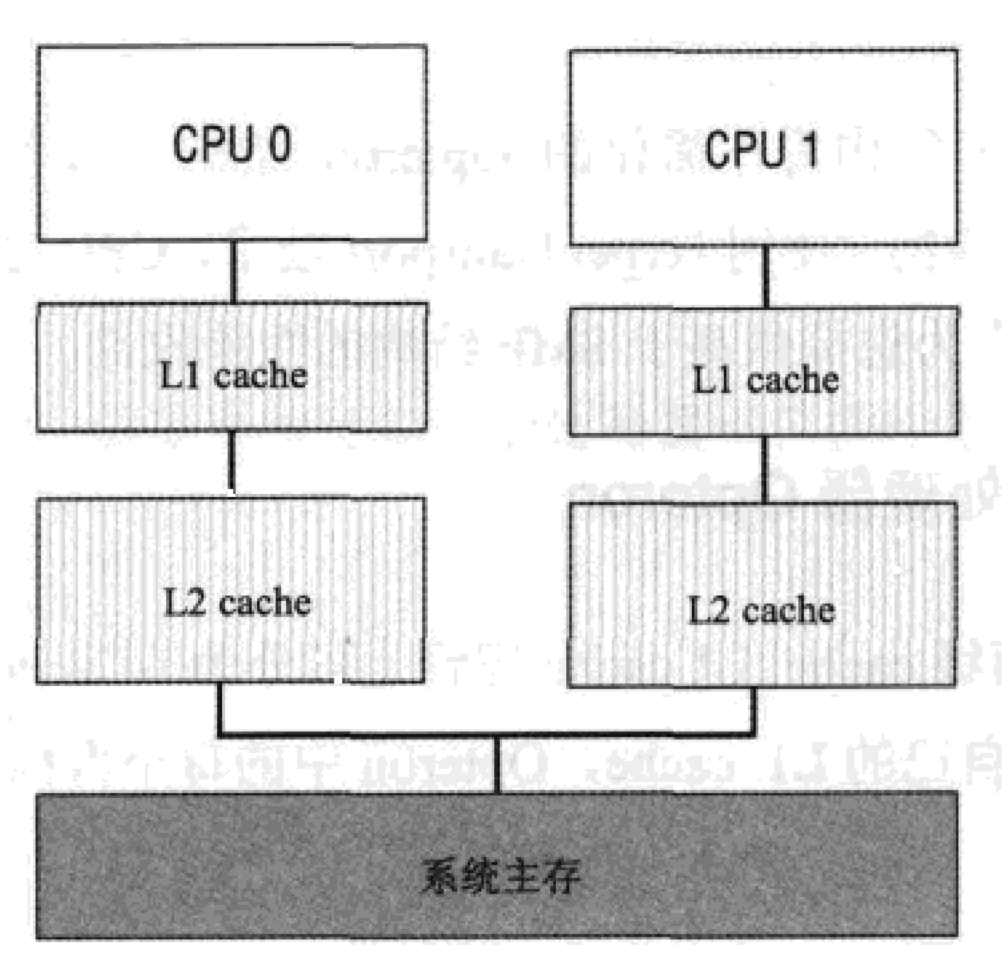


图 2-3

注意在图 2-3 中，多个处理器共享单一的内存。每个处理器的内存访问时间与其他处理器一致。图 2-3 中的处理器结构经常被称为对称(共享内存)多处理器(SMP)。这是由于所有处理器有着一致的内存延迟，即便内存按照多个 bank 来组织[Hennessy，Patterson，2007]。SMP 的单一主存和一致访问时间使得它要比 NUMA 的对应部分更容易实现。同时 UMA 中共享地址空间的概念也更直接一些，因为只需要考虑一个系统主存。

相比之下，图 2-4 显示了 NUMA 结构的简化框图。

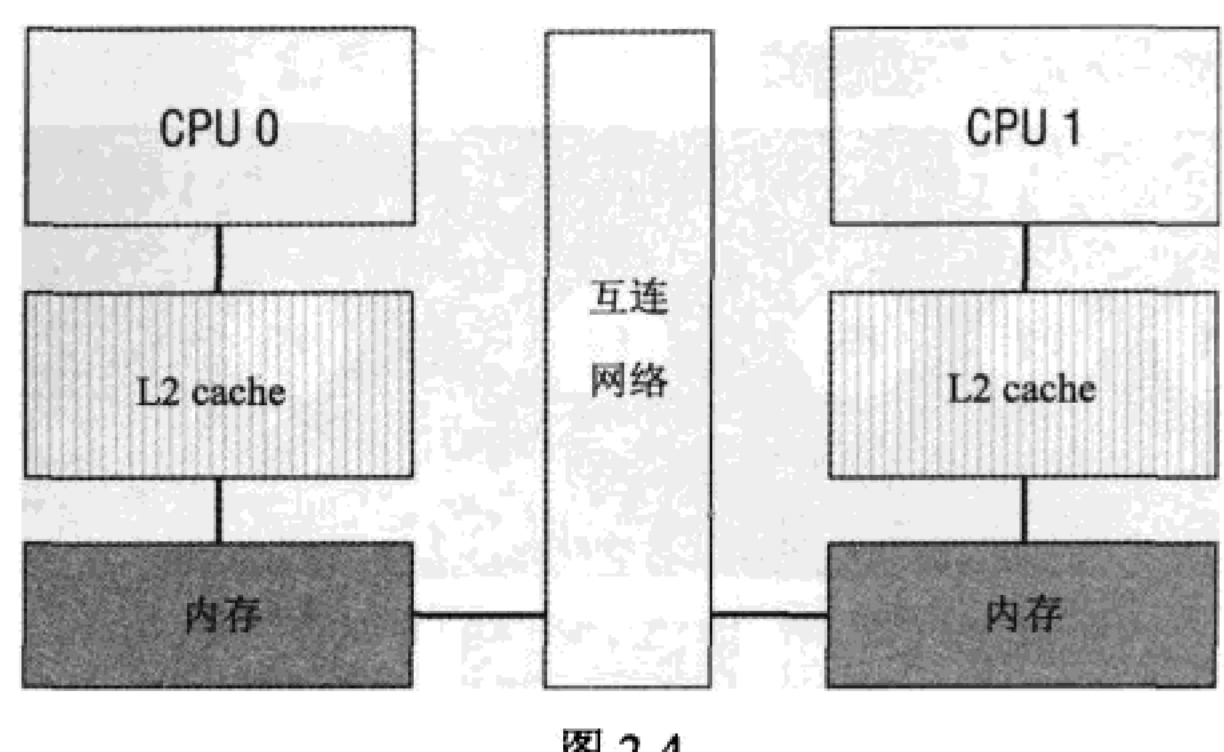


图 2-4

NUMA 是一个分布式共享存储(distributed shared memory，DSM)结构。注意在图 2-4 中，每个处理器有它自己的内存块，但每块内存共享同一个地址空间。也就是说，两个处理器上相同的物理地址指向内存中相同的位置[Hennessy，Patterson，2007]。在 UMA 和 NUMA 两种结构中，处理器均共享地址空间。然而，在 NUMA 结构中，地址空间是从逻辑角度共享的，而在 UMA 结构中，处理器物理地共享相同的地址块。SMP 结构能够满足较小的结构，但是一旦处理器数目增加，唯一的内存控制器可能会成为瓶颈，因此使得总体系统性能降级。而另一方面，NUMA 结构可以很好地扩展，因为每个处理器有着自己的内存控制器。

如果您将图 2-4 中的结构看作简化的 Opteron 结构，那么网络互连是通过 Opteron HyperTransport 技术来完成的。使用 HyperTransport 技术，CPU 之间直接连接，而且 I/O 直接连接到 CPU。这最终会为您提供高于 SMP 结构的性能收益。

2.1.4 cache 以及多处理器 Opteron

双核 Opteron 支持两级 cache。L1 cache 逻辑上可分为 I-Cache(用于指令)和 D-Cache(用于数据)。每个内核有着自己的 L1 cache。Opteron 中的每个内核还在处理器和系统主存之间有 1MB L2 cache。

2.2 Sun UltraSparc T1 多处理器

UltraSparc T1 是八核 CMP，并且提供了对芯片级多线程(chip-level multithreading，CMT)的支持。每个内核能够运行 4 个线程，这有时也被称作超线程。UltraSparc T1 的 CMT 意味着 T1 能够处理最多达 32 个硬件线程。这对软件开发者意味着什么呢？8 个运行 4 线程的内核是作为 32 个逻辑处理器呈现给应用程序的。程序清单 2-1 包含的代码可用于查看多少个处理器是操作系统明显可用的(不需要特殊的编译器等)。

程序清单 2-1

```
// Listing 2-1
// uses sysconf() function to determine how many
// processors are available to the OS.

using namespace std;
#include <unistd.h>
#include <iostream>

int main(int argc,char *argv[])
{
   cout < < sysconf(_SC_NPROCESSORS_CONF) < < endl;
   return(0);
}
```

注意：

在适当的时候，本书中的程序清单将伴有一个程序概要，用于说明程序的环境平台。如果希望在非兼容的操作系统上运行代码，则需要使用该操作系统的 POSIX-兼容特性。

程序概要 2-1

程序名：

program2-1.cc

描述：

这个程序使用 sysconf( )函数来确定多少个处理器可供操作系统使用。

必需的库：

无

必需的头文件：

<unistd.h> <iostream>

编译和链接指令：

g++ -o program2-1 program2-1.cc

测试环境：

SuSE Linux 10 和 gcc 3.4.3

硬件：

AMD Opteron Core 2、UltraSparc T1 和 CBE

执行指令：

```
./program2-1
```

注释：

无

当这个程序在 T1 上执行时，将打印出 32。函数 sysconf( )为应用程序提供了获取系统限制或变量的值的方法。在本例中，_SC_NPROCESSORS_CONF 参数询问配置的处理器的数目。_SC_NPROCESSORS_MAX 参数可用于获得支持的最大处理器数目。UltraSparc T1 提供了本书中所讨论的架构的最大片上线程数。8 个内核中的每一个都等同于能够运行 4 个线程的 64 位执行流水线。图 2-5 包含了 UltraSparc T1 多处理器的功能概览。

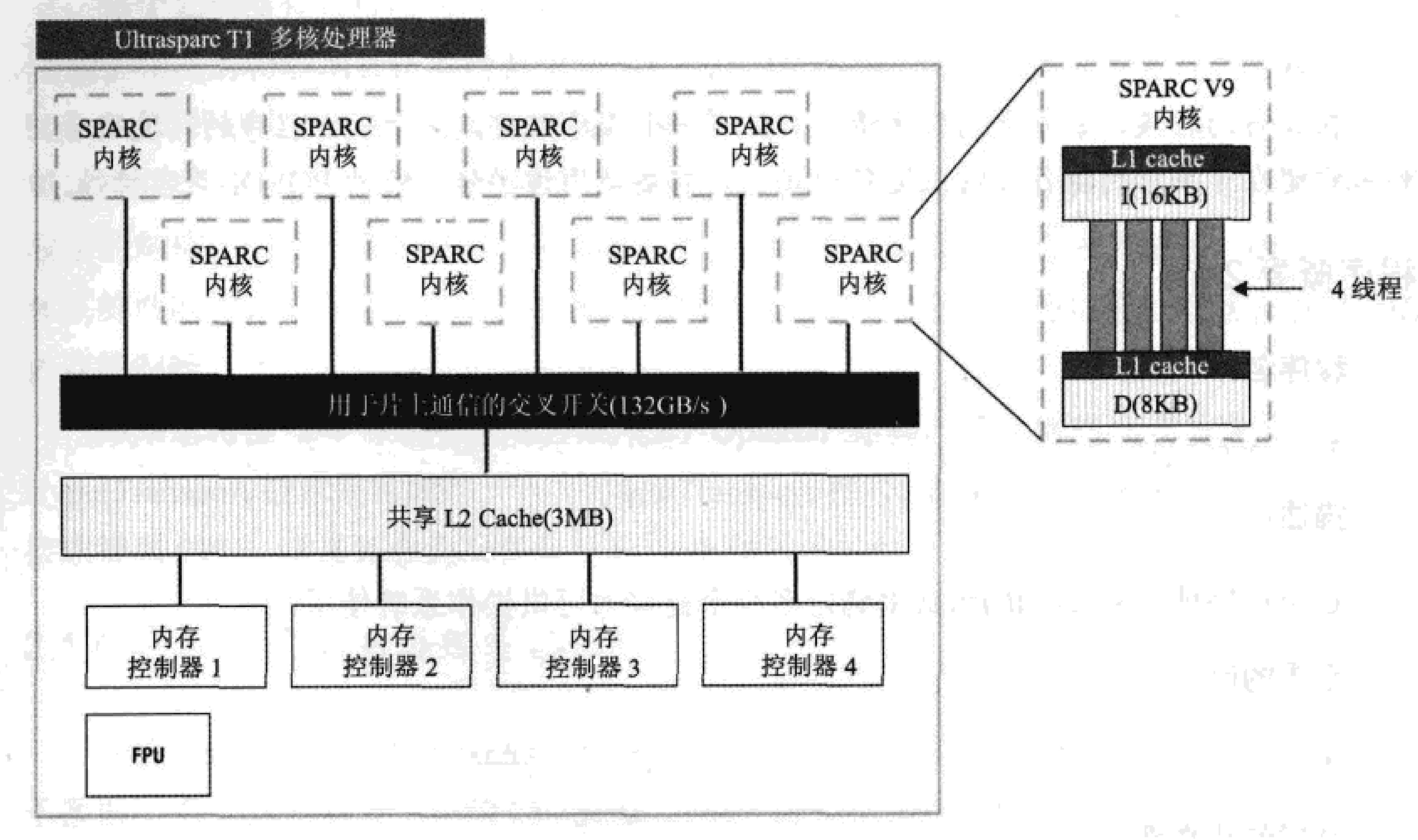


图 2-5

2.2.1 UltraSparc T1 内核

T1 由 8 个 Sparc V9 内核组成。V9 内核采用的是 64 位技术。每个内核均有 L1 cache。在图 2-5 中可以注意到有 16KB L1 指令 cache 和 8KB L1 数据 cache。8 个内核共享一个单独的浮点单元(FPU)。图 2-5 显示了 L2 cache 和 8 个内核的访问路径。4 个线程共享 L2 cache。每个内核有一条 6 级流水线：

- Fetch (获取)
- Thread selection (线程选择)
- Decode (解码)

- Execute (执行)
- Memory access (内存访问)
- Write back (写回)

2.2.2　Cross Talk 与 Crossbar

注意在图 2-5 中，内核与 L2 cache 是通过 cross-switch 或交叉开关(crossbar)来连接的。crossbar 对片上通信提供了 132GB/s 的带宽。crossbar 已经为 L2 cache 到内核以及内核到 L2 cache 的通信进行了优化。FPU、4 个 L2 cache bank、I/O 桥以及内核都通过 crossbar 进行通信。基本上，crossbar 承担中介的任务，允许 T1 的部件相互进行通信。

2.2.3　DDRAM 控制器和 L2 cache

UltraSparc T1 有 4 个独立的内存控制器。每个控制器同 L2 cache 的一个 bank 连接。L2 cache 在 T1 中分为 4 个 bank。T1 最多可支持 128GB RAM。

2.2.4　UltraSparc T1、Sun 和 GNU gcc 编译器

在介绍 UltraSparc T1 的架构时，将同 AMD Opteron、IBM Cell Broadband 架构、Intel Core 2 Duo 进行对比。尽管这些架构都是多核的，但是在实现上的区别非常明显。从最高的级别来看，利用多核的应用程序将它们都视为两个或多个处理器的集合。然而，从优化的角度来看，则需要考虑更多的内容。UltraSparc T1 上最常用的两种编译器是 Sun C/C++ 编译器(Sun Studio 的一部分)和 GNU gcc(标准的开源 C/C++)编译器。在 Sun 的编译器对他们的处理器提供了最好的支持的同时，GNU gcc 也为 T1 提供了大量的支持，有很多选项可用来利用线程、循环展开、向量操作、分支预测以及特定的 Sparc 平台选项。事实上，本书中所有的程序实例均已经在有着八核 T1 处理器的 SunFire 2000 上编译并执行。查看程序清单中的程序概要，您将会看到我们为 T1 探讨了哪些编译器开关。

2.3　IBM Cell Broadband Engine

CBE 是一种异构多核芯片。它是一种异构架构，因为它由两种不同类型的处理器组成：PowerPC Processing Element(PPE)和 Synergistic Processor Element(SPE)。CEB 由 1 个 PPE、8 个 SPE、1 个高速内存控制器、1 条高带宽部件互连总线、高速内存和 I/O 接口组成，且都集成在芯片上。这使得它成为了一种混合型的九内核处理器。图 2-6 显示了 CBE 处理器的概况。

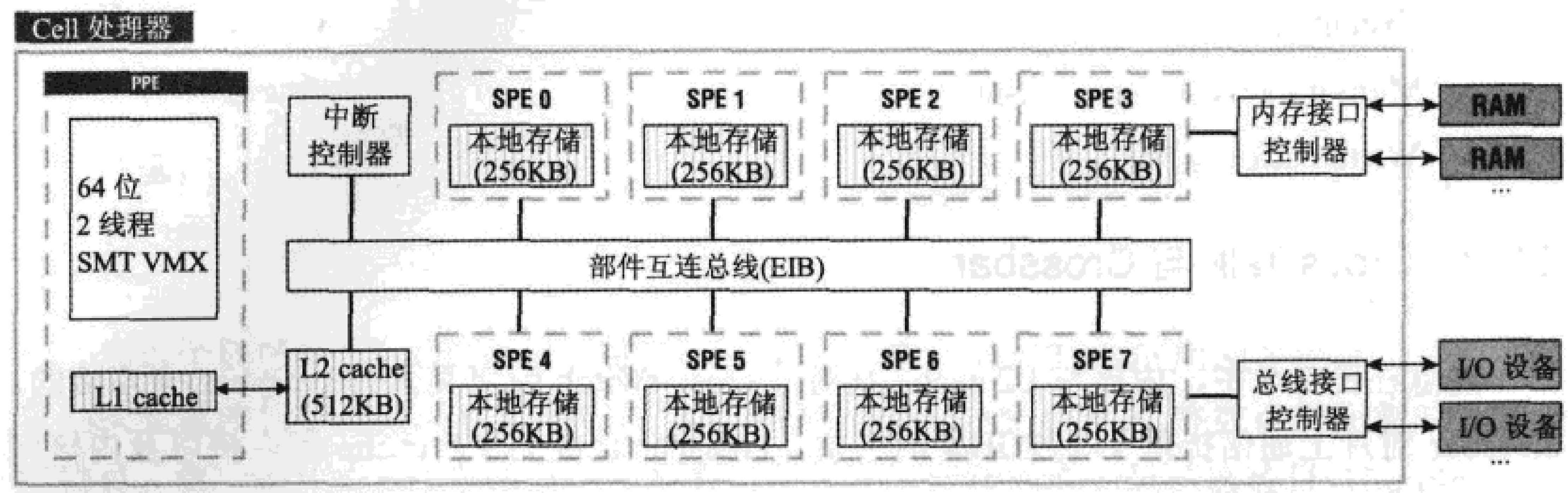


图 2-6

多数 CMP 的处理器是同构的，也就是说，这些处理器使用相同的指令集。CBE 上的处理器有两组不同的指令集。尽管每个处理器部件都已经为特定类型的操作进行优化，但是两种部件均可用于通用计算。

- Cell 处理器中的第一个部件是 64 位 PowerPC 处理器。这个部件完全符合 64 位 PowerPC 架构，而且可以执行 32 位或 64 位操作系统及应用程序。
- 第二种类型的处理器部件是 SPE。SPE 已经为运行单指令多数据(SIMD)应用程序进行了优化。

尽管 CBE 有一些商业科学用途，但是它最常用作 Sony 公司的 Playstation 3 的处理器。

2.3.1 CBE 与 Linux

我们将 CBE 选作 4 种有影响的多核架构设计之一，是因为它能够在 Linux 环境中提供很高的性能。Playstation 3 是一个灵活的设备，而且预先安装有 Linux。当前，为 CBE 发布的 Linux 有 Fedora 和 Yellow Dog 版本。Playstation 3(PS3)的低成本使得任何软件开发者都可以进行异构多核应用程序开发。PPE 部件和 SPE 可以使用标准 GNU gcc 编译器编程。IBM 提供了可下载的 CBE SDK，它包含了编译 SPE 代码所必需的工具。基本上，SPE 代码单独编译，然后同 PPE 代码链接在一起，构成一个执行单元。PPE 和 SPE 合作，均会发挥各自的特色。典型情况下，SPE 使用 PPE 来运行操作系统代码和多数应用程序中的主线程或最高层线程。PPE(通用处理器)将 SPE 作为应用程序的高性能劳动力(workhorse)来使用。SPE 对 SIMD 操作、计算密集型应用、向量类型操作提供了很好的支持。当您在 CBE 上运行程序清单 2-1 中的代码时，打印到控制台的数字为 2，这是因为 SPE 可直接访问。2 表示 PPE 是 CMT 的事实，它是一个双线程处理器。因此在完整的配置下，在 CBE 结构中可以有多个逻辑处理器(包括 SPE)。异构架构还导致了一些有趣的设计选择。

尽管 PPE 部件可以使用标准 POSIX 线程(pthread)和进程管理，但 SPE 必须使用作为 CBE SKD 的一部分的线程库来进行编程。好消息是 SPE 线程调用被设计为同 pthread 兼容，因此对于熟悉 pthread 库的开发者而言，不存在学习障碍。

2.3.2　CBE 内存模型

PPE 在访问内存方面与 SPE 不同。尽管只有一个内存流控制器，但是 CBE 避免了潜在的单总线瓶颈，因为每个 SPE 均有自己的本地内存。图 2-7 显示了 PPE 和 SPE 的内存配置。

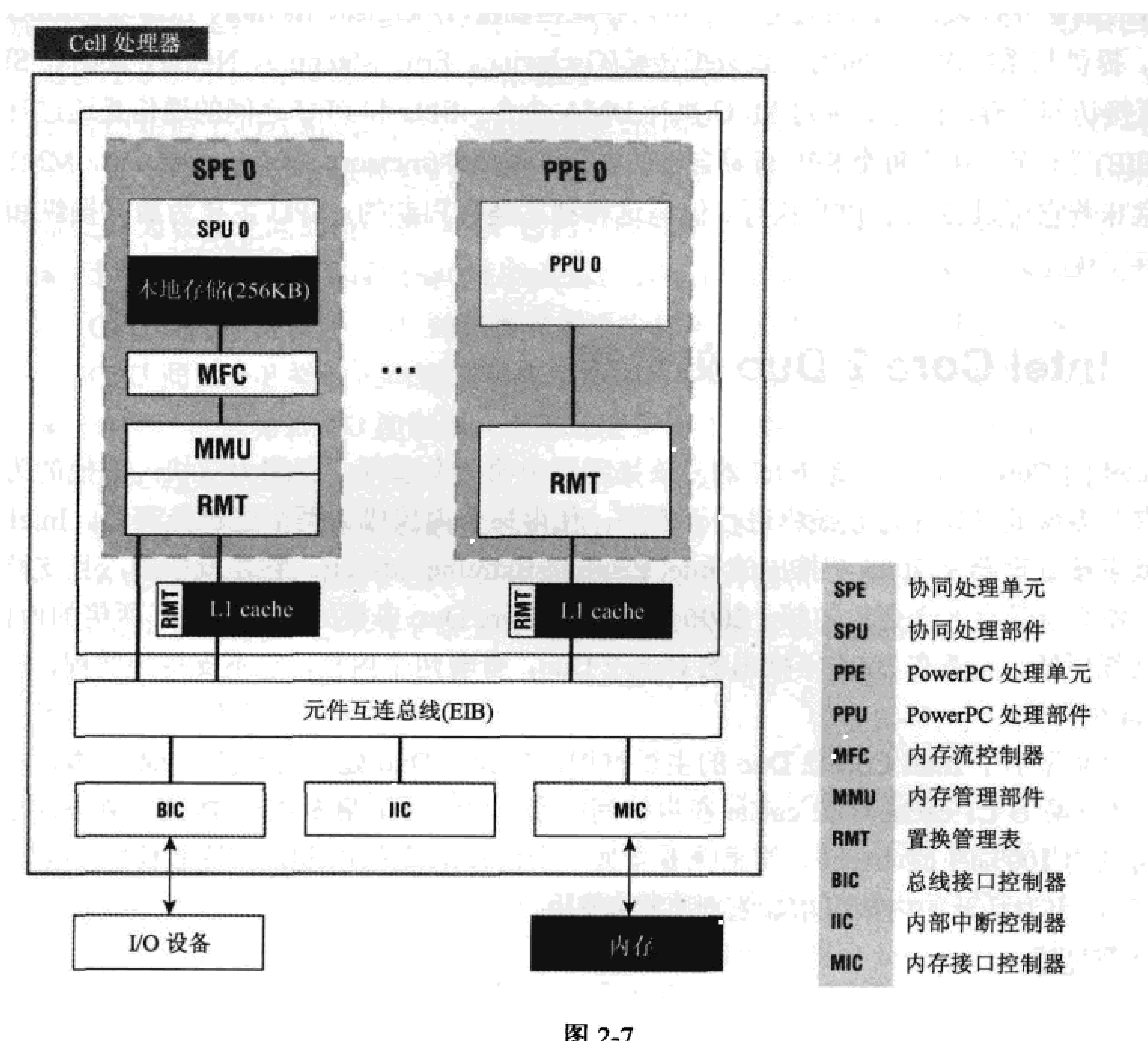


图 2-7

SPE 结构是得到这些收益的原因。SPE 提供了三级内存访问。它通过本地存储、寄存器文件、直接存储访问(DMA)传输到主存。这种三层内存结构允许程序员同时调度数据和代码的传输。CBE 处理器最多可支持 SPE 本地存储和主存储之间的 128 路同时传输。尽管 SPE 是为 SIMD 类型操作优化的，PPE 也支持并行向量/SIMD 操作。

2.3.3　对操作系统隐藏

CBE 是为了获得最大性能而必须直接寻址的很好的多核示例。标准 Linux 系统调用可以知道 PPE 的双线程，但是并不完全知道 SPE。开发者必须显式地开发和编译使用 SPE 的代码，然后该代码必须同来自 PPE 的代码连接起来。这样，Linux 才能知道如何处理 8 个 SPE 处理器。CBE 的异构架构还为希望进行进一步深入研究的开发者提供了令人激动的设

计选择。

2.3.4 协处理器部件

SPE 包含了协处理器部件(synergistic processor unit，SPU)，用于加速各种工作负荷，提供高效的数据并行架构。此外还包含了协内存流控制器(synergistic memory flow controller，MFC)，提供与系统内存之间的一致数据传输[Gschwind，Erb，Manning，Nutter，2007]。SPU 并不直接访问主存，而是必须对 MFC 执行 DMA 命令。SPU 和 PPU 之间的通信是通过互连总线(EIB)进行的。由于每个 SPE 有着自己的内存管理部件(memory management unit，MMU)，这就意味着它可以独立于 PPE 执行。但是这种独立是有限制的。SPU 主要为数据操纵和计算进行了优化。

2.4 Intel Core 2 Duo 处理器

Intel 的 Core 2 Duo 只是 Intel 的多核处理器中的一个系列。有些为双核，其他的为四核。有些多核处理器通过超线程进行了增强，使得每个内核成为两个逻辑处理器。Intel 的第一款多核处理器是 2005 年推出的 Intel Pentium Extreme Edition。它是双核的，且支持超线程，使系统具有 8 个逻辑内核。2006 年推出的 Core Duo 多核处理器提供了两倍的内核，而且功耗更低。同是在 2006 年推出的 Core 2 Duo，有着两个内核，它不支持超线程，但是支持 64 位架构。

图 2-8 显示了 Intel Core 2 Duo 的主板框图。Core 2 Duo 处理器有两个 64 位内核，每个内核有 64KB L1 cache。L2 cache 在内核间共享。L2 cache 最多可达 4MB。两个内核都最多可利用 100%的 L2 cache。这意味着当另一个内核未被充分利用，因此不要求很多 L2 cache 时，更为活跃的内核可增加它对 L2 的使用。

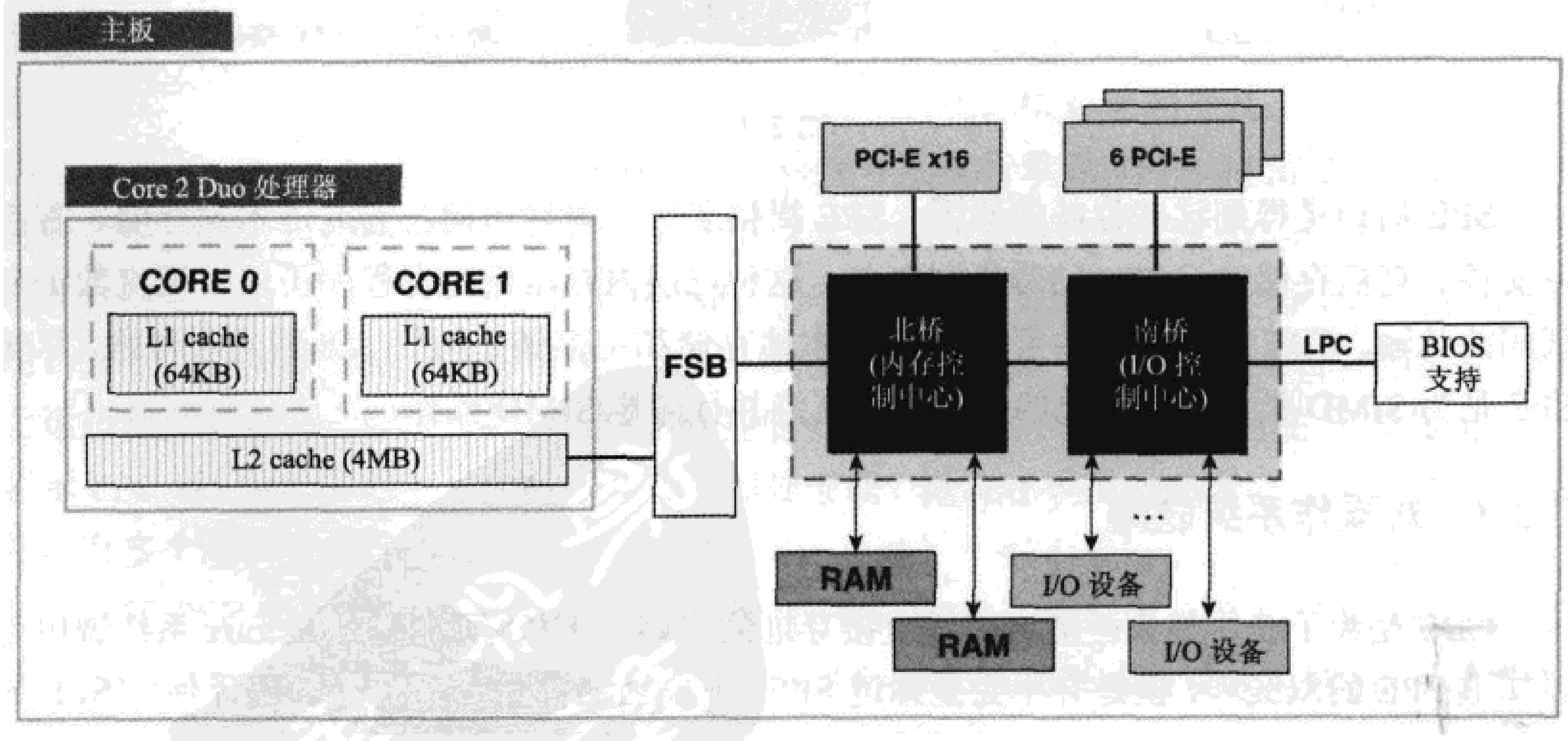


图 2-8

2.4.1　北桥和南桥

除了 CPU 以外，主板上最重要的部件是芯片组(chipset)。在图 2-8 中，芯片组是被设计为连接 CPU 和主板上其他部件的一组集成电路。它是主板的集成部分，因此不能被移走或升级。它用于和特定类别的 CPU 或 CPU 系列共同工作，以优化 CPU 性能和系统性能。芯片组将数据在 CPU 和主板上其他部件之间来回移动，这些部件包括内存、显卡、I/O 设备，如图 2-8 所示。所有到 CPU 的通信都经由芯片组。

芯片组由两个芯片组成：北桥(Northbridge)和南桥(Southbridge)。之所以如此命名它们，是由芯片在主板上的位置以及它们的用途决定。北桥位于北部区域，南桥位于南部区域。两者都是作为设备之间的桥梁或连接，它们为部件提供桥接，以确保数据到达期望的位置。

- 北桥，也被称为内存控制中心(memory controller hub)，直接通过前端总线(FSB)与 CPU 通信。它将 CPU 同高速设备连接起来，如主存。它还通过一条内部总线将 CPU 同 PCI-E 插槽及南桥连接起来。数据在到达南桥之前，首先要经过北桥。
- 南桥，也被称为 I/O 控制器，它的速度要比北桥慢。由于它不是直接连接到 CPU，因此它负责主板上较慢的部分，如音频、磁盘接口等 I/O 设备。南桥通过串行外设接口(Serial Peripheral Interface，SPI)、6 个 PCI-E 插槽、图中未显示的其他 I/O 设备连接到 BIOS 支持。SPI 使用主从配置来允许南桥与 BIOS 支持之间的数据交换(每次 1 比特)。它使用全双工方式，意味着数据可以双向传送。

2.4.2　Intel 的 PCI Express

PCI-E 或 PCI Express 是计算机扩展卡接口。该插槽是作为在主板上与声卡、显卡和网卡的串行连接。串行连接速度较慢，每次发送 1 比特。PCI-E 是高速串行连接，其工作方式更像是网络，而不是总线。它使用一个开关来控制很多被称为 lane 的点到点的全双工(同时双向通信)串行连接。每个插槽可以有 4、8 或 16 个 lane。每个 lane 有两对从开关到设备的线，一对发送数据，一对接收数据。这决定了数据的传送速度。这些 lane 从开关直接扇出到数据要去的设备。PCI-E 是 PCI 的替代产品，并且提供了更多的带宽。设备不共享带宽。加速图形端口(Accelerated Graphics Port，AGP)被 PCI-E x16(16 lane)插槽所替代，它能够提供更高的数据传送速度(8GB/s)。

2.4.3　Core 2 Duo 的指令集

Core 2 Duo 通过支持 SSE(Streaming SIMD Extensions)以及执行向量化指令的特殊寄存器提高了性能。SSE3 提供了 13 条指令，用于在打包的整型或浮点数据单元上执行 SIMD 操作。这样就加快了利用 SIMD 操作的应用程序，例如高度密集图形、加密以及数学应用。处理器有 16 个寄存器用于执行 SIMD 指令：8 个 MMX 和 8 个 XMM 寄存器。MMX 寄存器支持对 64 比特打包字节、字、双字整数进行 SIMD 操作。XMM 数据寄存器和 MXCSR

寄存器支持对 128 比特打包单精度和双精度浮点值以及 128 比特打包字节、字、双字和四字整数进行 SIMD 操作。表 2-3 给出了 3 种寄存器 XMM、MMX 和 MXCSR 的简要描述，以及涉及的 SIMD 操作。

表 2-3

寄 存 器 组	描　　述
MMX	8 个寄存器的集合，用于对 64 比特打包整数数据类型执行操作
XMM	8 个寄存器的集合，用于对 128 比特打包单精度和双精度浮点数执行操作
MXCSR	和 XMM 寄存器一同使用，用于状态管理指令的寄存器

有很多编译器开关可用来激活多核处理器的各种特性。对于 Intel C/C++编译器，有着可用来激活向量选项以利用 SIMD 指令、自动并行选项、循环展开、为特定处理器进行的代码生成优化等的编译器开关。

注意：

您可能回想起在第 1 章中，表 1-1 列出了同 CPU 交互的编译器开关的种类，以及影响程序或应用程序如何执行和利用内核资源的指令集。

2.5 小结

尽管操作系统的主要工作之一就是封装硬件的细节，并提供硬件无关的接口，但是特定类型的开发人员需要了解硬件特性。这些人员包括库开发人员、编译器设计人员、系统程序员、内核程序员、服务器开发人员、游戏设计人员和开发人员以及主要设计目标为最大化系统性能的人员。4 种有影响但是互不相同的多核架构设计是：

- Opteron
- UltraSparc T1
- Cell Broadband Engine
- Core 2 Duo

如果我们所看到的，每种设计都有唯一特性，作为开发人员的您可以在考虑从多核角度进行编程时进行衡量。C/C++编译器是这些设计的第一级接口。同构 CMP 设计有着一致的内核，异构设计中的内核有着不同的指令集和架构。CBE 是异构 CMP 的很好实例。

本章介绍了 4 种架构，我们将在全书中对它们进行引用。所有的代码实例均在一种或多种上述架构中进行了编译和测试。多数实例已经在全部环境中进行了编译和测试。必要时，程序清单后的程序概要包含了特定编译器开关和连接选项。尽管每种架构各不相同，但是我们示范了如何以标准方式对待它们的方法。我们希望您能够尽可能以最常用的方式来利用硬件特性。对于很多软件应用程序，硬件实现的区别都被隐藏了，因为主要设计目的之一是使软件在尽可能多的硬件平台上兼容。所以我们会尽量避开特别的硬件特性，就

像操作系统所做的主要工作之一。但是在有些应用程序中，您需要了解硬件实现的特性，从而可以优化代码。这些应用程序的优化要比兼容性重要。这些应用程序包括高度事务性数据库服务器、网络服务器、应用服务器、硬件密集型游戏引擎、编译器、操作系统内核、设备驱动、高性能科学建模和可视化软件。这些应用程序的开发人员被迫寻找并利用平台特性，从而使得他们的应用程序能够被终端用户所接受。如果您是此类开发人员，熟悉特定处理器或处理器系列是有效的软件开发的前提。

在第 3 章中，我们将介绍多核编程的挑战。

第3章 多核编程的挑战

迄今为止，最常见的软件开发工具和技术仍集中在计算机程序执行的顺序模型的概念上。在大学、学院、技术院校中，对信息技术和计算机科学基本的设想就是软件开发人员将会在单处理器计算机环境下工作。直到最近，教育学府仍对并行编程不够重视的事实就很好地证明了这一点。对并行编程缺少重视的两个主要原因是成本和传统。

- 成本：首先，单处理器计算机要比多处理器计算机便宜很多，而且更容易购买到。成本和易购性使得单处理器计算机成为多数商业、学院和政府机关的配置选择。
- 传统：其次，软件开发和计算机编程的基本思想是几十年前产生的，而且当时受到单处理器环境的约束。查找、排序、计数、解析、检索的基本算法都是在串行编程模型下开发、精化、完善的。这些基本算法、数据结构、编程模型以及软件工程方法学构成了当前使用的多数软件开发方法的基础。

顺序编程技术是重要的，而且将来会保留其一席之地。然而，多处理器计算机配置正日益普遍。这为程序分解和软件组织带来了一种非常不同的方法。包含顺序编程、多处理、多线程的软件架构会非常常见。对于多数开发人员，这些混合型软件架构将会让他们一头雾水。多处理器计算机将会在商业、学院和政府中取代单处理器配置已经是大势所趋。为了利用多处理器环境，软件开发人员必须掌握一套新的工具和技术。要求多核或并行编程的软件项目对只熟悉顺序编程模型的开发人员提出了挑战，本章主要介绍开发人员在转向要求多核或并行编程的项目时所面临的挑战。我们将讨论软件开发生命周期(Software Development Life Cycle，SDLC)和将它应用到并发模型的方法。同时，我们将讨论问题分解和解决方法，以及过程式模型和声明式模型。

3.1 什么是顺序模型

在基本的顺序编程模型中，计算机程序的指令每次执行一条。程序被看作是配方(recipe)，由计算机按照指定的顺序和数量来执行每个步骤。程序的设计人员将软件分成一组任务，每个任务按照指定的顺序执行，而且每个任务排好队，并等待轮到它。在顺序模

型中计算机程序的设置与一个故事的结构类似，程序有清楚的开头、中间部分和结局。设计人员或开发人员将每个程序预想为任务的简单顺序推进。任务不仅必须在一个文件中行进，而且任务之间的关系必须是：如果第一个任务因为某些原因不能够结束它的工作，则第二个任务不能够开始。每个任务在能够执行之前，必须等待前一个任务得出的计算结果。在顺序模型中，任务通常是连续依赖的，即 A 需要来自 B 的内容，而且 B 需要来自 C 的内容，C 需要来自 D 的内容，依此类推。如果 B 因为一些原因失败，那么 C 和 D 将无法执行。在顺序环境中，开发人员习惯于将软件设计为先执行步骤 1，然后执行步骤 2，继而步骤 3。这种“每次一步”的模型在软件设计和开发过程中根深蒂固，很多程序员很难想到任何其他方法。毕竟每个问题的解决方法、每个算法的设计、每种数据结构的布局，都依赖于计算机每次访问一条指令或一个数据块。

当软件需求中包括多线程或多处理组件时，情况就发生了变化。当需要并行处理时，实质上软件设计和实现的每个方面都会受到影响。开发人员面临着并发带来的十大挑战：

(1) 将软件分解为需要同时执行的指令或任务集

(2) 并行执行的两个或多个任务之间的通信

(3) 被两个或多个指令或任务并发访问或更新数据

(4) 确定并发执行的任务片之间的关系

(5) 当任务与资源之间存在多对一的关系时，控制资源竞争

(6) 确定需要并行执行的最优或可接受的单元数目

(7) 创建测试环境来仿真并行处理需求和条件

(8) 重建软件异常或错误以去除软件缺陷

(9) 记录并沟通包含多处理和多线程的软件设计

(10) 为涉及多处理和多线程的组件实现操作系统和编译器接口

3.2 什么是并发

如果两个事件在相同的时间间隔发生，则我们称它们是并发的。两个或多个任务在相同的时间间隔内执行，被称作并发执行。并发并不一定意味着在完全相同的时刻，例如两个任务可能在同一秒内并发，但是每个任务是在这一秒内的不同部分执行的。第一个任务可能在前十分之一秒内执行，然后暂停。第二个任务可能在接下来的十分之一秒内执行，然后暂停。第一个任务再次启动，并执行十分之一秒。每个任务交替执行。然而，一秒的时间非常短，因此看上去两个任务是同时执行的。

我们可以将这种概念扩展到更长的时间间隔。两个在同一小时内执行某些任务的程序，使得在那个小时里，任务持续得到进展。不论它们是否在相同的时刻执行，我们都说在那个小时里，这两个程序是并发执行的。在相同时刻存在并在相同的时段内执行的任务就是并发的。并发任务可以在单处理或多处理环境中执行。在一个单处理环境中，并发任务在相同的时间存在，而且通过上下文切换在相同的时段内执行。在多处理器环境中，如

果有足够的空闲处理器，并发任务可以在相同的时段内在同一个时刻执行。并发的可接受时段的决定性因素是同应用程序相关的。在本书中，我们将在 3 个方面应对并发的挑战：

- 软件开发
- 软件部署
- 软件维护

尽管还可以从其他角度来考虑和分组与多处理和并行编程相关的问题，但我们选择这些类别，因为根据我们的经验，多核编程涉及的多数困难可归到这些类别中。

注意：

本章主要讨论软件开发，第 10 章将讨论维护和部署。

3.3　软件开发

软件开发的形式多种多样，从设备驱动开发到构建大规模 N 层企业级应用软件。尽管涉及的软件开发技术随着应用程序的规模和范围而变化，但是使用多处理或多线程的任何应用程序都面临一组相同的挑战。这些挑战在 SDLC 的各个阶段都存在。理解多核编程与 SDLC 之间的联系非常重要，这是因为处理多核编程的复杂性、需求以及潜在的陷阱的最简单的方法就是在 SDLC 的适当阶段来着手处理问题。SDLC 描述了设计人员和开发人员在生产高质量软件的过程执行的必要活动。由于创建好的软件是艺术、工程和科学的结合，因此关于 SDLC 的组成存在多种理论。表 3-1 列出了在多数版本的 SDLC 中均出现的主要活动。

表 3-1

主要 SDLC 活动	描　述
Specifications (规格说明)	通过指定软件必须完成的任务以及软件的约束，记录开发人员与客户之间的协定
Design (设计)	确定软件如何满足规格说明中所规定的内容；设计决定了软件的内部结构；设计可被分为两种方法：结构化设计(系统被划分为模块)和详细设计(对模块进行描述)
Implementation(实现)	将详细设计转换为代码
Testing and evaluation (测试和评价)	试用软件，通过判断软件满足指定需求的程度来判断其质量的过程
Maintenance(维护)	软件产品被交付之后，为了修正错误、改进性能、改进属性或使软件适应环境改变而进行的修改

有很多方式来考虑和组织表 3-1 中的活动。此外，表 3-1 中列出的活动只是多数版本的 SDLC 都有的核心活动。每种对 SDLC 中的活动的组织方法都派生出自己的软件开发方法。一旦建立了软件开发方法，则会创立工具集、语言和软件库来支持该方法。例如面向

对象软件革命派生出如下概念：

- 面向对象软件工程(Object-Oriented Software Engineering，OOSE)
- 面向对象分析(Object-Oriented Analysis，OOA)
- 面向对象设计(Object-Oriented Design，OOD)
- 面向对象编程(Object-Oriented Programming，OOP)
- 面向对象数据库管理系统(Object-Oriented Database Management Systems，OODBMS)，等等

这些软件开发方法有着专用的语言，例如 Eiffel、Smalltalk、C++、Objective C、Java、Python、CLOS 等。从这些语言和方法中又派生出了对应的库和工具，例如标准模板库(Standard Template Library，STL)、统一建模语言(Unified Modeling Language，UML)、通用对象请求代理体系结构(Common Object Request Broker Architecture，CORBA)、Rational Rose、Together、Eclipse 等。这些语言、库和工具集与用在逻辑编程或使用结构化编程技术的软件开发中所使用的那些语言、库和工具集完全不同。表 3-2 列出了一些经常使用的软件开发方法。

表 3-2

软件开发方法	描 述	活动/阶段
敏捷开发	软件以较短时间间隔进行开发；每个间隔或迭代是一个小型的开发项目，完成软件功能的一部分	• 计划编制 • 需求分析 • 设计 • 编码 • 测试 • 文档编制
边做边改	开发构建软件，然后不断进行多次修改，直到客户对产品满意为止	• 构建第一版 • 修改，直到用户满意 • 维护阶段 • 退出
极限编程	基于增量模型的模型，开发人员告诉客户每个特性的实现时间以及成本，然后客户选择在后续开发中包含哪些特性	• 规格说明 • 设计 • 实现/整合 • 交付
增量	软件以增量式或逐步构建；按照模块来设计、实现和整合；对于每次构建，模块都进行装配来满足特定功能	• 需求分析 • 规格说明 • 结构设计 • 循环构建 • 维护 • 退出

(续表)

软件开发方法	描　　述	活动/阶段
面向对象	基于对系统中对象的确认的软件开发，这是一种自底向上的方法	• 需求分析 • OO 分析 • OO 设计 • 实现/整合 • 操作模式 • 维护
快速原型	根据模型快速创建系统原型；原型被接受之后，SDLC 继续；在每个阶段中，都要同客户进行交互，以测试或验证产品的进展	• 快速原型 • 规格说明 • 设计 • 实现 • 整合 • 维护 • 退出
螺旋式	螺旋式模型类似于增量式模型，但更重视每个阶段的风险分析和验证；包括这些阶段的每一遍会迭代发生(称为螺旋式)	• 计划编制 • 风险分析 • 评价 • 工程
结构化	一种自顶向下的软件开发方法，系统根据功能进行迭代的分解，从最高层抽象到最底层功能	• 需求分析 • 设计 • 实现 • 测试 • 部署
瀑布	最常见且经典的模型，也被称做线性顺序模型；在这个模型中，每个阶段必须完全完成之后才能够进入下一个阶段	• 需求分析 • 规格说明 • 设计 • 实现 • 整合 • 维护 • 退出

选择软件开发方法本身就是一种挑战，而且一旦选择了一种方法，也就确定了将要使用的工具集和技术。方法的选择对在软件中如何实现多处理和多线程有关键影响。有多核编程需求的开发人员在选择特定方法时需要小心谨慎，因为该方法所对应的工具集和技术可能会将开发人员限制在难使用、易错的多处理或多线程实现上。过程驱动的软件方法在

处理多线程和多处理时，与对象驱动或数据驱动的方法有着巨大的差别，面向对象编程方法为开发人员提供的选项集与逻辑编程中可用的选项集也有很大差别。一旦软件开发已经开始，而且人力资源和工具就位，则很难在中途或在软件部署之后改变范型。在一些软件开发过程中，甚至在理解软件需求或规格说明之前，工具集、语言和库就已经被选好了。这样经常会导致软件实现被迫使用选中的语言和工具集，无论它们是否合适。理解 SDLC 中的各种活动与多核编程之间的关系非常重要，我们将在本书中强调这种关系。

尽管可能在何种方法是最好的这一问题上存在争议，但是所有方法中都有一些共同的基本活动。这些活动在每个软件开发中都以某种形式存在，无论软件的大小。例如，每种方法都有得到项目需求和规格说明的过程，每种方法都有集中于在实际编码之前设计解决方案的活动。另一个基本活动的实例是软件在发布之前的测试。这些通用活动可能会在各种软件开发方法中出现在不同的位置，并以不同的次数出现。如果您在 SDLC 的适当活动中应付并发带来的 10 个挑战，那么写出正确且可靠的程序的几率就能够大幅提高。如果您要开发的软件要求并发，那么表 3-1 中某些部分的每个活动都会受到影响。我们在这里聚焦于 SDLC，因为我们提倡多核应用程序开发的软件工程方法，而不是一些用来令“可感知多核(multicore-aware)”应用程序快速投入市场的反复试验，或创建特别插件的方法。尽管有方法可以隐藏和抽象出并行编程和多线程的一些复杂性，但是并没有真正的捷径。部署可利用 CMP 的稳定、正确、可扩展软件应用程序，要求拥有扎实的软件工程以及对 SDLC 有效且可靠的理解。

确定何时、何地以及如何将多处理和多线程合并到软件开发中是本书的主要议题，这给我们提出了两个问题：

(1) 如何知道软件应用何时需要多核编程？

(2) 如何知道在软件的哪个部分进行多核编程？

这些问题都和本章前面部分列出的 10 个挑战中的第一个相关，即这两个问题都集中于软件分解的挑战。

3.3.1 挑战 1：软件分解

多线程或多处理的需要或机会多数情况下是在分解活动中被发现的。分解是将一个问题或解决方案划分成基本部分的过程。有时候这些部分按照逻辑范围(即搜索排序、计算、输入、输出等)进行分组，在其他情况下，这些部分按照逻辑资源(处理器、数据库、通信等)进行分组。软件解决方案的分解实际上是工作分解结构(Work Breakdown Structure，WBS)或架构制品(Architectural Artifacts，AA)。

- WBS 决定了软件块的功能
- AA 决定了软件解决方案被划分为什么概念或事物

WBS 通常反映过程化分解，而 AA 代表面向对象分解。遗憾的是，并没有现成的方法来识别软件解决方案中适合 WBS 或 AA 的部分。

注意：

我们可以说模型驱动的分解是最实际的方法之一，而且本书中将对模型进行大量介绍。

除非已经对问题和解决方案进行了分解，否则您无法讨论在软件解决方案的哪里使用线程或是否使用同时执行的进程。问题和解决方案分解通常在 SDLC 的分析和设计活动中进行。

成功的分解是成功的软件开发的主要因素之一。另一方面，差的或不适当的问题和解决方案分解几乎必然导致产生失败的软件。

1. 分解的实例

为了说明分解的含义，我们将使用一个简单的问题实例，即在客人假期到达之前给房屋刷漆。当然，我们将利用这个机会来使用最新的软件自动油漆机(software-automated painter)。下面给出了给房屋刷漆这个问题的可能的分解，以及解决方案。

分解 1

问题可被分解为：

- 决定油漆的颜色和类型
- 获得油漆和刷漆工具
- 决定首先给哪个房间刷漆
- 确定使用哪种类型的自动油漆机
- 选择在一周中的哪些天进行刷漆
- 确定何时开始刷漆

这是对房屋进行刷漆这个问题的一种分解。

解决方案的分解可能为：

- 选择并购买同家具搭配的油漆
- 使用朋友的软件自动油漆机
- 让自动油漆机从房屋的前面开始，然后向后进行刷漆
- 只在工作日(周一～周五)的早上 6 点～下午 1 点期间进行刷漆
- 从下一个工作日开始进行刷漆

您将很快看到分解带来的部分挑战。首先，您可能会注意到问题分解通常不仅一种方式。当您看到问题和解决方案的分解时，可能在脑海里会有截然不同的步骤。实际上，对于在客人假期到来之前对房屋进行刷漆这个问题，您可以选择一个完全不同的方法：

分解 2

下面是另一种问题分解方法：

- 确定最好是使用墙纸的房间(不需要刷漆)
- 找出可以增加窗户以减少面积的墙
- 验证是否可以通过清洁墙面来替代刷漆

- 确认朋友可以赠送多少油漆
- 看看可以将哪面墙除去，而不是刷漆
- 获得客人的旅游计划
- 获得可以免费试用 30 天的软件自动油漆机样品

您可以使用来自第一种方法的解决方案分解，或者可以选择一个完全不同的解决方案分解：

- 策略性地使用灯光来展示最好看的墙
- 当灯光不合适时，可以明智地使用镜子
- 在外墙上增加窗户
- 如果镜子不方便，则使用墙纸
- 使用尽可能多的油漆机样品
- 推迟客人的到达时间，直到软件自动油漆机完成工作

您可以观察到的第二点就是分解可能会不完全或不合适。理想情况下，分解的基本部分应当共同地表示最初的问题或解决方案。就像将拼图再恢复到一起那样。如果各个碎片不能共同表示整体，则分解是不完全的。这意味着您没有抓住整个问题，而且将无法得到完整的解决方案。在给房屋刷漆的问题中，确定刷漆的成本是最初问题的一部分吗？您无法从问题的陈述中得到答案，因此不知道分解 1 和分解 2 是否完全。另一方面，显然您需要在客人到达之前为房屋刷漆。分解 1 中的问题和解决方案根本没有提到客人的到达，因此分解 1 中的解决方案是否可被接受就不那么清楚。尽管分解 2 的确尝试处理客人的到达，但是问题和解决方案都倾向于寻找途径来不给房屋刷漆或给尽可能少的墙壁刷漆。这种分解可能是不适当的。它可能反映了对最初问题的目的没有很好地理解。

在分解 1 的解决方案中，你还可以看到建议使用一台软件自动油漆机，而且分解 2 中则选择了使用多台油漆机。因此分解 2 不仅倾向于减少需要刷漆的墙的数目，而且尝试尽可能快地完成工作。恰当的分解是基于顺序模型的应用程序所面临的主要挑战，更是并行处理中需要解决的问题。有一些软件工具和库可以帮助开发人员实现分解，然而，这个处理本身仍然是问题解决和设计活动的一部分。直到您正确地对问题和解决方案进行了分解，才能清楚多线程或多处理应用程序是怎样的。

在前面，我们将分解定义为把问题或解决方案划分为基本部分的过程。但什么是问题或解决方案的基本部分呢？答案取决于您使用何种模型来表示问题和解决方案。软件分解面临的挑战之一就是有多种方式来表示问题及其解决方案。有些人可能会争论，既然没有一种正确的方式来分解问题或解决方案，那么您应当选择哪种分解？另一个挑战在于要确保分解是完全、适当且正确的，但是怎样才能够知道分解是否正确？在某些情况下，问题不在于在多个可能甚至冲突的 WBS 中进行选择，而是在于提出任何一种分解的难度，这可能会由最初问题的复杂性决定。

在任何软件开发中，分解问题都是非常重要的。当需要配置并行处理工具或技术时尤其重要。但是 WBS 或 AA 的选择依赖于模型。在任何需要进行分解的时候，都会有一个

或多个模型可供选择。在分解 1 和分解 2 这两个选择的表面下，所隐藏的是一个假定与共享模型。

2. 找出正确的模型

模型是构成分解的材料。使得分解带来的挑战更加复杂的是选择合适的模型来恰当地表示问题、任务或解决方案。

什么是模型？

软件开发是将概念、思想、工作模式、规则、算法或公式转换为能够被计算机运行或处理的指令集和数据的过程。它是严重依赖于对模型的使用的过程。在本书中，模型是一些实际过程、事物、概念或思想的按比例的人工表示。按比例表示是一些较小的、较简单的、较容易处理的表示，它们是过程、事物、概念或思想的近似。

模型的主要功能是模仿、描述或复制一些现实世界里的实体的行为和特性。模型是一个包含足够信息来进行分析或做出决定的替代者。模型对现实世界实体表示地越好，则分解、WBS 或 AA 将会更加自然。

多核编程的挑战之一就是选择恰当的问题和解决方案模型。在并行编程、多处理、多线程方面，如果使用恰当的模型则可以成功，如果选择错误的模型则会失败。如何将应用程序划分为并发执行部分？这个问题通常可以通过对解决方案模型或问题模型的分析得到答案。选择的模型会影响可用的分解选择。

例如，在房屋刷漆的问题中，我们假定了房屋的常见模型。

- 模型 1：房屋是有着墙、房间和窗框的事物。您应当为这个模型增加天花板、门、过道、地板、楼梯扶栏、屋顶等。

这可能是您在考虑房屋刷漆问题时立刻会映入脑海的模型。但是在分解时，对于指定的问题或解决方案往往有着多个模型。您可以为房屋选择一个完全不同的模型。

- 模型 2：房屋是按照平方米度量的住所，有着一个入口和一个出口，为两人或多人构成的家庭提供合理的空间，保护人们不受天气的影响，提供少量的隐密空间，而且为所有居民提供休息的场所。

尽管模型 2 在特定场景下很好(例如，房屋选择)，但是对于房屋刷漆问题而言，模型 1 显得更有帮助。

这说明分解的概念同模型密切相关。实际上，分解遵从模型所使用的部分、过程和结构。尤其是，分解受到问题和解决方案底层模型的限制。因此，分解的挑战也部分是模型选择的挑战。

过程式模型还是声明式模型？

在本章的前面部分，我们介绍了解决方案的 WBS 和 AA。WBS 将问题或解决方案分成需要执行的任务或需要完成的工作。另一方面，AA 将问题或解决方案分为人、地点、事物、想法的集合。表 3-3 显示了 WBS 和 AA 方法的区别。

表 3-3

属　　性	WBS	AA
定义	将问题或解决方案分成需要执行的任务或需要完成的工作	将问题或解决方案分为人、地点、事物和想法的集合
使用的模型	使用任务驱动模型	使用面向对象或关系驱动模型
分解模型	使用过程式模型	使用声明式模型
扩展性/复杂性	不能够很好地扩展，不适用于复杂系统	可以很好地扩展，适用于复杂系统

如您所见，WBS 将问题和解决方案分解为动作的集合，AA 将问题和解决方案分解为事物的集合。WBS 使用任务驱动模型，AA 使用面向对象或关系驱动模型。最重要的是，WBS 分解是按照过程式模型，AA 按照声明式模型。

在包含多处理或并行编程的软件设计中，也许最重要、最关键的决定就是是否使用过程式模型或声明式模型，或者将两者联合使用。过程式模型和声明式模型在方法、技术、设计和实现上存在的基本区别非常巨大，以至于要求完全不同的计算机编程范型[Saraswat，1993]。在某些情况下，这些范型是被非常不同的语言所支持的。其他情况下，这些范型是通过以非常不同的方式使用常见的语言来支持的。随着向单芯片多处理器(成百上千个)的发展趋势，过程式模型以及对应的 WBS 将不能够扩展。在同步和通信的复杂性之下，它们将无能为力，因此不得不使用声明式模型以及分解。

转换到声明式模型是一个主要的挑战，因为过程式模型以及 WBS 是基于在第 3.1 节中讨论的传统顺序编程方法。计算的顺序模型有着最常用的语言、工具和技术。直到不久之前，计算的顺序模型仍然是大学、学院、职业学校等最常讲授的模型。尽管声明式模型已经出现了一段时间，但是并未得到广泛的使用或讲授(OOP 除外)。表 3-4 显示了声明式编程范型的列表，以及该范型下一些经常使用的语言。

表 3-4

声明式编程范型	常 用 语 言
面向对象	C++ Java Eiffel SmallTalk Python
函数式	C++ Haskell Lisp ML Scheme

(续表)

声明式编程范型	常 用 语 言
并发约束	C++ Prolog 基于 Prolog 的语言
约束	Prolog 基于 Prolog 的语言

C++程序员在 CMP 开发中的优势之一就在于 C++支持多范型开发。不像 Java 语言中一切都是对象那样，C++支持面向对象、参数化和命令式(过程式)编程。由于 C++在表达性和灵活性上的强大，它可以用于实现来自表 3-4 中列出的所有编程范型的想法。关于声明式并行化方法与过程式并行化方法的对比将是全书重要的内容。对于多数问题和解决方案，挑战在于学习使用正确的工具来完成工作。

每次刷一个房间还是一次刷全部房间?

在本章前面，对于在客人假期到达之前为房屋刷漆的问题，已经给出了两个问题和解决方案的 WBS。分解 1 选择使用一台软件自动油漆机，分解 2 选择使用尽可能多的软件自动油漆机。注意在分解 1 的解决方案中指定了软件自动油漆机从房屋的前部开始，向后进行。然而，分解 2 没有提到多台软件自动油漆机如何进行工作。

每次涂一个房间是否明智？还是最好同时刷尽可能多的房间？如果您的确试图同时涂多个房间，需要多个漆刷吗？每台软件自动油漆机将共享漆刷、桶或高度调整设备吗？需要多少台自动油漆机，每个房间一台吗？还是每面墙一台呢？各台自动油漆机之间是否需要通信？如果一些油漆机在其他油漆机之前完成工作，它们应当去帮助那些没有完成的油漆机吗？如果不是所有房间，只是部分房间可以同时被刷漆，会怎样？如果可以同时刷漆的房间一天一天地变化，会怎样？软件自动油漆机之间如何通信和协作？如果从意识到房屋需要刷漆到客人到达之间的时间足够长，一台自动油漆机可以轻松完成工作，会怎样？无论如何都要使用多台油漆机吗？如何对房屋刷漆问题进行声明式分解？这些都是您在进行问题和解决方案分解时将会遇到的各类问题。而且就像您很快将看到的那样，分解问题带来了其他挑战。

注意：

由于设计决策或用户规格说明，要求软件解决方案或架构使用多处理或线程是一回事，而经过对现有一款软件或软件设计进行深入分析，提示出否则不会被要求，并且不会被包括的多处理的机会是另一回事，两者并不相同。在本书中，我们将注意力集中于根据设计决策的结果或用户规格说明要求多处理的软件解决方案或架构上。

3.3.2　挑战 2：任务间通信

如果有两个任务 A 和 B，它们并发执行，而且其中一个任务依赖于另一个任务的信息，

那么两个任务之间需要有某种途径来进行信息通信。如果任务需要共享一些资源(即文件、内存、对象、设备等)而且该资源任意时刻只支持一个任务对它进行访问，那么任务需要有某些方法来传递资源可用或资源被占用的信息。如果涉及的任务是独立的操作系统进程，那么任务之间的通信被称为进程间通信(Interprocess Communication，IPC)。

进程有着独立的地址空间。这些独立的地址空间用于将进程相互隔离。这种隔离是一种有用的保护机制，而且这种保护有时候是选择多进程而不是多线程的理由。操作系统单独保存进程的资源。这意味着如果您拥有两个进程，分别为 A 和 B，那么进程 A 中声明的数据对 B 不可见。此外，进程 A 中发生的事件也不为 B 所知，反之亦然。如果进程 A 和 B 共同完成某些任务，则必须在两个进程间进行显式的信息和事件通信。每个进程的数据和事件对于该进程是本地的。第 5 章将详细讨论进程的组成，但是目前我们将使用图 3-1 来显示两个进程的基本布局以及它们的资源。

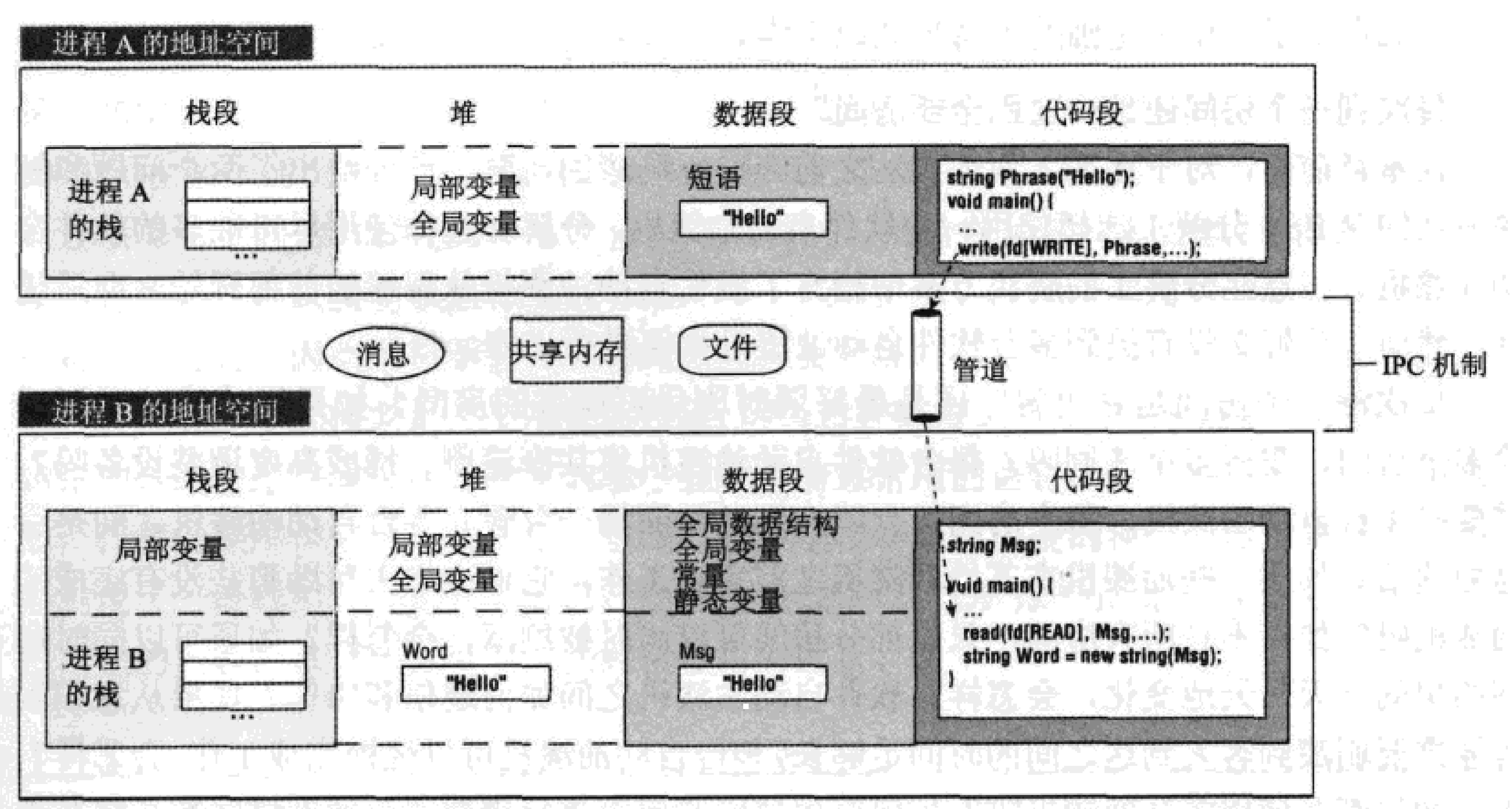


图 3-1

注意在图 3-1 中，进程 A 的资源同进程 B 的资源是隔离的。进程拥有代码段、数据段和栈段。进程还可能在堆中分配有一些空间。进程所拥有的数据一般位于数据段、栈段和进程自己动态分配的内存中。这些数据受到操作系统的保护，不会被其他进程影响。进程为了访问其他进程的数据，必须使用特殊的 IPC 机制。与此类似，如果进程希望知道另一个进程的代码段中发生的事情，则必须在进程间建立一种通信方式，这也需要操作系统级 IPC 的协助。多处理的程序带来的主要挑战之一就是管理 IPC。随着应用程序中涉及的进程数目的增加，IPC 机制的数目也随之增加。更多的进程几乎总意味更多的 IPC 机制和应用。在很多实例中，协调多个处理器之间的通信是真正的挑战。

1. 管理 IPC 机制

POSIX 规范支持 6 种用于完成进程间通信的基本机制：

- 具有加锁和解锁能力的文件
- 管道(无名管道、命名管道，也被称为 FIFO(First-In，First-Out))
- 共享内存
- POSIX 消息队列
- Socket
- 信号量

表 3-5 包含用于进程的 POSIX IPC 机制的简要描述。

表 3-5

POSIX 进程间通信	描　述
命令行参数	可以在调用 exec 或 spawn 函数时传递给子进程
环境变量/文件描述符	子进程可以收到父进程的环境数据和文件描述符的副本；父进程可以设置变量，子进程可以读取那些值；父进程可以打开文件并向前移动文件指针的位置，然后子进程可以使用相同的偏移来访问文件
具有加锁功能的文件	用于在两个进程间传送数据；加锁功能用于对两个进程对文件的访问进行同步
管道	在相关或无关进程之间的一种通信信道形式；通常使用文件读写功能进行访问
共享内存	被多个进程访问的、位于它们地址空间以外的内存块
消息队列	可以在进程之间共享的消息链表
信号量	用于同步线程或进程对资源的同步访问的变量
Socket	进程间利用端口和 IP 地址的双向通信链路

以上这些 IPC 机制，每一种都有其优势、弱点、缺陷和陷阱，软件设计人员和开发人员为了促进两个或多个进程间可靠且高效通信，必须对它们进行管理。我们将在第 5 章详细介绍这些内容，在这里，仅提出一些使用这些 IPC 机制的主要挑战：

- 必须正确地创建它们，否则应用程序会失败。
- 它们的使用要求适当的用户权限。
- 它们的使用要求适当的文件权限。
- 有些情况下，它们有着严苛的命名约定。
- 它们不是对象友好的(即它们使用底层字符表示)。
- 它们必须被恰当地释放，否则会导致锁住和资源泄露。
- 在它们的使用中，源进程和目标进程不容易识别。
- 软件的初始配置可能非常棘手，因为不是兼容所有的环境。

- 机制对发送和接收的数据的大小是否正确非常敏感。
- 错误的数据类型或大小可能导致锁住和失败。
- 清空(flushing)机制并不总是非常直观。
- 部分机制在使用用户实用工具时不可见。
- 依据类型，一个进程所能够访问的 IPC 的数目可能会受限。

这些 IPC 机制可用作并发执行的进程间的桥梁。桥梁有时候是双向的，有时候是单向的。例如，POSIX 消息队列在创建时可能允许进程读消息和写消息。有些进程可能为了读取消息打开队列，有些进程则是为了写入，而有些进程则既读取又写入。软件开发人员必须记录哪个进程为了何种目的打开了哪个队列。如果进程是为了只读访问打开队列，然后又试图对队列进行写入，则会导致问题。如果涉及的并发执行的任务较少，则很容易管理。然而，一旦并行执行的任务数达到了十几个，则管理 IPC 机制就是一种挑战了。对于本章前面部分提到的分解的过程式模型，情况尤其如此。即使双向或单向通信要求得到了适当地管理，您仍将面对在两个或多个进程间传递的信息的完整性问题。消息传递模式可能会遇到问题。例如传输中断(部分执行)、错乱消息、丢失消息、错误消息、错误接收者、错误发送者、过长的消息、过短的消息、迟到的消息、早到的消息等。

需要注意的是，这些特别的通信挑战仅仅是对进程而言的，线程不会面对这些问题。这是因为每个进程拥有自己的地址空间，而且 IPC 机制被用作进程之间的通信桥梁。而线程则共享相同的地址空间。同时执行的线程共享相同的数据段、栈段和代码段。图 3-2 显示了线程组成的简要概况，并可将它同进程进行比较。

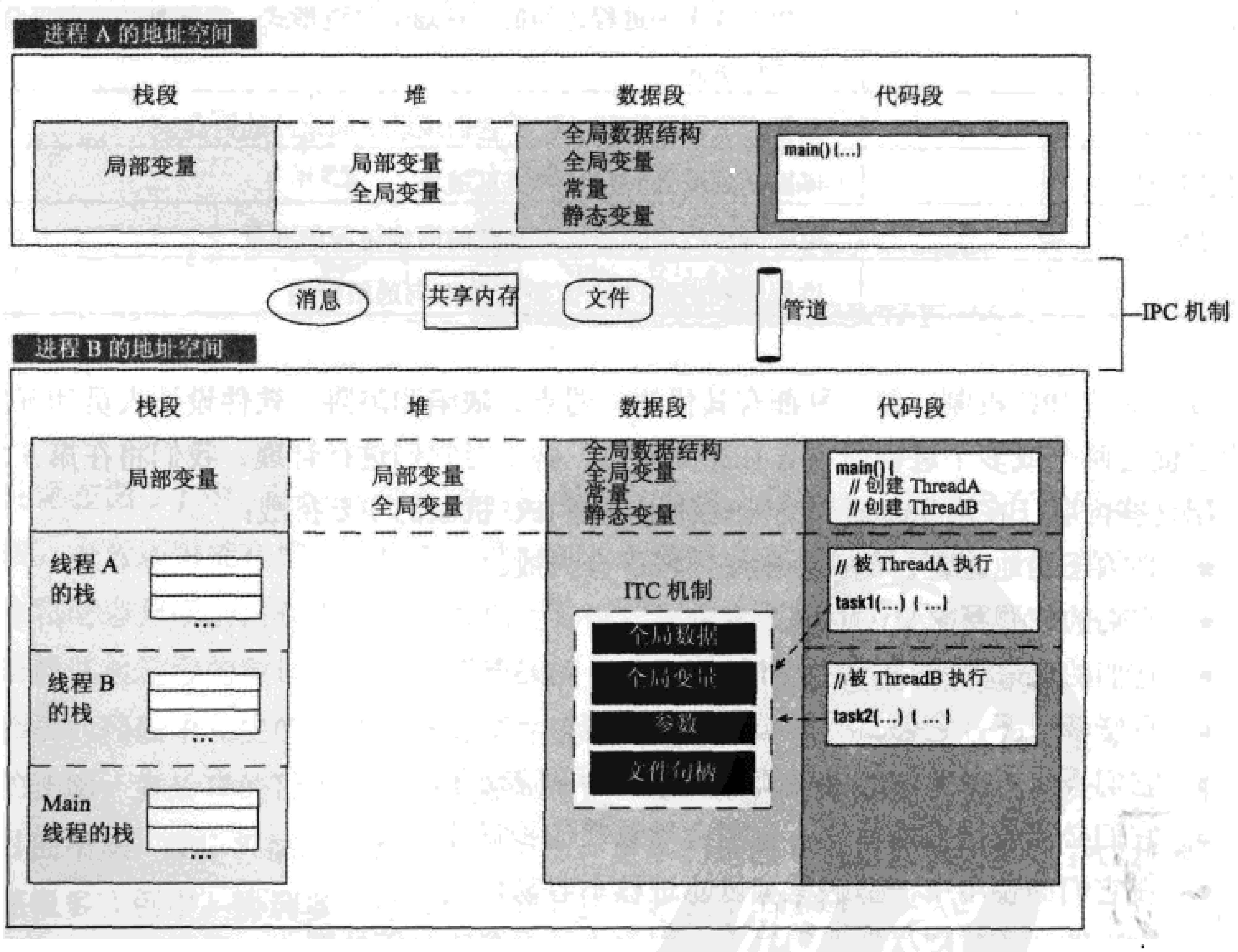


图 3-2

两个或多个线程(有时候也被称作轻量级进程)之间的通信要容易些，因为线程不需要担心地址空间边界。这意味着程序中的每个线程可以轻易地向函数传递参数、从函数得到返回值、访问全局数据。如图 3-2 所示，相同进程的线程访问进程被存储在数据段的全局变量中。这里我们着重强调进程间通信(Interprocess Communication，IPC)机制与线程间通信(Interthread Communication，ITC)机制之间的基本区别：IPC 机制位于进程的地址空间之外，而 ITC 机制位于进程的地址空间之内。这并不是说线程就没有它们自身所面临的挑战，只是它们不会产生必须穿越地址空间的问题。

2. 油漆机将如何通信

在本章前面部分中，在客人假期到达之前对房屋刷漆的问题实例中，分解 2 使用尽可能多的软件自动油漆机。但是如果油漆机位于不同的房间，那么它们之间将如何通信？它们是否需要相互通信？如果它们共享一个桶，多少台油漆机可以同时访问它？当桶需要再装满时会发生什么事？油漆机是等待桶被装满还是在桶被装满的同时进行其他工作？如果多台油漆机同时需要高级设备，将会怎样？您需要增加更多的高级设备吗？多少台高级设备才够用？一台油漆机如何令另一台油漆机知道有高级设备可用？这些类型的问题会增加多进程和多线程的工作的麻烦。如果它们没有在 SDLC 的适当阶段得到处理，那么要求多进程或多线程的应用程序会处于危险之中，即使在安装之前也是如此。如果通信没有得到很好的设计，那么会轻易出现死锁、无限延期以及其他数据竞争状况。数据竞争、死锁、无限延期是多线程或多进程面临的最严重的问题，而且它们是挑战 3 的核心问题。

3.3.3　挑战 3：多个任务或 agent 对数据或资源的并发访问

当并发执行的指令、任务、应用程序被要求共享数据、设备或其他资源时，3 种常见问题会发生。这些问题会导致数据损坏、任务停滞或 agent 冻结。这 3 种问题是：

- 数据竞争
- 死锁
- 无限延期

1. 问题 1：数据竞争

如果两个或多个任务试图同时更改共享数据，而且数据的最终值取决于哪个任务先到达，那么就出现了竞争条件(race condition)。当两个或多个任务试图同时更新相同数据资源时，竞争条件被称作数据竞争。由竞争条件所决定的信息是不可靠的，这既适用于输入值，也适用于输出值。在房屋刷漆问题中，使用分解 2 时，漆桶的状态就是一种数据竞争。

考虑下面的漆桶更新过程的描述：每个油漆机的 bucket 例程包含从漆桶中得到 1 加仑或更多油漆的 get 指令。read 指令用于读取漆桶的状态，而 write 指令用于在油漆被取走之后更新漆桶的状态。因此，油漆机的 bucket 进程可能类似于如下形式：

```
Read  Bucket Status into Total
```

```
If N is <= Total
   Get N Gallons of Paint
   Total = Total - N
   Write Total to Bucket Status
end if
```

在这个过程中，每台油漆机一旦从桶中取走油漆后，就基于桶的之前的油漆状态来记录还剩下多少油漆。假定有两台油漆机，Painter A 和 Painter B，同时从漆桶中取油漆，Painter A 启动 bucket 例程，然后取走 20 加仑油漆。在 Painter A 更新漆桶的状态之前，Painter B 从漆桶取走 10 加仑的油漆，然后首先更新状态。这是可能的，因为 Painter B 需要的油漆少于 Painter A，因此会首先结束并首先更新 Bucket Status。Painter B 在更新漆桶状态时将使用错误的数量。因为尽管 Painter A 首先开始移走 20 加仑油漆，但是更新状态仍未保存。此外，Painter B 已经在 Painter A 的更新之前读取了 Bucket Status 的值。而且负责基于 Bucket Status 填满漆桶的软件自动油漆机，恰好在 Painter A 更新状态之前、Painter B 更新之后查看状态。该监视器基于漆桶状态所做出的任何决定都会是不正确的。Painter A 完全不知道 Painter B 的活动和用 10 加仑作为值对 Bucket Status 的更新。此时，作为 Painter A 和 Painter B 移走油漆的结果，漆桶实际上已经空了，但是 Bucket Status 变量的值为 10。图 3-3 显示了 Painter A 和 Painter B 的数据竞争场景。

在多线程或多进程的环境中，因为操作系统调度线程和进程的方式，上述情形完全是可能的。这可以归因于时钟周期。既然在这个场景中，将油漆从桶中取出的过程和更新漆桶状态的过程是独立的，这两个事件在被操作系统调度时也是独立的。哪个油漆机首先获得漆桶状态就取决于操作系统调度、处理器状态、延迟和机会了。这种情形造成了竞争条件。在这种环境下，漆桶的实际状态是怎样的？

区分共享的、可修改的资源和只读的资源是非常重要的。如果多个线程或进程试图同时访问不能够被修改的(即只读内存或常量对象)资源，则无需担心数据竞争。与此类似，如果多个线程或进程只是试图同时读取一个数据块，也不会发生数据竞争。竞争条件出现的前提是，资源必须是可修改的，而且多个线程或进程必须同时访问资源，其中至少一个线程或进程试图修改资源。

当任务并发共享一个可修改资源时，必须对任务的访问应用规则和策略。例如，对于 bucket 例程，您可能必须部署互斥读互斥写(Exclusive Read Exclusive Write，EREW)策略，这样当一台油漆机启动例程后，其他油漆机必须等待，直到整个例程完成。或者您可能必须使用一台油漆机，其工作是专门用来更新漆桶状态。如果有多台油漆机同时需要使用漆桶，那么必须拦住这些请求，然后根据某些规则进行组织，确保每次只有一台油漆机被授权。但是如果您设立了漆桶状态，而且每次只能有一台油漆机存取油漆，那么不是违背了拥有多线程或多进程的目的了吗？共享的漆桶状态会不会成为性能瓶颈？识别数据竞争条件是很微妙的，因为并发运行的线程或进程的精确执行顺序是由当前操作系统的其他部分决定的。它依赖于其他正在执行的不相关的进程或线程。即使有着多个处理器的计算机也会在操作系统管理的线程或进程数目大于可用处理器数目时，被迫使用多道程序设计。这

意味着操作系统将会在必要时挂起和重新开始线程或进程。而且，将哪个进程或线程挂起以及何时挂起通常是由操作系统决定的。这在一组进程或线程执行时引入了一定程度的不确定性。

注意：

在第 5 章、第 6 章和第 7 章中将会介绍关于这一点的更多内容。在这里，只是希望您能够注意到数据竞争是多核编程中的潜在陷阱这个事实。

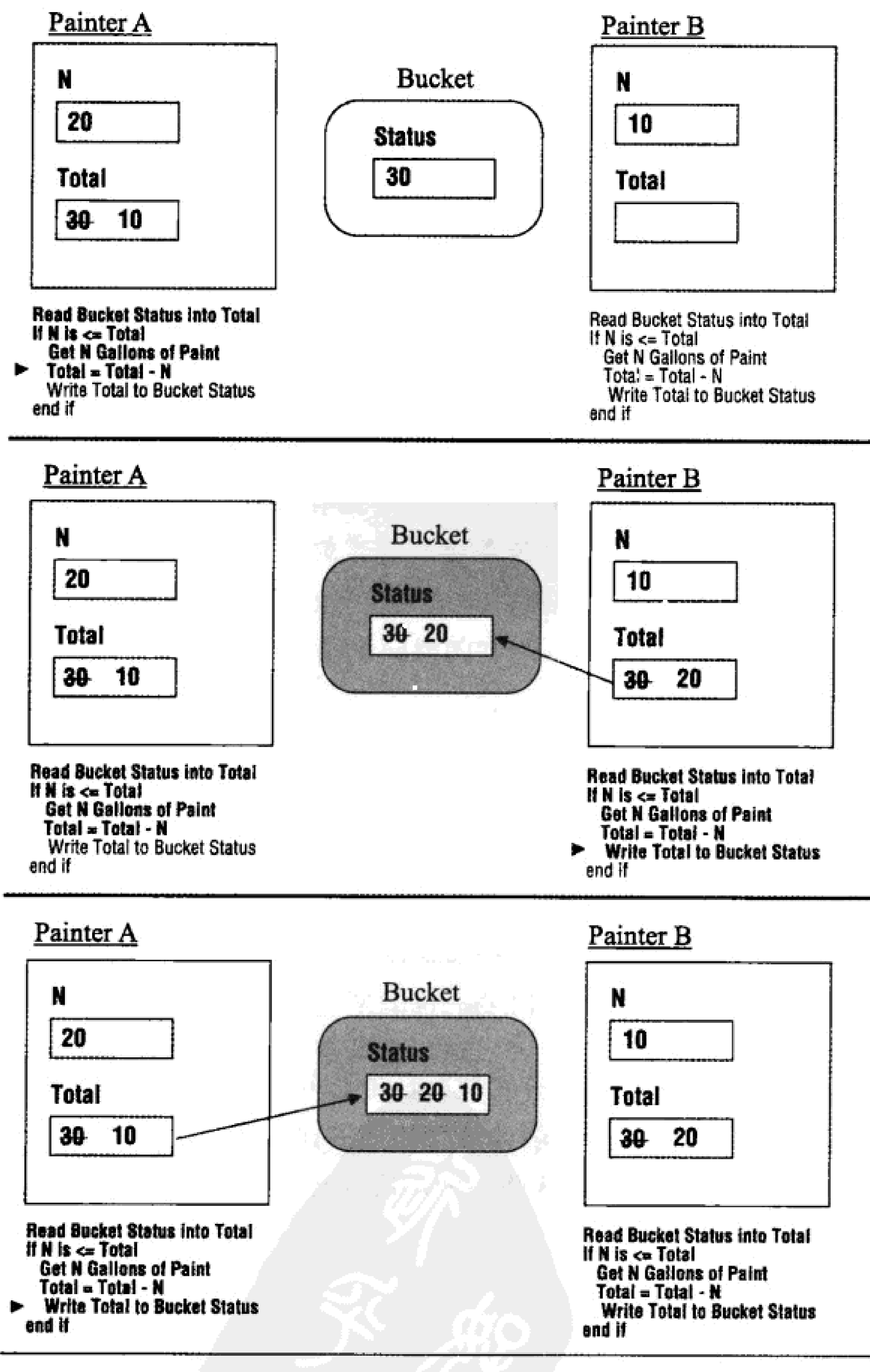


图 3-3

2. 问题 2：死锁

死锁是另外一种等待类型的陷阱。为了示范死锁的实例，假定在房屋刷漆问题中有 3 台油漆机(A、B 和 C)使用两个装有不同颜色油漆的漆桶(1 和 2)。A 负责更新两个漆桶的状态。Painter A、Painter B 和 Painter C 可以并行执行它们的工作。然而，Painter B 和 Painter C 每次只能够使用一个漆桶。Painter A 保证漆桶的状态按照先来先服务的原则进行更新。假定 Painter B 对 Bucket 1 进行排他访问，Painter C 对 Bucket 2 进行排他访问。然而，Painter B 需要访问 Bucket 2 来完成刷漆，而且 Painter C 需要访问 Bucket 1 来完成它的处理。Painter B 决定持有 Bucket 1 并等待 Painter C 释放 Bucket 2，同时 Painter C 决定持有 Bucket 2 并等待 Painter B 释放 Bucket 1。Painter B 和 Painter C 便处于一种致命的僵局，也被称作死锁。图 3-4 显示了 Painter B 和 Painter C 之间的死锁局面。

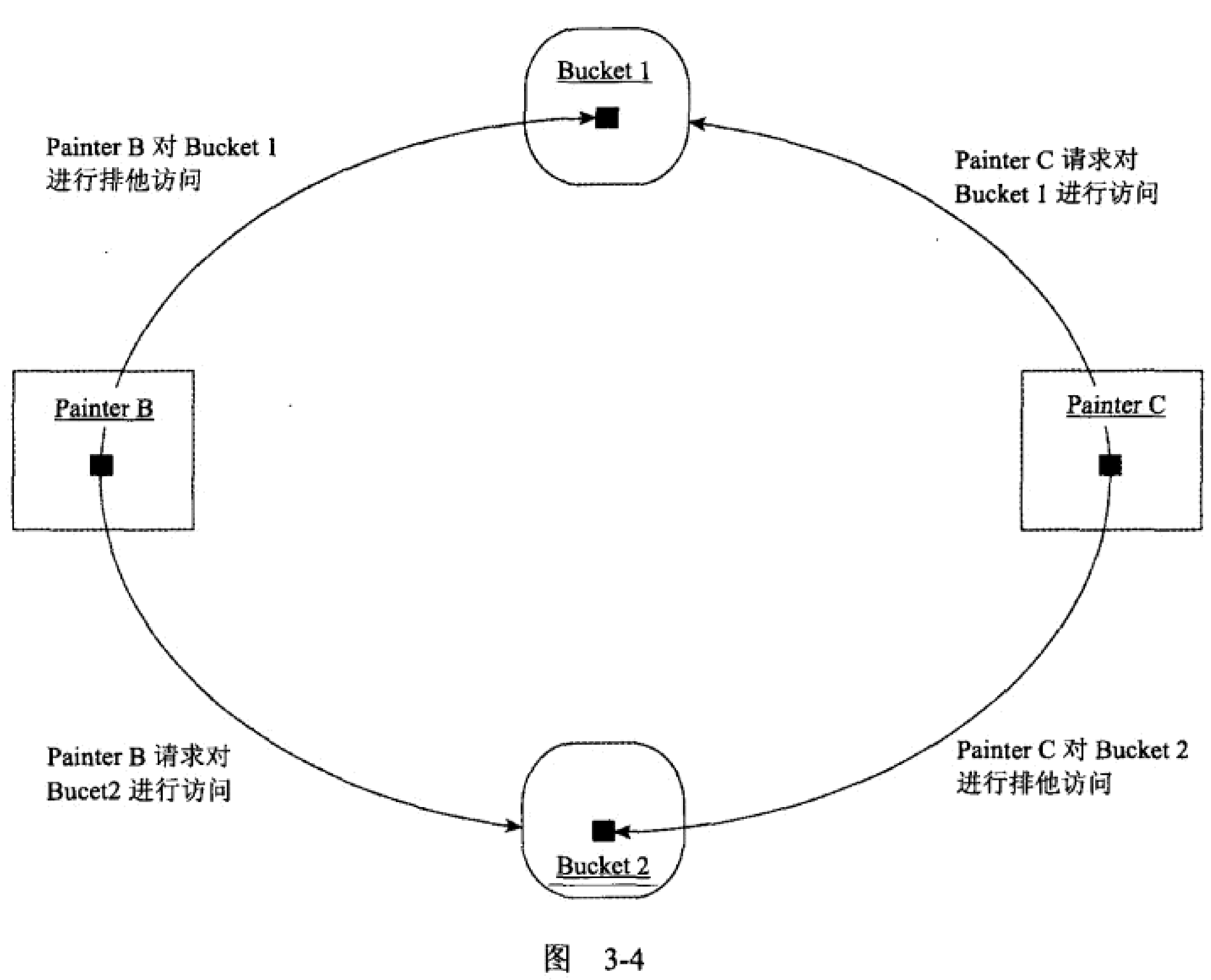


图 3-4

图 3-4 所显示的死锁的形式要求并发执行任务访问一些共享可修改的资源，这些资源是它们必须相互等待对方结束使用之后才能够访问的。在图 3-4 中，共享资源是 Bucket 1 和 Bucket 2。两台 Painter 有权使用这些漆桶。恰好某一时刻不是一台 Painter 在两个漆桶取油漆，而是每台 Painter 各自占据了一个漆桶。由于 Painter B 在得到 Bucket 2 之前不能够释放 Bucket 1，而且 Painter C 在得到 Bucket 1 之前不能够释放 Bucket 2，软件自动刷漆过程就锁住了。注意 Painter B 和 Painter C 可以令另外一个任务无限延期(下一节中将详细讨论)。例如，Painter A 负责更新漆桶状态，它也在等待 Painter B 或 Painter C 发出一个写或读的请求。如果其他任务在等待对 Bucket 1 或 Bucket 2 的访问，而且 Painter B 和 Painter

C 陷于死锁之中，那么这些任务等待永远不会发生的情形。

如果您尝试协调并发执行的任务，死锁和无限延期是两个必须克服的障碍。更糟糕的是，是否发生了这些状况也并不总是很明确。涉及的任务可能正在等待合法事件的发生。也有可能 Task A 占用的时间只不过比预期的长了一些。因此识别死锁也为多核编程提出了另一个挑战。识别死锁(死锁检测)、防止死锁、避免死锁所包括的步骤在使用多进程或多线程的应用程序中非常关键。

注意：

我们将在第 7 章中讨论检测、防止和避免死锁的技术。

3. 问题 3：无限延期

对一个或多个任务进行时间安排，让它们一直等待，直到一些事件或状况发生是很有技巧性的。首先，事件或状况必须及时地发生，其次，它要求任务间仔细设立通信。如果一个或多个任务在执行之前在等待通信，而通信没有到来、到来太迟、不完整，那么这些任务可能将永远不会执行。与此类似，如果您预计的最终会发生的事件或状况实际上永远不会发生，那么您已经挂起的任务将永远处于等待状态。如果一个或多个任务被挂起，等待一些永远不会发生的状况或事件，就被称作无限延期。在软件自动刷漆解决方案中，如果 Painter B 在得到 Bucket 2 之前不释放 Bucket 1，Painter C 在得到 Bucket 1 之前不释放 Bucket 2，Painter B 和 Painter C 在完成之前不会请求 Bucket Status 更新，而且 Painter A 等待 Painter B 和 Painter C 的请求，那么所有涉及的油漆机都将被无限延期。

数据竞争、死锁、无限延期都是同步问题的实例。这些类型的问题采取的形式都是两个或多个任务同时对相同的资源进行竞争。资源可以是软件或硬件。

- 软件资源包括文件、文件中的记录、记录中的字段、共享内存、程序变量、管道、socket 和函数。
- 硬件资源包括中断、物理端口和外设(打印机、调制解调器、显示器、存储设备和多媒体设备)。

这些资源中的一部分很容易共享，例如磁盘或文件。其他资源要求对访问进行仔细地管理，例如中断。当两个或多个任务试图同时改变相同资源的状态时，可能会导致数据丢失、不正确的程序结果、系统失效、设备损坏等。

3.3.4　挑战 4：识别并发执行的任务之间的关系

数据竞争、死锁和无限延期的同步问题，有时候会被建立线程或进程之间的正确执行关系的挑战所放大。

1. 基本的同步关系

在一个进程内的任意两个线程或一个应用程序内的任意两个进程之间，存在 4 种基本同步关系。表 3-6 列出了这 4 种基本同步关系以及对它们的描述。

表 3-6

同 步 关 系	描　　述
Start-to-start(SS)	一个任务不能够启动，除非另一个任务启动
Finish-to-start(FS)	一个任务不能够结束，除非另一个任务启动
Start-to-finish(SF)	一个任务不能够启动，除非另一个任务结束
Finish-to-finish(FF)	一个任务不能够结束，除非另一个任务结束

这样，如果有两个任务，假定为 A 和 B：

- 在 start-to-start(SS)关系中，直到 Task A 启动，Task B 才能够启动。Task B 可以在 Task A 启动的同时或之后启动，但是不会在 Task A 启动之前启动。
- 在 finish-to-start(FS)关系中，直到 Task A 结束或完成特定操作，Task B 才能够启动。例如，如果 Task B 正在从 Task A 进行写入的 POSIX 消息队列中进行读取，那么 Task A 需要至少在队列中写入一个元素，然后 Task B 可以处理它。如果 Task B 需要在 Task A 必须排序的列表上执行二分搜索，则 Task B 应当与 Task A 进行同步，这样，Task B 只有在 Task A 完成排序之后才会开始进行搜索。finish-to-start 关系通常表示存在信息依赖。
- start-to-finish(SF)同步关系是指直到 Task B 结束，Task A 才能够开始。这种关系在父进程从子进程请求进程间通信(IPC)时，或进程或线程在为父亲提供了需要的信息或事件之后递归调用自身时经常发生。
- 最后，finish-to-finish(FF)同步关系是指直到 Task B 结束，Task A 才可以结束。Task A 可以在 Task B 结束之后结束，但 Task A 不允许在 Task B 之前结束。

图 3-5 显示了这 4 种同步关系。SS、FS、SF 和 FF 同步关系在多线程或多进程应用程序中都存在。有时候这些关系非常细微，在 SDLC 的各种活动中发现所有这些关系会非常复杂。有些任务间的关系非常明显，而其他的则是隐含的，需要仔细的检查。除了同步关系，还有计时考虑(timing consideration)。

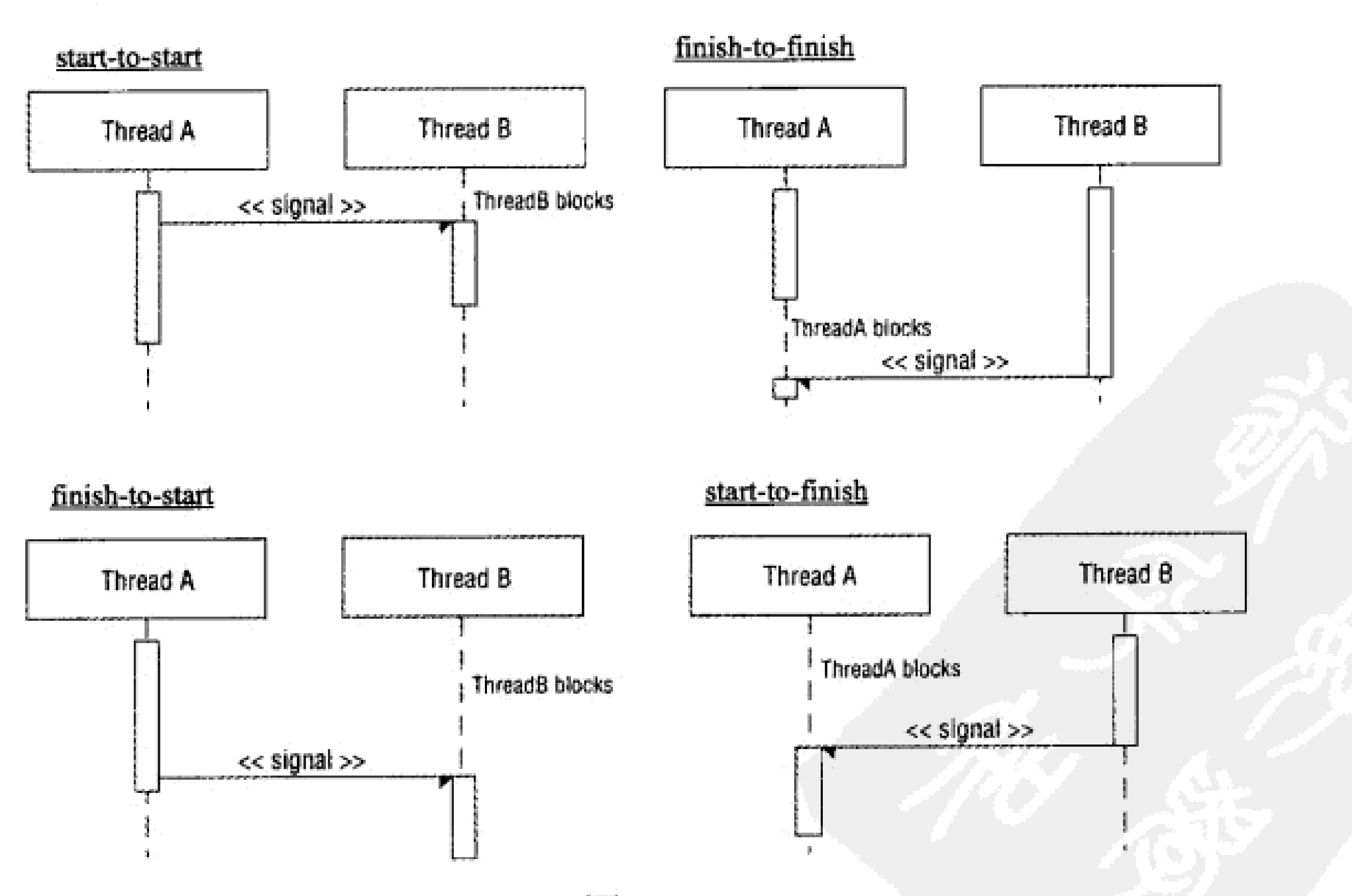


图 3-5

2. 计时考虑

如果在一个房间里有多台软件自动油漆机，那么对天花板刷漆的油漆机是否应当最先开始工作？对墙壁进行刷漆的油漆机是否应当等待对天花板刷漆的油漆机结束工作之后再开始？它们应当同时开始吗？或者只要它们同时结束您就很满意？也许对墙壁进行刷漆的油漆机应当在开始之前先等待 15 分钟。这不仅仅是使用尽可能多的油漆机的问题，而且还有油漆机之间的某种同步关系的问题。

有时候同步关系需要增加上特定计时信息。这意味着在涉及同步关系时，需要考虑时间和事件。例如，如果您让 Task A 和 Task B 并发执行，Task A 执行通信任务，Task B 是关注超时的监视器，Task A 和 Task B 使用 start-to-start 关系进行同步。Task B 直到 Task A 开始通信任务之后才需要开始检查超时状况。然而，一旦 Task A 开始其通信任务，而且在很多毫秒中都没有任何活动，则 Task B 可能发出超时消息。这种情况下，Task B 在发出超时消息前在使用迟滞时间(lag time)。迟滞时间用于进一步定义同步关系。迟滞时间要求在同步关系的规范中加上一个时间要素。例如，您可以说 Task A 和 Task B 之间是 start-to-start 同步，而且有着 Task B 必须在 Task A 启动之后等待 10ns 的额外要求。

这些类型的计时考虑是在 SDLC 的适当活动中密切关注多处理或多线程需求的另一个主要原因。同时，同步关系和计时考虑的实现很大程度上将受到是选择过程式模型还是声明式模型进行分解的影响。

3.3.5　挑战 5：控制任务之间的资源争夺

当多个任务对相同资源的使用进行竞争时，就会发生资源竞争。这个话题将在第 7 章中进行介绍。

3.3.6　挑战 6：需要多少个进程或线程

存在一个临界点，此时管理多个处理器的开销会超过并行在速度改进和其他优势方面所带来的收益。老的格言“处理器多多益善”并不是正确的。线程间的通信或处理器间的同步都有一定的成本。同步的复杂性或处理器间通信的数量可能会要求大量的计算，使得用于完成工作的任务的性能会受到负面影响。在这些情况下，基于顺序模型编写程序会更加有效。例如，如果您希望对包含 100 个数字的列表进行排序，可以试着将 100 个元素的列表分为每组 10 个元素的 10 个组，并行对每个组进行排序，然后将 10 个组合并成一个排好序的列表。但是将列表分组、各组同列表间的通信，以及在避免数据竞争的前提下将结果合并为一个组，所需要花费的努力和时间会超过使用顺序方法对数字进行排序所需要的。另一方面，如果有几个 TB(terabyte)的数字需要排序，那么并行方法会更有效率。

问题在于程序应当被分为多少个进程、任务或线程？是否对任意给定并行程序都存在最优处理器数目？在什么时候为计算池增加更多的处理器或计算机会使得速度减慢，而不是速度提高？结果证明该数量根据程序的不同会发生变化。一些科学模拟可能需要几千个

处理器，而对于一些商业应用，几百个处理器就足够了。对于一些客户端服务器配置而言，8 个处理器是最优的，9 个处理器可能就会使得服务器性能变得很差。

您可能在达到最优的处理器或计算机数据之前，就已经达到了软件进程的限制。类似地，您可能在达到最优的并发执行任务数之前，就在硬件方面发现收益递减。

理想情况下，问题的模型分解或解决方案的模型分解能够帮助决定必需拥有多少个线程或进程。然而，模型的实际实现会引入很多复杂性和开销，从而必须选择一个新的模型。本章前面的部分介绍了多个进程间 IPC 机制的一些挑战，这些 IPC 机制必须进行同步。如果两个或多个任务之间的通信没有进行适当地同步，那么数据竞争条件、死锁和无限延迟就会出现在软件中。

尽管我们强烈提倡技术、工具、语言和软件库应当遵从分解模型而不是相反，但是必须考虑实现解决方案分解模型的复杂性。您可能会在包含多线程和多处理的应用程序中面临停机问题(halting problem)的很多变体。当应用程序中协同工作的任务数目增加，相互依赖的复杂性也会随之增加，这样会导致非常脆弱且多变的软件实现。当多个任务协同提供某些问题的解决方案时，如果一个或多个任务失效会怎样呢？程序应当停止，还是应当将工作重新分配？如果只有两个并发执行的任务，这不成问题。但是随着应用程序中线程或进程数目和相关性的增加，解决任务失效的难度会发生指数级增加。

3.3.7 挑战 7 和挑战 8：寻找可靠的、可重现的调试和测试

当测试顺序程序时，可以逐步跟踪程序的逻辑。如果以相同的数据启动，而且可以确保系统处于相同的状态，则输出或逻辑流程是可预测的。可以通过以必要的状态启动程序、使用恰当的输入、逐步跟踪程序逻辑来找出软件中的 bug。在顺序模型中的测试和调试依赖于程序对于给定的特定输入、初始状态和当前状态之间的可预测性。

在多进程和多线程环境中，情况就不一样了。由于操作系统调度策略、计算机的动态负载、处理器时间片、进程和线程优先级、通信延迟、执行延迟和并行环境中的随机因素，使得我们很难重现并行或并发任务的精确上下文。任务使用不同的数据集以及任务处理的数据的语义变化等问题都增加了工作量。为了在测试和调试中重现精确的环境状态，操作系统需要重新创建当时的每个任务，而且必须已知处理器调度状态。虚拟内存的状态以及上下文切换也要被精确重现，还要对中断和信号状态进行重建。在某些情况下，网络流甚至也需要被重建！数据必须被设置并恢复为其初始值和初始状态。测试和调试工具也会影响精确的环境，这导致了调试或测试的不确定性。不确定性的状况是指对于一些初始状态，其最终状态不是明确的[Gries，Scheider，1993]。

根据我们的经验，除了最常见的那些之外，所有其他的多处理或多线程应用程序都有一定的不确定性。这意味着重建相同的事件顺序来测试或调试一个程序往往是不可能的。这些必须被重建的原因在于它们都可以帮助确定哪个线程或进程可以执行，以及它们可以在哪个处理器上执行。正在执行的线程和进程的特定混合可能是导致死锁、无限延期、数据竞争或其他问题的原因。尽管这些问题也影响顺序编程，但是它们不会违背顺序模型的

假设。顺序模型中存在的那种可预测性在并发编程中并不存在。这迫使开发人员采用新的策略来测试和调试程序，还要求开发人员找到新的方法来证明程序的正确性。在选择声明式模型和过程式模型时，人们会从完全不同的角度来观察测试和调试所涉及的问题。程序正确性对涉及复杂并行处理模式的程序而言，可能是一个非常难以捉摸的概念。

这种非确定性对于跨平台开发也有影响。操作系统对进程和线程的处理在不同操作系统环境中，如 Linux、Solaris、Darwin 等，会存在区别。某些系统令线程具有高、中、低优先级的选择，某些系统则有用户定义的优先级，某些系统有关键任务优先级、实时优先级、普通优先级、后台优先级等。操作系统可以有不同类型的调度器、实现(implementation)不同的 IPC 机制、实现(implementation)不同的内核线程和用户线程。

寻找正确的调试器和分析器

很多常用的调试器和分析器(profiler)都是基于单处理器计算机开发的。多核应用开发要求调试器和分析器能够看到所有可用的物理和逻辑处理器。调试器应当尽可能地深入到操作系统工作负荷中。调试器需要对内核进程和系统调用有着清楚地了解，需要能够附加(attach)或卸下(detach)一个进程或线程，还需要能够看到操作系统可能会放入一个进程或线程的所有的处理器状态或线程状态。用于多线程或多进程应用程序的优质调试器应当能够启动和停止线程和进程，它应当能够检查线程栈和堆。

3.3.8　挑战 9：与拥有多进程组件的设计的相关人员进行沟通

您还面临如何在文档编制中精确地捕捉并行设计的挑战。您必须能够描述 WBS 以及任务、对象、进程、线程之间的同步和通信。设计人员必须能够与开发人员进行有效沟通。开发人员必须能够与系统的维护和管理人员进行沟通。理想情况下，这应当使用一种所有有关人员都可以使用的标准标记和表示。然而，找到一种能够被广泛理解且清楚地表示这些系统的多范型本质的文档语言是非常困难的。为了达到这个目的，我们选择了统一建模语言(Unified Modeling Language，UML)。表 3-7 列出了对多线程和并行程序有帮助的 UML 图。

表 3-7

UML 图	描　述
结构/架构图	
组件图	显示系统中一组代码模块(包)之间的依赖和组织的图
部署图	显示系统中处理节点、硬件、软件组件的运行时结构的图
行为图	
状态/并发状态图	显示对象在响应系统中的事件时的变换序列图；如果是并发状态图，这些转换可以在相同的时间间隔发生
顺序图	一种显示了发送并接收消息的对象的组织结构的交互图
协作图	显示了消息的时间顺序的交互图
活动图	显示了从一个活动到另一个活动的流图，类似于流程图，但是可以显示多个对象的活动以及多个并行活动的流程

表 3-7 中列出的只是 UML 中可用的图类型的一个子集。但是这些图均立即可用于并发设计之中。尤其是 UML 中的活动图、部署图和状态图在沟通并行处理行为时非常有用。

注意：

由于 UML 是交流面向对象和面向 agent 设计的实际标准，所以我们在本书中将使用它。附录 A 包含了在这些图中使用的标记和符号的描述和解释。

3.3.9 挑战 10：在 C++中实现多处理和多线程

使用 C++的软件开发人员如何能够利用新的 CMP？如何在 C++中实现多进程？C++语言并没有为并行化包含任何关键字原语(primitive)。C++ ISO 标准实际上没有对多线程做出规定。没有办法在语言中指定两条或多条语句应当并行执行。其他语言使用内置并行化作为卖点。C++语言的发明者 Bjarne Stroustrup 则另有主张。在 Stroustrup 看来：

> 设计并发支持库可方便且高效地解决内置并发支持问题。通过库，可以支持很多并发模型，同单独的内置并发模型相比，这样做可以为需要这些不同模型的用户提供更好的服务。我希望这能够成为多数人努力的方向，而且当多个并发支持库被使用时，所产生的可移植性问题能够通过一小层接口类得到处理[Stroustrup，1994]。

此外，Stroustrup 说："我建议用 C++中的库来表示并行性，而不是将其作为通用语言特性"。我们发现 Stroupstrup 的主张以及将并行性作为库来实现的建议是最实际的选择。正是由于可用于并行和分布式编程的高质量的库的存在，才使得本书能够面市。我们用于增强 C++的库，实现了并行和分布式编程的国家标准和国际标准，而且被世界各地上千名 C++程序员所使用。

3.4 C++开发人员必须学习新的库

尽管有实现并行性的特殊版本的 C++，我们还是要给出如何使用符合 ISO 标准的 C++来实现并行性的方法。如同在上一节结束位置所暗示的那样，并行性的库方法是最灵活的。系统库和用户级库可用来在 C++中支持并行性。系统库是那些由操作系统环境提供的库。例如，POSIX 线程库是一组可以和 C++共同使用以支持并行性的系统调用。POSIX(Portable Operating System Interface，可移植操作系统接口)线程是新的 Single Unix Specification 的一部分。POSIX 线程被包含在 IEEE Std. 1003.1-2001 中。Single Unix Specification 是由 Open Group 发起，由 Austin Common Standards Revision Group 完成。根据 Open Group 的主张，Single Unix Specification 是：

- 为软件开发人员提供一组所有 Unix 系统都支持的 API
- 将焦点从不兼容的 Unix 系统产品实现转移到符合一组 API 上
- 是 Unix 系统实践的共同核心部分的法律和实质标准

- 对程序员和应用程序源代码均有着基本的可移植性目标

Single Unix Specification Version 3 被包含在 IEEE Std. 1003.1-2001 和 Open Group Base Specification Issue 6 中。IEEE POSIX 标准现在是 Single Unix Specification 的正式部分，反之亦然。目前只有一个关于可移植操作系统接口的国际标准。C++开发人员受益于这个标准包含创建线程和进程的 API。除了指令级并行外，将程序划分为多个线程或进程是用 C++实现并行化的唯一途径。新的标准提供了实现并行化的工具。开发人员可以使用：

- POSIX 线程(也被称为 pthread)
- POSIX spawn 函数
- exec( )函数系列

它们都被系统 API 调用和系统库所支持。如果一个操作系统符合 Single UNIX Specification Version 3，那么 C++开发人员就可以使用这些 API。

注意：

这些 API 将在第 5 章、第 6 章和第 7 章中讨论。它们将被用于本书的很多例子中。此外，附录 C 和附录 D 中包含了 POSIX 标准中的相关部分。

3.5　处理器架构的挑战

我们在第 2 章中了解了 4 种有影响的多核架构。它们是 Opteron、Cell、UltraSparc T1 和 Intel Core 2。尽管这些处理器都提供了多核能力，但是它们使用了不同的架构。这些不同的架构体现在不同的编译器开关和指示集上。为了最大限度利用这些不同的架构，开发人员必须熟悉特定编译器和链接器特性。本书将着眼于 GNU C++编译器、Intel C/C++编译器、Sun C/C++编译器对多核的支持。每种编译器都提供了支持多线程和多进程的开关与指示集。在某些情况下(例如 Cell 处理器)，需要使用多种类型的编译器来生成一个可执行程序。危险性在于利用特定架构会使得软件不可移植。尽管不是所有的应用程序都被要求可以移植，但是大部分用户都有这个要求。如何在不使用损害可移植性的特性的前提下，最大程度利用多核架构呢？这是您在开发多核应用程序时必须解决的另一个问题。

3.6　小结

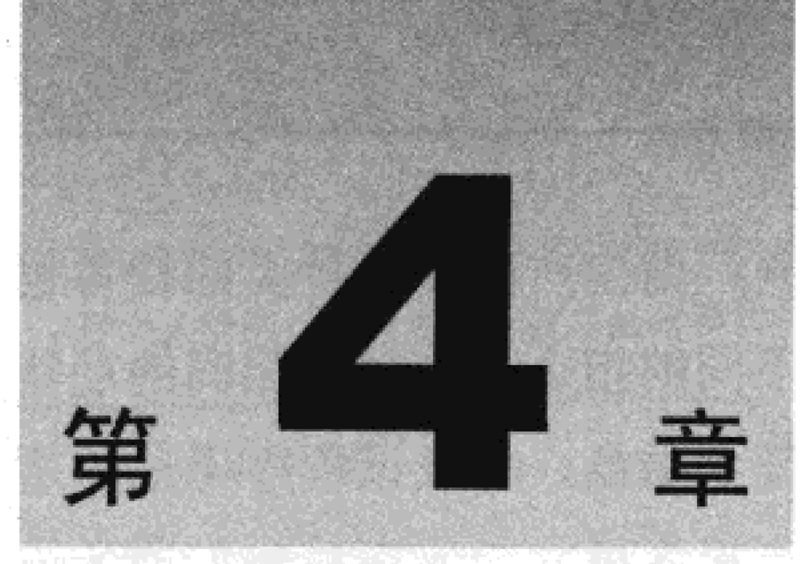
并行和分布式编程在多个方面提出了挑战，必须采用新的软件设计和架构。顺序编程模型中的许多基本假设在并行编程模型中都无法使用。开发人员面临本章列出的许多并发挑战。一些关键点包括：

- 4 种主要的协同问题(即数据竞争、无限延期、死锁和通信同步)是要求并发的程序遇到的主要障碍。

- 当需求中包括并行性或分布时，软件开发生命周期(SDLC)的每个方面都会受到影响，从初始设计到测试和文档编制。并行化和多处理的机会将会在 SDLC 的各种活动中确定。软件开发人员对多核编程与 SDLC 之间的关系的理解非常重要。
- 要在多进程或并行编程的软件设计中需要做出的最重要、最关键的决定可能就是使用过程式模型还是声明式模型了。过程式模型和声明式模型在方法、技术、设计和实现之间的基本差别非常明显，以至于它们要求完全不同的计算机编程范型。

本书给出了很多这类问题的解决方法。除了结构化方法，我们还可利用 C++的多范型能力来提供管理并行和分布式程序的复杂性的技术。

趋势是在大多数情况下，多处理器计算机将会在商业、学校和政府中取代单处理器配置。就像本章中所介绍的那样，为了利用多处理器环境，软件开发人员必须扩展已经具有的工具和技术。要求多核或并行程序的软件项目对那些只熟悉顺序编程模型的人提出了挑战。在隐藏和抽象出并行编程和多线程的复杂性方面，没有真正的捷径。部署能够利用 CMP 的稳定的、正确的、可扩展的软件应用程序，要求拥有可靠的软件工程以及对 SDLC 的充分理解。下一章将会介绍作为软件开发人员，如何做才能够解决本章所介绍的这些问题。



操作系统的任务

到目前为止，我们已经描述了多核编程的一些主要挑战，并简要介绍了多线程、多处理和多道程序设计的一些概念。第 2 章介绍了 Opteron、Cell、Duo Core 2、UltraSparc T1 等多核架构。这些芯片代表 4 种有影响但是非常不同的多核架构设计方法。我们解释了有时候为了利用芯片多处理器(CMP)的特定特性，必须使用针对特定硬件的编译器开关。但是我们很少提及操作系统在多核程序和应用软件的设计、开发和执行中的作用。本章将会介绍这方面的内容，具体包括：

- 提供操作系统的概览
- 开发人员与多处理器的接口
- 探究线程、进程和处理器是如何通过操作系统关联起来的
- 检查操作系统应用程序接口(API)和系统调用如何同 C++联合使用进行多核编程和应用程序开发
- 解释操作系统如何起到多处理器的看门人(gatekeeper)的作用
- 讨论如何使用可移植操作系统接口(POSIX)标准来设计和实现能够在所有主要硬件和操作系统平台上工作的多核应用程序

4.1 操作系统扮演什么角色

我们将集中研究如何把操作系统作为开发工具。本书将从系统程序员和应用程序员的视角来讨论多核编程。从这些视角出发，操作系统的作用主要有以下两个。

- 软件接口：为计算机的硬件资源提供一致且定义良好的接口
- 资源管理：管理硬件资源以及其他正在执行的应用软件、作业和程序

4.1.1 提供一致的接口

在操作系统出现之前，程序员必须熟悉特定指令集以及每个设备的特性。显卡、磁盘驱动、打印机、键盘等，所有这些都有特定而且不同的指令集和命令集。不仅对每个设备

的访问是不同的，而且由不同厂商生产的相同种类的设备也有着不同的指令集和新特性。这使得程序员经常必须使用不同的指令集来重写相同的功能。例如，如果一名开发人员已经编写了一个程序将文件整理到磁盘，那么该程序无法重用在另一家厂商的磁盘上，除非更新所有的设备 id、指令集、设备模式等来适应新厂商的设备。除了唯一的指令集，每种连接到计算机的设备都拥有特定的地址、端口或中断。在操作系统出现之前，程序员必须知道设备的物理地址、端口或中断，然后才能够访问该设备。因此，程序中包含设备 id、硬件地址、端口号和中断。实际上，程序员必须为程序所访问的每块硬件编写一个设备驱动。程序和软件的移植是不可能的。

操作系统的概念改变了这一切。操作系统为程序员提供了到类似设备的通用接口。操作系统为设备封装了内部结构，例如显卡、声卡、键盘、显示器、磁盘驱动器、打印机等。操作系统为程序员在开发人员的程序和连接到计算机的硬件资源之间提供了两个软件层，从而不用强迫程序员使用奇怪的设备特定指令。这些层被称作应用程序接口(Application Program Interface，API)和系统程序接口(System Program Interface，SPI)。直接处理硬件资源以及所有特性现在成为了操作系统的工作，这样程序员只需要使用简化的 API 和 SPI，操作系统将处理所有的特定设备的转换。

4.1.2 管理硬件资源和其他应用软件

除了为开发人员提供 API 和 SPI，操作系统将控制程序的进程或线程访问处理器、内存、I/O 端口、中断和存储器。在多数工作站环境和服务器环境中，在任意时刻都有多个程序被执行或等待执行。由于处理器的数目以及内存的数量是有限的，操作系统必须决定哪个程序使用哪个处理器、使用多长时间和何时使用。操作系统决定进程或进程集可以占有多少内存以及占用多长时间。对于那些太大以至于无法在主存中存放的程序，操作系统将管理切换软件片段来执行相关过程。操作系统为进程分配硬件资源，这样它可以保护一个进程的资源不会被另一个进程访问或侵犯。通常，操作系统管理计算机中所有的硬件资源。除了管理硬件资源，它还调度并管理进程和线程。

4.1.3 开发人员与操作系统的交互

无论您是否在多核开发中使用类库、高级函数库、应用框架，操作系统都将起到处理器、内存、文件系统等连接到计算机的设备的看门人的作用。这意味着类库、高级函数库或应用框架中包含的多线程或多处理功能仍需要经过操作系统 API 或 SPI。图 4-1 显示了开发人员视角中的软件层和操作系统。

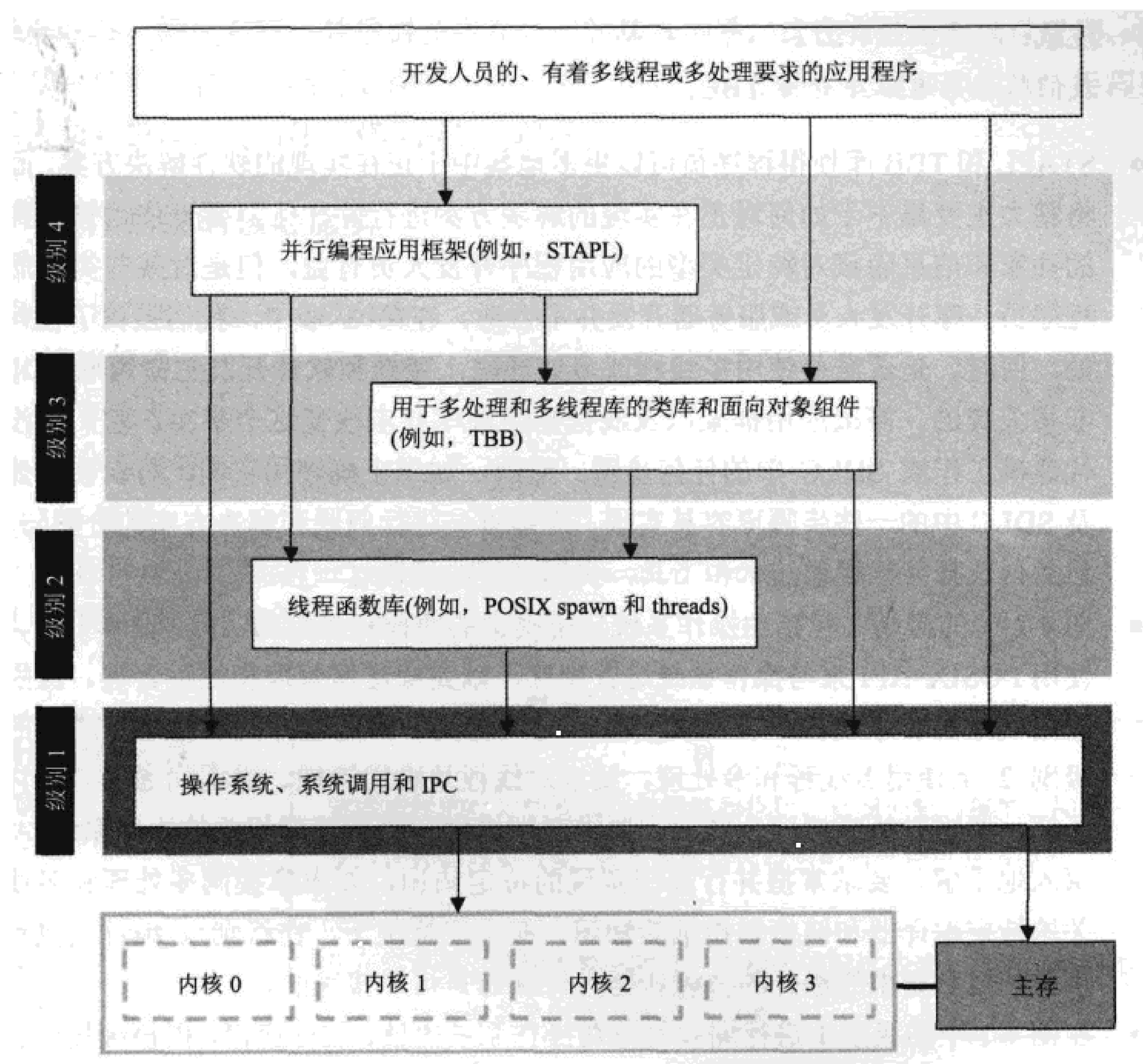


图 4-1

图 4-1 显示了可用于为应用软件提供多线程或多处理功能的软件层。注意图 4-1 中每个软件层的级别。在图 4-1 中，级别越低，开发人员需要控制的、负责正确使用的、必须了解的并行编程机制越详细。级别越低，为了正确实现软件所要求的设计和编程技巧越多。

- 级别 4 是最高的层。这一层为开发人员提供了与并行编程细节最高程度的隔离。标准模板适配并行库(Standard Template Adaptive Parallel Library，STAPL)是这种应用框架的一个实例。STAPL 是采用 C++开发并行程序的框架。类似 STAPL 的框架的目标是允许开发人员在应用软件中提供并行性，同时不需要担心并行编程所涉及的所有特定实现的相关问题。

  注意：
  我们将在第 8 章中详细介绍 STAPL。

- 图 4-1 中的级别 3 以模板库或类库为代表，例如 Intel Threading Building Blocks(TBB)库。Intel Threading Building Block 库是一组高级通用组件，封装了多处理和多线程的大部分细节。开发人员使用 TBB 来调用完成底层工作的高级算法模板和同步对象。

注意：

我们将在第 8 章中介绍 TBB。

- STAPL 和 TBB 库使得程序员可以更多地集中于正在实现的软件解决方案，而不是将精力主要集中于如何对正在实现的解决方案进行并行化。要记住尽管这种类型的抽象和信息隐藏对特定类型的应用程序开发人员有益，但是对某些类别的系统程序员、库开发人员或服务器开发并不合适。在第 3 章中，我们强调了在判断何处、何时、是否需要使用多线程或多处理时，建模和软件开发生命周期(SDLC)是非常关键的。高级应用框架以及线程构建块库并未改变这个事实。它们并没有取代建模工作或 SDLC 中的任何步骤。然而，如果正确使用它们，可以使得建模以及 SDLC 中的一些步骤更容易实现。应当首先进行问题和解决方案的分解与建模，然后再选择并行编程技术和工具，我们一直在强调这一点。
- 图 4-1 中的级别 2 包括由操作系统环境提供的线程和进程的 API。在本书中，我们使用 POSIX API 来与操作系统进行交互，以完成进程管理和线程管理。级别 2 为应用程序员和系统程序员提供了最大的灵活性，但是这种灵活性是有成本的。在级别 2 上使用多线程和多处理，要求对线程及进程管理、进程间通信有着详细地了解，要求熟练掌握同步技术，要求对同进程和线程管理相关的操作系统 API 有深入地了解，要求掌握并行算法实现的特定知识，要求掌握同多处理和多线程相关的特定编译器和链接器指示的知识。在某些情况下，在级别 2 编程的灵活性的价值超过所要求的额外的技能和努力，从而是可以接受的。
- 在图 4-1 中的级别 1 进行编程要求对操作系统内核、硬件接口、内核级接口有着尽可能多的了解。级别 1 上的编程会直接访问硬件，很少或没有软件阻碍。在级别 1 编程属于系统编程的范畴。

无论开发的应用程序使用的是级别 4 还是级别 3 的工具和技术，框架、模板和类库最终必须调用位于级别 2 和级别 1 的 API。级别 1 和级别 2 提供了 SPI 和 API 到操作系统的入口，而且不论如何，操作系统将控制对我们所感兴趣的多个内核的访问。

尽管不是每个编写软件以利用多核计算机的开发人员都会使用级别 1 和级别 2，但是在 SDLC 中，对这些层如何工作有基本的理解是很重要的。理解这些基础之所以重要，就是因为没有哪个库、框架或工具提供多数应用程序需要的所有服务。此外，这些工具中的大部分必须被混合与协调使用。实际上所有中等规模到大规模的应用软件都是使用多个库的联合来构建的。这些库并不总是线程安全(thread safe)或可感知多核(multicore-aware)的。当发生问题时，软件开发人员至少需要理解在进程和线程管理方面发生了什么事情。图 4-1 中在级别 3 和级别 4 中使用的高级工具必须被配置，配置过程要求人们对其工作原理有基本的理解。在某些情况下，混合和协调会导致产生一些需要解决的冲突。

除此之外，不是所有的高级工具在每种环境中都能够运行。例如，TBB 能够在很多基于 Intel 的处理器上运行，但是在所有主要的非 Intel 处理器上均不可用。为了使之可用或完全与您的平台兼容，可能会需要进行移植等。多线程和多处理应用程序的本质要求开发

人员理解软件、操作系统、处理器及内存之间的基本关系。这在有效处理调试过程、测试过程、最终软件部署中是绝对必须的。高质量、正确、可靠的多处理和多线程应用程序要求开发人员对操作系统的任务有着清楚的理解。

4.1.4　操作系统的核心服务

操作系统的核心服务可以分为：

- 进程管理
- 内存管理
- 文件系统管理
- I/O 管理
- 进程间通信管理器(Interprocess Communication Manager)

表 4-1 简要描述了这些核心服务。

表 4-1

操作系统核心服务	描　述
进程管理	管理进程的行为和资源，包括进程执行、资源分配与保护、同步
内存管理	管理进程的内存分配，包括如何将内存分配给进程以及当内存完全被利用时如何处理
文件系统管理	在存储设备上组织收集的数据，并提供访问这些设备上的数据的接口
I/O 管理	管理来自硬件设备的输入和输出请求
进程间通信管理器	管理进程间通信

尽管这些服务与所有应用程序开发人员都有关，但对于多线程或多处理应用程序的开发人员则更为明显。这是因为类似于进程调度或进程间通信等功能对于顺序处理应用程序往往是透明的。例如，在一个顺序处理应用程序中，操作系统只需要加载开发人员的程序。开发人员通常不关心操作系统如何将应用程序划分为进程、进程如何被调度、被调度的进程优先级如何，不需要担心共享内存违规，只要操作系统给应用程序足够的内存，则万事大吉。因为在顺序处理的应用程序中不存在并发执行的子任务，所以任务间通信和同步不构成问题。对于多处理和多线程应用程序，状况则完全不同。在第 3 章中，我们已经讨论了开发人员在开发这种类型的应用程序时所面临的挑战。根据该章讲述的内容，一些特别的、与操作系统服务相关的挑战是：

- 软件被分解为需要同时执行的指令或任务集
- 并行执行的两个或多个任务之间的通信
- 两个或多个指令或任务并发访问或更新数据
- 确定并发执行任务片之间的关系
- 当任务和资源之间存在多对一比例时控制资源竞争

- 确定需要并行执行的最优的或可接受的单元数目
- 记录并沟通包含多处理和多线程的软件设计
- 创建测试环境来仿真并行处理需求和条件
- 重建软件异常或错误以消除软件缺陷
- 为涉及多线程和多处理的组件实现操作系统和编译器接口

一旦软件设计过程确定应用程序最好分成两个或多个并发执行的任务，则操作系统的透明性立刻会成为问题。这是因为操作系统不具备自动任务分解的能力，但是需要由操作系统来负责创建和管理进程与线程。最终，并发执行的任务必须映射到进程、线程或两者的结合。当前的操作系统和编译器还不能够自动完成这种映射。必须有人在应用程序的并发需求和支持多处理和多线程的操作系统 API 和 SPI 之间担当联络官。如果您在使用来自图 4-1 的级别 3 和级别 4 中的技术和工具，那么操作系统的作用大部分是透明的(但明确存在)。如果您在图 4-1 中所示的级别 1 或级别 2 上进行工作，那么需要具备操作系统 API 和 SPI 的特定知识。

为了更仔细地观察操作系统在开发必须并发执行任务中的作用，您可以查看一个已经被分解为 4 个同时执行的任务的应用程序。使用当前的现代操作系统的 C++开发人员可以有 3 种基本选择来实现这些任务。任务可以被实现为：

- 进程
- 线程
- 进程和线程的组合

图 4-2 显示了 4-任务应用程序的基本分解选择的框图。

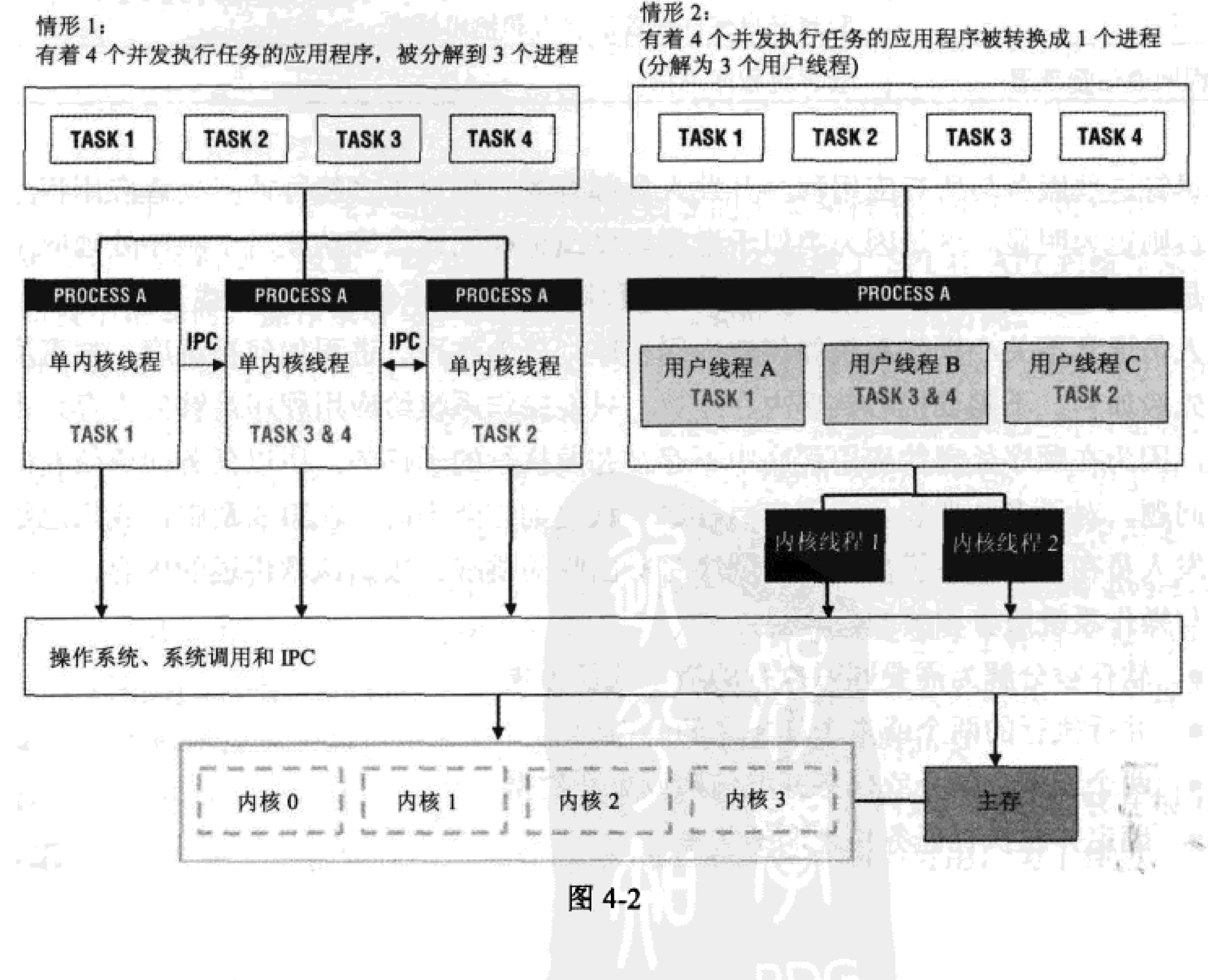


图 4-2

在图 4-2 的情形 1 中，应用程序被分为 4 个任务。这 4 个任务是通过 3 个操作系统进程来实现的。图 4-2 显示了应用程序将部署在 4 核计算机上。4 个任务通过 3 个进程来实现，意味着任务中的 3 个有可能同时在 3 个不同的处理器上实际执行，或者在任意数目的处理器上采用多道程序设计。如果采用多道程序设计，则操作系统在进程间快速切换，这样允许多个进程在指定的时间间隔内并发完成工作。尽管在某个时刻只有一个进程实际使用处理器，但是进程之间的切换速度非常快，一秒钟之内会有两个或多个进程在处理器上执行工作。尽管在图 4-2 中是 4 核计算机，我们实际上只能够同时执行 3 个任务。这是因为 4 个用户任务已经被映射到 3 个进程。操作系统只能够将进程或内核线程(轻量级进程)调度到处理器上执行。它不能够调度逻辑任务，除非它们已经被映射到进程或内核线程。即使 4 个内核都空闲，但是在情形 1 中，只有 3 个内核会被应用程序同时使用。任务 3 和任务 4 共享一个进程。情形 1 使用多处理，因为它通过将其任务指派给操作系统进程来利用操作系统的多处理器。系统指派进程到任意空闲内核。所以，如果应用程序被划分为进程，它可以利用系统能够并行运行的进程数目与处理器数目相同的事实。

1. 如何实现从任务到进程的转换

为了将用户任务映射到系统进程，您需要使用操作系统 API。在本例中，使用的是 posix_spawn( )函数。函数 posix_spawn( )用于创建一个新的操作系统进程。posix_spawn( )是操作系统的进程管理 API 的一部分。任何通过 posix_spawn( )关联到进程的任务可以被操作系统调度，与其他进程并行运行。注意在情形 1 中，Task 1 需要与 Task 3 和 Task 4 进行单向通信，而 Task 3 和 Task 4 需要与 Task 2 进行双向通信。这带来了关于同操作系统交互的另一个问题，即如何在并发执行的进程之间传递信息？有很多方法可以完成上述工作，但是所有这些方法都要求与操作系统 API 进行交互。在本例中，您可以使用 POSIX 消息队列。表 4-2 包含了 POSIX 消息队列函数的简要描述。它们是允许并发执行的进程传递信息的 POSIX API 函数的实例。

表 4-2

POSIX 消息队列函数	描　述
mq_open( )	通过消息队列描述符，建立进程和消息队列之间的联系
mq_close( )	解除消息队列描述符与它对应的消息队列之间的关联
mq_send( )	将消息加入到指定的消息队列
mq_receive( )	从指定消息队列中接受优先级最高且时间最久的消息
mq_notify( )	记录当消息到达与指定消息队列描述符关联的空的消息队列时需要通报的调用进程
mq_getattr( )	获得消息队列的属性的状态信息以及与消息队列描述符关联的消息队列描述
mq_setattr( )	设置消息队列的状态信息和属性

2. 使用线程方法

在图 4-2 中的情形 1 中的分解中，将进程作为了分解单元。情形 2 将线程作为了分解单元。与情形 1 中应用程序产生 3 个进程不同，在情形 2 中，应用程序只有一个进程。然而，这个进程被分成 3 个可以并发执行的用户线程。在这个场景中，4 个并发任务的需求通过使用 3 个线程来实现。线程 A、B 和 C 均被指派任务，意味着 4 个任务必须在 3 个线程之间进行分布。值得注意的是，情形 2 只有两个内核线程。由于操作系统为处理器调度进程或调度内核线程，所以情形 2 中唯一的进程可以被调度到一个处理器，或者两个内核线程可以被调度到单独的处理器。这样，如果所有 4 个内核都是空闲的，则最多两个任务可以被同时执行。

在情形 2 中，线程 A、B、C 是用户线程。用户线程在某些情况下可以绑定到内核线程，或者与内核线程解除绑定(如本书稍后将会讨论的那样)。在情形 2 中，用户线程在真正被执行之前必须与内核线程关联。这样，在我们的线程方法中，可将软件任务映射到用户线程，然后用户线程被映射到内核线程或轻量级进程(lightweight processes，lwp)。

注意：
第 6 章将解释用户线程和内核线程之间的区别。

3. 如何实现从任务到线程的转换

pthread_create( )操作系统 API 用于将软件任务与线程进行关联。这个函数还被归入操作系统负责进行进程管理的部分。我们将在第 6 章详细介绍 POSIX 线程 API。除非用户是使用来自级别 3 或级别 4(见图 4-1)的工具和技术进行工作，否则应将软件任务和线程关联起来，并要求理解如何使用 POSIX 线程 API。注意在图 4-2 中，在使用线程方法进行分解时，任务的通信需求不要求特殊的操作系统干涉。这是因为线程共享相同的地址空间，因此可以共享进程的数据段中保存的数据结构。

4.1.5 应用程序员的接口

值得注意的是，在图 4-2 的情形 1 和情形 2 中，软件任务必须被映射到能够被操作系统管理和调度的实体。程序员不能够简单地将处理器指派给必须执行的每个任务。这只能够由操作系统来完成。程序员必须使用操作系统能够理解的执行单元来使得操作系统理解软件任务。操作系统是开发人员的软件与多个内核之间的软件层。操作系统提供一组接口(API)来允许应用程序开发人员使用硬件资源和操作系统服务。为了利用所有操作系统服务，开发人员必须使用 API。问题在于使用哪个操作系统 API？每个操作系统厂商都提供了自己特有的 API，尽管这些 API 的功能基本相同，但是它们不能够在不同平台之间移植。也就是说，使用 Mac OS X(Darwin)API 开发的软件不能够直接在 Solaris 上编译和执行，Solaris API 不能够直接在 Windows 环境中编译和执行，依此类推。因此，需要通过使用操作系统 API 来对多个内核进行充分利用的程序，如果使用的是特定系统的 API，那么就不

能够被移植。这意味着如果希望将应用程序在新的环境中使用，就必须重写该应用程序。在多数情况下，这是不能够被接受的。这就是本书中我们使用 POSIX API 的原因。

1. 什么是 POSIX 以及为何使用它

POSIX(Portable Operating System Interface，可移植操作系统接口)是定义了标准操作系统接口和环境的标准，包括命令解释器(shell)和常见的实用程序，用于在源代码级别支持应用程序的可移植性。应用程序开发人员和系统实现人员均可使用这个标准。为了使得本书尽可能适用于更多的系统开发人员和应用程序开发人员，我们选择使用 POSIX 标准来介绍操作系统 API。主流操作系统环境均声称对 POSIX 标准的基本支持，包括 ZOS、Solaris、AIX、Windows、Mac OS X、Linux、HP-UX 和 IRIX。尽管每种环境有着自己专有的 API，但均同时支持 POSIX 标准。既然我们讨论的概念、实例和程序都是基于 POSIX 标准的，您可以在任何环境中对它们进行验证。POSIX 标准扮演跨平台伪码的角色，使得我们可以用一种能够在所有主要环境中实现的语言介绍多核编程的主要概念。此外，POSIX 实现了一种“共同特性”操作系统接口。这意味着在多数情况下，将概念、原理和函数调用在必要时转换成专用的操作系统 API 是非常直观的。

由于 POSIX 标准的目的是在源码级别上提供可移植性，所以我们可以在 POSIX 组件之上构建类库、模板库和应用程序框架，然后将它们在所有主流操作系统环境中编译并使用。显然，对于特定平台的操作系统 API，是不能够做到这一点的。尤其是工作于图 4-1 中的级别 3 和级别 4 的开发人员可以从这类可移植性中受益。用于并行处理的应用程序框架(如 STAPL)以及模板库和类库(如 TBB)，可以通过对底层实现时的进程和线程使用 POSIX API 来得到可移植性。此外，如果使用 POSIX API，则在多个环境中混合和协调高级应用程序框架和构建块库就变得可行了。在级别 1 和级别 2 工作的开发人员使用 POSIX API 可以做到一次编写、到处编译。由于大规模计算机配置，如集群、企业级服务器(大型机)乃至超级计算机均有兼容 POSIX 的操作环境，当可扩展性是重要的问题时，开发人员有着规格齐全的硬件支持。尽管多核处理器刚刚在台式机、开发者工作站和小的服务器上出现，但它们已经广泛用于大规模计算机配置达十余年。因此，当您投入到 POSIX API 的学习中时，从小的商业应用服务器到最大的基于集群的配置，均能够应用它。

POSIX 标准使得我们可以以跨平台的方式来谈论多核编程与表 4-1 中列出的核心操作系统服务之间的交互。本书中的所有实例和程序均在兼容 POSIX 的环境中书写和编译，而且本书的两个附录包含了 POSIX 关于进程管理和线程管理的参考资料。

2. 进程管理

进程生命周期是本书不断提到的进程管理的一个重要方面。本书将进程生命周期概括为：

- 进程创建
- 进程调度/执行

PDG

- 进程终止

标准 C++库没有提供任何处理进程生命周期中主要活动的服务，因此您在需要对进程编程时，需要寄希望于操作系统 API。即使在 CMP 中，也可能没有足够的处理器来同时运行所有的进程。操作系统必须对进程进行多任务(multitasking)。多任务使得可以同时执行多个进程，而多线程允许一个进程同时执行多个任务。当操作系统使用调度策略来允许两个或多个进程并发共享 CPU 时，这被称为多任务。每个进程执行，直到运行了预设的时间长度或直到发生一些事件。给出的进程在内核上执行的时间间隔被称作时间片(quantum)。然后操作系统会切换到另一个进程。这个切换时间极短，产生进程同时执行的假象，而实际上在一个内核上，一次只有一个进程是活动的。这种进程间切换持续发生，直到所有进程都结束。调度策略决定了何时应当切换进程。调度策略还将控制当发生以下情形时该如何做：

- 进程或线程是一个运行线程，然后它成为了阻塞线程。
- 进程或线程是一个运行线程，然后它成为了被抢占线程。
- 进程或线程是一个阻塞线程，然后它成为了可运行线程。
- 一个运行线程调用了能够改变进程或线程的优先级或调度策略的函数。

在本书中，我们假定您的环境支持 4 种 POSIX 标准支持的基本调度策略：

```
SCHED_FIFO
SCHED_RR
SCHED_SPORADIC
SCHED_OTHER
```

表 4-3 包含了可以使用的每种基本调度策略的描述。

表 4-3

POSIX 调度策略	描　述
SCHED_FIFO	当时间片用完，线程被放置到其优先级队列的头部
SCHED_RR	当时间片用完，线程被放置到其优先级队列的尾部
SCHED_SPORADIC	使用服务器的调度策略
SCHED_OTHER	根据实现(implementation)定义，这是用于一般用途的最有效的调度策略

每个进程都被关联的调度策略和优先级所控制。同每种策略关联的是优先级范围。每种策略的定义指定了该策略的最小优先级范围。每种策略的优先级范围可能会与其他策略的优先级范围重叠。

操作系统也用来在进程间传输信号。当 Process A 必须向 Process B 发送一个终止信号时，操作系统将传输该信号。进程生命周期的每个主要步骤都受到操作系统的管理，而且您必须使用 POSIX API 来访问这些服务。要记住存储在磁盘的软件或程序不是进程或线程。进程是执行中的程序，拥有进程控制块(process control block)和进程表，而且被操作系统调度。线程是进程的一部分。在创建任何进程之前，必须由操作系统加载软件和程序。在创

建线程之前，必须已经创建了进程或轻量级进程。

3. 进程管理实例：游戏场景

为了说明上述内容，我们将要看一个经典的游戏。我想出一个 6 字符的编码，这个编码中包含的字符可以多次出现，但是只能包含数字 0～9 或字符 a～z。您的任务是猜出我所想的编码。在游戏中，时间限定为 5 分钟，如果能够在 5 分钟之内猜到我所想的内容，您就赢了。您拿出纸笔，进行简单的计算之后发现有 4 496 388 种可能。然后，在接下来的 2 分钟，您匆匆处理整个 SDLC 并提出如下的策略。首先，碰巧您有着一个文件包含 4 496 388 种可能编码，因此，您编写一个 C++程序来完成类似示例 4-1 中的工作。

示例 4-1

```
Example 4-1

//...

  bool Found = false;
  ifstream Fin(Possibilities)
  while(!Fin.eof() && !Fin.fail() && !Found)
  {
      getline(Fin,Guess);
      if(Guess == MagicCode){
          Found = true;
      }

   }

//...
```

问题是您不知道我所想的编码在包含 4 496388 种可能的文件的什么位置出现。根据我的编码在文件中的位置，您可能会花费 5 分钟以上的时间才能够找到。对文件进行排序并不能够提供帮助，因为您除了编码长度和能够包含的字符之外，对编码一无所知，这样就不能够使用类似于二叉搜索的便利的技术。

然而，假如您能够使用双核 CMP，那么在策略中就可以将包含超过四百万种可能的大的文件分成两个包含两百多万种可能的文件。这样，您开发了名为 find_code 的程序。这个程序接收一个包含编码的文件作为输入，并执行穷尽(穷举/顺序)搜索来查找编码。技巧在于您需要操作系统帮助您在搜索中使用两个内核，您将令两个内核同时各对两个文件之一进行搜索。您的想法是“三个臭皮匠，赛过诸葛亮”，如果将列表一分为二，那么找到编码所需的时间也会减半。这样，您希望操作系统同时运行 find_code 程序的两个版本，每个版本负责查找最初文件的一半。使用 posix_spawn( )调用来启动程序，如程序清单 4-1 所示。

程序清单 4-1

```
//Listing 4-1  Program (guess_it) used to launch find_code.

 1  using namespace std;
 2  #include <iostream>
 3  #include <string>
 4  #include <spawn.h>
 5  #include <sys/wait.h>
 6
 7  int main(int argc,char *argv[],char *envp[])
 8  {
 9
10     pid_t ChildProcess;
11     pid_t ChildProcess2;
12     int RetCode1;
13     int RetCode2;
14     int Value;
15     RetCode1 = posix_spawn(&ChildProcess,"find_code",NULL,
16                            NULL,argv,envp);
17     RetCode2 = posix_spawn(&ChildProcess2,"find_code",NULL,
18                            NULL,argv,envp);
19     wait(&Value);
20     wait(&Value);
21     return(0);
22  }
```

第 15 行和第 17 行的 posix_spawn( )调用启动了名为 find_code 的程序。程序 find_code 必须是操作系统可以在计算机上找到的二进制可执行程序。在本例中，find_code 是实现示例 4-1 的基本思想的独立程序。当执行程序清单 4-1 中的程序时，操作系统生成 3 个进程。回想一下，是操作系统而不是程序将进程指派到内核和 CPU 上来执行。进程可以同时执行。两个进程与 find_code 程序关联，第三个进程是名为 posix_spawn( )的进程。作为调用 posix_spawn 的结果而产生的进程被称作子进程。

注意：

我们将在第 5 章中详细介绍 posix_spawn( )。

程序概要 4-1

程序名：

guess_it.cc(程序清单 4-1)

描述：

posix_spawn( )启动名为 find_code 的程序。find_code 是一个独立程序，实现示例 4-1 的基本思想。程序使得操作系统生成 3 个同时执行的进程。两个进程执行 find_code 程序，

第三个进程执行 posix_spawn( )。

必需的库：

无

必需的用户定义头文件：

无

编译和链接指令：

```
c++ -o guess_it guess_it.cc
```

测试环境：

Linux Kernel 2.6
Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

值得注意的是，由 posix_spawn 所产生的子进程总是存在于调用程序之外的二进制可执行程序。不像 pthread_create( )调用程序内的例程那样，posix_spawn( )使用存在于调用程序之外的代码。

每个操作系统环境各自有着独特的方法来产生子进程。posix_spawn( )方法能够在有着适当的 POSIX 兼容的操作环境中使用。因此，您可以构建用于进程创建的跨平台组件。

调用 posix_spawn( )的进程被称作父进程。这样，操作系统创建了两个子进程和一个父进程。如果两个内核都空闲，那么操作系统可以指派 3 个进程中的两个来同时执行。现在您意识到，尽管已经将列表分成两半，而且有两个搜索同时进行，但是仍不确定能够及时在两百万种可能中找到编码，因此您需要再一次地将列表进行分割。这样就得到了 4 个列表，每个列表包含一百万种编码，并发搜索所需要的时间就会更短一些。当然，您能够在 5 分钟之内在包含一百万种可能编码的列表中找到目标编码。为了生成 4 个搜索，可以将 find_code 程序分为两个执行线程。这样，主程序为程序清单 4-1 中的 guess_it，它产生两个 child_processes 来执行 find_code 程序。find_code 程序创建两个名为 task 1 和 task 2 的线程。程序清单 4-2 是新的多线程版本的 find_code。

程序清单 4-2

```
//Listing 4-2  A multithreaded version of the find_code program.

 1  #include <pthread.h>
 2  using namespace std;
 3  #include <iostream>
 4  #include <fstream>
 5  #include "posix_queue.h"
 6  string MagicCode("yyzzz");
 7  ofstream Fout1;
 8  ofstream Fout2;
 9  bool Found = false;
10  bool magicCode(string X)
11  {
12     //...
13
14    return(X == MagicCode);
15  }
16
17
18
19  void *task1(void *X)
20  {
21    posix_queue PosixQueue;
22    string FileName;
23    string Value;
24    if(PosixQueue.open()){
25       PosixQueue.receive(FileName);
26       ifstream Fin(FileName.c_str());
27       string FileOut(FileName);
28       FileOut.append(".out");
29       Fout1.open(FileOut.c_str());
30       while(!Fin.eof() && !Fin.fail() && !Found)
31       {
32          getline(Fin,Value);
33          if(!Fin.eof() && !Fin.fail() && !Found){
34             if(magicCode(Value)){
35                Found = true;
36             }
37          }
38       }
39       Fin.close();
40       Fout1.close();
41    }
42    return(NULL);
43  }
44
45
46
```

```
47  void *task2(void *X)
48  {
49
50     posix_queue PosixQueue;
51     string FileName;
52     string Value;
53     if(PosixQueue.open()){
54        PosixQueue.receive(FileName);
55        ifstream Fin(FileName.c_str());
56        string FileOut(FileName);
57        FileOut.append(".out");
58        Fout2.open(FileOut.c_str());
59        while(!Fin.eof() && !Fin.fail()  && !Found)
60        {
61           getline(Fin,Value);
62           if(!Fin.eof() && !Fin.fail()  && !Found){
63              if(magicCode(Value)){
64                 Found = true;
65              }
66           }
67        }
68        Fin.close();
69        Fout2.close();
70     }
71     return(NULL);
72  }
73
74
75
76
77
78  int main(int argc, char *argv[])
79  {
80
81     pthread_t ThreadA, ThreadB;
82     pthread_create(&ThreadA,NULL,task1,NULL);
83     pthread_create(&ThreadB,NULL,task2,NULL);
84     pthread_join(ThreadA,NULL);
85     pthread_join(ThreadB,NULL);
86     return(0);
87
88  }
```

第 82 行和第 83 行的 pthread_create( )函数用于为 task1 和 task2 创建线程(我们将在第 6 章中详细介绍 POSIX pthread 功能)。程序清单 4-2 中的程序仅用于说明的目的，其中并不包含任何同步、异常处理、信号处理等。我们将它包含在这里，目的是让您对示例 4-1 中介绍的 guess_it 程序的结构有一个清晰的认识。注意第 19 行中的 task1 和第 47 行的 task2 是正常的 C++函数。它们被用作 ThreadA 和 ThreadB 的主例程。还要注意第 24 行和第 53

行中这两个线程访问 PosixQueue。PosixQueue 是一个用户定义对象，它含有每个线程将要进行查找的不同文件名。

这样，程序清单 4-1 中的程序产生两个子进程。每个进程执行 find_code，find_code 又依次创建两个线程。这样一共产生了 4 个线程，每个线程从 PosixQueue 对象中读取出一个文件名。因此，不再是有一个包含 4 496 388 种可能的大的文件，而是有 4 个包含稍大于一百万种可能的较小的文件。您将让每个线程使用简单的穷尽搜索，其中之一将会找到我所想的 MagicCode。由于在程序清单 4-2 的第 9 行声明了 Found 变量，这里属于 ThreadA 和 ThreadB 的文件作用域或全局作用域，因此可用作控制变量来令两个线程停止。但是第二个进程中的其他两个线程怎么办？程序中使用 PosixQueue 来将文件名传递给两个进程和所有 4 个线程。一旦一个线程找到了 MagicCode，是否能够使用队列来另外 3 个进程和 4 个线程及时知道应当停止呢？

程序概要 4-2

程序名：

`find_code.cc`（程序清单 4-2）

描述：

程序 find_code 创建名为 task1 和 task2 的两个线程。每个线程访问 PosixQueue。这是一个用户定义对象，它包含每个线程将要进行编码搜索的不同文件名。

必需的库：

`pthread`

必需的用户定义头文件：

`posix_queue.h`

编译和链接指令：

```
c++ -o find_code find_code.cc posix_queue.cc -lpthread -lrt
```

测试环境：

Linux Kernel 2.6
Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

4.2 分解以及操作系统的任务

由于以下两个原因，本书中将多次回顾分解这个主题：

- 软件设计的基本活动是将问题和解决方案按某种方式分解，使得解决方案(有时候是问题本身)能够在软件中加以实现。
- 并行编程、多线程和多处理都要求将软件分解为可以由操作系统调度，从而可被并发处理的执行单元。

这使得分解对于多核编程非常重要。注意在前面章节中所介绍的经典游戏示例中，并没有提到并行编程、多线程、操作系统等。有的只是对问题的简单陈述，即在 5 分钟或更短时间之内猜到我所想的 6 字符编码。我们是从对问题的简单平实的描述开始，最终以一个要求进程间通信和操作系统干预和协助的多处理、多线程程序结束。简单的描述与同时执行的操作系统线程之间的主要连接是分解的过程。应从逻辑细分和物理细分的角度来考虑示例。图 4-3 包含了对来自程序清单 4-1 和程序清单 4-2 的程序 guess_it 的操作系统组件的功能(逻辑)细分。

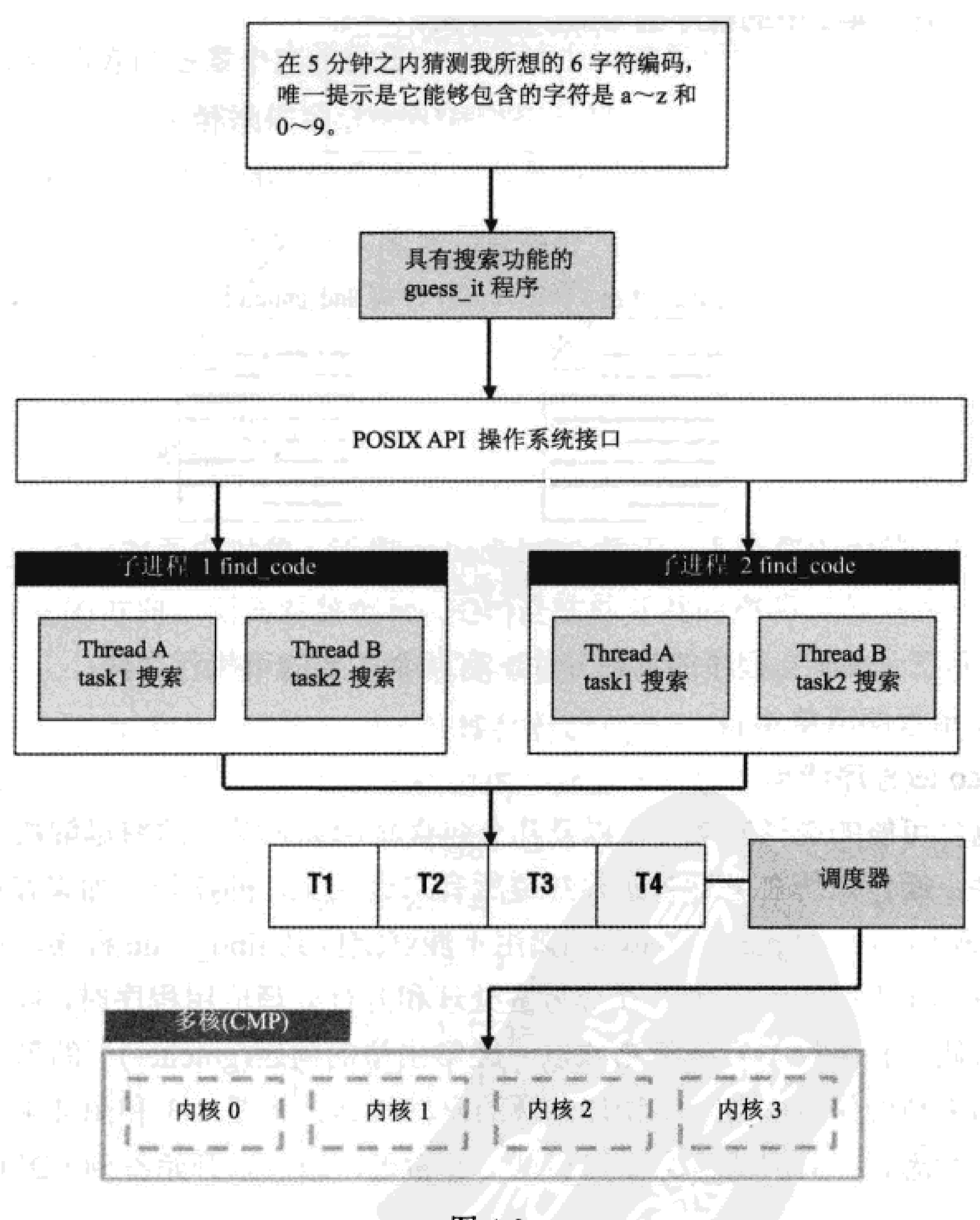


图 4-3

如图 4-3 所示，操作系统一共要负责 6 个执行单元，包括两个进程和 4 个线程。回想一下在示例 4-1 中，我们最初是用一个程序在一个文件中进行查找。我们使用操作系统 API 来产生程序的两个示例，这样就可以同时查找两个文件了。注意在图 4-3 中，find_code 又被分成两个线程。因此，实际工作组件可以很容易地在图 4-3 中看到。图 4-3 中所没有显示出来的是将一个包含超过四百万种可能的数据文件分解为 4 个较小的、包含超过一百万种可能的文件。图 4-3 中的每个线程在自己的文件上进行查找，这样，除了对工作进行分解之外，我们还做了数据分解。程序 guess_it 的分解是单指令多数据(SIMD)并发模型的一个示例。在 SIMD 并发模型中，多个任务在不同数据集上执行相同的指令序列。我们有 4 个线程，每个线程在 4 组不同的数据上执行相同的代码(单指令)。即使我们有着工具上的优势，例如 STAPL 或 TBB，但我们最终仍然需要同操作系统进行交互以实际实现这个 SIMD 模型。这种类型的分解以及操作系统接口是图 4-1 中所示的级别 1 和级别 2 上的编程。尽管在级别 3 和级别 4(见图 4-1)上工作的开发人员通常不受这种级别的交互的影响，但是应当清楚操作系统的任务。

除了涉及操作系统的逻辑单元的细分之外，还有物理单元的细分。图 4-4 显示了程序清单 4-1 和程序清单 4-2 中的程序的 UML 部署图。

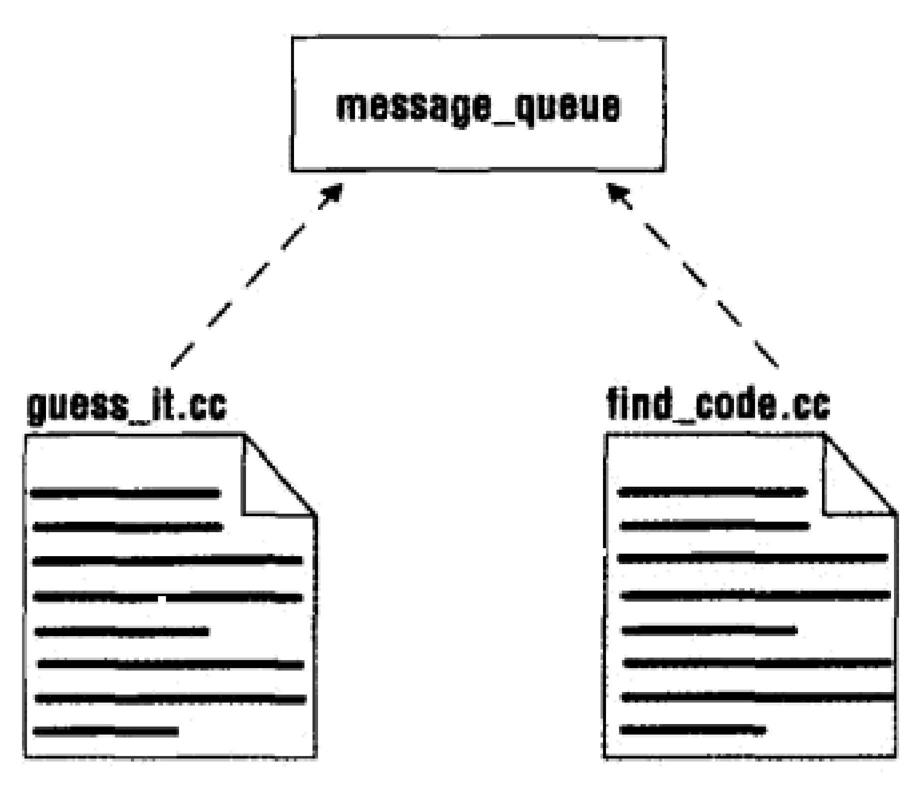


图 4-4

图 4-4 显示了 guess_it 程序的物理分片。有两个主要的可执行文件：

- guess_it(程序清单 4-1)
- find_code(程序清单 4-2)

有 4 个包含可能的选择的文件，以及几个包含猜测编码游戏的穷尽解决方案以及消息队列的源文件。操作系统在运行时必须知道所有二进制文件的路径。如果在程序清单 4-1 的第 15 行和第 16 行中的 posix_spawn( )调用不能够定位到 find_code 程序，那么程序清单 4-2 中的 guess_it 程序将无法工作。在部署多处理和并行处理应用程序时，操作系统找到并加载待执行代码的任务经常会被忽略或以“经常出错的问题(gotchas)”的形式出现，您可以使用部署图来帮助保留应用程序物理分解的审核线索。将图 4-3 和图 4-4 中的分解、问题的最初陈述考虑、在其解决方案上的初试一并考虑，您可以开始看到 SDLC 如何在多核应用的设计与实现中发挥主要作用。从最初的问题陈述：“在 5 分钟内猜出我所想的 6 字符

编码”，我们设计了一种查找可能编码列表的解决方案。但是由于列表过大而且我们有时间约束，因此我们又提出了需要将可能编码列表分成较小的列表，这样可以同时在这些较小的列表中查找 MagicCode。

这个策略是 SDLC 中的设计活动的示例。最初的问题陈述是 SDLC 中的需求定义的实例。使用 posix_spawn( )、pthread_create( )和 PosixQueue 来对策略进行实现是 SDLC 中编码活动的一部分。尽管在 SDLC 的需求定义或设计活动中，未必需要考虑操作系统，但是在编码、部署和维护活动中，就必须考虑操作系统了。这里，我们的目的是想弄清楚它的功能在哪些方面适合需要利用多核 CMP 的应用程序。在第 5 章和第 6 章中，我们将详细介绍进程和线程，它们是可以被操作系统调度为同时执行的执行单元。

4.3　隐藏操作系统的任务

真正的目标是在软件设计不需要陷入到进程和线程实现细节的前提下，理解在执行多线程和多处理程序中操作系统的作用。您最终希望摆脱线程和进程实现细节的主要原因是因为 CMP 发展的方向是多个内核位于一个芯片上，终极目标是一个芯片上具有成百上千个内核的大规模并行。了解操作系统的任务将很重要，但是您不会希望在设计中接触到它。在程序清单 4-1 和程序清单 4-2 中，涉及了操作系统。随着芯片数目的增加和并行程度的增强，您需要在软件开发中实现两个重要的目标：

(1) 利用操作系统的同时令它对软件设计透明。

(2) 从并行编程的过程式范型转到声明式范型。

4.3.1　利用 C++抽象和封装的能力

幸运的是，C++对面向对象、泛型(genericity)、谓词、多范型编程的支持，给出了未来软件设计和开发的方向。面向对象编程(OOP)是软件开发的声明式范型的一部分[Meyer，1988][Stroustrup，1997]。在第 4.3.2 节中您将看到，C++所支持的封装的概念有助于实现软件设计中操作系统级透明。模板可用于实现并行编程高阶(higher-order)声明式方法中的泛型技术，C++中的类、谓词和断言的思想可用于转向支持大规模复杂并行编程技术的声明式编程技术。类和模板库，如 STAPL 和 TBB，是支持转向 CMP 大规模并行的初步的组件。其思想是构建封装底层过程化驱动的操作系统 API 的功能的类，并提供更高级别的声明式接口。C++接口类非常适用于为底层操作系统 API、同步机制和通信组件提供包装[Stroustrup，1997]。同时，您可以使用 C++模板来捕获并行模式，在用户访问更高层、更具功能性接口的同时实现细节。还可以从捕获支持并行化架构的 C++组件应用框架开始进行构建。使用高级组件、框架和架构，可以直接实现在 SDLC 的设计和规格说明等活动中产生的模型。

4.3.2 POSIX API 的接口类

令 POSIX API 透明的最容易的方法是提供 C++接口类。接口类是提供函数、数据或其他类的包装的类。接口类就像某种装束一样，使得某个物体看上去跟平常时候不一样。接口类改变函数、数据、另一个类的外观。接口类也被称作适配类。接口类提供的新的接口被设计为令类更易于使用、更具功能性、更安全或语义正确。以程序清单 4-2 中第 81 行～第 85 行所显示的 POSIX 线程函数为例，我们希望这个程序的主线不要暴露操作系统调用，而且希望对 guess_it 程序加入更多的 C++面向对象风格。程序清单 4-3 包含了程序清单 4-2 中的 find_code 程序的新的实现。

程序清单 4-3

```
//Listing 4-3  A more object-oriented find_code:  ofind_code.

 1  #include "thread_object.h"
 2
 3
 4  int main(int argc, char *argv[])
 5  {
 6
 7
 8
 9     user_thread Thread[2];
10     Thread[0].name("ThreadA");
11     Thread[1].name("ThreadB");
12     for(int N = 0; N < 2;N++)
13     {
14        Thread[N].run();
15        Thread[N].join();
16     }
17     return(0);
18
19  }
```

程序清单 4-3 中的代码替换了程序清单 4-2 代码中的第 78 行～第 88 行。尽管没有减少代码行数，但是我们改变了线程创建和执行过程的接口。现在使用一个 user_thread 类来封装 pthread_t 线程的 id 以及其他一些 pthread 函数。现在我们是声明对象并调用方法，而不是调用 POSIX API 函数。程序清单 4-3 中的程序用于创建并执行两个线程，然后在退出之前与线程结合。在程序清单 4-2 中可以较容易地看出线程将要做些什么工作，但是在程序清单 4-3 中，线程在做些什么不是很明显。在程序清单 4-2 的第 82 行和第 83 行，调用了 pthread_create，并将函数 task1 和 task2 的名称传递给它，它们将会被 ThreadA 和 ThreadB 执行。在程序清单 4-3 中，由于封装的缘故，ThreadA 和 ThreadB 将执行什么就不是很明显。我们只能够看到调用了 run( )方法。为了更好地理解程序清单 4-3 如何替代程序清单 4-2，需要看一下程序清单 4-3 的第一行中包含的 thread_object.h 中的声明。在头文件

thread_object.h 中包含名为 thread_object 的抽象类。我们知道它是一个抽象类，是因为在程序清单 4-4 的第 14 行中声明的抽象虚方法。

程序概要 4-3

程序名：

`ofind_code.cc`(程序清单 4-3)

描述：

程序清单 4-3 中的程序创建并执行两个线程。然后它在退出之前与这些线程结合。run( ) 方法调用要执行的任务。用程序清单 4-3 取代程序清单 4-2，请在 thread_object.h 中查看声明。

必需的库：

`rt,pthread`

需要的其他源文件：

`thread_object2.cc`(程序清单 4-5)，`user_thread.cc`(程序清单 4-6)

必需的用户定义头文件：

`thread_object.h`(程序清单 4-4)，`posix_queue.h`

编译和链接指令：

```
c++ -o ofind_code ofind_code.cc user_thread.cc thread_object.cc
posix_queue.cc -lrt -lpthread
```

测试环境：

Linux Kernel 2.6
Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

程序清单 4-4

```
//Listing 4-4  A declaration of a simple thread_object.


 1  #ifndef  __THREAD_OBJECT_H
 2  #define  __THREAD_OBJECT_H
 3
 4  using namespace std;
 5  #include <iostream>
 6  #include <pthread.h>
 7  #include <string>
 8  #include "posix_queue.h"
 9
10  class thread_object{
11    pthread_t Tid;
12    string Name;
13  protected:
14    virtual void do_something(void) = 0;
15  public:
16    thread_object(void);
17    ~thread_object(void);
18    void name(string X);
19    string name(void);
20    void run(void);
21    void join(void);
22    friend void *thread(void *X);
23  };
24
25
26
27  class user_thread : public thread_object{
28    private:
29    posix_queue *PosixQueue;
30  protected:
31    virtual void do_something(void);
32  public:
33    user_thread(void);
34    ~user_thread(void);
35  };
36
37
38  #endif
```

其中 do_something( ) = 0 方法防止用户简单地声明一个 thread_object 对象。为了使用 thread_object 类，用户必须为 do_something( )提供功能性，方法是继承 thread_object 并为 do_something( )提供一个实现。在程序清单 4-2 的程序上下文中，do_something 方法与程序清单 4-2 的第 19 行和第 47 行的 task1 及 task2 有相同的功能。方法 do_something 搜索文件，寻找 MagicCode。还要注意程序清单 4-4 的第 22 行。该 friend 函数与 do_something( )方法

一起使用来为 pthread_create 功能提供保证。类 user_thread 继承 thread_object 类，并为 do_something( )方法提供定义。注意在程序清单 4-4 中，user_thread 类有 posix_queue 数据成员。这是程序清单 4-2 的第 25 行和第 53 行所使用的 PosixQueue。这个 thread_object 简单示例示范了与程序清单 4-2 中纯过程式方法的不同之处，虽然差别不大，但的确是向着面向对象方法的改变。

图 4-5 显示了 user_thread 类的 UML 类关系图。

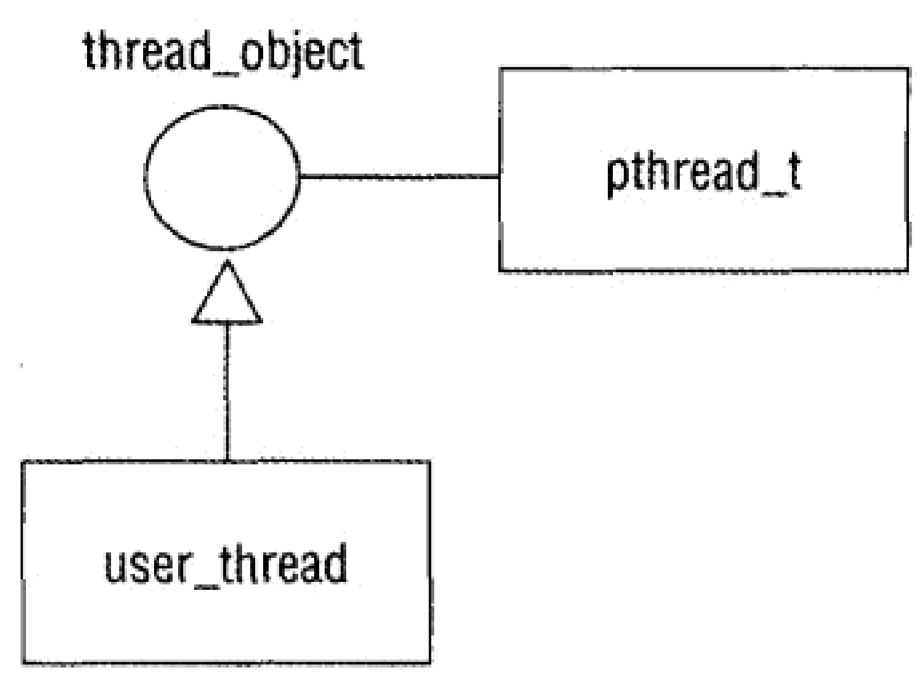


图 4-5

迄今为止，thread_object 类只是一个简单的框架类(skeleton class)，我们将逐步对这个类进行定义。thread_object 类还是一个接口类，它的目的是封装 POSIX 线程接口并提供面向对象语义和组件，从而使得我们可以更加容易地实现 SDLC 中产生的模型。比较图 4-3 和图 4-5 中组件的逻辑细分，显然图 4-5 中的焦点明显不同，因为我们是对 find_code 程序进行面向对象的分解。user_object 定义了继承自 thread_object 的 find_code 函数。面向对象方法隐藏了图 4-3 中显示的实现细节。程序清单 4-5 包含简单了 thread_object 类的一些实现。

程序清单 4-5

```
// Listing 4-5 A definition of a simple thread_object.

 1  #include "thread_object.h"
 2
 3
 4  thread_object::thread_object(void)
 5  {
 6
 7
 8
 9  }
10  thread_object::~thread_object(void)
11  {
12     pthread_join(Tid,NULL);
13  }
14
```

```
15
16  void thread_object::run(void)
17  {
18     pthread_create(&Tid,NULL,thread,this);
19  }
20
21  void thread_object::join(void)
22  {
23     pthread_join(Tid,NULL);
24  }
25
26
27  void thread_object::name(string X)
28  {
29     Name = X;
30  }
31
32  string thread_object::name(void)
33  {
34     return(Name);
35  }
36
37
38  void *  thread (void * X)
39  {
40
41     thread_object *Thread;
42     Thread = static_cast<thread_object *>(X);
43     Thread->do_something();
44     return(NULL);
45
46
47  }
```

现在您可以看到 run( )和 thread( )方法如何用来共同提供 pthread_create( )调用的功能。这只是开始，我们可以做得更好。注意这里没有 thread_object 类中声明的 do_something( )方法的实现，这个方法将会在 thread_object 类派生子类时由用户提供。程序清单 4-5 中第 43 行的 Thread->do_something( )调用将会由派生类提供的方法。在我们的示例中，这是由程序清单 4-6 中的定义来定义的。

程序清单 4-6

```
//Listing 4-6  The definition for the user_thread class.

 1  #include "thread_object.h"
 2  #include <iostream>
 3  #include <fstream>
 4
 5  bool Found = false;
```

```
 6
 7
 8  user_thread::user_thread(void)
 9  {
10
11     PosixQueue = new posix_queue("queue_name");
12     PosixQueue->queueFlags(O_RDONLY);
13     PosixQueue->messageSize(14);
14     PosixQueue->maxMessages(4);
15
16  }
17
18
19  user_thread::~user_thread(void)
20  {
21
22     delete PosixQueue;
23
24  }
25
26
27  void user_thread::do_something(void)
28  {
29     ofstream Fout;
30     string FileName;
31     string Value;
32
33     if(PosixQueue->open()){
34        PosixQueue->receive(FileName);
35        ifstream Fin(FileName.c_str());
36        string FileOut(FileName);
37        FileOut.append(".out");
38        Fout.open(FileOut.c_str());
39
40        while(!Fin.eof() && !Fin.fail() && !Found)
41        {
42           getline(Fin,Value);
43           if(!Fin.eof() && !Fin.fail() && !Found){
44              if(Value == MagicCode){
45
46                 Found = true;
47
48              }
49
50           }
51        }
52        Fin.close();
53        Fout.close();
54     }
55
```

```
56  }
```

user_thread 类中的主要工作是由 do_something( )方法来执行的。通过覆盖 do_something( )方法，我们可以使用这个 user_thread 类来完成通过 pthread_create 功能能够做的任何类型的工作。在本案例中，do_something( )方法执行文件搜索。在程序清单 4-3 中，user_thread 对象调用的线程的 run( )方法最终执行 do_something( )方法。由于在第 5 行中定义的 Found 变量是一个全局变量，而且拥有文件作用域，因此一旦定位到了待搜索的值，我们就可以使用它来终止线程。

程序概要 4-4

程序名：

`user_thread.cc`(程序清单 4-6)

描述：

user_thread 类通过 do_something( )方法来完成，该方法能够做 pthread_create 功能所能够做的任何类型的工作。do_something( )方法用于完成文件搜索。被 user_thread 对象调用的线程中的 run( )方法执行 do_something( )方法。由于 Found 变量是全局的，而且有着文件作用域，它可以在定位到待搜索的值之后停止线程。

必需的库：

`pthread`

需要的其他源文件：

`thread_object2.cc`(程序清单 4-6)

必需的用户定义头文件：

`thread_object.h`(程序清单 4-4)

编译指令：

```
cc++ -c user_thread.cc
```

测试环境：

Linux Kernel 2.6
Solaris 10、gcc 3.4.3 和 gcc 3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

联合使用接口类及 POSIX，使得您可以创建跨平台组件，它们有助于跨平台多线程或多处理应用程序的实现。当然，程序清单 4-4 中声明的 thread_object 接口类在能够用于生产环境之前，必须被大幅度充实。C++接口类广泛用于高级组件库和应用框架中，如 STAPL 和 TBB。如果您理解了接口类如何与操作系统 API 结合使用，那么 TBB、STAPL 和操作系统之间的关系将会更明显。接口类可用来将您自己的构建块加入到 TBB、STAPL 以及并行处理和多线程所使用的其他高级库中。

4.4 小结

应用程序开发人员和系统开发人员都需要对操作系统在多处理器系统中的作用有清楚的理解。理想情况下，应用程序员不需要直接与操作系统本身打交道。但是他们仍应了解一些基础知识，因为会在测试、调试、软件部署中遇到一些挑战。本章讨论了操作系统在多核编程中的作用。包含的重点有：

- 操作系统是 CMP 的看门人。任何希望使用多处理器的软件必须与操作系统打交道。由于 C++标准中不包括对进程管理或线程管理的直接支持，您可以使用 POSIX API 来访问与进程管理和线程管理相关的操作系统服务。
- 操作系统的作用可以分成以下两个主要方面。
  - 软件接口：为计算机的硬件资源提供一致而且定义良好的接口。
  - 资源管理：管理硬件资源和其他正在执行的应用软件、作业和程序。
- 操作系统为程序员在开发人员的程序和连接到计算机的硬件之间提供了两个软件层(API 和 SPI)。
- 操作系统的核心服务可分为：
  - 进程管理
  - 内存管理
  - 文件系统管理
  - I/O 管理
  - 进程间通信管理器
- 类似 STAPL 的框架的目标是允许开发人员在不需要担心并行编程涉及的所有与特定实现相关的问题的前提下，在软件应用程序中提供并行性。类似 TBB 的库是一组高级通用组件，同时封装了多处理和多线程的很多细节。
- 一旦软件设计过程确定最好将应用程序分成两个或多个并发执行的任务，操作系统的透明性就成了一个问题。思路是构建封装底层操作系统 API 的过程驱动功能的类，同时为应用程序开发人员提供高级声明式接口。

- 当您转向更多内核和更好并行性时，您需要进行两个重要的操作，从而使得大规模并行 CMP 上的软件开发变得切实可行：
  - 利用操作系统的同时，让它对软件设计保持透明
  - 从并行编程的过程式范型转到声明式范型
- 在 C++组件中封装操作系统进程和线程的管理服务。这样，从 C++组件出发，构建捕获支持并行性的架构的应用框架。

总之，类库、高级函数库或应用程序框架中所包含的多线程或多处理功能仍需要通过操作系统 API 或 SPI。在接下来的两章里，我们将更详细地了解进程和线程在多核编程中的使用方法。

进程、C++接口类和谓词

在第 4 章中，我们查看了操作系统在要求并行编程的应用程序中作为开发工具的作用，提供了操作系统在进程管理和线程管理中的作用的简要概述。我们为读者介绍了操作系统应用程序接口(API)和系统程序接口(SPI)的概念，并特别介绍了 POSIX API。在本章中，我们将进一步了解：

- 进程在哪里和 C++编程以及多核计算机相契合
- 用于进程管理的 POSIX API
- 进程调度和优先级
- 构建可用于简化进程管理中 POSIX API 的 C++接口组件

基本上，为了达到并发并利用多核处理器，程序可以被分成多个进程和/或线程。在本章中，我们将介绍操作系统如何识别进程以及应用程序如何利用多个进程。

5.1 多核是指多处理器

多核是单芯片多处理器或 CMP 的流行的简称。多处理器是指有着两个或多个 CPU 或处理器的计算机。尽管多处理器计算机已经面世一段时间了，但是 CMP 的广泛可用以及低成本才会令所有软件开发人员具备多处理器开发的能力。这引发了一系列的问题：单个应用程序如何利用 CMP？单用户应用程序与多用户应用程序如何利用 CMP？使用 C++如何利用操作系统的多处理和多道程序设计的能力？一旦软件设计中包含了一些任务需要并发执行的需求，如何将这些任务映射到多核计算机的多个可用处理器上？

回顾第 4 章，操作系统调度它能够理解的执行单元。如果您的软件设计中包含一些能够并行执行的任务，则必须找出一种方法来将这些任务关联到操作系统能够理解的执行单元。将任务和操作系统执行单元联系起来，是包含 3 次转换的四阶段过程的一部分。

图 5-1 中的每次转换都改变了模型的视图，但是模型的含义仍应当保持不变。也就是说，应用程序框架、类库、进程与线程模板的实现不应当改变这些组件的含义或语义。第四阶段的执行单元是操作系统可以直接进行处理的。图 5-1 中第四阶段内显示的执行单元

是唯一可以直接指派给内核的事物。从操作系统的视角，应用程序是一个或者多个进程的集合。在一个 C++应用程序中，并发最终是通过将程序分解为多个进程或多个线程来实现的。尽管 C++程序的逻辑组织可能多种多样(例如，在对象、谓词、函数或通用模板内)，但是并行化(除了指令级并行)都是通过多个进程或线程的使用来产生的。

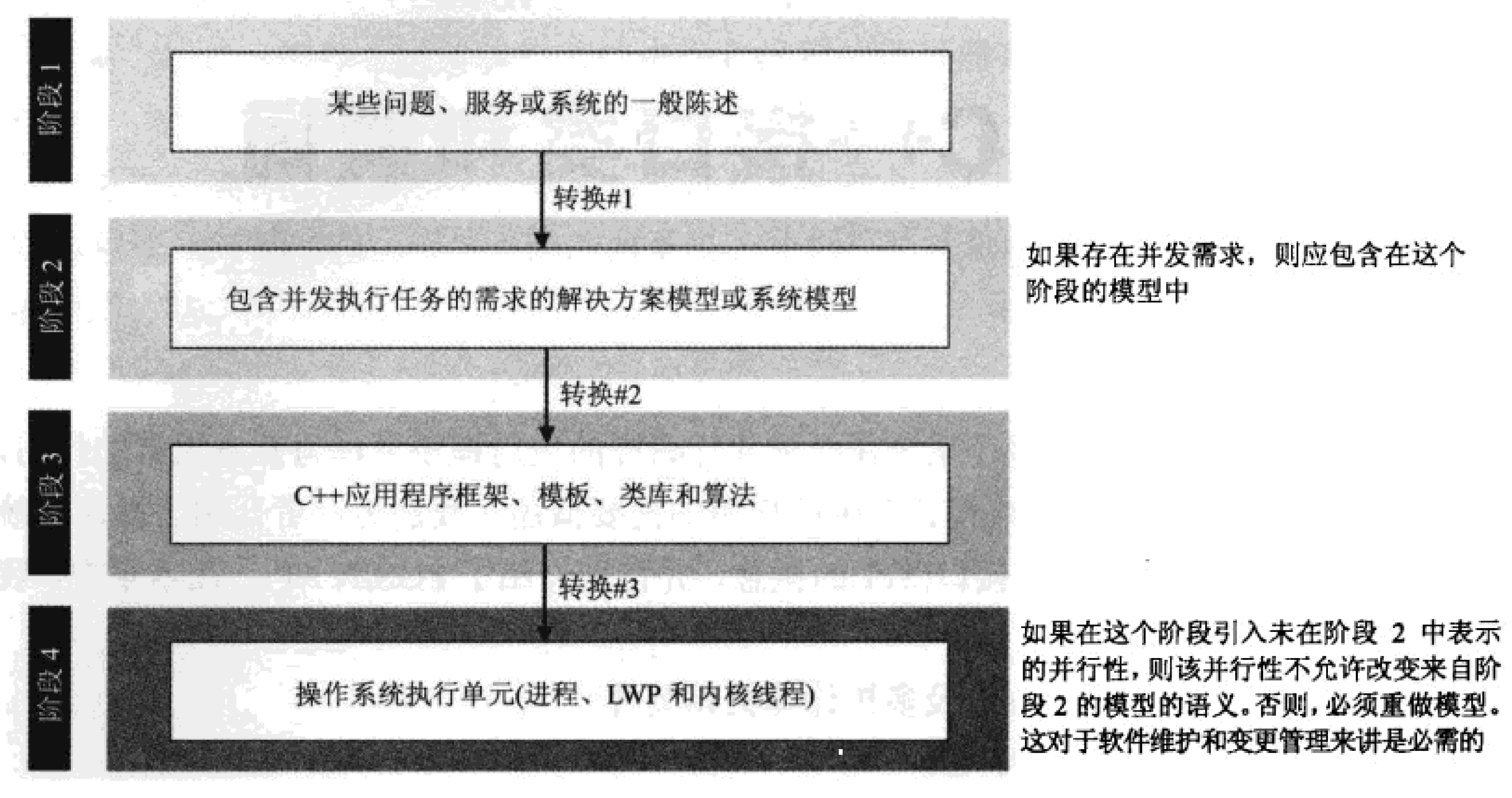


图 5-1

本章集中介绍进程的概念以及如何使用 POSIX API 进程管理服务将 C++应用程序和程序分成多个进程。

5.2 什么是进程

进程是由操作系统创建的工作单元。值得注意的是，进程和程序未必是等同的。一个程序可能由多个任务组成，而每个任务可以和一个或多个进程关联。进程是由操作系统创造的，而程序是由开发人员创造的。当今的操作系统能够管理成百乃至上千的并发加载进程。一个工作单元要想被称为进程，它必须拥有由操作系统指派给它的地址空间，必须拥有进程 id，必须拥有状态和进程表中的表项。根据 POSIX 标准，它必须有一个或多个控制流在这个地址空间中执行，而且拥有这些控制流所要求的系统资源。进程在其地址空间中有一组执行指令，要为这些指令、进程所拥有的所有数据、函数调用栈和局部变量分配空间。进程和线程之间的重要区别就在于每个进程有着自己的地址空间，而线程共享创建它们的进程的地址空间。程序可以被分解为一个或多个进程。

5.3 为什么是进程而不是线程

当您将 C++任务映射为操作系统能够理解的执行单元时，结果证明线程更易于编程。

这是因为线程共享相同的地址空间，使得线程间的通信和同步都要容易很多。操作系统创建或终止一个线程所需要进行的工作少于创建或终止一个进程所需要的。一般而言，在只有一台计算机的环境下，您可以创建的线程数目要多于进程数目，线程的启动和停止速度也要快于进程。

那么为什么还要使用进程呢？首先，进程有它们自己的地址空间。这是很重要的，因为隔离的地址空间能够在与流氓进程或设计较差的进程之间提供一定数量的安全性和隔离性。其次，线程能够使用的打开的文件数目受限于一个进程能够拥有的打开的文件数目。将 C++应用程序分成多个进程，而不是分成多个线程，或者将进程和线程联合使用，能够提供对更多打开的文件资源的访问。对于多用户应用程序，您会希望每个用户的进程都是孤立的。如果一个用户的进程失败，那么其他用户可以继续进行工作。如果为多用户应用程序使用线程方法，一个出错的线程就会中断所有的用户。操作系统的资源主要是分配给进程，然后再被线程共享，一般来说，线程受限于进程能够拥有的资源数量。这样，当隔离安全性、地址空间隔离性、并发执行的任务能够拥有的最大资源数目是最主要的考虑时，使用进程要比使用线程好些。进程间通信以及启动时间是主要的代价。

表 5-1 中列出的函数将在 spawn.h 中声明。这个头文件包含了用于产生和管理进程的 POSIX 函数。

表 5-1

POSIX 函数类型	POSIX 函数
创建进程	posix_spawn( ) posix_spawnp( )
初始化属性	posix_spawnattr_init( )
销毁属性	posix_spawnattr_destroy( )
设置和获取属性值	posix_spawnattr_setsigdefault( ) posix_spawnattr_getsigdefault( ) posix_spawnattr_setsigmask( ) posix_spawnattr_getsigmask( ) posix_spawnattr_setflags( ) posix_spawnattr_getflags( ) posix_spawnattr_setpgroup( ) posix_spawnattr_getpgroup( )
进程调度	posix_spawnattr_setschedparam( ) posix_spawnattr_setschedpolicy( ) posix_spawnattr_getschedparam( ) posix_spawnattr_getschedpolicy( ) sched_setcheduler( ) sched_setparm( )

(续表)

POSIX 函数类型	POSIX 函数
增加文件动作	posix_spawn_file_actions_addclose( ) posix_spawn_file_actions_addup2( ) posix_spawn_file_actions_addopen( ) posix_spawn_file_actions_destroy( ) posix_spawn_file_actions_init( )

5.4 使用 posix_spawn( )

类似于创建进程的 fork-exec( )和 system( )方法，posix_spawn( )函数从指定的进程映像创建新的子进程，但是 posix_spawn( )函数在创建时有着更细粒度的控制。尽管 POSIX API 也支持 fork-exec( )类别的函数，但我们只聚焦于 posix_spawn 进程创建函数，以得到更好的跨平台可兼容性。有些平台在实现 fork( )时可能会有问题，因此 posix_spawn( )函数可用作替代品。这些函数控制着子进程从父进程继承到的属性，包括：

- 文件描述符
- 调度策略
- 进程组 id
- 用户 id 和组 id
- 信号掩码(signal mask)

这些函数还控制着被父进程忽略的信号是否被子进程忽略或重设为一些默认动作。对文件描述符的控制允许子进程独立访问父进程打开的数据流。能够设置子进程的组 id，可以影响子进程的作业控制如何与父进程的作业控制发生联系。子进程的调度策略可以被设置成不同于父进程的调度策略。

调用形式

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
            const posix_spawn_file_actions_t *file_actions,
            const posix_spawnattr_t *restrict attrp,
            char *const argv[restrict],
            char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
             const posix_spawn_file_actions_t *file_actions,
             const posix_spawnattr_t *restrict attrp,
             char *const argv[restrict],
             char *const envp[restrict]);
```

这两个函数的区别在于 posix_spawn( )有一个 path 参数，而 posix_spawnp( )有一个 file 参数。在 posix_spawn( )函数中的 path 参数是可执行程序文件的绝对或相对路径名。在 posix_spawnp( )中的 file 参数是可执行程序的名字。如果参数包含斜杠，那么 file 将被当作路径名来使用，如果不包含，则可执行文件的路径将由 PATH 环境变量确定。

5.4.1　file_actions 参数

file_actions 参数是到 posix_spawn_file_actions_t 结构体的指针：

```
struct posix_spawn_file_actions_t{
{
   int __allocated;
   int __used;
   struct __spawn_action *actions;
   int __pad[16];
};
```

posix_spawn_file_actions_t 是一种包含了新进程将要对文件描述符执行的动作信息的数据结构。file_actions 用于将父进程的打开的文件描述符集合更改为被创建的子进程的文件描述符集合。这个结构体可以包含多个将要执行的文件动作操作，这些动作执行的顺序是按照它们加入到被创建的文件动作对象的顺序。这些文件动作操作是在父进程的打开的文件描述符上进行的。这些操作可以代表子进程复制、复制并重置、增加、删除或关闭一个特定的文件描述符，即使在子进程被创建之前亦可。如果 file_actions 是空指针，那么由父进程打开的文件描述符对于子进程仍是打开的，没有任何改变。表 5-2 列出了用于将文件动作加入到 posix_spawn_file_actions 对象的函数。

表 5-2

文件动作属性函数	描　述
`int posix_spawn_file_actions_addclose (posix_spawn_file_actions_t *file_actions, int fildes);`	将 close( )动作加入到被创建的由 file_actions 指定的文件动作对象；这使得当使用这个文件动作对象产生新的进程时，文件描述符 fildes 会被关闭
`int posix_spawn_file_actions_addopen (posix_spawn_file_actions_t *file_actions, int fildes, const char *restrict path, int oflag, mode_t mode);`	将 open( )动作加入到被创建的由 file_actions 指定的文件动作对象；这导致当使用这个文件动作对象创建新的进程时，会打开由 path 和返回的文件描述符 fildes 确定的文件

(续表)

文件动作属性函数	描　述
`int posix_spawn_file_actions_adddup2 (posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);`	将 dup2( )动作加入到被创建的由 file_actions 指定的文件动作对象；这导致当使用这个文件动作对象产生新的进程时，文件描述符 fildes 会被文件描述符 newfildes 复制
`int posix_spawn_file_actions_destroy (posix_spawn_file_actions_t *file_actions);`	销毁指定的 file_actions 对象；这导致对象成为未初始化的，然后可以通过使用 posix_spawn_file_actions_init( )将对象再次初始化
`int posix_spawn_file_actions_init (posix_spawn_file_actions_t *file_actions);`	初始化指定的 file_actions 对象；一旦初始化之后，对象中将不包含待执行的文件动作

5.4.2 attrp 参数

attrp 参数指向 posix_spawnattr_t 结构：

```
struct posix_spawnattr_t
{
   short int __flags;
   pid_t __pgrp;
   sigset_t __sd;
   sigset_t __ss;
   struct sched_param __sp;
   int __policy;
   int __pad[16];
}
```

这个结构包含关于新进程的调度策略、进程组、信号、标志等信息。各个属性的描述如下：

- __flags：用于指示在生成的进程中将要修改哪个进程属性。它们是以下值中的 0 个或多个的按位异或：
  - POSIX_SPAWN_RESETIDS
  - POSIX_SPAWN_SETPGROUP
  - POSIX_SPAWN_SETSIGDEF
  - POSIX_SPAWN_SETSIGMASK

  - POSIX_SPAWN_SETSCHEDPARAM
  - POSIX_SPAWN_SETSCHEDULER
- __pgrp：新进程即将加入的进程组的 id。
- __sd：代表新进程被迫使用默认信号处理的信号集合。
- __ss：代表新进程将要使用的信号掩码。
- __sp：代表将要赋给新进程的调度参数。
- __policy：代表新进程将要使用的调度策略。

表 5-3 列出了用于设置和获取 posix_spawnattr_t 结构中各个属性的函数。

表 5-3

Spawn Process 属性函数	描　述
`int posix_spawnattr_getflags (const posix_spawnattr_t *restrict attr, short *restrict flags);`	返回保存在指定的 attr 对象中的__flags 属性的值
`int posix_spawnattr_setflags (posix_spawnattr_t *attr, short flags);`	将保存在指定的 attr 对象中的__flags 属性的值设置为 flags
`int posix_spawnattr_getpgroup (const posix_spawnattr_t *restrict attr, pid_t *restrict pgroup);`	返回保存在指定的 attr 对象中的__pgroup 属性的值，并将它保存到 pgroup 中
`int posix_spawnattr_setpgroup (posix_spawnattr_t *attr, pid_t pgroup);`	如果__flags 属性中的 POSIX_SPAWN_SETPGROUP 被设置，则将指定的 attr 对象中的__pgroup 属性的值设置为 pgroup
`int posix_spawnattr_getschedparam (const posix_spawnattr_t *restrict attr, struct sched_param *restrict schedparam);`	返回保存在指定 attr 对象的__sp 属性的值，并将它保存到 schedparam
`int posix_spawnattr_setschedparam (posix_spawnattr_t *attr, const struct sched_param *restrict schedparam);`	如果在__flags 属性中的 POSIX_SPAWN_SETSCHEDPARAM 被设置，则将保存在指定的 attr 对象中的__sp 属性的值设置为 schedparam
`int posix_spawnattr_getpschedpolicy (const posix_spawnattr_t *restrict attr, int *restrict schedpolicy);`	返回保存在指定的 attr 对象中__policy 属性的值，并将它保存到 schedpolicy

(续表)

Spawn Process 属性函数	描　　述
int posix_spawnattr_setpschedpolicy (posix_spawnattr_t *attr, int schedpolicy);	如果在__flags 属性中，POSIX_SPAWN_SETSCHEDULER 被设置，则将保存在指定的 attr 对象中的__policy 属性的值设置为 schedpolicy
int posix_spawnattr_getsigdefault (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigdefault);	返回保存在指定的 attr 对象中的__sd 属性的值，并将它保存在 sigdefault
int posix_spawnattr_setsigdefault (posix_spawnattr_t *attr, const sigset_t *restrict sigdefault);	如果在__flags 属性中，POSIX_SPAWN_SETSIGDEF 被设置，则将保存在指定的 attr 对象中的__sd 属性的值设置为 sigdefault
int posix_spawnattr_getsigmask (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigmask);	返回保存在指定的 attr 对象中的__ss 属性的值，并将它保存在 sigmask
int posix_spawnattr_setsigmask (posix_spawnattr_t *restrict attr, const sigset_t *restrict sigmask);	如果在__flags 属性中，POSIX_SPAWN_SETSIGMASK 被设置，则将保存在指定的 attr 对象中的__ss 属性的值设置为 sigmask
int posix_spawnattr_destroy (posix_spawnattr_t *attr);	销毁指定的 attr 对象；之后该对象可以通过 posix_spawnattr_init( )重新初始化
int posix_spawnattr_init (posix_spawnattr_t *attr);	使用在结构体中包含的所有属性的默认值来初始化指定的 attr 对象

5.4.3 简单的 posix_spawn( )示例

示例 5-1 显示了如何使用 posix_spawn( )函数来创建进程的方法。

示例 5-1

```
// Example 5-1 Spawns a process, using the posix_spawn()
// function that calls the ps utility.

#include <spawn.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
{
```

```
    //...
    `posix_spawnattr_t X;
    posix_spawn_file_actions_t Y;
    pid_t Pid;
    char * argv[] = {"/bin/ps","-lf",NULL};
    char * envp[] = {"PROCESSES=2"};
    posix_spawnattr_init(&X);
    posix_spawn_file_actions_init(&Y);
    posix_spawn(&Pid,"/bin/ps",&Y,&X,argv,envp);
    perror("posix_spawn");
    cout << "spawned PID: " << Pid << endl;
    //...
    return(0);

}
```

在示例 5-1 中，初始化了 posix_spawnattr_t 和 posix_spawn_file_actions_t 对象。使用参数 PID;path;Y;X;argv;envp 调用 posix_spawn( )，其中 argv 将命令作为第一个元素，而将参数作为第二个元素，envp 是环境列表。如果 posix_spawn( )成功，那么 pid 中保存的值将会是新生成的进程的 PID。perror 显示为：

```
posix_spawn: Success
```

而且 pid 将被发送到输出设备。在本例中，新生成的进程执行：

```
/bin/ps -lf
```

这些函数在 pid 参数中将子进程的进程 id 返回给父进程，并以 0 作为返回值。如果函数没有成功，则不会创建子进程，这样，不返回 pid，而且函数会以错误值作为返回值。在使用 spawn 函数时，错误可能会在 3 个级别上发生。

- 如果 file_actions 或 attr 对象无效，则会发生错误。如果这发生在函数成功返回之后(子进程已经被产生)，那么子进程会有着退出状态(127)。
- 如果 spawn attribute 函数导致错误，那么将返回为该特定函数生成的错误(列于表 5-2 和表 5-3 中)。如果 spawn 函数已经成功返回，那么子进程可能有着退出状态(127)。
- 错误也可能发生在您试图创建子进程时。这些错误将会与 fork( )或 exec( )函数生成的错误相同。如果这些错误发生，则它们会成为 spawn 函数的返回值。

如果子进程发生错误，它不会返回到父进程。如果父进程需要知道子进程发生错误，则必须使用其他机制，因为错误不会保存在子进程的退出状态中。可以使用进程间通信，或者子进程可以被设置一些对父进程可见的标志。

5.4.4　使用 posix_spawn 的 guess_it

程序清单 5-1 回顾了第 4 章的程序清单 4-1 中的“猜测神秘编码”中的代码，它新创

建了两个子进程。

程序清单 5-1

```
// Listing 5-1  Program used to launch ofind_code.

 1  using namespace std;
 2  #include <iostream>
 3  #include <string>
 4  #include <spawn.h>
 5  #include <sys/wait.h>
 6
 7  int main(int argc,char *argv[],char *envp[])
 8  {
 9
10     pid_t ChildProcess;
11     pid_t ChildProcess2;
12     int RetCode1;
13     int RetCode2;
14     int Value;
15     RetCode1 = posix_spawn(&ChildProcess,"find_code",NULL,
16                            NULL,argv,envp);
17     RetCode2 = posix_spawn(&ChildProcess2,"find_code",NULL,
18                            NULL,argv,envp);
19     wait(&Value);
20     wait(&Value);
21     return(0);
22  }
```

在示例 5-1 中，我们使用 posix_spawn 来运行 ps shell 实用工具。在程序清单 5-1 中，我们使用 posix_spawn 来运行 ofind_code 程序。这示范了 posix_spawn( )的一个重要特性，即可用于运行调用程序外部的程序。位于本地计算机上的任何程序都可以通过 posix_spawn( )来轻松地运行。在程序清单 5-1 的第 15 行和第 17 行调用的 posix_spawn( )有着简练的接口。在第 4 章中，我们引入了接口类的概念，它可以帮助您实现更具声明式风格的多核编程。接口类易于实现。程序清单 5-2 显示了一个简单的接口类，可用于封装 posix_spawn( )函数的基本要点。

程序清单 5-2

```
//Listing 5-2  An initial interface class for a posix process.

 1  #ifndef __POSIX_PROCESS_H
 2  #define __POSIX_PROCESS_H
 3  using namespace std;
 4
 5  #include <spawn.h>
 6  #include <errno.h>
 7  #include <iostream>
```

```
 8  #include <string>
 9
10
11  class posix_process{
12  protected:
13     pid_t  Pid;
14     posix_spawnattr_t   SpawnAttr;
15     posix_spawn_file_actions_t   FileActions;
16     char **argv;
17     char **envp;
18     string ProgramPath;
19  public:
20     posix_process(string Path,char **av,char **env);
21     posix_process(string Path,char **av,char **env, posix_spawnattr_t X,
                     posix_spawn_file_actions_t Y);
22     void run(void);
23     void pwait(int &X);
24  };
25
26
27  #endif
28
```

这个简单的接口类可用于为进程管理增加一种更加面向对象的方法。它使得从图 5-1 的阶段 2 中的模型转换到阶段 4 中的执行单元会更加容易，它还使得操作系统 API 调用对用户透明。例如，在清单 5-1 中显示的 guess_it 程序可以采用以下形式重写(见程序清单 5-3 所示)。

程序清单 5-3

```
//Listing 5-3  Our guess_it program using an interface class for the
posix_spawn capability.


 1  #include "posix_process.h"
 2
 3  int main(int argc,char *argv[],char *envp[])
 4  {
 5     int Value;
 6     posix_process  Child1("ofind_code",argv,envp);
 7     posix_process  Child2("ofind_code",argv,envp);
 8     Child1.run();
 9     Child2.run();
10     Child1.pwait(&Value);
11     Child2.pwait(&Value);
12     return(0);
13  }
14
```

回顾第 4 章，guess_it 程序新创建两个子进程，每个子进程依次创建两个线程，得到的四个线程用于搜索文件。接口类，作为将过程式范型转换为面向对象声明式方法的价值无以复加。一旦有了 posix_process 类，就可以将它作为数据类型用在容器类中。您可以这样写：

```
vector<posix_process>
list<posix_process>
multiset<posix_process>
etc...
```

将进程和线程视作对象，而不是动作序列，这是在通往并行编程的声明式模型的方向上迈出的一大步。程序清单 5-4 显示了 posix_process 接口类的初始化方法的定义。

程序清单 5-4

```
// Listing 5-4 The initial method definitions for the posix_process interface
class.

 1  #include "posix_process.h"
 2  #include <sys/wait.h>
 3
 4
 5  posix_process::posix_process(string Path,char **av,char **env)
 6  {
 7
 8     argv = av;
 9     envp = env;
10     ProgramPath = Path;
11     posix_spawnattr_init(&SpawnAttr);
12     posix_spawn_file_actions_init(&FileActions);
13
14
15  }
16
17  posix_process::posix_process(string Path,char **av,char **env,
                 posix_spawnattr_t  X, posix_spawn_file_actions_t Y)
18  {
19     argv = av;
20     envp = env;
21     ProgramPath = Path;
22     SpawnAttr = X;
23     FileActions = Y;
24     posix_spawnattr_init(&SpawnAttr);
25     posix_spawn_file_actions_init(&FileActions);
26
27
28
29  }
30
```

```
31  void posix_process::run(void)
32  {
33
34     posix_spawn(&Pid,ProgramPath.c_str(),&FileActions,
                   &SpawnAttr,argv,envp);
35
36
37  }
38
39  void posix_process::pwait(int &X)
40  {
41
42     wait(&X);
43  }
```

程序清单 5-4 的第 31 行中定义的 run( )方法改写了 posix_spawn( )函数的接口。您可以通过加入方法来改写表 5-2 和表 5-3 中列出的所有函数的接口。一旦完成之后，您可以将进程构建块加入到您的面向对象工具集中。

5.5　哪个是父进程，哪个是子进程

有两个函数用来返回当前进程和父进程的进程 id(PID)：

- getpid( )返回调用进程的进程 id。
- getppid( )返回调用进程的父进程 id。

这些函数总是会成功返回，因此没有定义错误。

调用形式

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

5.6　对进程的详细讨论

当执行进程时，操作系统将它指派到一个处理器上。进程将在一个时间片(quantum)内执行它的指令。进程是可抢占的，因此另外一个进程可以被指派到这个处理器上。操作系统调度器将进程、用户或系统的代码切换给另一个进程的代码，为每个进程提供执行其指令的机会。进程可分为系统进程和用户进程。

- 执行系统代码的进程被称作系统进程，有时候也被称作内核进程。系统进程用于管理整个系统。它们执行内部管理(housekeeping)任务，例如分配内存、在内部和

二级存储器之间交换内存页面、检测设备等。它们还为用户进程执行任务，例如满足I/O请求、分配内存等。

- 用户进程可执行它们自己的代码，有时候也会进行系统函数的调用。当一个用户进程执行它自己的代码时，它将处于用户模式。在用户模式下，进程不能够执行某些特权机器指令。当用户进程发出系统函数调用(例如，read( )、write( )或open( ))时，它将执行操作系统指令。用户进程将会暂停，直到系统调用结束。处理器交给内核来完成系统调用。此时，我们说用户进程处于内核模式，不能够被任何用户进程抢占。

5.6.1 进程控制块

进程拥有一些用来标识它们并决定运行期间的行为的特性。内核维护数据结构并提供允许用户访问这些信息的系统函数。有些信息保存在进程控制块(process control block，PCB)中。PCB中保存的信息将向操作系统描述进程。PCB是进程的一部分。操作系统需要这些信息来管理每个进程。当操作系统在当前利用CPU的进程同另一个进程间进行切换时，它将当前执行进程的状态以及上下文保存到PCB的保存区中，这样当进程下一次被指派到CPU时可以重新启动。PCB可被操作系统的多个模块读取和更改。涉及监视操作系统性能、调度、资源分配和中断处理的模块都会访问和/或更改PCB。通过PCB，使得进程对操作系统可见，而类似用户线程的实体对操作系统不可见。

PCB中的信息包括：

- 进程的当前状态和优先级
- 进程标识符、父进程标识符和子进程标识符
- 指向已分配资源的指针
- 指向进程内存位置的指针
- 指向进程的父进程和子进程的指针
- 进程所使用的处理器
- 控制和状态寄存器
- 栈指针

PCB中保存的信息可以按照如下方式来组织：

- 同进程控制相关的信息，例如进程的当前状态和优先级、指向父/子PCB的指针、指向已分配资源和内存的指针。还包括与进程间通信(IPC)相关的任何调度相关的信息、进程权限、标志、消息和信号。操作系统需要进程控制信息，从而才能够协调并发的活动进程。
- 与处理器状态相关的用户上下文、控制/状态寄存器和栈指针。当进程运行时，信息被置于CPU的寄存器中。一旦操作系统决定切换到另一个进程，这些寄存器中的所有信息都必须被保存下来。当进程再次获得对CPU的使用权时，这些信息将被恢复。

- 同进程标识相关的其他信息。这包括进程 id(PID)和父进程 id(PPID)。这些标识数字对于每个进程是唯一的，它们是正整数。

5.6.2 进程的剖析

进程的地址空间分为 3 个逻辑段：代码段、数据段和栈段。图 5-2 显示了进程的逻辑布局。地址空间的底部是代码段，该段包含被称作程序代码的待执行指令。位于代码段之上的数据段包含该进程的被初始化的全局变量、外部变量和静态变量。栈段包含分配的局部变量和传递给函数的参数。由于进程既可以进行系统函数调用，也可以进行用户定义函数调用，所以在栈段中维护着两个栈，分别是用户栈和内核栈。当发起函数调用时，会构建一个栈帧，根据进程处于用户模式还是内核模式，将栈帧放置到用户栈或内核栈。栈段的增长方向是向下朝着数据段。当函数返回时，栈帧从栈中弹出。代码段、数据段和栈段以及进程控制块共同构成了进程映像(process image)。

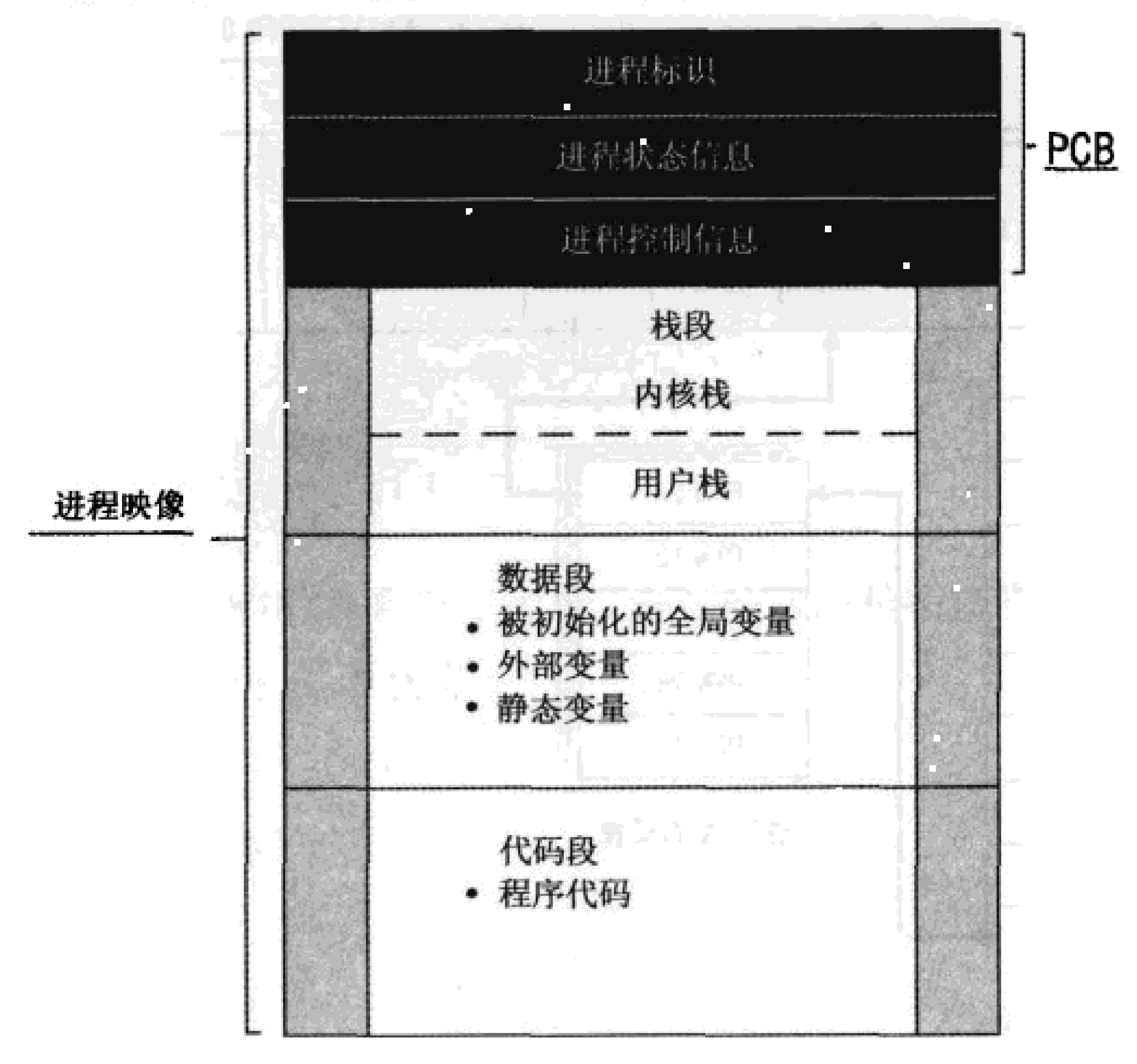


图 5-2

进程的地址空间是虚拟的(virtual)。虚拟存储使得在执行的进程中引用的地址同内部内存中的实际可用地址是无关的。这使得可寻址的存储空间的大小远远大于实际内存大小。进程的虚拟地址空间的段是连续的内存块。每个段以及物理地址空间被分成被称为页(page)的区块(chunk)。每个页有着唯一的页帧号(page frame number)。虚拟页帧号(virtual page frame number，VPFN)用作进程的页表中的索引。页表中的每一项中包含物理页帧号，这样将虚拟页帧映射到物理页帧。如图 5-3 所示，虚拟地址空间是连续的，但可以以任何顺序映射到物理页面。

虽然每个进程的虚拟地址空间都受到保护，以防止其他进程访问它，但是进程的代码

段可以在多个进程之间共享。图 5-3 还显示了两个进程如何共享相同的程序代码。相同的页帧号保存在两个进程的页表的表项中。如图 5-3 所示，进程 A 的虚拟页帧 0 映射到物理页帧 5，而进程 B 的虚拟页帧 2 也映射到物理页帧 5。

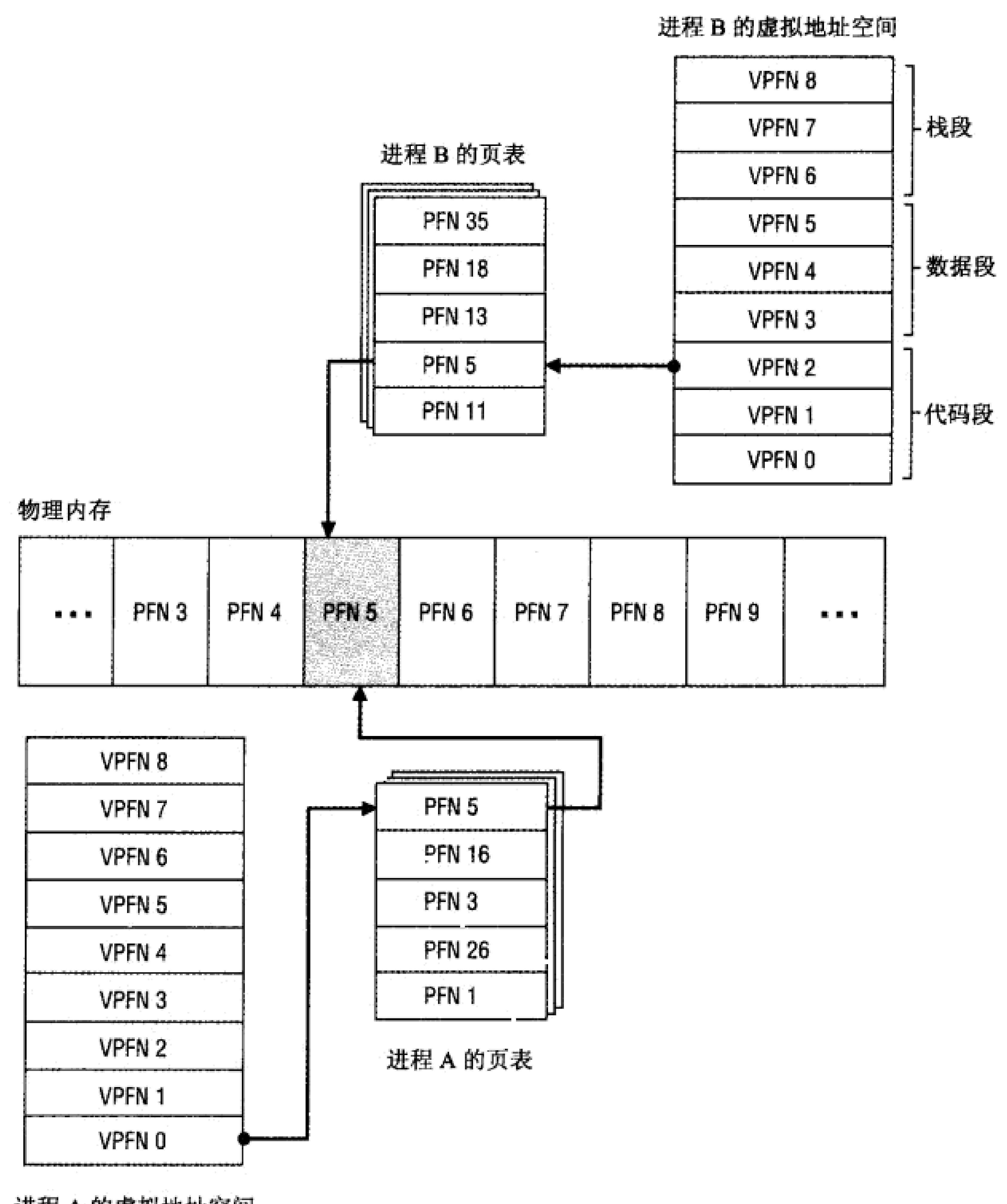


图 5-3

操作系统为了管理保存在内存中的所有进程，它将创建并维护进程表。实际上，操作系统拥有一个表，内容包括它所管理的所有实体。要知道操作系统不仅管理进程，而且还管理所有的计算机资源，包括设备、内存和文件。有些内存、设备和文件是为了用户进程而管理的。这些信息会在将资源分配给进程时被引用到 PCB 中。内存中的每个进程映像在进程表中均有一个条目，如图 5-4 所示。每个条目包含进程 id 和父进程 id，实际有效用户 id 和组 id，未决信号(pending signal)列表，代码段、数据段和栈段的位置，进程的当前状

态。当操作系统需要访问一个进程时，会在进程表中查找该进程，然后再在内存中定位进程映像。

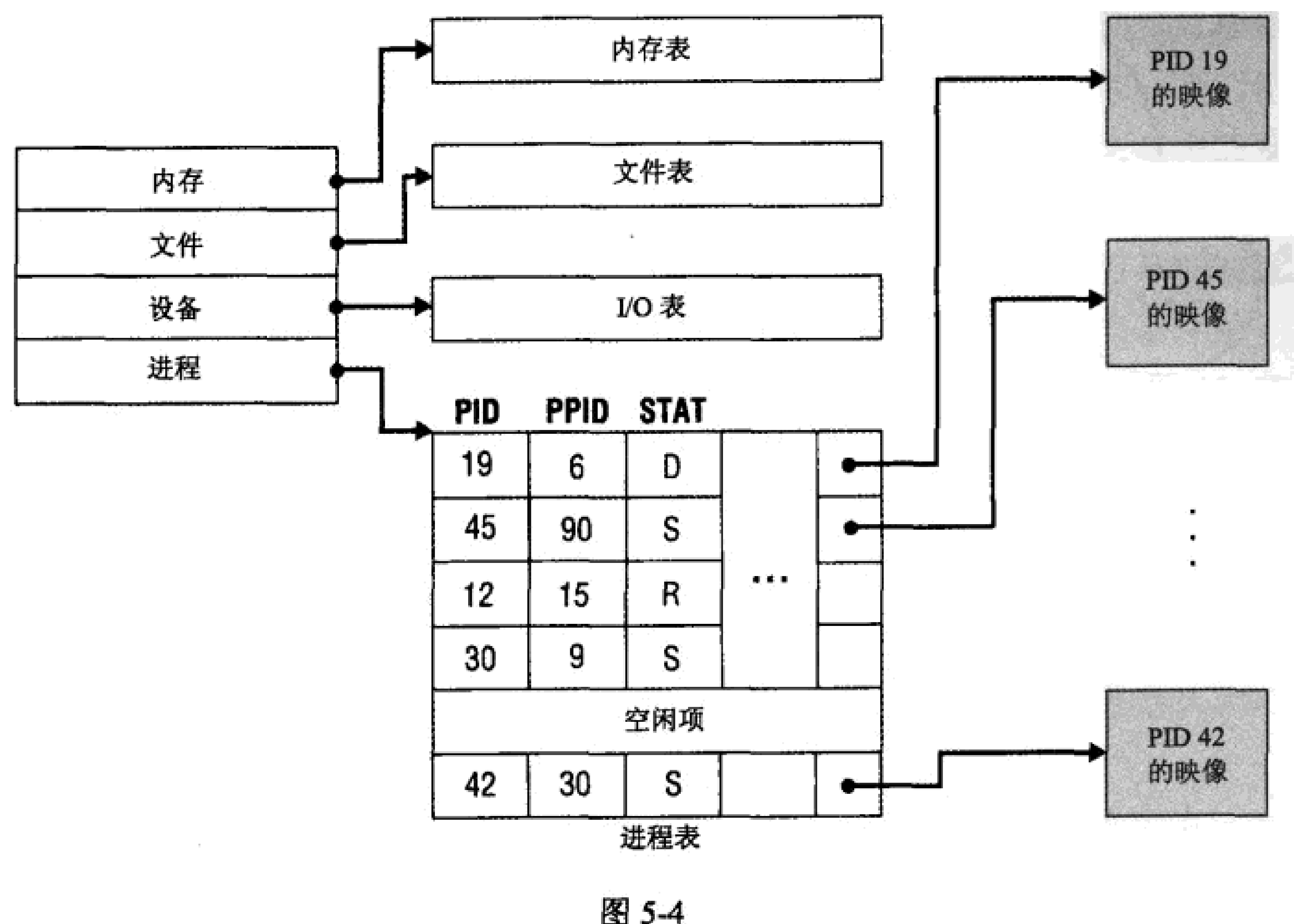


图 5-4

5.6.3　进程状态

在进程的执行期间，它的状态会发生改变。进程的状态是指进程的当前情况。在 POSIX 兼容的环境中，进程可以处于以下状态：

- 运行(running)
- 就绪(runnable，ready)
- 僵死(zombied)
- 等待(waiting，blocked)
- 停止(stopped)

进程的当前状况取决于进程或操作系统所创造的境况(circumstance)。当特定境况出现时，进程将改变它的状态。状态转换(state transition)是导致进程改变其状态的境况。图 5-5 是进程的状态图，状态图由节点以及节点之间的有向边组成。每个节点代表进程的状态，节点之间的有向边是状态转换。表 5-4 列出了状态转换以及简要描述。如图 5-5 和表 5-4 所示，状态之间只能发生特定转换。例如，在就绪和运行之间存在转换，但是在休眠与运行之间不存在转换。这意味着有一些境况能够导致一个进程从就绪状态转到运行状态，但是没有哪个境况能够使得进程从休眠状态转到运行状态。

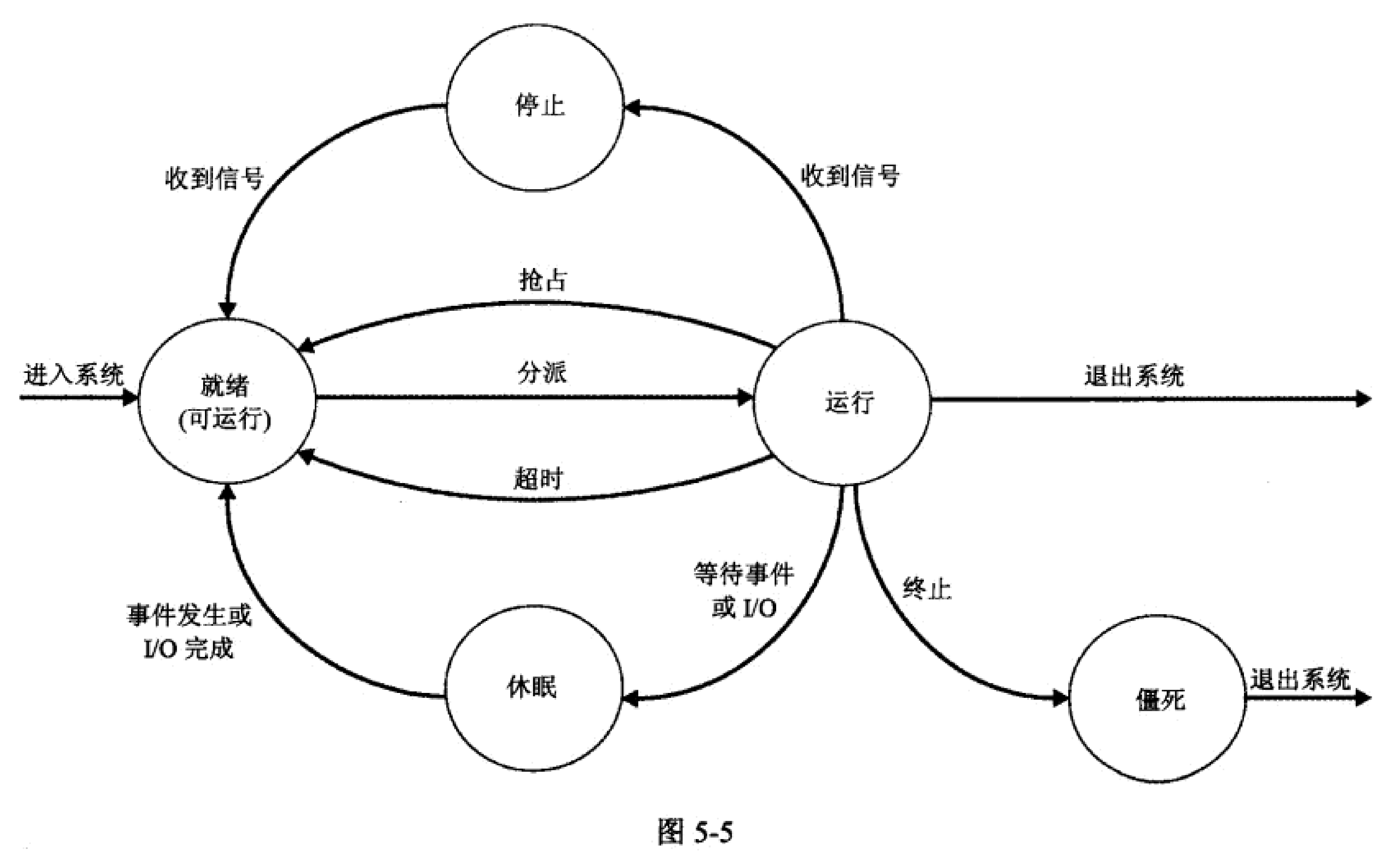


图 5-5

表 5-4

状 态 转 换	描　　述
就绪→运行(分派)	进程被指派到处理器
运行→就绪(超时)	处理器指派给进程的时间片用完，进程被放回到就绪队列
运行→就绪(抢占)	在时间片用完之前，进程被抢占；当有着更高优先级的进程就绪时就可能发生这种情况；进程会被放回到就绪队列
运行→休眠(阻塞)	在时间片用完之前，进程放弃对处理器的使用；进程可能需要等待某个事件或者已经进行了系统调用，例如 I/O 请求；进程同其他休眠进程被放入到一个队列中
休眠→就绪(阻塞解除)	进程等待的事件已经发生，或系统调用结束；例如，I/O 请求被满足；进程被重新放入到就绪队列
运行→停止	由于收到了停止信号，进程放弃了处理器
停止→就绪	进程收到了继续信号，然后被放入到就绪队列中
运行→僵死	进程已经结束，等待父进程从进程表中提取其退出状态
僵死→退出	父进程已经提取了退出状态，进程退出系统
运行→退出	进程已经停止，父进程已经提取了它的退出状态，而且进程退出系统

当创建了一个进程后，它就已经准备好执行它的指令了，但是它首先必须等待，直到处理器可用。每个进程只允许使用处理器一小段时间，被称作时间片(time slice)。等待使用处理器的进程位于就绪队列中，只有就绪队列中的进程会被调度器选中来使用处理器。

就绪队列中的进程是可运行的(runnable)。当处理器可用时，会由分派器(dispatcher)指派一个可运行的进程给处理器。当时间片用完之后，无论它是否运行完它的全部指令，进程都将被从处理器上移除。进程会被放回到就绪队列中，等待下一次轮到它使用处理器。从就绪队列中选出一个新的进程并给它时间片来执行。系统进程是非抢占式的，当它们占用处理器后，会一直运行到结束。如果时间片没有用完，进程如果因为等待某事件的发生而不能够继续执行，也可能主动放弃处理器。进程也可能通过发起系统调用来发出访问 I/O 设备的请求，或者可能需要等待一个同步变量被释放。由于等待事件发生而不能够继续执行的进程会处于休眠状态，它会同其他休眠进程一同被放到一个队列中。当事件发生之后，会从该队列中将它们删除，并放回到就绪队列中。在时间片被用完之前，进程所使用的处理器也可能会被夺走，这发生在有着更高优先级的进程可运行时，如系统进程。被抢占的进程仍然是可运行的，因此会被放回到就绪队列中。

运行中的进程可能会接收到一个停止信号。停止状态同休眠状态是不同的，进程的时间片没有用完，而且进程也没有发起任何系统请求。进程可能会接收到停止信号，是因为它正在被调试，或者在系统中发生了某种情况。进程接收到信号后，会进行从运行状态到停止状态的转换，稍后进程可能会被唤醒，或者可能会被销毁。

当进程执行完所有的指令之后，它会退出系统。进程会从进程表中被删除，PCB 将被销毁，它的所有资源将被释放并退回到系统可用资源池中。不能够继续执行但也不能够退出系统的进程的状态被称为僵死状态。僵死进程不使用任何系统资源，但是仍然在进程表中占用一个条目。当进程表中包含过多的僵死进程时，会影响系统的性能，甚至可能导致系统重新启动。

5.6.4　进程是如何被调度的

当一个就绪队列包含多个进程时，调度器必须决定首先将哪个进程指派给处理器。调度器维护使得它可以以高效的方式调度进程的数据结构。每个进程将被赋予一个优先级别，并同其他有着相同优先级别的可运行进程放置在同一个优先级队列中。存在多个优先级队列，每个队列代表系统所使用的不同的优先级别。这些优先级队列是分不同等级的，而且被放置在一个名为多优先级队列(multilevel priority queue)的分派数组(dispatch array)中。图 5-6 描述了多优先级队列。数组中的每个元素指向一个优先级队列。调度器将位于非空最高优先级队列头部的进程指派给处理器。

优先级可以是动态的或静态的。一旦进程的静态优先级被设置，则不能够改变它，而动态优先级则是可以改变的。有着最高优先级的进程可以垄断对处理器的使用。如果一个进程的优先级是动态的，最初的优先级可以被调整为更合适的值。进程会被放置到有着更高优先级的优先级队列。还可以给垄断处理器的进程设置较低的优先级，或者把其他进程的优先级设置得更高一些。当您为用户进程设置优先级时，需要考虑该进程多数时间是做什么类型的工作。有些进程是 CPU 密集型的，这些进程在整个时间片内都使用处理器，有些进程则将多数时间用于等待 I/O 或一些其他事件的发生。当这样的一个

进程准备好使用处理器时，应当立即将处理器交给它使用，这样它能够处理下一个 I/O 请求。交互进程可能会要求高优先级来保证良好的响应时间。系统进程的优先级要高于用户进程。

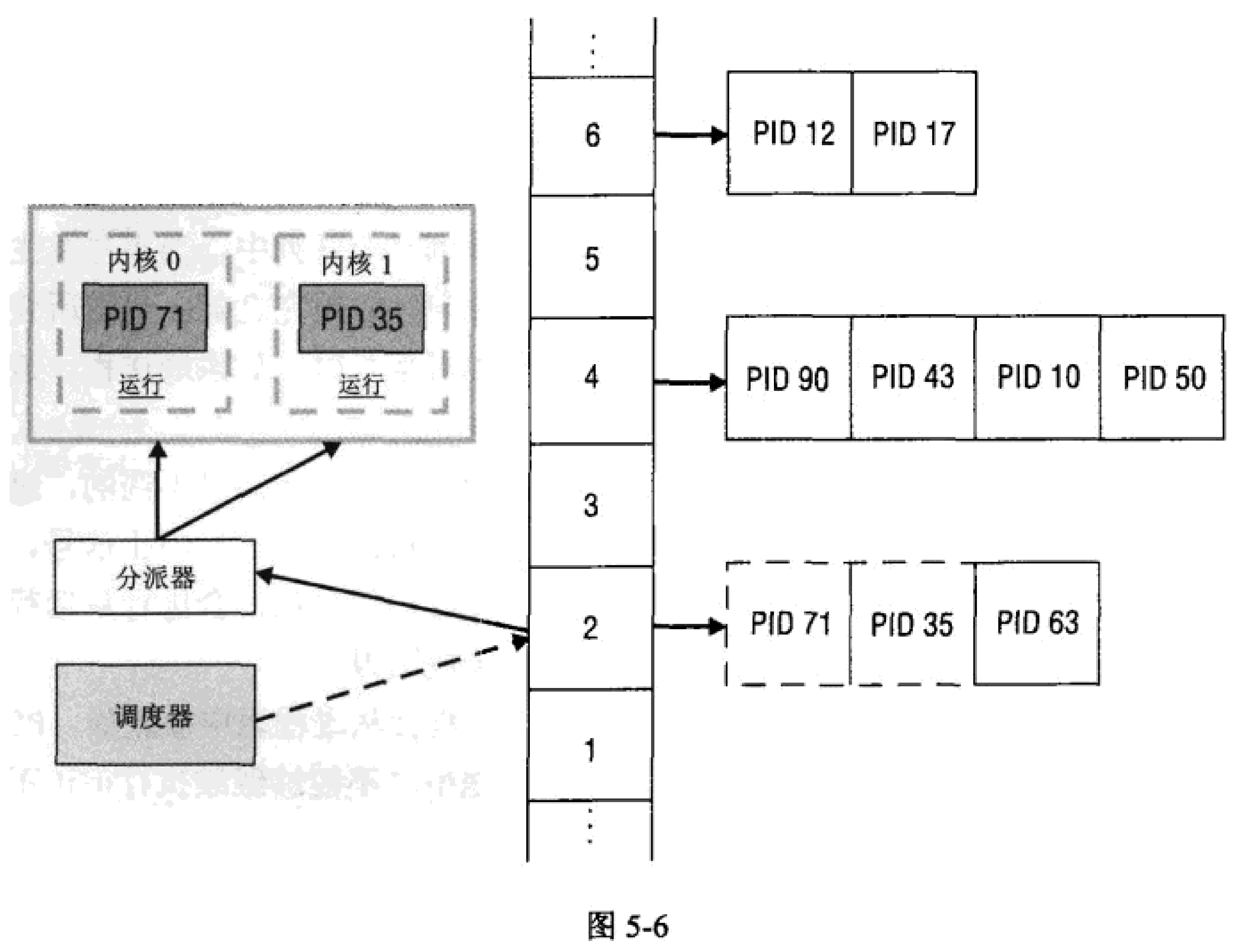


图 5-6

进程是根据调度策略放置在优先级队列中的。在 POSIX API 中使用的两个主要调度策略是 FIFO(First-In，First-Out，先进先出)和 RR(round robin，轮询)策略。

- 图 5-7(a)显示了 FIFO 调度策略。使用 FIFO 调度策略时，进程是根据到达队列的时间被指派给处理器的。当正在运行的进程的时间片耗尽时，它会被放置到优先级队列的头部。当一个休眠进程变为可运行时，它将被放置在优先级队列的尾部。进程可以发起系统调用并放弃处理器，将处理器交给有着相同优先级等级的另一个进程，然后这个进程会被放置到优先级队列的尾部。
- 在轮询调度策略下，所有进程都被同等对待。图 5-7(b)描述了 RR 调度策略。RR 调度和 FIFO 的区别在于：当时间片耗尽时，进程被放置到队列的后端，同时队列中的下一个进程被指派给处理器。除了这一点之外，RR 和 FIFO 相同。

图 5-7 显示了 FIFO 和 RR 调度策略的行为。FIFO 调度策略根据进程到达队列的时间将它们指派到处理器，进程将一直运行，直到结束。RR 调度策略使用 FIFO 调度指派线程，但是当时间片耗尽时，进程将被放置到就绪队列的尾部。

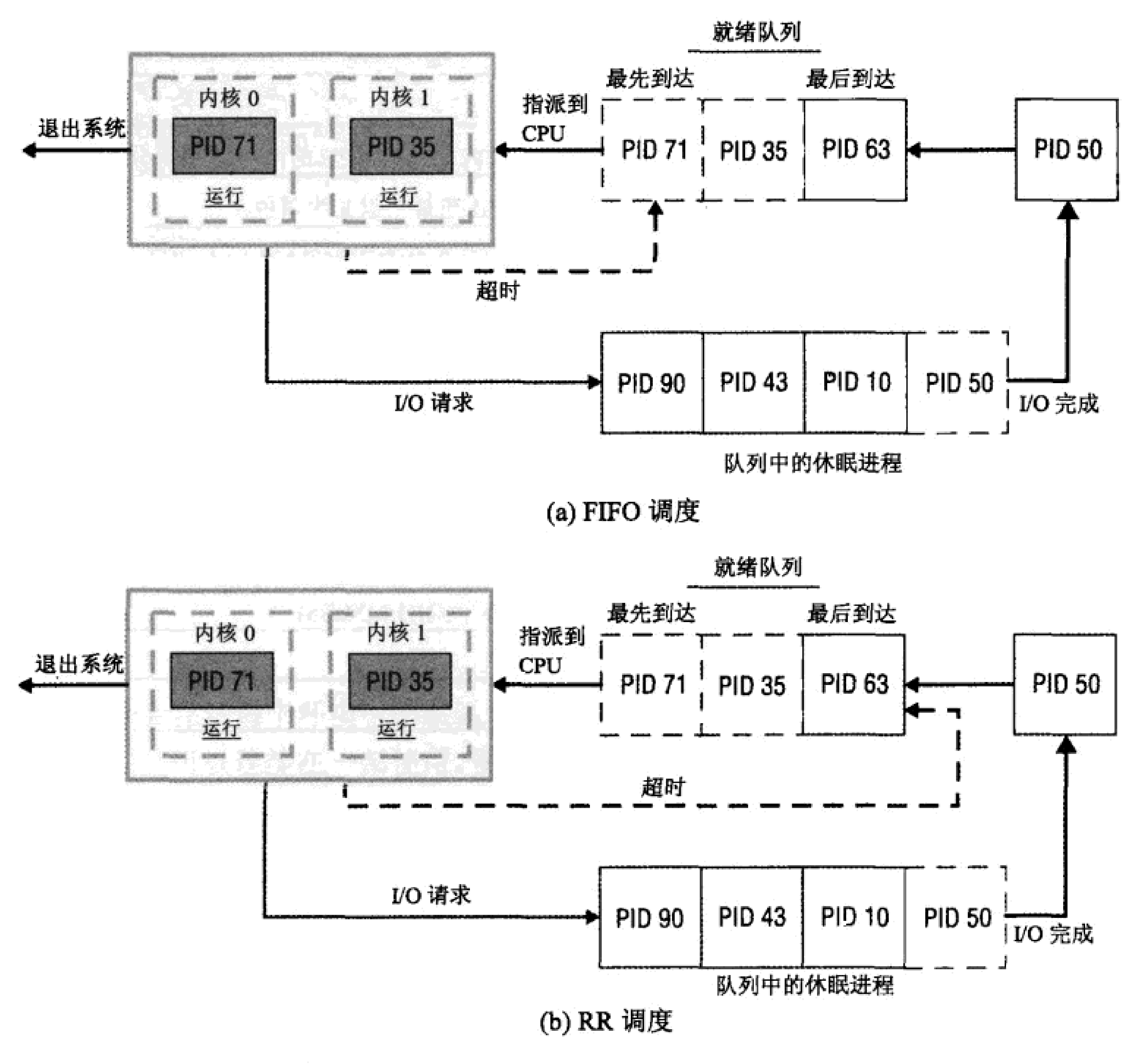


图 5-7

5.7　使用 ps 实用工具监视进程

实用工具 ps 能够生成一个汇总当前进程执行情况的统计报表。这个信息可用于监视当前进程的状态。表 5-5 列出了在 Solaris/Linux 环境中，ps 实用工具的输出中的常见标头及其含义。

表 5-5

标　　头	描　　述
USER, UID	进程所有者的用户名
PID	进程 ID
PPID	父进程 ID
PGID	进程组头领进程(leader)的 ID

(续表)

标　　头	描　　述
SID	会话头领进程的 ID
%CPU	进程在刚刚过去的一分钟内使用的 CPU 时间所占的百分比
RSS	进程当前使用的实际 RAM 数量，以 k 为单位
%MEM	进程在刚刚过去的一分钟内使用的实际 RAM 所占的百分比
SZ	进程的数据和栈使用的虚拟内存的大小，以 k 或页为单位
WCHAN	导致进程休眠的事件的地址
COMMAND	命令名以及参数
CMD	
TT, TTY	进程的控制终端(controlling terminal)
S, STAT	进程的当前状态
TIME	进程所使用的总的 CPU 时间(HH:MM:SS)
STIME, START	进程启动的时间或日期
NI	进程的 nice 值
PRI	进程的优先级
C, CP	短期内调度器计算 PRI 的 CPU 使用率
ADDR	进程的内存地址
LWP	lwp(线程)的 ID
NLWP	lwp 的个数

在多处理器环境中，ps 实用工具在监视状态、CPU 和内存使用、处理器使用、优先级、当前进程执行的启动时间方面很有帮助。命令选项用于控制列出哪些进程以及为每个进程显示哪些信息。在 Solaris 环境中，在默认情况下(即不使用任何命令行选项)，与调用发起者(calling invoker)有着相同有效用户 id 和控制终端的进程会被显示。在 Linux 环境中，在默认情况下，与调用者有着相同用户 id 的进程会被显示。在以上两种环境中，显示出来的信息默认只有 PID、TTY、TIME 和 COMMAND。有一些选项用于控制哪些进程将被显示。

- -t term：列出与被 term 指定的终端关联的进程
- -e：所有当前进程
- -a：(Linux)除了会话头领进程之外，拥有 tty 终端的所有进程
- (Solaris)除了组头领进程以及不同终端关联的进程以外的最频繁被请求的进程
- -d：除了会话头领进程之外的所有当前进程
- T：(Linux)本终端内的所有进程
- a：(Linux)包含其他用户的进程的所有进程
- r：(Linux)只有运行进程

调用形式

```
(Linux)
ps -[Unix98 options]
   [BSD-style options]
   --[GNU-style long options

(Solaris)
ps [-aAdeflcjLPy][-o format][-t termlist][-u userlist]
   [-G grouplist][-p proclist][-g pgrplist][-s sidlist]
```

下面列出了用于控制显示进程哪些信息的命令选项。

- -f：完整列表
- -l：长格式
- -j：作业格式

下面是在 Solaris/Linux 环境中使用 ps 实用工具的实例：

```
ps -f
```

该命令会显示每个环境中的默认进程的信息。图 5-8 显示了在 Solaris 环境中的输出。命令行选项也可以连接在一起使用。图 5-8 显示了在 Solaris 环境中同时使用-l 和-f 的输出。

```
ps -lf
```

```
//SOLARIS

$ ps -f
     UID    PID   PPID  C    STIME    TTY     TIME CMD
 cameron   2214   2212  0 21:03:35 pts/12     0:00 -ksh
 cameron   2396   2214  2 11:55:49 pts/12     0:01 nedit

$ ps -lf
F S     UID    PID   PPID  C PRI NI     ADDR  SZ    WCHAN    STIME    TTY  TIME   CMD
8 S cameron   2214   2212  0  51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00  -ksh
8 S cameron   2396   2214  1  53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit
```

图 5-8

命令选项-l 显示了额外的 F、S、PRI、NI、ADDR、SZ 和 WCHAN 标头。

命令选项 P 显示了 PSR 标头。在这个标头下是进程被指派或绑定的处理器的编号。

图 5-9 显示了在 Linux 环境中使用 Tux 命令选项时 ps 实用工具的输出。

```
//Linux

[tdhughes@colony]$ ps Tux
USER        PID %CPU %MEM   VSZ   RSS   TTY   STAT   START   TIME COMMAND
tdhughes 19259  0.0  0.1  2448  1356  pts/4    S     20:29   0:00 -bash
tdhughes 19334  0.0  0.0  1732   860  pts/4    S     20:33   0:00 /home/tdhughes/pv
tdhughes 19336  0.0  0.0  1928   780  pts/4    S     20:33   0:00 /home/tdhughes/pv
tdhughes 19337 18.0  2.4 26872 24856  pts/4    R     20:33   0:47 /home/tdhughes/pv
tdhughes 19338 18.0  2.3 26872 24696  pts/4    R     20:33   0:47 /home/tdhughes/pv
tdhughes 19341 17.9  2.3 26872 24556  pts/4    R     20:33   0:47 /home/tdhughes/pv
tdhughes 19400  0.0  0.0  2544   692  pts/4    R     20:38   0:00 ps Tux
tdhughes 19401  0.0  0.1  2448  1356  pts/4    R     20:38   0:00 -bash
```

图 5-9

进程的%CPU、%MEM 和 STAT 信息也会显示出来。在多处理器环境中，这个信息可用来监视哪个进程占用了大部分的 CPU 和内存。STAT 标头下显示了进程的状态。表 5-6 列出了状态编码以及它们的含义。

表 5-6

进 程 状 态	描　　述
D	不可中断休眠(通常是 I/O)
R	运行中或可运行(位于运行队列)
S	可中断休眠(等待一个事件的完成)
T	由于作业控制信号或由于被跟踪而停止
Z	“僵死”进程，已终止且没有父进程

STAT 标头可以揭示出进程状态的其他信息。

- D：(BSD)磁盘等待
- P：(BSD)页等待
- X：(System V)Growing：等待内存
- W：(BSD)被换出
- K：(AIX)可用的内核进程
- N：(BSD)Niced：执行优先级被降低
- >：(BSD)Niced：执行优先级人工提升
- <：(Linux)高优先级进程
- L：(Linux)页被锁在内存中

这些编码位于状态编码前面。如果 N 位于状态之前，则意味着进程运行在较低优先级上。如果进程状态为 S 小于 W，意味着进程正在休眠、被换出，而且有着高优先级别。

5.8　设置和获得进程优先级

进程的优先级可以通过使用 nice( )函数进行修改。每个进程都有一个 nice 值，用于计算调用进程的优先级。进程会继承创建它的进程的优先级，但是进程的优先级可以通过提高它的 nice 值来降低。只有超级用户和内核进程可以提高优先级。

调用形式

```
#include <unistd.h>

int nice(int incr);
```

较低的 nice 值会提高进程的优先级。参数 incr 是加到调用进程当前 nice 值上的值，incr 可以为负数或正数，nice 值是一个非负数。正的 incr 值加大了 nice 值，这样就降低了优先

级，负的 incr 值减小了 nice 值，这样就提升了优先级。如果 incr 值使得 nice 值高于或低于其界限，那么进程的 nice 值会相应地被设置为其上界或下界。如果成功，则 nice( )函数返回进程的新的 nice 值。如果不成功，则返回-1，同时 nice 值不变。

调用形式

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

函数 setpriority( )设置进程、进程组或用户的 nice 值，getpriority( )返回进程、进程组或用户的优先级。示例 5-2 显示了函数 setpriority( )和 getpriority( )设置和返回当前进程的 nice 值的句法。

示例 5-2

```
//Example 5-2 shows how setpriority() and getpriority() can be used.

#include <sys/resource.h>

//...
id_t pid = 0;
int which  = PRIO_PROCESS;
int value = 10;
int nice_value;
int ret;

nice_value = getpriority(which,pid);
if(nice_value < value){
   ret = setpriority(which,pid,value);
}
//...
```

在示例 5-2 中，调用进程的优先级被返回并被设置。如果调用进程的 nice 值小于 10，那么将进程的 nice 值设置为 10。目标进程由参数 which 和 who 中保存的值来决定。参数 which 可以指定一个进程、进程组或用户。它的值可以为：

- PRIO_PROCESS：指示一个进程
- PRIO_PGRP：指示一个进程组
- PRIO_USER：指示一名用户

根据 which 的值，参数 who 为进程、进程组或实际用户的 id 号。在示例 5-2 中，which 被设置为 PRIO_PROCESS。who 的值为 0 指示当前进程、进程组或用户。在示例 5-2 中，who 被设置为 0，指示当前进程的 setpriority( )值将会成为指定的进程、进程组或用户的 nice 值。

在 Linux 环境中，nice 值的范围是-20～19。在示例 5-2 中，如果当前 nice 值小于 10，

则会被设置为 10。与函数 nice( )的工作原理不同，传递给 setpriority( )的值是 nice 的实际值，而不是要加到当前 nice 值的偏移量。在有着多个线程的进程中，对优先级的修改会影响该进程中所有线程的优先级。如果成功，getpriority( )将返回指定进程的值。如果成功，setpriority( )返回 0。如果不成功，两个函数均返回-1。返回值-1 是一个进程的合法 nice 值。为了确定是否发生了错误，需要检查外部变量 errno。

5.9 什么是上下文切换

上下文切换发生在处理器的使用从一个进程切换到另一个进程时。当发生上下文切换时，系统保存当前运行进程的上下文并恢复下一个要使用处理器的进程的上下文。被抢占的进程的 PCB 会被更新，其中的进程状态域从运行状态变为适当的状态(可运行、阻塞、僵死等)。处理器的寄存器、栈状态、用户和进程的标识及权限、调度和计数信息等内容都会被保存和更新。

系统必须记录进程的 I/O 和其他资源，以及任何内存管理数据结构的状态。被抢占的进程会被放置到适当的队列中。

上下文切换发生在：

- 进程被抢占时
- 进程主动让出处理器时
- 进程发起 I/O 请求或需要等待某个事件时
- 进程从用户模式切换到内核模式时

当被抢占的进程被选中再次使用处理器时，系统将恢复它的上下文，并从它上一次停止的位置开始继续执行。

5.10 进程创建中的活动

要想运行任何一个程序，操作系统首先必须创建一个进程。当创建了新的进程之后，在主进程表(main process table)中会加入一个新的条目，创建并初始化一个新的 PCB。PCB 中进程标识部分会包含一个唯一的进程 id 号以及父进程 id。程序计数器被设置为指向程序的入口点，系统栈指针将被设置，从而为进程定义栈边界。系统会用所有要求的属性初始化进程。如果没有给进程提供优先级，则该进程默认具有最低的优先级。进程一开始并不拥有任何资源，除非有对资源的显式请求或从创建进程(creator process)继承而来。该进程的状态是可运行，而且会被放到可运行或就绪队列，系统会为进程分配地址空间。设置多少空间可以基于进程的类型默认地决定，也可以根据进程创建者的请求来设置。创建进程(creator process)可以在创建进程时将地址空间的大小传递给系统。

5.10.1　使用 fork( )函数调用

除了 posix_spawn( )，POSIX API 在创建进程时还支持 fork/exec 函数。这些函数在所有的 Unix/Linux 衍生产品中均可用。fork( )调用创建了一个新的进程，它是调用进程(即父进程)的副本。如果成功，fork( )会返回两个值，分别传给父进程和子进程。它将 0 返回给子进程，并将子进程的 PID 返回给父进程。父进程和子进程从紧接 fork( )调用之后的指令继续执行。如果没有成功，意味着没有创建进程，会将-1 返回给父进程。

调用形式

```
#include <unistd.h>

pid_t fork(void);
```

如果系统没有资源来创建另一个进程，则 fork( )会失败。如果进程可以生成的子进程或者系统范围内执行进程的数目有限制，而且已经超过了该限制，那么 fork( )会失败，并将设置 errno 以指示错误。

5.10.2　使用 exec( )系统调用系列

exec( )函数系列使用一个新的进程映像将调用进程的映像替换掉。fork( )调用创建一个新的进程，该进程是父进程的复制，而 exec( )函数用新的进程映像来替换复制的进程映像。新的进程映像是一个正常的可执行文件，而且会立即被执行。可执行程序可以使用路径或文件名来指定。这些函数可以将命令行参数传递给新的进程，还可以指定环境变量。如果函数没有成功，则没有返回值，因为包含对 exec( )的调用的进程映像已经被覆盖。如果函数未能成功，则会返回-1 到调用进程。

所有的 exec( )函数会在以下情况下失败。

- 许可被拒绝
  - 对可执行程序的文件目录的查找许可被拒绝
  - 可执行文件的执行许可被拒绝
- 文件不存在
  - 可执行文件不存在
  - 目录不存在
- 文件不可执行
  - 文件不能够执行，因为要被另一个进程写入而处于打开状态
  - 文件不是一个可执行文件
- 符号链接有问题
  - 当解析可执行文件的路径名时遇到的符号链接存在循环
  - 符号链接导致到可执行文件的路径名过长

exec( )函数和 fork( )一起使用。fork( )使用父进程的副本创建并初始化子进程，然后子进程通过调用 exec( )来替换其进程映像。示例 5-3 显示了 fork-exec 的用法示例。

示例 5-3

```
// Example 5-3 Using the fork-exec system calls.

//...
RtValue = fork();
if(RtValue == 0){
   execl("/home/user/direct","direct",".");
}
```

在示例 5-3 中，调用 fork( )函数并将返回值保存到 RtValue。如果 RtValue 为 0，那么它是子进程，调用 execl( )函数。第一个参数是到可执行模块的路径，第二参数是执行语句，第三个参数是执行语句的参数。例子中的 direct 是一个实用工具，它用来列出指定目录下所有的目录和子目录，在本例中，指定的目录是当前目录。共有 6 个版本的 exec 函数，每种都有不同的调用约定和用法，接下来将会对它们进行讨论。

1. execl( )函数

函数 execl( )、execle( )和 execlp( )将命令行参数作为列表来传递。命令行参数的数目应当在编译时就已知，这样这些函数才能够使用。

- `int execl (const char *path, const char *arg0, … /*,(char *0) */);`
  其中 path 参数是到可执行程序的路径名。可以被指定为绝对路径或者是当前路径的相对路径。下一个参数是命令行参数的列表：arg0～arg*n*。一共可以有 *n* 的参数，列表最后跟随一个 NULL 指针。
- `int execle(const char *path, const char *arg0,.../*,(char *)0 *, char *const envp[]*/);`
  这个函数除了增加了一个参数 envp[ ]以外，其他均与 execl( )相同。这个参数包含新进程的新的环境。envp[ ]是一个指向以 null 结束的数组的指针，数组中是以 null 结束的字符串。每个字符串形如：

  ```
  name = value
  ```

  其中 name 是环境变量的名称，value 是将要保存的字符串。envp[ ]可以使用以下的形式来赋值：

  ```
  char *const envp[] = {"PATH=/opt/kde5:/sbin", "HOME=/home",NULL};
  ```

  其中的 PATH 和 HOME 是环境变量。

- `int execlp(const char *file,const char *arg0,.../*,(char *)0 */);`
  其中 file 是可执行程序的名称，它使用 PATH 环境变量来定位可执行文件。其余的参数是命令行参数列表，就像 execl( )函数中所解释的那样。

下面是使用这些参数的 execl( )函数系列的句法示例：

```
char *const args[] = {"direct",".",NULL};
char *const envp[] = {"files=50",NULL};

execl("/home/tracey/direct","direct",".",NULL);
execle("/home/tracey/direct","direct",".",NULL,envp);
execlp("direct","direct",".",NULL);
```

这些都显示了 execl( )函数如何创建进程来执行 direct 程序的句法。

调用形式

```
#include <unistd.h>

int execl(const char *path,const char *arg0,.../*,(char *)0 */);
int execle(const char *path,const char *arg0,.../*,
         (char *)0 *,char *const envp[]*/);
int execv(const char *path,char *const arg[]);
int execlp(const char *file,const char *arg0,.../*,(char *)0 */);
int execve(const char *path,char *const arg[],
         char *const envp[]);
int execvp(const char *file,char *const arg[]);
```

2. execv( )函数

execv( )、execve( )和 execvp( )函数使用指针向量来传递命令行参数，指针指向以 null 作为结束的字符串。命令行参数的数目应当在编译时就已知，这样这些函数才能够使用。argv[0]通常为执行语句。

- `int execv(const char *path,char *const arg[]);`
  其中 path 参数是可执行程序的路径名，可以被指定为绝对路径名或当前目录的相对路径名。第二个参数是以 null 作为结束的向量，其中包含通过以 null 作为结束的字符串来表示的命令行参数。可以有 *n* 个参数，向量最后跟着一个 NULL 指针。
  参数 arg[ ]可以以如下方式赋值：

```
char *const arg[] = {"traverse",".", " > ","1000",NULL};
```

下面是函数调用的一个示例：

```
execv("traverse",arg);
```

在本例中，调用 traverse 实用工具来列出当前目录下所有大于 1000 字节的文件。

- `int execve(const char *path,char *const arg[],char *const envp[]);`
  这个函数除了增加如前所述的envp[ ]参数之外，其他与execv( )均相同。
- `int execvp(const char *file,char *const arg[]);`
  参数file是可执行程序的文件名。第二个参数是以null作为结束的向量，其中包含通过以null作为结束的字符串来表示的命令行参数。可以有*n*个参数，向量最后跟着一个NULL指针。

下面是使用这些参数的execv( )函数系列的句法实例：

```
char *const arg[] = {"traverse",".", ">","1000",NULL};
char *const envp[] = {"files=50",NULL};

execv("/home/tracey/traverse",arg);
execve("/home/tracey/traverse",arg,envp);
execvp("traverse",arg);
```

这些都显示了execv( )函数如何创建进程来执行traverse程序的句法。

3. 确定exec( )函数的限制

在传递给exec( )函数时，argv[ ]和envp[ ]的大小有一定的限制。函数sysconf( )可用于确定命令行参数的大小加上接受envp[ ]的函数的环境变量的大小的最大值，它们会被传递给exec( )函数。为了返回这个值，应当将name赋值为_SC_ARG_MAX。

调用形式

```
#include <unistd.h>

long sysconf(int name);
```

在使用exec( )以及其他用于创建进程的函数时的另一个限制是每个用户id允许的并发进程的最大数目。为了返回这个值，应当将name赋值为_SC_CHILD_MAX。

5.11 进程环境变量的使用

环境变量是以null作为结束的字符串，用来保存系统相关信息，例如到包含进程所使用的命令、库、函数和过程的目录的路径。还可以用于将任何有用的用户定义信息在父进程和子进程之间进行传递。它们是不需要在程序代码中进行硬编码即可为进程提供特定信息的一种机制。系统环境变量是预先定义的，而且对系统中所有的shell和进程都是同样的。变量是通过启动文件初始化的。下面是一些常见的系统变量。

- $HOME：home目录的绝对路径名
- $PATH：查找命令时的目录列表
- $MAIL：邮箱的绝对路径名

- $USER：用户 id
- $SHELL：login shell 的绝对路径名
- $TERM：终端类型

它们可以被保存在文件中，或者被保存在环境列表中。环境列表包含指向以 null 作为结束的字符串。变量：

```
extern char **environ
```

在进程开始执行时指向环境列表。这些字符串形如：

```
name=value
```

含义如前所述。函数 execl( )、execlp( )、execv( )和 execvp( )初始化的进程继承了父进程的环境。函数 execve( )和 execle( )初始化的进程为新的进程设置环境。

有一些函数和实用工具可用来检查、增加或修改环境变量。getenv( )被用于判断是否已经设置了某个特定变量。参数名就是要考察的环境变量。如果特定变量还没有被设置，则返回 NULL，否则函数返回一个包含返回值的字符串的指针。

调用形式

```
#include <stdlib.h>

char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
```

例如：

```
string Path;

Path = getenv("PATH");
```

字符串 Path 被赋值为预定义环境变量 PATH 中包含的值。

函数 setenv( )用于改变或增加环境变量。参数 name 包含使用 value 中存储的值来增加的环境变量的名称。如果 name 变量已经存在，那么如果 overwrite 参数非零，则将值改为 value。如果 overwrite 为 0，则指定环境变量的内容不会被修改。如果成功，setenv( )返回 0，否则返回-1。函数 unsetenv( )删除由 name 指定的环境变量。

5.12　使用 system( )生成新的进程

system( )是另一个用于执行命令或可执行程序的函数。system( )可导致 fork( )、exec( )和一个 shell 的执行。函数 system( )执行 fork( )，然后子进程调用 exec( )以及执行给定的命令或程序的 shell。

调用形式

```
#include <stdlib.h>

int system(const char *string);
```

参数 string 可以是系统命令或可执行文件的名称。如果成功，函数返回命令的结束状态或程序的返回值(如果存在)。在多个级别上可能会发生错误，fork( )或 exec( )可能会失败，或者 shell 可能不能够执行命令或程序。

该函数向父进程返回一个值。如果 exec( )失败，则返回 127，如果发生一些其他错误，则返回-1。如果函数成功，则会返回命令的返回代码。这个函数不会影响任何子进程的等待状态。

5.13 删除进程

当进程终止之后，PCB 会被清除掉，而且已经终止的进程所使用的地址空间和资源都会被释放。主进程表中的对应表项会被放置上退出码。一旦父进程接受了退出码，则该项会被删除。在以下几种情况下会发生进程的终止：

- 所有指令均被执行。进程进行显式返回或发起一个终止进程的系统调用。当父进程终止时，子进程可能会自动终止。
- 父进程发出一个信号来终止它的子进程。

当进程本身做出一些它不应当做的事情时，进程会异常终止：

- 进程要求的内存多于系统能够为它提供的。
- 进程试图访问一些不允许访问的资源。进程试图执行一条无效指令或被禁止的计算。

当进程是交互式时，用户也可以终止进程。

父进程负责它的子进程的终止/释放。父进程应当处于等待状态，直到它的所有的子进程已经终止。当父进程提取子进程的退出码后，子进程将正常地退出系统。进程在父进程接受信号之前，一直处于僵死状态。如果父进程永远不接受信号，因为它已经终止并退出系统，或者因为它没有等待子进程，那么子进程会一直处于僵死状态，直到 init 进程(最初的系统进程)接受它的退出码。很多的僵死进程都会对系统的性能带来负面影响。

5.13.1 调用 exit( )和 abort( )

进程可以调用两个函数来终止自身，它们是 exit( )和 abort( )。函数 exit( )会导致调用进程正常终止。与进程关联的所有打开文件描述符都会被关闭。函数会刷新所有包含未写入缓冲数据的打开的流，然后关闭这些打开的流。参数 status 是进程的退出状态。它会被返回到正在等待的父进程，然后父进程会重新启动。status 的值可能为 0、EXIT_FAILURE

或 EXIT_SUCCESS。值 0 意味着进程已经成功地终止，正在等待的父进程只访问 status 的低 8 位。如果父进程没有等待进程的终止，则会有 init 进程来接纳僵死进程。函数 abort( )将导致调用进程的异常终止。进程的异常终止对所有打开的流会产生和 fclose( )相同的效果。等待的父进程会接收到一个信号，即子进程已异常终止。进程只有在它遇到无法通过编程来处理的错误时才会异常终止。

调用形式

```
#include <stdlib.h>

void exit(int status);
void abort(void);
```

5.13.2　kill( )函数

函数 kill( )可导致另一个进程的终止。函数 kill( )发送一个信号给通过参数 pid 指定的进程。参数 sig 是将要发送给指定进程的信号。信号列在头文件<signal.h>中。为了删除一个进程，sig 的值应当为 SIGKILL。调用进程必须有着适当的权限来发送信号到目标进程，或者它有着同接收信号的进程的实际或保存的 set-user-ID 相同的实际或有效用户 id。调用进程可能有着只能发送特定信号给进程的权限，而不是所有信号。如果函数成功地发送了信号，则会将 0 返回给调用进程，如果失败，则返回-1。

调用进程可以在这些条件下发送信号给一个或多个进程。

- pid>0：信号会发送给 PID 等于 pid 的进程。
- pid=0：信号发送给进程组 id 与调用进程相同的所有进程。
- pid=-1：信号发送给调用进程有发送该信号的许可的所有进程。
- pid<-1：信号发送给进程组 id 等于 pid 的绝对值的所有进程，以及调用进程有发送该信号的许可的所有进程。

调用形式

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

5.14　进程资源

进程为了能够执行它所要执行的任何任务，可能需要将数据写入文件、将数据发送到打印机、或者在屏幕上显示数据。进程可能需要来自用户的输入，经由键盘或从文件中输入。进程还可以使用其他进程作为资源，例如子程序。子程序、文件、信号量、互斥量、键盘、显示器都是进程可以利用的资源的示例。资源是被进程在任何指定时间作为数据源、

作为处理或计算的工具、作为显示数据或信息的工具来使用的任何事物。

进程若想访问资源，必须首先对操作系统发出请求。如果资源可用，则操作系统允许进程使用该资源。进程使用资源，然后释放它，这样资源对其他进程就可用了。如果资源不可用，则请求将被拒绝，进程必须等待。当资源变为可用时，进程将被唤醒。这是资源分配的基本形式。图 5-10 显示了资源分配图。资源分配图显示出哪些进程持有资源以及哪些进程正在请求资源。在图 5-10 中，进程 B 对资源 2 进行请求，该资源被进程 C 持有。进程 C 发出对资源 3 的请求，该资源被进程 D 持有。

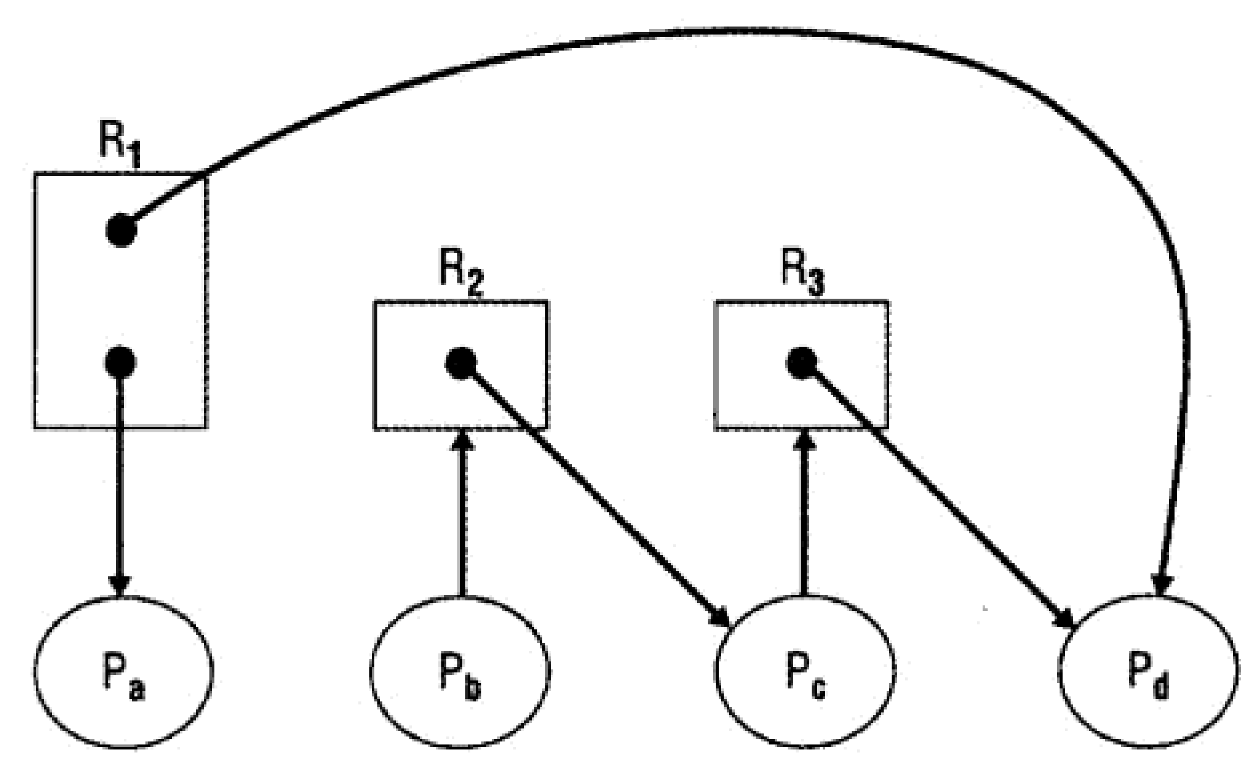


图 5-10

当允许对一个资源的多个请求时，该资源是可共享的，如图 5-10 所示。进程 A 和进程 D 共享资源 1。资源可能允许很多进程并发访问，或者在允许另一个进程访问之前，只允许一个进程访问有限的时间。这种类型的共享资源的一个示例就是处理器。一个进程会被指派到处理器上运行很短的时间间隔，然后另一个进程被指派给处理器。当任意时刻只能够有一个访问资源的请求被允许，而且必须发生在资源被另一个进程释放后，此时我们称这个资源是不可共享的，而且进程对资源进行的是排他访问。在多处理器环境中，了解一个共享资源是可以被同时访问还是每次只能被一个进程访问是非常重要的，能够避免一些并发带来的陷阱。

进程可能会更改一些资源，而有些资源不允许进程更改它们。共享的可更改资源或共享的不可更改资源的性质是由资源的类型决定的。

5.14.1 资源的类型

有 3 种基本的资源类型：

- 硬件
- 数据
- 软件

硬件资源是连接到计算机的物理设备。硬件资源的示例包括处理器、内存和所有的 I/O 设备，包括处理器、硬盘、磁带机、zip 驱动器、显示器、键盘、声卡、网卡、显卡、调制

解调器等。所有这些设备都可以被多个进程共享。

有些硬件资源是抢占式的，允许不同进程访问。例如，处理器是抢占式的，从而可以运行不同的进程。RAM 是另一个共享的可抢占资源的示例。当一个进程不被使用时，它所占有的一些物理页帧可能会被换出到二级存储，使得其他进程可以换入一些页帧来占据那些可用空间。一块内存在任意时刻只能够被一个进程的页帧所占据。非抢占式共享资源的一个示例是打印机。当打印机被共享时，每个进程发送给打印机的作业都会被保存在队列中。每个作业打印完毕之后，另一个作业才能够开始。打印机不会被任何等待中的作业抢占，除非当前作业被取消。

数据资源包括：对象；环境变量、文件和句柄等系统数据；信号量等全局定义变量；互斥量。这些都是被进程共享和更改的资源。正规文件和可以打开的物理设备(如打印机)相关联的文件，限制了能够访问该文件的进程类型。可能会允许只读或只写的访问或读/写访问。对于有着父子关系的进程，子进程继承父进程在创建它时所拥有的资源和对这些资源的访问权利。子进程可以在由父进程打开的文件中推动文件指针，或关闭、修改、重写文件内容。共享内存和允许写入的文件要求同步对它们的访问。类似信号量或互斥量的共享数据可用于对这些共享数据资源的同步。

共享库是软件资源的示例。共享库为进程提供了服务或函数的公共集合。进程还可以共享应用程序、程序和实用工具。在这种情况下，只有一份程序代码的副本将放到内存中，然而有着多份数据副本，每个用户(进程)一份。不会改变的程序代码(也被称作可重入)可以被多个进程同时访问。

5.14.2　设置资源限制的 POSIX 函数

POSIX 定义了限制进程使用特定资源能力的函数。操作系统对进程利用系统资源的能力设置了限制。这些资源限制影响如下方面：

- 进程栈的大小
- 创建的文件大小和内核文件的大小
- CPU 使用数量(时间片的大小)
- 内存使用数量
- 打开的文件描述符的数目

操作系统对进程的资源使用设置了硬限制。进程可以设置或改变其资源的软限制。它的值不应当超过由操作系统设置的硬限制。进程可以降低它的硬限制。这个值应当大于或者等于软限制。当进程降低其硬限制时，这个过程是不可逆的。只有少数有着特定特权的进程可以增加它们的硬限制。

调用形式

```
#include <sys/resource.h>
```

```
int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

函数 setrlimit( )用于对特定资源的消耗设置限制。这个函数可以设置硬限制和软限制。参数 resource 表示资源类型。表 5-7 列出了 resource 的值以及简要的描述。指定资源的软限制和硬限制由 rlp 参数表示。参数 rlp 指向 rlimit 结构体，其中包含两个类型为 rlim_t 的对象：

```
struct rlimit
{
    rlim_t rlim_cur;
    rlim_t rlim_max;
}
```

rlim_t 是无符号整数类型。rlim_cur 包含当前的软限制，rlim_max 包含硬限制的最大值。可以对 rlim_cur 和 rlim_max 赋任意值，还可以将它们赋值为下列在头文件<sys/resource.h>中定义的符号常量。

- RLIM_INFINITY：代表无限
- RLIM_SAVED_MAX：代表无法表示保存的硬限制
- RLIM_SAVED_CUR：代表无法表示已经保存的软限制

软限制或硬限制可以设置为 RLIM_INFINITY，意味着资源是不受限制的。

表 5-7

资 源 定 义	描　　述
RLIMIT_CORE	进程可以创建的内核文件的最大规模，以 byte 为单位
RLIMIT_CPU	进程能够使用的 CPU 时间最大数量，以 s 为单位
RLIMIT_DATA	进程数据段的最大规模，以 byte 为单位
RLIMIT_FSIZE	进程能够创建的文件的最大规模，以 byte 为单位
RLIMIT_NOFILE	比系统能够赋给新创建的文件描述符的最大值大 1 的数字
RLIMIT_STACK	进程的栈的最大规模，以 byte 为单位
RLIMIT_AS	进程总的可用内存的最大规模，以 byte 为单位

函数 getrlimit( )返回在 rlp 对象中指定的资源的软限制和硬限制。函数 getrlimit( )和 setrlimit( )都是在成功时返回 0，失败时返回-1。示例 5-4 包含了进程为文件(以字节为单位)设置软限制的示例。

示例 5-4

```
//Example 5-4 Using setrlimit() to set the soft limit for file size.

#include <sys/resource.h>
```

```
//...
struct rlimit R_limit;
struct rlimit R_limit_values;

//...

R_limit.rlim_cur = 2000;
R_limit.rlim_max = RLIM_SAVED_MAX;
setrlimit(RLIMIT_FSIZE,&R_limit);
getrlimit(RLIMIT_FSIZE,&R_limit_values);
cout << "file size soft limit: " << R_limit_values.rlim_cur << endl;

//...
```

在示例 5-4 中，文件规模的软限制被设置为 2000 字节，硬限制被设置为其最大值。传递给 setrlimit( )的参数为 R_limit 和 RLIMIT_FSIZE。传递给 getrlimit( )函数的参数是 RLIMIT_FSIZE 和 R_LIMIT_values。软限制的值被发送给 cout。

函数 getrusage( )返回关于调用进程使用资源的度量信息，还返回关于调用进程正在等待的终止的子进程的信息。参数 who 可以为以下值：

- RUSAGE_SELF
- RUSAGE_CHILDREN

如果 who 的值为 RUSAGE_SELF，那么返回的信息是和调用进程相关的。如果 who 的值为 RUSAGE_CHILDREN，那么返回的信息是和调用进程的子进程相关的。如果调用进程不等待它的子进程，那么关于子进程的信息会被抛弃。信息在 r_usage 中返回，r_usage 指向 rusage 结构体，该结构体中包含表 5-8 中列出并描述的信息。如果函数成功，则返回 0，如果失败，则返回-1。

表 5-8

struct rusage 属性	描　述
struct timeval ru_utime	使用的用户时间
struct timeval ru_sutime	使用的系统时间
long ru_maxrss	实际驻留在内存中的内存数的最大值
long ru_maxirss	共享内存大小
long ru_maxidrss	非共享数据大小
long ru_maxisrss	非共享栈大小
long ru_minflt	要求的页数
long ru_majflt	故障页的数目
long ru_nswap	页交换的数目
long ru_inblock	阻塞输入操作
long ru_oublock	阻塞输出操作

(续表)

struct rusage 属性	描　　述
long ru_massnd	发送的消息数目
long ru_msgrcv	接收的消息数目
long ru_nsignals	接收的信号数目
long ru_nvcsw	自愿上下文切换的数目
long ru_nivcsw	非自愿上下文切换的数目

5.15 异步进程和同步进程

异步进程的执行相互之间是独立的。进程A运行直到结束，不需要考虑进程B。异步进程可以有父子关系，也可以没有。如果进程A创建进程B，它们都可以独立执行，但是在某个时刻，父进程会获取子进程的退出状态。如果进程之间没有父子关系，它们可能共享同一个父进程。

异步进程可以顺序执行或同时执行，它们的执行也可以重叠。这些情形显示在图5-11中。

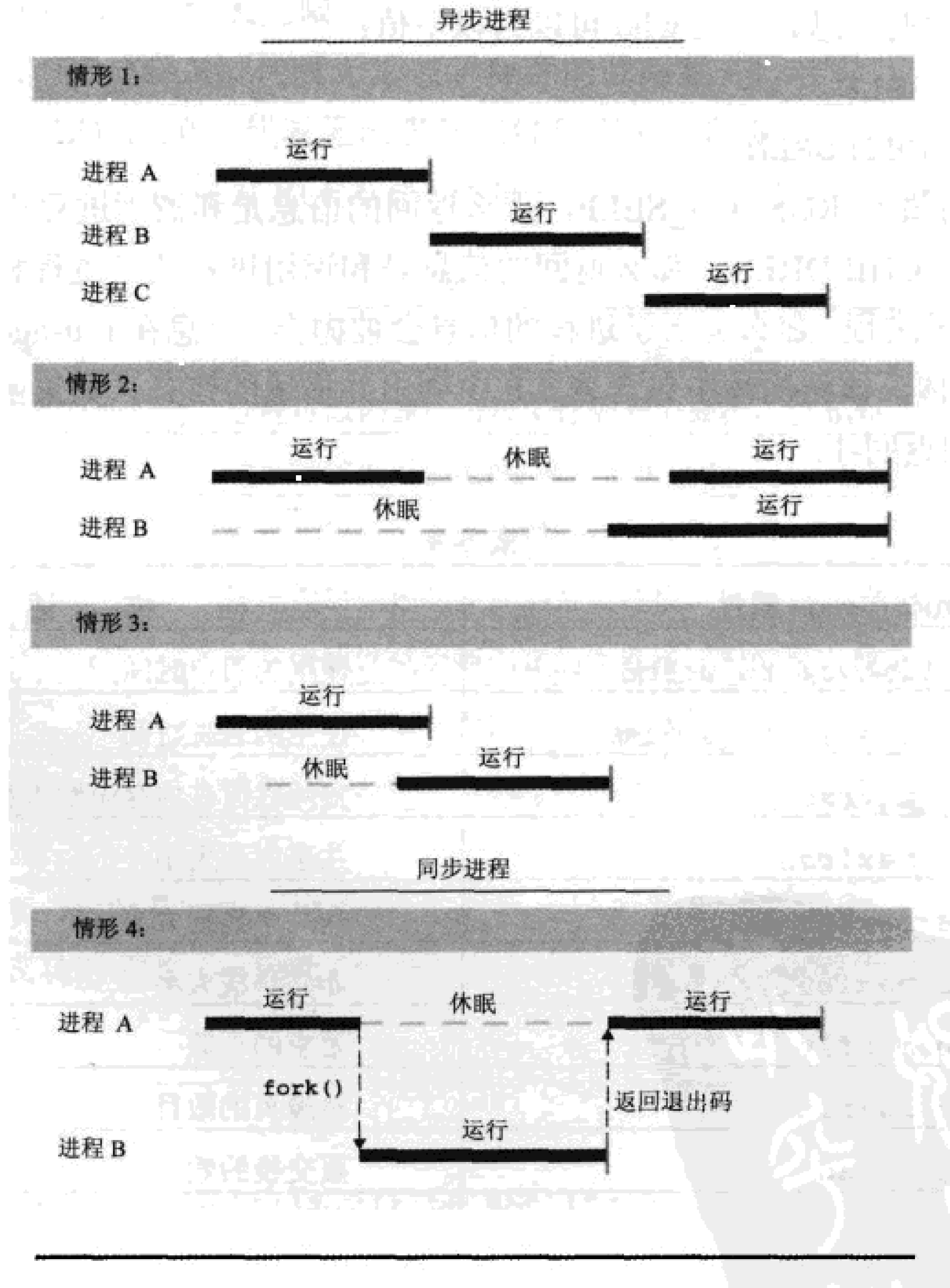


图5-11

在情形 1 中，进程 A 一直运行到结束，然后进程 B 一直运行到结束，接下来进程 C 一直运行到结束。这就是线程的顺序执行。

情形 2 描述了进程的同时执行。进程 A 和 B 是活跃进程。当进程 A 运行时，进程 B 正在休眠。在某一时刻，两个进程都在休眠。进程 B 先于进程 A 被唤醒，然后进程 A 被唤醒，此时两个进程同时运行。这说明异步进程可能只在执行过程中的特定间隔内同时执行。

在情形 3 中，进程 A 的执行和进程 B 的执行发生了重叠。

异步进程可以共享资源，如文件或内存。对资源的使用可能会要求同步或协调，也可能不会有要求。如果进程是顺序执行的(情形 1)，那么它们不需要任何同步。例如，A、B、C 三个进程可能共享一个全局变量，进程 A 在它终止之前对该变量进行写操作，接下来，当进程 B 运行时，它读取变量中读取的数据，在终止之前，它也对变量进行写操作。当进程 C 运行时，它从变量中读取数据。但是在情形 2 和情形 3 中，进程可能会同时试图更改变量，这样就要求对变量的使用进行同步。

在这里，我们将同步进程定义为交错执行的进程，一个进程挂起自身的执行，直到另一个进程结束。例如，父进程 A 执行并创建子进程 B。进程 A 挂起自身的执行，直到进程 B 运行结束。当进程 B 结束后，它的退出码会被放置到进程表中。进程 A 会被通知进程 B 已经结束。进程 A 可以继续其他的处理然后结束，或者立即结束。进程 A 和进程 B 是同步进程。图 5-11 比较了进程 A 和 B 的同步执行和异步执行。

函数 fork( )、posix_spawn( )、system( )和 exec( )中同步进程和异步进程的比较

由 fork( )、fork-exec( )和 posix_spawn( )等函数创建的进程是异步进程。当使用 fork( )时，父进程的映像被复制。一旦子进程被创建，函数将子进程的 PID 和返回值 0 返回给父进程，表示进程创建成功。父进程不会挂起其执行，两个进程继续从紧跟在 fork( )之后的语句相互独立地执行。

使用 fork-exec( )联合创建的子进程使用新的进程映像来初始化子进程的进程映像。函数 exec( )不返回到父进程，除非初始化未成功。

函数 posix_spawn( )在一个函数调用内完成子进程映像的创建和初始化工作。PID 和返回值共同返回给 posix_spawn( )，返回值用来表示进程是否成功生成。在 posix_spawn( )返回之后，两个进程同时执行。

通过 system( )函数创建的进程是同步进程。会创建一个 shell 来执行系统命令或可执行文件。父进程被挂起，直到子进程结束且 system( )调用返回。

5.16　wait( )函数调用

异步进程可以通过执行 wait( )系统调用来挂起其执行，直到子进程结束。当子进程结束后，等待中的父进程收集子进程的退出状态，这样就防止了出现僵死进程。函数 wait( )

从进程表中得到退出状态。参数 status 指向包含子进程退出状态的位置。如果父进程有着多个子进程，而且其中多个已经终止，那么 wait( )函数仅从进程表中获取一个子进程的退出状态。如果状态信息在 wait( )函数的执行之前就可用，那么函数会立即返回。如果父进程没有任何子进程，那么函数将返回错误代码。调用进程等待信号到来，然后执行一些信号处理动作时，也可以调用 wait( )函数。

调用形式

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

函数 waitpid( )与 wait( )基本相同，只是它多了一个参数 pid。参数 pid 指定了一个子进程的集合，集合中的进程的退出状态会被获取。集合中包含哪些进程是由 pid 的值决定的。

- pid>0：单独的子进程
- pid=0：组 id 与调用进程相同的所有子进程
- pid<0：组 id 与 pid 的绝对值相等的所有子进程
- pid=-1：所有子进程

参数 options 决定了等待将表现出怎样的行为，可以为如下值，这些值是在头文件<sys/wait.h>中定义的常量。

- WCONTINUED：报告所有继续的子进程(由 pid 指定)的退出状态，这些进程的状态从继续之后还没有被报告过。
- WUNTRACED：报告所有停止的子进程(由 pid 指定)的退出状态，这些进程的状态从停止之后还没有被报告过。
- WNOHANG：如果指定子进程的退出状态还不可用，则调用进程不会被挂起。

这些常量可以进行逻辑或，然后作为 options 参数进行传递(例如：WCONTINUED | WUNTRACED)。

函数 wait( )和 waitpid( )都会返回子进程的 PID，这些子进程的退出状态都已经被获取。如果状态中保存的值为 0，那么子进程已经在如下情形下终止：

- 进程从函数 main( )返回 0。
- 进程使用 0 作为参数调用某个版本的 exit( )。
- 进程由于最后一个线程终止而终止。

表 5-9 列出了可以用于计算退出状态值的宏。

表 5-9

计算 status 的宏	描　述
WIFEXITED	如果是正常终止的子进程返回的状态，则计算结果为非零值
WEXITSTATUS	如果 WIFEXITED 为非零值，则这个宏为子进程传递给_exit( )和 exit( )的 status 的低 8 位，或者是从 main( )返回的值

（续表）

计算 status 的宏	描　述
WIFSIGNALED	如果 status 是从由于未捕捉到发送给它的信号而终止的子进程返回的，则 WIFSIGNALED 的值为非零值
WTERMSIG	如果 WIFSIGALED 为非零值，则 WTERMSIG 为导致子进程终止的信号的编号
WIFSTOPPED	如果 status 是从当前已经停止的子进程返回的，则 WIFSTOPPED 为非零值
WSTOPSIG	如果 WIFSTOPPED 为非零值，则 WSTOPSIG 为导致子进程停止的信号的编号
WIFCONTINUED	如果 status 是从作业控制停止继续执行的子进程返回的，则 WIFCONTINUED 为非零值

5.17　谓词、进程和接口类

在 C++中，谓词是一个返回布尔值的函数对象[Stroustrup，1997]。在这个定义中，经常被忽略的词是“对象”。谓词不仅仅是函数，它有着对象的语义。谓词为动作序列提供了声明式解释。谓词是可以计算为真或假的语句。为了得到更具声明性的并行化方法，您将会发现有些时候将进程或线程封装为 C++谓词会很方便，使得您可以将进程或线程当作与容器和算法一起使用的对象来对待。在 C++中，这种对谓词的巧妙使用会使用户大幅度脱离并行化的过程式方法，得到更加声明式的方法。

以本章之前部分中，程序清单 5-3 中的 guess_it 程序为例。尽管我们使用了一个接口类来为 posix_spawn 函数提供更具声明性的接口，但我们还可以做得更好。换句话说，程序清单 5-3 是第 4 章中 guess_it 程序的更具声明性的版本，而现在程序清单 5-5 是程序清单 5-3 中的程序的更具声明性的版本。

程序清单 5-5

```
// Listing 5-5  A restatement of the guess_it program from Chapter 4.

 1  #include "posix_process.h"
 2  #include "posix_queue.h"
 3  #include "valid_code.h"
 4
 5
 6  char **av;
 7  char **env;
 8
 9
10  int main(int argc,char *argv[],char *envp[])
11  {
12
13    valid_code  ValidCode;
```

```
14      ValidCode.determination("ofind_code");
15      cout <<  (ValidCode() ? "you win" : "you lose");
16      return(0);
17  }
```

这里，我们决定将游戏的代码建模为名为valid_code的C++类，将对posix_process.run( )方法的调用封装到C++谓词中。程序清单5-5中第15行包含ValidCode( )谓词。如果谓词ValidCode( )为真，则用户已经在5分钟的时间约束内猜出了正确的6字符编码。程序清单5-5中的程序新产生两个进程使用posix_process类，新产生4个线程使用来自第4章的user_thread类。posix_process类和user_thread类都是接口类，用来与posix_spawn( )和pthread_create( )的接口适配。

程序概要5-1

程序名称：

oguess_it.cc(程序清单5-5)

描述：

程序oguess_it是程序清单5-3中的程序更具声明性的版本。对posix_process.run( )方法的调用被封装在C++谓词中。如果谓词ValidCode( )为真，那么用户已经在5分钟的时间约束内正确地猜到了该6字符编码。程序新产生两个进程使用posix_process类，新产生4个线程使用user_thread类。

必需的库：

pthread, rt

需要的其他源文件：

oguess_it.cc(程序清单5-5),posix_process.cc(程序清单5-4)

必需的用户定义头文件：

posix_process.h(程序清单5-2)

编译和链接指令：

```
c++ -o oguess_it oguess_it.cc posix_process.cc -lrt
```

测试环境：

Linux Kernel 2.6
Solaris 10、gcc 3.4.3和3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

我们通过将 Process.run( )调用封装为 C++谓词来进一步加强声明性解释。程序清单 5-6 是 valid_code 谓词类的声明。

程序清单 5-6

```
//Listing 5-6  Declaration of our valid_code predicate class.

 1  #ifndef __VALID_CODE_H
 2  #define __VALID_CODE_H
 3  using namespace std;
 4
 5  #include <string>
 6  class valid_code{
 7  private:
 8     string Code;
 9     float  TimeFrame;
10     string Determination;
11     bool InTime;
12  public:
13     bool operator()(void);
14     void determination(string X);
15  };
16
17  #endif
```

我们将它称为谓词类是因为我们已经在第 13 行重载了 operator( )。注意 opterator( )返回类型为 bool，这就是谓词与函数对象的区别。由于 valid_code 是一个类，它有着我们需要作为基础的声明性语义。由于 valid_code 是一个类，它可以和容器类以及算法一同使用。这种将进程和线程通过谓词概念来处理的用法打开了思考并行编程的新方法。程序清单 5-7 包含了 valid_code 类的定义。

程序清单 5-7

```
//Listing 5-7  Definition of our valid_code predicate class.

 1  #include "valid_code.h"
 2  #include "posix_process.h"
 3  #include "posix_queue.h"
 4
 5  extern char **av;
```

```
 6 extern char **env;
 7
 8
 9 bool valid_code::operator()(void)
10 {
11    int Status;
12    int N;
13    string Result;
14    posix_process Child[2];
15    for(N = 0; N < 2; N++)
16    {
17       Child[N].binary(Determination);
18       Child[N].arguments(av);
19       Child[N].environment(env);
20       Child[N].run();
21       Child[N].pwait(Status);
22    }
23    posix_queue PosixQueue("queue_name");
24    PosixQueue.receive(Result);
25    if((Result == Code) && InTime){
26       return(true);
27    }
28    return(false);
29 }
30
31
32 void valid_code::determination(string X)
33 {
34    Determination = X;
35 }
36
```

由 operator( )定义的谓词的定义开始于程序清单 5-7 的第 9 行。注意在第 25 行中，如果 Result 是正确的，而且 InTime 为真，则这个谓词返回 true，否则返回 false。第 16 行～第 21 行将导致产生两个进程。程序清单 5-5 中的第 14 行决定将二进制文件 ofind_code 同新产生的两个进程关联起来。在本例中，每个 ofind_code 的实例创建两个线程来进行搜索，使得我们总共有 4 个线程。但是所有这些关于进程和线程的实现对于程序清单 5-5 中的程序是完全透明的。程序清单 5-5 中的程序只关心 ValidCode 以及 ValidCode( )谓词为 true 还是 false。

您可以在不需要改动并行化的声明式解释的前提下，扩展进程和线程的数目。我们在这里强调声明式语义，是因为当您扩展到 CMP 上更多的内核时，过程式的思考会面临巨大的挑战。可以使用声明式模型来帮助处理并行编程的复杂性。将 C++接口类和 C++谓词联合使用，以封装进程、线程和 POSIX API，是一个正确的方向。

5.18　小结

本章主要介绍了进程以及如何借助它们来帮助进行多核编程。涵盖的关键点包括：

- C++程序中的并发是通过将程序分解为多个进程或多个线程来实现的。当隔离性、安全性、地址空间、并发执行任务能够拥有的资源最大数目是主要的考虑因素时，使用进程要比使用线程更好些。当决定使用进程而不是线程时，进程间通信以及启动时间是两个主要的考虑因素。
- 进程的主要特性和属性被保存在进程控制块(PCB)中，操作系统使用 PCB 来识别进程。操作系统需要这些信息来管理每个进程。PCB 是使得进程使用起来比线程更重量级或更昂贵的结构之一。
- 进程与它创建的其他进程之间具有父子关系。进程创建者是父进程，被创建的进程是子进程。子进程从父进程继承很多属性。父进程的关键职责是等待子进程，这样父进程才可以退出系统。
- 有多个系统调用可用来创建进程：fork( )、fork-exec( )、system( )和 posix_spawn( )。函数 fork( )、fork-exec( )和 posix_spawn( )创建和父进程异步的进程，而 system( )创建的子进程与父进程同步。一些平台可能会在实现 fork( )时存在问题，可以使用 posix_spawn( )作为替代。
- 使用接口类，例如本章中的 posix_process 类，来为用于进程管理的 POSIX API 函数构建声明式接口。如果您为进程和线程构建接口类，那么您就已经开始从面向对象的视角来实现并发任务了，而不是从过程式视角。
- 在使用转向并行化的更加声明式的方法时，您还会看到某些时候将进程或线程封装为 C++谓词会很有用。通过谓词来封装进程或线程使得您可以将它们视作对象，并与容器和算法一起使用。这种在 C++中对谓词的精巧使用，可以在从并行化的过程式方法转向声明式方法方面迈出一大步。

下一章将讨论多线程。线程是进程中可执行代码的序列，被操作系统调度到处理器或内核上运行。在多核处理器上采用线程的并行编程比起多进程来有很多好处。下一章将讨论这些优点以及一些缺点，并讨论创建和管理线程的 POSIX API 函数，以及如何将这种功能封装到在第 4 章中引入的线程类中。

多 线 程

在第 5 章中，我们查看了如何通过将程序分解为多个进程或多个线程而在 C++程序中实现并发。我们讨论了进程，它是由操作系统创建的工作单元，解释了用于进程管理的 POSIX API 以及多个可用于创建进程的系统调用：fork( )、fork-exec( )、system( )和 posix_spawn( )。还示范了如何构建 C++接口组件、接口类和可用于简化一部分用于进程管理的 POSIX API 的声明式接口。本章将介绍：

- 什么是线程
- 用于线程管理的 pthread API
- 线程调度及优先级
- 线程竞争范围
- 扩展 thread_object 以封装线程属性功能

6.1 什么是线程

线程是进程中可执行代码流的序列，它被操作系统调度，并在处理器或内核上运行。所有的进程都有一个主线程(primary thread)。主线程是进程的控制流或执行线路。具有多个线程的进程拥有和线程数目一样多的控制流。每个线程独立且并发地执行自身的指令序列。具有多个线程的进程是多线程的。线程分为用户级线程和内核级线程。与进程相比，内核级线程在创建、维护和管理方面给操作系统带来的负担都要轻很多，因为与线程关联的信息很少。内核线程被称作轻量级进程，因为它的开销要比进程少。

线程执行程序中无关的并发任务。线程可用于简化具有固有并发的应用程序的程序结构，其方式与通过函数或过程来封装功能性使得应用程序的结构更简单相同。线程可以封装并发功能。线程在一个进程的地址空间中使用最少的共享资源，相比之下，应用程序则使用多个进程。这使得操作系统得到总体更加简单的程序结构。如果正确使用，线程可以通过利用多核处理器并发来改进应用程序的吞吐率和性能。每个线程负责被分配的一个子任务，然后线程独立管理子任务的执行。可以为每个线程指定反映它执行的子任务的重要性的优先级。

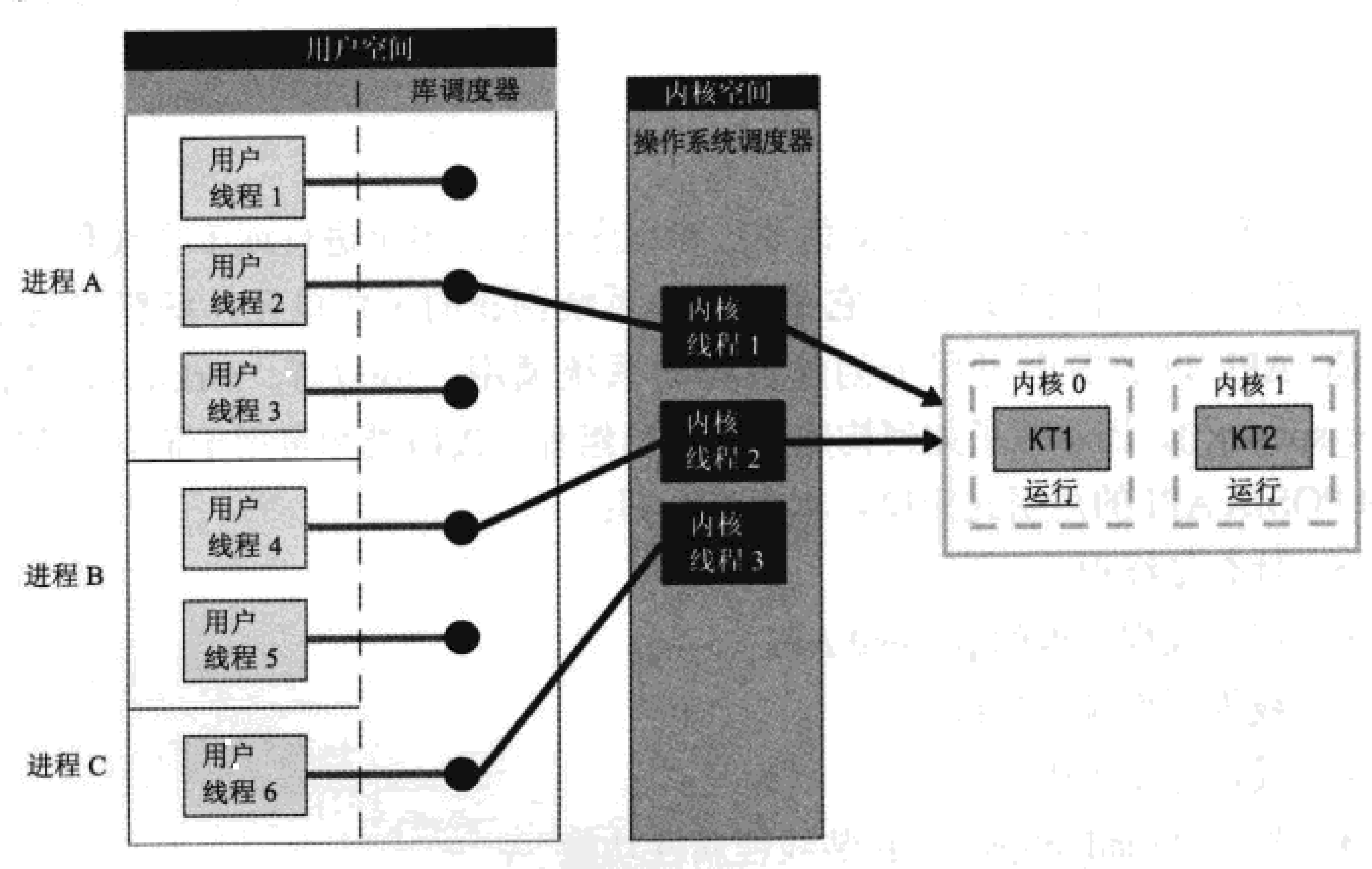
PDG

6.1.1 用户级线程和内核级线程

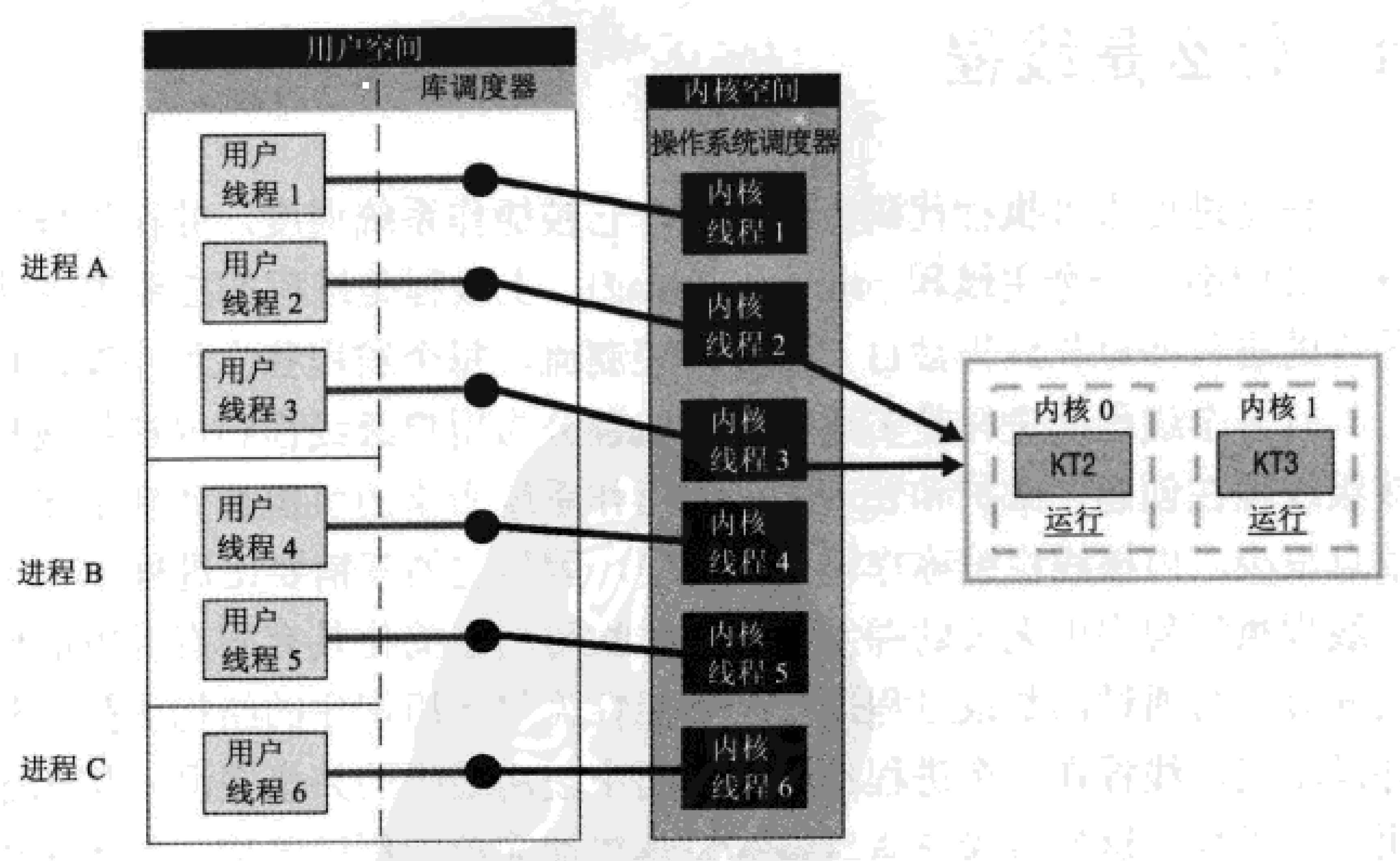
线程有 3 种实现模型：

- 用户级或应用程序级线程
- 内核级线程
- 用户级和内核级混合线程

图 6-1 显示了 3 种线程实现模型。图 6-1(a)显示了用户级线程，图 6-1(b)显示了内核级线程，图 6-1(c)则显示了用户线程和内核线程的混合。

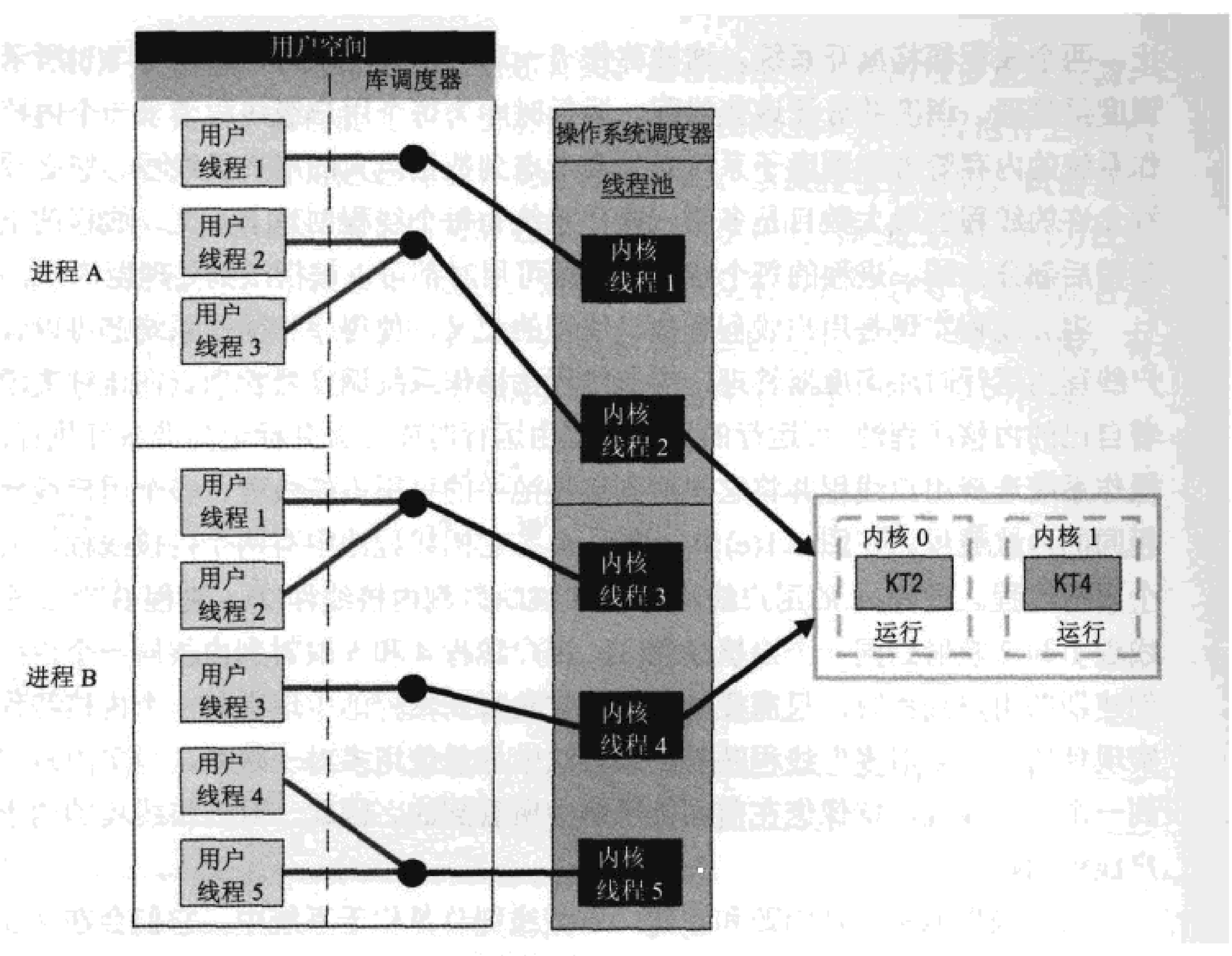


(a) 用户级线程



(b) 内核级线程

图 6-1



(c) 混合线程

图 6-1(续)

这些实现之间的较大的区别之一就是它们的模式以及要指派给处理器的线程的能力。这些线程运行在用户模式下或内核模式下。

- 在用户模式下，进程或线程是执行程序或链接库中的指令，它们不对操作系统内核进行任何调用。
- 在内核模式下，进程或线程是在进行系统调用，例如访问资源或抛出异常。同时，在内核模式下，进程或线程可以访问在内核空间中定义的对象。

用户级线程驻留在用户空间或模式。运行时库管理这些线程，它也位于用户空间。它们对于操作系统是不可见的，因此无法被调度到处理器内核。每个线程并不具有自身的线程上下文。因此，就线程的同时执行而言，任意给定时刻每个进程只能够有一个线程在运行，而且只有一个处理器内核会被分配给该进程。对于一个进程，可能有成千上万个用户级线程，但是它们对系统资源没有影响。运行时库调度并分派这些线程。如同在图 6-1(a)中看到的那样，库调度器从进程的多个线程中选择一个线程，然后该线程和该进程允许的一个内核线程关联起来。内核线程将被操作系统调度器指派到处理器内核。用户级线程是一种“多对一”的线程映射。

内核级线程驻留在内核空间，它们是内核对象。有了内核线程，每个用户线程被映射或绑定到一个内核线程。用户线程在其生命期内都会绑定到该内核线程。一旦用户线程终

止，两个线程都将离开系统。这被称作“一对一”线程映射，如图 6-1(b)所示。操作系统调度器管理、调度并分派这些线程。运行时库为每个用户级线程请求一个内核级线程。操作系统的内存管理和调度子系统必须要考虑到数量巨大的用户级线程。您必须了解每个进程允许的线程的最大数目是多少。操作系统为每个线程创建上下文。线程的上下文将在本章稍后部分介绍。进程的每个线程在资源可用时都可以被指派到处理器内核。

混合线程实现是用户线程和内核线程的交叉，使得库和操作系统都可以管理线程。用户线程由运行时库调度器管理，内核线程由操作系统调度器管理。在这种实现中，进程有着自己的内核线程池。可运行的用户线程由运行时库分派并标记为准备好执行的可用线程。操作系统选择用户线程并将它映射到线程池中的可用内核线程。多个用户线程可以分配给相同的内核线程。在图 6-1(c)中，进程 A 在它的线程池中有两个内核线程，而进程 B 有 3 个内核线程。进程 A 的用户线程 2 和 3 被映射到内核线程(2)。进程 B 有 5 个线程，用户线程 1 和 2 映射到同一个内核线程(3)，用户线程 4 和 5 映射到内核同一个内核线程(5)。当创建新的用户线程时，只需要简单地将它映射到线程池中现有的一个内核线程即可。这种实现使用了“多对多”线程映射。该方法中尽量使用多对一映射。很多用户线程将会映射到一个内核线程，就像您在前面的示例中所看到的。因此，对内核线程的请求将会少于用户线程的数目。

内核线程池不会被销毁和重建，这些线程总是位于系统中。它们会在必要时分配给不同的用户级线程，而不是当创建新的用户级线程时就创建一个新的内核线程，而纯内核级线程被创建时，就会创建一个新的内核线程。只对池中的每个线程创建上下文。有了内核线程和混合线程，操作系统分配一组处理器内核，进程的线程可以在这些处理器内核之上运行。线程只能在为它们所属线程指派的处理器内核上运行。

在确定线程的调度模型和竞争范围时，用户级线程和内核级线程变得很重要。竞争范围决定了指定的线程与那些线程竞争处理器的使用，而且对于操作系统对大量线程的内存管理也非常重要。

6.1.2 线程上下文

操作系统管理很多进程的执行。有些进程是来自各种程序、系统和应用程序的单独进程，而某些进程来自被分解为很多进程的应用或程序。当一个进程从内核中移出，另一个进程成为活动的，这些进程之间便发生了上下文切换。操作系统必须记录重启进程和启动新进程使之活动所需要的所有信息。这些信息被称作上下文，它描述了进程的现有状态。当进程成为活动的，它可以继续从被抢占的位置开始执行。进程的上下文信息包括：

- 进程 id
- 指向可执行文件的指针
- 栈
- 静态和动态分配的变量的内存
- 处理器寄存器

进程上下文的多数信息都与地址空间的描述有关。进程的上下文使用很多系统资源，而且会花费一些时间来从一个进程的上下文切换到另一个进程的上下文。线程也有上下文。表 6-1 将线程上下文和第 5 章讨论的进程上下文进行了对比。当线程被抢占时，就会发生线程之间的上下文切换。如果线程属于相同的进程，它们共享相同的地址空间，因为线程包含在它们所属于的进程的地址空间内。这样，进程需要恢复的多数信息对于线程而言是不需要的。尽管进程和它的线程共享了很多内容，但最为重要的是其地址空间和资源，有些信息对于线程而言是本地且唯一的，而线程的其他方面包含在进程的各个段的内部。

表 6-1

上下文内容	进　程	线　程
指向可执行文件的指针	x	
栈	x	x
内存(数据段和堆)	x	
状态	x	x
优先级	x	x
程序 I/O 的状态	x	
授予权限	x	
调度信息	x	
审计信息	x	
有关资源的信息 ● 文件描述符 ● 读/写指针	x	
有关事件和信号的信息	x	
寄存器组 ● 栈指针 ● 指令计数器 ● 诸如此类	x	x

对线程唯一或本地的信息包括线程 id、处理器寄存器(当线程执行时寄存器的状态，包括程序计数器和栈指针)、线程状态及优先级、线程特定数据(thread-specific data，TSD)。线程 id 是在创建线程时指定的。线程能够访问它所属进程的数据段，因此线程可以读写它所属进程的全局声明数据。进程中一个线程做出的任何改动都可以被进程中的所有线程以及主线程获得。在多数情况下，这要求某种类型的同步以防止无意的更新。线程的局部声明变量不应当被任何对等线程访问。它们被放置到线程栈中，而且当线程完成时，它们便会被从栈中移走。

注意：
线程间的同步将在第7章中讨论。

TSD是一种结构体，包含线程私有的数据和信息。TSD可以包含进程全局数据的私有副本，还可以包含线程的信号掩码。信号掩码用来识别特定类型的信号，这些信号在发送给进程时不会被该线程接收。否则，如果操作系统给进程发送一个信号，进程的地址空间中的所有线程也会接收到那个信号。线程会接收所有没有被掩码遮蔽的信号。

线程与它所属的进程共享代码段和栈段。它的指令指针指向进程的代码段的某个位置，是下一条可执行的线程指令，而且栈指针指向进程栈中线程的栈的顶部位置。线程还可以访问任何环境变量。进程的所有资源(例如文件描述符)都将与线程共享。

6.1.3 硬件线程和软件线程

线程可以在硬件中实现，也可以在软件中实现。芯片生产商实现了有着多个硬件线程的内核，用作逻辑内核。有着多个硬件线程的内核被称作同时多线程(simultaneous multithreaded，SMT)内核。SMT将多线程的概念引入到硬件中，方式类似于软件线程。支持SMT的处理器在处理器内核中同时执行很多软件线程或进程。让软件线程同时在单独的处理器内核中执行，增加了内核的效率，因为类似I/O延迟等因素产生的等待时间被减到最短。操作系统将逻辑内核按照独特的处理器内核来对待。它们会要求一些冗余的硬件来存储线程的上下文信息，例如指令计数器和寄存器组。根据处理器内核的不同，其他硬件或结构将被复制或在多个线程的上下文之间共享。

Sun公司的UltraSparc T1、IBM公司的Cell Broadband Engine(CBE)以及各种Intel多核处理器利用SMT或芯片级多线程(chip-level multithreading，CMT)，实现了每个内核上2个～8个线程。超线程是Intel对SMT的实现，其主要目的就是改善对多线程代码的支持。超线程或SMT技术通过在一个处理器内核上并行执行线程，在特定负载下提供了对CPU资源的有效使用。

6.1.4 线程资源

线程将大部分的资源同相同进程中其他线程进行共享。线程拥有一些定义它们上下文的资源。线程必须共享其他资源，例如处理器、内存、文件描述符等。文件描述符是单独为每个进程分配的，相同进程中的线程将竞争对这些描述符的使用权。线程可以分配额外的资源，例如文件或互斥量，但是进程中所有的线程都可以访问它们。

一个进程能够消耗的资源是受限制的。因此，对等线程拥有的全部资源不能够超过进程的资源限制。如果一个线程试图消耗的资源数量多于定义的软件资源限制，它会收到一个信号，被告知达到了进程的资源限制。

当线程利用它们的资源时必须很小心，不能够在它们被取消时将这些资源置于不稳定的状态。如果终止的线程放任文件处于打开状态，可能导致文件受损，或者当应用程序终止时导致数据丢失。在终止之前，线程应当执行一些清理工作，防止这些不期望出现的状况发生。

6.2 线程和进程的比较

线程和进程都能够提供并发程序执行。当您决定是使用多个进程还是使用多个线程时，上下文切换需要使用的系统资源、吞吐量、实体间通信、程序简化等都是需要考虑的问题。

6.2.1 上下文切换

当您创建一个进程时，可能只需要主线程这一个线程就可以实现进程的功能了。当进程有着多个并发子任务时，多个线程能够在上下文切换的开销较少的情况下提供子任务的异步执行。如果处理器可用性较低或者只有一个内核，并发执行的进程由于需要进行上下文切换而带来较大的开销。相同的情况下，如果使用线程，只有当下一个要指派到处理器的线程来自另一个进程时，才会发生进程上下文切换。较少的开销意味使用的系统资源较少，而且上下文切换的时间也更短。当然，如果有足够的处理器用于周转，那么上下文切换就不再是一个问题。

6.2.2 吞吐量

使用多个线程可以增加应用程序的吞吐量。当只有一个线程时，I/O 请求将会让整个进程暂停。有了多个线程之后，当一个线程等待 I/O 请求时，应用程序将继续执行。当一个线程被阻塞时，另一个线程便可以执行。整个应用程序不需要等待每个 I/O 请求被满足，其他不依赖于被阻塞线程的任务可以继续执行。

6.2.3 实体间的通信

线程与被称为对等线程的进程中其他线程之间，不要求特殊的通信机制。线程可以直接与其他对等线程进行数据的传递和接收。这节省了使用多个进程时，为了建立和维护特殊的通信机制所使用的系统资源。线程是通过使用进程地址空间中的共享内存来通信的。例如，如果进程声明了一个全局队列，进程中的线程 A 可以保存对等线程 B 将要处理的文件名。线程 B 可以从队列中读取该文件名并处理数据。

进程也可以通过共享内存进行通信，但是进程有着独立的地址空间，因此共享内存存在于进行通信的两个进程的地址空间外部。如果有一个进程希望将它处理的文件名传递给其他进程，可以使用消息队列。需要在涉及的进程的地址空间外部建立这个消息队列，而

且通常需要大量的设置才能够很好地工作。这增加了用于维护和访问共享内存所使用的时间和空间。

6.2.4 破坏进程数据

线程可以很轻易地破坏进程的数据。如果没有同步，线程对相同数据片的写入访问可以导致数据竞争，对于进程则不会这样。每个进程有它自己的数据，而且除非设置特殊通信，否则其他进程不能够访问到它们。进程的隔离的地址空间保护数据不受其他进程无意的破坏。线程共享相同地址空间的事实使得若不使用同步，则数据便会面临被破坏的风险。例如，假定一个进程中包含 3 个线程：Thread A、Thread B 和 Thread C。Thread A 和 Thread B 更新某个计数器，Thread C 读取每个被更新的值，并将这个值用于计算中。Thread A 和 Thread B 都试图并发地写入到内存的某个位置中，在 Thread C 读取之前，Thread B 重写了 Thread A 写入的数据。应当使用同步来确保在 Thread C 读取数据之前，计数器不会被更新。

注意：
线程间和进程间的同步问题将会在第 7 章中讨论。

6.2.5 删除整个进程

如果线程产生严重的访问违规，则可能导致整个进程的终止。访问违规不局限于线程，因为它发生在进程的地址空间。线程导致的错误往往比进程导致的错误代价更大。线程可以产生影响所有线程的整个内存空间的数据错误。线程不是隔离的，而进程是隔离的。进程可以发生导致进程终止的访问违规，但是如果违规后果不是很严重，则所有其他进程继续执行，数据错误可以限制在一个进程内。进程可以保护资源不被其他进程任意地访问。线程与进程中所有其他线程共享资源。损害资源的线程会影响整个进程或程序。

6.2.6 被其他程序重用

线程依赖于进程，不能够从它们所属的进程中分离开。进程比线程要独立得多。应用程序可以在多个进程之间分配任务，这些进程可以被封装为可以在其他应用程序中使用的模块。线程不能够在创建它们的进程外部生存，因此是不可重用的。

6.2.7 线程与进程的关键类似和差别

线程和进程有很多相似之处，也存在巨大的差别。线程和进程都有 id、寄存器组、状态和优先级，而且都支持某种调度策略。与进程类似，线程也有一个环境来向操作系统描述实体，即进程上下文或线程上下文。上下文用来重新构建被抢占的进程或线程。尽管进程所需要的信息远多于线程所需要的信息，但它们的目的是相同的。

线程和子进程不需要额外的初始化或准备就能够共享父进程的资源。进程打开的资源对线程或子进程是立即可访问的。与内核实体类似，线程和子进程会竞争对处理器的使用。父进程对子进程或线程有一定的控制。父进程可以对子进程或线程进行如下操作：

- 取消
- 挂起
- 重新开始
- 改变优先级

线程或进程可以改变自身的属性和创建新的资源，但是它不能够访问属于其他进程的资源。

如同我们已经指出的那样，线程和进程之间最大的差别在于每个进程有自己的进程空间，而线程则包含在所属进程的地址空间内。这就是为何线程可以非常容易地共享资源，而且线程间通信如此简单的原因。子进程有自己的地址空间以及它的父进程的数据段的副本，所以当子进程改动它的数据时，不会影响父进程的数据。如果父进程同子进程期望共享数据，则需要创建一块共享内存区域。共享内存是进程间通信机制的一种类型，其中包含了管道以及先进先出(FIFO)调度策略。进程间通信机制用于在进程之间传递数据。

注意：

进程间通信将在第 7 章讨论。

尽管进程可以对它的子进程进行控制，对等线程却是处于相同的级别，无论是谁创建了它们。任意线程只要有权使用另一个对等线程的线程 id，就能够对该线程进行取消、挂起、重新开始或改变优先级的操作。实际上，进程中的任意线程都可以通过取消主线程来删除进程，从而终止进程中的所有线程。对主线程的任何更改可能会影响进程中的所有线程。如果主线程的优先级发生了变化，则进程中继承该优先级的所有线程也都会发生变化。

表 6-2 汇总了线程与进程的关键类似和差别。

表 6-2

线程与进程的类似	线程与进程的差别
都有 id、寄存器组、状态、优先级和调度策略	线程共享创建它的进程的地址空间，进程有自己的地址空间
都有用于为操作系统描述实体的属性	线程能够对所属进程的数据段进行直接访问；进程有着父进程的数据段的自己的副本
都包含一个信息块	线程可以同所属进程的其他线程直接通信；进程必须使用进程间通信才能够和兄弟进程进行通信
都与父进程共享资源	线程几乎没有开销，进程则有相当大的开销
都可作为与父进程独立的实体	创建新的线程很容易，创建新的进程则需要复制父进程
创建者可以对线程或进程进行一些控制	线程可以对相同进程的其他线程进行相当大的控制，进程只能够对子进程进行控制

(续表)

线程与进程的类似	线程与进程的差别
都可以改变它们的属性	对主线程的改动(取消、优先级改动等)可能会影响到进程中其他线程的行为；对父进程的改动不会影响到子进程
都可以创建新的资源	
都不能够访问另一个进程的资源	

6.3 设置线程属性

存在一些可用来确定线程上下文的关于线程的信息。这些信息用于重建线程的环境。令对等线程相互之间产生区别的是 id、定义线程状态的寄存器组、优先级和它的栈。这些属性使得线程有了自己的身份。

POSIX 线程库定义了线程属性对象(attribute object)，它封装了线程属性的一个子集。这些属性可以被线程的创建者访问和更改。下面是可以被更改的线程属性：

- 竞争范围
- 栈大小
- 栈地址
- 分离的状态
- 优先级
- 调度策略和参数

线程属性对象可以同一个或多个线程关联。属性对象是定义了一个或一组线程的行为的概要(profile)。一旦对象被创建并初始化，可以在对线程创建函数的调用中重复引用它。如果重复使用，便会创建一组有着相同属性的线程。所有使用该属性对象的线程继承所有的属性值。一旦使用线程属性对象创建了线程，多数属性就不能够在线程使用中被改动。

范围属性描述了哪些线程同特定线程竞争资源。线程在两个竞争范围内争夺资源：

- 进程范围
- 系统范围

线程依照竞争范围和分配域(它被指派到的处理器集)来同其他线程竞争处理器的使用。有着进程竞争范围的线程同进程中其他线程竞争，而有着系统竞争范围的线程同系统分配的其他进程的线程竞争资源。有着系统范围的线程和系统中所有线程一起被排序和调度。

线程的栈大小和位置是在线程被创建时设置的。如果线程栈的大小和位置没有在创建期间指定，则系统会赋给它默认的大小和位置。默认大小同系统相关，是由进程所允许的线程最大数目、进程地址空间的指定大小、系统资源使用的空间等决定的。线程的栈大小

必须足够大，以满足任何函数调用、线程调用的进程外部代码(如库代码)、局部变量存储的需要。有着多个线程的进程应当有足以满足其所有线程栈的栈段。分配给进程的地址空间限制了栈的大小，从而限制了每个线程栈的大小。线程栈地址对于访问有着不同属性的内存区域的应用程序，可能会很重要。当您指定栈的位置时，需要注意的是，线程要求多少空间以及确保该位置不会同其他对等线程的栈发生重叠。

分离的线程是那些已经从它们的创建者中分离出来的线程。它们在终止或退出时，不需要同其他对等线程或主线程进行同步。它们仍共享所属进程的地址空间，但是由于它们是分离的，创建它们的进程或线程不能够对它们进行控制。当线程终止时，线程的 id 和状态由系统保存。在默认情况下，一旦线程终止，该情况会通知给创建者，线程的 id 和状态会返回给创建者。如果线程是分离的，则不会使用资源来保存状态或线程 id。这些资源立即可以被系统重用。如果线程的创建者不需要在继续处理之前等待线程终止，或者如果线程不要求在终止时同其他对等线程进行任何类型的同步，那么该线程可以成为一个分离的线程。

线程从进程继承调度策略。线程有优先级，而且优先级最高的线程会在较低优先级的线程之前执行。通过对线程区分优先次序，系统中需要立即执行或响应的任务会被指定到处理器上并得到时间片。如果有着更高优先级的线程可以运行，则正在执行的线程会被抢占。线程的优先级可以被降低或提高。调度策略也决定了哪个线程会被指派到处理器上。可使用的策略有先进先出(FIFO)、轮询(RR)等。通常，没有必要在进程执行期间改变线程的调度属性。如果进程环境发生变动，改变了时间约束，使得您需要改进进程的性能，则可能需要对调度进行改动。但是要考虑到改动应用程序中指定进程的调度属性，可能会对应用程序的总体性能产生负面影响。

6.4 线程的结构

我们已经讨论了进程以及线程同它所属进程的关系。图 6-2 显示了包含多个线程的进程结构。进程通过上下文和属性区别于系统中其他进程，线程也可以通过上下文和属性区别于其他对等线程。进程有代码段、数据段和栈段。线程同进程共享代码段和栈段。进程的栈通常从内存的高地址开始，向下增长。线程栈以下一个线程栈的开始位置为边界。可以看到，线程栈包含其局部变量。进程的全局变量位于数据段中。Thread A 和 Thread B 的上下文包括线程 id、状态、优先级、处理器寄存器等。程序计数器(PC)指向代码段中函数 task1()和 task2()中下一条可执行指令。栈指针(SP)指向它们各自的栈的顶部。线程属性对象同一个线程或一组线程相关联。在本例中，两个线程使用相同的线程属性。

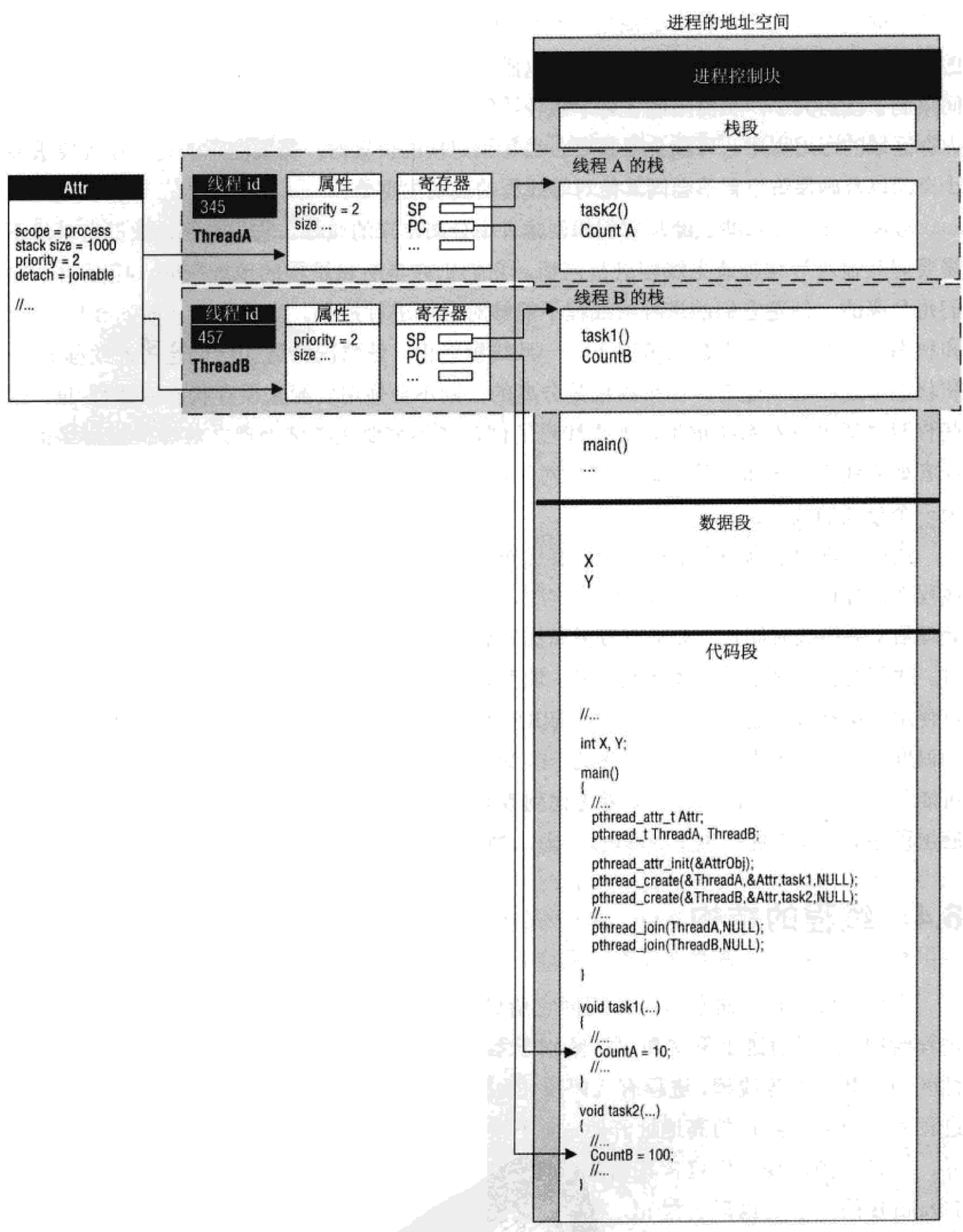


图 6-2

6.4.1 线程状态

线程是当进程被调度执行时的执行单元。如果进程中只有一个线程，该线程是指派到

处理器内核的主线程。如果进程有着多个线程，而且对于该进程有多个处理器可用，那么所有的线程都会被指派到处理器上。

当线程被调度到处理器内核上执行时，它会改变自身的状态。线程状态是指在任意指定时间所处的模式或情形。线程有着同第 5 章为进程介绍的状态和转换相同的状态和转换。有 4 种常见的状态：

- 可运行
- 运行(活动)
- 停止
- 休眠(阻塞)

存在如下的转换：

- 抢占
- 接到信号
- 分派
- 时间片用完

主线程可以决定整个进程的状态。如果主线程是唯一的线程，则主线程的状态同进程的状态相同。如果主线程在休眠，进程也在休眠。如果主线程在运行，进程也在运行。对于有着多个线程的进程，只有进程中所有线程都处于休眠或停止状态时，我们才能够认为整个进程休眠或停止。另一方面，如果一个线程是活动的(可运行或运行)，那么进程会被认为是活动的。

6.4.2 调度和线程竞争范围

线程有两种竞争范围：

- 进程竞争
- 系统竞争

有着进程竞争范围的线程与相同进程的其他线程进行竞争。这些是混合线程(用户级和内核级线程)，系统将创建内核级线程池，用户级线程将映射到它们。这些内核级线程是非绑定(unbound)的，可以映射到一个线程或多个线程。然后内核根据调度属性将内核线程调度到处理器上。

有着系统竞争范围的线程同系统范围内进程的线程进行竞争。这个模型由每个内核级线程对应一个用户级线程组成。用户线程在线程的生命期内都绑定到内核级线程上。内核线程单独负责在一个或多个处理器上调度线程执行。这个模型使用线程的调度属性，根据系统中所有线程来进行线程调度。线程的默认竞争范围根据实现定义。例如，对于 Solaris 10，默认竞争范围是进程，但是对于 SuSe Linux 2.6.13，默认竞争范围是系统范围。事实上，对于 SuSe Linux 2.6.13，根本不支持进程竞争范围。

图 6-3 显示了进程和系统两种线程竞争范围的区别。在有 8 个内核的多核环境中，有两个进程。进程 A 有 4 个线程，进程 B 有两个线程。进程 A 的 4 个线程中的 3 个线程竞

争范围为进程范围，一个线程的竞争范围为系统范围。进程B的两个线程中，一个线程竞争范围为进程范围，一个线程的竞争范围为系统范围。进程A中有着进程范围的线程竞争内核0和1，进程B中有着进程范围的线程将使用内核2。进程A和B中系统范围的线程竞争内核4和5。有着进程范围的线程映射到线程池。进程A的线程池中有3个内核级线程，进程B的线程池中有两个内核级线程。

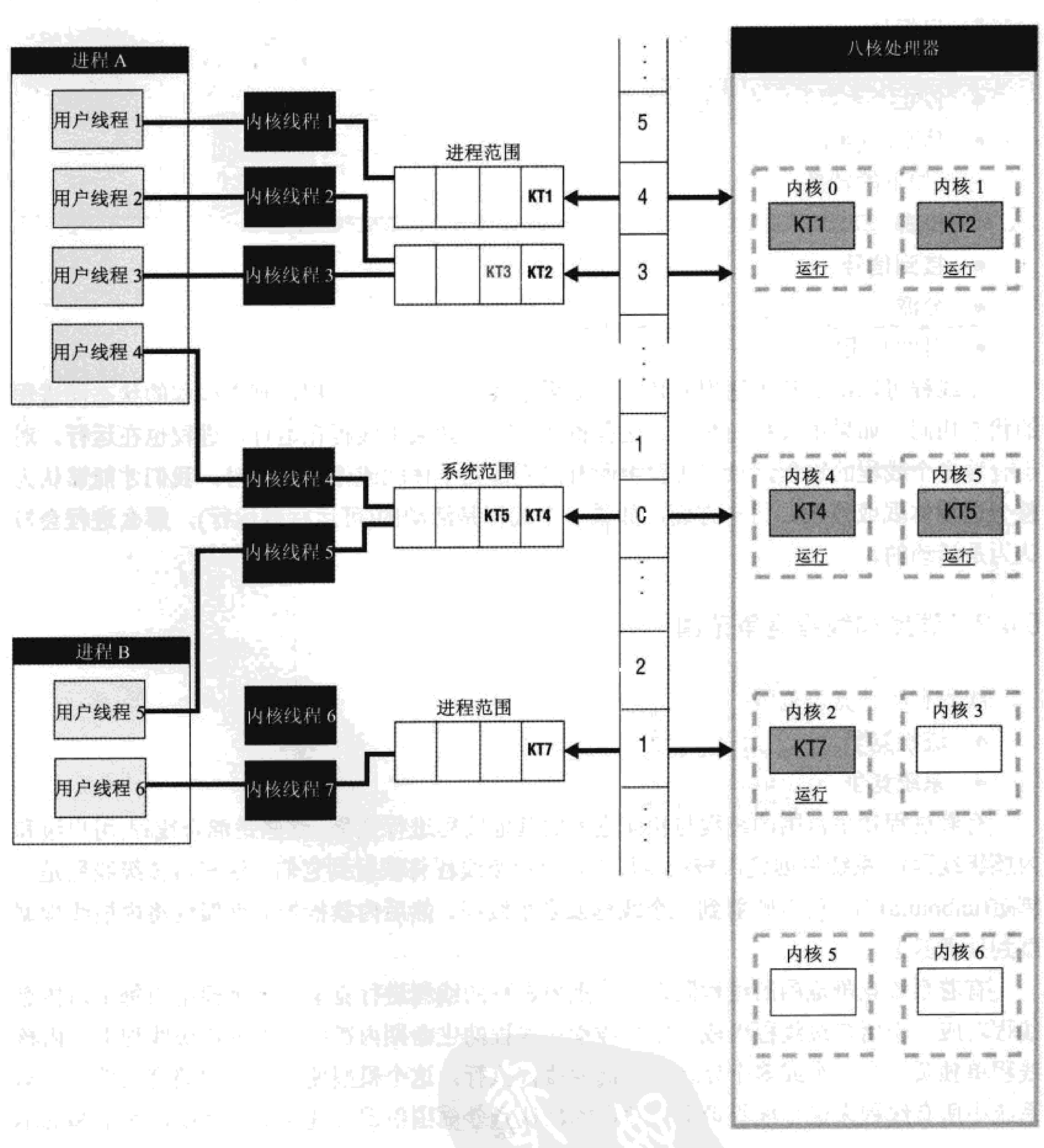


图6-3

竞争范围可以对应用程序的性能产生潜在影响。进程调度模型潜在地为做出调度决策提供较低的开销，因为只需要对一个进程中的线程进行调度。

6.4.3 调度策略和优先级

进程的调度策略和优先级属于主线程。每个线程可以有着与主线程不同的调度策略和优先级。优先级是拥有最大值和最小值的整数。当区分线程优先次序时，系统中要求立即执行或响应的任务会优先。在一个抢占式的操作系统中，如果有更高优先级(数字越小，优先级越高)而且竞争范围相同的线程可运行，则正在执行的线程会被抢占。

例如，在图 6-3 中，进程 A 有两个优先级为 3 的线程(2 和 3)，还有一个优先级为 4 的线程(1)。它们被分配到处理器内核 0 和 1 上。优先级为 4 和 3 的线程是可运行的，每个线程被分配到一个处理器上。一旦优先级为 3 的线程 3 变为活动的，则线程 1 会被抢占，线程 3 被分配到处理器。在进程 B 中，有一个线程有着进程竞争范围，而且它的优先级为 1。进程 B 只有一个可用处理器。有着系统竞争范围的线程不会被进程 A 或 B 中任何有着进程竞争范围的线程抢占。它们只和其他有着系统竞争范围的线程争夺处理器的使用。

就绪队列被组织为有序列表，其中每个元素是一个优先级。这在第 5 章中也讨论过。在第 5 章中，图 5-6 显示了就绪队列。列表中每个优先级是有着相同优先级的线程的队列。所有有着相同优先级的线程使用调度策略被分配到处理器上，调度策略为 FIFO、RR 或其他策略。

- 轮询(RR)调度策略认为所有线程有着相等的优先权，而且只让每个线程在一个时间片内使用处理器。任务的执行是交错的。例如，一个从文本文件中筛选字符的程序被分成 3 个线程。主线程为线程 1，它从文件中读入每一行，然后将读入的内容作为字符串写入到向量中。然后主线程创建 3 个线程并等待这些线程返回。每个线程有着自己的字符集，它们要将属于该字符集的字符从字符串中删除掉。每个线程利用两个队列，一个队列包含之前已经被另一个线程筛选过的字符串。一旦线程已经筛选了一个字符串，就会将结果写入到第二个队列。队列是全局数据。主线程位于就绪队列中，它可抢占地运行，直到创建了其他线程，然后会休眠，直到所有线程返回。其他的线程有着相等的优先级，而且使用轮询调度策略。线程不能够筛选尚未写到队列中的字符串，因此需要对源队列的访问进行同步。线程测试互斥量，如果互斥量被加锁，那么没有可用的字符串，或者源队列正在被使用。线程必须等待，直到互斥量被解锁。如果互斥量可用，则源队列中有字符串，而且源队列没有被使用。从队列中读取一个字符串，然后线程对字符串进行筛选，并将它写入到输出队列。输出队列作为另一个线程的源队列。在将来某个时刻，线程 2 被分配到处理器。它的源是包含所有将要被筛选的字符串的向量。线程 1 必须筛选字符串，然后将筛选过的字符串写入到它的输出队列，这样线程 2 才有要处理的内容，然后是线程 3，等等。RR 调度影响有着两个处理器内核的线程的执行。这种调度策略抑制了这个程序的适当执行。我们将在本章后面讨论使用正确的并发模型。

- 如果使用的是 FIFO 调度策略，而且优先权较高，则这些任务的执行不会发生交错。分配到处理器上的线程会一直占有处理器，直到它的执行结束。这种调度策略可用于有一组线程需要尽可能快地完成的应用程序。
- “其他”调度策略可以是一种定制的调度策略。例如，FIFO 调度策略可以被定制为允许线程随机解除阻塞，或者您可以使用能够加速线程执行的适当调度的策略。

6.4.4 调度分配域

FIFO 和 RR 调度策略在多个处理器上具有不同的特性。调度分配域决定了进程或应用程序的线程能够运行在哪些处理器集合上。调度策略会受到处理器内核数目以及进程中线程数目的影响。在从字符串中筛选字符的线程实例中，如果线程的数目同内核的数目相同，则使用 RR 调度策略会产生较好的吞吐量。但是线程的数目并不总是会同内核数目相同，有可能线程的数目多于内核数目。通常，依赖于内核的数目来大幅度影响应用程序的性能往往不是最好的方法。

6.5 简单的线程程序

下面是一个简单的线程程序示例。这个简单的多线程程序有一个主线程以及线程将要执行的函数。并发模型决定了线程创建和管理的方式。我们将在下一章中讨论并发模型。线程可以一次性创建，或者在特定条件下创建。在示例 6-1 中，使用了委托模型来说明简单的多线程程序。

示例 6-1

```
// Example 6-1 Using the delegation model in a simple threaded program.

using namspace std;
#include <iostream>
#include <pthread.h>

void *task1(void *X) //define task to be executed by ThreadA
{
   cout << "Thread A complete" << endl;
   return (NULL);
}

void *task2(void *X) //define task to be executed by ThreadB

{
   cout << "Thread B complete" << endl;
   return (NULL);
}
```

```
int main(int argc, char *argv[])
{
  pthread_t ThreadA,ThreadB; // declare threads

  pthread_create(&ThreadA,NULL,task1,NULL); // create threads
  pthread_create(&ThreadB,NULL,task2,NULL);
  // additional processing
  pthread_join(ThreadA,NULL); // wait for threads
  pthread_join(ThreadB,NULL);
  return (0);
}
```

在示例 6-1 中，主线程是 boss 线程。boss 线程声明两个线程，即 ThreadA 和 ThreadB。pthread_create( )创建线程并将它们同将要执行的任务关联起来。task1 和 task2 这两个任务被分别发送消息到标准输出。pthread_create( )使得线程立即执行它们分配到的任务。函数 pthread_join 的工作方式同 wait( )对进程的工作方式相同。主线程等待，直到两个线程都返回。图 6-4 包含了显示示例 6-1 的控制流的顺序图。在图 6-4 中，pthread_create( )使得在主线程的控制流中产生了分支。增加了两个并发执行的控制流，分别是 ThreadA 和 ThreadB。pthread_create( )在创建完线程之后立即返回，因为它是一个异步函数。当每个线程执行自己的指令集合时，pthread_join( )使得主线程等待，直到线程终止并重新加入到主控制流。

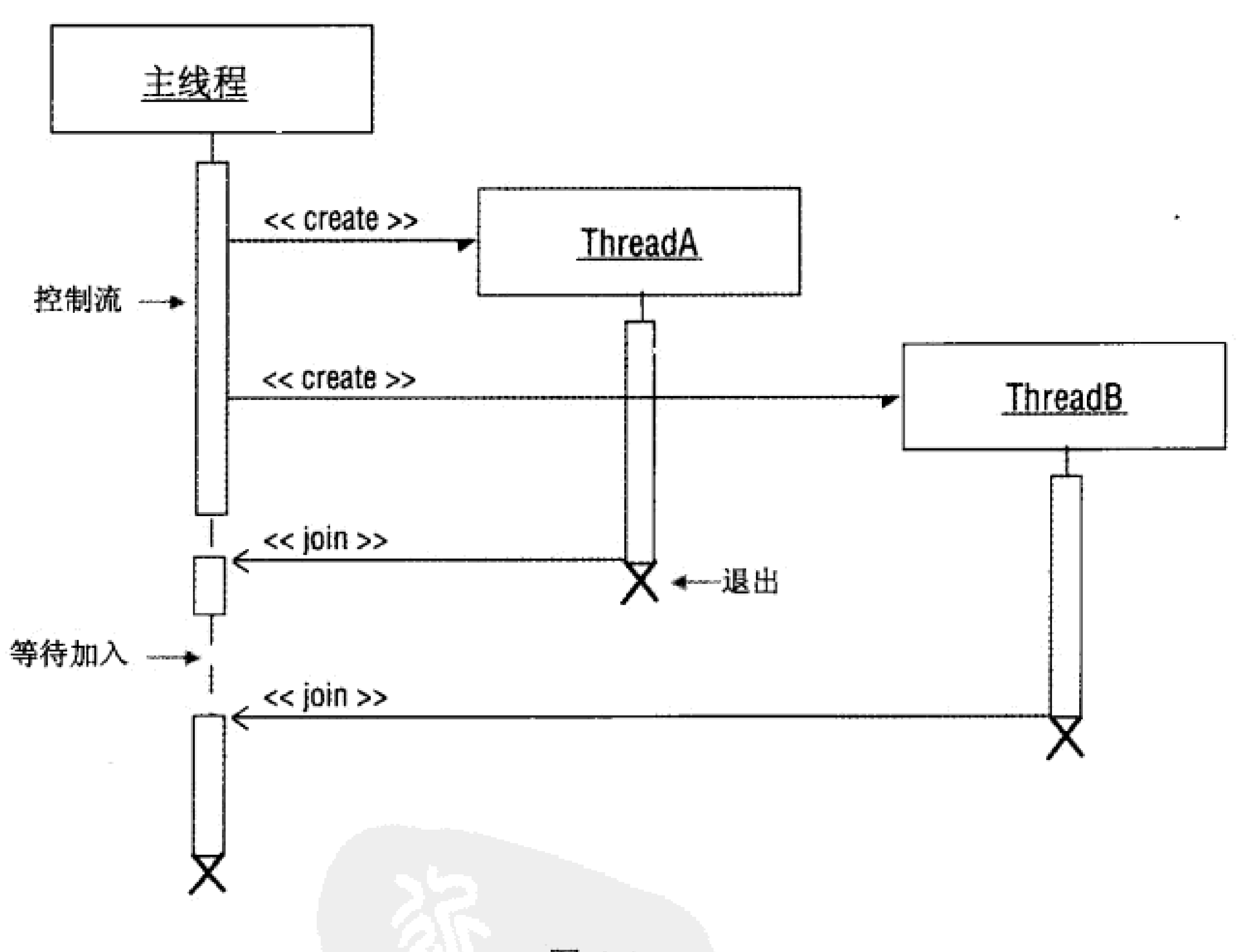


图 6-4

编译和链接线程程序

所有使用 POSIX 线程库的多线程程序必须包含这个头文件：

```
<pthread.h>
```

为了在 Unix 或 Linux 环境中使用 g++或 gcc 命令行编译器来编译和链接多线程程序，需要确保通过使用-l 编译器开关将 pthread 库链接到您的应用程序。这个开关之后紧接的就是库的名称：

```
-lpthread
```

这样将导致您的应用程序链接到同 POSIX 1003.1c 标准定义的多线程接口兼容的库。名为 libpthread.so 的 pthread 库应当位于系统保存标准库的目录下，通常为/usr/lib。如果它位于该标准目录下，则编译行应当类似于：

```
g++ -o a.out test_thread.cpp -lpthread
```

如果它没有位于标准位置，则使用-L 选项来让编译器在搜索标准位置之前，首先在特定目录下进行查看：

```
g++ -o a.out -L /src/local/lib test_thread.cpp -lpthread
```

这样就会告知编译器在搜索标准位置之前，首先在/src/local/lib 目录下查找 pthread 库。

注意：
如同您将在本章稍后部分看到的那样，本书中的完整程序都伴随有程序概要。程序概要包含实现细节，例如必需的头文件和库以及编译器和链接指令。概要还包含注释部分，其中包含了在执行程序时需要遵从的任何特殊考虑。对于示例，则没有概要。

6.6 创建线程

pthread 库可用于创建、维护和管理多线程程序和应用中的线程。当您创建一个多线程程序时，您可以在进程的执行期间的任何时候创建线程，因为它们是动态的。pthread_create( ) 在进程的地址空间中创建一个新的线程。

调用形式

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread, const pthread_attr_t
*restrict attr,
              void *(*start_routine)(void*), void *restrict arg);
```

参数 thread 指向将要创建的线程的线程句柄或线程 id。新的线程有着由属性对象 attr 所指定的属性。参数 thread 立即以 arg 指定的参数执行 start_routine 中的指令。如果函数成功地创建了线程，它返回线程 id 并将这个值保存在 thread 中。关键字 restrict 也被加了进来，目的是为了同之前的 IEEE 标准一致。下面是示例 6-1 中的对 pthread_create( )的调用：

```
pthread_create(&ThreadA,NULL,task1,NULL);
```

在这里，attr 为 NULL，新的线程 ThreadA 将会使用默认的线程属性。没有特定的参数。为了使线程具有新属性，会创建并初始化一个 pthread_attr_t 对象，并将这个对象传递给 pthread_create( )。新线程会在创建后具有 attr 的属性。如果在线程被创建之后，attr 发生改动，则不会对线程的任何属性产生影响。如果 start_routine 返回，线程会返回，就好像使用 start_routine 的返回值作为它的退出状态来调用 pthread_exit( )。

如果成功，函数返回 0。如果函数没有成功，则不会创建新的线程，而且函数返回一个错误号。如果系统没有资源来创建线程，或者达到了进程能够拥有的线程数目限制，那么函数会失败。如果线程属性无效，或者调用线程没有权限来设置必要的线程属性，函数也会失败。

6.6.1 向线程传递参数

程序清单 6-1 显示了主线程从命令行给线程执行的函数传递一个参数。命令行参数还用于确定要创建的线程的数目。

程序清单 6-1

```
//Listing 6-1 Passing arguments to a thread from the command line.

1  using namespace std;
2
3  #include <iostream>
4  #include <pthread.h>
5
6
7  void *task1(void *X)
8  {
9      int *Temp;
10     Temp = static_cast<int *>(X);
11
12     for(int Count = 0;Count < *Temp;Count++)
13     {
14         cout << "work from thread: " << Count << endl;
15     }
16     cout << "Thread complete" << endl;
17     return (NULL);
18  }
19
20
21
22  int main(int argc, char *argv[])
23  {
24     int N;
25
```

```
26    pthread_t MyThreads[10];
27
28    if(argc != 2){
29       cout << "error" << endl;
30        exit (1);
31    }
32
33    N = atoi(argv[1]);
34
35    if(N > 10){
36       N = 10;
37    }
38
39    for(int Count = 0;Count < N;Count++)
40    {
41       pthread_create(&MyThreads[Count],NULL,task1,&N);
42
43    }
44
45
46    for(int Count = 0;Count < N;Count++)
47    {
48       pthread_join(MyThreads[Count],NULL);
49
50    }
51    return(0);
52
53
54  }
55
56
```

第 26 行声明了 MyThreads，它是规模为 10 的数组，数组中的每一项类型为 pthread_t。N 持有命令行参数。在第 41 行，创建了 MyThreads 数组中的 N 个线程。N 作为类型为 void *的参数传递给每个线程。在函数 task1 中，该参数从 void *强制类型转换为 int *，如下所示：

```
10 Temp = static_cast < int * > (X);
```

函数执行一个循环，其迭代次数为传递给函数的值。函数将它的消息发送到标准输出。创建的每个线程执行这个函数。程序清单 6-1 的编译和执行指示包含在稍后位置的程序概要 6-1 中。

这个例子向线程函数传递一个命令行参数，并且使用该命令行参数来决定要创建的线程的数目。如果有必要向线程函数传递多个参数，您可以创建一个包含所有需要的参数的结构体(struct)或容器，然后将指向该数据结构的指针传递给线程函数。我们将在本章稍后部分介绍一种更容易的方式来完成这个目的，即创建线程对象。

程序概要 6-1

程序名：

program6-1.cc (程序清单 6-1)

描述：

从命令行接收一个整数，并将该值传递给线程函数。线程函数执行一个循环，该循环向标准输出发送消息。参数用作循环变量的结束条件。参数还决定了要创建的线程数目，每个线程执行相同的函数。

必需的库：

libpthread

必需的头文件：

<pthread.h> <iostream>

编译和链接指令：

c++ -o program6-1 program 6-1.cc –lpthread

测试环境：

Solaris 10、gcc 3.4.3 和 gcc 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

./program6-1 5

注释：

这个程序要求一个命令行参数。

6.6.2 结合线程

pthread_join( )用于结合或再次结合进程中的控制流。pthread_join( )导致调用线程将它的执行挂起，直到目标进程终止。它类似于进程所使用的 wait( )函数。这个函数由线程的创建者调用，该调用线程等待新的线程终止并返回，然后再次结合到调用线程的控制流中。如果线程句柄是全局的，则 pthread_join( )也可以被对等线程调用。这样使得任何线程可以将控制流同进程中任何其他线程结合。如果调用线程在目标线程返回之前被取消，这会导

致目标线程成为僵死线程。本章稍后将会讨论分离的线程。如果不同的对等线程同时对同一个线程调用 pthread_join( )函数，产生的行为未被定义。

调用形式

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

参数 thread 是调用线程正在等待的目标线程。如果目标线程成功返回，则它的退出状态保存在 value_ptr 中。如果目标线程不是一个可结合的线程，换句话说，如果它是作为分离的线程创建的，则函数失败。如果指定的 thread 线程不存在，则函数也会失败。

应当为所有可结合的线程调用 pthread_join( )函数。一旦线程被结合，则操作系统可以收回线程所使用的存储空间。如果可结合的线程没有同任何线程结合，或者如果调用 join 函数的线程被取消，则目标线程继续利用存储空间。这是一种类似于父进程尚未接受子进程的退出状态而产生的僵死进程的状态。子进程继续在进程表中占据一个表项。

6.6.3 获得线程 id

如本章前面所提到的，进程和它的地址空间内的线程共享资源。线程自身拥有的资源很少，但是线程 id 是线程的独特资源中的一种。函数 pthread_self( )返回调用线程的线程 id。

调用形式

```
#include <pthread.h>

pthread_t pthread_self(void);
```

当一个线程被创建之后，会将线程 id 返回到调用线程。一旦线程有了自己的 id 之后，可以将 id 传递给进程中的其他线程。这个函数返回线程 id，且未定义错误。

下面是调用这个函数的实例：

```
pthread_t ThreadId;
ThreadId = pthread_self();
```

线程调用这个函数，函数将线程 id 返回并赋给 pthread_t 类型的变量 ThreadId。

线程 id 还会被返回到 pthread_create( )的调用线程。如果成功创建了线程，则线程 id 保存在 pthread_t 中。

比较线程 id

您可以将线程 id 按照非透明类型进行处理。线程 id 可以进行比较，但是使用的不是一般的比较操作符。您可以通过调用 pthread_equal( )来比较两个线程的 id 是否相等。

调用形式

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

如果两个线程 id 指向相同的线程，则 pthread_equal( )返回一个非零值。如果它们指向不同的线程，则返回零。

6.6.4 使用 pthread 属性对象

线程有一组属性是可以在线程被创建时指定的。该组属性被封装在一个对象中，该对象可用来设置一个或一组线程的属性。线程属性对象的类型为 pthread_attr_t。这个结构体可用来设置这些线程属性：

- 线程栈的大小
- 线程栈的位置
- 调度继承机制、策略和参数
- 线程是否是分离的或可结合的
- 线程的范围

pthread_attr_t 拥有一些方法来设置和获取这些属性。表 6-3 列出了用于设置属性的方法。

表 6-3

属性函数的类型	pthread 属性函数
初始化	pthread_attr_init( ) pthread_attr_destroy( )
栈管理	pthread_attr_setstacksize( ) pthread_attr_getstacksize( ) pthread_attr_setguardsize( ) pthread_attr_getguardsize( ) pthread_attr_setstack( ) pthread_attr_getstack( ) pthread_attr_setstackaddr( ) pthread_attr_getstackaddr( )

(续表)

属性函数的类型	pthread 属性函数
分离状态	pthread_attr_setdetachstate( ) pthread_attr_getdetachstate( )
竞争范围	pthread_attr_setscope( ) pthread_attr_getscope( )
调度继承机制	pthread_attr_setinheritsched( ) pthread_attr_getinheritsched( )
调度策略	pthread_attr_setschedpolicy( ) pthread_attr_getschedpolicy( )
调度参数	pthread_attr_setschedparam( ) pthread_attr_getschedparam( )

pthread_attr_init( )和 pthread_attr_destroy( )函数用于初始化和销毁线程属性对象。

调用形式

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

pthread_attr_init( )使用默认值对线程属性对象的所有属性进行初始化。attr 是指向 pthread_attr_t 对象的指针。一旦 attr 被初始化之后，它的属性值可以通过表 6-3 中列出的 pthread_attr_set 函数进行改变。一旦属性已经进行了适当的更改，则 attr 可用作任何对 pthread_create( )函数的调用中的参数。如果函数调用成功，则返回 0，如果调用没有成功，则函数返回错误号。如果没有足够的内存来创建对象，则 pthread_attr_init( )函数失败。

pthread_attr_destroy( )函数可用于销毁由 attr 指定的 pthread_attr_t 对象。对这个函数的调用会删除所有同该线程属性对象相关联的隐藏的存储。如果成功，则函数返回 0，如果失败，函数返回一个错误号。

1. 属性对象的默认值

属性对象首先会通过对所有个别属性使用给定实现所使用的默认值进行初始化。有些实现不支持某个属性的可能值。如果成功完成，则 pthread_attr_init( )返回 0。如果返回的是一个错误号，这可能意味着该值不被支持。例如，对于竞争范围，Linux 环境不支持 PTHREAD_SCOPE_PROCESS。调用：

```
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

会返回一个错误编码。表 6-4 列出了 Linux 和 Solaris 环境的默认值。

表 6-4

pthread 属性函数	SuSE Linux 2.6.13 的默认值	Solaris 10 的默认值
pthread_attr_setdetachstate( )	PTHREAD_CREATE_JOINABLE	PTHREAD_CREATE_JOINABLE
pthread_attr_setscope( )	PTHREAD_SCOPE_SYSTEM (PTHREAD_SCOPE_PROCESS 不被支持)	PTHREAD_SCOPE_PROCESS
pthread_attr_setinheritsched( )	PTHREAD_EXPLICIT_SCHED	PTHREAD_EXPLICIT_SCHED
pthread_attr_setschedpolicy( )	SCHED_OTHER	SCHED_OTHER
pthread_attr_setschedparam( )	sched_priority = 0	sched_priority = 0
pthread_attr_setstacksize( )	未指定	NULL 由系统分配
pthread_attr_setstackaddr( )	未指定	NULL 1～2MB
pthread_attr_setguardsize( )	未指定	PAGESIZE

2. 使用 pthread 线程对象创建分离的线程

在默认情况下，当线程退出时，操作系统在线程同另一个线程结合时保存线程的完成状态以及线程 id。如果退出的线程不同其他线程结合，则称退出的线程是分离的(detached)。这种情况下不保存完成状态和线程 id。在分离的线程上不能够使用 pthread_join( )。如果使用了，则 pthread_join( )返回一个错误。

调用形式

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int *detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

pthread_attr_setdetachstate( )函数可用于设置属性对象的 detachstate 属性。detachstate

参数描述了线程是分离的还是可结合的。它可以为以下的值之一：

- PTHREAD_CREATE_DETACHED
- PTHREAD_CREATE_JOINABLE

值 PTHREAD_CREATE_DETACHED 使得所有使用这个属性对象的线程都被创建为分离的线程。值 PTHREAD_CREATE_JOINABLE 使得所有使用这个属性对象来创建的线程都被创建为可结合的线程。detachstate 的默认值是 PTHREAD_CREATE_JOINABLE。如果成功，则函数返回 0；如果没有成功，函数返回一个错误号。如果 detachstate 的值无效，则函数 pthread_attr_setdetachstate( )会失败。

函数 pthread_attr_getdetachstate( )返回属性对象的 detachstate。如果成功，则函数将 detachstate 的值返回给 detachstate 参数，并以 0 作为返回值。如果没有成功，则函数返回一个错误号。

已经在运行的线程也可以成为分离的。例如，线程可能不再对目标线程的结果感兴趣。线程可以分离，使得线程一旦退出，它的资源可以被收回。

调用形式

```
int pthread_detach(pthread_t tid);
```

在示例 6-2 中，ThreadA 是使用属性对象作为分离的线程创建的。ThreadB 是在创建之后分离的。

示例 6-2

```
// Example 6-2 Using an attribute object to create a detached thread and
changing
// a joinable thread to a detached thread.

//...

int main(int argc, char *argv[])
{

   pthread_t ThreadA,ThreadB;
   pthread_attr_t DetachedAttr;

   pthread_attr_init(&DetachedAttr);
   pthread_attr_setdetachstate(&DetachedAttr,PTHREAD_CREATE_DETACHED);
   pthread_create(&ThreadA,&DetachedAttr,task1,NULL);

   pthread_create(&ThreadB,NULL,task2,NULL);

   //...

   pthread_detach(pthread_t ThreadB);
```

```
    //pthread_join(ThreadB,NULL); cannot call once detached
    return (0);
}
```

示例 6-2 声明了一个属性对象 DetachedAttr。函数 pthread_attr_init( )用来初始化属性对象。ThreadA 是使用 DetachedAttr 属性对象创建的。这个属性对象已经将 detachstate 设置为 PTHREAD_CREATE_DETACHED。ThreadB 是使用 detachstate 的默认值，即 PTHREAD_CREATE_JOINABLE 来创建的。一旦创建完毕，就调用 pthread_detech( )。既然 ThreadB 是分离的，就不能够为这个线程调用 pthread_join( )。

6.7 管理线程

到此为止，我们已经谈论了如何创建线程、使用线程属性对象、创建可结合和分离的线程，以及返回线程 id 的方法。接下来我们将讨论如何管理线程。当创建有着多个线程的应用程序时，有多种方法可以控制线程的行为以及它们如何使用资源和竞争资源。管理线程中包含设置调度策略、线程优先级等部分，这能够提高线程的性能，因此也提高了应用程序的性能。线程的性能还由线程如何竞争资源所决定，无论是系统范围还是进程范围。可以通过使用线程属性对象来设置线程的调度策略、优先级和竞争范围。由于线程共享资源，因此对资源的访问必须加以同步。线程同步包含线程何时以及如何终止和取消。

6.7.1 终止线程

当线程到达程序指令的结尾时，就会终止。当线程终止后，pthread 库收回线程使用的系统资源并保存它的退出状态。线程也可能会在执行完它所有的指令之前，被另一个对等线程提前终止。线程可能已经破坏某些进程数据，因此必须被终止。

线程的执行可以通过几种方式来停止：

- 通过从它被分派的任务返回，返回时有或者没有退出状态或返回值
- 显式终止自身并提供一个退出状态
- 被相同地址空间中的其他线程取消

1. 自终止

线程可以通过调用 pthread_exit( )来自终止。

调用形式

```
#include <pthread.h>

int pthread_exit(void *value_ptr);
```

当可结合线程函数结束执行之后，它返回到将它作为目标线程调用 pthread_join( )的线程。当终止的线程调用 pthread_exit( )时，它在 value_ptr 中得到了退出状态。退出状态被返回到 pthread_join( )。还没有执行的取消清理处理任务(cancellation cleanup handler task)与线程特有数据的析构器一起执行。

当调用这个函数时，线程所使用的资源不会被释放。应用程序可以看到的进程资源也不会被释放，如互斥量和文件描述符。不会执行进程级清理动作。当进程中最后一个线程退出时，进程终止，且退出状态为 0。这个函数不能够返回到调用线程，也没有为它定义错误。

2. 终止对等线程

有时候一个线程有必要终止另一个对等线程。pthread_cancel( )用于终止对等线程。参数 thread 是要取消的线程。这个函数如果成功则返回 0，如果不成功就返回一个错误。当参数 thread 不对应任何现有线程时，函数 pthread_cancel( )会失败。

调用形式

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

应用程序中可能会有一个线程监视其他线程的工作。如果某个线程执行不力或不再需要，为了节省系统资源，有必要终止该线程。用户可能期望取消执行中的操作。多个线程可能用于解决一个问题，但是一旦某个线程得到了解答，所有其他线程可以被监视线程或得到解答的线程取消。

对 pthread_cancel( )的调用是取消一个对等线程的请求。这个请求可能立即被同意、稍后被同意、甚至被忽略。目标线程可能立即终止，或者延迟到它的执行中的某个逻辑点。线程可能必须在终止之前执行一些清理任务，线程也可以选择拒绝终止。

3. 理解取消过程

在取消一个对等线程的请求被同意时，会有一个取消过程同 pthread_cancel( )的返回异步发生。目标线程的取消类型和取消状态决定了取消何时真正发生。可取消性状态描述了线程的取消状况为可取消或不可取消。线程的可取消性类型决定了线程在收到取消请求后继续执行的能力。可取消性状态和类型是由线程自己动态设置的。

调用线程的可取消性状态和类型是由 pthread_setcancelstate( )和 pthread_setcanceltype( )设置的。pthread_setcancelstate( )将调用线程设置为 state 所指定的可取消性状态，并将之前的状态在 oldstate 中返回。pthread_setcanceltype( ) 将调用线程设置为 type 所指定的可取消性类型，并将之前的类型在 oldtype 中返回。

调用形式

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

用于设置线程取消状态的 state 和 oldstate 的值是：

- PTHREAD_CANCEL_DISABLE
- PTHREAD_CANCEL_ENABLE

PTHREAD_CANCEL_DISABLE 使得线程忽略取消请求。PTHREAD_CANCEL_ENABLE 使得线程允许取消请求。PTHREAD_CANCEL_ENABLE 是任何新近创建的线程的默认状态。如果成功，函数返回 0。如果没有成功，函数返回一个错误号。如果没有传递有效的 state 值，则 pthread_setcancelstate( )会失败。

函数 pthread_setcanceltype( )将调用线程的可取消性状态设置为 type 指定的类型，并将之前的状态通过 oldtype 返回。type 和 oldtype 的值可以为：

- PTHREAD_CANCEL_DEFFERED
- PTHREAD_CANCEL_ASYNCHRONOUS

PTHREAD_CANCEL_DEFFERED 使得线程推迟终止，直到它到达它的取消点。这是任何新近创建的线程的默认可取消性类型。PTHREAD_CANCEL_ASYNCHRONOUS 使得线程立即终止。如果成功，函数返回 0。如果不成功，函数返回一个错误号。如果没有传递有效的 type，则 pthread_setcanceltype( )会失败。

pthread_setcancelstate ( )和 pthread_setcanceltype( )共同使用来建立线程的可取消性。表 6-5 列出了状态和类型的组合以及对每种组合的作用的描述。

表 6-5

可取消性状态	可取消性类型	描　述
PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_DEFERRED	延迟取消；这是线程的默认取消状态和类型；当线程到达一个取消点或程序员通过调用 pthread_testcancel( )定义的取消点时，会发生线程的取消
PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS	异步取消；立即发生线程的取消
PTHREAD_CANCEL_DISABLE	Ignored	禁止取消；不会发生线程的取消

请看示例 6-3。

示例 6-3

```
// Example 6-3 task3 thread sets its cancelability state to allow thread
// to be canceled immediately.

void *task3(void *X)
{
   int OldState,OldType;

   // enable immediate cancelability

   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
   pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&OldType);

   ofstream Outfile("out3.txt");
   for(int Count = 1;Count < 100;Count++)
   {
      Outfile << "thread C is working: " << Count << endl;

   }
   Outfile.close();
   return (NULL);
}
```

在示例 6-3 中，取消被设置为立即发生。这意味着取消线程的请求可以在线程的函数执行中的任何一点发生。这样，线程可以打开文件并在对文件进行写入时被取消。

取消对等线程不应当轻易进行。有些线程具有非常敏感的性质，可能会要求安全保护以防止不合时宜的取消。在线程的函数中安装安全保护能够防止不期望的状况发生。例如，考虑共享数据的线程。根据使用的线程模型，一个线程可能正在处理数据，这些数据要传递给另一个线程进行处理。当线程处理数据时，它通过对互斥量的加锁获得对数据的独自占有。如果线程在互斥量释放之前被取消，这将会导致死锁。在数据能够再次被使用之前，可能会要求它处于某种状态。如果线程在完成这些之前被取消，会发生不期望的状况。根据线程正在进行的处理的类型，线程取消只应当在安全的时候进行。

重要的线程可能会完全防止取消。因此，线程取消应当被限制在不重要的线程上，而且是没有对资源加锁或者没有在执行重要代码的执行点上。您应当将线程的可取消性设置为适当的状态和类型。取消应当被延迟，直到所有重要的清理都发生了，例如释放互斥量、关闭文件等。如果线程有着取消清理处理程序任务，它们应当在取消之前进行。当返回最后的处理程序后，调用线程特有数据的析构器，然后线程被终止。

使用取消点

当推迟取消请求时，线程的终止会推迟到线程的函数执行的后期。当取消发生时，应当是安全的，因为它不处于对互斥量加锁、执行关键代码、令数据处于某种不可用状态的

情况中。代码的执行中，这些安全的位置是很好的取消点位置。取消点是一个检查点，线程在这里检查是否有任何取消请求未决，如果有，则终止。

取消点通过调用 pthread_testcancel( )来标记。这个函数用于检查任意的未决取消请求。如果有未决的请求，它会导致取消过程发生在这个函数被调用的位置。如果没有未决的取消请求，则函数继续执行，调用不产生任何影响。这个函数调用应当放置在代码中认为可以安全终止进程的任意位置。

调用形式

```
#include <pthread.h>

void pthread_testcancel(void);
```

在示例 6-3 中，线程的可取消性被设置为立即可取消。示例 6-4 使用延迟取消，这是默认的设置。对 pthread_testcancel( )的调用标记了在哪里取消这个线程是安全的，即在文件被打开之前，或线程关闭文件之后。

示例 6-4

```
// Example 6-4 task1 thread sets its cancelability state to be deferred.


void *task1(void *X)
{
   int OldState,OldType;

   //not needed default settings for cancelability
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
   pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,&OldType);

   pthread_testcancel();

   ofstream Outfile("out1.txt");
   for(int Count = 1;Count < 1000;Count++)
   {
      Outfile << "thread 1 is working: " << Count << endl;

   }
   Outfile.close();
   pthread_testcancel();return (NULL);
}
```

在示例 6-5 中，我们创建了两个线程并将它们取消。

示例 6-5

```
//Example 6-5 shows two threads being canceled.

 //...
int main(int argc, char *argv[])
{
   pthread_t Threads[2];
   void *Status;

   pthread_create(&(Threads[0]),NULL,task1,NULL);
   pthread_create(&(Threads[1]),NULL,task3,NULL);


    // ...

   pthread_cancel(Threads[0]);
   pthread_cancel(Threads[1]);


   for(int Count = 0;Count < 2;Count++)
   {
      pthread_join(Threads[Count],&Status);
      if(Status == PTHREAD_CANCELED){
         cout << "thread" << Count << " has been canceled" << endl;
      }
      else{
              cout << "thread" << Count << " has survived" << endl;
      }
   }
   return (0);
}
```

在示例 6-5 中，主线程创建两个线程，然后对每个线程发出了取消请求。主线程为每个线程调用了 pthread_join( )。函数 pthread_join( )如果试图同已经被终止的线程进行结合，则不会失败。结合函数提取被终止线程的退出状态。这种方式较好，因为发出取消请求的线程可能不是调用 pthread_join( )的线程。监视所有 worker 线程的工作情况可能是某个线程的主要任务，该线程也负责取消线程。另一个线程可能会通过调用 pthread_join( )函数来检测线程们的退出状态。这种信息可用于静态评估哪个线程的性能最好。在本例中，主线程在循环中结合并检查每个线程的退出状态。被取消的线程会返回退出状态 PTHREAD_CANCELED。

利用可安全取消的库函数和系统调用

在这些例子中，通过调用 pthread_testcancel( )标记的取消点放置在用户定义的函数中。当您从使用异步取消的线程函数中调用库函数时，取消这些线程是安全的吗？

pthread 库定义了可作为取消点的函数以及认为可异步安全取消的函数。这些函数阻塞调用线程，当调用线程被阻塞时，取消线程是安全的。这些是用作取消点的 pthread 库

函数：

- pthread_testcancel( )
- pthread_cond_wait( )
- pthread_timedwait( )
- pthread_join( )

如果状态为延迟取消的线程在做出对这些 pthread 库函数的调用时，有挂起的取消请求，则开始进行取消过程。

表 6-6 列出了一些被要求为取消点的 POSIX 系统调用。这些 pthread 和 POSIX 函数可以安全地用作延迟取消点，但是可能对于异步取消是不安全的。如果不是可安全异步取消的库调用在执行期间被取消，则可能导致库数据处于矛盾的状态。库可能已经为线程分配了内存，当线程被取消时，可能仍占有该内存。在这种情况下，当从不是可异步取消的线程进行这样的库调用时，有必要在调用之前改变可取消状态，并在函数返回后将可取消状态改变回来。

表 6-6

POSIX 系统调用(取消点)		
accept( )	nanosleep( )	sem_wait( )
aio_suspend( )	open( )	send( )
clock_nanosleep( )	pause( )	sendmsg( )
close( )	poll( )	sendto( )
connect( )	pread( )	sigpause( )
create( )	pthread_ccnd_timedwait( )	sigsuspend( )
fcntl( )	pthread_cond_wait( )	sigtimedwait( )
fsync( )	pthread_join( )	sigwait( )
getmsg( )	putmsg( )	sigwaitinfo( )
lockf( )	putpmsg( )	sleep( )
mq_receive( )	pwrite( )	system( )
mq_send( )	read( )	usleep( )
mq_timedreceive( )	readv( )	wait( )
mq_timedsend( )	recvfrom( )	waitpid( )
msgrcv( )	recvmsg( )	write( )
msgsnd( )	select( )	writev( )
msync( )	sem_timedwait( )	

对于其他不是可安全取消(异步或延迟)的库函数和系统函数，可能需要书写代码，通过禁止取消或将取消延迟到函数已经返回来防止线程终止。

示例 6-6 是对库调用或系统调用的封装。通过封装将可取消性改为延迟的、进行函数或系统调用，然后将可取消性重设回之前的类型。现在就可以安全地调用 pthread_testcancel( )了。

示例 6-6

```
//Example 6-6 shows a wrapper for system functions.

int OldType;
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,&OldType);
system_call(); //some library of system call
pthread_setcanceltype(OldType,NULL);
pthread_testcancel();

//...
```

终止之前进行清理

我们在前面提到过，线程在终止之前，可能需要执行一些最终的处理，例如关闭文件、将共享资源重设为一致的状态、释放锁和释放资源。pthread 库定义了一种机制，来为每个线程在终止之前执行最后的任务。清理栈(cleanup stack)同每个线程关联，在清理栈中包含了指向取消过程中要执行的例程的指针。函数 pthread_cleanup_push( )将一个指向例程的指针压入清理栈中。

调用形式

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

参数 routine 是要被压入栈中的函数指针，参数 arg 被传递给该函数。当线程在这些环境下退出时，会以 arg 为参数调用函数 routine:

- 当调用 pthread_exit( )时
- 当线程接受终止请求时
- 当线程使用非零 execute 值显式调用 pthread_cleanup_pop( )时

该函数不进行返回。

函数 pthread_cleanup_pop( )从调用线程的清理栈的顶部删除 routine 的指针。参数 execute 的值可以为 1 或 0。如果为 1，线程执行 routine，即使它没有被终止。线程从调用该函数之后的位置继续执行。如果值为 0，则指针会从栈顶移除，指向的函数不会被执行。

对于每次入栈，需要在相同的词法范围(lexical scope)中存在出栈。例如，task4( )要求函数退出或被取消时执行清理处理程序。

在示例 6-7 中，task4( )通过调用 pthread_cleanup_push( )函数将清理处理程序 cleanup_task4( )压入栈中。对于每个对 pthread_cleanup_push( )函数的调用，都要求有相应的 pthread_cleanup_pop( )。pop 函数被传递参数 0，意味着此时处理程序已经从清理栈中移出，但是此时尚未被执行。如果取消了执行 task4( )的线程，则会执行该处理程序。

示例 6-7

```
//Example 6-7 task4 () pushes cleanup handler cleanup_task4 () onto cleanup
stack.

void *task4(void *X)
{
   int *Tid;
   Tid = new int;
   // do some work
   //...
   pthread_cleanup_push(cleanup_task4,Tid);
   // do some more work
   //...
   pthread_cleanup_pop(0);
}
```

在示例 6-8 中，task5( )将清理处理程序 cleanup_task5( )压入到清理栈中。这个例子同上一个例子的区别在于传递给 pthread_cleanup_pop( )的参数为 1，意味着处理程序从清理栈中移出，并在这个位置被执行。无论执行 task5( )的线程是否被取消，处理程序都会被执行。清理处理程序 cleanup_task4( )和 cleanup_task5( )都是正规的函数，可用于关闭文件、释放资源、解锁互斥量等。

示例 6-8

```
//Example 6-8 task5 () pushes cleanup handler cleanup_task5 () onto cleanup
stack.

void *task5(void *X)
{
   int *Tid;
   Tid = new int;
   // do some work
   //...
   pthread_cleanup_push(cleanup_task5,Tid);
   // do some more work
   //...
   pthread_cleanup_pop(1);
}
```

6.7.2 管理线程的栈

管理线程的栈包括设置栈的大小以及确定栈的位置。线程的栈通常是由系统自动管理的。但是您应当清楚由默认的栈管理系统导致的系统特定限制。它们可能过于受限，这时就有必要进行一些栈管理了。如果应用程序有着大量的线程，则您可能不得不增加具有默认大小的栈的上限。如果应用程序利用递归或调用多个函数，则会需要很多栈帧。有些应用程序要求对地址空间进行精确的控制。例如，有着垃圾收集的应用程序必须跟踪内存的分配。

进程的地址空间分成代码段、静态数据段、堆和栈段。线程栈的位置和大小是从它所属的进程的栈中切分出来的。线程栈为线程调用且尚未退出的每个例程保存一个栈帧。栈帧包含临时变量、局部变量、返回地址以及线程回到之前执行的例程所需要的任何附加信息。一旦例程退出，该例程的栈帧会从栈中删除。图 6-5 显示了栈帧如何生成以及如何放置到栈中。

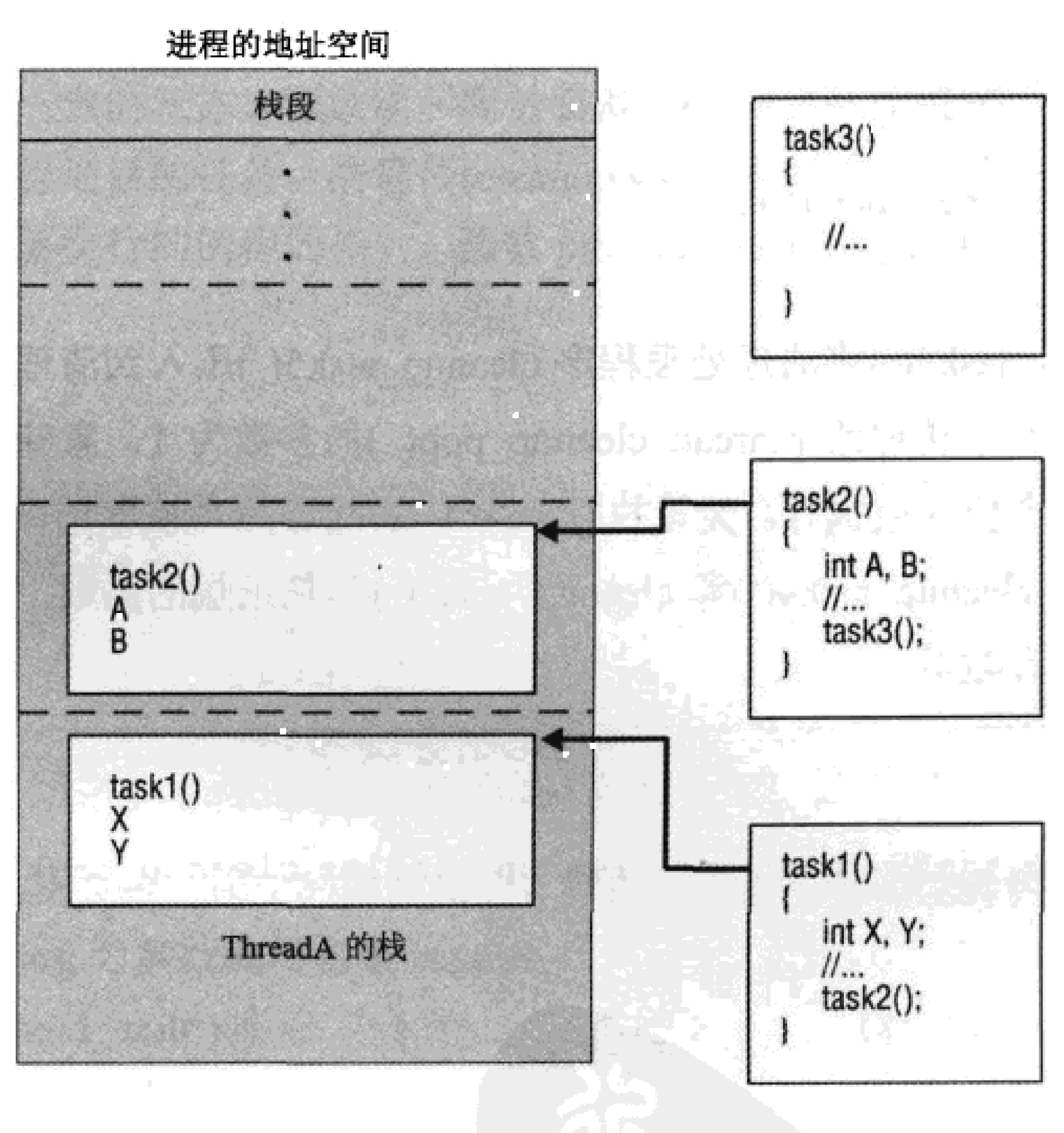


图 6-5

在图 6-5 中，ThreadA 执行 task1( )。task1()创建一些局部变量，进行一些处理，然后调用 task2( )。会为 task1( )创建一个栈帧并放置到栈中。task2( )创建局部变量，然后调用 task3( )。task2( )的栈帧也被放置到栈中。当 task3( )结束之后，控制流返回到 task2( )，它从栈中弹出。当 task2( )执行完之后，控制流返回到 task1( )，它也从栈中弹出。每个栈必须足够大，以容纳所有对等线程的函数的执行以及它们将会调用的例程链。线程栈的大小和位置可以通过由属性对象定义的几种方法进行设置或检测。

1. 设置栈的大小

有两个属性方法同线程栈的大小有关。

调用形式

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t *stacksize);
```

pthread_attr_getstacksize( )返回默认栈大小的最小值。attr 是从中提取默认栈大小的线程属性对象。当函数返回时，默认栈大小保存在 stacksize 中(以字节为单位)，且返回值为0。如果没有成功，则函数返回错误号。

pthread_attr_setstacksize( )设置栈大小的最小值。attr 是设置栈大小的线程属性对象。stacksize 是栈大小的最小值(以字节为单位)。如果函数成功，返回值为 0。如果没有成功，函数返回一个错误号。如果 stacksize 小于 PTHREAD_MIN_STACK 或小于系统最小值，则函数会失败。PTHREAD_STACK_MIN 很可能会是一个比由 pthread_attr_getstacksize( )返回的默认栈最小值还要小的最小值。在增加线程栈的最小大小之前，需要考虑由 pthread_attr_getstacksize( )返回的值。

在示例 6-9 中，线程的栈大小通过使用线程属性对象进行了更改。它从属性对象中提取默认值，然后判断默认值是否小于期望的最小栈大小。如果是，则将偏移量加到默认栈大小上，得到的结果成为线程新的最小栈大小。

示例 6-9

```
// Example 6-9 Changing the stack size of a thread using an offset.

#include <limits.h>
//...

pthread_attr_getstacksize(&SchedAttr,&DefaultSize);
if(DefaultSize < PTHREAD_STACK_MIN){
   SizeOffset = PTHREAD_STACK_MIN - DefaultSize;
   NewSize = DefaultSize + SizeOffset;
   pthread_attr_setstacksize(&Attr1,(size_t)NewSize);
}
```

在设置大小时需要进行权衡。栈大小是固定的，较大的栈意味着发生栈溢出的可能性较小，但是另一方面，较大的栈意味着在栈的交换空间和实际内存方面占用得更多。

PDG

注意：

设置栈大小和栈的位置可能使得您的程序无法移植。您在一个平台上为程序设置的栈大小和位置可能无法匹配上另一个平台上的栈大小和位置。

2. 设置线程栈的位置

一旦您决定管理线程的栈，您可以通过使用这些属性对象方法来提取并设置栈的位置。

调用形式

```
#include <pthread.h>

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                              void **restrict stackaddr);
```

pthread_attr_setstackaddr( )将栈的基地址设置为stackattr指定的地址，该栈用于使用attr线程属性对象创建的线程。地址addr应当位于进程的虚拟地址空间中。栈的大小最少应当等于由PTHREAD_STACK_MIN所指定的最小栈大小。如果成功，函数返回0。如果不成功，函数返回一个错误号。

pthread_attr_getstackaddr( )获取线程的栈地址的基地址，该线程是通过使用由attr指定的线程属性对象创建的。地址返回并保存在stackaddr中。如果成功，则函数返回0。如果不成功，函数返回一个错误号。

3. 设置一个函数设置栈大小和位置

栈属性(大小和位置)可以通过使用一个函数来设置。

调用形式

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                          size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr, size_t *restrict
stacksize);
```

函数pthread_attr_setstack( )为使用指定的属性对象attr创建的线程设置栈大小和位置。栈的基地址设置为stackaddr，栈的大小设置为stacksize。pthread_attr_getstack( )提取使用指

定属性对象 attr 创建的线程的栈大小和位置。如果提取成功，则栈的位置保存在 stackaddr，栈的大小保存在 stacksize。如果成功，这些函数会返回 0。如果不成功，则返回错误号。如果 stacksize 小于 PTHREAD_STACK_MIN 或超出一些根据实现定义的限制，则 pthread_attr_setstack( )会失败。

6.7.3 设置线程调度和优先级

线程是独立执行的。它们被分派到处理器内核上，并执行分给它们的任务。每个线程均有一个调度策略和优先级，决定何时以及如何分配到处理器上。线程或线程组的调度策略可以使用这些函数通过属性对象来设置：

调用形式

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
void pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict param);
```

pthread_attr_setinheritesched( )用于确定如何设置线程的调度属性，可以从创建者线程或从一个属性对象来继承调度属性。inheritsched 可以为如下值。

- PTHREAD_INHERIT_SCHED：线程调度属性是从创建者线程继承得到，attr 的任何调度属性都被忽略。
- PTHREAD_EXPLICIT_SCHED：线程调度属性设置为属性对象 attr 的调度属性。

如果 inheritsched 值是 PTHREAD_EXPLICIT_SCHED，则 pthread_attr_setschedpolicy( )被用于设置调度策略，而 pthread_attr_setschedparam( )被用于设置优先级。

pthread_attr_setschedpolicy( )设置线程属性对象 attr 的调度策略。policy 的值可以为在 <sched.h>头文件中定义的以下值。

- SCHED_FIFO：先进先出调度策略，执行线程运行到结束。
- SCHED_RR：轮询调度策略，按照时间片将每个线程分配到处理器上。
- SCHED_OTHER：另外的调度策略(根据实现定义)。这是任何新创建线程的默认调度策略。

使用 pthread_attr_setschedparam( )可以设置调度策略所使用的属性对象 attr 的调度参数。param 是包含参数的结构体。sched_param 结构体至少需要定义这个数据成员：

```
struct sched_param {
   int sched_priority;
   //...
```

```
};
```

它可能还有其他的数据成员，以及多个用来返回和设置最小优先级、最大优先级、调度器、参数等的函数。如果调度策略是 SCHED_FIFO 或 SCHED_RR，那么要求具有值的唯一成员是 sched_priority。

按照如下方法使用 sched_get_priority_max( )和 sched_get_priority_max( )，可以得到优先级的最大值和最小值。

调用形式

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

两个函数都以调度策略 policy 为参数，目的是获得对应调度策略的优先级值，而且都返回调度策略的最大或最小优先级值。

示例 6-10 显示了如何使用线程属性对象设置线程的调度策略和优先级。

示例 6-10

```
// Example 6-10 Using the thread attribute object to set scheduling
// policy and priority of a thread.

#include <pthread.h>
#include <sched.h>

//...

pthread_t ThreadA;
pthread_attr_t SchedAttr;
sched_param SchedParam;
int MidPriority,MaxPriority,MinPriority;

int main(int argc, char *argv[])
{
   //...

   // Step 1: initialize attribute object
   pthread_attr_init(&SchedAttr);

   // Step 2: retrieve min and max priority values for scheduling policy
   MinPriority = sched_get_priority_max(SCHED_RR);
   MaxPriority = sched_get_priority_min(SCHED_RR);
```

```
    // Step 3: calculate priority value
    MidPriority = (MaxPriority + MinPriority)/2;

    // Step 4: assign priority value to sched_param structure
    SchedParam.sched_priority = MidPriority;

    // Step 5: set attribute object with scheduling parameter
    pthread_attr_setschedparam(&SchedAttr,&SchedParam);

    // Step 6: set scheduling attributes to be determined by attribute object
    pthread_attr_setinheritsched(&SchedAttr,PTHREAD_EXPLICIT_SCHED);

    // Step 7: set scheduling policy
    pthread_attr_setschedpolicy(&SchedAttr,SCHED_RR);

    // Step 8: create thread with scheduling attribute object
    pthread_create(&ThreadA,&SchedAttr,task1,NULL);

    //...
}
```

在示例 6-10 中，ThreadA 的调度策略和优先级是使用线程属性对象 SchedAttr 来设置的。通过 8 个步骤完成：

(1) 初始化属性对象

(2) 为调度策略提取最大和最小优先级值

(3) 计算优先级值

(4) 将优先级值赋给 sched_param 结构体

(5) 使用调度参数设置属性对象

(6) 将调度属性设置为由属性对象决定

(7) 设置调度策略

(8) 使用调度属性对象创建一个线程

在示例 6-10 中，我们将优先级设置为一个平均值。但是优先级可以设置为介于线程调度策略所允许的最大和最小优先级值之间的任何值。有了这些方法，调度策略和优先级可以在线程被创建或运行之前，先设置在线程属性对象中。为了动态改变调度策略和优先级，可以使用 pthread_setschedparam( )和 pthread_setschedprio( )。

调用形式

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *restrict policy,
                          struct sched_param *restrict param);
```

```
int pthread_setschedprio(pthread_t thread, int prio);
```

pthread_setschedparam( )不需要使用属性对象即可直接设置线程的调度策略和优先级。thread 是线程的 id，policy 是新的调度策略，param 包含调度优先级。如果成功，则 pthread_getschedparam( )返回调度策略和调度参数，并将它们的值分别保存在 policy 和 param 参数中。如果成功，则两个函数都返回 0。如果不成功，两个函数都返回错误号。表 6-7 列出了这些函数可能失败的条件。

pthread_setschedprio( )用来设置正在执行中的线程的调度优先级，该进程的 id 由 thread 指定。prio 指定了线程的新调度优先级。如果函数失败，线程的优先级不发生变化，返回一个错误号。如果成功，则函数返回 0。表 6-7 也列出了这个函数可能失败的条件。

表 6-7

pthread 调度和优先级函数	失败的条件
int pthread_getschedparam (pthread_t thread, int *restrict policy, struct sched_param *restrict param);	thread 参数所指向的线程不存在
int pthread_setschedparam(pthread_t thread, int *policy, const struct sched_param *param);	参数 policy 或同参数 policy 关联的调度参数之一无效； 参数 policy 或调度参数之一的值不被支持； 调用线程没有适当的权限来设置指定线程的调度参数或策略； 参数 thread 指向的线程不存在； 实现不允许应用程序将参数改动为特定的值
int pthread_setschedprio(pthread_t thread, int prio);	参数 prio 对于指定线程的调度策略无效； 参数 prio 的值不被支持； 调用线程没有适当的权限来设置指定线程的调度优先级； 参数 thread 指向的线程不存在； 实现不允许应用程序将优先级改变为指定的值

注意：

要记得仔细考虑为何有必要改变运行线程的调度策略或优先级。这可能会严重影响应用程序的总体性能。有着较高优先级的线程会抢占运行的较低优先级的线程。这可能会产生饿死，即线程持续被抢占，因此无法完成执行。

6.7.4 设置线程的竞争范围

线程的竞争范围决定了线程同哪些其他线程竞争处理器的使用。竞争范围是由线程属性对象设置的。

调用形式

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                    int *restrict contentionscope);
```

pthread_attr_setscope( )设置由 attr 指定的线程属性对象的竞争范围属性。线程属性对象的竞争范围会设置为保存在 contentionscope 中的值。contentionscope 可以为以下值。

- PTHREAD_SCOPE_SYSTEM：系统调度竞争范围
- PTHREAD_SCOPE_PROCESS：进程调度竞争范围

系统竞争范围意味线程同系统范围内其他进程的线程进行竞争。pthread_attr_getscope( )从 attr 指定的线程属性对象返回竞争范围属性。如果函数成功，则返回线程属性对象的竞争范围，并保存到 contentionscope 中。这两个函数如果成功则返回 0，否则返回错误号。

6.7.5 使用 sysconf( )

了解系统的线程资源限制是使得应用程序恰当地管理它们的关键。前面已经讨论了利用系统资源的示例。当设置线程的栈大小时，最小值为 PTHREAD_MIN_STACK。栈大小不应当低于由 pthread_attr_getstacksize( )返回的默认栈大小的最小值。每个进程的最大线程数决定了能够为每个进程创建的 worker 线程的上限。函数 sysconf( )用于返回可配置系统限制或选项的当前值。系统中定义了同线程、进程和信号量相关的多个变量和常量。在表 6-8 中，列出了部分变量和常量。

表 6-8

变　量	名字值(Name Value)	描　述
_SC_THREADS	_POSIX_THREADS	支持线程
_SC_THREAD_ATTR_STACKADDR	_POSIX_THREAD_ATTR_STACKADDR	支持线程栈地址属性
_SC_THREAD_ATTR_STACKSIZE	_POSIX_THREAD_ATTR_STACKSIZE	支持线程栈大小属性
_SC_THREAD_STACK_MIN	PTHREAD_STACK_MIN	线程栈存储区的最小大小，以字节为单位

(续表)

变　　量	名字值(Name Value)	描　　述
_SC_THREAD_THREADS_MAX	PTHREAD_THREADS_MAX	每个进程的最大线程数
_SC_THREAD_KEYS_MAX	PTHREAD_KEYS_MAX	每个进程关键字的最大数目
_SC_THREAD_PRIO_INHERIT	_POSIX_THREAD_PRIO_INHERIT	支持优先级继承选项
_SC_THREAD_PRIO	_POSIX_THREAD_PRIO_	支持线程优先级选项
_SC_THREAD_PRIORITY_SCHEDULING	_POSIX_THREAD_PRIORITY_SCHEDULING	支持线程优先级调度选项
_SC_THREAD_PROCESS_SHARED	_POSIX_THREAD_PROCESS_SHARED	支持进程共享同步
_SC_THREAD_SAFE_FUNCTIONS	_POSIX_THREAD_SAFE_FUNCTIONS	支持线程安全函数
_SC_THREAD_DESTRUCTOR_ITERATIONS	_PTHREAD_THREAD_DESTRUCTOR_ITERATIONS	决定在线程退出时尝试销毁线程特定数据的尝试次数
_SC_CHILD_MAX	CHILD_MAX	每个UID允许的最大进程数目
_SC_PRIORITY_SCHEDULING	_POSIX_PRIORITY_SCHEDULING	支持进程调度
_SC_REALTIME_SIGNALS	_POSIX_REALTIME_SIGNALS	支持实时信号
_SC_XOPEN_REALTIME_THREADS	_XOPEN_REALTIME_THREADS	支持X/Open POSIX实时线程特性组
_SC_STREAM_MAX	STREAM_MAX	决定进程能够打开的流的数目
_SC_SEMAPHORES	_POSIX_SEMAPHORES	支持信号量
_SC_SEM_NSEMS_MAX	SEM_NSEMS_MAX	决定线程能够拥有的信号量的最大数目
_SC_SEM_VALUE_MAX	SEM_VALUE_MAX	决定信号量的最大值
_SC_SHARED_MEMORY_OBJECTS	_POSIX_SHARED_MEMORY_OBJECTS	支持共享内存对象

下面是调用sysconf( )的示例：

```
if(PTHREAD_STACK_MIN == (sysconf(_SC_THREAD_STACK_MIN))){
   //...
```

```
}
```

代码中将 sysconf( )返回的_SC_THREAD_STACK_MIN 的值同 PTHREAD_STACK_MIN 这个常量值进行了比较。

6.7.6 线程安全和库

如果不需要采取任何其他动作，库中的函数就可以在某个时刻被多个线程调用，则称该库是线程安全或可重入的。在设计多线程应用程序时，必须小心地确保并发执行的函数是线程安全的。我们已经讨论了如何使用户定义的函数成为线程安全的，但是应用程序经常会调用系统定义的库或第三方提供的库。我们前面讨论了在取消点安全的系统函数，但是这些函数和库中，部分是线程安全的，而部分不是。如果函数不是线程安全的，则意味着该函数：

- 包含静态变量
- 访问全局数据
- 是不可重入的

如果函数包含静态变量，那么这些变量在函数调用之间保持它们的值。函数要求静态变量的值，才能够正确地操作。当多个并发线程调用这个函数时，会产生竞争条件。

如果函数更改一个全局变量，那么多个调用该函数的线程可能每个都尝试更改该全局变量。如果对全局变量的多个并发访问不是同步的，则也可能会产生竞争条件。考虑多个并发线程执行设置 errno 的函数。对于某些线程，函数失败，errno 被设置为错误消息。与此同时，其他线程成功执行。依赖于编译器实现，errno 是线程安全的，但假如它不是线程安全的，那么当一个线程检查 errno 的状态时，它会报告哪个消息？

可重入代码是在使用期间不能够被改变的代码块。可重入代码通过去除对全局变量以及可改动静态数据的引用避免竞争条件。多个并发线程或进程可以共享代码，而其不会发生竞争条件。POSIX 标准将多个函数定义为可重入的。可以简单地通过函数名后缀_r 来同对应的不可重入的函数进行识别。下面列出了部分可重入函数：

- getgrgid_r( )
- getgrnam_r( )
- getpwuid_r( )
- sterror_r( )
- strtok_r( )
- readdir_r( )
- rand_r( )
- ttyname_r( )

如果函数访问未经保护的全局变量、包含静态可更改变量、不可重入，则认为该函数不是线程安全的。

1. 使用多线程版本的库和函数

系统库和第三方提供的库可能为它们的标准库提供两个不同的版本，一个版本用于单线程应用程序，另一个版本用于多线程应用程序。只要预计到可能会用于多线程环境，就要链接到库的多线程版本。其他环境不要求链接到库的多线程版本的多线程应用程序，只要求为声明可重入版本的函数定义一些宏。然后这个应用程序就可以作为线程安全的来进行编译。

并不总是可能使用函数的多线程版本。在某些实例中，对于给定的编译器或环境，特定函数的多线程版本不可用。某些函数的接口不能够简单地变成线程安全的。此外，您可能面临增加线程到一个环境中，该环境使用了只打算用在单线程环境中的函数。在这些情况下，可以在程序中使用互斥量来封装所有这样的函数。

例如，某个程序有 3 个并发执行的线程。其中两个线程 ThreadA 和 ThreadB 都并发地执行 task1( )，task1( )不是线程安全的。第三个线程，即 ThreadC，执行 task2( )。为了解决 task1( )的问题，方法就是简单地通过一个互斥量来封装 ThreadA 和 ThreadB 对 task1( )的访问：

```
ThreadA
{
   lock()
   task1()
   unlock()
}

ThreadB
{
   lock()
   task1()
   unlock()
}

ThreadC
{
   task2()
}
```

如果这样做，则任意时刻只有一个线程访问 task1( )。但是如果 task1( )和 task2( )都改动相同的全局变量或静态变量会怎样呢？尽管 ThreadA 和 ThreadB 对 task1( )使用了互斥量，但 ThreadC 执行 task2( )同它们两者之一是并发的。在这种情况下，会发生竞争条件。为了避免竞争条件，需要对全局数据的访问进行同步。我们将在第 7 章中讨论这个话题。

2. 线程安全标准输出

为了示范在涉及 iostream 库时的另一种竞争条件，假定有两个线程 ThreadA 和 ThreadB 向标准输出流 cout 发送输出。cout 是类型为 ostream 的对象，使用插入符(>>)和提取符(<<)

来调用 cout 对象的方法。这些方法是线程安全的吗？如果 ThreadA 在发送如下的消息：

Global warming is a real problem.

到 stdout，且 ThreadB 发送如下消息：

Global warming is not a real problem.

那么输出是否会交错并产生如下的消息？

Global warming is a Global warming is not a real problem real problem.

在某些情况下，线程安全函数是通过原子(atomic)函数来实现的。原子函数是一旦开始执行就不能够被中断的函数。对于 cout，如果插入操作是原子的，那么这种交错不会发生。当您多次调用插入符操作时，它们会像按照串行顺序那样来执行。先显示 ThreadA 的消息，然后显示 ThreadB 的消息，或者反之。这是将函数或操作串行化，使之成为线程安全的实例。

这并非令函数线程安全的唯一途径。如果没有不利的影响，函数可能会交错地操作。例如，如果某个方法向未排序的结构中增加或删除元素，而且两个不同的线程调用该方法，那么交错它们的操作将不会有不利的影响。

如果不知道库中的哪些函数是线程安全的，而哪些不是，您有以下 3 种选择：

- 所有非线程安全函数的使用限制为单线程使用
- 不使用任何非线程安全函数
- 将所有潜在的非线程安全的函数封装到单独的一套同步机制中

为了扩展最后一个选择，您可以为所有将被用在多线程应用程序中的所有非线程安全函数创建接口类。包装器的思想已经在本章前面部分中为系统调用设置取消点时示范过。非线程安全的函数封装在一个接口类中。该类可以同适当的同步对象结合，并可被宿主类(host class)通过继承或组合来使用。这种方法降低了竞争条件的可能性，将在第 7 章中讨论。然而，首先我们希望讨论第 4 章引入的 thread_object 接口类，并对它进行扩展，以封装线程属性对象。

6.8 扩展线程接口类

线程接口类是在第 4 章引入的。接口类的作用就像包装器，使得某些事物显得同正常的情况不同。接口类提供的新的接口是为了使得类更易用、功能性更强、更安全或语义上更加正确。在本章中，我们已经介绍了很多用于管理线程的 pthread 函数，包括线程属性对象的创建和使用。thread_object 类是一个简单的框架类。它的目的是封装 pthread 线程接口并提供面向对象的语义和组件，使得您可以更加容易地实现在 SDLC 中产生的模型。现在我们将对 thread_object 类进行扩展，封装线程属性对象的一些功能。程序清单 6-2 显示了新的 thread_object 类和 user_thread 类的声明。

程序清单 6-2

```
//Listing 6-2 Declaration of the new thread_object and user_thread.

1   #ifndef __THREAD_OBJECT_H
2   #define __THREAD_OBJECT_H
3
4   using namespace std;
5   #include <iostream>
6   #include <pthread.h>
7   #include <string>
8
9   class thread_object{
10       pthread_t Tid;
11
12  protected:
13      virtual void do_something(void) = 0;
14      pthread_attr_t SchedAttr;
15      struct sched_param SchedParam;
16      string Name;
17      int NewPolicy;
18      int NewState;
19      int NewScope;
20  public:
21      thread_object(void);
22      ~thread_object(void);
23      void setPriority(int Priority);
24      void setSchedPolicy(int Policy);
25      void setContentionScope(int Scope);
26      void setDetached(void);
27      void setJoinable(void);
28
29      void name(string X);
30      void run(void);
31      void join(void);
32      friend void *thread(void *X);
33  };
34
35
36  class filter_thread : public thread_object{
37  protected:
38      void do_something(void);
39  public:
40      filter_thread(void);
41      ~filter_thread(void);
42  };
43
44  #endif
45
46
```

对于 thread_object，我们加入了设置如下内容的方法：

- 调度策略
- 优先级
- 状态
- 竞争范围

我们定义了 filter_thread 类，其中定义了 do_something( )方法，而不是在 user_thread 中定义该方法。这个类在下一章介绍同步时会使用到。

程序清单 6-3 是新的 thread_object 类的定义。

程序清单 6-3

```
//Listing 6-3  A definition of the new thread_object class.

1   #include "thread_object.h"
2
3   thread_object::thread_object(void)
4   {
5      pthread_attr_init(&SchedAttr);
6      pthread_attr_setinheritsched(&SchedAttr,PTHREAD_EXPLICIT_SCHED);
7      NewState = PTHREAD_CREATE_JOINABLE;
8      NewScope = PTHREAD_SCOPE_PROCESS;
9      NewPolicy = SCHED_OTHER;
10  }
11
12  thread_object::~thread_object(void)
13  {
14
15  }
16
17  void thread_object::join(void)
18  {
19     if(NewState == PTHREAD_CREATE_JOINABLE){
20        pthread_join(Tid,NULL);
21     }
22  }
23
24  void thread_object::setPriority(int Priority)
25  {
26     int Policy;
27     struct sched_param Param;
28
29     Param.sched_priority = Priority;
30     pthread_attr_setschedparam(&SchedAttr,&Param);
31  }
32
33
34  void thread_object::setSchedPolicy(int Policy)
```

```
35  {
36     if(Policy == 1){
37        pthread_attr_setschedpolicy(&SchedAttr,SCHED_RR);
38        pthread_attr_getschedpolicy(&SchedAttr,&NewPolicy);
39     }
40
41     if(Policy == 2){
42        pthread_attr_setschedpolicy(&SchedAttr,SCHED_FIFO);
43        pthread_attr_getschedpolicy(&SchedAttr,&NewPolicy);
44     }
45  }
46
47
48  void thread_object::setContentionScope(int Scope)
49  {
50     if(Scope == 1){
51        pthread_attr_setscope(&SchedAttr,PTHREAD_SCOPE_SYSTEM);
52        pthread_attr_getscope(&SchedAttr,&NewScope);
53     }
54
55     if(Scope == 2){
56        pthread_attr_setscope(&SchedAttr,PTHREAD_SCOPE_PROCESS);
57        pthread_attr_getscope(&SchedAttr,&NewScope);
58     }
59  }
60
61
62  void thread_object::setDetached(void)
63  {
64     pthread_attr_setdetachstate(&SchedAttr,PTHREAD_CREATE_DETACHED);
65     pthread_attr_getdetachstate(&SchedAttr,&NewState);
66
67  }
68
69  void thread_object::setJoinable(void)
70  {
71     pthread_attr_setdetachstate(&SchedAttr,PTHREAD_CREATE_JOINABLE);
72     pthread_attr_getdetachstate(&SchedAttr,&NewState);
73  }
74
75
76  void thread_object::run(void)
77  {
78     pthread_create(&Tid,&SchedAttr,thread,this);
79  }
80
81
82  void thread_object::name(string X)
83  {
84     Name = X;
```

```
85  }
86
87
88  void *  thread (void * X)
89  {
90     thread_object *Thread;
91     Thread = static_cast<thread_object *>(X);
92     Thread->do_something();
93     return(NULL);
94  }
```

在程序清单 6-3 中，第 3 行～第 10 行定义的构造函数为 SchedAttr 这个类初始化一个线程属性对象。它将 inheritsched 属性设置为 PTHREAD_EXPLICIT_SCHED，这样使用这个属性的对象创建的线程负责定义它的调度策略和优先级，而不是从创建者线程继承调度策略和优先级。默认情况下，线程的状态是 JOINABLE。其他方法的含义一看便知：

```
setPriority(int Priority)
setSchedPolicy(int Policy)
setContentionscope(int Scope)
setDetached()
setJoinable()
```

在第 20 行调用 pthread_join( )之前，使用 join( )检查线程是不是可结合的。当在第 78 行中创建线程时，pthread_create( )使用 SchedAttr 对象：

```
pthread_create(&Tid, &SchedAttr, thread,this);
```

程序清单 6-4 显示了 filter_thread 的定义。

程序清单 6-4

```
//Listing 6-4  A definition of the filter_thread class.

1   #include "thread_object.h"
2
3
4   filter_thread::filter_thread(void)
5   {
6      pthread_attr_init(&SchedAttr);
7
8
9   }
10
11
12  filter_thread::~filter_thread(void)
13  {
14
15  }
16
17  void filter_thread::do_something(void)
```

```
18  {
19     struct sched_param Param;
20     int Policy;
21     pthread_t thread_id = pthread_self();
22     string Schedule;
23     string State;
24     string Scope;
25
26     pthread_getschedparam(thread_id,&Policy,&Param);
27     if(NewPolicy == SCHED_RR){Schedule.assign("RR");}
28     if(NewPolicy == SCHED_FIFO){Schedule.assign("FIFO");}
29     if(NewPolicy == SCHED_OTHER){Schedule.assign("OTHER");}
30     if(NewState == PTHREAD_CREATE_DETACHED){State.assign("DETACHED");}
31     if(NewState == PTHREAD_CREATE_JOINABLE){State.assign("JOINABLE");}
32     if(NewScope == PTHREAD_SCOPE_PROCESS){Scope.assign("PROCESS");}
33     if(NewScope == PTHREAD_SCOPE_SYSTEM){Scope.assign("SYSTEM");}
34     cout << Name << ":" << thread_id << endl
35          << "----------------------" << endl
36          << " priority: "<< Param.sched_priority    << endl
37          << " policy:   "<< Schedule                << endl
38          << " state:    "<< State                   << endl
39          << " scope:    "<< Scope                   << endl << endl;
40
41  }
42
```

在程序清单6-4中，第4行～第9行的filter_thread构造函数使用线程属性对象SchedAttr进行初始化。定义了do_something( )方法。在filter_thread中，这个方法将如下线程信息发送到cout：

- 线程名称
- 线程id
- 优先级
- 调度策略
- 状态
- 范围

有些值可能未被初始化，因为它们不是在属性对象中设置的。下一章将会对这个方法进行重新定义。

现在可以创建多个filter_thread对象，每个对象可以设置线程的属性。程序清单6-5显示了如何创建多个filter_thread对象。

程序清单6-5

```
//Listing 6-5 is main line to create multiple filter_thread objects.

 1  #include "thread_object.h"
 2  #include <unistd.h>
```

```
 3
 4
 5  int main(int argc,char *argv[])
 6  {
 7     filter_thread  MyThread[4];
 8
 9     MyThread[0].name("Proteus");
10     MyThread[0].setSchedPolicy(2);
11     MyThread[0].setPriority(7);
12     MyThread[0].setDetached();
13
14     MyThread[1].name("Stand Alone Complex");
15     MyThread[1].setContentionScope(1);
16     MyThread[1].setPriority(5);
17     MyThread[1].setSchedPolicy(2);
18
19     MyThread[2].name("Krell Space");
20     MyThread[2].setPriority(3);
21
22     MyThread[3].name("Cylon Space");
23     MyThread[3].setPriority(2);
24     MyThread[3].setSchedPolicy(2);
25
26     for(int N = 0;N < 4;N++)
27     {
28        MyThread[N].run();
29        MyThread[N].join();
30     }
31     return (0);
32  }
```

在程序清单 6-5 中，创建了 4 个 filter_threads。下面是程序清单 6-5 的输出：

```
Proteus:Stand Alone Complex:32

----------------------
 priority:  7
----------------------
 policy:   FIFO priority:5
 state:     policy:    DETACHEDFIFO

 scope:     state:     PROCESSJOINABLE


 scope:    SYSTEM

Krell Space:4
----------------------
 priority:  3
 policy:   OTHER
 state:    JOINABLE
```

```
 scope:     PROCESS

Cylon Space:5
----------------------
 priority:  2
 policy:    FIFO
 state:     JOINABLE
 scope:     PROCESS
```

主线程不等待分离的线程(Proteus)，输出有一些混乱。Proteus 开始进行输出，然后被来自 Stand Alone Complex 的输出打断。如前所述，标准 cout 不是线程安全的。如果所有的线程都是可结合的，那么输出将会像您所期望的那样：

```
Proteus:2
----------------------
 priority:  7
 policy:    FIFO
 state:     JOINABLE
 scope:     PROCESS

Stand Alone Complex:3
----------------------
 priority:  5
 policy:    FIFO
 state:     JOINABLE
 scope:     SYSTEM

Krell Space:4
----------------------
 priority:  3
 policy:    OTHER
 state:     JOINABLE
 scope:     PROCESS

Cylon Space:5
----------------------
 priority:  2
 policy:    FIFO
 state:     JOINABLE
 scope:     PROCESS
```

程序概要 6-2

程序名：

program6-2.cc

PDG

描述：

示范 filter_thread 类的使用。创建了 4 个线程，对每个线程进行了命名。每个线程调用更改将要创建的线程的一些属性的方法。

必需的库：

libpthread

必需的头文件：

thread_object.h

编译和链接指令：

c++ -o program6-2 program6-2.cc thread_object.cc filter_thread.cc –lpthread

测试环境：

Solaris 10、gcc 3.4.3 和 gcc 3.4.6

处理器：

AMD Opteron 和 UltraSparc T1

执行指令：

./program6-2

thread_object 类封装了线程属性对象的部分功能。filter_thread 是用户线程，它继承了 thread_object 并定义了 do_something( )。该函数由线程执行。在第 7 章中，将会再次扩展这个类的功能，以构成用作流水线模型中的一部分的 assertion 类。

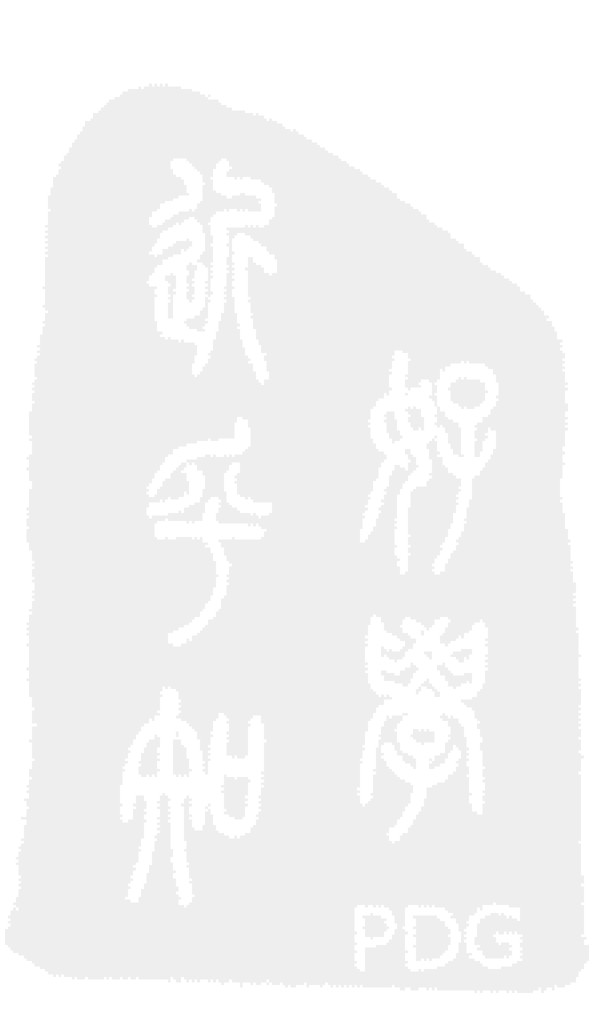
6.9 小结

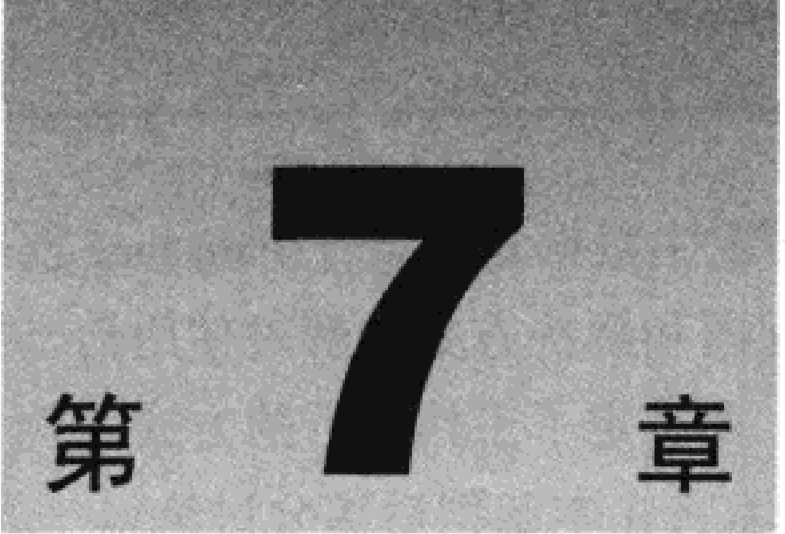
线程是进程中可执行代码序列或流，操作系统将线程调度到处理器或内核上运行。本章介绍了多线程的相关内容，通过以上内容，您应当了解的关键知识包括：

- 所有进程都有一个主线程，它是进程的控制流。有着多个线程的进程有同样数目的控制流，它们独立且并发地执行。有着多个线程的进程是多线程的。
- 内核级线程或轻量级进程同进程相比，在创建、维护、管理方面带给操作系统的负担要轻一些，因为同线程关联的信息很少。内核线程在处理器上执行，它们由系统创建和管理。用户级线程通过运行时库来创建和管理。

- 通过使用线程，可以简化程序结构、使用最少的资源对固有并发进行建模、执行程序中独立的并发任务。线程可以改进应用程序的吞吐量和性能。
- 线程和进程都有 id、寄存器组、状态和优先级，而且都遵从某种调度策略。两者均有上下文，用于重新构建被抢占的进程或线程。线程和子进程共享它们的父进程的资源并竞争对处理器的使用。父进程可以对子进程或线程具有一定的控制。线程或进程可以改变它们的属性并创建新的资源，但是不能够访问属于其他进程的资源。线程和进程之前最大的区别在于每个进程有着自己的地址空间，而线程则位于它所属于的进程的地址空间内。
- POSIX 线程库定义了线程属性对象，它封装了线程属性的一个子集。这些属性是可访问和可更改的。线程属性的类型为 pthread_attr_t。pthread_attr_init( )使用默认值来初始化线程属性对象。一旦属性被适当地更改了，则可以把属性对象用作对任何 pthread_create( )函数的调用中的参数。
- thread_object 接口类用作一个包装器，使得事物看上去同通常情况下的行为不同。设计新的接口的目的是使得类更易于使用、更具功能性、更安全或语义更正确。可以扩展 thread_object 来封装属性对象。

第 7 章将讨论进程和线程之间的通信和同步。并发任务之间可能会需要通信，以同步工作或对共享全局数据的访问。





第 7 章 并发任务的通信和同步

第 6 章讨论了进程与线程之间的类似之处和不同之处。进程与线程之间最大的区别在于每个进程拥有自己的地址空间，而线程被包含在它们所属进程的地址空间内。我们讨论了线程和进程如何拥有 id、寄存器组、状态、优先级以及如何遵从某个调度策略。我们还解释了如何创建和管理线程，并且创建了一个线程类。

本章将进一步讨论线程间和进程间的通信和协作。包括如下主题：

- 通信和协作依赖
- 进程间通信和线程间通信
- PRAM 模型和并发模型
- 指定执行模型
- 面向对象消息队列和互斥量
- 用于流水线的简单的 agent 模型

7.1 通信和同步

第 3 章讨论了协调并发任务的执行带来的挑战。使用的例子是在客人假期到达之前使用软件自动油漆机来为房间刷漆。在分解问题和解决方案时，列出了很多问题，目的是确定为房屋刷漆的哪种方法是最好的。其中一些问题与通信和同步资源的使用相关。

- 油漆机之间是否相互通信？
- 油漆机应当在完成任务时还是在要求某些资源(如油刷或油漆)时进行通信？
- 油漆机应当直接相互通信？还是应当有一个中央油漆机，所有的通信都经由它？
- 只能同一个房间中的油漆机通信会更好些吗？还是不同房间的所有油漆机都能进行通信更好些？
- 就共享资源而言，多台油漆机是否能够共享一个资源？还是对资源的使用必须串行化？

这些问题关注的是对这些并发任务之间的通信和同步进行协调。如果相关任务之间的

PDG

通信未经过适当的设计，则会发生数据竞争条件(data race condition)。对任务之间的通信和同步进行适当的协调要求在问题和解决方案分解时使用恰当的并发模型。并发模型指示了通信何时发生以及如何发生，还决定了工作执行的方式。例如，对于软件自动油漆机的例子，可以使用 boss-worker 模型(将在本章稍后部分讨论)。一台 boss 油漆机可以委托工作或指挥油漆机在某个特定时间对哪个房间进行刷漆。boss 油漆机还可以管理资源的使用。所有油漆机与 boss 通信，告诉 boss 它为了完成任务需要哪些资源，然后由 boss 决定何时将资源授予油漆机。可以使用依赖关系(dependency relationship)检查哪些任务依赖于同其他任务进行通信或协作。

7.1.1 依赖关系

当进程或线程相互之间要求通信或协作以完成一个共同的目标时，它们有着依赖关系。Task A 依赖于 Task B 来提供计算的值、给出待处理的文件名、释放某个资源。Task A 可能依赖于 Task B，但是 Task B 未必就会依赖于 Task A。对于任意给定的两个任务，它们之间能够存在 4 种依赖关系：

- A→B：Task A 依赖于 Task B。
- A←B：Task B 依赖于 Task A。
- A↔B：Task A 依赖于 Task B，且 Task B 依赖于 Task A。
- A NULL B：在 Task A 和 Task B 之间不存在依赖。

上述第一种和第二种情况是单向依赖。第三种情况是一种双向依赖，A 和 B 相互依赖。在第四种情况下，Task A 和 Task B 之间为 NULL 依赖，即不存在依赖。

1. 通信依赖

当 Task A 要求来自 Task B 的数据才能够执行它的工作时，Task A 和 Task B 之间就存在依赖关系。可以将一台软件自动油漆机设计为在 boss 油漆机的命令下，给所有用完油漆的漆桶填满油漆。这将意味着所有油漆机(实际涂漆的)都必须同 boss 油漆机进行通信，告诉 boss 油漆机它们的油漆用完了。然后 boss 油漆机会通知 refill(再装满)油漆机，告诉它有漆桶需要添加油漆。这意味 worker 油漆机同 boss 油漆机存在通信依赖，refill 油漆机也同 boss 油漆机存在通信依赖。

在第 5 章中，我们使用 posix_queue 对象来在进程间进行通信。posix_queue 是 POSIX 消息队列的接口，该消息队列是字符串链表。posix_queue 对象包含了 worker 进程将要搜索编码的文件的名字。worker 进程从 posix_queue 中读取文件名。posix_queue 是驻留在所有进程的地址空间以外的数据结构。另一方面，线程也能够通过使用全局变量和数据结构来和它所属进程的地址空间内的其他线程进行通信。如果两个线程之间希望传递数据，线程 A 将会把文件名写到全局变量，然后线程 B 只需要读取该变量即可。这些是单向通信依赖的例子，其中只有一个任务依赖于另一个任务。图 7-1 显示了两个单向通信依赖的例子：进程使用的 posix_queue 和线程 A 和 B 用来保存文件名的全局变量。

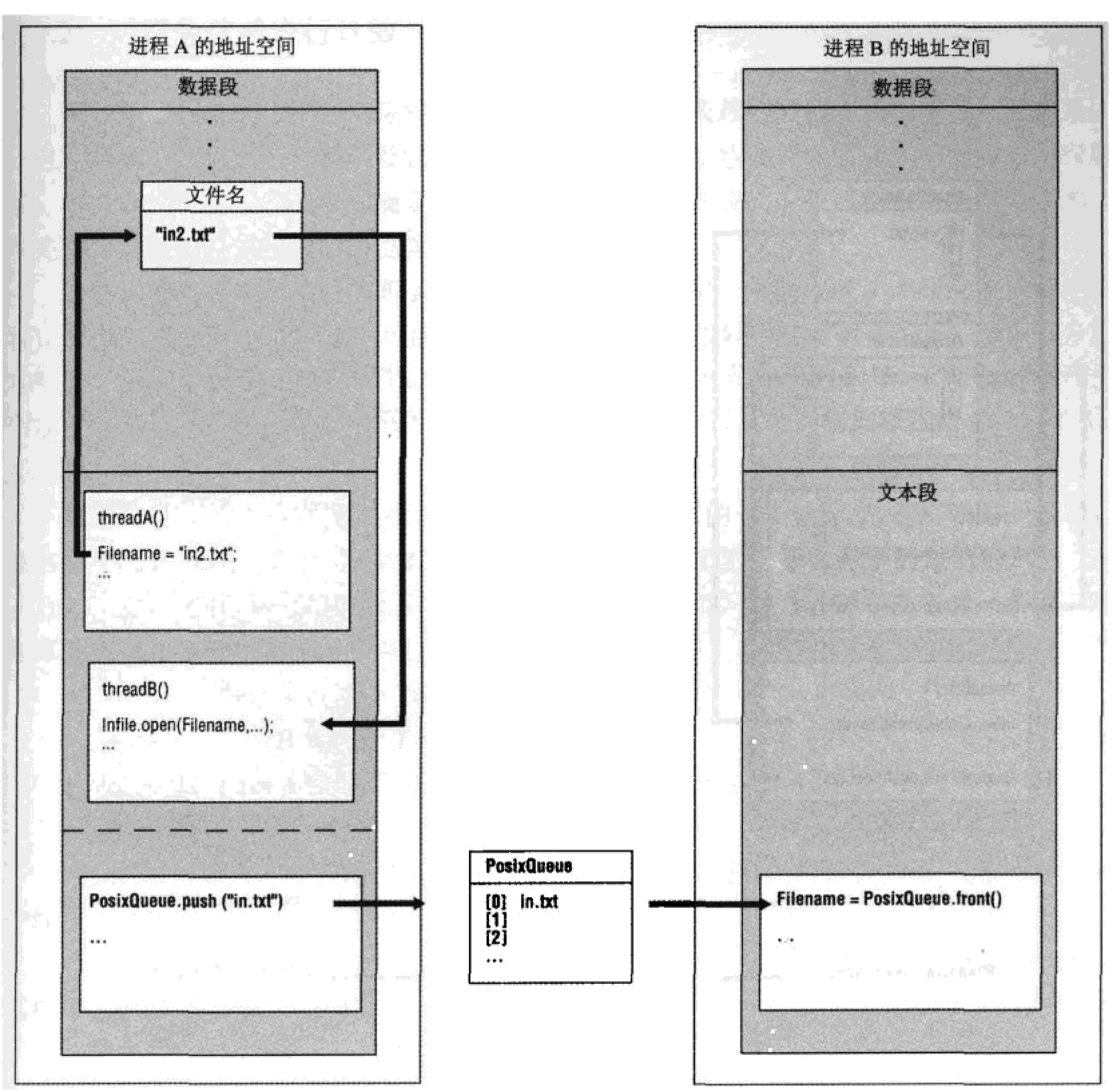


图 7-1

双向依赖的例子是两个先进先出(FIFO)管道。管道是用来构成两个进程之间的通信通道的数据结构。进程 A 使用管道 1 的输入端来发送进程 B 必须处理的文件名。进程 B 从管道 1 的输出端读取文件名。当它处理文件内容之后，将结果写入到一个新的文件。新文件的名字会写到管道 2 的输入端。进程 A 会从管道 2 的输出端读取文件名。这是双向通信依赖。进程 B 依赖于进程 A 来了解文件的名称，进程 A 依赖于进程 B 来了解新文件的名称。线程 A 和线程 B 可以使用两个全局数据结构(例如队列)，一个用来包含源文件名，另一个用来包含结果文件名。图 7-2 显示了两个双向通信依赖的例子，分别是进程 A 和 B 之间的依赖和线程 A 和 B 之间的依赖。

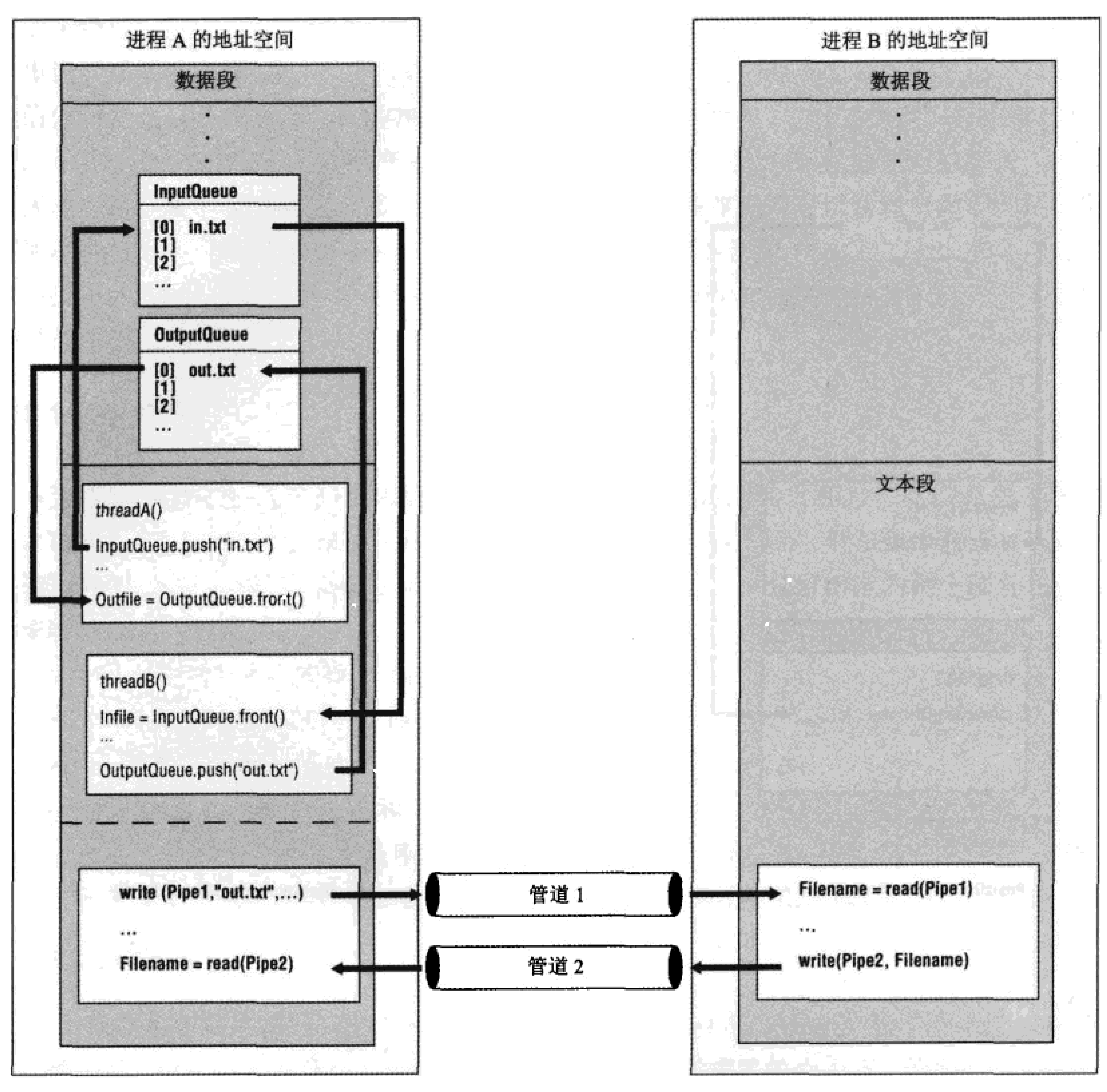


图 7-2

2. 协作依赖

当 Task A 要求的资源被 Task B 所拥有，而且在 Task A 能够使用该资源之前，Task B 必须释放它，则存在协作依赖(cooperation dependency)。当两个任务并发执行，而且都试图利用相同的资源时，必须进行协作，然后才能够成功使用该资源。假定在一个房间内有多台自动油漆机，而且它们共享一个漆桶。它们同时都试图访问漆桶。考虑到漆桶不能够被油漆机同时访问(这样做不是线程安全的)，必须对访问进行同步，这就要求协作。

协作依赖的另一个例子是对 posix_queue 的写访问。如果多个进程都要将编码所位于的文件的名称写到 posix_queue，这将要求每次只有一个进程能够写入到 posix_queue。写入访问将必须被同步。

7.1.2　对任务依赖进行计数

可以通过列举应用程序中进程或线程间存在的可能依赖的数目来理解总体任务关系。一旦已经列举了可能的依赖以及它们的关系，可以确定哪些线程必须为通信和同步进行编码。这类似于应用程序中用来确定可能的决策分支的真值表。一旦线程间的依赖关系都列举完毕，进程的总体线程结构也就完成了。

例如，如果有 3 个线程，即 A、B 和 C(3 个线程来自一个进程或 3 个线程来自 3 个进程)，您可以检查这些线程之间可能存在的依赖。如果在依赖中涉及两个线程，使用组合来计算 3 个线程中可能涉及依赖中的线程：$C(n,k)$，其中 n 是线程的数目，k 是涉及依赖中的线程的数目。因此，对于本例为 $C(3,2)$，结果为 3，即有 3 种可能的线程组合：A 和 B、A 和 C、B 和 C。

如果将每种组合考虑成图(有两个节点以及它们之间的一条边)，它是简单图，即不存在自身环(self-loop)和并行边(即没有哪两条边有相同的端点)，那么图中的边的数目为 $n(n-1)/2$。这样，对于两个节点的简单图，边数为 2(2－1)/2，即 1。每个图包含一条边。现在每条边具有前面讨论的 4 种可能的依赖关系中的 1 种：

- A→B：Task A 依赖于 Task B。
- A←B：Task B 依赖于 Task A。
- A↔B：Task A 依赖于 Task B，且 Task B 依赖于 Task A。
- A NULL B：在 Task A 和 Task B 之间不存在依赖。

因此，每个图有 4 种可能的关系。如果您对 3 个线程间可能的依赖关系(关系中涉及两个线程)计数，那么一共有 12 种可能的关系。

可以使用邻接矩阵来列举两线程组合的实际依赖关系。邻接矩阵是一个图 $G=(V,E)$，其中 V 是图中顶点或节点的集合，而 E 是边的集合，这样：

$A(i,j)=1$ 如果(i,j)是 E 的一个元素

$\quad\quad=0$ 其他情况

$A(i,j) <> A(j,i)$

其中 i 表示行，j 表示列。矩阵的规模为 $n \times n$，其中 n 是线程的总数。图 7-3(a)显示了 3 个线程的邻接矩阵。0 代表不存在依赖，1 代表存在依赖。邻接矩阵可用来标定任意两个线程之间的所有依赖关系。在对角线上，值均为 0，因为不存在自身依赖。

- $A(1,2)=1$　意味着对于 A→B，A 依赖于 B。
- $A(1,3)=0$　意味着对于 A→C，A 不依赖于 C。
- $A(2,1)=0$　意味着对于 B→A，B 不依赖于 A。
- $A(2,3)=1$　意味着对于 B→C，B 依赖于 C。
- $A(3,1)=1$　意味着对于 C→A，C 依赖于 A。
- $A(3,2)=0$　意味着对于 C→B，C 不依赖于 B。

	A	B	C
A	0	1	0
B	0	0	1
C	1	0	0

(a) 邻接矩阵

	A	B	C
A		S,Co	
B			S,C
C	S,C		

(b) 依赖矩阵

图 7-3

依赖图在记录依赖关系的类型方面很有用，例如，*C* 代表通信，C_0代表协作。*S* 代表同步(如果通信依赖或协作依赖要求同步)。可以在软件开发生命周期(SDLC)的设计或测试阶段使用依赖图。为了构建依赖图，需要使用邻接矩阵。对于矩阵中为 1 的行列位置，用关系的类型来替换掉数字 1。图 7-3(b)显示了 3 个线程的依赖图。矩阵中的 0 和 1 被 *C* 或 C_0替代。如果某个位置上的值为 0，意味着不存在依赖关系，该位置将为空白。对于 A(1,2)，因为同步的协作，A 依赖于 B；对于 A(2,3)，因为同步的通信，B 依赖于 C；对于 A(3,1)，因为同步的通信，C 依赖于 A。双向关系如 A↔B 也可以表示，但是本例中没有这样的关系。因此，如您所见，所有的关系都能够在矩阵中表示。对于 NULL 关系，在邻接矩阵中使用 0 来表示，在依赖矩阵中对应的位置会为空白。图 7-4 显示了这 3 个线程的 UML(统一建模语言)依赖。

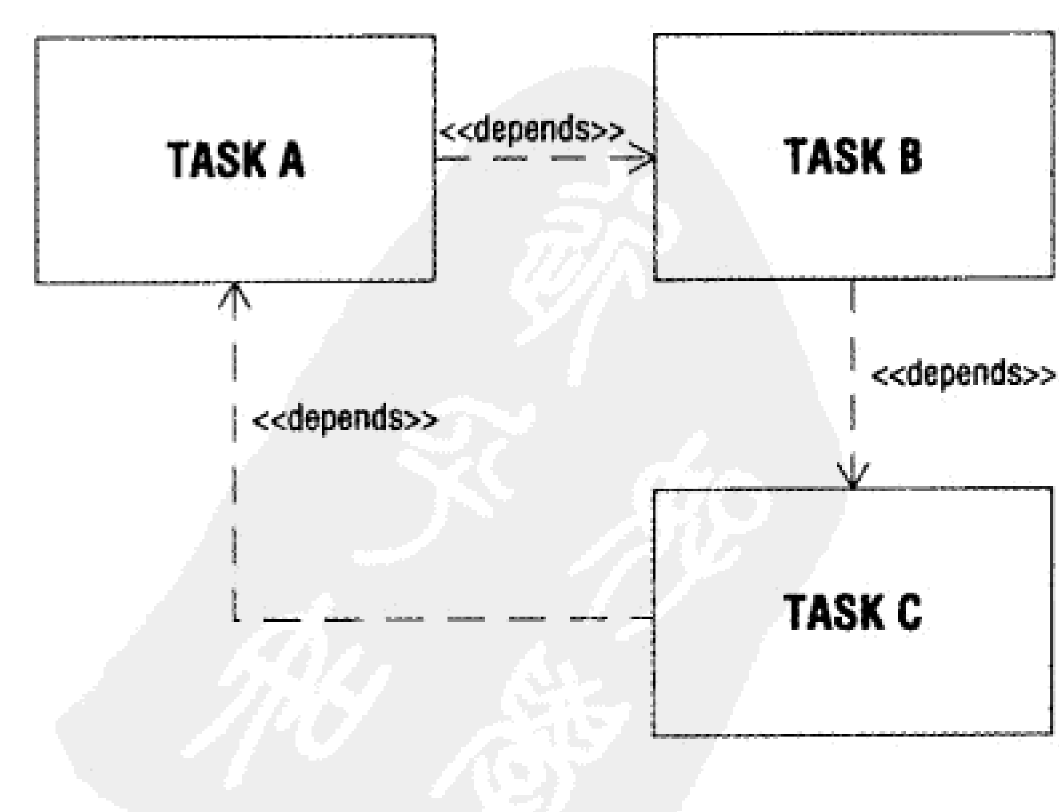


图 7-4

这些工具和方法是非常有用的。了解可能的关系的数目并确定它们是怎样的关系，能够在建立进程和应用程序的总体线程结构方面提供帮助。我们已经将它们用在较少数量的线程上。依赖中只涉及两个线程时，才能够使用矩阵方法。对于较大数目的线程，不能够使用矩阵方法(除非是多维矩阵)。但是即使对于中等数目的线程，列举每种关系将是不实用的。这就是声明式方法非常有用的原因。

7.1.3　什么是进程间通信

进程有着自己的地址空间。一个进程中声明的数据，在另一个进程中是不可用的。一个进程中发生的事件也不被另一个进程所了解。如果进程 A 和进程 B 共同完成某个任务，例如从文件中筛选掉特殊字符，或在文件中搜索某个编码，必须有一定的方法来在两个进程之间通信并协调事件。第 5 章描述了进程的布局。进程有代码段、数据段和栈段。进程还可能在堆中分配有其他内存。进程拥有的数据通常位于栈段或数据段，或者是从内存中动态分配的，这些内存是受保护的，不会被其他进程访问。如果一个进程要想了解另一个进程的数据或事件，需要使用操作系统 API 来创建某种通信手段。当进程向另一个进程发送数据，或者通过操作系统 API 使另一个进程知道某个事件，这被称为 IPC(进程间通信)。IPC 涉及推动进程之间的通信的技术和机制。操作系统的内核被用作进程之间的通信通道。posix_queue 是 IPC 的一个实例。在相关或不相关的进程之间通信时，还可以使用文件。

进程驻留在用户模式或用户空间。IPC 机制可以驻留在内核空间或用户空间。用于通信的文件驻留在用户空间和内核空间外部的文件系统中。进程通过利用文件来共享信息时，必须使用系统调用来经过内核，例如 read、write、lseek，或者通过使用 iostreams。当文件同时被两个进程更新时，需要某种类型的同步。进程间的共享信息驻留在内核空间。访问共享信息的操作将会涉及对内核的系统调用。一种不驻留在用户空间的 IPC 机制是共享内存。共享内存是每个进程都可以引用的内存区域。通过使用共享内存，进程可以在不进行内核调用的前提下，访问共享区域中的数据。这也要求同步。

1. IPC 的持久性

对象的持久性是指对象在创建它的程序、进程或线程执行期间或执行期间之外的超出该期间的存在性。存储类别指定了程序执行期间对象会存在多久。对象的存储类别可以被声明为 automatic、static 或 dynamic。

- automatic 对象在代码块被调用期间存在。对象的空间和值只在块内存在。当控制流离开该代码块时，对象不再存在，若再引用则会产生错误。
- static 对象在程序的执行期间一直存在且保留它的值。
- dynamically(动态)分配的对象，其生命期不超过 static 对象，但是可以短于 automatic 对象。程序员在运行时决定何时动态声明一个对象，而且该对象将会在程序的整个执行期间存在。

对象的持久性不一定和对象在存储设备上的存储一致。例如，automatic 或 static 对象

可能被保存在程序执行期间用作虚拟内存的外部存储中，但是程序结束后，对象将会被销毁。

IPC 实体驻留在文件系统、内核空间或用户空间，持久性也采用这种方式定义：文件系统持久性、内核持久性和进程持久性。

- 有着文件系统持久性的 IPC 对象会一直存在，直到显式删除该对象。如果内核重新启动，对象将保持它的值。
- 内核持久性定义的 IPC 对象会一直存在，直到内核重新启动或对象被显式删除。
- 有着进程持久性的 IPC 对象一直存在，直到创建它的进程关闭它。

IPC 有多种类型，如表 7-1 所列出的那样。多数 IPC 作用于相关的进程上，即子进程和父进程。对于那些不相关且要求进程间通信的进程，IPC 对象有着同它关联的名字，这样创建它的 server 进程同 client 进程可以使用相同的对象。管道是不被命名的，因此它们只能够用于相关的进程。FIFO 或命名管道可用于不相关的进程。文件系统中的路径名用作 FIFO IPC 机制的标识符。对指定类型的 IPC 机制，为所有可能的名称设置了一个名称空间。对于要求 POSIX IPC 名称的 IPC，该名称必须以斜线开始，并且不包含其他斜线。为了创建 IPC，必须对目录具有写权限。

表 7-1

IPC 的类型	名称空间	持久性	进程
管道	未命名	进程	相关
FIFO	路径名	进程	双方
互斥量	未命名	进程	相关
条件变量	未命名	进程	相关
读写锁	未命名	进程	相关
消息队列	Posix IPC 名称	内核	双方
信号量(基于内存)	未命名	进程	相关
信号量(命名的)	Posix IPC 名称	内核	双方
共享内存	Posix IPC 名称	内核	双方

表 7-1 还显示出每种类型的 IPC(FIFO 和管道)均有进程持久性。消息队列和共享内存必须有内核持久性，但也可以使用文件系统持久性。当消息队列和共享内存利用文件系统持久性时，它们是通过使用从文件到内存的映射来实现的。这被称作映射文件(mapped file)或内存映射文件(memory mapped file)。一旦文件被映射到在进程之间共享的内存，文件的内容会使用内存位置来被更改和读取。

2. 环境变量和命令行参数

父进程同子进程共享它们的资源。通过使用 posix_spawn 或 exec 函数，父进程可以创

建有着它的环境变量的精确副本的子进程，也可以使用新的值来对子进程进行初始化。环境变量保存系统相关信息，例如包含进程使用的命令、库、函数和过程的目录的路径。它们可用于在父子进程之间传递有用的用户定义信息。它们提供了一种机制来指定信息传递将到相关的进程，同时不需要将它写死到程序代码中。系统环境变量对所有 shell 和系统中的进程都是共同的，且是预先定义好的。变量通过启动文件(startup file)进行初始化。

注意：
第 5 章中列出了常见的环境变量。

还可以将环境变量和命令行参数传递给新近初始化的进程。

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict], char *const envp[restrict]);
```

argv[ ]和 envp[ ]用来将命令行参数列表和环境变量传递给新进程。这是单向、一次性的通信，一旦创建了子进程，子进程对这些变量的任何改变都不会反映到父进程中的数据，而且父进程不能够对子进程已经收到的这些变量进行任何改动。

3. 文件

使用文件在进程间传递数据是传输或共享数据的最简单且最灵活的方式之一。可以使用文件在相关或不相关的进程之间传送数据。文件可以使得原本并未被设计为共同工作的进程共同工作。当然，文件具有文件系统持久性，这样，文件的持久性不受系统重新启动的影响。

当您使用文件来在进程间通信时，需要在文件传送过程中遵循以下 7 个基本步骤：

(1) 必须沟通文件的名称。
(2) 必须验证文件是否存在。
(3) 确保得到了访问文件的正确权限。
(4) 打开文件。
(5) 同步对文件的访问。
(6) 在读/写文件时，检查流是否正常以及是否位于文件的结尾处。
(7) 关闭文件。

首先，必须在进程间沟通文件的名称。您可能会回想起在第 4 章和第 5 章中，使用了文件来保存必须被 worker 处理的工作。每个文件包含了超过百万的字符串。posix_queue 包含文件名。文件名还可以通过其他 IPC 方式传递给子进程，例如管道。

当进程访问文件时，如果其他进程也可以访问这个文件，就需要同步。回顾第 4 章的程序清单 4-2，文件被分割为多个较小的文件，每个进程对文件进行独占访问。然而，如果只有一个文件，则对文件的访问必须进行同步。每个进程在某时应当有着互斥读的能力，并将从文件中读取一个字符串并推进读指针。本章稍后将讨论读/写锁以及其他类型的同步。

将文件留在打开状态可能会导致数据损坏，而且会阻止其他进程对文件的访问。对文件进行读写的进程应当知道文件的格式，这样才能正确地处理文件。文件的格式是指文件类型和文件的组织。文件的类型还隐含了文件中数据的类型。它是文本文件还是二进制文件？进程还应当知道文件布局或数据在文件中如何组织。

4. 文件描述符

文件描述符是进程用来标识一个打开文件的无符号整数。它们在父进程和子进程之间共享。文件描述符是在文件描述符表中的索引，该表是内核为每个进程维护的块。当创建一个子进程时，会为该子进程副本描述符表，使得子进程与父进程有着相同的文件访问。能够为进程分配的文件描述符的数目受到资源限制的支配。这个限制可以通过 setrlimit( )来改变。文件描述符是由 open( )返回。文件描述符经常被其他 IPC 使用。

5. 共享内存

共享内存块可用来在进程间传递信息。内存块不属于共享该内存的任何进程。每个进程有着自己的地址空间，内存块同进程的地址空间是分隔的。进程通过临时性地将共享内存块同它自己的内存块连接起来，获得对共享内存的访问。一旦附加了内存块，它可以像任何其他指向内存块的指针那样使用。同其他数据传送机制类似，共享内存也被设置上适当的访问权限。它几乎可以像文件那样灵活地传送数据。如果进程 A、B、C 在使用共享内存块，任何进程做出的任何更改可以被其他所有进程看到。这不是一种单向、一次性的通信机制。

管道在被使用之前，要求至少有两个进程连接到管道上。共享内存可以被一个进程读写，并且为该进程保持打开。其他进程可以根据需要附加到共享内存或者同共享内存分开。这使得大块数据的传递要比使用管道和 FIFO 快很多。然而，考虑为共享区域分配多少内存非常重要。

当您访问保存在共享内存中的数据时，要求进行同步。类似于当多个进程试图同时对同一个文件进行读/写访问时必须进行文件加锁一样，对共享内存的访问也必须进行控制。控制对内存访问的标准技术是信号量。之所以必须进行访问控制，是因为当两个进程试图同时更新相同的内存时，会发生数据竞争。

6. 使用 POSIX 共享内存

共享内存映射：

- 文件
- 内部内存

到共享内存区域：

调用形式

```
#include <sys/mman.h>

void  *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t
offset);
int mumap(void *addr, size_t len);
```

该函数将文件描述符 fd 指定的文件或对象中，从偏移量为 offset 开始的 len 个字节映射到内存中，最好是从地址 addr 开始，addr 通常被指定为 0。该函数返回对象被映射到的实际位置，而且永远不会为 0，它的类型是 void *。prot 描述了期望的内存保护，它必须同文件的打开模式不发生冲突。flags 指定了被映射对象的类型，它还可以指定映射选项以及对映射过来的页面的拷贝所做的更改为进程所私有，或者是同其他引用共享。表 7-2 显示了 prot 和 flags 的可能值以及简要的描述。为了从进程的地址空间中删除内存映射，我们使用了 mumap( )。

表 7-2

mmap 的 flag 参数	描　述
prot	描述基于内存的区域的保护
PROT_READ	数据可以被读取
PROT_WRITE	数据可以被写入
PROT_EXEC	数据可以被执行
PROT_NONE	数据是不可访问的
flags	描述数据可以如何被使用
MAP_SHARED	改动被共享
MAP_PRIVATE	改动是私有的
MAP_FIXED	addr 被正确地解释

为了创建共享内存，打开一个文件并保存文件描述符，然后使用适当的参数调用 mmap( )并保存返回的 void *。当访问变量时使用互斥量。根据您将要处理的数据，可能必须要对 void *进行类型转换：

```
fd = open(file_name,O_RDWR);
ptr = casting<type>(mmap(NULL,sizeof(type),PROT_READ,MAP_SHARED,fd,0));
```

这是一个通过文件来进行内存映射的例子。当通过内部内存来使用共享内存时，使用创建共享内存的函数，而不是打开一个文件的函数：

调用形式

```
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

函数 shm_open( )创建并打开一个新的 POSIX 共享内存对象，或者打开一个已有的 POSIX 共享内存对象。该函数同 open( )非常类似。name 指定了创建和/或打开的共享内存对象。为了确保 name 的可移植性，使用斜线作为开头，并且不要使用内嵌斜线。oflag 是由 ORing 同特定标志之一(O_RDONLY 或 O_RDWR 以及表 7-3 中列出的所有其他标志)共同创建的位掩码，表 7-3 中还列出了 mode 的可能的值以及简要的描述。shm_open( )返回一个新的文件描述符，它指向共享内存对象。文件描述符用于 mmap( )函数调用。

```
fd = sh_open(memory_name,O_RDWR, MODE);
ptr = casting<type>(mmap(NULL,sizeof(type),PROT_READ,MAP_SHARED,fd,0));
```

现在，可以像所有其他数据指针那样使用 ptr。要确保在进程间使用信号量：

```
sem_wait(sem);
... *ptr;
sem_post(sem);
```

表 7-3

共享内存参数	描　　述
oflag	描述共享内存如何被打开
O_RDWR	为读访问或写访问打开对象
O_RDONLY	为只读访问打开对象
O_CREAT	如果共享对象不存在则创建它
O_EXCL	检测对象的存在性和创建；如果指定了 O_CREAT，而且对象以指定的名字存在，则返回错误
O_TRUNC	如果共享内存对象存在，则将它截取到零字节
mode	指定许可
S_IRUSR	用户有读许可
S_IWUSR	用户有写许可
S_IRGRP	组有读许可
S_IWGRP	组有写许可
S_IROTH	其他有读许可
S_IWOTH	其他有写许可

7. 管道

管道是用来在进程之间传送数据的通信通道。尽管使用文件进行的数据传送通常不要求数据的发送和接收同时为活动的，但是使用管道的数据传送要求进程同时为活动的。尽

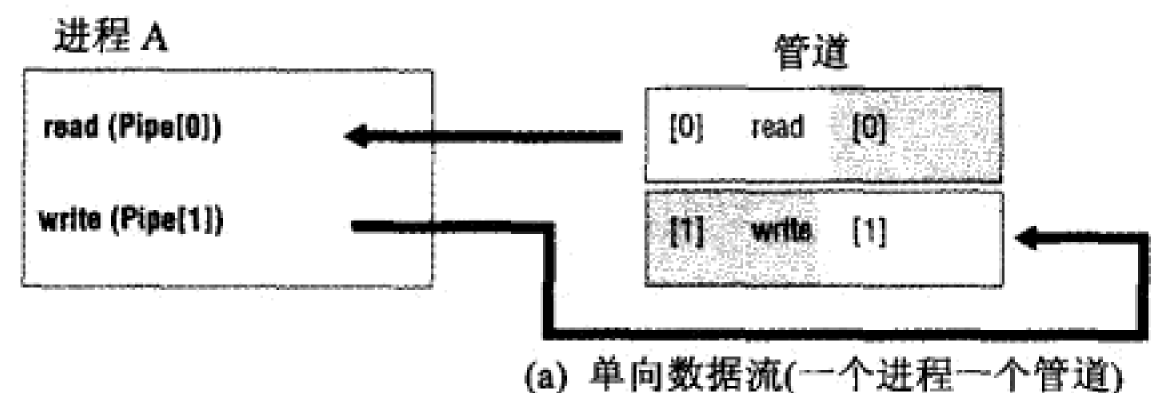
管有一些例外，但是一般的规则是管道用于两个或多个活动的进程之间。一个进程(写入方)打开或创建管道，然后阻塞，直到另一个进程(读取方)打开相同的管道进行读或写。

有两种类型的管道：

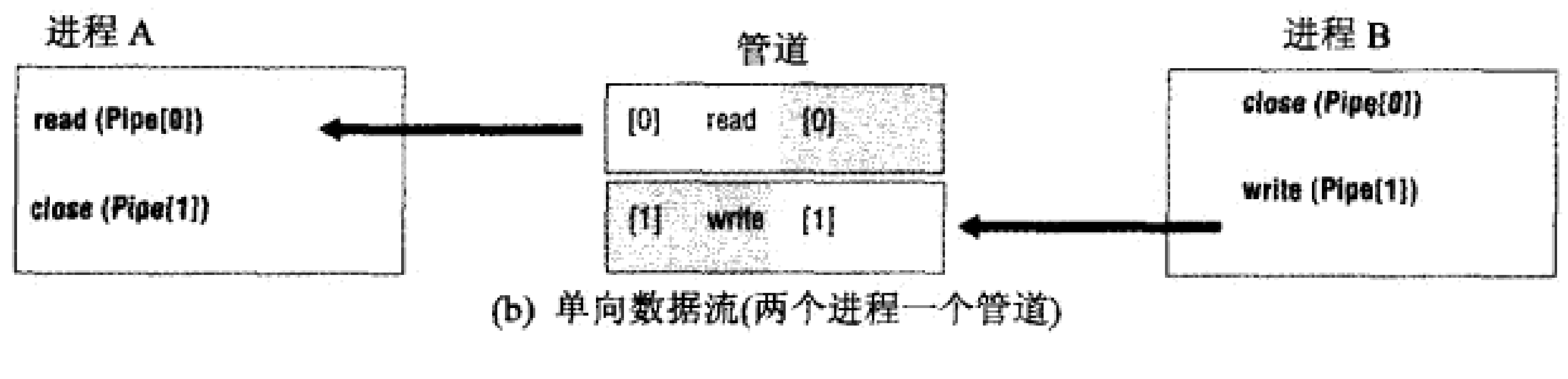
- 匿名管道
- 命名管道(也被称作 FIFO)

匿名管道用于在相关进程(父子进程)之间传送数据。命名管道用于相关或不相关进程之间的通信。使用 fork( )创建的相关进程可以使用匿名管道。使用 posix_spawn( )创建的进程使用命名管道。不相关进程是程序分别创建的，不相关进程可能在逻辑上相关，并且共同执行某些任务，但是它们仍然是不相关的。相关进程或不相关进程对命名管道的使用是通过与管道相关联的名字来引用的。命名管道是内核对象，因此它们驻留在内核空间，有着内核持久性，但是文件结构有着文件系统持久性，直到它被显式地从文件系统中删除。

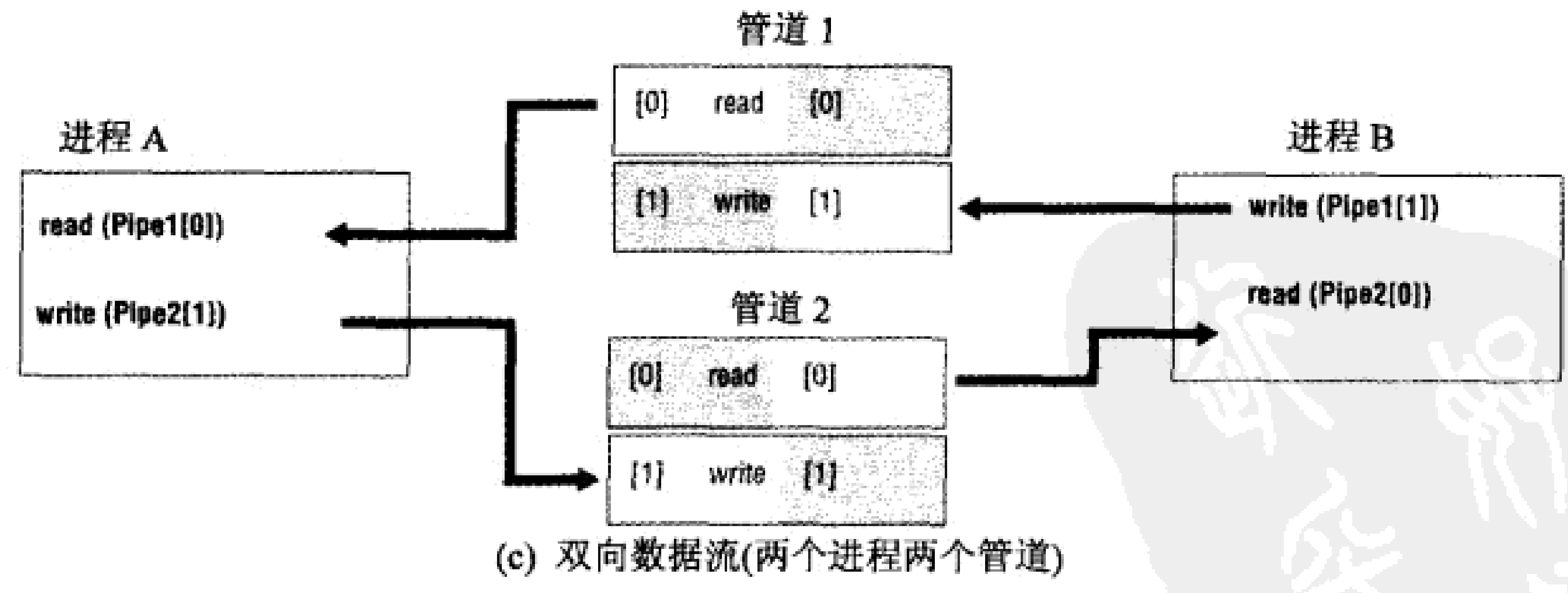
管道是由一个进程创建的，但是它们很少用在一个进程中。管道是到另一个相关或不相关进程的通信通道。管道创建了从一个进程中的一端(输入端)到另一个进程中的另一端(输出端)的数据流。数据成为了字节流，通过管道沿一个方向流动。可以通过使用两个管道来在进程间创建双向通信流。图 7-5 显示了管道的各种使用，从一个进程，到使用一个管道进行从进程 A 到进程 B 的单向数据流动的两个进程，再到使用两个管道进行双向数据流动的两个进程。



(a) 单向数据流(一个进程一个管道)



(b) 单向数据流(两个进程一个管道)



(c) 双向数据流(两个进程两个管道)

图 7-5

创建双向数据流需要两个管道的原因是由于管道设置的方式。每个管道有两端，每个管道有单向数据流。因此，一个进程将管道用作输入端(将数据写入到管道)，另一个进程使用相同的管道，但是使用输出端(从管道读取数据)。每个进程关闭了它所不访问的端，如图 7-5 所示。

匿名管道是暂时性的，只在创建它们的进程没有终止之前存在。命名管道是特殊类型的文件，并且存在于文件系统中。当创建它的进程终止后，命名管道仍可保持，除非进程显式地将它们从文件系统中删除。创建命名管道的程序可以结束执行并将命名管道留在文件系统中，但是放置到管道中的数据将不会再出现。将来的程序和进程可以访问命名管道，将新的数据写入到管道中。通过这种方式，命名管道可以作为一种永久性的通信通道来建立。命名管道有着同它们相关联的文件许可设置，而匿名管道则没有。

使用命名管道(FIFO)

命名管道是通过 mkfifo( )创建的：

调用形式

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
int unlink(const char *pathname);
```

mkfifo( )使用 pathname 作为 FIFO 的名字来创建一个命名管道，它有着由 mode 指定的许可。mode 由文件许可位组成，它们已经在表 7-3 中列出。

mkfifo( )是使用 O_CREAT | O_EXCL 标志创建的，意味着如果不存在则使用指定的名字创建一个新的命名管道。如果它已经存在，则返回 EEXIST 错误。这样，如果您希望打开一个已经存在的命名管道，调用这个函数，然后检测是否发生这个错误。如果发生错误，则使用 open( )来替代 mkfifo( )。

unlink( )将文件名 pathname 从文件系统中删除。程序清单 7-1 中的程序使用 mkfifo 创建一个命名管道。

程序清单 7-1 和程序清单 7-2 是示范了如何使用命名管道将数据从一个进程传送到另一个不相关进程的程序清单。程序清单 7-1 包含了写入方的程序，程序清单 7-2 则包含了读取方的程序。

程序清单 7-1

```
// Listing 7-1 A program that creates a named pipe with mkfifo().

 1  using namespace std;
 2  #include <iostream>
 3  #include <fstream>
```

```
 4  #include <sys/wait.h>
 5  #include <sys/types.h>
 6  #include <sys/stat.h>
 7
 8  int main(int argc,char *argv[],char *envp[])
 9  {
10
11     fstream Pipe;
12
13     if(mkfifo("Channel-one",S_IRUSR | S_IWUSR
14                              | S_IRGRP
15                              | S_IWGRP) == -1){
16        cerr << "could not make fifo" << endl;
17     }
18
19     Pipe.open("Channel-one",ios::out);
20     if(Pipe.bad()){
21        cerr << "could not open fifo" << endl;
22     }
23     else{
24           Pipe << "2 3 4 5 6 7 " << endl;
25
26     }
27
28     return(0);
29  }
```

程序清单 7-1 中的程序使用 mkfifo( )系统调用创建了一个命名管道。然后程序在第 19 行使用一个名为 Pipe 的 fstream 对象打开管道。注意管道使用 iso::out 标志来为输出打开。如果 Pipe 在调用 open 之后不是处于 bad( )状态，那么 Pipe 已经准备好用于写数据到 Channel-one 了。尽管 Pipe 为输入做好了准备，但是它会阻塞(等待)，直到另一个进程已经为了读取而打开 Channel-one。当通过管道使用 iostreams 时，重要的是无论写入方或读取方都必须是使用 ios::in | ios::out 来为输入和输出而打开的。通过这种方式打开读取方或写入方将会防止死锁。在本例中，我们将读取方为输入和输出打开(程序清单 7-2)。这个程序清单中的程序被称作读取方(reader)，是因为它从管道中读取信息。写入方可以将一行输入写入到 Pipe。

程序清单 7-2

```
// Listing 7-2 A program that reads from a named pipe.

 1  using namespace std;
 2  #include <iostream>
 3  #include <fstream>
 4  #include <string>
 5  #include <sys/wait.h>
 6  #include <sys/types.h>
```

```
 7 #include <sys/stat.h>
 8
 9
10 int main(int argc, char *argv[])
11 {
12    int type;
13    fstream NamedPipe;
14    string Input;
15
16    NamedPipe.open("Channel-one",ios::in | ios::out);
17
18    if(NamedPipe.bad()){
19       cerr << "could not open Channel-one" << endl;
20    }
21
22    while(!NamedPipe.eof() && NamedPipe.good()){
23
24       getline(NamedPipe,Input);
25       cout << Input << endl;
26    }
27    NamedPipe.close();
28    unlink("Channel-one");
29    return(0);
30
31 }
```

程序清单 7-1 中的程序使用<<操作符来将数据写入到管道中。在第 16 行，读取方也必须使用一个文件流 open、命名管道的名字 Channel-one、为输入和输出打开管道。如果 NamedPipe 在打开之后不是处于 bad 状态，那么从 NamedPipe 中读取数据，只要 NamedPipe 不是 eof( )而且仍然处于 good 状态。从管道中读取数据，并将数据保存到字符串 Input，该字符串被发送到 cout。接下来关闭 NamedPipe，然后管道会被解除链接。这些是不相关的进程。为了运行，必须单独地启动每个进程。

下面是程序清单 7-1 和程序清单 7-2 的程序概要 7-1。

程序概要 7-1

程序名：

program7-1.cc (程序清单 7-1)
program7-2.cc (程序清单 7-2)

描述：

程序清单 7-1 创建了一个管道并使用 fstream 对象打开该管道。在将一个字符串写入到管道之前，它一直等待，直到另一个进程(见程序清单 7-2)为了读取而打开管道。一旦读取进程为了读取而打开管道，写入程序将一个字符串写入到管道中、关闭管道、退出。读取

进程从管道中读取字符串并将字符串显示到标准输出中。运行读取方，然后是写入方。

必需的库：

无

必需的头文件：

< iostream > < fstream > < string > < sys/stat.h >

编译和链接指令：

c++ -o program7-1 program7-1.cc program7-2.cc

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

./program7-1
./program7-2

注释：

在单独的终端上运行每个程序。

fstream 的使用简化了 IPC，使得命名管道更易于访问。流的所有功能都在这个例子中有所体现：

```
mkfifo("Channel-one",...);
vector<int> X( 2,3,4,5,6,7);
ofstream OPipe("Channel-one",ios::out);
ostream_iterator<int> Optr(OPipe,"\n");
copy(X.begin(),X.end(),Optr);
```

这里使用了一个 vector(向量)来持有所有的数据。这一次使用一个 ofstream 对象替代了 fstream 对象来打开命名管道。声明了 ostream 迭代器并指向了命名 Channel-one。现在不需要在循环中连续地进行插入，copy 算法能够将所有的数据复制到管道。如果有成百上千个数字需要写入到管道中，这种方式非常方便。

FIFO 接口类

除了可以使用 io 流(iostreams)、迭代器(iterator)、算法来简化 IPC 机制的使用之外，还

可以通过将 FIFO 封装为 FIFO 接口类来简化 IPC 机制的使用。记住在这样做时，是在对 FIFO 结构进行建模。FIFO 类是为了在两个或多个进程之间进行通信的模型，它在这些进程之间传递某种形式的信息。这些信息被转换成数据序列、插入到管道、被管道另一端的进程提取，之后接收进程会将数据重新组合。在将数据从进程 A 传送到进程 B 时，必须有一些位置可以来保存数据，这个数据存储区域被称作缓冲区。使用插入操作来将数据放入到缓冲区，使用取出操作来讲数据从缓冲区中提取出来。在对数据缓冲区执行插入或取出操作之前，数据缓冲区必须已经存在。一旦通信结束，则不再需要该数据缓冲区。因此，您的模型必须能够在数据缓冲区不再是必需的时候将它删除。如前所述，管道有两端，一端用于插入数据，另外一端用于提取数据，而且这些端可以被不同的进程访问。因此模型还应当包括一个输入端口和一个输出端口，而且它们能够连接到不同的进程。下面是 FIFO 模型的基本组成：

- 输入/输出端口
- 插入和取出操作
- 创建/初始化操作
- 缓冲区创建、插入、取出、析构

对于这个例子，在通信中只涉及两个进程。但是如果有多个进程可以对命名管道流进行读取和写入，则需要进行同步。因此，这个类还需要一个互斥量对象。示例 7-1 显示了 FIFO 类的开头部分：

示例 7-1

```
// Example 7-1 Declaration of fifo class.

 class fifo{
    mutex Mutex;
    //...
 protected:
    string  Name;
 public:
    fifo &operator<<(fifo &In, int X);
    fifo &operator<<(fifo &In, char X);
    fifo &operator>>(fifo &Out, float X);
    //...
};
```

使用这种技术，可以很容易地在构造函数中创建 fifo，可以将它们作为参数和返回值来进行传递，可以将它们同标准容器类等联合使用。这样的一个组件的构建大幅降低了使用 FIFO 所需要的代码数量，提供类型安全的机会，并且通常使得程序员能够在更高的级别上工作。

8. 消息队列

消息队列是字符串或消息的链表。这种 IPC 机制允许对队列具有足够许可的进程写入或删除消息。消息的发送方会为消息指定一个优先级。消息队列不要求被多个进程使用。对于 FIFO，写入进程会阻塞，不能够写入到管道中，直到另一个进程为了读取而打开它。对于消息队列，写入进程可以写入到消息队列，然后就终止，数据会保留在队列中。之后一些其他进程可以对它进行读取或写入。消息队列有内核持久性。当从队列中读取消息时，会返回时间最久且有着最高优先级的消息。队列中的每个消息具有这些属性：

- 优先级
- 消息的长度
- 消息或数据

对于链表，其头部有着队列中消息的最大数目以及所允许的最大消息长度。

使用消息队列

消息队列是使用 mq_open( )来创建的：

调用形式

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag,mode_t mode,
              struct mq_attr *attr);
int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

mq_open( )创建了以 name 作为名字的消息队列。这个消息队列使用有着 3 种可能的值的 oflag 来指定访问模式。

- O_RDONLY：为接收消息打开队列。
- O_WRONLY：为发送消息打开队列。
- O_RDWR：为发送或接收消息打开队列。

这些标志可以同如下标志进行 OR。

- O_CREAT：创建一个消息队列。
- O_EXCL：如果同前一个标志进行 OR，则如果路径名已经存在，函数失败。
- O_NONBLOCK：决定队列是否等待当前不可用的资源或消息。

函数返回一个类型为 mq_dt 的消息队列描述符。

最后一个参数是 struct mq_attr *attr。这是一个属性结构体，描述了消息队列的属性：

```
struct mq_attr {
   long mq_flags;        //flags
   long mq_maxmsg;       //maximum number of messages allowed
```

```
    long mq_msgsize;      //maximum size of message
    long mq_curmsgs;      //number of messages currently in queue
  }
```

mq_close( )关闭消息队列，但是消息队列仍然存在于内核中。然而，调用函数再也不能使用该描述符。如果进程终止，则同进程关联的所有消息队列也关闭。数据仍然保留在队列中。

unlink( )将由 name 指定的消息队列从系统中删除。对消息队列的引用的数目会被记录，但是队列名仍然会从系统中删除，即便引用数目大于 0 也是如此。队列不会被销毁，直到所有使用队列的进程都关闭或调用了 mq_close( )。

有两个函数可以设置和返回属性对象，如下所示：

调用形式

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes,struct mq_attr *attr);
int mq_setattr(mqd_t mqdes,struct mq_attr *attr,struct mq_attr *oattr);
```

当您使用 mq_setattr 来设置属性时，在 attr 结构体中只有 mq_flags 被设置，其他属性不受影响。其中 mq_maxmsg 和 mq_msgsize 是在消息队列创建时设置的。mq_curmsg 可以被返回，而且没有被设置。oattr 包含属性的之前的值。

为了将消息发送或写入到队列，使用这些函数：

调用形式

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *ptr, size_t len,
            unsigned int prio);
ssize_t mq_receive(mqd_t mqdes, const char *ptr, size_t len,
            unsigned int priop);
```

对于 mq_receive( )，len 必须最少为消息的最大长度。返回的消息被保存在*ptr。

消息队列接口类 posix_queue

posix_queue 是一个简单的类，为消息队列的部分功能进行了建模。它封装了基本的函数以及消息队列属性。程序清单 7-3 显示了 posix_queue 类的声明。

程序清单 7-3

```
// Listing 7-3 Declaration of the posix_queue class.
```

```
 1  #ifndef __POSIX_QUEUE
 2  #define __POSIX_QUEUE
 3  using namespace std;
 4  #include <string>
 5  #include <mqueue.h>
 6  #include <errno.h>
 7  #include <iostream>
 8  #include <sstream>
 9  #include <sys/stat.h>
10
11
12  class posix_queue{
13  protected:
14     mqd_t  PosixQueue;
15     mode_t  OMode;
16     int   QueueFlags;
17     string QueueName;
18     struct mq_attr  QueueAttr;
19     int  QueuePriority;
20     int MaximumNoMessages;
21     int MessageSize;
22     int ReceivedBytes;
23     void setQueueAttr(void);
24  public:
25     posix_queue(void);
26     posix_queue(string QName);
27     posix_queue(string QName,int MaxMsg, int MsgSize);
28     ~posix_queue(void);
29
30     mode_t  openMode(void);
31     void openMode(mode_t OPmode);
32
33     int queueFlags(void);
34     void queueFlags(int X);
35
36     int queuePriority(void);
37     void queuePriority(int X);
38
39     int maxMessages(void);
40     void maxMessages(int X);
41     int messageSize(void);
42     void messageSize(int X);
43
44     void queueName(string X);
45     string queueName(void);
46
47     bool open(void);
48     int send(string Msg);
49     int receive(string &Msg);
```

```
50      int remove(void);
51      int close(void);
52
53
54  };
55


#endif
```

消息队列执行的基本函数被封装在 posix_queue 类中：

```
47      bool open(void);
48      int send(string Msg);
49      int receive(string &Msg);
50      int remove(void);
51      int close(void);
```

我们已经讨论过这些函数的功能。示例 7-2～示例 7-6 显示了这些方法的定义。示例 7-2 是 open( )的定义。

示例 7-2

```
// Example 7-2 The definition of open().

122  bool posix_queue::open(void)
123  {
124     bool Success = true;
125     int RetCode;
126     PosixQueue = mq_open(QueueName.c_str(),QueueFlags,OMode,&QueueAttr);
127     if(errno == EACCES){
128        cerr  << "Permission denied to created " << QueueName << endl;
129        Success = false;
130     }
131     RetCode = mq_getattr(PosixQueue,&QueueAttr);
132     if(errno == EBADF){
133        cerr << "PosixQueue is not a valid message descriptor" << endl;
134        Success = false;
135        close();
136
137     }
138     if(RetCode == -1){
139        cerr << "unknown error in mq_getattr() " << endl;
140        Success = false;
141        close();
142     }
143     return(Success);
144  }
```

在第 126 行调用 mq_open( )之后，会检测打开消息队列的操作是否会因为已经存在名为 QueueName.c_str( )的消息队列而失败。如果的确存在，则调用不成功，bool Success 被返回时的值为 false。

在第 131 行，mq_getattr( )返回队列属性结构体。为了确保消息队列被打开而且成功地初始化它的属性，在第 132 行再次检查了 errno。EBADF 意味着描述符不是一个有效的消息队列描述符。返回的编码会在第 138 行进行检查。

示例 7-3 是 send( )的定义。

示例 7-3

```
// Example 7-3 The definition of send().

146 int posix_queue::send(string Msg)
147 {
148
149    int StatusCode = 0;
150    if(Msg.size() > QueueAttr.mq_msgsize){
151       cerr << "message to be sent is larger than max queue
                   message size " << endl;
152       StatusCode = -1;
153    }
154    StatusCode = mq_send(PosixQueue,Msg.c_str(),Msg.size(),0);
155    if(errno == EAGAIN){
156       StatusCode = errno;
157       cerr << "O_NONBLOCK not set and the queue is full " << endl;
158    }
159    if(errno == EBADF){
160       StatusCode = errno;
161       cerr << "PosixQueue is not a valid descriptor open for
                   writing" << endl;
162    }
163    if(errno == EINVAL){
164       StatusCode = errno;
165       cerr << "msgprio is out side of the priority range for the
                   message queue or " << endl;
166       cerr << "Thread my block causing a timing conflict with
                   time out" << endl;
167    }
168
169    if(errno == EMSGSIZE){
170       StatusCode = errno;
171       cerr << "message size exceeds maximum size of message
                   parameter on message queue" << endl;
172
173    }
174    if(errno == ETIMEDOUT){
175       StatusCode = errno;
176       cerr << "The O_NONBlock flag was not set, but the time expired
```

```
                    before the  message " << endl;
177         cerr << "could be added to the queue " << endl;
178      }
179      if(StatusCode == -1){
180         cerr << "unknown error in mq_send() " << endl;
181      }
182      return(StatusCode);
183
184  }
```

在第 150 行，对消息进程检查以确保它的大小不超过消息允许的大小。在第 154 行，调用 mq_send( )。所有其他代码是为了检测错误。

示例 7-4 是 receive( )的定义。

示例 7-4

```
//Example 7-4 The definition of receive().

187  int posix_queue::receive(string &Msg)
188  {
189
190     int StatusCode = 0;
191     char QueueBuffer[QueueAttr.mq_msgsize];
192     ReceivedBytes = mq_receive(PosixQueue,QueueBuffer,
                                  QueueAttr.mq_msgsize,NULL);
193     if(errno == EAGAIN){
194        StatusCode = errno;
195        cerr << "O_NONBLOCK not set and the queue is full " << endl;
196
197     }
198     if(errno == EBADF){
199        StatusCode = errno;
200        cerr << "PosixQueue is not a valid descriptor open for writing"
               << endl;
201     }
202     if(errno == EINVAL){
203        StatusCode = errno;
204        cerr << "msgprio is out side of the priority range for the message
                    queue or " << endl;
205        cerr << "Thread my block causing a timing conflict with time out"
               << endl;
206     }
207     if(errno == EMSGSIZE){
208        StatusCode = errno;
209        cerr << "message size exceeds maximum size of message parameter on
                    message queue" << endl;
210     }
211     if (errno == ETIMEDOUT){
212        StatusCode = errno;
```

```
213         cerr << "The O_NONBlock flag was not set, but the time expired
                  before the message " << endl;
214         cerr << "could be added to the queue " << endl;
215      }
216      string XMessage(QueueBuffer,QueueAttr.mq_msgsize);
217      Msg = XMessage;
218      return(StatusCode);
219
220  }
```

在第 191 行，以消息的最大大小作为长度，创建了缓冲区 QueueBuffer。调用 mq_receive( )。返回的消息保存在 QueueBuffer，而且字节数被返回并保存在 ReceivedBytes 中。在第 216 行，从 QueueBuffer 中提取消息，并在第 217 行中赋给字符串。

示例 7-5 是 remove( )的定义。

示例 7-5

```
//Example 7-5 The definition for remove().

221  int  posix_queue::remove(void)
222  {
223     int StatusCode = 0;
224     StatusCode = mq_unlink(QueueName.c_str());
225     if(StatusCode != 0){
226        cerr << "Did not unlink " << QueueName << endl;
227     }
228     return(StatusCode);
229  }
230
```

在第 224 行中，调用 mq_unlink( )来将消息队列从系统中删除。

示例 7-6 给出了 close( )的定义。

示例 7-6

```
//Example 7-6 The definition for close().

231  int posix_queue::close(void)
232  {
233
234     int StatusCode = 0;
235     StatusCode = mq_close(PosixQueue);
236     if(errno == EBADF){
237        StatusCode = errno;
238        cerr << "PosixQueue is not a valid descriptor open for
                 writing" << endl;
239     }
240    if(StatusCode == -1){
241        cerr << "unknown error in mq_close() " << endl;
```

```
242    }
243    return(StatusCode);
244
245  }
```

在第 235 行中，调用了 mq_close( )来关闭消息队列。

返回到程序清单 7-3，注意从示例 7-2～示例 7-6 中加粗的方法将消息队列的属性封装为设置和返回消息队列的属性：

```
33  int queueFlags(void);
34  void queueFlags(int X);
35
36  int queuePriority(void);
37  void queuePriority(int X);
38
39  int maxMessages(void);
40  void maxMessages(int X);
41  int messageSize(void);
42  void messageSize(int X);
```

这些属性中的一部分也可以在构造函数中设置。在程序清单 7-3 中共有 3 个构造函数：

```
25  posix_queue(void);
26  posix_queue(string QName);
27  posix_queue(string QName,int MaxMsg, int MsgSize);
```

第 25 行的为默认构造函数。示例 7-7 显示了它的定义。

示例 7-7

```
// Example 7-7 The definition of the default constructor.

 4  posix_queue::posix_queue(void)
 5  {
 6
 7
 8     QueueFlags = O_RDWR | O_CREAT | O_EXCL;
 9     OMode = S_IRUSR | S_IWUSR;
10     QueueName.assign("none");
11     QueuePriority = 0;
12     MaximumNoMessages = 10;
13     MessageSize = 8192;
14     ReceivedBytes = 0;
15     setQueueAttr();
16
17
18  }
```

posix_queue 类是消息队列的一个简单模型。这里并未包含进所有的功能，但是您可以看到消息队列类使得消息队列更加易于使用。posix_queue 类对 IPC 机制的所有主要函数进行错误检查。应当增加的是 mq_notify( )函数。有了通知信号，当空消息队列有消息时会向进程发送信号。这个类不具备同步能力，如果多个进程希望对 posix_queue 进行写入，应当在消息发送或接收时实现并使用一个内置互斥量。

7.1.4　什么是线程间通信

我们已经讨论了由 POSIX 定义的各种进行进程间通信的机制，这些进程可以是相关的或是不相关的。我们还讨论了这些 IPC 驻留在哪里以及它们的持久性。由于线程驻留在所属进程的地址空间内，推测线程间通信不会很难或者不要求仅仅为通信就需要使用特殊机制是有把握而且符合逻辑的，事实的确如此。当对等线程要求相互进行通信时，需要处理的最重要的问题是同步。进行线程间通信(Interthread Communication，ITC)时可能会遇到数据竞争和无限延迟。

线程间的通信用于：

- 共享数据
- 发送消息

多个线程共享数据的目的是使并发执行的处理成为流线型的。每个线程可以执行不同的处理或在数据流上的相同的处理。可以更改数据，或者可以创建新的数据作为结果，生成的结果也被共享。也可以传递消息。例如，如果某个线程中发生了一个事件，这可能触发另一个线程中的另一个事件。线程可能会传递一个信号给其他对等线程，或者是主线程可能会发信号给 worker 线程。

当两个进程需要通信时，它们会使用位于它们外部的结构。当两个线程通信时，它们通常使用属于它们所属的进程的一部分的结构。线程不能够同它们所属进程以外的线程通信，除非是引用进程的主线程。在那种情况下，您是将它们作为两个进程来使用的。进程内的线程可以通过进程的数据段或每个线程的栈段来传递值。

在多数情况下，进程间通信的成本要高于线程间通信的成本。在 IPC 期间，必须由操作系统创建的外部结构同 ITC 涉及的结构相比，会要求更多的系统处理。ITC 机制的效率使得线程在很多要求并发的编程场景成为了更有吸引力的替代方法，但是不是所有的编程场景都是如此。

注意：

第 5 章讨论了同进程相比，在使用线程中出现的一些问题和该方法的缺点。

表 7-4 列出了基本的线程间通信以及简要的描述。

表 7-4

ITC 的类型	描　述
全局数据、全局变量和全局数据结构	在 main()函数外部声明或有着全局作用域；对数据的任何更改可以被所有对等线程立即得到
参数	参数在线程创建期间传递给线程；泛型指针(generic pointer)可以被转换为任何数据类型
文件句柄	文件在线程间共享；这些线程共享相同的读写指针以及文件偏移

1. 全局数据、全局变量和全局数据结构

同进程相比，线程的一个重要优点就在于线程可以共享全局数据、全局变量和全局数据结构。进程中所有线程可以平等地访问它们。如果任何线程更改了数据，则这个改动对其他所有对等线程都是立即可利用的。例如，假定有 3 个线程 ThreadA、ThreadB 和 ThreadC。ThreadA 进行了计算并将结果保存到全局变量 Answer 中。ThreadB 读取 Answer，对它进行计算，然后将它的结果保存到 Answer。然后 ThreadC 也做相同的事情。最终的结果将会由主线程来显示。这个实例显示在程序清单 7-4、程序 7-5 和程序 7-6 中。

程序清单 7-4

```
// Listing 7-4 thread_tasks.h.

 1
 2  void *task1(void *X);
 3  void *task2(void *X);
 4  void *task3(void *X);
 5
```

程序清单 7-5

```
// Listing 7-5 thread_tasks.cc.

 1  extern int Answer;
 2
 3  void *task1(void *X)
 4  {
 5     Answer = Answer * 32;
 6  }
 7
 8  void *task2(void *X)
 9  {
10     Answer = Answer / 2;
11  }
12
```

```
13  void *task3(void *X)
14  {
15     Answer = Answer + 5;
16  }
```

程序清单 7-6

```
// Listing 7-6 main thread.

 1  using namespace std;
 2  #include <iostream>
 3  #include <pthread.h>
 4  #include "thread_tasks.h"
 5
 6  int Answer = 10;
 7
 8
 9  int main(int argc, char *argv[])
10  {
11
12     pthread_t ThreadA, ThreadB, ThreadC;
13
14     cout << "Answer = " << Answer << endl;
15
16     pthread_create(&ThreadA,NULL,task1,NULL);
17     pthread_create(&ThreadB,NULL,task2,NULL);
18     pthread_create(&ThreadC,NULL,task3,NULL);
19
20     pthread_join(ThreadA,NULL);
21     pthread_join(ThreadB,NULL);
22     pthread_join(ThreadC,NULL);
23
24     cout << "Answer = " << Answer << endl;
25
26     return(0);
27
28  }
```

在这些程序清单中，线程将要执行的任务定义在单独的文件中。Answer 是在文件 program7-6.cc 的主线(main line)上声明的全局数据。它在 thread_tasks.cc 中定义的任务的使用范围之外。它被声明为 extern，因此它有着全局作用域。如果线程将要使用前面描述的方式来处理数据，即 ThreadA、ThreadB 和 ThreadC 依次执行它们的计算，则要求同步。如果 ThreadA 和 ThreadB 有其他的工作必须先做，则并不保证返回正确的结果，即 165。多个线程是将数据从一个线程传送给另一个。假如使用两个多核，ThreadA 和 ThreadB 可以被执行。ThreadA 工作一个时间片，然后 ThreadC 得到处理器。当 ThreadA 被抢占时，它可能还没有对 Answer 执行计算。如果 ThreadC 在被抢占之前结束了，则 Answer 的值将会是 15。然后 ThreadB 结束，Answer 为 7。接下来 ThreadA 进行计算，Answer 为 224，

而不是 165。尽管数据通信的流水线模型是期望的模型，但是在执行中却没有在适当的位置进行同步。

线程还可以像使用变量那样的方式来共享数据结构。IPC 能够使用的仅仅是有限的数据结构集合(例如消息队列)，相反，任何类型的全局集合、映射等或任何其他集合类或容器类可被用来实现 ITC。例如，线程可以共享一个集合。对于集合，不同的线程可以使用多指令单数据(SIMD)或单指令单数据(SISD)内存访问模型来执行 membership、intersection、union 等操作。用于实现可用作 IPC 机制的集合容器的编码是被禁止的。

下面是程序清单 7-4、程序清单 7-5 和程序清单 7-6 的程序概要 7-2。

程序概要 7-2

程序名：

program7-6.cc (程序清单 7-6)

描述：

对于这个程序，在 program7-6.cc 的主线中声明了一个全局变量 Answer，它被声明为 extern，因此在 thread_tasks.cc 中有着全局作用域。Answer 将会被 ThreadA 和 ThreadB 处理，然后被 ThreadC 处理。它们将会执行它们的计算，要求进行同步。正确的结果是 165。尽管数据通信的流水线模型是被期望的，但是没有适当的位置来实现同步。

必需的库：

libpthread

必需的头文件：

< iostream > < pthread.h > "thread_tasks.h"

编译和链接指令：

c++ -o program7-6 program7-6.cc thread_tasks.cc –lpthread

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

./program7-6

注释：

无

2. 线程间通信的参数

线程的参数可用于线程之间或主线程与对等线程之间的通信。线程创建 API 支持线程参数。参数的形式是 void 指针：

```
int pthread_create(pthread_t *threadID,const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *restrict parameter);
```

在 C++中，void 指针是一种泛型指针，可用来指向任意数据类型。parameter 的值用来传递的值从简单的 char *到复杂的指向容器的指针或用户定义对象。在程序清单 7-7 和程序清单 7-8 的程序中，我们使用两个字符串队列作为全局数据结构。一个线程将队列作为输出队列使用，而另外一个线程使用同一个队列作为输入数据流，然后写入到第二个全局字符串队列。

程序清单 7-7

```
// Listing 7-7 Thread tasks that use two global data structures.

 1  using namespace std;
 2  #include <queue>
 3  #include <string>
 4  #include <iostream>
 5
 6  extern queue<string> SourceText;
 7  extern queue<string> FilteredText;
 8
 9  void *task1(void *X)
10  {
11     char Token = '?';
12
13     queue<string> *Input;
14
15
16     Input = static_cast<queue<string> *>(X);
17     string Str;
18     string FilteredString;
19     string::iterator NewEnd;
20
21     for(int Count = 0;Count < 16;Count++)
22     {
23        Str = Input->front();
24        Input->pop();
25        NewEnd = remove(Str.begin(),Str.end(),Token);
26        FilteredString.assign(Str.begin(),NewEnd);
```

```
27       SourceText.push(FilteredString);
28
29    }
30
31
32 }
33
34
35 void *task2(void *X)
36 {
37    char Token = '.';
38
39    string Str;
40    string FilteredString;
41    string::iterator NewEnd;
42
43    for(int Count = 0;Count < 16;Count++)
44    {
45       Str = SourceText.front();
46       SourceText.pop();
47       NewEnd = remove(Str.begin(),Str.end(),Token);
48       FilteredString.assign(Str.begin(),NewEnd);
49       FilteredText.push(FilteredString);
50
51
52    }
53
54 }
```

这些任务过滤一个字符串文字。task1 从字符串中删除“?”，task2 从字符串中删除“.”。task1 接受一个队列，该队列用作待过滤的字符串的容器。在第 16 行中，void *被类型转换为指向字符串队列的指针。task2 不要求有队列作为输入。它使用由 task1 生成的全局队列 SourceText。在它们的循环内部，将字符串从队列中移出，在字符串中将 token 删除，然后将新的字符串推入到全局队列中，对于 task1，字符串队列为 SourceText，对于 task2，字符串队列为 FilteredText。在第 6 行和第 7 行中将这两个队列均声明为 extern。

程序清单 7-8

```
// Listing 7-8 Main thread declares two global data structures.

 1  using namespace std;
 2  #include <iostream>
 3  #include <pthread.h>
 4  #include "thread_tasks.h"
 5  #include <queue>
 6  #include <fstream>
 7  #include <string>
 8
 9
```

```
10
11
12 queue<string> FilteredText;
13 queue<string> SourceText;
14
15 int main(int argc, char *argv[])
16 {
17
18    ifstream Infile;
19    queue<string> QText;
20    string Str;
21    int Size = 0;
22
23
24    pthread_t ThreadA, ThreadB;
25
26    Infile.open("book_text.txt");
27    for(int Count = 0;Count < 16;Count++)
28    {
29       getline(Infile,Str);
30       QText.push(Str);
31
32    }
33
34    pthread_create(&ThreadA,NULL,task1,&QText);
35    pthread_join(ThreadA,NULL);
36
37    pthread_create(&ThreadB,NULL,task2,NULL);
38    pthread_join(ThreadB,NULL);
39
40    Size = FilteredText.size();
41
42    for(int Count = 0;Count < Size ;Count++)
43    {
44       cout << FilteredText.front() << endl;
45       FilteredText.pop();
46
47    }
48
49    Infile.close();
50
51    return(0);
52
53 }
```

程序清单 7-8 中的程序显示了主线程的代码。它在第 12 行和第 13 行声明了两个全局队列。从文件中将字符串读入到字符串队列 QText 中。这个队列是第 34 行中的 ThreadA 的数据源队列。主线程对 ThreadA 调用 join 并等待它返回。当 ThreadA 返回后，ThreadB 使用由 task1 刚刚生成的全局队列 SourceText。当 ThreadB 返回后，由主线程将全局队列

FilteredText 中的字符串发送到 cout。通过由主线程以这种方式调用 join，这些线程不是并发执行的。主线程直到 ThreadA 返回之后才会创建 ThreadB。如果线程不是相继创建的，它们将会并发地执行，这样会导致 core dump 的威胁。如果 ThreadB 在 ThreadA 生成源队列之前执行，那么 ThreadB 会试图从空队列中进行弹出。在试图读取队列之前，可以对队列的长度进行检查，但是您希望在进行这样的处理时利用多核。本例的队列中只有几个字符串，而且只是希望从字符串中删除一个 token，但是如果将这个问题扩展到上千个字符串，而且希望删除的 token 有很多个，您会意识到必须采用其他的方法。您不会希望使用串行的解决方案。对队列的访问可以被同步，但是一次对所有字符串进行筛选也可以被并行化。本章稍后的部分将再次探讨这个问题并提供一个更好的解决方案。

下面是程序清单 7-7 和程序清单 7-8 的程序概要 7-3。

程序概要 7-3

程序名：

program7-8.cc (程序清单 7-8)

描述：

程序清单 7-8 中的程序显示了主线程的代码。它声明了两个全局队列来用作输入和输出。从文件中将字符串读入到字符串队列 QText，该队列是 ThreadA 的数据源队列。然后主线程对 ThreadA 调用 join 并等待它返回。当 ThreadA 返回后，ThreadB 使用由 task1 刚刚生成的全局队列 SourceText。当 ThreadB 返回后，由主线程将全局队列 FilteredText 中的字符串发送到 cout。

必需的库：

libpthread

必需的头文件：

< iostream > < pthread.h > < fstream > < queue > < string > "thread_tasks.h"

编译和链接指令：

c++ -o program7-8 program7-8.cc thread_tasks.cc –lpthread

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

./program7-8

注释：

无

对于进程，命令行参数通过使用 exec( )函数系列或 posix_spawn( )来进行传递，就像第 5 章讨论的那样。命令行参数被限制为简单数据类型，例如数字和字符。对进程的参数传递是单向通信，子进程只是简单地复制参数的值。对数据的任何更改都不会对父进程产生影响。对于线程，参数不是一个副本，而是一些数据位置的地址。线程对数据所做的任何更改都可以被使用该数据的所有线程看到。

但是这种类型的透明性可能不是您所期望的。线程也可以保留传递给它的数据的自己的副本。它可以将数据复制到它的栈中，但是栈中的内容会经常变化。线程可能会被多次调用，反反复复地执行相同的任务。通过使用线程特定数据，可以将数据与线程关联起来，而且数据是私有且具有持久性的。

3. 将文件句柄用于线程间通信

作为 ITC 的一种形式，在多个线程间共享文件要求像使用全局变量那样小心。如果 Thread A 移动了文件指针，则 Thread B 会从移动后的位置开始访问文件。如果一个线程关闭了文件，同时另外一个线程试图对文件进行写入，会发生什么情况呢？当某个线程对文件进行写入时，其他线程可以对文件进行读取吗？多个线程可以同时对文件进行写入吗？在多线程环境中，对文件的访问必须小心地进行串行化或同步。由于线程可以共享实际的读写指针，所以必须使用协作技术。

7.2　对并发进行同步

在任何计算机系统中，资源都是有限的。系统中就只有有限的内存、I/O 设备和端口、硬件中断，甚至处理器内核也需要进行分配。I/O 设备的数目通常会受到系统中 I/O 端口和硬件中断的限制。在一个有着有限硬件资源的环境中，由多个进程和线程组成的应用程序必须竞争内存位置、外围设备和处理器时间。有些线程和进程会密切地共同工作，使用系统中有限的可共享资源来执行任务并达成目标，而其他线程和进程异步地工作，并独立地为这些可共享资源竞争。决定进程或线程何时利用系统资源以及利用多长时间是操作系统的工作。对于抢占式调度，操作系统可以中断进程或线程，以容纳所有竞争系统资源的进程和线程。资源分为软件资源和硬件资源，软件资源的实例是共享库，它为进程或线程提供了公共的服务集或函数集。其他可共享的软件资源有：

- 应用程序
- 程序

- 实用工具

为了共享软件资源，只需要将程序代码的一份副本引入到内存中。数据资源是对象、系统数据文件(例如环境变量)、全局定义的变量和数据结构。第7.1节，讨论了用于数据通信的数据源、进程和线程有可能拥有共享数据资源的自己的副本。在其他情况下，可能会希望(甚至是必须)数据是共享的。共享数据很有技巧性，可能会导致竞争条件(同时更改数据)或在需要数据时找不到数据。在试图同步对资源的访问时，如果没有被适当地执行或使用了错误的IPC或ITC机制，则可能会导致问题。这可能会导致无限延迟或死锁。同步使得多个线程或进程同时活动，而且可以在相互不影响操作的情况下共享资源。同步过程暂时地串行化(在某些情况下)多个任务的执行，以防止问题的发生。如果必须允许对硬件资源或软件资源"每次一个(one-at-a-time)"的访问，则会发生串行化。但是过多的串行化会使并发和并行化的优势丧失，内核会处于空闲状态。串行化被用作没有其他办法时的最后的方法。协调是最关键的。

7.2.1 同步的类型

我们前面讨论的系统资源是共享的硬件资源和软件资源，它们是系统中要求同步的实体。还应当将任务包含进来，任务也应当被同步。在程序清单7-7和程序清单7-8中的程序证明了这一点。task1必须在task2可以开始之前执行并结束，因此，一共有3种主要的同步种类：

- 数据
- 硬件
- 任务

表7-5总结了每种类型的同步。

表7-5

同步的类型	描　述
数据	对防止竞争条件是必需的；它允许并发线程/进程安全地访问一个内存块
硬件	当需要多个硬件设备来执行一个任务或一组任务时是必需的；它要求任务间的通信，而且要求对实时性能和优先级设置有着紧密的控制
任务	对防止竞争条件是必需的；它强制执行合理过程的前置条件和后置条件

7.2.2 同步对数据的访问

至此，本章已经讨论了IPC和ITC。如同已经讨论的那样，进程数据共享同线程数据共享的区别在于线程共享相同的地址空间，而进程有着独立的地址空间。IPC存在于通信所涉及的进程的地址空间外部，在内核空间或文件系统中。共享内存将一个结构映射到一块内存中，该内存块是进程可以访问的。ITC是全局变量和数据结构。要求同步

的是 IPC 和 ITC 机制。图 7-6 显示了在进程的布局中 IPC 和 ITC 机制存在的位置。

之所以需要进行数据同步，目的是为了控制竞争条件以及允许并发线程或进程安全地访问某个内存块。数据同步控制内存块何时可以被读取或被更改。在多线程环境中，对共享内存、全局变量和文件的并发访问必须进行同步。在任务代码试图访问和其他并发执行的线程或进程共享的内存块、全局变量或文件时，需要数据同步。这被称作临界区(critical section)。临界区可以是任何对文件读写、关闭文件、读写全局变量或数据结构的代码块。

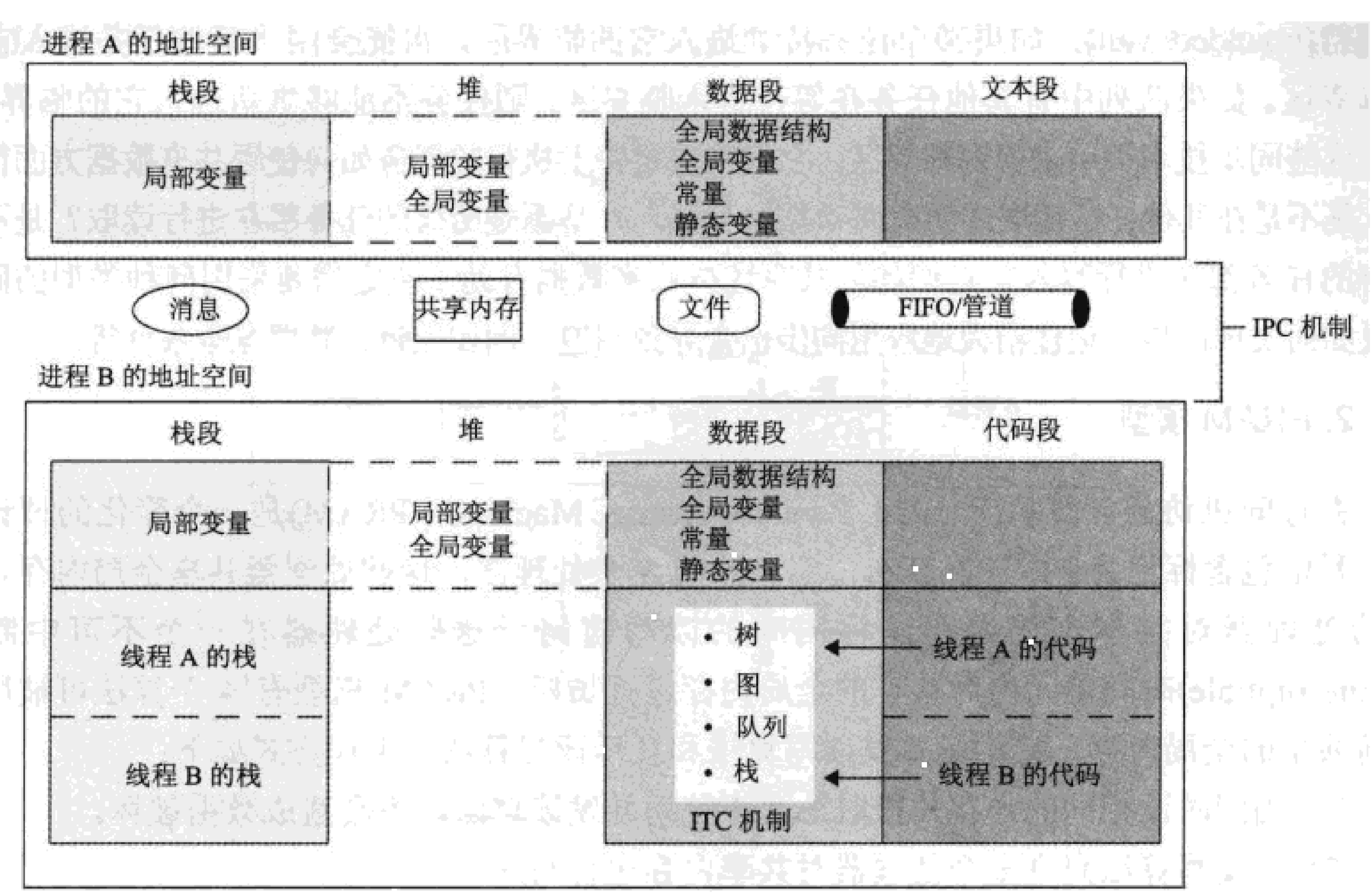


图 7-6

1. 临界区

临界区是访问共享资源的代码区或代码块，由于资源被多个并发任务共享，因此必须对这些代码块进行控制。临界区通过一个入口点(entry point)和一个出口点(exit point)来标记。在临界区中的实际代码可以只是一行代码，其中进程或线程对内存或文件进行读写，也可以是多行代码，其中的处理和对其他方法的调用涉及共享数据。入口点标记进入临界区，出口点标记离开临界区。

为了解决由多个并发任务共享资源而导致的问题，必须满足3个条件：

(1) 如果一个任务位于其临界区中，其他共享资源的任务不能够在它们的临界区中执行。它们将被阻塞。这被称作互斥现象(mutual exclusion)。

(2) 如果没有哪个任务位于临界区中，则任何被阻塞的任务可以进入到它们的临界区中，这被称作进行(progress)。

(3) 应当过一定的时间，才允许某个任务重新进入它的临界区，这个时间段被称作有限等待(bounded wait)。如果某个任务持续进入它的临界区，可能会阻止其他任务进入它们的临界区。如果队列中有其他任务在等待进入临界区，则任务不能够重新进入它的临界区。

这些同步技术被用来管理临界区。它们在决定并发执行的任务如何使用共享数据方面很重要。是不是在其他任务读取的同时对数据进行写入？是不是所有的任务都在进行读取？是不是所有的任务都在进行写入？它们如何共享这些共享数据有助于决定需要采用何种类型的同步以及如何实现同步。记住错误地应用同步也会导致问题，例如死锁、数据竞争条件等。

2. PRAM 模型

并行随机访问计算机(Parallel Random-Access Machine，PRAM)是一个简化的理论模型，其中包含标记为P1，P2，P3，…，PN的N个处理器，这些处理器共享全局内存。所有的处理器对共享全局内存进行同时的读写访问。这些处理器在一个不可中断的(uninterruptible)时间单元内对共享的全局内存进行访问。PRAM模型有4个算法可被用来访问共享的全局内存，它们是并发读写算法和互斥读写算法，工作方式如下：

(1) 当同时读相同的内存片段时，允许使用并发读算法，不会造成数据破坏。

(2) 并发写算法允许多个处理器对共享内存进行写入。

(3) 互斥读算法被用来保证不会有两个处理器同时对相同的内存位置进行读取。

(4) 互斥写算法被用来保证不会有两个处理器同时对相同的内存位置进行写入。

这个PRAM模型可被用来刻画多个任务对共享内存的并发访问。

并发和互斥内存访问

并发和互斥读写算法可以组合为如下的可能读写访问算法组合类型：

- 互斥读互斥写(Exclusive Read and Exclusive Write，EREW)
- 并发读互斥写(Concurrent Read and Exclusive Write，CREW)
- 互斥读并发写(Exclusive Read and Concurrent Write，ERCW)
- 并发读并发写(Concurrent Read and Concurrent Write，CRCW)

这些算法可被视为由共享数据的任务实现的访问策略。图7-7显示了这些访问策略。EREW意味着对共享内存的访问被串行化，每次只能够有一个任务访问共享内存，无论它是读访问还是写访问。EREW访问策略的实例就是生产者-消费者(producer-consumer)。第5章的程序清单5-7中的程序对进程间共享的posix_queue使用了EREW访问策略。程序中一个进程将文件名写入到队列，该文件名是另一个进程要在其中搜索编码的文件的名字。对包含文件名的队列的访问对于生产者限制为互斥写，对消费者限制为互斥读。在任何时

刻，只有一个任务能够访问队列。

CREW 访问策略允许对共享内存发生多个读取和互斥的写入。在能够并发读取共享内存的任务数目方面没有限制，但是任何时刻只有一个任务能够对共享内存进行写入。在互斥写发生的期间可以进行并发读。使用这种访问策略时，在其他任务正在写入时，每个读取任务可能读到不同的值。下一个读取共享内存的任务将会看到与某些其他任务不同的数据。这可能是有意的，也可能不是。ERCW 访问策略同 CREW 策略正好相反。只有一个任务能够读取共享数据，但是并发写入是允许的。CRCW 访问策略允许并发读取和并发写入。

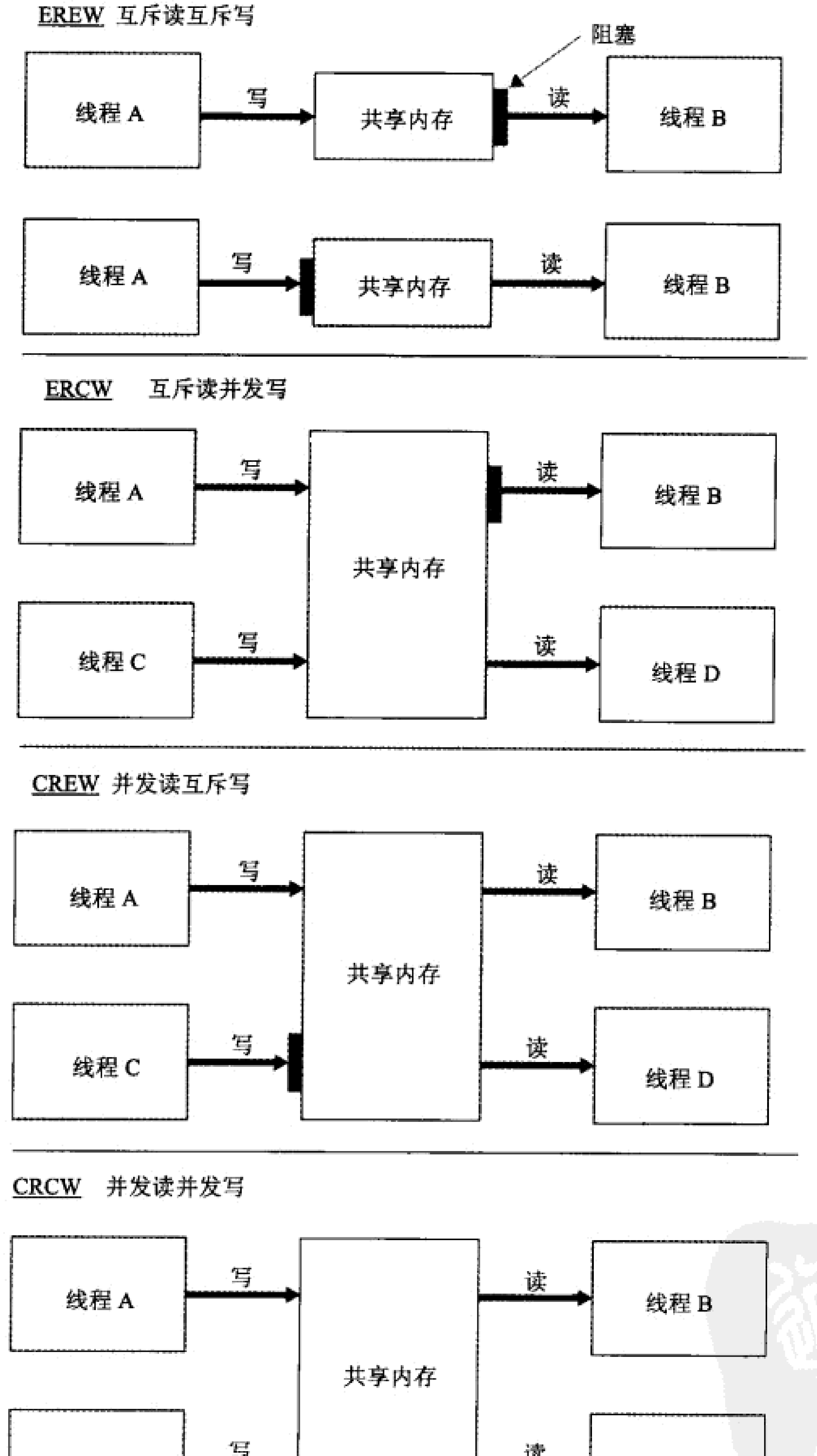


图 7-7

这 4 种算法类型中的每一个要求不同级别和类型的同步。可以通过一个连续区间来对它们进行分析，区间的一端是需要实现最少量同步的访问策略，另一端是需要最大量同步的访问策略。EREW 是最易于实现的策略，因为 EREW 本质上是强制顺序处理。您可以认为 CRCW 是最简单的，但是它提出了最多的挑战。它可能看上去对内存可以不加限制地访问，但是它是最难以实现的，而且要求最多的同步，这样才能够达成实现保持数据完整性并满足系统性能的同步处理的目标。

并发任务：协调执行顺序

协调并发任务的执行顺序也需要同步。执行顺序在程序清单 7-5 和程序清单 7-6 的程序中非常重要。如果任务是乱序执行的，则 Answer 的最终结果将会是错误的。在程序清单 7-7 和程序清单 7-8 的程序中，如果 task1 还没有结束，则 task2 会试图从一个空的队列中进行读取。这时候就需要使用同步来协调这些任务，使得工作可以继续进行或使得产生的结果正确。数据同步(访问同步)和任务同步(顺序同步)是执行多个并行任务时要求的两种同步类型。任务同步确保了合理过程的前置条件和后置条件。

协作任务之间的关系

在一个进程中的任意两个线程之间，或一个应用程序中的任意两个进程之间，存在着 4 种基本的同步关系：

- start-to-start(SS)
- finish-to-start(FS)
- start-to-finish(SF)
- finish-to-finish(FF)

这 4 种基本关系刻画了线程间和进程间工作的协调。图 7-8 显示了每种同步关系的活动图。

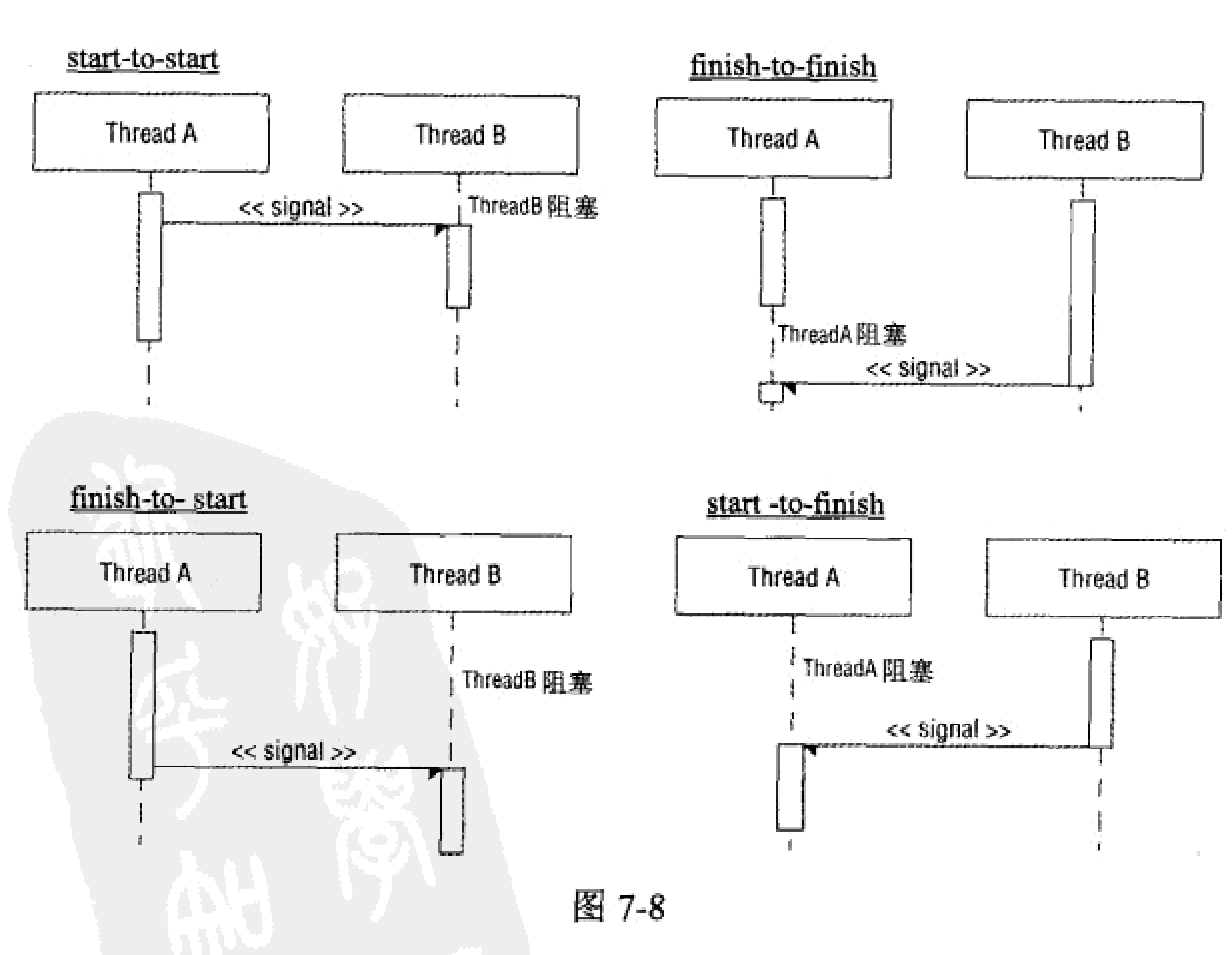
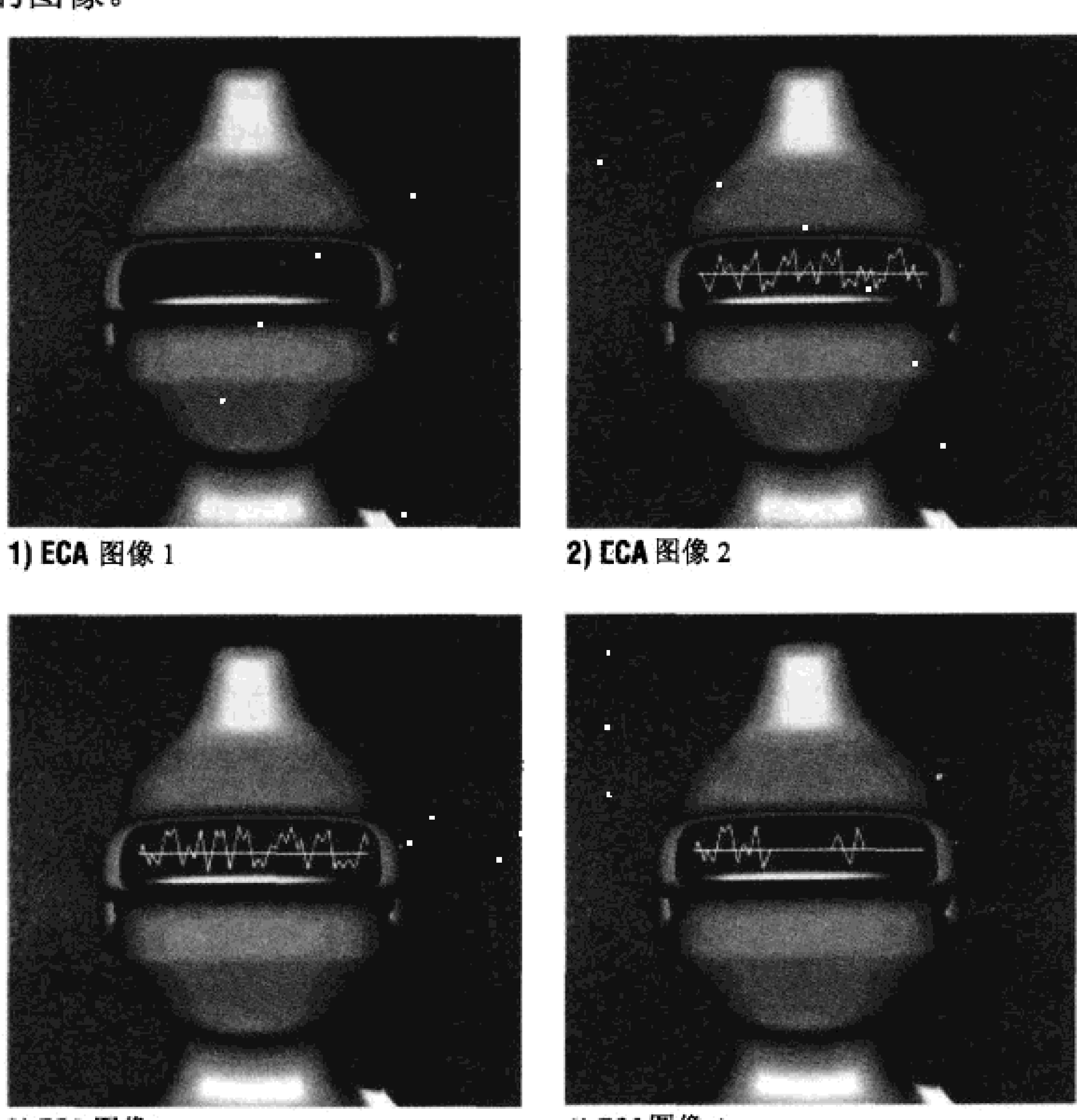


图 7-8

SS 关系

在 SS 同步中，某个任务不能够在另一个任务开始之前开始。一个任务可能在另一个任务之前开始，但是决不会在它之后开始。例如，假定有一个程序实现嵌入式会话 agent(Embedded Conversational Agent，ECA)。ECA 是一个由计算机生成的头部特写(talking head)，它为软件提供了人性化。实现 ECA 的程序有多个线程。这里，我们主要关注控制眼睛动画的线程(ECA 没有嘴巴、眼睛会动)以及控制声音的线程。您希望产生声音和眼睛动画同步的效果。理想情况下，它们应当正好在相同的瞬间执行。如果有多个处理器内核，则两个线程可以同时开始。线程有 SS 关系。由于计时的条件，允许产生声音的线程(Thread A)稍早于开始动画的线程(Thread B)开始，但是为了视觉效果，不能够开始得太早。由于声音的初始化需要的时间稍微长一些，因此它可以稍早些开始，而图形的加载非常快。图 7-9 显示了 ECA 的图像。



1) ECA 图像 1　2) ECA 图像 2　3) ECA 图像 3　4) ECA 图像 4

图 7-9

FS 关系

在 FS 同步中，Task A 不能够结束，直到 Task B 启动。这种关系对于父子进程是很常见的。父进程不能够结束某些操作的执行，直到它产生一个子进程或接收到来自子进程的通信，该子进程已经开始它的操作。子进程一旦给父进程发送信号或提供需要的信息之后，会继续执行。然后父进程就可以自由地完成它的操作。

SF 关系

SF 同步关系与 FS 恰好相反。在 SF 同步关系中，某个任务不能够开始，直到另一个任务结束为止。Task A 不能够开始执行，直到 Task B 结束执行或者完成特定的操作。程序清单 7-7 和程序清单 7-8 中的程序具有 SF 同步。直到 task1 结束，task2 才能够开始。主线程为了同步使用了合并。主线程被阻塞，直到 task1 返回，然后它创建一个执行 task2 的线程。

如果进程 A 从连接到进程 B 的管道中进行读取，则进程 B 在进程 A 从管道中读取之前必须首先写入到管道。在进程 A 开始之前，进程 B 必须完成至少一个操作，即将一个元素写入到管道。对于管道，其内部通过使用阻塞具有内置的有限的同步。但是如果有多个读进程和写进程，则要求使用更为精细的同步。

FF 关系

FF 同步关系意味着某个任务不能够结束，直到另外一个任务结束为止。在 Task B 结束之前，Task A 不能够结束。这种关系可以描述父进程同子进程之间的关系。父进程必须等待，直到它所有的子进程已经终止，然后父进程才可以终止。如果父进程在子进程之前终止，那么那些未被终止的子进程会成为僵死(zombie)进程。父进程为每个子进程调用 wait( )，就像为线程调用 join 那样，或者等待由子进程广播的互斥量或条件变量。

FF 关系的另外一个实例就是 boss-worker 并发模型。boss 的任务是将工作委派给 worker。如果 boss 在 worker 之前终止是不合需要的。对系统的新的请求将不会被处理，现有线程将没有工作来执行，而且将不会创建新的线程。如果 boss 是主线程，而且它终止了，那么进程将会和所有 worker 线程一同终止。在对等(peer-to-peer)模型中，如果线程 A 动态分配一个对象并传送给线程 B，并且线程 A 终止，那么对象也会随同线程 A 销毁。如果这发生在线程 B 有机会使用该对象之前，那么会发生段错误或数据访问违规。为了防止线程发生这些错误，线程的终止通过使用 pthread_join( )进行同步。这产生了 FF 同步。

7.2.3 同步机制

这一部分讨论的同步机制既适用于进程，也适用于线程。这些机制可通过实现我们已经提到过的同步访问策略并管理任务的临界区来防止多个任务之间发生竞争条件和死锁。接下来我们将介绍：

- 信号量和互斥量
- 读-写锁
- 条件变量

1. 信号量

信号量是一种同步机制，用于管理同步关系并实现访问策略。信号量是一种特殊类型的变量，只能够通过非常特殊的操作来访问。信号量帮助线程和进程同步对共享可更改内

存的访问，或管理对设备或其它资源的访问。信号量就像准许对资源进行访问的钥匙。这把钥匙在某一时刻只可以被一个进程或线程拥有。无论是哪个任务拥有了这把钥匙(信号量)之后，就为了它对资源的独占使用而加锁。对资源加锁导致任何其他希望访问该资源的任务必须等待，直到资源被解锁。当信号量被解锁，队列中等待该信号量的下一个任务会得到信号量，这样就可以访问资源了。哪个任务是“下一个任务”是由线程或进程使用的调度策略决定的。图 7-10 显示了信号量的基本概念。

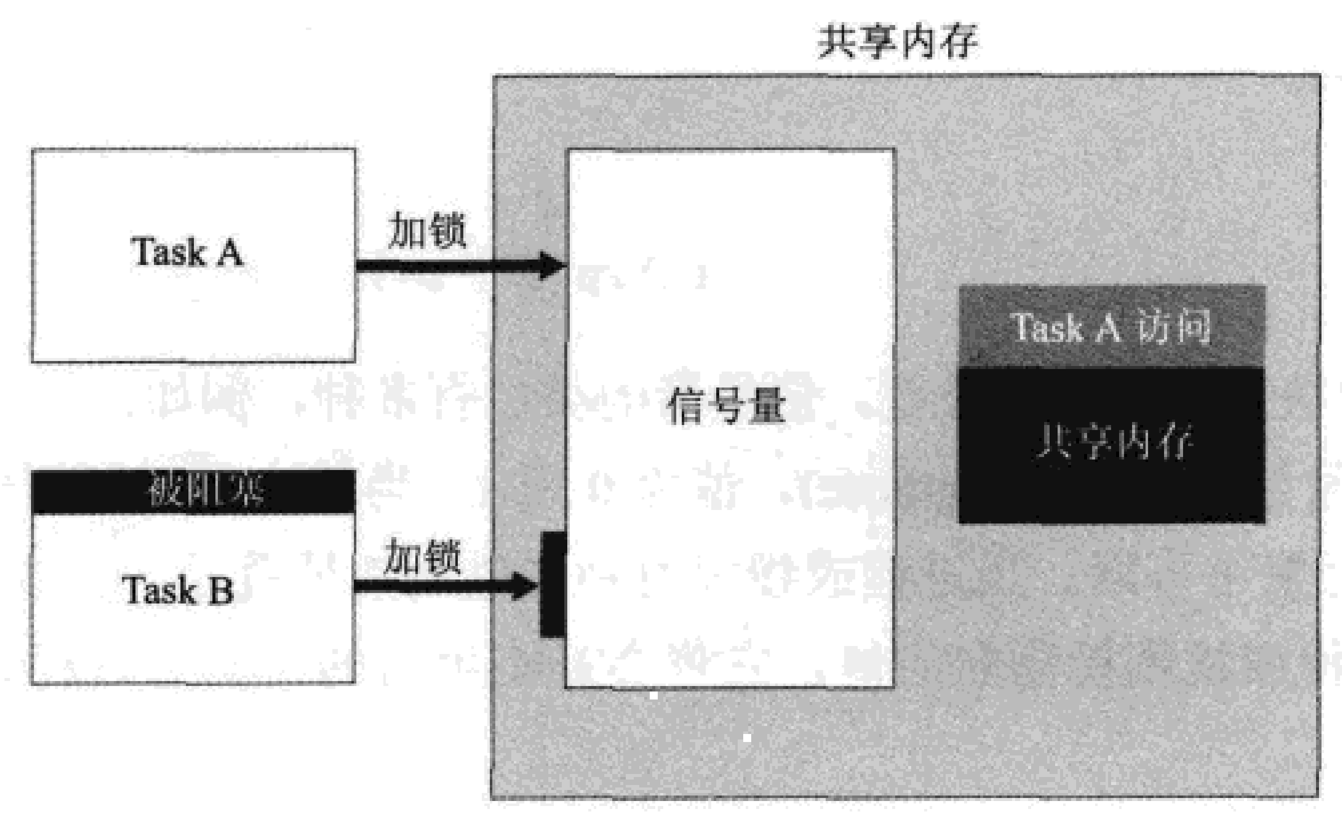


图 7-10

基本的信号量操作

信号量只能够通过指定的操作来进行访问。对信号量可以执行两个操作：P( )操作和 V( )操作。P( )操作减少信号量，而 V( )操作增加信号量：

```
P(Mutex)
if(Mutex > 0){
  Mutex--;
}
else {
    Block on Mutex;
}
```

```
V(Mutex)
if(Blocked on Mutex N processes){
  pass on Mutex;
}
else{
    Mutex++;
}
```

这里，Mutex 是信号量。实际实现是由系统决定的。这些操作是不可分割的，意味着操作一旦开始，则不能够被抢占。如果多个任务试图对 P( )操作进行调用，只有一个任务能够被允许进行。如果 Mutex 已经被减少，那么任务将会被阻塞并放到一个队列中。V( )操作由拥有 Mutex 的任务调用。如果有其他任务等待 Mutex，会根据调度策略将 Mutex 交

给队列中的下一个任务。如果没有任务在等待，则 Mutex 会增加。

信号量操作可以以其它名字来进行，例如：

```
P() operation
lock()
wait()
own()


V() operation
unlock()
post()
unown()
```

信号量的值取决于信号量的类型。信号量的类型有多种，例如：

- 二进制信号量(binary semaphore)，值为 0 或 1。当值为 1 时，信号量可用，当值为 0 时，信号量不可用。当进程或线程得到二进制信号量后，值被减为 0，这样如果另一个进程或线程检测它的值，它将会是不可用的。一旦进程或线程完成，信号量增加。
- 计数信号量(counting semaphore)有某个非负整数值。它的初始值表示可用资源的数目。

POSIX 标准定义了多种类型的信号量。有些信号量仅被线程使用，其他信号量可以被进程或线程使用。任何兼容 Single Unix Specification 或 POSIX 标准的操作系统都能够提供这些信号量的实现。它们是 libpthread 库的一部分，而且它们的函数在 pthread.h 头文件中声明。

POSIX 信号量

POSIX 信号量定义了一个命名二进制信号量。名称对应于文件系统中的路径。表 7-6 列出了使用信号量的基本功能以及简要介绍。

表 7-6

基本的信号量操作	描　述
初始化	分配持有信号量所需要的内存并给内存赋初始值；还决定了信号量是否是私有(private)、可共享(sharable)、被占有(owned)和未被占有(unowned)
请求占有权	发出占有信号量的请求；如果信号量已经被其他线程占有，则线程会被阻塞
释放占有权	释放信号量，从而使它可以被那些被阻塞的线程得到
尝试占有权	测试信号量的占有权；如果信号量已经被占有，请求方不会被阻塞，而是继续执行；在继续之前可以等待一段时间
销毁	释放同互斥量相关联的内存；如果互斥量被占有或还有线程正在等待，则该互斥量不能够被销毁或关闭

程序清单 7-9 显示了如何在多个进程间使用信号量。

程序清单 7-9

```
// Listing 7-9 A process using a semaphore on an output file.

 1  using namespace std;
 2  #include <semaphore.h>
 3  #include <iostream>
 4  #include <fstream>
 5
 6
 7  int main(int argc, char *argv[])
 8  {
 9
10     int Loop, PN;
11     sem_t *Sem;
12     const char *Name;
13     ofstream Outfile("out_text.txt",ios::app);
14
15     PN = atoi(argv[1]);
16     Loop = atoi(argv[2]);
17     Name = argv[3];
18
19     Sem = sem_open(Name,O_CREAT,O_RDWR,1);
20     sem_unlink(Name);
21
22
23     for (int Count = 1; Count < Loop; ++Count) {
24        sem_wait(Sem);
25        Outfile << "Process:" << PN << " counting " << Count << endl;
26        sem_post(Sem);
27
28     }
29     Outfile.close();
30
31     exit(0);
32
33
34
35  }
```

程序清单 7-9 中的程序在第 19 行打开了一个类型为 sem_t 的信号量。命名的信号量 Sem 使用命令行中键入的第三个参数作为 Name 打开。O_CREATE 和 O_RDWR 是指定如何打开信号量的标志。在这个案例中，信号量只在不存在的情况下被创建并打开。由于设置了 O_RDWR 标志，信号量是用读写许可打开的。Sem 使用 1 进行初始化。sem_wait 和 sem_post 操作封装了对 Outfile 的访问。在第 25 行的执行期间，不应当有其它进程访问文件。所有使用这个文件进行输入或输出的进程都应当使用同一个信号量。在程序清单 7-10

中，一个读取文件的进程也使用了这个信号量。

程序清单 7-10

```
// Listing 7-10 A process using a semaphore on an input file.

 1  using namespace std;
 2  #include <semaphore.h>
 3  #include <iostream>
 4  #include <fstream>
 5  #include <string>
 6
 7
 8  int main(int argc, char *argv[])
 9  {
10
11     string Str;
12     const char *Name;
13     sem_t *Sem;
14     ifstream Infile("out_text.txt");
15
16     if(Infile.is_open()){
17        Name = argv[1];
18        Sem = sem_open(Name,O_CREAT,O_RDWR,1);
19        sem_unlink(Name);
20
21        while(!Infile.eof() && Infile.good()){
22           sem_wait(Sem);
23           getline(Infile,Str);
24           cout << Str << endl;
25           sem_post(Sem);
26
27        }
28        cout << "--------------------------------" << endl;
29
30        Infile.close();
31
32     }
33
34     exit(0);
35
36
37
38  }
```

在程序清单 7-10 的程序中，命名信号量 Sem 使用命令行中输入的第一个参数作为 Name 打开。O_CREATE 和 O_REWR 是用来指定如何打开信号量的标志，与程序清单 7-9 中一样。sem_wait 和 sem_post 操作封装了对 Infile 的访问。在第 23 行的执行期间，程序清单 7-9 中的进程不能够对文件进行写入。

下面是程序清单 7-9 和程序清单 7-10 的程序概要 7-4。

程序概要 7-4

程序名：

program7-9.cc (程序清单 7-9)
program7-10.cc (程序清单 7-10)

描述：

程序清单 7-9 中的 program7-9 打开了类型为 sem_t 的命名信号量 Sem。它是使用作为第三个命令行参数输入的 Name 打开的。Sem 使用 1 来进行初始化。sem_wait 和 sem_post 操作封装了对 Outfile 的访问。在执行期间不应当有其他进程访问文件。所有使用这个文件进行输入或输出的进程应当使用相同的信号量。在程序清单 7-10 中，读取文件的进程也使用这个信号量。程序从命令行中查找进程号、循环变量以及信号量的名字。程序将进程、进程号和循环迭代次数写入到文件。

程序清单 7-10 中的 program7-10 使用作为第一个命令行参数输入的 Name 来打开命名信号量 Sem。它应当同 program7-9 使用相同的信号量名字，这样才能够协调对 out_text.txt 的访问。sem_wait 和 sem_post 操作封装了对 Infile 的访问。在执行期间，program7-9 不能够对文件进行写入。这个程序要求信号量的名称采用命令行参数的形式。它使用了命名信号量。它打开文件 out_text.txt 并将该文件的内容写入到 stout。

必需的库：

```
librt
```

必需的头文件：

```
< semaphore.h > < iostream > < fstream > < string > < fcntl.h >
```

编译和链接指令：

```
c++ -o program7-9 program7-9.cc -lrt
c++ -o program7-10 program7-10.cc -lrt
```

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

这些程序要求命令行参数。对于 program7-9，第一个参数是进程号，第二个参数是循

环变量，第三个参数是信号量的名字。program7-10 只要求信号量的名称。

```
./program7-9 3 4 /sugi & ./program7-10 /sugi
```

注释:

确保信号量的名字包含一个“/”。这些程序将要同时被执行。

互斥量信号量

POSIX 标准定义了类型为 pthread_mutex_t 的互斥量信号量，它可以被线程或进程使用。互斥量意味着互斥现象。互斥量是信号量的一种，但是它们之间又存在一些差别。互斥量必须总是由对它加锁的线程来解锁。对于信号量，可以由执行 wait(或 lock)的线程以外的其他线程来执行 post(或 unlock)。因此，一个线程或进程可以调用 wait( )，另一个进程/线程可以在同一个信号量上调用 post( )。

pthread_mutex_t 提供了必要的基本操作，使得它成为了一种实用的同步机制：

- 初始化(Initialization)
- 请求占有权(Request ownership)
- 释放占有权(Release ownership)
- 尝试占有权(Try ownership)
- 销毁(Destruction)

表 7-7 列出了用于执行这些基本操作的 pthread_mutex_t()函数。初始化操作分配持有互斥量信号量所要求的内存，并给内存赋初始值。二进制信号量的初始值为 0 或 1。计数信号量的值代表信号量要跟踪的资源的数目。它可以表示在一个会话中，程序能够处理的请求限制。同正常的变量相反，不保证对互斥量的初始化操作一定发生。一定要采取预防措施来保证互斥量被初始化，方法是检查返回值或检查 errno 的值。如果为互斥量留出的空间已经被使用、超出允许的信号量数目、命名信号量已经存在，或出现一些其他内存分配问题，则系统对互斥量的创建会失败。

表 7-7

互斥量操作	函数原型/宏 #include <pthread.h>
初始化	int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);

请求占有权	<time.h>
	int pthread_mutex_lock(pthread_mutex_t *mutex);
	int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abs_timeout);

释放占有权	int pthread_mutex_unlock (pthread_mutex_t *mutex);
尝试占有权	int pthread_mutex_trylock(pthread_mutex_t *mutex);
销毁	int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread 互斥量有一个属性对象，它封装了互斥量的所有属性。这个属性对象的使用类似于线程的属性对象。这里的差别在于尽管线程的属性对象是集合，但是互斥量的属性没有同它关联的属性组的集合。我们将在本章稍后部分讨论这一点。目前需要理解的重要的一点是属性对象可以被传递给使用对象中那些集合的属性来创建互斥量的初始化函数。如果没有使用属性对象，则会使用默认值来初始化互斥量。pthread_mutex_t 被初始化为未加锁的(unlocked)和私有的(private)。私有的互斥量在同一个进程内的线程间共享，而共享的互斥量在多个进程的线程间共享。如果要使用默认属性，则可以使用该宏对互斥量进行静态地初始化：

```
pthread_mutext Mutex = PTHREAD_MUTEX_INITIALIZER;
```

这样就创建了一个静态分配的互斥量对象。这种方法的开销较小，但是没有执行错误检查。

请求占有权操作将互斥量的占有权授予调用进程或线程。互斥量可以是被占有的或者是未被占有的。一旦被占有，占有它的线程或进程就可以对这个资源进行独占地访问。如果其他进程或线程尝试去占有互斥量(通过调用这个操作)，它们会被阻塞，直到互斥量重新可用。当互斥量被释放之后，会导致被阻塞的下一个进程或线程解除阻塞并获得对互斥量的占有权。对于 pthread_mutex_lock( )，被授予对某个互斥量的占有权的线程是唯一可以释放该互斥量的线程。

尝试所有权操作对互斥量进行测试，查看它是否已经被占有。如果该互斥量已经被占有，则函数会返回某些值。这个操作的优点在于线程或进程不会被阻塞，它们可以继续执行代码。如果互斥量没有被占有，则调用这个操作的进程或线程会被授予占有权。

销毁操作释放同互斥量关联的内存。如果互斥量被占有或有进程或线程在等待，则内存不能够被销毁或关闭。

使用互斥量属性对象

pthread_mutex_t 拥有一个属性对象，它的用法同线程的属性对象类似。如前所述，属性对象封装了互斥量对象的属性。一旦被初始化，属性对象可以在被传递给 pthread_mutex_init( ) 时被多个互斥量对象使用。同线程属性函数相反，没有强制属性同对象关联。可以用于设置互斥量属性的函数同如下内容有关：

- 优先级最大值
- 协议
- 共享
- 类型

表 7-8 列出了这些函数以及简要的描述。

表 7-8

pthread_mutex_t 属性对象函数原型 #include <pthread.h>	描　　述
创建/销毁 int pthread_mutexattr_init (pthread_mutexattr_t * attr); int pthread_mutexattr_destroy (pthread_mutexattr_t * attr);	为所有属性使用由实现定义的默认值来初始化由参数 attr 指定的互斥量属性对象； 销毁由 attr 指定的互斥量属性对象，这会导致互斥量属性对象成为未被初始化的；可以通过调用 pthread_mutexattr_init( )函数来重新初始化

优先级最大值
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t * attr, int prioceiling);
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t*restrict attr, int *restrict prioceiling);
	定义互斥量的最小优先级级别；
	设置并返回由 attr 指定的互斥量的优先级最高限度属性；prioceiling 包含互斥量的优先级最高限度；这个值介于由 SCHED_FIFO 定义的优先级最大范围之内

协议
int pthread_mutexattr_setprotocol(pthread_mutexattr_t * attr,int protocol);
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr, int *restrict protocol);
	定义如何利用互斥量的调度和优先级；
	设置并返回由 attr 指定的互斥量属性的 protocol；protocol 包含了协议属性的值：
	PTHREAD_PRIO_NONE
	线程的优先级和调度不受互斥量占有权的影响；
	PTHREAD_PRIO_INHERIT
	使用该协议时，由于占有这样一个互斥量，线程会阻塞其他有着较高优先级的线程，使得它运行在等待本线程占有的任何互斥量的任何线程的最高优先级上；
	PTHREAD_PRIO_PROTECT
	使用该协议时，占有这样一个互斥量的线程运行在该线程占有的所有互斥量的优先级最高限度上，无论是否有其他线程被这些互斥量中的任何一个所阻塞

共享 int pthread_mutexattr_setpshared (pthread_mutexattr_t * attr, int pshared); int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,int *restrict pshared);	判断互斥量是否用于进程； 设置或返回由 attr 指定的互斥量属性对象的 shared 属性；pshared 包含如下值之一： PTHREAD_PROCESS_SHARED 允许互斥量被任何有权访问互斥量被分配的内存的线程所共享，即使线程位于不同的进程； PTHREAD_PROCESS_PRIVATE 作为被初始化的互斥量，互斥量被同一进程内的线程所共享

(续表)

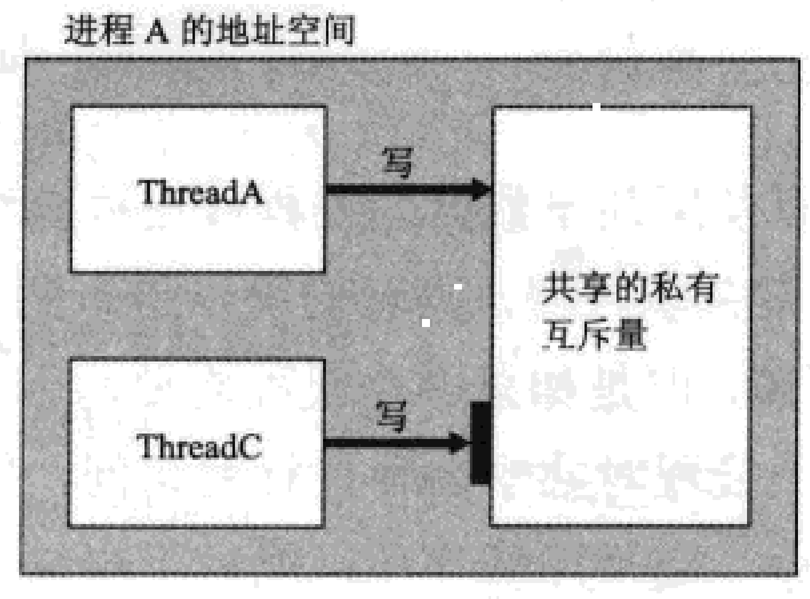
pthread_mutex_t 属性对象函数原型 #include <pthread.h>	描　述
类型	描述互斥量的行为，决定是否检测死锁、执行错误检查等；
int pthread_mutexattr_settype (pthread_mutexattr_t * attr,int type); int pthread_mutexattr_gettype (const pthread_mutexattr_t *restrict attr,int *restrict type);	设置并返回由 attr 指定的互斥量的 type 属性；type 包含下列值之一： PTHREAD_MUTEX_DEFAULT PTHREAD_MUTEX_RECURSIVE PTHREAD_MUTEX_ERRORCHECK PTHREAD_MUTEX_NORMAL

在不同进程的线程之间使用 pthread 互斥量要求属性是共享的。这个属性决定了互斥量是私有的或是共享的。私有的互斥量仅在同一个进程的线程间共享。它们可以被声明为全局的或被声明为可以在线程间传递的句柄。共享的互斥量被所有有权访问互斥量内存的线程所使用，这包括来自不同进程的线程。为此，使用 pthread_mutexattr_setshared( )并将属性设置为 PTHREAD_PROCESS_SHARED，如下所示：

```
pthread_mutexattr_setpshared( & MutexAttr,PTHREAD_PROCESS_SHARED);
```

这样就可以允许 Mutex 被不同进程的线程所共享。

图 7-11 比较了私有的互斥量和共享的互斥量的概念。如果不同进程的线程要共享互斥量，则这个互斥量必须在进程间共享的内存中分配。我们已经在本章前面的部分中讨论了共享内存。进程间的互斥量可以用来保护访问文件、管道、共享内存和设备的临界区。



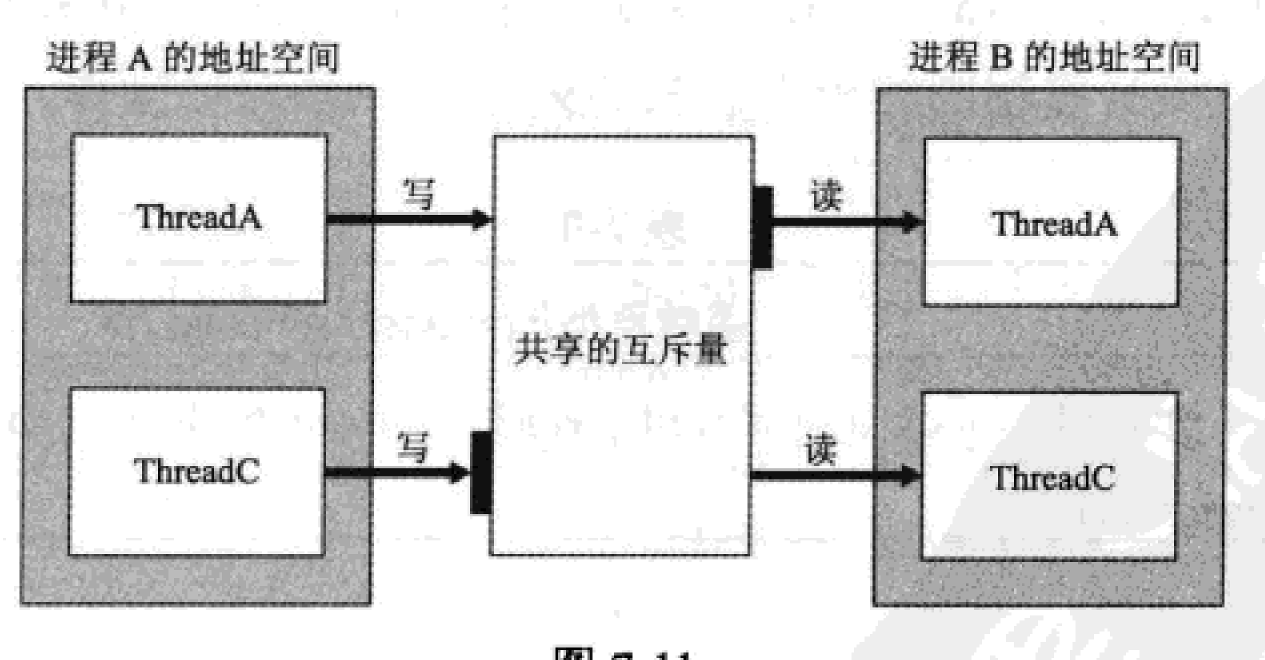


图 7-11

使用互斥量信号量来管理临界区

可以使用互斥量来管理进程和线程的临界区，以控制竞争条件。互斥量是通过将对临界区的访问串行化来避免竞争条件的。示例 7-8 显示了在程序清单 7-5 中定义的新任务的代码。使用互斥量来保护它们的临界区。

示例 7-8

```
// Example 7-8 New code for tasks in Listing 7-5.

 3  void *task1(void *X)
 4  {
 5     pthread_mutex_lock(&Mutex);
 6     Answer = Answer * 32; //critical section
 7     pthread_mutex_unlock(&Mutex);
 8     cout << "thread A Answer = " << Answer << endl;
 9
10  }
```

在示例 7-8 中，task1 在改动全局变量 Answer 时使用互斥量。在第 8 行，任务将 Answer 的新值发送到 cout，这是任务的临界区。除了每个任务将它的线程名发送到 stout 之外，还可以让每个任务执行相同的代码。因此，得到如下输出：

```
Before threads Answer = 10
thread 1 Answer = 320
thread 2 Answer = 160
thread 3 Answer = 165
After threads Answer = 165
```

2. 读-写锁

互斥量信号量串行化临界区。只有使用共享数据的线程或进程能够进入到临界区。使用读-写锁时，如果多个线程只是要读取共享内存，则它们可以进入到临界区。因此，任意数目的线程可以拥有用于读取的读-写锁，但是如果多个线程将要写入或更改共享内存，则只有一个线程会被允许访问。如果某个线程对共享内存进行写访问，则不允许其他线程进入临界区。如果应用程序有着多个线程，则互斥量互斥可能过于极端。应用程序的性能能够从允许多个读取中受益。POSIX 标准定义了类型为 pthread_rwlock_t 的读-写锁。类似于互斥量信号量，读-写锁有着相同的操作。表 7-9 列出了读-写锁操作。

表 7-9

读-写锁操作	函数原型#include <pthread.h>
初始化	int pthread_rwlock_init(pthread_rwlock_t *restrictrwlock, const pthread_rwlockattr_t *restrict attr);

(续表)

读-写锁操作	函数原型#include <pthread.h>

请求占有权	#include <time.h>
	int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
	int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
	int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock, const struct timespec *restrict abs_timeout);
	int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock, const struct timespec *restrict abs_timeout);

释放占有权	int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
尝试占有权	int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock); int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
销毁	int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

常规互斥量同读-写互斥量的区别在于它们的加锁请求操作。常规互斥量有一个加锁请求操作，而读-写互斥量有两个加锁请求操作：

```
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
```

pthread_rwlock_rdlock( )获得一个读锁，pthread_rwlock_wrlock( )获得一个写锁。如果线程请求一个读锁，只要没有其他线程持有写锁，则会得到读锁。如果有其他线程持有写锁，则调用线程会被阻塞。如果线程请求一个写锁，则只有在没有其他线程持有写锁或读锁时，才会被允许，否则调用线程被阻塞。

读-写锁的类型是 pthread_rwlock_t。这种类型也拥有一个属性对象，该对象封装了它的属性。属性函数列在表 7-10 中。

表 7-10

pthread_rwlock_t 属性对象函数原型 #include <pthread.h>	描　述
int pthread_rwlockattr_init(pthread_rwlockattr_t * attr);	使用默认值初始化由attr指定的读-写锁属性对象的所有属性，这些属性根据具体实现定义
int pthread_rwlockattr_destroy(pthread_rwlockattr_t * attr);	销毁由 attr 指定的读-写锁属性对象；可以通过调用 pthread_rwlockattr_init( )被再次初始化
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t * attr, int pshared); int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr,int *restrict pshared);	设置或返回由 attr 指定的读-写锁属性对象的进程共享属性；pshared 参数包含如下值之一： PTHREAD_PROCESS_SHARED 允许读-写锁被有权使用为读-写锁分配的内存的所有线程所共享，即是这些线程属于不同的进程； PTHREAD_PROCESS_PRIVATE 读-写锁被同被初始化的 rwlock 相同的进程中的线程共享

pthread_rwlock_t 可以是线程间私有的，或是不同进程的线程间共享的。

使用读-写锁来实现访问策略

读-写锁可被用于实现 CREW 访问策略。可以允许多个任务并发读取，但是只允许一个任务进行写入访问。使用读-写锁可以阻止并发读取和独占写入的共同发生。示例 7-9 包含了使用读-写锁来保护临界区的任务。

示例 7-9

```
// Example 7-9 Threads using read-write locks.
//...

pthread_t ThreadA,ThreadB,ThreadC,ThreadD;
pthread_rwlock_t RWLock;

void producer1(void *X)
{
   pthread_rwlock_wrlock(&RWLock);

  //critical section
   pthread_rwlock_unlock(&RWLock);
}

void producer2(void *X)
{
   pthread_rwlock_wrlock(&RWLock);
  //critical section
   pthread_rwlock_unlock(&RWLock);
}

void consumer1(void *X)
{
   pthread_rwlock_rdlock(&RWLock);
  //critical section
   pthread_rwlock_unlock(&RWLock);
}

void consumer2(void *X)
{
   pthread_rwlock_rdlock(&RWLock);
  //critical section
   pthread_rwlock_unlock(&RWLock);
}

int main(void)
{

   pthread_rwlock_init(&RWLock,NULL);
   //set mutex attributes
```

```
    pthread_create(&ThreadA,NULL,producer1,NULL);
    pthread_create(&ThreadB,NULL,consumer1,NULL);
    pthread_create(&ThreadC,NULL,producer2,NULL);
    pthread_create(&ThreadD,NULL,consumer2,NULL);
    //...
    return(0);
}
```

在示例 7-9 中，创建了 4 个线程。两个线程(ThreadA 和 ThreadC)是生产者，两个线程(ThreadB 和 ThreadD)是消费者。所有线程都有一个受到读-写锁(RWLock)保护的临界区。ThreadB 和 ThreadD 可以并发地或串行地进入它们的临界区，但是如果 ThreadA 或 ThreadC 位于它们自己的临界区中，则 ThreadB 和 ThreadD 均不能够进入临界区。ThreadA 和 ThreadC 不能够同时进入它们的临界区。表 7-11 显示了这个程序的部分决策表。

表 7-11

ThreadA(写入者)	ThreadB(读取者)	ThreadC(写入者)	ThreadD(读取者)
N	N	N	Y
N	N	Y	N
N	Y	N	N
N	Y	N	Y
Y	N	N	N

面向对象的互斥量类

程序清单 7-11 是一个面向对象的互斥量类的声明。

程序清单 7-11

```
// Listing 7-11 Declaration of an object-oriented mutex class.


 1  #ifndef _PERMIT_H
 2  #define_PERMIT_H
 3  #include <pthread.h>
 4  #include <time.h>
 5  class permit{
 6  protected:
 7     pthread_mutex_t Permit;
 8     pthread_mutexattr_t PermitAttr;
 9  public:
10     permit(void);
11     bool available(void);
12     bool not_in_use(void);
13     bool checkAvailability(void);
14     bool  availableFor(int secs,int nanosecs);
```

```
15  };
16
17
18  #endif/*  _PERMIT_H */
```

permit 类为互斥量类提供了基本操作。程序清单 7-12 包含了 permit 类的定义。

程序清单 7-12

```
// Listing 7-12 Definition of the permit class.

 1  #include "permit.h"
 2
 3
 4  permit:: permit(void)
 5  {
 6     int AValue,MValue;
 7     AValue = pthread_mutexattr_init(&PermitAttr);
 8     MValue = pthread_mutex_init(&Permit,&PermitAttr);
 9  }
10  bool permit::available(void)
11  {
12     int RC;
13     RC = pthread_mutex_lock(&Permit);
14     return(true);
15
16  }
17  bool permit::not_in_use(void)
18  {
19     int RC;
20     RC = pthread_mutex_unlock(&Permit);
21     return(true);
22
23  }
24  bool permit::checkAvailability(void)
25  {
26     int RC;
27     RC = pthread_mutex_trylock(&Permit);
28     return(true);
29  }
30  bool permit::availableFor(int secs,int nanosecs)
31  {
32     //...
33     struct timespec Time;
34     return(true);
35
36  }
```

程序清单 7-12 只显示了基本操作。该类如果要成为一个完全可行的互斥量类，必须加入错误检查，就像本章前面部分的示例 7-3 中的 posix_queue 类那样。

下面是程序清单 7-11 和程序清单 7-12 的程序概要 7-5。

程序概要 7-5

程序名：

permit.h (程序清单 7-11)
permit.cc (程序清单 7-12)

描述：

清单 7-11 包含了 permit.h 的开始部分，程序清单 7-12 包含 permit.cc。

必需的库：

`libpthread`

必需的头文件：

`< pthread.h > < time.h > “permit.h”`

编译和链接指令：

`c++ -c permit.cc`

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

N/A

注释：

无

3. 条件变量

互斥量通过控制对共享数据的访问来同步任务。条件变量可以根据数据的值来同步任务。条件变量是当一个事件发生时发送信号的信号量。一旦事情发生，可能有多个进程会线程在等待信号。条件变量通常用于对操作的顺序进行同步。

条件变量的类型为 pthread_cond_t。下面是它能够执行的操作的类型：

- 初始化(Initialize)

- 销毁(Destroy)
- 等待(Wait)
- 计时等待(Timed wait)
- 发信号(Signal)
- 广播(Broadcast)

其中，初始化操作和销毁操作同其他互斥量的工作方式类似。等待和计时等待操作将调用者挂起，直到另一个进程/线程在条件变量上发送信号。计时等待允许您指定线程等待的时间段。如果在指定的时间内没有收到信号，则线程被释放。条件变量和互斥量共同使用。如果一个线程或进程试图对互斥量加锁，则会被阻塞，直到互斥量被释放。一旦解除阻塞，它就会获得互斥量并继续执行。如果使用一个条件变量，则必须同一个互斥量关联。某个任务等待信号，另一个任务发信号或广播该信号已经发生。表 7-12 列出了为条件变量定义的基本操作。

表 7-12

条件变量操作	函数原型/宏 #include <pthread.h>
初始化	int pthread_cond_init(pthread_cond_t *restrictcond, const pthread_condattr_t *restrict attr); pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
发信号	int pthread_cond_signal(pthread_cond_t *cond); int pthread_cond_broadcast(pthread_cond_t *cond);
销毁	int pthread_cond_destroy(pthread_cond_t *cond);

某个任务试图对互斥量加锁。如果互斥量已经被加锁，那么任务将被阻塞。一旦解除阻塞，任务释放互斥量并同时等待条件变量的信号。如果互斥量没有被加锁，则它释放互斥量并等待，等待是无限期的。使用计时等待时，任务只等待指定的时间段。如果在任务被通知之前，等待的时间超过了指定的时间段，则函数返回一个错误。它接下来将获得互斥量。

发信号操作使得某个任务用信号通知另一个线程或进程某个事件已经发生。如果某个任务在等待一个条件变量，它会被解除阻塞并被赋予互斥量。如果有多个任务等待条件变量，则只有一个任务被解除阻塞。这些任务在队列中等待并根据调度策略解除阻塞。广播操作发信号通知所有等待条件变量的任务。如果多个任务解除阻塞，则任务根据调度策略竞争对互斥量的占有权。同等待操作相反，发信号的任务不要求占有互斥量，尽管这是被推荐的。

条件变量也有一个属性对象。表 7-13 列出了属性对象的函数以及简要的描述。

表 7-13

pthread_cond_t 属性对象函数原型 #include <pthread.h>	描　　述
int pthread_condattr_ init(pthread_condattr_t * attr);	用默认值初始化由 attr 指定的条件变量属性对象的所有属性，这些属性由具体实现定义
int pthread_condattr_destroy (pthread_condattr_t * attr);	销毁由 attr 指定的条件变量属性对象；可以通过调用 pthread_condattr_init( )再次初始化
int pthread_condattr_setpshared (pthread_condattr_t * attr,int pshared); int pthread_condattr_getpshared (const pthread_condattr_t * restrict attr, int *restrict pshared);	设置或返回由 attr 指定的条件变量的进程共享属性；pshared 包含如下值之一： PTHREAD_PROCESS_SHARED 允许条件变量被任何有权访问为条件变量分配的内存的线程共享，即使线程位于不同的进程中； PTHREAD_PROCESS_PRIVATE 条件变量在同被初始化的 cond 相同的进程的线程间共享
int pthread_condattr_setclock (pthread_condattr_t *attr,clockid_t clock_id); int pthread_condattr_getclock(const pthread_condattr_t * restrict attr,clockid_t *restrict clock_id);	设置并返回由 attr 指定的条件变量属性对象的 clock 属性；clock 是用于度量 pthread_cond_timedwait( )函数的超时服务的时钟的时钟 id；clock 的默认值是系统时钟

使用条件变量管理同步关系

条件变量和互斥量可被用来实现本章前面提到的同步关系：

- start-to-start(SS)
- finish-to-start(FS)
- start-to-finish(SF)
- finish-to-finish(FF)

这些关系可以在相同进程的线程间或不同进程的线程间存在。程序清单 7-13 包含了一个程序来示范如何实现 SF 同步关系。每个例子中使用了两个互斥量。一个互斥量被用来同步对共享数据的访问，另一个互斥量和条件变量共同使用。

程序清单 7-13

```
// Listing 7-13 SF synchronization relationship implemented with
// condition variables and mutexes.

 1  using namespace std;
 2  #include <iostream>
```

```
 3  #include <pthread.h>
 4
 5  int Number;
 6  pthread_t ThreadA,ThreadB;
 7  pthread_mutex_t Mutex,EventMutex;
 8  pthread_cond_t Event;
 9
10  void *worker1(void *X)
11  {
12     for(int Count = 1;Count < 10;Count++){
13        pthread_mutex_lock(&Mutex);
14        Number++;
15        pthread_mutex_unlock(&Mutex);
16        cout << "worker1: Number = " << Number << endl;
17        if(Number == 7){
18           pthread_cond_signal(&Event);
19        }
20     }
21
22     return(0);
23  }
24
25  void *worker2(void *X)
26  {
27     pthread_mutex_lock(&EventMutex);
28     pthread_cond_wait(&Event,&EventMutex);
29     pthread_mutex_unlock(&EventMutex);
30     for(int Count = 1;Count < 10;Count++){
31        pthread_mutex_lock(&Mutex);
32        Number = Number + 20;
33        cout << "worker2: Number = " << Number << endl;
34        pthread_mutex_unlock(&Mutex);
35
36     }
37
38     return(0);
39  }
40
41
42  int main(int argc, char *argv[])
43  {
44     pthread_mutex_init(&Mutex,NULL);
45     pthread_mutex_init(&EventMutex,NULL);
46     pthread_cond_init(&Event,NULL);
47     pthread_create(&ThreadA,NULL,worker1,NULL);
48     pthread_create(&ThreadB,NULL,worker2,NULL);
49
50     pthread_join(ThreadA,NULL);
51     pthread_join(ThreadB,NULL);
52
```

```
53    return (0);
54  }
```

在程序清单 7-13 中，实现了 SF 同步关系。ThreadB 不能够开始，直到 ThreadA 结束。一旦 Number 的值为 7，ThreadA 发信号给 ThreadB，接下来它可以继续执行直到结束。ThreadB 不能够开始它的计算，直到它从 ThreadA 接收到一个信号。两个线程都使用了 EventMutex 和条件变量 Event。Mutex 被用来同步对共享数据 Number 的写访问。任务可以使用多个互斥量来同步不同临界区和同步不同的事件。这些技术可以很容易地用在进程的执行顺序的同步上。

下面是程序清单 7-13 的程序概要 7-6。

程序概要 7-6

程序名：

program7-13.cc. (程序清单 7-13)

描述：

program7-13.cc(程序清单 7-13)具有 SF 同步关系。ThreadB 不能够开始，直到 ThreadA 结束。一旦 Number 的值为 7，ThreadA 发信号给 ThreadB，接下来它可以继续执行直到结束。ThreadB 不能够开始它的计算，直到它从 ThreadA 接收到信号。两个线程都将 EventMutex 和条件变量 Event 一同使用。Mutex 被用来同步对共享数据 Number 的写访问。ThreadA 和 ThreadB 将 Number 发送到 stdout。在循环的每次迭代中，ThreadA 将 Number 的值加 1，ThreadB 将 Number 的值加 20。

必需的库：

```
Libpthread
```

必需的头文件：

```
< iostream > < pthread.h >
```

编译和链接指令：

```
c++ -o program7-13 program7-13.cc -lpthread
```

测试环境：

Solaris 10、gcc 3.4.3 和 3.4.6

处理器：

Opteron 和 UltraSparc T1

执行指令：

```
./program7-13
```

注释：

无

4. 线程安全的数据结构

有了本章讨论的同步基础，使得构建被多个线程安全地并发使用的复杂数据结构成为可能。数据结构可以包含用于保护对内部数据的访问的互斥量类。

7.3 线程策略方法

线程策略决定了在将应用程序线程化时可能使用的方法。方法决定了被线程化的应用程序如何将它的工作委派给任务以及通信如何进行。策略提供了线程化的方法并帮助决定访问策略。

线程的目的是代表进程执行任务。如果进程有着多个线程，每个线程执行应用程序将要完成的任务中的一些子任务。线程是根据特定的策略和方法被赋予工作的。如果应用程序对一些过程或实体进行建模，那么选择的方法应当反映该模型。

常见的模型包括：

- 委托(boss-worker)
- 对等(peer-to-peer)
- 流水线(pipeline)
- 生产者-消费者(producer-consumer)

每个模型有着自己的工作分解结构(WBS)，WBS 决定了每块软件完成什么任务，例如，谁负责线程创建以及在什么条件下创建线程。

注意：

WBS 已经在第 3 章中详细讨论过。

对于集中式的方法，存在一个单独的进程/线程来创建其他进程/线程，并给它们分配工作。装配线(assembly line)方法在相同数据上执行不同阶段的工作。一旦这些进程/线程被创建，它们可以在不同数据集上执行相同的任务、在相同数据集上执行不同的任务、在不同的数据集上执行不同的任务。线程可以被分类，每个类别的线程仅执行特定类型的任务。可以有一组线程只执行计算，而另外的线程处理输入或产生输出。

还要注意的一点就是建模的对象可能在贯穿应用程序或进程时不是同类的，在这种情况下有必要在不同的阶段采用不同的模型。一个模型可以嵌入到另一个模型中。使用流水线模型时，线程或进程可以创建其他线程或进程并局部利用委托模型来处理该阶段的数据。

7.3.1　委托模型

在委托模型中，一个线程(boss)创建其他的线程(worker)并给每个线程指派任务。在能够继续执行代码之前，boss 线程可能必须等待，直到每个 worker 线程完成任务。boss 线程的代码可能会基于 worker 线程的结果。boss 线程是通过指定一个函数来委托每个 worker 线程将要执行的任务的。当每个 worker 线程都被分配到任务，由每个 worker 线程负责执行该任务并产生输出，或者与 boss 线程或其他线程同步以产生输出。

boss 线程可以把创建线程作为对系统发出的请求的结果。对每种类型的请求的处理可以委派给 worker 线程。boss 线程执行一个事件循环。当事件发生时，创建 worker 线程并被赋予它们的职责。会为每个进入到系统的新的请求创建一个新的线程。使用这种方法可能会导致进程超出它的资源或线程限制。另一种方法是让 boss 线程创建一个线程池，这些线程可以再分派到新的请求上。boss 线程在初始化期间会创建很多线程，然后这些线程挂起，直到有请求加入到它们的队列。当请求放置到队列中时，boss 线程发信号给一个 worker 线程，让它来处理请求。当线程结束后，它让下一个请求出列。如果没有可用的请求，则线程将自己挂起，直到 boss 发信号告诉它队列中有待处理的工作。如果所有的 worker 线程共享一个队列，那么这些线程只处理特定类型的请求。每个线程的结果被放置到另一个队列中。boss 线程的主要用途是：

- 创建所有的线程
- 将工作放置到队列中
- 当有工作需要处理时，唤醒 worker 线程

worker 线程的主要用途是：

- 检查队列中的请求
- 执行被指派的任务
- 如果没有可供处理的工作，则将自己挂起

所有的 worker 线程和 boss 进程并发地执行。示例 7-10 包含了这种方法的委托模型的事件循环伪代码。

示例 7-10

```
// Example 7-10 Skeleton program for delegation model where boss creates a
// pool of threads.

pthread_t Thread[N]

// boss thread
{

   pthread_create(&(Thread[1]...taskX...);
   pthread_create(&(Thread[2]...taskY...);
   pthread_create(&(Thread[3]...taskZ...);
   //...
```

```
    pthread_create(&(Thread[N]...?...);

    loop while(Request Queue is not empty
       get request
       classify request
       switch(request type)
       {
           case X :
                    enqueue request to XQueue
                    broadcast to thread XQueue request available

           case Y :
                    enqueue request to YQueue
                    broadcast to thread YQueue request available

           case Z :
                    enqueue request to ZQueue
                    broadcast to thread ZQueue request available
            //...
        }

    end loop
}

void *taskX(void *X)
{
   loop
      waiting for signal
      when signaled
      loop while XQueue is not empty
         lock mutex
         dequeue request
    release mutex
         process request
         set mutex
         enqueue results
         release queue
      end loop
   until done
}

void *taskY(void *X)
{
   loop
      waiting for signal
      when signaled
      loop while YQueue is not empty
         lock mutex
         dequeue request
    release mutex
```

```
            process request
            set mutex
            enqueue results
            release queue
         end loop
      until done
   }

   void *taskZ(void *X)
   {
      loop
         waiting for signal
         when signaled
         loop while ZQueue is not empty
            lock mutex
            dequeue request
      release mutex
            process request
            set mutex
            enqueue results
            release queue
         end loop
      until done
   }

   //...
```

在示例 7-10 中，boss 线程创建 *N* 个线程。每个任务与处理用 taskX、taskY、taskZ 表示的请求类型关联。在事件循环中，boss 线程从请求队列中出列一个请求，判断请求的类型，然后将请求入列到适当的请求队列。它对线程进行广播，告诉它们一个请求在某个特定队列中可用。函数还包含一个事件循环。线程被挂起，直到从 boss 线程接到队列中有请求的信号。一旦被唤醒，在内部循环中，线程处理队列中所有的请求，直到队列为空为止。它从队列中移出一个请求、处理它，然后将结果放置到结果队列。会对输入队列和输出队列使用互斥量。

7.3.2　对等模型

同委托模型中有一个 boss 线程来对 worker 线程委派任务相反，在对等模型中，所有的线程有着平等的工作状态。有一个线程会最初创建执行所有任务所需要的全部线程，但是该线程仍被认为是一个 worker 线程，它不会进行工作的委派。worker(对等)线程具有更为局部的职责。对等线程可以处理被所有线程共享的一个输入流中的请求，或者每个线程有着自己负责的输入流。输入还可以被保存在文件或数据库中。对等线程可能会进行通信并共享资源。示例 7-11 包含了对等模型的伪代码。

示例 7-11

```
// Example 7-11  Skeleton program using the peer-to-peer model.
//...
pthread_t Thread[N]

// initial thread
{

   pthread_create(&(Thread[1]...taskX...);
   pthread_create(&(Thread[2]...taskY...);
   pthread_create(&(Thread[3]...taskZ...);
   //...
   pthread_create(&(Thread[N]...?...);
  }

void *taskX(void *X)
{
   loop while (Type XRequests are available)
         set mutex
         extract Request
         unlock mutex
         process request
         lock mutex
         enqueue results
         unlock mutex
   end loop
   return(NULL)
}

//...
```

7.3.3 生产者-消费者模型

在生产者-消费者模型中，生产者线程负责产生数据，这些数据将会被消费者线程消费。数据保存在被生产者线程和消费者线程间共享的内存块中。生产者线程必须产生数据，然后消费者提取它。如果生产者线程存放数据的速度同消费者线程消费数据的速度相比过快，则生产者线程可能会在消费者线程提取数据之前多次重写之前的结果。另一方面，如果消费者线程提取数据的速度同生产者存放数据的速度相比过快，则消费者线程可能会提取同样的数据或者试图提取还未存放的数据。因此，这个过程要求同步。我们在本章前面部分讨论了读-写锁，并包含了一个生产者进行写入而消费者进行读取的实例。示例 7-12 包含了生产者-消费者模型的伪代码。生产者-消费者模型在大规模程序和应用中，也被称作客户端-服务器模型。

示例 7-12

```
// Example 7-12 Skeleton program using the producer-consumer model.

/...

// initial thread
{
    pthread_create(&(Thread[1]...producer...);
    pthread_create(&(Thread[2]...consumer...);
    //...
  }


void *producer(void *X)
{
   loop
      perform work
        lock mutex
         enqueue data
        unlock mutex
         signal consumer
      //...
   until done
}

void *consumer(void *X)
{
   loop
      suspend until signaled
      loop while(Data Queue not empty)
          lock mutex
           dequeue data
          unlock mutex
           perform work
          lock mutex
          enqueue results
          unlock mutex
      end loop
   until done
}
```

7.3.4　流水线模型

流水线模型可以通过装配线方法来进行刻画，其中产品流被分阶段进行处理。在每个阶段，由一个线程在输入单元上进行工作。当单元经过流水线中的所有阶段之后，则对输入的处理就已经完成，并退出系统。这种方法允许多个输入被同时处理。一旦数据已经在某个阶段被处理完毕，则它就即将处理流中的下一个数据。每个线程负责产生它的中间结

果或输出，并使得它们对于流水线中的下一个阶段可用。最后的阶段或线程产生流水线的结果。

当输入沿着流水线移动时，可能在特定阶段有必要缓冲输入单元，因为线程正在处理前一个输入。如果某个特定阶段的处理比其他阶段要慢，则可能会导致流水线的减速，从而导致积压。为了防止积压，有必要为那个阶段创建额外的线程来处理到来的输入。这是混合模型的例子。在流水线中的这个阶段，线程创建一个委托模型来处理它的输入并防止积压。

流水线中的工作阶段应当加以平衡，使得某个阶段不会比其他阶段花费更多的时间。工作应当在流水线中平均地分布。流水线中还可以加入更多的阶段以及因此而加入更多的线程，这也防止了积压。示例 7-13 包含了流水线模型的伪代码。

示例 7-13

```
// Example 7-13 Skeleton program using the pipeline model.
//...

pthread_t Thread[N]
Queues[N]

// initial thread
{
   place all input into stage1's queue
   pthread_create(&(Thread[1]...stage1...);
   pthread_create(&(Thread[2]...stage2...);
   pthread_create(&(Thread[3]...stage3...);
   //...
  }

void *stageX(void *X)
{
     loop
     suspend until input unit is in queue
     loop while XQueue is not empty
        lock mutex
        dequeue input unit
        unlock mutex
        perform stage X processing
        enqueue input unit into next stage's queue
      end loop
    until done
    return(NULL)
}

//...
```

7.3.5 用于线程的 SPMD 和 MPMD

在并发模型中，线程可能会在不同的数据集上反复执行相同的任务，或者可能会在不

同的数据集上执行不同的任务。图7-12显示了并行化的不同模型。并发模型利用单指令多数据(SIMD)或多程序多数据(MPMD)。它们是并行化的两种模型，根据指令和数据流对程序进行分类。它们可被用于描述线程模型并行实现的工作类型。为了这个讨论的目的，最好将MPMD定义为MTMD(Multiple Threads Multiple Data，多线程多数据)。这个模型描述了一个系统，该系统执行不同的线程来处理不同数据集或数据流。在图7-12(a)中，可以看到线程1处理数据集1，线程2处理数据集2。同样地，出于这个讨论的目的，最好将SIMD(也被称作单程序多数据，或SPMD)重新定义为单线程多数据(Single Thread Multiple Data，STMD)。这个模型描述了一个系统，该系统执行一个线程来处理不同数据集或数据流。在图7-12(b)中，线程1执行例程A并处理数据集1，线程2也执行例程A，但是处理数据集2。这意味着多个同样的线程执行相同的例程，但是处理的是不同的数据集。多线程单数据(Multiple Threads Single Data，MTSD)描述了一个系统，系统中不同的指令应用到相同的数据集。在图7-12(c)中，线程1执行例程A，线程2执行例程B，两个线程都处理相同的数据集，即数据集1。单指令单数据(Single Instruction Single Data，SISD)描述了的系统中只有一个指令处理一个数据集。在图7-12(d)中，线程1执行例程A，它顺序地处理数据集。

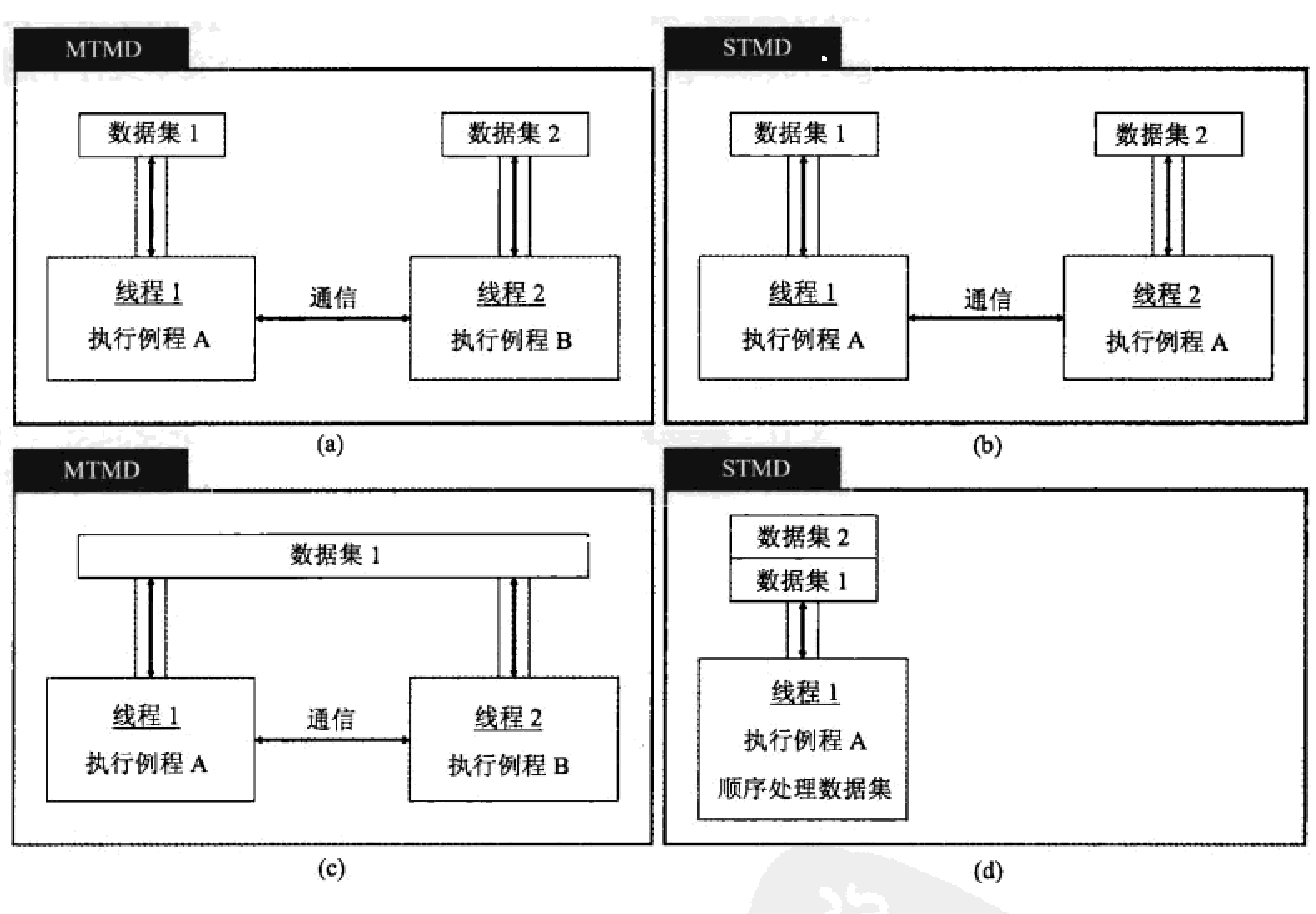


图7-12

委托模型和对等模型都使用STMD或MTMD并行化模型。如前所述，线程池可以执行不同的例程来处理不同的数据集。这种方法利用了MTMD模型。也可以将相同的例程交给线程池来执行。提交到系统的请求或作业可能是不同的数据集，而不是不同的任务。这种情况下，会有一个线程集合执行相同的指令，但是作用在不同的数据集上，这样就使

用了 STMD。对等模型可以是执行相同或不同任务的线程。每个线程可以有它自己的数据流或每个线程都要处理的多个数据文件。流水线模型使用了 MTMD 并行模型。在每个阶段，执行不同的处理，因此多个输入单元是出于不同的完成阶段。如果在每个阶段执行相同的处理，则流水线隐喻将会失去价值。

7.4 工作的分解和封装

此刻，我们已经讨论了并发执行的任务之间的通信与协作，无论这些任务是进程还是线程。我们还讨论了通信关系以及 IPC 和 ITC 通信机制。还介绍了任务协作、内存访问模型、同步关系、数据和通信的同步，以及很多可用来防止竞争条件的技术。并发模型可用来规划通信和工作委托的方法。下面我们希望使用这些技术和模型来做一些工作。

7.4.1 问题陈述

我们有大量的文本文件需要进行过滤。这些文本文件必须经过过滤，才能够用于我们的自然语言处理(Natural Language Processing，NLP)系统。我们希望从多个文本文件中删除指定的 token 或字符，字符例如 “,”、“.”、“?” 和 “!”]，而且我们希望能够实时地完成。

可以立即确定的对象是：

- 文本文件
- 待删除的字符
- 过滤后的结果文件

7.4.2 策略

我们有超过 70 个文件需要处理。每个文件中可能包含成百上千行的文本。为了简化起见，我们假定希望从中删除某个字符集。我们希望能够指定文件的名称、令程序过滤出所有不希望的字符、让程序创建新的过滤过的文件、能够给程序提供下一个文件。在本书前面使用过的 guess_it 示例中，使用了一种方法来将文件分解成较小的任务。任务是在文件中搜索一个编码，文件非常大，有超过 400 万的编码需要进行搜索，因此我们将文件分解成较小的文件，并针对这些较小的文件进行搜索。由于存在时间约束，搜索必须并行执行。

对于这里的问题，被过滤的文件必须与将不期望的字符删除后的最初文件相同。文本不能够被更改。尽管可以将文件分解成多个较小的文件，但是将它们重新组装起来不是我们希望做的。因此我们应当采用什么方法呢？记住下面的内容才是我们的目标：

- 删除所有多余的字符是必须要做的
- 实时地完成工作
- 保持每个文件中的内容的完整性

我们可以从整个文件中删除字符，或者每次从每行删除字符。下面是可行的方法。

- 方法 1：在文件中查找一个字符。当找到该字符后将它删除，然后查找这个字符的下次出现位置。当所有这些字符都被删除之后，再次针对下一个多余的字符对文件进行查找。为每个文件重复这个过程。可以满足后置条件，因为我们是在最初的文件上进行查找，并从中删除多余的字符。
- 方法 2：从每个文件中删除某个字符的所有出现位置，然后为每个多余的字符重复这个过程。后置条件也会被满足，原因与方法 1 相同。
- 方法 3：读进一行文本，删除一个多余字符。检查该行文本并删除下一个多余字符，依此类推。当从该行文本中删除所有的字符之后，将过滤后的文本行写入到新的文件中。对每个文件进行这样的操作。后置条件可以被满足，因为我们会在进行过程中重新构造一个新的文件。当一行被处理之后，它就被写到新的文件中。
- 方法 4：基本与方法 3 相同，但是我们从一行文本中删除一个多余字符之后(可能会多次出现)就将它写入到文件或容器中。一旦整个文件被处理完毕，则为下一个字符再次进行处理。当最后一个字符被删除之后，文件就被过滤完毕。如果文本位于容器中，则现在可以写入到文件中。为每个文件重复这个过程。容器在重新构建文件时变得非常重要。

7.4.3　观察

在考虑每种方法时，我们看到对文件或对一行文本会进行多趟遍历。必须对此加以考虑，看看它对性能会产生怎样的影响。这个过滤必须实时完成。对于方法 1 和方法 2，对于字符的每次出现，都要重入文件。对每个多余的字符，都要遍历整个文件。一共有 4 个多余的字符，中间结果是删除一个字符之后的整个文件，然后是删除两个字符之后的整个文件，依此类推。对于方法 3 和方法 4，过滤的是一行文本。这样，中间结果是一行文本。在方法 3 中，一行文本可以很快地完成过滤。根据将要进行的处理的类型，一行文本可能会有用，也可能没有用处。等待整个文件是一种较长的等待。在每种方法中很明显可以看出，无论您处理的是整个文件还是一行文本，它们不是相互依赖的任务。一个文件的处理可以同其他文件独立地进行。这对于一行文本而言也是成立的。但是对于一行文本(记住文件的完整性必须维持)，执行的顺序必须同行在文件中出现的顺序相同。因此，文本行必须按照顺序过滤：第 1 行、第 2 行、第 3 行，等等。

7.4.4　问题和解决方案

我们将使用方法 3。基于观察得到的结果，我们得出这种方法将会最快地给出结果的结论，即便对于大的数据集也是如此，而且产生中间结果。现在我们可以开始考虑并发模型。模型帮助决定使用哪种通信和何种类型的协作来解决这个问题。一行应当在另一行企图开始之前被处理。这使我们想起流水线模型。在每个阶段，处理的同样是一行文本以及相同的待删除的多余字符。在各个阶段的最后，一行文本被完整地过滤。它可以被写入到一个文件。这将会非常快地完成，而且文件可以保持它的完整性。应当使用队列，因为它

们有着先进先出(FIFO)的访问顺序。这确保文本行保持相互之间的顺序。

每个阶段有一个输入队列和一个输出队列。输入队列是前一个阶段的输出队列。一旦文本已经被处理，它被写入到一个队列，下一个阶段从队列中获取文本行。当然，这要求同步。任务在前一个阶段已经将文本放入到队列之后才从队列中获取文本。由于对于任何队列的共享，只有两个任务涉及进来，因此互斥量就可以很好地工作。我们可以使用来自程序清单 7-10 和程序清单 7-11 的 permit 类。

下面是这个解决方案中的通信和协作需求：

- 队列要求同步。
- 需要 EREW 访问策略。
- 主 agent 组装第一个队列并创建线程(为每个阶段创建一个线程)。
- 将输入队列和输出队列以及一个互斥量传递给各个阶段。

7.4.5 流水线的简单 agent 模型实例

我们可以将这个问题的解决方案作为一个 agent 模型来讨论。每个阶段将会被一个 agent 来管理。队列是包含来自文件的文本行的列表。对于每个列表，给 agent 提供了一个许可来访问列表。这个解决方案中的新的对象是 agent、list 和 permit。图 7-13 显示了这个解决方案的类关系图。

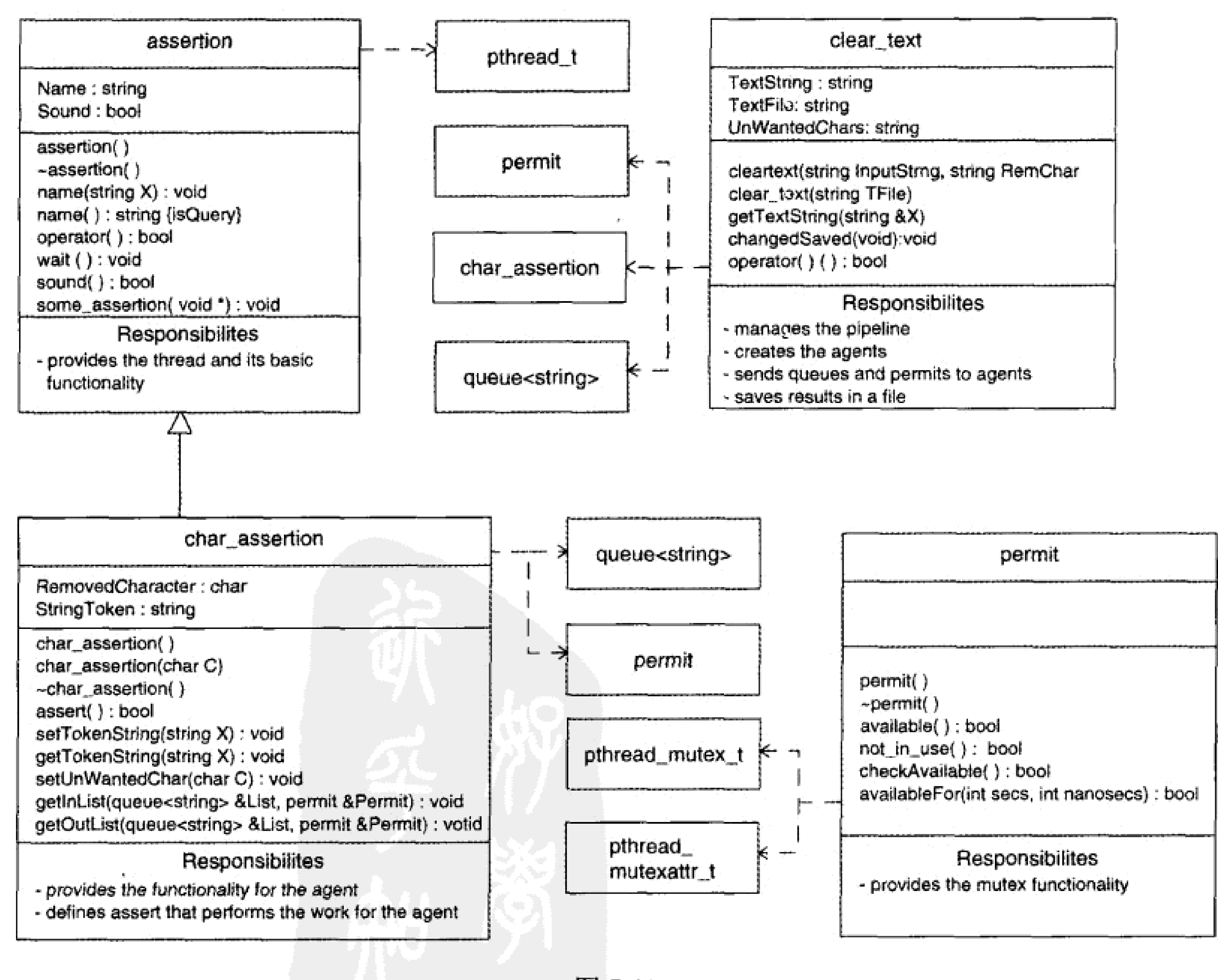


图 7-13

示例 7-14 是简单 agent 解决方案的主线。

示例 7-14

```
//Example 7-14 The main line for character removal agent.

 1  #include "clear_text.h"
 2  #include <iostream>
 3
 4
 5  int main(int argc, char** argv) {
 6
 7     if(argc != 2){
 8        cerr << "usage:  characters_removed FileName:" << endl;
 9        exit(0);
10     }
11     clear_text  CharactersRemoved(argv[1]);
12     if(CharactersRemoved()){
13        CharactersRemoved.changesSaved();
14        return (1);
15     }
16     return(0);
17  }
18
```

类型为 clear_text 的 CharactersRemoved 对象是主 agent，它管理流水线。待过滤的文件的名字是第二个命令行参数。CharactersRemoved( )执行流水线。如果它返回 false，意味着有一个 agent 失败，该 agent 负责删除的多余的字符可能还没有从文件或某些文本行中删除。changeSaved( )从最近的列表中得到结果(它包含所有过滤完毕的文本行)并将它们写入到文件中。

示例 7-15 包含了 operator( )方法。

示例 7-15

```
//Example 7-15 The pipeline method for the clear_text object.

 1  bool clear_text::operator()(void)
 2  {
 3     bool Sound = true;
 4     char_assertion  CharacterRemoved[4];
 5     CharacterRemoved[0].setUnWantedChar(',');
 6     CharacterRemoved[1].setUnWantedChar('.');
 7     CharacterRemoved[2].setUnWantedChar('?');
 8     CharacterRemoved[3].setUnWantedChar('\'');
 9
10     for(int N = 0; N < 3;N++)
11     {
12        CharacterRemoved[N].getInList(TextQ[N],Permit[N]);
```

```
13        CharacterRemoved[N].getOutList(TextQ[N+1],Permit[N+1]);
14     }
15
16     for(int N = 0; N < 4; N++)
17     {
18        CharacterRemoved[N]();
19     }
20
21     CharacterRemoved[3].wait();
22     CharacterRemoved[0].wait();
23     CharacterRemoved[1].wait();
24     CharacterRemoved[2].wait();
25
26     for(int N = 0; N < 4;N++)
27     {
28        Sound = Sound * CharacterRemoved[N].sound();
29     }
30     return(Sound);
31
32  }
```

在示例 7-15 中的第 4 行中，声明了 4 个 char_assertion agent。每个 agent 被传递了它将要从文件中删除的多余字符。第 10 行～第 14 行，for 循环将源列表以及该列表的 permit 还有输出列表及其permit传递给每个agent。第16行～第19行的for循环实际完成起动agent的工作。

operator( )、wait( )和 sound( )都定义在基类 assertion 中。示例 7-16 包含了 operator( )、wait( )和 sound( )在 assertion 类中的定义。

示例 7-16

```
//Example 7-16 The methods defined in the base class assertion.

 1  bool assertion::operator()(void)
 2  {
 3     pthread_create(&Tid,NULL,some_assertion,this);
 4     return(Sound);
 5  }
 6
 7  void assertion::wait(void)
 8  {
 9     pthread_join(Tid,NULL);
10  }
11
12
13  bool assertion::sound(void)
14  {
15     return(Sound);
16  }
```

PDG

在示例 7-16 中，可以看到在第 3 行中，assertion 类为 char_assertion agent 创建了线程。线程/agent 将要执行 some_assertion 函数。assertion 类是第 6 章中的 user_thread 类的改进版本，而其 some_assertion 是 do_something 方法。

示例 7-17 包含了来自 assertion 类的 some_assertion 方法，示例 7-18 包含了来自 char_assertion 类的 assert 方法。

示例 7-17

```
//Example 7-17 The some_assertion method defined in the base class assertion.

 1  void *  some_assertion (void * X)
 2  {
 3
 4     assertion *Assertion;
 5     Assertion = static_cast<assertion *>(X);
 6     if(Assertion->assert()){
 7        Assertion->Sound = true;
 8     }
 9     else{
10           Assertion->Sound = false;
11     }
12     return(NULL);
13
14
15  }
```

示例 7-18

```
//Example 7-18 The assert method defined in the class char_assertion.

 1  bool char_assertion::assert(void)
 2  {
 3
 4
 5     if(PermitIn.available()){
 6        TokenString = ListIn.front();
 7        ListIn.pop();
 8        remove(TokenString.begin(),TokenString.end(),RemovedCharacter);
 9        PermitIn.not_in_use();
10      }
11     if(PermitOut.available()){
12        ListOut.push(TokenString);
13        PermitOut.not_in_use();
14     }
15
16     return(true);
17  }
```

在示例 7-17 的第 6 行中，调用了 assert( )。这个方法是 agent 开始在流水线中进行它的

工作的位置。示例 7-18 包含了 assert( )的定义，它是 agent 的工作。如果 ListIn 列表的 PermitIn 可用，其中 ListIn 是源文本字符串，则将字符串弹出，在第 8 行中将多余的字符删除，然后将 PermitIn 释放。现在如果 PermitOut 可用，则新的字符串将会被压入到 ListOut。

这个示例显示了利用 MTSD 的并发模型(即流水线)的用法。一个文本字符串在流水线的每个阶段中进行处理，每个阶段将一个多余字符从字符串中删除。每个阶段的线程可以被指派给它自己占有的处理器内核。这是一个简单的处理，即将一个字符从字符串中删除，但是描述的过程(确定问题、分解问题和确定解决方法)可用于解决大规模的问题。我们可以做得更好。本章已经描述了多核编程的面向任务的软件分解。第 8 章将讨论针对大规模问题且有着大量内核需要管理时的声明式的面向谓词的分解。

7.5 小结

本章讨论了管理并发任务间的同步通信以及同步对全局数据、资源和任务执行的访问。还讨论了可用来委托工作并在运行在多个处理器内核的并发执行的任务间通信的并发模型。本章讨论了如下知识点：

- 依赖关系可用来检查哪个任务依赖于其他任务的通信或协作。依赖关系同这些并发任务间的协调通信和同步有关。如果相互依赖的任务之间的通信没有进行恰当的设计，那么会发生竞争条件。
- 进程间通信(IPC)是便于在进程之间通信的技术和机制。当一个进程发送数据给另一个进程时，或者通过操作系统 API 的方式使得另一个进程清楚某个事件时，要求使用 IPC。POSIX 队列、共享内存、管道、互斥量/信号量、条件变量等都是 IPC 的实例。
- 线程间通信(ITC)是为了方便驻留在所属进程的相同地址空间内的线程进行通信的技术，在考虑 ITC 时，需要处理的最为重要的问题是数据竞争和无限延期。
- 可以同步对数据、资源和任务执行的访问。当线程位于其临界区时，任务同步是必需的。临界区可以通过 PRAM 模型来管理，例如 EREW 和 CREW。一个进程中的两个任务或一个应用程序中的两个进程之间有 4 种基本的同步关系。
- 同步机制既可用于进程，也可用于线程。这些机制可以通过使用信号量和互斥量、读-写锁、条件变量来实现同步访问策略，从而被用来防止多个任务间发生竞争条件和死锁。
- 线程策略决定了在线程化应用程序时能够应用的方法。方法决定了分成线程的应用程序如何将它的工作委派给任务以及如何进行通信。策略提供了结构和方法来实现线程化，并帮助决定访问策略。

下一章将讨论并行应用程序设计层(Parallel Application Design Layer，PADL)。PADL 是一个用于设计要求并行编程的软件的 5 层分析模型。它被用来帮助组织软件分解。PADL 会被用于 SDLC(软件开发生命周期)的需求分析和软件设计活动中，SDLC 也将在第 8 章中进行介绍。

PADL 和 PBS：应用程序设计方法

第 4 章示范了操作系统对于具有并行化或多核支持需求的应用程序的作用。在该章中，我们解释了只有进程和内核线程是操作系统能够调度到处理器上执行的执行单元。在第 5 章和第 6 章中，我们解释了如何创建进程和线程以及如何使用它们来获得对芯片多处理器(CMP)的访问，但是进程和线程是相对较低级别的构造。问题仍旧存在，即“当存在并发需求时如何进行应用程序设计或设计模型中何时拥有可以并行操作的组件？”在第 7 章中，我们探究了支持进程间通信(IPC)的机制：互斥量、信号量和同步。这些也是低层构造，要想正确地编程，需要对操作系统级应用程序接口(API)和系统程序接口(System Program Interface)有着清楚的了解。

迄今为止，我们讨论的场景仅涉及双核或四核处理器，但趋势是 CMP 产品将会很快取代双核和四核处理器。第 2 章介绍的 UltraSparc T1 在一个芯片上有着 8 个核，而且每个内核有 4 个硬件线程。这使得它能够并发执行 32 个线程或进程。当前双核和四核 CMP 是最常见的配置，然而双核和四核配置将很快被八核 CMP 所替代，例如 UltraSparc T1。因此，当有着 32 个硬件线程可用时应当如何考虑应用程序设计？如果在一个 CMP 上有 64、128、256 个处理器又将如何设计？当可用硬件线程数目接近几百个时应当如何考虑软件分解？本章将致力于解决这些问题，讨论的内容有：

- 运行在多处理器内核的应用程序设计方法
- 应用程序设计的 PADL 和 PBS 方法
- 面向任务的软件分解与声明式的面向谓词的软件分解
- 知识源和多 agent 架构
- Intel 线程构建块(Thread Building Block，TBB)以及支持并发的 C++新标准

8.1 为大规模多核处理器设计应用程序

如果有着成百个或上千个并发执行的进程或线程，如何在应用程序设计中进行进程管理或实现 IPC 呢？在应用程序设计阶段考虑大量并发执行的任务是令人望而生畏的。

在这里我们宽泛地使用术语“应用程序设计”，对于不同类型和级别的软件开发人员，这个术语有着不同的含义。对于系统级程序员和应用程序级程序员，它分别有着不同的意思。此外，软件和计算机应用程序还有各种各样的分类。表 8-1 是摘录自美国计算机协会(Association of Computing Machinery，ACM)的 CCS(Computing Classification System)的很小一部分。

表 8-1

ACM 分类号	软　件	多　用　户	单　用　户
H.4.1	办公自动化	群件(groupware) 项目管理 时间管理	字处理 电子表格
H.4.3	通信应用程序	远程会议 电视会议 计算机会议	电子邮件 信息浏览器
D.2.2	软件开发工具	软件库	编辑器
D.2.3	编码工具和技术	软件代码维护	编译器 链接器
D.2.4	软件可靠性验证		
I.3.3	图像生成	高级可视化 虚拟环境	数字化和扫描 图形包
I.3.4	图形实用工具	虚拟现实系统	动画
I.3.5	3D 图形		3D 渲染

这个分类首先查看了 ACM CCS 中的 H 节和 D 节的软件类别。这些类别又进一步分组为多用户和单用户。在多用户应用程序中，两个或者多个用户能够同时访问软件应用程序的一些特性。从表 8-1 中可以看出，多用户应用程序的规模多种多样，从 Intranet 电视会议到基于 Internet 的源代码管理应用程序。很多数据库服务器、Web 服务器、电子邮件服务器等都是多用户应用程序的很好的示例。此外，尽管单用户应用程序不需要担心对某些特性的多用户访问，但是单用户应用程序也可能被要求并发执行多个任务。要求对音频和视频进行同步的单用户多媒体应用程序就是很好的示例。

此外，请看表 8-1 中应用程序的多样化(diversity)。在每个领域中的软件开发人员可能会在应用程序设计中采用不同的方法。将来，共同的一点是表 8-1 中多数应用程序分类将会运行在中等规模到大规模 CMP 上。表 8-1 中的分类仅仅是 ACM 对软件和计算的分类中的节选。此外，在 ACM CCS 中，仅提供了单一视图，即多用户视图。表 8-2 包含了计算机应用程序根据领域的 CCS 分类。这个分类来自 CCS 的 J 节。

表 8-2

(J.1)管理数据处理	(J.2)物理科学和工程
1. 商业	1. 航空航天
2. 教育	2. 考古学
3. 金融	3. 天文学
4. 政府	4. 化学
5. 法律	5. 地球和大气科学
6. 制造业	6. 电子学
7. 市场	7. 工程学
8. 军事	8. 数学和统计
	9. 物理
(J.3)生命和医药科学	**(J.4)社会和行为科学**
1. 生物遗传学	1. 经济学
2. 卫生学	2. 心理学
3. 医疗信息系统	3. 社会学
(J.5)艺术和人文	**(J.6)计算机辅助工程**
1. 建筑学	1. 计算机辅助设计
2. 艺术、造型与表演	2. 计算机辅助制造
3. 美术	
4. 语言翻译	
5. 语言学	
6. 文学	
7. 音乐	
8. 表演艺术	

当您从表 8-2 中所示的各种领域中考虑软件开发方法时，显然在应用程序设计阶段，每组会使用不同的概念。如果您要加入到多处理器计算机和并行编程技术的讨论中时，至少要弄清楚讨论的应用程序设计方法是针对哪个领域的。

因此，当您查看表 8-1 中的应用程序时，可能想象如何分解多用户应用程序要比如何分解单用户应用程序容易些。然而，一旦可用内核的数目到达特定阈值时，分解复杂性和任务管理仍然会成问题。随着可用内核数目的增加，使用过程性的自底向上的方法进行多线程和多处理将变得非常困难。作为开发人员，您将很快拥有廉价且广泛使用的 CMP，而且它们支持各种级别的并行。但是问题仍然存在：“如何进行多核应用程序设计才不会被可用的并行性搞得不知所措？”

根据表 8-1 中显示的分类节选和表 8-2 中显示的计算机应用程序的分集，很明显可以看出没有哪个工具、库、厂商解决方案、产品或指南能够成为所有问题的答案。相反，我们将与您共享一些作为软件工程师所使用的一些方法。尽管这些方法不是万用良方，但是它们足够通用，能够应用到表 8-1 和表 8-2 中的很多领域中。它们大部分是平台中立或厂商中立的，或者只是假定 ISO 标准 C++实现和 POSIX 兼容操作环境。本章提供的技术不是产品或厂商驱动的。这样做的一个原因就是没有哪一个厂商或产品能够产生仙丹妙药一样的解决方案，能够使得所有应用程序利用单一的芯片多处理器。

相反，就像您已经注意到的，我们将重心放在并行编程的范型转换，从命令式的面向任务的软件分解转移到声明式的面向谓词的分解。我们的建议来自我们作为软件工程师的经验，结合上很多来自面向对象软件工程和逻辑编程中的基本概念。本章中介绍的技术、模型、方法依赖于模态逻辑、模态逻辑的扩展和情境演算[Fagin 等人，1995]。第 5 章～第 7 章包含了主要和并行编程的命令式方法结合使用的底层操作系统原语及 POSIX API。我们的意图是为了说明这些相同的底层原语也可以被用在并行编程的声明式方法中。

在本章中，我们将解释当在最初软件开发请求中存在并发需求或者解决方案分解中显式或隐式地要求并行化时，如何实现应用程序设计的过程。我们将介绍并行应用程序设计层(Parallel Application Design Layer，PADL)。PADL 是一个 5 层分析模型，我们在 CTEST 实验室将它用于软件开发生命周期(SDLC)中的需求分析、软件设计、分解等活动中。我们使用 PADL 来将并发和并行性放置到适当的环境。PADL 被用于最初的问题和解决方案分解中。我们还使用 PADL 模型来规避并行编程的自底向上方法导致的复杂性。在本章中，我们提供了并行化的架构方法(architectural approach)，而不是面向任务的过程性方法。尽管并发执行的任务代表了利用 CMP 的应用程序中的基本工作单元，但是我们将这些任务放置到声明式架构以及基于谓词的软件模型中。本章将重点放在使用 SDLC 的基本原理来进行应用程序设计。

我们还将简要介绍谓词分解结构(Predicate Breakdown Structure，PBS)。软件设计的 PBS 提供了一种视角，将软件作为断言、命题和逻辑谓词的集合。PBS 给出了软件设计的声明式或基于谓词的视角。在 CTEST 实验室，我们使用 PADL 和 PBS 作为使用多线程、多处理或并行编程进行应用程序设计时的基本工具。PADL 和 PBS 与过程式及自底向上、面向任务的方法不同，但是它们是可以和接口类、应用程序框架、谓词、算法模板共同使用的工具，使得应用程序设计的过程能够利用中等规模到大规模单芯片多处理器的可用性。

8.2 什么是 PADL

如前所述，并行应用程序设计层(PADL)是一个用于要求并行编程的软件设计中的 5 层分析模型。PADL 用于帮助组织软件分解的工作。PADL 模型是一个精化模型(refinement model)。从顶层开始，下面的每一层都包含接近于操作系统和编译器原语的更多的细节。

PADL 旨在用于 SDLC 的需求分析和软件设计等活动中。SDLC 的工业标准描述包含在 IEEE Std 1074，即 Guide for Developing Software Life Cycle Processes 中。IEEE Std 1074 帮助澄清了最小活动集合是什么。

注意：

第 3 章中的表 3-1 显示了 SDLC 中的一些常见活动，但它不是详尽的。

PADL 模型用来在书写软件之前帮助设计或分解工作模式。这就是为什么我们将重点放在它在 SDLC 的设计和分析活动中的使用的原因。图 8-1 显示了 PADL 的 5 个层。

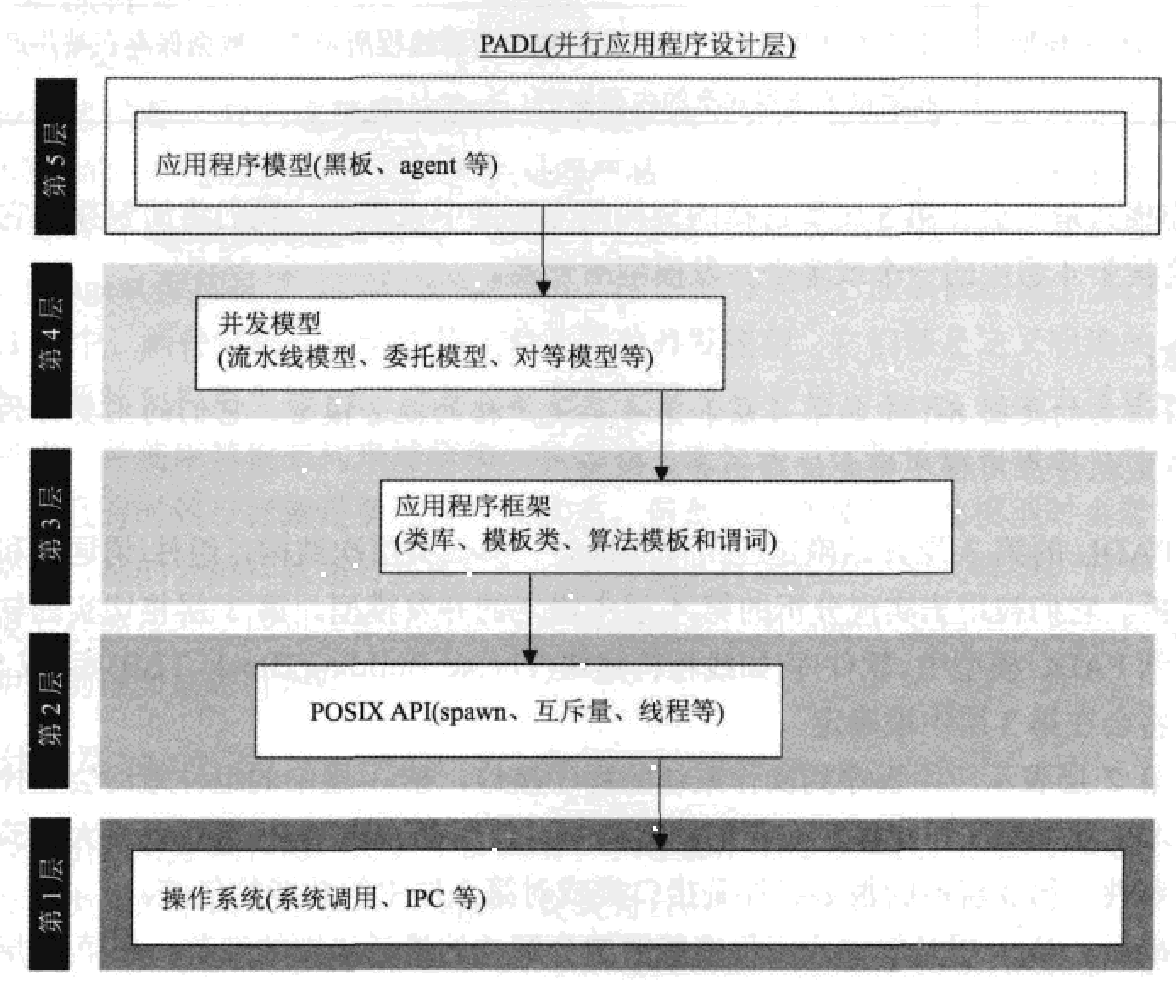


图 8-1

- 第 5 层是关于应用程序架构选择。应用程序的架构选择是最关键的决定之一，因为其他的任何事情都遵循架构，而且一旦决定之后，再进行改动的代价非常昂贵。架构在允许了特定的功能性和特性的同时防止了其他的功能性和特性。架构提供了应用程序的基本底层结构。
- 图 8-1 中的第 4 层确定了应用程序将要使用的并发模型。架构必须具有足够的灵活性来使用所需要的并发模型，同时并发模型应当具有足够的灵活性来支持所选择的架构。表 8-3 回顾了基本的并发模型以及它们的定义。

表 8-3

线 程 模 型	描　　述
Boss-worker 模型	一个中央进程/线程(boss)创建进程/线程(worker)，并为每个 worker 指派任务；boss 可能会等待每个 worker 结束它的任务
对等模型	所有进程/线程有着相等的工作状态，没有领导者；一个进程/线程创建执行任务所需要的所有 worker，但是不负责委派的任务；这些对等线程可能从被所有线程共享的一个输入流中处理请求，或者每个线程可以有它自己的输入流
流水线	使用装配线方法来按照阶段处理输入流；每个阶段是一个在输入单元上执行工作的线程；当输入单元通过所有阶段之后，则对输入的处理已经完成
生产者-消费者模型	生产者产生数据，这些数据被消费者线程所消费；数据保存在被生产者线程和消费者线程共享的内存块中

工作模式最初会在第 5 层所选择的架构的上下文中来表示。我们将简要探讨它。然后该架构会被第 4 层中的一个或多个并发模型所充实。

注意：

值得注意的是图 8-1 中的第 4 层和第 5 层是严格的概念模型。它们的主要目的是帮助描述和确定软件应用程序将会使用的并发架构。

- PADL 的第 3 层开始确定应用程序框架、模板类层次结构、组件/谓词库和算法模板，它们被用来实现分析的第 4 层中确定的并发模型。第 3 层将切实的软件引入到 PADL 模型中。软件库(如线程构建块(Thread Building Block，TBB)库，或 STAPL)将会在第 3 层中被确定。
- 第 2 层表示应用程序到操作系统的程序接口。第 3 层中的组件最终会和操作系统 API 发生交互和协作。在我们的案例中，使用 POSIX API 来获得最大的跨平台兼容性。第 3 层中的很多组件是接口类或对第 2 层中的功能的包装。
- 最后，第 1 层是软件应用程序能够被分解成的最低级别的细节。在第 1 层中，将会处理内核执行单元、信号、设备驱动等。第 1 层的软件执行第 2 层表示的应用程序接口的实际工作。第 1 层分解还包括编译器和链接器开关。

第 4 层和第 5 层被认为是设计层，而第 1 层～第 3 层则是实现层。第 3 层通常会为应用程序级软件开发所保留。第 1 层和第 2 层通常为系统级软件开发所保留。显然，第 2 层和第 3 层之间的界限可以很容易地跨跃。一旦进行了 PADL 模型分析，应用程序的并发结构对所有涉及软件开发中的人都是很清楚的。应用程序的 PADL 分析提供了 SDLC 测试活动的框架。PADL 分析提供了应用程序 SDLC 中的必要实现和部署目标。在 PADL 的最高层是设计概念和模型，在 PADL 的最底层是操作系统原语。这样，通过 PADL 模型，您能够对应用程序的并发有完整的认识。PADL 模型的目的是推动应用程序的声明式分解，因为您是从操作系统原语的架构出发，而不是从架构的原语出发。但是为了让 PADL 保证声

明式分解，它必须将架构限制在有着声明式、面向目标或基于谓词的语义的软件设计中。下一节将会详细查看 PADL 模型如何帮助您实现并行编程的声明式方法。

8.2.1 第 5 层：应用程序架构选择

内核、线程和进程的数目选择不是第 5 层中的分析的一部分。实际上，在第 5 层中，计算机都是不可见的。第 5 层所考虑的应用程序架构表示概念性或逻辑结构，这是与平台、操作系统和计算机无关的。这些架构的值是一些已知能够支持某些工作模板的架构。在 PADL 模型中，我们使用两种主要架构：

- 多 agent 架构
- 黑板架构

值得注意的是，我们这里所讨论的是问题解决方式和工作模式，而不是任何基于多 agent 和黑板的一些想法的特定工具或供应商产品。我们对术语 agent 和黑板的使用不应当同很多声称具有 agent 或黑板功能的广告噱头相混淆。我们使用的是 agent 和黑板的更正式的含义。多 agent 架构位于 ACM CCS 的第 I.2.11 节的分类中，而黑板则位于 ACM CCS 的第 D.2.11 节中。两种架构都能够支持大量不同的并发模型。它们都是定义明确的，而且很容易理解。两种架构都支持 PBS 的概念。我们已经选择使用多 agent 架构来捕获面向目标的工作模式，并使用黑板架构来捕获面向状态的工作模式。尽管目标和状态有时候可以互换使用，而且有时候目标就是要达到特定状态，但是我们在使用并发模型时会对它们进行区分。基于目标的分解和状态转移能够帮助您得到并行编程的声明式模型，而并行编程的声明式模型能够帮助您应对大规模并行芯片多处理器。多 agent 架构和黑板架构可以通过使用声明式的语义来进行理解。

1. 什么是 agent

当最初引入对象编程时，关于什么构成一个对象存在很多争议。在什么构成一个 agent 方面也存在类似的争议。很多人将 agent 定义为自治的、持续执行的和代表用户行事的程序。然而，这个定义能够应用到某些 Unix 守护进程乃至一些设备驱动上。还有些人加上了 agent 必须有着用户特殊的知识、必须执行在驻留有其他 agent 的环境中、必须只在特定环境中发生作用等判定标准。这些判定标准将排除被其他人认为是 agent 的程序。例如，很多 E-mail agent 单独行动，并且可以在多个环境下发挥功能。除了 agent 的判定标准，agent 社区中各个组还引入了类似软件机器人(softbot)、知识机器人(knowbot)、软件代理(software broker)、智能对象(smart object)等术语来描述 agent。agent 的一个常见的定义是："能够在有其他处理发生并存在其他 agent 的环境中持续地、自治地发挥功能的实体"。

尽管接受这个定义并继续前进是很有诱惑力的，但是我们不能够这样做，因为它太容易描述其他种类的软件构造。更为正式的资料，即 FIPA(the Foundation for Intelligent Physical Agents)规范，将术语 agent 定义为："agent 是一个域中基本的行动者，它将一种或多种服务能力合并到统一且集成的执行模型中，该模型可以包含对外部软件、人类用户和通信设

备的访问。”

尽管这个定义给人感觉更加结构化，但是仍需要进一步地澄清，因为很多服务器(有些是面向对象的，而有些则不是)也符合这个定义。这个定义包括过多类型的程序和软件构造，使得它变得不是很有用。在我们的 PADL 模型中，我们将使用 agent 的 5 部分的定义，就像[Luger,2002]中所陈述的：

(1) agent 是自治的或半自治的。也就是说，在问题解决过程中，每个 agent 具有一定的职责，而且无需过多了解其他 agent 做什么或者怎么做。每个 agent 完成问题解决中它自己的独立片段，要么自身产生一个结果，要么将它的结果报告给 agent 集合体中其他的 agent。

(2) agent 是位于某地的。每个 agent 对它的周边环境敏感，而且(经常是)不了解所有 agent 构成的完整领域。

(3) agent 是交互的。也就是说它们构成了个体的集合，相互协作以完成特定任务。从这个角度而言，它们可以被看作是一个“社会”。

(4) agent 的社会是有结构的。在多数面向 agent 的问题解决中，每个个体尽管有着自己独特的环境和技能集合，但是会在整体问题解决中同其他 agent 协作。这样，最终的解决方案不能够仅被看作是集合的，同时也是协作的。

(5) 尽管 agent 个体被看作拥有一组技能职责，但是 agent 社会共同协作的结果要比个别 agent 的贡献的总和大。

2. 什么是多 agent 架构

多 agent 架构，是指由两个或多个 agent 组成的、必要时可以并发执行的、基于 agent 的架构。从 Luger 对 agent 的定义中可以清楚地看出，多 agent 架构可用作工作模式的极为灵活的解决方法，这些工作模式要求并发或可从并发中受益。注意在 agent 的这种定义中，没有提到特定计算机性能、处理器数目、线程管理等，对涉及的 agent 的数目也没有限制。对 agent 集合能够执行的工作模式没有复杂度约束。如果您是以多 agent 架构的计划开始的，那么将能够处理任意规模的并发。

3. 从问题陈述到多 agent 架构

在软件开发中，分解是主要的挑战。当软件开发中要求进行并行编程时，这个挑战更为艰巨。我们针对这个挑战的解决方法是找出适当的问题模型和适当的解决方案模型。使用 PADL 分析，我们基于解决方案模型来选择应用程序架构。为了更好地了解其工作原理，请再次考虑第 4 章中的猜测编码的问题。

回顾第 4 章中的游戏场景，如下所示：我正在想着一个包含 6 个字符的编码，这个编码包含的字符可以重复，同时，编码只能够包含来自数字 0～9 和字符 a～z 的任意字符。您的任务是猜出我脑海中的编码。在游戏中，时间限制被设置为 5 分钟。如果您能够在 5 分钟之内猜到我所想的编码，则您获胜。在第 4 章中，我们计算出您需要从 4 496 388 种可

能的编码中进行选择。

现在，在这个简单的例子中，我们希望使得游戏更加具有挑战性。现在您只有 2 分钟来进行猜测。除此之外，我的忠实助手将初始编码交给我，而不是由我来编出这个编码。如果我的助手确定您在 15 秒之内已经做出了超过 *N* 次不正确的猜测，则他会给我一个新的编码，而且新的编码保证是在您已经做出的猜测之内的。

策略

回顾第 4 章，使用了一个文件来包含所有可能的 6 字符编码。编码总数是由总的对象数目 *n* 和采样大小 *r* 所决定。可能性的计算公式如下：

```
( n -1 + r )! / ( n -1)! r !
```

在本例中，*n* 为 36，而 *r* 为 6。最初的策略是将文件分为 4 部分，并对每一部分并行进行搜索。这样做的想法是最长的搜索所需要的时间也只是搜索文件中四分之一的时间。但是由于现在的新时间约束是 2 分钟，必须更改策略为将原始文件分割成 8 部分，各部分并发进行搜索。因此，应当能够在搜索八分之一个文件的时间内正确地猜到编码。您还可以做出另外的预防措施。如果能够搜索所有的 8 个文件，而且没有正确地猜到编码并且没有超时，可以将文件分成 64 份并再次尝试。如果搜索失败而且仍有时间，可以将文件分成 128 份并再次尝试。

观察

如果仔细查看要被猜测的编码，您将看到如果助手确定在 15 秒时间间隔内做出了不正常次数的猜测，那么助手会将编码改变到已经猜测过的编码。并未指定什么是不正常的次数，因此您的策略是在更短的时间间隔内做出更多的猜测，因为时间约束更紧了。此外，如果能够遍历所有可能的编码，而仍未得到正确的编码，则可以推测出编码在 2 分钟的时间间隔内发生了改变，因此，需要试着增加猜测的速度，使得它比编码被改变的速度更快。

多 agent 下的问题模型和解决方案模型

通过问题陈述，可以很轻易地看出最初是同 3 个 agent 打交道。如果用 agent 的术语来改写游戏，则得到如下结果：Agent A 为 Agent B 提供编码，Agent C 试着猜测 Agent B 的编码。如果 Agent C 在 15 秒之内提出了过多的猜测，则 Agent A 将会为 Agent B 提供一个新的编码，而且保证这个编码是 Agent C 已经猜过的。如果 Agent C 能够遍历所有的可能，而且仍未被宣布获得胜利，同时 Agent C 还有剩余时间，则 agent 会以更快的速度进行同样的猜测。Agent C 意识到为了能够生成足够的猜测来确保在指定时间期限内成功，它需要得到帮助，因此，Agent C 招募了一队 agent 来帮助它提出猜测。对于对全部可能性的每趟遍历，Agent C 招募更大的 agent 队伍来帮助它进行猜测。

显然，您现在可以开始考虑如何将 agent 映射到线程或进程，但是我们将会抵制这种诱惑。在这个级别上，还不应当考虑线程和进程。在 PADL 模型方法中，是在第 5 层和第 4 层制订应用程序的完整而且正确的逻辑陈述，在进行到第 4 层之前，如果您是在将 PADL 模型应用到游戏场景，则要精化所有涉及的 agent 之间的关系和交互。要了解所有假设和 agent 运转的条件，并给出多 agent 交互的完整的描述。图 8-2 是一幅 UML 活动图，表示了多 agent 社会中的交互。

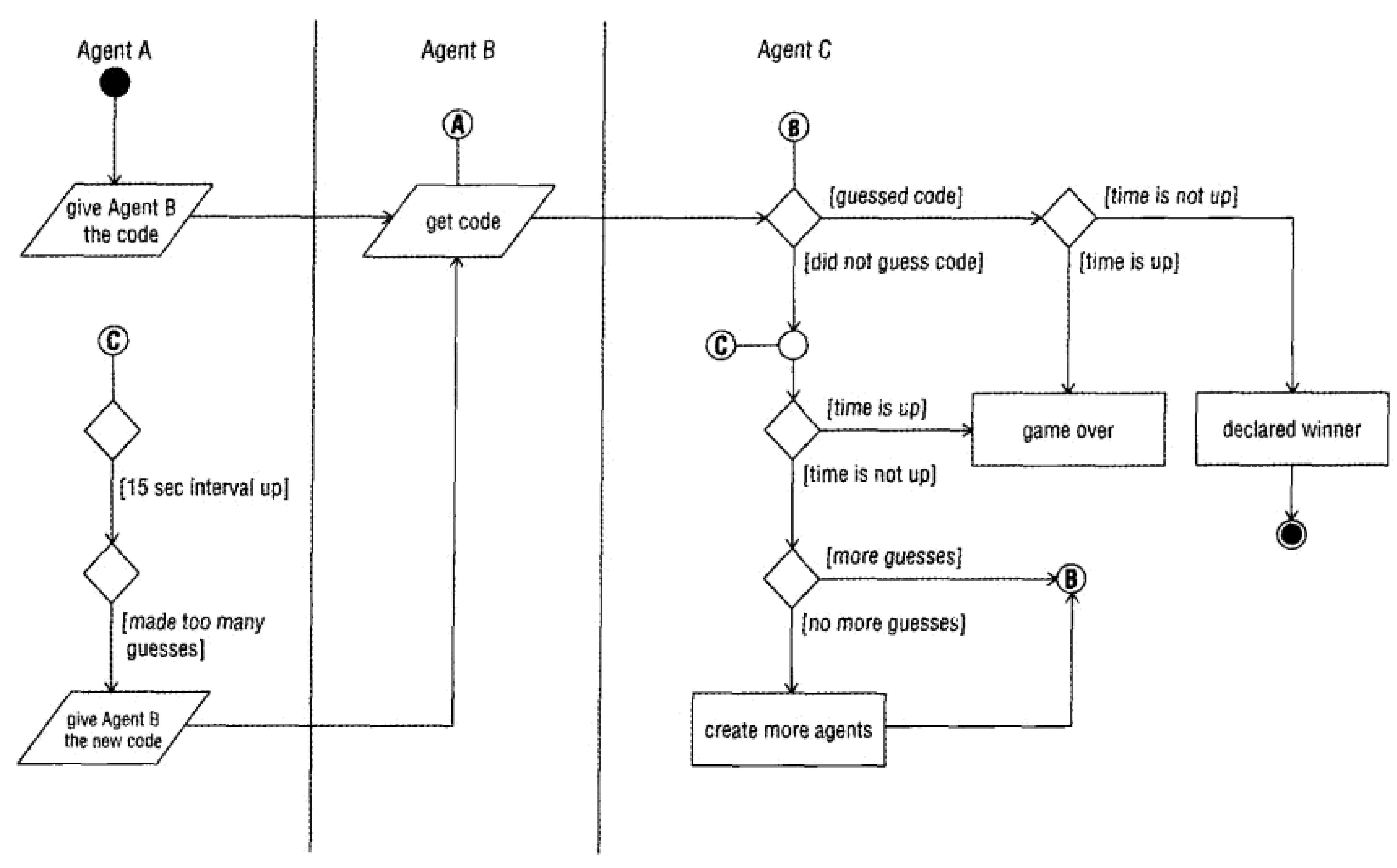


图 8-2

4. 黑板架构

黑板模型是一种协同解决问题的方法。黑板用来在两个或多个基于软件的问题解决者之间进行记录、协调和沟通。在黑板模型中主要有两种类型的组件：

- 黑板
- 问题解决者

黑板是一个集中的对象，每个问题解决者都能够使用它。问题解决者可以对黑板进行读取并改变黑板的内容。黑板的内容在任何指定时间将是不同的。黑板的最初内容将会包括要解决的问题。

除了要解决的问题之外，其他表示问题初始状态、问题约束、目的、总体目标等的信息也可以包含在黑板上。随着问题解决者逐步对问题进行解决，中间结果、假设和结论都会记录在黑板上。一个问题解决者写到黑板上的中间结果可能会作为另外一个问题解决者阅读黑板的催化剂。试验性的解决方案会被发布到黑板上。如果解决方案被判定为不够充

分，这些解决方案会被删除，然后提出其他的解决方案。问题解决者使用黑板而不是在相互之间直接传递局部结果和发现。在某些配置中，黑板扮演仲裁者，通知问题解决者何时达成了解决方案或何时开始或结束工作。黑板是一个活动的对象，而不仅仅是一个存储区域。在某些情况中，黑板决定哪些问题解决者参与进来以及接受或拒绝什么内容。黑板可以组织问题解决者的增量式结果或中间结果，还可以转换或解释来自一组问题解决者的工作，从而使得它可以被另外一组问题解决者所使用。

问题解决者是一种软件，通常具有在某些方面或问题领域具有专门的知识或处理能力。问题解决者可以就是一个非常简单的将摄氏温度转换到华氏温度的例子，或者是非常复杂的智能 agent，用来处理医疗诊断。在黑板模型中，这些问题解决者被称作知识源(knowledge sources，KS)。为了使用黑板来解决问题，需要两个或多个知识源，而且每个知识源通常有着特定的关注领域或专长领域。如果问题可以被分成隔离的任务，而且这些任务可以独立或半独立地完成，则黑板是很好的选择。在基本的黑板配置中，每个问题解决者应付问题中不同的部分。每个问题解决者只看到问题中它所熟悉的那一部分。如果问题任何部分的解决方案依赖于总体解决方案或者是对问题其他部分的不完全的解决方案，那么黑板可用于协调问题解决者以及不完全解决方案的集成。

黑板的问题解决者不需要是同构的。每个问题解决者可以使用不同的技术来实现。例如，某些问题解决者可能是使用面向对象的技术实现的，而其他的问题解决者可能通过函数来实现。此外，问题解决者可以使用完全不同的问题解决范型。例如，解决者 A 可能使用反向链接方法(backward chaining approach)来解决问题，而解决者 B 可能使用反向传播(counterpropagation)方法。同时黑板的问题解决者也不要求一定使用相同的编程语言来实现。

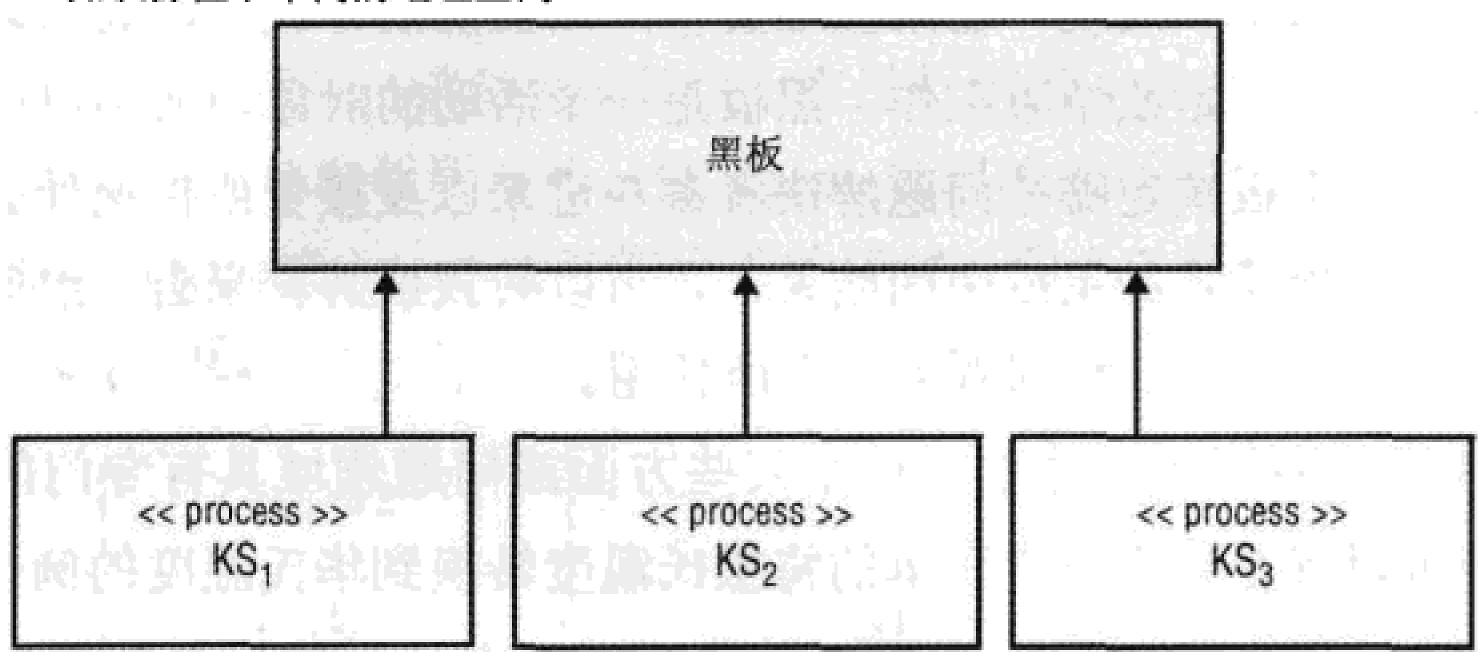
黑板模型没有指定黑板的任何特定结构或布局，也没有建议知识源应当如何构建。在实践中，黑板的结构是同问题相关的，知识源的实现同样根据待解决的问题而不同。黑板框架是一个概念模型，没有描述黑板以及知识源的结构，只是描述了关系。黑板模型并不规定知识源的数目或目的。黑板可以是一个全局对象，或者是组件位于多个计算机上的分布式对象。黑板系统甚至可以由多个黑板组成，每个黑板专用于最初问题的一部分。这使得黑板成为问题解决中的一个非常灵活的模型。黑板模型支持并行编程以及很多并发模型。黑板可以分成单独的部分，允许多个知识源对它们进行并发访问。黑板可以很容易地支持并发读互斥写(CREW)、互斥读互斥写(EREW)和多指令多数据(MIMD)。知识源可以同时执行，每个知识源工作在问题中对应的部分上。

图 8-3 显示了黑板的两种内存配置。

在图 8-3 所示的两种情况中，所有知识源均可访问黑板。这种配置为问题的解决提供了非常灵活的模型。

黑板内存配置 1：

知识源位于不同的地址空间



黑板内存配置 2：

知识源共享相同的地址空间

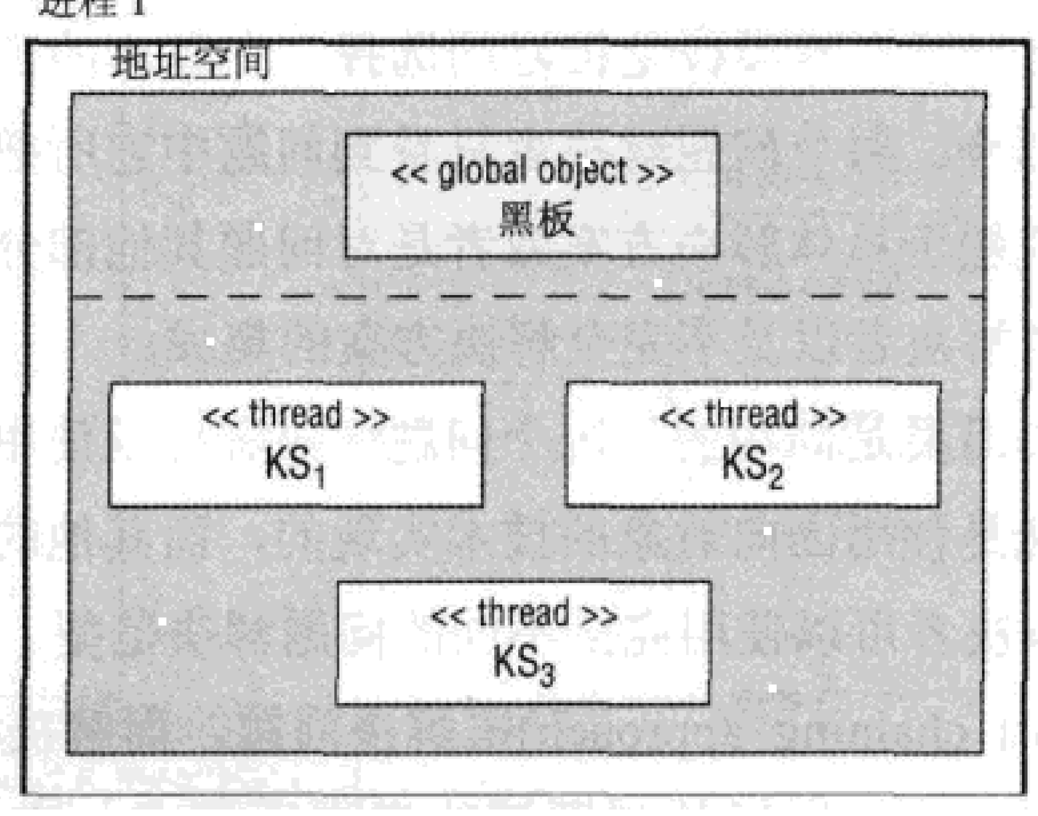


图 8-3

5. 设计黑板的方法

如同我们已经介绍的，设计黑板的方式不止一种，但是多数黑板有着特定的共同特性和属性。黑板的初始内容通常包含待解决问题的解空间的某种划分。解空间包含问题的所有局部解和完整解。

“问答浏览器”黑板示例

假定您需要一个特定领域信息的浏览器，也就是说该浏览器允许用户仅在某个狭小主题或特定主题范围内搜索信息(例如，卡通、机器人、YouTube 等)。用户应当能够使用图形界面来浏览可用信息或选择高级搜索特性。高级搜索特性使得用户能够简化问题的类型(只要它属于该主题)，而且浏览器应当基于它所访问的信息返回一个完整的答复。我们会假定多数用户倾向于高级搜索，因为它的速度更快。

并行性在哪里？黑板在哪里？

在上面的浏览器的要求中，可以看到没有提到多核、并发线程或进程。只是简单地假定浏览器将会有着让用户感觉可以接受的性能。您现在立即就会面对软件开发中的主要挑

战(尤其是对 CMP 部署)之一：分解。分解从理解问题开始，然后为该问题设计一个解决方案。在本例中，使用图形用户接口来浏览信息是相当容易理解的。另一方面，允许用户输入一个问题，拥有可以理解用户的问题的软件，在信息中查找，找到一个适当的答案，然后用一种可以接受的格式来呈现它构成了另外一个挑战。这个问题涉及判定问题是用哪种语言来提出的(也许您可以假定是本国的语言)。因此，对问题的快速浏览得出了如下的挑战：

- 问题所使用的语言是软件所了解的吗？
- 问题所针对的主题是软件的信息中已有的吗？
- 问题是否被清楚而且无歧义地陈述了？
- 问题中是否有未知的或拼错的单词？(如果出现了怎么回答？)
- 软件如何应对问题中的含义？
- 语言的语法(包括句法、语义、词法和语用)是什么？

经过一段时间的深思熟虑之后，您会意识到及时(如 1 秒之内)地接受、理解、恰当地答复输入到信息浏览器中的随机问题是非常具有挑战性的问题。经过研究之后会发现要求使用自然语言处理技术。您会在计算语言学和计算语义学中找到部分解。现在您对问题就有了一个粗糙但是比较完整的认识。

解决方案的组成

为了提供问题的解决方案，您会意识到需要包括解析器、单词专家、句法分析、语义分析、语用分析等内容的软件组件。结果是可以在多个层次上同时分析用户最初的问题。确定问题所使用的语言可以同识别未知的单词或拼错的单词同时进行。将问题分成不同部分的句法分析也可以同时进行。还可以确定一旦问题的句法部分解决了，则可以开始进行语义分析过程。类似地，一旦问题的部分语义解决了，语用(上下文及含义)的识别可以开始。您已经确定了问题分成哪些部分。尽管它们之间存在一些部分依赖(例如，在句法和语义之间以及语义和语用之间)，但是在分析用户的问题时，大部分的分析均可并发进行。

在第 5 层上观察 PADL 分析模型，显然您既可以使用多 agent 架构，也可以使用黑板架构作为解决方案模型。在这里选择黑板架构，因为语义和语用分析可以在部分解的情况下开始工作，而且黑板架构非常适合增量式问题解决。

浏览器程序的知识源

这样，您将最初的用户问题放在黑板上，每个知识源均有其专门的任务：

- 为未知的或拼错的单词查找字典
- 将问题分解为不同词性(名词、动词、疑问句等)
- 理解词法，即单词形式(复数、现在时和过去时)
- 单词的意思和使用的语义理解
- 语用分析(单词在上下文中的使用)

负责检查问题中未知或拼错的单词的知识源将它的候选放在黑板上。当问题出现之

PDG

后，词法知识源做出可能由于复数、过去式、缩写词等方面的修正。它们可以将修正后的单词放回到黑板上。在这些事情发生的同时，句法知识源将问题中修正之后的单词按照词性进行分割，并将短语放回到黑板上。一旦黑板上有一些是语义和语用知识源可以做的，则它们就读取黑板，并用一种搜索引擎可以使用的形式将用户最初的问题根据潜在的意思提出来。

这是一种并发处理的协作类型，因为部分解可以被不同知识源在很多级别上使用和解释。每个知识源在黑板上写下正在处理的试验性的假设。语义知识源发现的内容可能会帮助句法解析器精化它的工作。句法解析器精化的内容可能会为语用知识源澄清一些疑点，诸如此类。黑板表示一个解空间，它被分成部分解构成的层级结构，最顶部是适当形式的问题和答案，而最底层则是词性和词形。如果解空间的某个部分符合语法或语言用法中的规则，则该部分的解将会作为部分解写入到黑板中的另一块区域。某个 KS 可能会将动词短语写在黑板上，另一个可能会将上下文的选择写在黑板上。一旦这两个信息已经写在黑板上了，另外一个 KS 可以使用这些信息来帮助识别问题的实际主题或目标。所有这些必须在几秒之内发生。

黑板是好的选择吗？

问答浏览器要求的最初朴素陈述并没有真正地暗示并发，您现在已经做出了一个解决方案，它需要使用并行工作模型来满足假定的速度要求。黑板解决方案非常适合于利用不完整或阶段性信息开展工作的知识源的并发。在这个级别上，还没有考虑到线程、内核数目或进程。

黑板作为迭代的共享解空间

解空间有时候是按照层次来组织的。在问答处理实例中，有效的问题分类将会位于层次结构的最顶层，而下一层可能由各种分类的视图组成。例如，根据提问的不同形式，包括谁提出问题、提出什么问题、在哪里提出问题以及提出问题的时间，每一层描述了问题分类的更小但是可能稍不明显的方面(例如，是及物动词吗)。知识源可能同时在层级结构的多个层上工作。解空间也可以组织成一张图，其中每个节点表示解的某些部分，每条边表示两个部分解之间的关系。解空间可以通过一个或多个矩阵来表示，矩阵的每个元素包含一个解或部分解。解空间表示是黑板架构的一个重要组成。问题的本质经常会决定解空间应当如何划分。这个特性在 PADL 模型中非常关键，因为我们使用第 5 层来描述应用程序架构，而且应用程序架构必须足够灵活，以映射到解模型。黑板的结构支持这种灵活性。

除了解空间组件之外，黑板通常有一个或多个规则(启发式)组件。规则组件用来决定部署哪个知识源，以及接受或拒绝哪些解。规则组件来可以用来将部分解从解空间层级结构中的一层传递到另外一层。规则组件还可能被用来按优先顺序排列知识源方法。规则组件支持多个知识源之间的并发。这一层是需要使用并行处理的声明式解释来应对并发的位置。有些知识源可能会进入到死胡同。黑板取消选择一组知识源，以支持另一组知识源。黑板可能会使用规则组件来给知识源提供建议，该建议基于已经生成的部分假说来给出更

加合适的潜在的假说。

除了解空间组件和规则组件，黑板还经常包含初始值、约束值以及辅助目标。在某些情况下，黑板包含一个或多个事件队列，用于捕捉来自问题空间或知识源的输入。图 8-4 显示了基本黑板架构的逻辑布局。

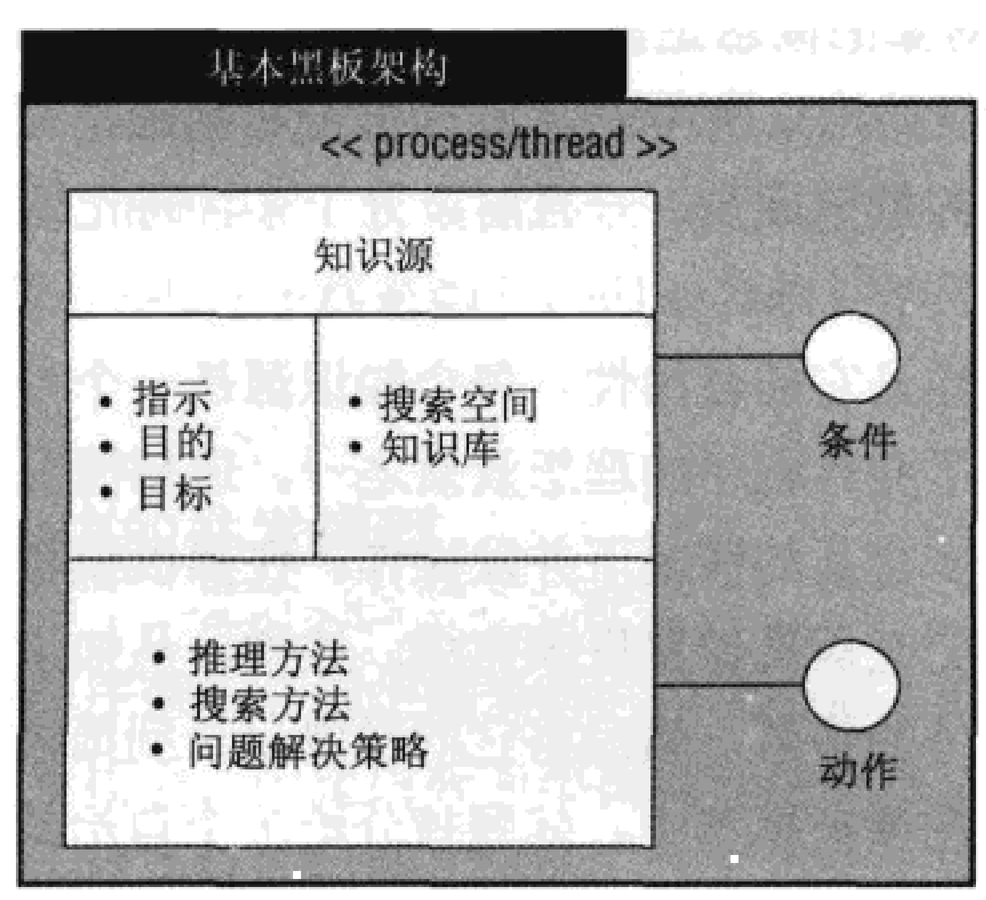


图 8-4

图 8-4 显示出黑板包含多个部分。图 8-4 中的每个部分都有多种实现。这说明黑板不仅仅是全局内存块或传统数据库。尽管图 8-4 显示了多数黑板所具有的常见核心组件，但是黑板架构不仅限于这些组件。其他有用的黑板组件包括问题的上下文模型以及可用于帮助问题解决器在解空间中导航的域模型。C++所具有的对面向对象设计和编程的支持很好地适合了黑板模型的灵活性要求。多数黑板架构可以使用 C++中的类来建模。回想一下，类可以用来对人、位置、事物或思想来建模。黑板用来解决涉及人、位置、事物或思想的问题。因此，使用 C++类来对黑板包含的对象或实际的黑板进行建模是一种合适的选择。我们将在黑板模型的实现中利用 C++容器类的优点以及标准算法。

6. 知识源的剖析

知识源可通过对象、过程、规则集、逻辑验证，乃至整个程序来表示。知识源包括一个条件部分和一个动作部分。当黑板包含的一些信息满足了某些知识源的条件部分，则激活了该知识源的动作部分。Robert Englemore 和 Tony Morgan 在他们的著作 *Blackboard Systems* 中清晰地定义了知识源的责任：

注意：

每个知识源负责了解在什么条件下它可以为解做出贡献。每个知识源具有前置条件，它决定了黑板上的哪些条件必须存在，然后才会激活知识源的主体部分。您可以将知识源视作一个大的规则。规则和知识源之间的主要区别在于所持有的知识的粒度大小。这个大的规则的条件部分被称作知识源前置条件，而动作部分被称作知识源主体[Englemore, Morgan, 1998]。

在以上定义中，Englemore 和 Morgan 并没有指定知识源的条件部分或动作部分的任何细节，它们是逻辑构造。条件部分可以简单如黑板上某些布尔标志的值，或者复杂到在特定时间阶段内特定顺序的事件到达某个事件队列。类似地，动作部分可以简单如一条执行表达式赋值的语句，或者像专家系统中的前向推理那样棘手。这再一次说明了黑板模型可以非常灵活。C++类以及对象的概念足以满足我们的目的。每个知识源将是一个对象(对于非常复杂的，使用对象层级结构)。知识源的动作部分将会通过对象的方法来实现。知识源的条件部分将会作为对象的数据成员。一旦对象处于某种特定状态，则对象的动作部分将被激活。

知识源的一个重要属性是它的自治性。每个知识源是一个专家，而且基本上独立于其他的问题解决者。这提供了并行程序的理想素质之一。理想情况下，并行程序中的任务可以在不同其他任务进行过多交互的情况下并发操作。在黑板模型中，情况即是如此。知识源独立动作，任何主要的交互都通过黑板来完成。这样，从知识源的角度看来，它是单独行动的，从黑板上得到额外的信息，然后将它的发现记录到黑板上。其他知识源的活动、策略和结构都是未知的。在黑板模型中，问题被分给多个自治或半自治的程序解决者。这是黑板模型优于其他并发模型的地方。在最灵活的配置中，知识源是理性 agent(rational agent)，意味着 agent 是完全自足的，能够在同黑板发生最少交互的前提下完成它的任务。理性 agent 联合上黑板，为大规模并行或大规模并行 CMP 提供了最大的机会。这种将理性 agent 用作知识源的用法合并了 PADL 第 5 层的两种主要架构：多 agent 架构和黑板架构。对理性 agent 的完整讨论超出了本书的范围，但是我们可以说理性 agent 适合本章前面部分的"什么是 agent"中讨论的 agent 的范畴。当在大规模系统中将知识源通过理性 agent 来实现时，知识源的规则组件和动作组件是使用归纳逻辑编程(Inductive Logic Programming，ILP)技术来学习的[Bergadano, Gunetti, 1996]。ILP 技术提供了自底向上声明式编程的一种方法。

我们抛弃自底向上过程化编程技术，转向 PADL，目的是为了应对 CMP 上可用内核的规模不断增加带来的复杂度。然而，使用 ILP 或演化编程技术[Goertzel, Pennachin, 2007]的自底向上声明式编程是 PADL 分析模型中合理的部分。这些方法用在第 3 层中，在该层我们主要考虑应用程序框架、类库、算法模板等内容的可用性和实现。

7. 应用程序架构的并发灵活性

尽管有很多不同类型的软件架构都支持并行，但是我们在 PADL 的第 5 层使用多 agent 和黑板架构，因为它们具有通用的本质，而且它们支持的并发模型的范围很广。很多要求并发的不同类型的解可以使用多 agent 或黑板架构来表达。多 agent 架构和黑板是领域无关的。它们可以被用在很多不同的领域中。此外，这些架构可用作各种规模的程序的解，从小的程序到大规模企业范围的解。尽管这些架构可能对正在学习并行或多线程编程技术的开发人员比较陌生，但它们定义明确，而且有很多有用的资源介绍 agent 和黑板的基本思想，参见文献[Russell, Norvig, 2003]、[Englemore, Morgan, 1988]、[Goertzel, Pennachin, 2007]

和[Fagin et al., 1996]。您将会看到架构的选择会影响软件维护、测试和调试。多数成功的软件经历了不断的变化。如果软件中有需要并发和同步的组件，则如果一开始没有选择适当的应用程序架构，则软件的自然演化会非常有挑战性。PADL 分析模型提供了两个众所周知的且易于理解的架构，能够在软件演化的情况下表现得很好。最终，应用程序的并发会由线程和进程等底层操作系统原语来实现。如果应用程序架构明确、模块化、可伸缩且易于理解，那么到可管理的操作系统原语的转换就有成功的机会。另一方面，选择错误或糟糕的应用程序架构会导致出现脆弱、易出错的软件，不能够很容易地改变、维护或演化。尽管对于任何种类的计算机应用程序都是如此，但是当软件涉及并行编程、多线程或多处理时会更为突出。

8.2.2　第 4 层：PADL 中的并发模型

第 5 层中选择的应用程序架构必须足够灵活，以支持第 4 层中选择的并发模型。应用程序中可能会需要多个并发模型，因此选择的架构应当能够容纳多个模型。第 4 层也是一个设计层。当在第 4 层中选择并发模型时，我们不需要按照线程或进程进行思考。

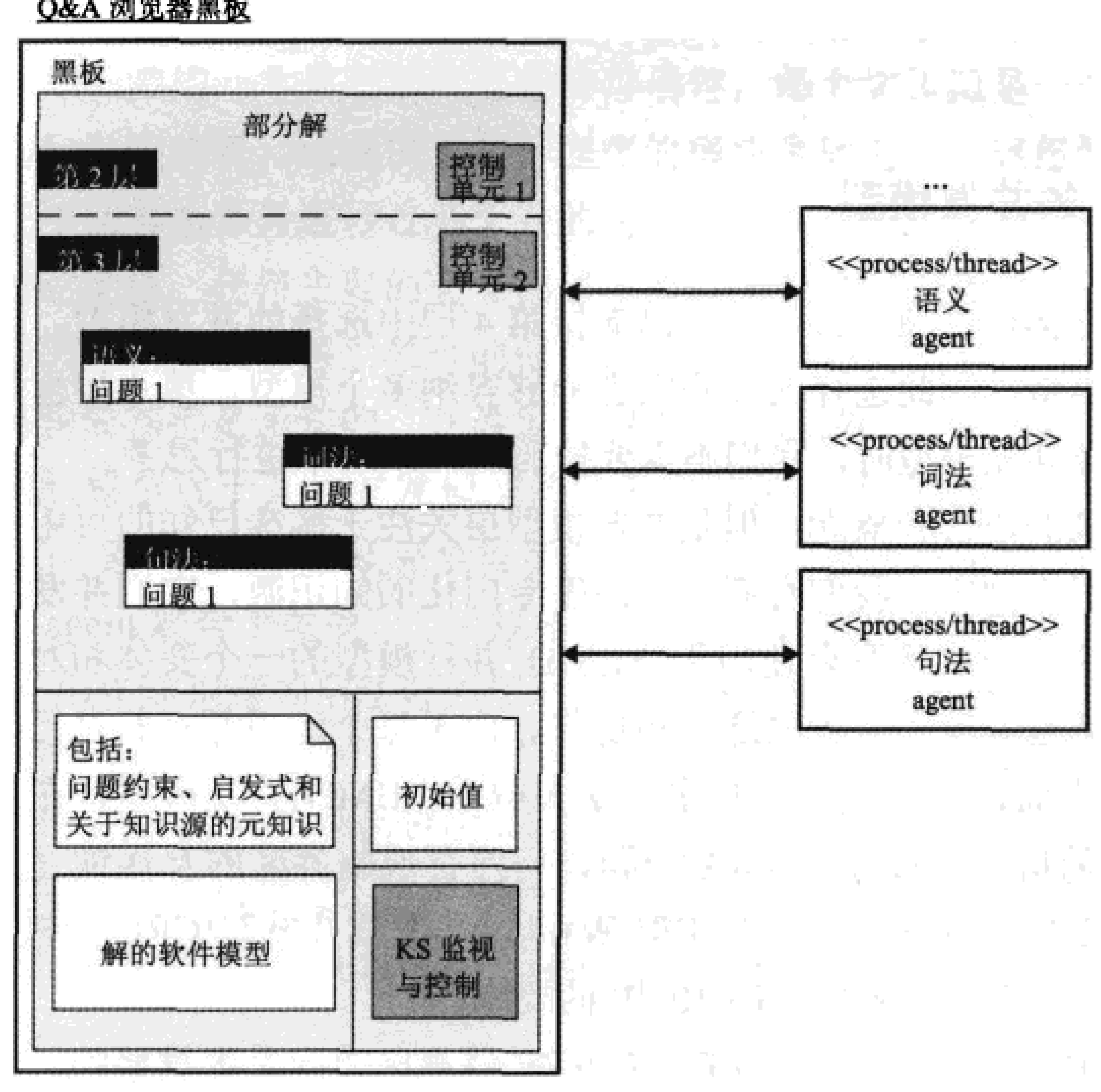
尽管架构聚焦在应用领域的语言和概念，但是并发模型层关注于挑选已知的可用并行模型。在本章所使用的浏览器实例中，我们有着多个高度专门化的知识源，它们并发工作在用户问题的不同部分。这个特定实例联合使用了 MIMD 并行模型的一个变体和对等模型。在这种情况下，对等体通过共享的黑板来通信和协作。由于每个知识源有着自己的专业，而且工作在问题的不同方面，因此我们选择多指令多数据(MIMD)并发模型。在第 7 章中，我们已经解释了并行随机访问计算机(PRAM)和访问内存或临界区的互斥读互斥写(EREW)、并发读互斥写(CREW)、互斥读并发写(ERCW)以及并发读并发写(CRCW)模型。很明显在问答浏览器中，对黑板有着 CREW 或 CRCW 的要求。

在这个阶段，选择 MIMD/对等模型结合 CREW/CRCW 临界区访问是很重要的，因为它会决定第 2 层和第 3 层中线程和进程的使用。这强调了一个事实，即实现模型中的并行化应当自然地遵从解模型中的并发。如果您允许实现模型和解模型过于不同，则您将无法保证软件是正确的。本章在前面讨论的简单游戏示例中使用多 agent 应用程序架构。它使用经典的 boss-worker 并发模型和单指令多数据(SIMD)PRAM 架构和 EREW 临界区访问。在该案例中，临界区是一个共享队列，所有的 agent 使用它来得到将要查找的部分文件的名字。agent 还使用一个共享队列来在找到正确编码时将结果报告给主 agent。表 8-4 显示了 PADL 此时应用到两个示例的分析。

表 8-4

	架　　构	并 发 模 型	PRAM 模型	SIMD/MIMD
猜编码 agent	多 agent	对等	CRCW 或 CREW	MIMD
问答浏览器	黑板	boss-worker	EREW	SIMD

尽管表 8-4 的确给出了两个示例的简化的并发设计选择，它不过是给出了软件并发底层构成的概要评定。一旦您已经确定了并发模型，则可以根据它们的长处和短处来做计划。例如，如果已经知道要涉及 SIMD 模型，则向量优化、循环展开和流水线都将成为重要的方面。您可以利用它们的长处并为它们的短处做计划。另一方面，如果您知道是同 EREW 临界区打交道，那么就可以提前知道存在潜在的瓶颈问题或无限延期问题。选择众所周知的模型的一个优势就在于在开始实现应用程序之前，就可以知道将会遇到何种情况。图 8-5 显示了两个示例的并发底层结构的块图。



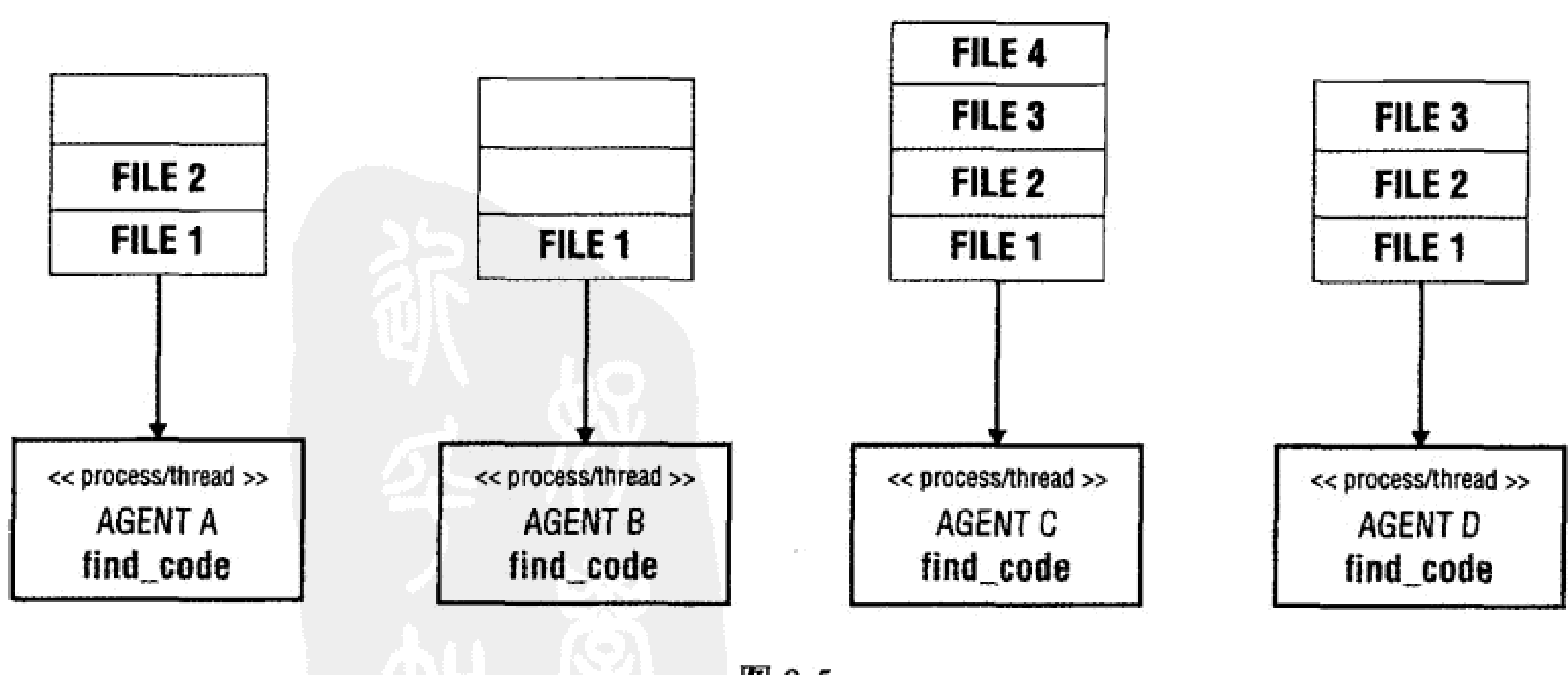


图 8-5

图 8-5 阐明了应用程序中的并发在什么位置。使用 PADL 模型分析的优势就是识别应用程序中的并发本质和并发位置。再次强调，要注意第 4 层和第 5 层是概念设计层。在这些层中，我们并没有真正地关心线程或进程。这些层起着蓝图的功能，指定了应用程序的并发架构和底层结构。第 4 层和第 5 层确定涉及并发的主要的 agent、对象、组件和进程，以及应用程序的并发在什么位置发生。第 4 层和第 5 层还指定了 agent、对象、组件、进程如何同并发模型相关，这些并发模型包括 boss-worker、流水线、SIMD、MIMD 等。在您已经在第 4 层和第 5 层完成分解之后，则可以开始考虑第 3 层中的实现模型。图 8-6 包含了 PADL 分析如何在分解期间应用的总体概述。

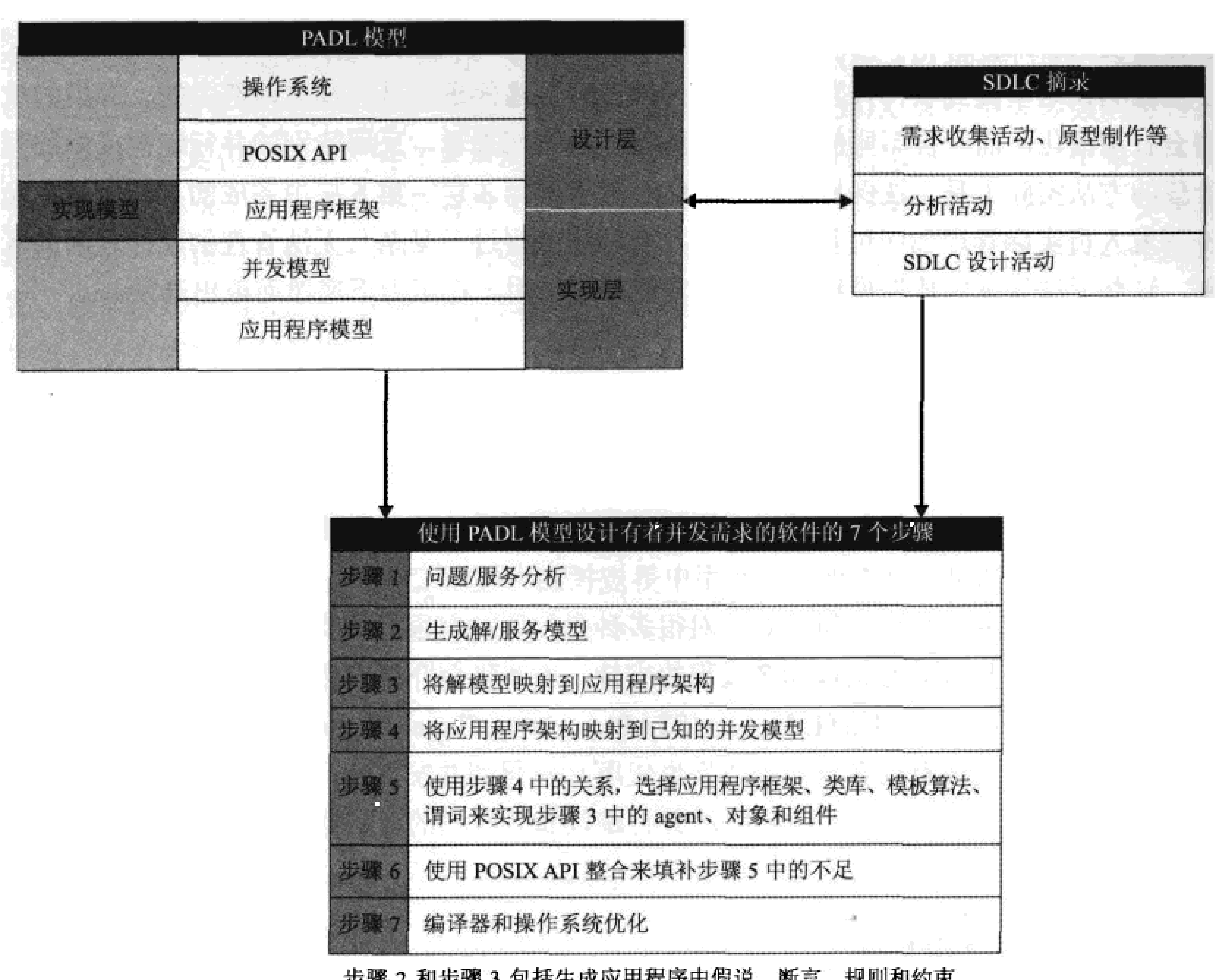


图 8-6

如图 8-6 所示，并发底层结构可以根据步骤 1～步骤 4 来识别。再次注意第 4 层和第 5 层是设计层，而第 3 层包含应用程序的实现模型。第 3 层扮演两个角色，因为它部分属于设计层，同时部分属于实现层。这是因为应用程序框架、模板类、类层次结构和模板算法在设计阶段有用，而且在实现中直接部署。一旦第 3 层的分析完成了，那么应用程序的并发底层结构具有特定的类集合和模板算法集合。接下来我们将关注第 3 层。

8.2.3 第 3 层：PADL 的实现模型

第 3 层的分析由选择应用程序框架、模板类、类库、类层次结构、算法模板、谓词和容器类组成，这些内容在实现解模型分解、应用程序架构选择、并发模型确认中确定的 agent、对象、组件、谓词、进程时是必需的。在第 3 层的分析中，来自第 4 层和第 5 层的所有的主要设计制品都同特定的面向对象组件、算法和谓词关联。一旦第 3 层的分析完成了，则我们就有了应用程序并发底层结构和并发实现模型的最清楚的画面。在图 8-6 中需要注意，第 3 层中的元素是可以追溯到最初的问题陈述的。由于整个 PADL 分析发生在结构良好的 SDLC 的上下文中，因此您交付的应用程序将会是可靠的、可扩展的而且可以被软件开发组所维护。

我们在这里提到软件开发组，是因为软件开发这件事情是需要集体努力的。有用的软件会长期演化，而且被不同的人改动和维护。从长期来看，采用特别的并行编程或多线程处理的方法实际上是无法保持的。在 PADL 分析的第 3 层～第 5 层中完成的工作，是在允许开发人员实际管理和应付复杂性的并发软件同变得过于复杂且无法管理的软件之间的区别，这些无法管理的软件最终会随着开发组和终端用户在压力下崩溃而退出开发。

1. C++组件的帮助

幸运的是，C++环境有着令人印象深刻的特性、库和技术的集合，它们可用来完成模型实现。我们在这里强调模型，因为并行和并发在模型的上下文中能够进行最好的管理。图 8-6 中显示的 7 个步骤从问题模型到解模型，直到最终的实现模型。如果在解中使用声明式模型，那么您可以可靠地开发利用中等规模到大规模 CMP 或大规模并行多核的软件。尽管 C++中没有并发构造，但是 C++对很多种类的库均有很好的支持。因此，可以通过库的使用来为 C++加入并行编程和多线程的支持。本章将会介绍支持并行编程的 3 个非常重要的 C++组件库：Parallel STL Library(STAPL)、Intel Thread Building Blocks(TBB)和新的 C++0x 标准。尽管还有很多不同的成果也使用 C++得到并发支持，但是我们将只介绍以上三个库，因为它们很快将成为支持并发、并行编程和多线程的最广泛可用且容易得到的 C++组件。

C++0x 或 C++09 标准

到写作本书时为止，新的 C++0x 标准正接近于被采用。标准很可能会在 2009 年被采用，而且 C++0x 的名称将会成为 C++09 标准。2003 年被采用的 C++03 是当前的 C++标准。C++0x 标准包括一些对语言所做的令人激动的更新。多数改进是以新的类和库来体现的。新的 C++标准将会通过增加并发编程库来对并行编程和多线程提供更多的支持。这对于 C++开发人员而言是一个非常好的消息，因为在新标准之前，每个 C++环境中都没有被允许的并行编程工具，新的标准将会改变这一点。表 8-5 列出了一些 C++0x(C++09)将支持的新库。

表 8-5

C++0x 标准中新的库	描　　述
MPI	Message Passing Interface(消息传递接口，MPI)库用于分布式内存和并行应用程序编程
Interprocess	包含进程间机制，例如共享内存、内存映射文件、进程共享互斥量和条件变量；还包括进程的容器和分配器
-asio	可移植网络库，包括套接字(socket)、定时器和主机名解析套接字 io 流

表 8-5 中的库包含了我们在第 4～第 7 章中讨论过的部分功能，包括线程、互斥量、条件变量、进程间通信(IPC)等能力。类库本质上是包装操作系统 API 的接口类。它们必须同 POSIX 线程管理和进程管理工具兼容。尽管新的标准为 POSIX API 中大多数的特性提供了接口，但是仍然可以使用其他库或通过提供对 POSIX API 的您自己的接口类来进行填补。程序清单 8-1 显示了实现新标准 C++的 Boost C++库的线程类以及为开发人员提供的服务中的一小部分。Boost 提供了免费的、评审过的可移植 C++源库，它同 C++标准库兼容。当前，10 个库已经包含在 C++ Standard Committee 的 Library Technical Report 中，而且将会成为将来 C++标准的一部分。

程序清单 8-1

```
//Listing 8-1 An implementation of the new C++0x thread class.


 1  #ifndef BOOST_THREAD_THREAD_PTHREAD_HPP
 2  #define BOOST_THREAD_THREAD_PTHREAD_HPP
 3  // Copyright (C) 2001-2003
 4  // William E. Kempf
 5  // Copyright (C) 2007 Anthony Williams
 6  //
 7  //  Distributed under the Boost Software License,
    //  Version 1.0. (See accompanying
 8  //  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
 9
10  #include <boost/thread/detail/config.hpp>
11
12  #include <boost/utility.hpp>
13  #include <boost/function.hpp>
14  #include <boost/thread/mutex.hpp>
15  #include <boost/thread/condition_variable.hpp>
16  #include <list>
17  #include <memory>
18
19  #include <pthread.h>
20  #include <boost/optional.hpp>
21  #include <boost/thread/detail/move.hpp>
```

```
22 #include <boost/shared_ptr.hpp>
23 #include "thread_data.hpp"
24 #include <stdlib.h>
25
26 #ifdef BOOST_MSVC
27 #pragma warning(push)
28 #pragma warning(disable:4251)
29 #endif
30
31 namespace boost
32 {
33   class thread;
34
35   namespace detail
36   {
37       class thread_id;
38   }
39
40   namespace this_thread
41   {
42       BOOST_THREAD_DECL detail::thread_id get_id();
43   }
44
45   namespace detail
46   {
47       class thread_id
48       {
49       private:
50           detail::thread_data_ptr thread_data;
51
52           thread_id(detail::thread_data_ptr thread_data_):
53               thread_data(thread_data_)
54           {}
55           friend class boost::thread;
56           friend thread_id this_thread::get_id();
57       public:
58           thread_id():
59               thread_data()
60           {}
61
62           bool operator==(const thread_id& y) const
63           {
64               return thread_data==y.thread_data;
65           }
66
67           bool operator!=(const thread_id& y) const
68           {
69               return thread_data!=y.thread_data;
70           }
71
```

```
72            bool operator<(const thread_id& y) const
73            {
74                return thread_data<y.thread_data;
75            }
76
77            bool operator>(const thread_id& y) const
78            {
79                return y.thread_data<thread_data;
80            }
81
82            bool operator<=(const thread_id& y) const
83            {
84                return !(y.thread_data<thread_data);
85            }
86
87            bool operator>=(const thread_id& y) const
88            {
89                return !(thread_data<y.thread_data);
90            }
91
92            template<class charT, class traits>
93            friend std::basic_ostream<charT, traits>&
94            operator<<(std::basic_ostream<charT, traits>& os,
                          const thread_id& x)
95            {
96                if(x.thread_data)
97                {
98                    return os<<x.thread_data;
99                }
100               else
101               {
102                   return os<<"{Not-any-thread}";
103               }
104           }
105       };
106   }
107
108   struct xtime;
109   class BOOST_THREAD_DECL thread
110   {
111   private:
112       thread(thread&);
113       thread& operator=(thread&);
114
115       template<typename F>
116       struct thread_data:
117           detail::thread_data_base
118       {
119           F f;
120
```

```
121            thread_data(F f_):
122                f(f_)
123            {}
124            thread_data(detail::thread_move_t<F> f_):
125                f(f_)
126            {}
127
128            void run()
129            {
130                f();
131            }
132        };
133
134        mutable boost::mutex thread_info_mutex;
135        detail::thread_data_ptr thread_info;
136
137        void start_thread();
138
139        explicit thread(detail::thread_data_ptr data);
140
141        detail::thread_data_ptr get_thread_info() const;
142
143    public:
144        thread();
145        ~thread();
146
147        template <class F>
148        explicit thread(F f):
149            thread_info(new thread_data<F>(f))
150        {
151            start_thread();
152        }
153        template <class F>
154        thread(detail::thread_move_t<F> f):
155            thread_info(new thread_data<F>(f))
156        {
157            start_thread();
158        }
159
160        thread(detail::thread_move_t<thread> x);
161        thread& operator=(detail::thread_move_t<thread> x);
162        operator detail::thread_move_t<thread>();
163        detail::thread_move_t<thread> move();
164
165        void swap(thread& x);
166
167        typedef detail::thread_id id;
168
169        id get_id() const;
170
```

```
171         bool joinable() const;
172         void join();
173         bool timed_join(const system_time& wait_until);
174
175         template<typename TimeDuration>
176         inline bool timed_join(TimeDuration const& rel_time)
177         {
178             return timed_join(get_system_time()+rel_time);
179         }
180         void detach();
181
182         static unsigned hardware_concurrency();
183
184         // backwards compatibility
185         bool operator==(const thread& other) const;
186         bool operator!=(const thread& other) const;
187
188         static void sleep(const system_time& xt);
189         static void yield();
190
191         // extensions
192         void interrupt();
193         bool interruption_requested() const;
194     };
195
196     inline detail::thread_move_t<thread> move(thread& x)
197     {
198         return x.move();
199     }
200
201     inline detail::thread_move_t<thread>
       move(detail::thread_move_t<thread> x)
202     {
203         return x;
204     }
205
206
207     template<typename F>
208     struct thread::thread_data<boost::reference_wrapper<F> >:
209         detail::thread_data_base
210     {
211         F& f;
212
213         thread_data(boost::reference_wrapper<F> f_):
214             f(f_)
215         {}
216
217         void run()
218         {
219             f();
```

```
220         }
221     };
222
223     namespace this_thread
224     {
225         class BOOST_THREAD_DECL disable_interruption
226         {
227             disable_interruption(const disable_interruption&);
228             disable_interruption& operator=(const
disable_interruption&);
229
230             bool interruption_was_enabled;
231             friend class restore_interruption;
232         public:
233             disable_interruption();
234             ~disable_interruption();
235         };
236
237         class BOOST_THREAD_DECL restore_interruption
238         {
239             restore_interruption(const restore_interruption&);
240             restore_interruption& operator=(const restore_interruption&);
241         public:
242             explicit restore_interruption(disable_interruption& d);
243             ~restore_interruption();
244         };
245
246         BOOST_THREAD_DECL thread::id get_id();
247
248         BOOST_THREAD_DECL void interruption_point();
249         BOOST_THREAD_DECL bool interruption_enabled();
250         BOOST_THREAD_DECL bool interruption_requested();
251
252         inline void yield()
253         {
254             thread::yield();
255         }
256
257         template<typename TimeDuration>
258         inline void sleep(TimeDuration const& rel_time)
259         {
260             thread::sleep(get_system_time()+rel_time);
261         }
262     }
263
264     namespace detail
265     {
266         struct thread_exit_function_base
267         {
268             virtual ~thread_exit_function_base()
```

```
269             {}
270             virtual void operator()() const=0;
271         };
272 
273         template<typename F>
274         struct thread_exit_function:
275             thread_exit_function_base
276         {
277             F f;
278 
279             thread_exit_function(F f_):
280                 f(f_)
281             {}
282 
283             void operator()() const
284             {
285                 f();
286             }
287         };
288 
289         BOOST_THREAD_DECL void
        add_thread_exit_function(thread_exit_function_base*);
290     }
291 
292     namespace this_thread
293     {
294         template<typename F>
295         inline void at_thread_exit(F f)
296         {
297             detail::thread_exit_function_base*
            const thread_exit_func=new detail::thread_exit_function<F>(f);
298             detail::add_thread_exit_function(thread_exit_func);
299         }
300     }
301 
302     class BOOST_THREAD_DECL thread_group
303     {
304     public:
305         thread_group();
306         ~thread_group();
307 
308         thread* create_thread(const function0<void>& threadfunc);
309         void add_thread(thread* thrd);
310         void remove_thread(thread* thrd);
311         void join_all();
312         void interrupt_all();
313         size_t size() const;
314 
315     private:
```

```
316         thread_group(thread_group&);
317         void operator=(thread_group&);
318 
319         std::list<thread*> m_threads;
320         mutex m_mutex;
321     };
322 } // namespace boost
323 
324 #ifdef BOOST_MSVC
325 #pragma warning(pop)
326 #endif
327 
328 
329 #endif
```

注意程序清单 8-1 的第 302 行，新的线程类也支持线程组。这提供了 POSIX pthread API 的大部分功能性。请注意在第 19 行，在这个实现中包含进了 pthread.h。我们已经在第 4 章介绍了操作系统的任务，并且解释了操作系统是硬件和多核的看门人(gatekeeper)。任何呈现多线程或多进程能力的类层次结构、应用程序框架或模板类都提供了操作系统 API 的接口类。因此，这是一个受欢迎的接口类，因为有了如同程序清单 8-1 中所显示的线程类那样的标准 C++类，您距离并行编程的声明式解释就更近了。这些类型的接口类不仅为底层操作系统原语增加了声明式或面向对象的风格，而且还简化了接口。考虑第 308 行～第 311 行同 POSIX API pthread 对等部分的比较。这些方法替换掉了对 pthread 调用的直接调用，如 pthread_create( )、pthread_join( )、使用属性对象来分离线程、线程取消函数等。使用这种类型的库来实现的接口类使得开发人员能够维持应用程序开发的面向对象或声明式方法。

C++0x(C++09)互斥量接口类

除了线程接口类，C++0x 标准还将包含互斥量接口类，见程序清单 8-2。

我们已经介绍了 POSIX pthread_mutex( )以及它们在应对多线程带来的一个主要的同步挑战中的用途。但是 pthread_mutex( )是位于 PADL 分析的第 2 层。理想情况下，您不希望在第 2 层完成多数的同步，不这样做的一个重要的原因就是因为 POSIX API 具有过程化语义。为了利用 POSIX API 服务，您将必须提供接口类。这里需要新的提供标准互斥量类的 C++标准。您可以看到，基本的 pthread 互斥量功能(例如加锁、解锁、销毁、定时加锁等功能)都被封装在互斥量类中。同时，定义了定时的互斥量类，该类封装了条件变量的部分功能。

程序清单 8-2

```
//Listing 8-2 An implementation of the new standard C++0x mutex class.

  1 #ifndef BOOST_THREAD_PTHREAD_MUTEX_HPP
  2 #define BOOST_THREAD_PTHREAD_MUTEX_HPP
```

```
 3 // (C) Copyright 2007 Anthony Williams
 4 // Distributed under the Boost Software License, Version 1.0. (See
 5 // accompanying file LICENSE_1_0.txt or copy at
 6 // http://www.boost.org/LICENSE_1_0.txt)
 7
 8 #include <pthread.h>
 9 #include <boost/utility.hpp>
10 #include <boost/thread/exceptions.hpp>
11 #include <boost/thread/locks.hpp>
12 #include <boost/thread/thread_time.hpp>
13 #include <boost/assert.hpp>
14 #ifndef WIN32
15 #include <unistd.h>
16 #endif
17 #include <errno.h>
18 #include "timespec.hpp"
19 #include "pthread_mutex_scoped_lock.hpp"
20
21 #ifdef _POSIX_TIMEOUTS
22 #if _POSIX_TIMEOUTS >= 0
23 #define BOOST_PTHREAD_HAS_TIMEDLOCK
24 #endif
25 #endif
26
27 namespace boost
28 {
29   class mutex:
30       boost::noncopyable31    {
32   private:
33       pthread_mutex_t m;
34   public:
35       mutex()
36       {
37           int const res=pthread_mutex_init(&m,NULL);
38           if(res)
39           {
40               throw thread_resource_error();
41           }
42       }
43       ~mutex()
44       {
45           BOOST_VERIFY(!pthread_mutex_destroy(&m));
46       }
47
48       void lock()
49       {
50           BOOST_VERIFY(!pthread_mutex_lock(&m));
51       }
```

```
52
53         void unlock()
54         {
55             BOOST_VERIFY(!pthread_mutex_unlock(&m));
56         }
57
58         bool try_lock()
59         {
60             int const res=pthread_mutex_trylock(&m);
61             BOOST_ASSERT(!res || res==EBUSY);
62             return !res;
63         }
64
65         typedef pthread_mutex_t* native_handle_type;
66         native_handle_type native_handle()
67         {
68             return &m;
69         }
70
71         typedef unique_lock<mutex> scoped_lock;
72         typedef scoped_lock scoped_try_lock;
73     };
74
75     typedef mutex try_mutex;
76
77     class timed_mutex:
78         boost::noncopyable
79     {
80     private:
81         pthread_mutex_t m;
82   #ifndef BOOST_PTHREAD_HAS_TIMEDLOCK
83         pthread_cond_t cond;
84         bool is_locked;
85   #endif
86     public:
87         timed_mutex()
88         {
89             int const res=pthread_mutex_init(&m,NULL);
90             if(res)
91             {
92                 throw thread_resource_error();
93             }
94   #ifndef BOOST_PTHREAD_HAS_TIMEDLOCK
95             int const res2=pthread_cond_init(&cond,NULL);
96             if(res2)
97             {
98                 BOOST_VERIFY(!pthread_mutex_destroy(&m));
99                 throw thread_resource_error();
```

```
100             }
101             is_locked=false;
102 #endif
103         }
104         ~timed_mutex()
105         {
106             BOOST_VERIFY(!pthread_mutex_destroy(&m));
107 #ifndef BOOST_PTHREAD_HAS_TIMEDLOCK
108             BOOST_VERIFY(!pthread_cond_destroy(&cond));
109 #endif
110         }
111 
112         template<typename TimeDuration>
113         bool timed_lock(TimeDuration const & relative_time)
114         {
115             return timed_lock(get_system_time()+relative_time);
116         }
117 
118 #ifdef BOOST_PTHREAD_HAS_TIMEDLOCK
119         void lock()
120         {
121             BOOST_VERIFY(!pthread_mutex_lock(&m));
122         }
123 
124         void unlock()
125         {
126             BOOST_VERIFY(!pthread_mutex_unlock(&m));
127         }
128 
129         bool try_lock()
130         {
131             int const res=pthread_mutex_trylock(&m);
132             BOOST_ASSERT(!res || res==EBUSY);
133             return !res;
134         }
135         bool timed_lock(system_time const & abs_time)
136         {
137            struct timespec const timeout=detail::get_timespec(abs_time);
138             int const res=pthread_mutex_timedlock(&m,&timeout);
139             BOOST_ASSERT(!res || res==EBUSY);
140             return !res;
141         }
142 #else
143         void lock()
144         {
145           boost::pthread::pthread_mutex_scoped_lock const local_lock(&m);
146             while(is_locked)
147             {
148                 BOOST_VERIFY(!pthread_cond_wait(&cond,&m));
149             }
```

```
150             is_locked=true;
151         }
152
153         void unlock()
154         {
155             boost::pthread::pthread_mutex_scoped_lock const local_lock(&m);
156             is_locked=false;
157             BOOST_VERIFY(!pthread_cond_signal(&cond));
158         }
159
160         bool try_lock()
161         {
162             boost::pthread::pthread_mutex_scoped_lock const local_lock(&m);
163             if(is_locked)
164             {
165                 return false;
166             }
167             is_locked=true;
168             return true;
169         }
170
171         bool timed_lock(system_time const & abs_time)
172         {
173             struct timespec const timeout=detail::get_timespec(abs_time);
174             boost::pthread::pthread_mutex_scoped_lock const local_lock(&m);
175             while(is_locked)
176             {
177                 int const cond_res=pthread_cond_timedwait(&cond,&m,&timeout);
178                 if(cond_res==ETIMEDOUT)
179                 {
180                     return false;
181                 }
182                 BOOST_ASSERT(!cond_res);
183             }
184             is_locked=true;
185             return true;
186         }
187 #endif
188
189         typedef unique_lock<timed_mutex> scoped_timed_lock;
190         typedef scoped_timed_lock scoped_try_lock;
191         typedef scoped_timed_lock scoped_lock;
192     };
193
194 }
195
196
197 #endif
```

在第 77 行～第 110 行中，您可以看到 POSIX API 函数，如 pthread_mutex_init( )、

pthread_cond_init( )、pthread_cond_destroy( )和 pthread_mutex_destroy( )，所提供的服务在 timed_mutex 接口类中是如何实现的。我们在第 4 章中解释了理解操作系统的任务的重要性，即使目的是在较高级别上编程。新的 C++标准中将会出现这样的互斥量类的实现可以用来证明这一点。这个类并没有完成您期望的互斥量类的全部功能。此外，这个互斥量类可能会同其他线程库协同使用，如 TBB。如果在运行时出现任何问题，您必须大概了解这些类在哪里同操作系统发生交互。C++线程类和 TBB 线程工具对 POSIX pthread 的使用可能会存在细微的差别。因此，当您在 PADL 模型的第 3 层中配置较高级别的类时，必须紧记这些类最终会映射到操作系统 API，即 POSIX API。

获得 C++0x 并发编程库的早期实现

程序清单 8-1 中的线程库和程序清单 8-2 中的互斥量类是来自 http://www.boost.org 上的 Boost C++库。Boost 提供了免费的、同行评审的可移植 C++源代码库。Boost 团队强调和 C++标准库一同工作得很好的库。Boost 库有意成为能够被很宽范围的应用程序所广泛使用的库。Boost 小组的目标是建立“既成惯例(existing practice)”并提供参考实现，从而使得 Boost 库适合于最终的标准化。我们在本章中介绍的并发编程库均有 Boost 参考实现，而且可以免费下载。

2. Intel 线程构建块

在 PADL 分析的第 3 层中的另一个重要的组件就是 Intel 线程构建块(Intel Threading Building Blocks，TBB)。TBB 是由通用算法模板、容器类、面向对象同步组件以及在多线程编程中非常有用的各种组件组成的 C++组件集合。TBB 是使用线程的 C++代码的基于运行时并行编程模型。它的设计目的主要是书写可扩展的应用程序，这些程序：

- 指定任务而不是线程
- 强调数据并行编程
- 利用并发集合

表 8-6 列出了 TBB 库中的一些主要模板算法和容器。

表 8-6

TBB 通用并行算法	具有并发支持的 TBB 容器
parallel_for	concurrent_queue
parallel_scan	concurrent_vector
parallel_reduce	concurrent_hash_map
parallel_while	
pipeline	
parallel_sort	

尽管 TBB 在某些领域发生交叉(例如，互斥量)，有了新的 C++标准中的新的并发编程

库，TBB 在很大程度上提供了补充，并提供了一组可用于 PADL 实现层的工具。回顾图 8-6 中的 7 个步骤。在第 4 层中，我们确定并发模型。第 4 层并不决定这些并发模型将由线程或进程来实现。实际上，第 4 层中的并发模型可以通过集群或其他分布计算模型来实现。使用黑板架构的“问答”实例并没有指示应当用线程还是进程来实现知识源。有可能通过多个线程或多个进程，甚至是进程与线程的组合，来实现一个知识源。为了满足知识源的逻辑要求，这可能是必须的。为了了解如何实现黑板示例中的第 3 层，您可以仔细察看黑板架构的控制策略。

黑板：临界区

在可以并发激活多个知识源的黑板的实现中，有着多层的控制。在最底层，存在同步配置来保护黑板的完整性。尽管黑板是 PADL 的第 5 层中的应用架构，而且在该层上是从概念上进行讨论的，但仍可以认为黑板是一个临界区，因为从本质上而言，它是一个共享的可更改的资源。实际上，表 8-4 显示出黑板应用到知识源时，采用了 CRCW 或 CREW 并发模型。在并行环境中，对知识源的读访问和写访问必须进行协调和同步。协调和同步可能会涉及文件加锁、互斥量等。这一层的控制不是直接与作为知识源工作目标的解相关的，它是一个有用的控制层，而且应当与黑板要解决的问题无关。在本章示例的架构方法中，这一层的控制应当被接口类实现，例如第 7 章中引入的互斥量类和信号量类。还可以利用 TBB 的并发容器来部分满足对 CRCW 或 CREW 的需求。黑板的一部分可以使用 TBB 中的 concurrent_vector 类来实现。concurrent_vector 类允许安全的同时访问，而且具有易用性，且可以与标准 C++库中的组件一同工作。程序清单 8-3 是“问答”黑板示例中的节选。

程序清单 8-3

```
//Listing 8-3  A Program that uses concurrent_vector and parallel_for from TBB.

  1  using namespace std;
  2  #include <iostream>
  3  #include <vector>
  4  #include <stdlib.h>
  5  #include <ctype.h>
  6  #include <algorithm>
  7  #include <iterator>
  8  #include <string>
  9  #include "tbb/blocked_range.h"
 10  #include "tbb/parallel_for.h"
 11  #include "tbb/task_scheduler_init.h"
 12  #include "tbb/concurrent_vector.h"
 13  #include <sstream>
 14  #include <fstream>
 15
 16  using namespace tbb;
 17  concurrent_vector<string>  Terms;
 18  concurrent_vector<string>  Question;
 19
```

```
20
21  class lower_case{
22
23  public:
24     char operator()(char X){
25
26        return(tolower(X));
27     }
28
29  };
30
31
32
33  void changeIt(string &X)
34  {
35     transform(X.begin(),X.end(),X.begin(),lower_case());
36  }
37
38
39
40  void tokenize(string &X)
41  {
42     stringstream Sin(X);
43     string Token;
44     while(!Sin.fail() && Sin.good())
45     {
46        Sin >> Token;
47        Terms.push_back(Token);
48     }
49  }
50
51
52  class parallel_lower_case{
53
54  public:
55
56     void operator() (const blocked_range<int> &X) const
57     {
58        for(int I = X.begin(); I != X.end(); I++)
59        {
60
61           changeIt(Terms[I]);
62
63         }
64
65     }
66
67  };
68
69  class valid_tokens{
```

```
70  public:
71     void operator() (const blocked_range<int> &X) const
72     {
73        for(int I = X.begin(); I != X.end(); I++)
74        {
75           tokenize(Question[I]);
76
77        }
78
79     }
80
81  };
82
83
84
85
86  int main(int argc,char *argv[])
87  {
88
89    task_scheduler_init Init;
90    ifstream Fin("question.txt");
91    istream_iterator<string> Ftr(Fin);
92    istream_iterator<string> Eof;
93    copy(Ftr,Eof,back_inserter(Question));
94    Fin.close();
95    parallel_lower_case Lower;
96    valid_tokens  Token;
97    parallel_for(blocked_range<int>(0,Question.size(),
                                    (Question.size() /2)) ,Token);
98    parallel_for(blocked_range<int>(0,Terms.size(),
                                    (Terms.size() /2)),Lower);
99    ostream_iterator<string> Out(cout,"\n");
100   copy(Terms.begin(),Terms.end(),Out);
101
102
103 }
```

对知识源的部分处理要求问题中的各个单词应当标注上词性或标注为 unknown、misspelled 等。程序清单 8-3 中的程序仅仅是为了展示。我们将它包含在这里，是为了说明可以如何轻易地从 PADL 的第 4 层和第 5 层映射到 TBB 组件。在第 17 行声明的 Terms 组件是一个 TBB concurrent_vector<T>。这个向量允许多个知识源对它进行并发读访问。这满足 CREW 并发模型的一部分。类似地，有一些处理要求将问题分解为单独的单词或 token，而且 token 被转换为小写。在第 97 行中对 TBB parallel_for 算法的调用将保存在 Question 向量中的每个 Question 分成单独的 token。这是在从第 40 行～第 49 行完成的。第 97 行的 parallel_for( )算法以 Token 作为 C++函数对象。第 98 行的 parallel_for( )将 Terms 向量中的 token 转换为小写。parallel_for 算法进行了高级循环展开。它可以并行执行它的函数对象中的组件。在本例中，第 61 行和第 75 行的函数是并行执行的。注意第 93 行和第 100 行，

TBB concurrent_vector<T>与 ostream_iterator<string>、istream_iterator<string>以及标准 C++ copy( )算法一同使用。

下面是程序清单 8-3 的程序概要。

程序概要 8-1

程序名：

convert_it.cc (程序清单 8-3)

描述：

程序清单 8-3 中的程序说明了如何从 PADL 的第 4 层和第 5 层映射到 TBB 组件。Terms 组件是一个 TBB concurrent_vector<T>。这个向量允许被多个知识源并发读取访问。TBB parallel_for 算法用来将保存在 Question 向量中的每个 Question 分割成单独的 token，使用 Token 作为 C++函数对象，并将 Terms 向量中的 token 转换为小写。TBB concurrent_vector<T>与 ostream_iterator<string>、istream_iterator<string>以及标准 C++ copy( )算法一同使用。

必需的库：

tbb(Intel Thread Building Blocks)

需要的其他源文件：

无

必需的用户定义头文件：

无

编译和链接指令：

c++ -o convert_it -I TBBIncludePath convert_it.cc -L TBBLibraryPath –ltbb

测试环境：

Linux Kernel 2.6

处理器：

Core 2 Duo

注释：

无

获取 TBB 库

TBB 库是开源的，可以从 http://www.threadingbuildingblocks.org 获得。它可用于 Intel 平台以及基于 Intel 的 Macs。在撰写本书时，该库尚未移植到 Solaris、HP-UX 或 AIX。它可以工作在 Linux 环境中。

3. 并行 STL 库

如前所述，并发编程库工具将会在新的 C++标准中出现，包括一个被期待已久的线程库以及同步组件。另外一组支持并行编程的重要的 C++组件是 Standard Template Adaptive Parallel Library(STAPL)。C++中的 TBB 以及即将到来的并发编程库被设计为在多核和并行计算机中工作，而 STAPL 则被设计为在共享内存并行计算机和分布式内存并行计算机中均可以工作。STAPL 库的设计目标是同设计并行程序的 PADL 分析方法一致。STAPL 被设计为允许开发人员在高级别的抽象上工作。它提供接口类和接口算法，隐藏了并行编程的很多特定细节。本书从头到尾都对应用级开发人员和系统级开发人员进行了区分。系统级开发趋向于接近操作系统 API 和 SPI 开展工作。STAPL 用户可分为以下 3 组。

- 用户：他们是应用程序开发人员，主要是使用 STAPL 组件，不需要进行大量的扩展或重新定义。
- 开发人员：这些开发人员经常会特定于领域或应用中，通过为用户增加新的数据结构和算法，扩展 STAPL 工具集。
- 专家：这组人员为用户和开发人员提供额外的编程框架来开发算法和应用程序。

C++的发明者 Bjarne Stroustrup 参与到了 STAPL 的开发。这意味着您可以相信 STAPL 会同 C++标准的哲学相一致。STAPL 由以下主要组件组成。

- pContainers：支持并发的容器。
- Views：支持迭代的概念以及对 pContainers 的对象访问。
- pAlgorithms：支持并行的标准模板库(Standard Template Library，STL)算法。
- pRange：pAlgorithms 是在一个范围内执行的。pRange 使得算法的工作分解结构(WBS)可以在任务依赖图(Task Dependency Graph)中规定。
- Runtime：提供性能监视、通信原语、pRanges 子视图或任务的调度等的运行时系统。

图 8-7 显示了 STAPL 库的结构框图。

标准 C++并发编程库将会与 STAPL 互补且兼容。

TBB 部分地由 STAPL 中的设计概念得到灵感，但是 STAPL 是更高级别的框架，扩展了 STL 和 TBB。在图 8-7 中，还要注意运行时系统会与 POSIX pthread API 以及其他操作系统线程和进程 API 存在接口。可以通过查看将会成为 C++0x、TBB 和 STAPL 的一部分的新的并发编程类库的结构，了解到使用封装了低级操作系统原语的接口类和组件是为 C++开发人员提供并发支持的最实际的方式。注意在图 8-7 中，用户应用程序代码最少要比 POSIX 线程高两层。查看图 4-1，该图显示了从开发人员角度来看，操作系统和类似 STAPL 的框架之间的关系。在第 3 章讨论的并行编程的复杂性和特殊的挑战，要求软件开发人员对类库、应用程序框架、模板类、算法模板以及操作系统提供的并发和同步服务之

间的集成有着清晰的理解。PADL 和 PBS 分析(将会在本章稍后部分讨论)是在对所有涉及的部分间的关系有着坚实的理解的条件下进行的。

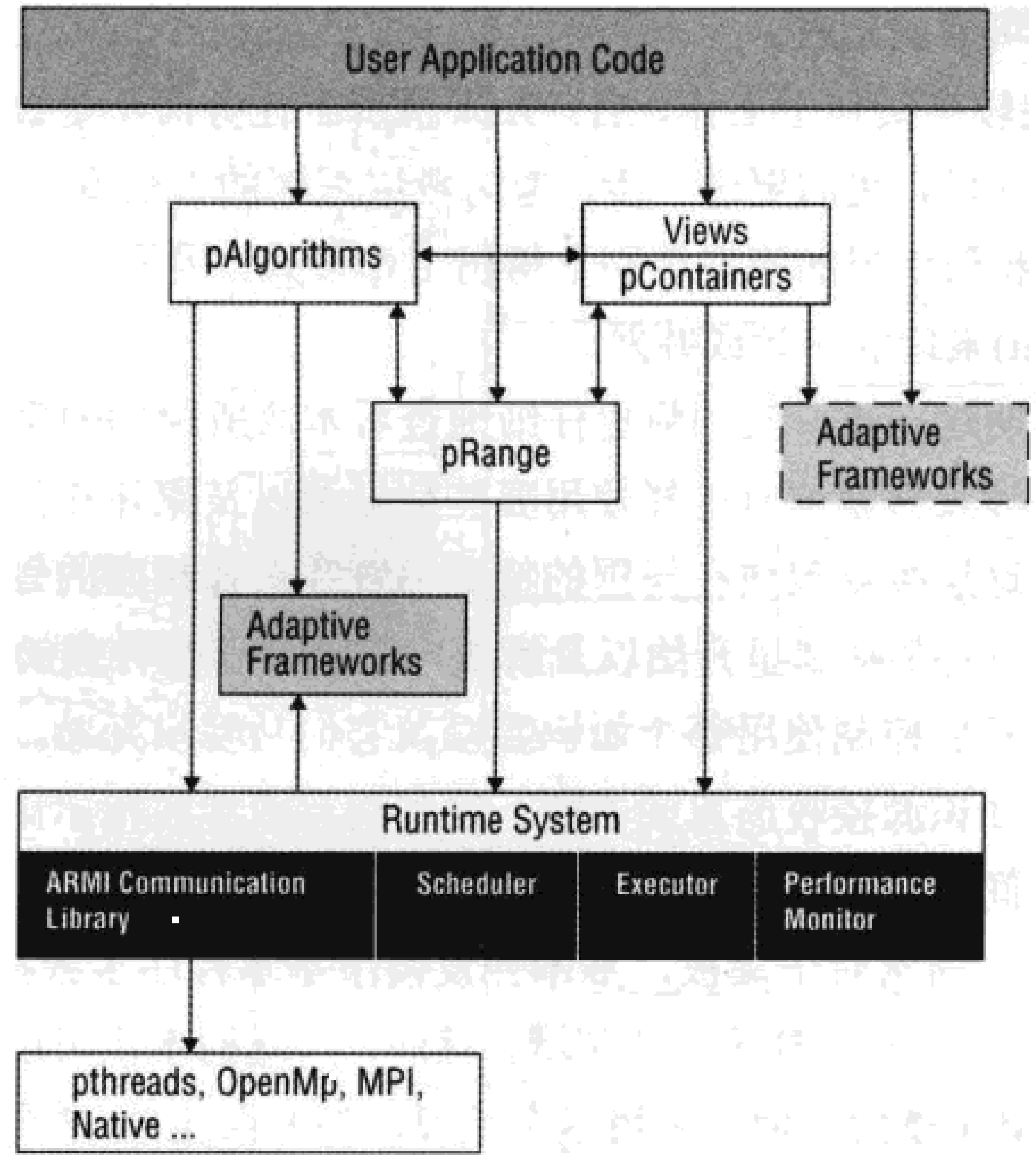


图 8-7

注意：

关于 STAPL 库的更多信息，可参见 http://parasol.tamu.edu/stapl。

新的标准 C++并发编程类库 TBB 和 STAPL，提供了 PADL 分析的第 3 层中所需要的大量与实现领域无关的组件。要记住第 3 层组件包括特定领域应用程序框架、类库等以及领域无关的组件。

4. “实现层”映射

在黑板示例中，黑板的分割方式决定了 CREW 或 CRCW 并发(在 PADL 的第 2 层确定)是否合适。对临界区最灵活的访问模型是 CRCW。CRCW 可以依靠黑板的结构来达到。例如，如果 16 个知识源涉及一次合作，而且每个知识源访问它自己的黑板分块，那么这些知识源可以对黑板进行并发读写，不会发生数据竞争问题。了解通信模型也可以帮助确定将会使用哪些第 3 层组件。显然，支持并发的容器，例如 TBB concurrent_vector<T>或 concurrent_queue<T>或来自 STAPL 的 pContainers 可被用来实现黑板中要求 CREW 的一些部分。通过一定的规划这些结构也能够支持 CRCW。

5. PADL：第 3 层类型控制策略

我们选择多 agent 架构和黑板架构作为 PADL 的第 5 层的两个基本应用程序架构的主

要理由在于这些架构能够捕捉的工作模式非常灵活，而且易于理解。

在开始对应用程序中的并发线程或进程进行协调和同步之前，最好在领域和应用程序级对工作模式有了可靠的把握。

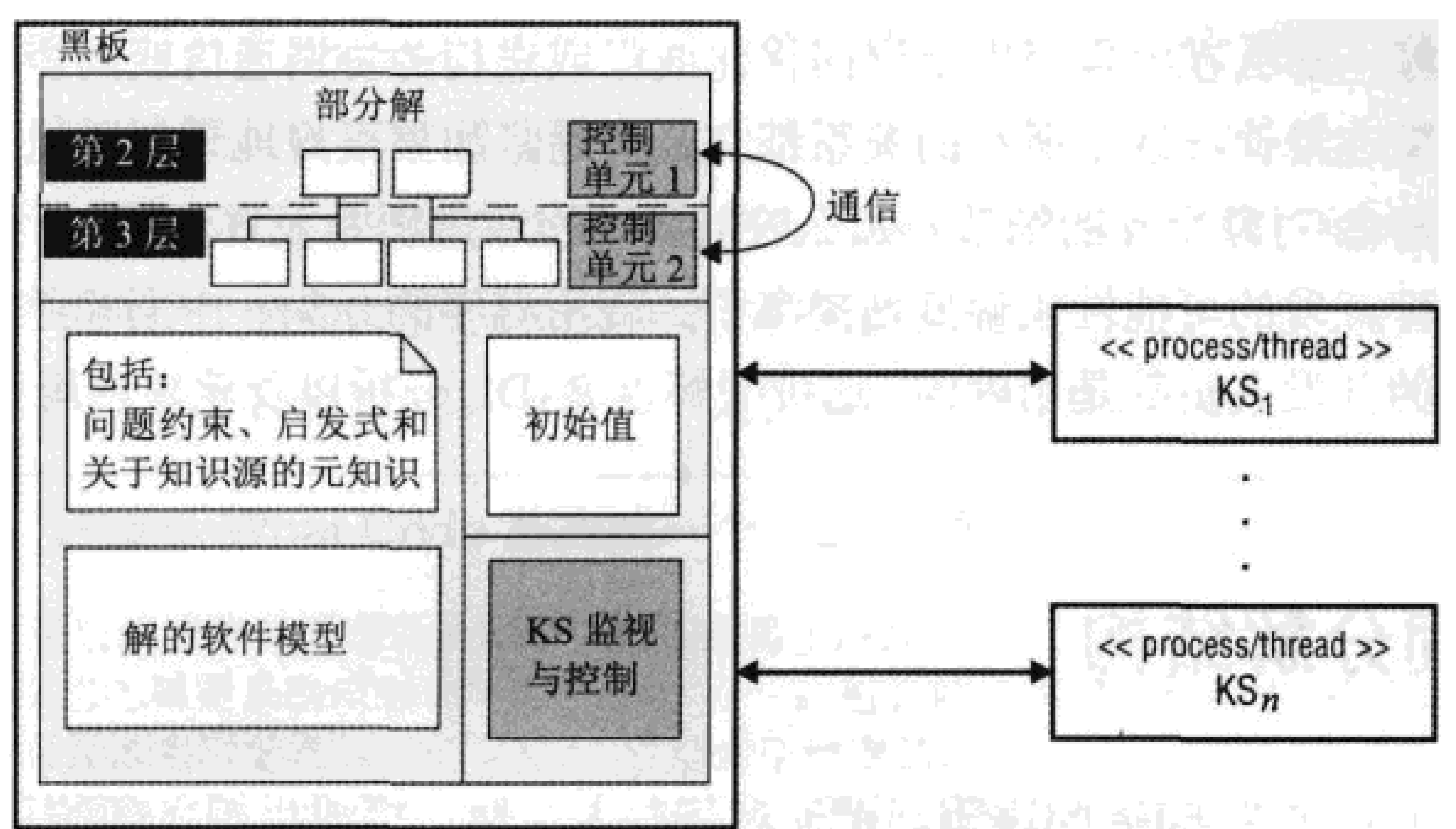
STAPL 支持在较高级别上工作。当开发人员基于任务而不是线程来思考时，TBB 能够得到最好的使用。这里，我们更进一步，考虑应用程序的“故事”，参与者及对象如何自然地交互。在第 4 层和第 5 层的模型级别上得出工作模式。根据一个支持并发的著名架构(例如，黑板和多 agent)来理解并行或并发。

为了进一步说明这一点，我们将更仔细地查看本章所使用的示例的黑板控制策略。

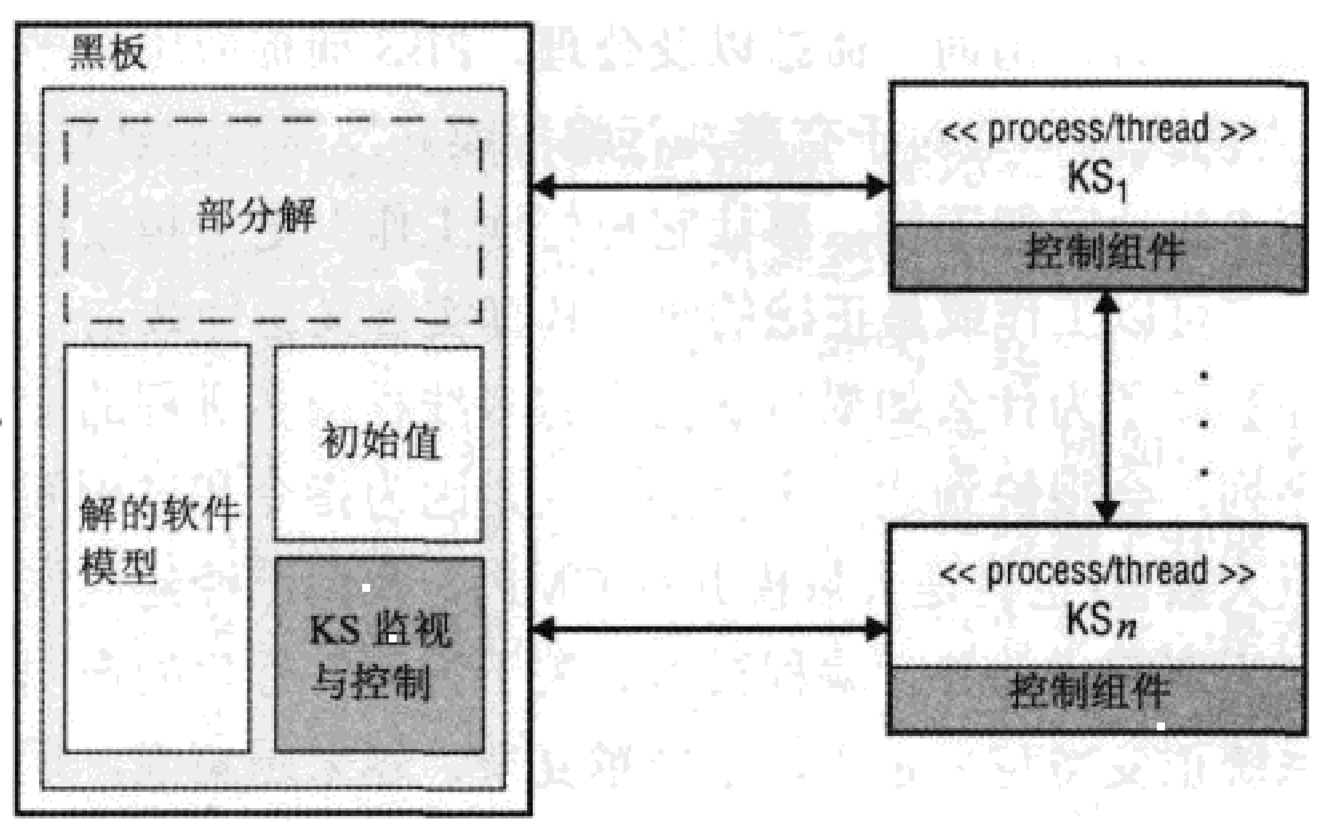
黑板的控制层涉及搜索解时选择使用哪些数据源以及聚焦在问题的哪个方面。这是一个焦点或关注层。这层控制聚焦在问题的特定领域并选择相应的知识源。在解决任何类型的问题时，主要的问题是从哪里开始以及解决问题需要哪些种类的信息。焦点/关注层评估问题的初始条件，然后控制使用哪个知识源以及它们从哪里开始。黑板知道有哪些可用的知识源，而且通常知识源接收消息或参数，这些消息或参数指示出它们应当如何继续，或者它们应当在解空间的什么位置开始进行查找。对于并行实现，这一层决定了 PADL 第 4 层的基本并发模型。通常对于黑板，基本并发模型是多程序多数据(MPMD)模型，因为每个知识源/问题解决者有着自身的专业领域。然而，问题的本质可能更适合流行的单层序多数据(SPMD 或 SIMD)模型。如果使用这个模型，控制层生成 N 个相同的知识源，但是给它们传递不同的参数。

下一层的控制涉及确定如何处理写到黑板的解或部分解。这一层的控制将会决定知识源是否能够停止工作或生成的解是可接受、不可接受或是部分可接受等。这一层的控制能够看到完整的黑板以及所有的部分或试验性的解。它指导着集体的总体问题解决策略。对于黑板的布局以及知识源的结构，黑板模型建议应当存在一个控制组件，但是没有指定它的结构应当是怎样的。有时候控制组件是黑板的一部分，有些时候控制组件是由知识源来实现的，在某些情况下，控制组件是由黑板外部的模块实现的，控制组件还可以由以上的任意组合来实现。知识源共同地查找某些问题的一个解。我们希望强调“一个解”是因为很多问题有着多个解。某些解可能比其他解位于解空间中更深的位置。找出某些解的成本可能要比其他的解高。某些解可能被认为不足够好。控制组件帮助管理知识源的集体查找策略。控制组件监视试验性的解或局部解，以确保知识源不是采用不实际的查找策略。控制组件会留心任何无限循环、死胡同或递归衰减。此外，控制组件还参与到为问题选择最佳或最适当的知识源。当知识源朝着解的方向有了进步之后，控制组件可能会替换掉某些知识源，同时指派其他的知识源。控制策略将会与知识源所使用的查找策略紧密相关。重要的是要记住每个知识源可能会使用不同的查找策略和问题解决技术。尽管它们作用在共同的黑板，知识源和问题解决者本质上是自治且自包含的。因此，这一层的控制同知识源有着双向通信。图 8-8 显示了在一个黑板架构中可能的控制配置以及它们的层级。

情况 1：
控制组件是黑板的一部分



情况 2：
控制组件是黑板和知识源的一部分



情况 3：
控制组件在黑板和知识源的外部

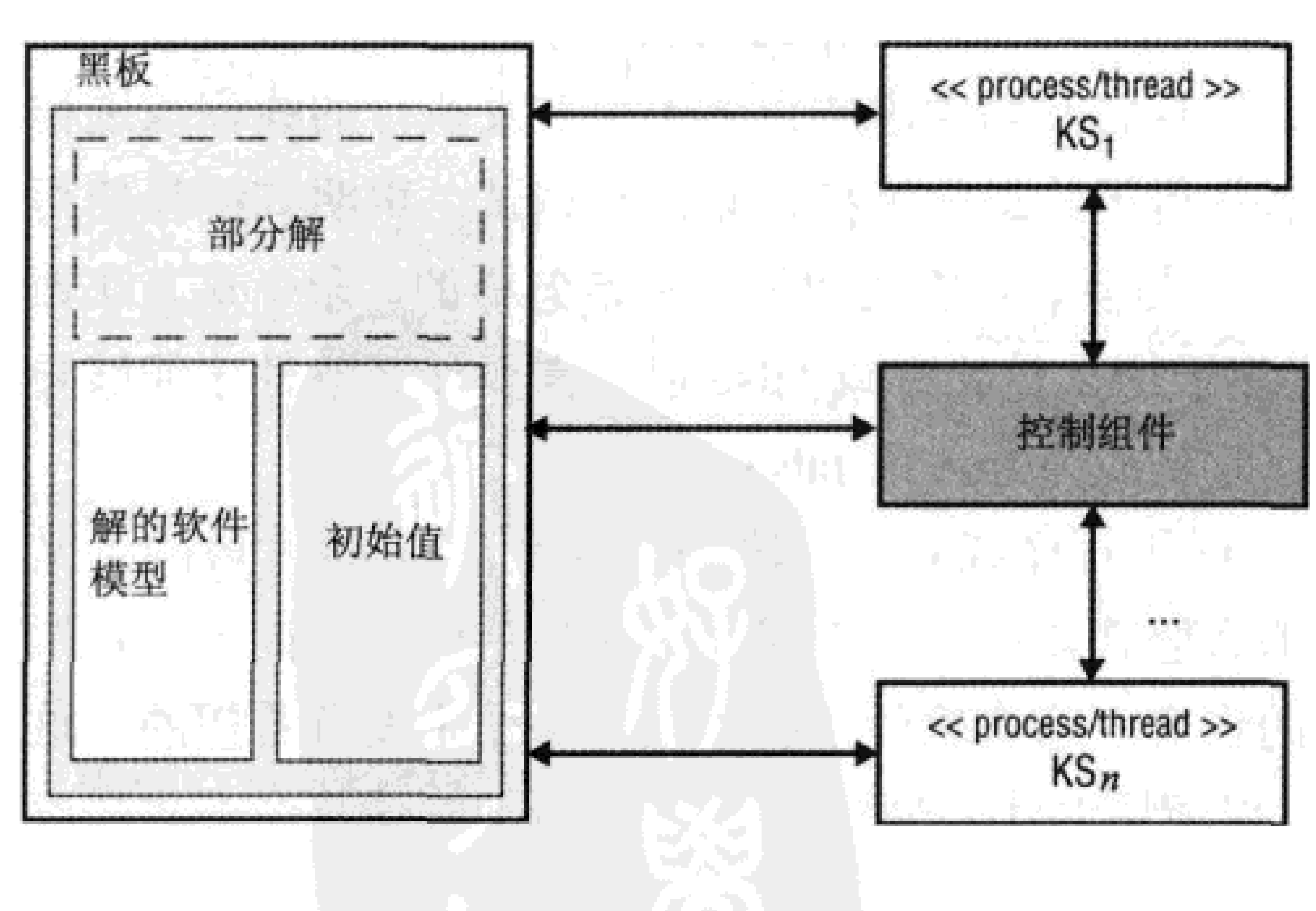


图 8-8

在图 8-8 中，注意黑板与知识源之间的关系。如果这个接口在设计层面就适当且完整地得到，则将知识源或 agent 映射到任务、线程或进程就会比较容易。如果在设计和领域级别上没有做好，那么在实现级别上试图优化线程或进程并协调通信或同步是不实际的。另一方面，如果在设计级别上所有的关系都完全清楚，如果很好地理解了工作模式，而且已经得到了参与者/对象交互的细节，那么映射到线程或进程是很直观的。

成功的软件开发的可能性大幅度地提高了。应用程序的开头、中间和结果，必须对于所有参与进来的开发人员都是明确的。这种理解以 PADL 分析以及软件应用程序的 PBS 作为开始。

8.3 谓词分解结构

谓词是为 true 或为 false 的语句。谓词对某些人、地、事或思想之间的关系进行陈述。应用程序的谓词分解结构(PBS)将应用程序分解为一组语句，它们描述了应用程序的断言和工作模式。应用程序的 PBS 包含应用到一个应用程序中的 agent、参与者、对象以及它们之间的关系的规则、约束、断言、谓词、命题以及公理。PBS 捕捉应用程序或软件片段的最具声明式的构造。这个声明式的构造对于有着并行编程或并发需要的软件是非常关键的。对于并行编程，老的格言“先让它能工作，再让它更好地工作”是一种生存需要。PBS 帮助您理解当您表示应用程序可以工作或是正确的时，其真实含义是什么。

贯穿本书，我们已经解释了为什么如果自底向上将多线程或多进程的应用程序考虑为并行指令或并行过程的序列，会很快到达一个极限，这是因为指令和过程间交互的复杂性以及最终可用的内核的巨大数量(也就是大规模并行 CMP)。我们已经建议您需要开始从过程式的思考并行编程转变为声明式的并行编程技术。关于为何并行化的声明式解释有益处的原因有很多。当今的系统正变得越来越大、越来越复杂、越来越集成。现在又在客户端和服务器端的计算机增加了多核的能力。命令式编程技术在这种复杂性级别下无计可施。另一方面，声明式技术是可扩展的，而且拥有被设计为处理复杂性的模型(例如，一阶逻辑分析、模型检测、布尔代数等)。声明式解释描述软件系统是什么以及它们的思想，而不是描述做什么以及如何做。一旦您开始考虑成百上千个并发执行的线程或进程，很难记住系统中正在发生些什么。这使得无法进行维护、测试和调试。声明式解释使得您可以考虑任何给定时刻系统中 agent、参与者、对象之间的关系是怎样的。命令式方法聚焦于做什么以及何时做，声明式方法聚焦于什么是 true 或 false 以及需要什么条件才能够导致应用程序中的一些语句或谓词在任何时刻成为 true 或 false。

PBS 结构帮助您把握应用程序或软件片段的“含义”。软件片段的含义非常重要，因为如果您理解了软件，那么您就知道如何在保证可靠和正确的前提下，软件可以扩展和演化。如果您不了解软件的含义，那么软件维护和软件演化是没有希望的。这对于涉及并行编程的软件尤为突出。可以使用 PADL 分析和 PBS 来帮助完成从命令式顺序编程模型向声明式语义设计模型的转变。

8.3.1 示例：Guess-My-Code 游戏的 PBS

您已经能够编写一个 guess-it 应用程序，该程序使用并发执行进程，然后这些进程会分解成并发执行的线程，以在时间期限内进行足够多的猜测并赢得游戏。你可以考虑您将需要多少个并行线程或进程才能够在 2 分钟之内遍历 400 万个编码。您还可以聚焦在进程和线程必须采取怎样的动作。然而，这更多的是问题的实验方法。对 guess-what-code-I'm-thinking-of 游戏的 PBS 分解将看上去更好些。示例 8-1 是游戏的一个 PBS 分解。

示例 8-1：Guess-What-Code 游戏的一个 PBS 分解

分解 1：如果您的猜测是正确且及时的，则您赢得了游戏。

分解 2：如果您的猜测是由只包含字符（a～z，0～9）组合的 6 字符编码组成，则您的猜测是正确的，允许考虑重复，而且该编码是我的一个 agent 已经递交给我的。

分解 3：如果猜测正确，而且发生在 2 分钟之内，则您的猜测是及时的。

分解 4：如果有足够的 agent 进行查找，则对编码进行穷尽查找将会成功。

分解 5：*N* 个 agent 足以从四百万个编码的样本中在 2 分钟内找到正确的编码。

分解 6：如果每 15 秒编码改动 1 次，则需要 4*N* 个 agent 才能够从四百万个编码的样本中在 2 分钟内找到正确的编码。

示例 8-1 中的 PBS 由构成应用程序的规则、陈述和谓词所组成。

这些语句要么为 true，要么为 false。如果您设计的代码正确地表达了这些语句，那么如果最初的语句是正确的，则应用程序会是正确的、起作用的。另一方面，如果一条或多条语句为 false，那么底层的代码也将会处于错误状态。例如，分解 5 声明 *N* 个 agent 是足够的。如何知道 *N* 个 agent 是足够的呢？如果在这个级别上开展工作，那么后续转换成并发执行的指令或线程将会效率更高而且正确。*N* 应当为多大，是在 PADL 分析和 SDLC 设计活动期间应当发现的信息。它不应当是在线程管理级别上处理的反复试验过程的一部分。尽管实例 8-1 中展现的 PBS 是简化的，但它仍然是一个用来说明 PBS 的意义的完整实例。注意分解 6 处理动态改变的编码，而且重新提交了整个猜测数据集。

8.3.2 将 PBS、PADL 和 SDLC 联系起来

一旦 PBS 完成，而且开发组对应用程序的 PBS 满意，则使用 PBS 来帮助进行在第 5 层完成的应用程序架构选择以及在 PADL 的第 4 层完成的并发模型的确定。在 SDLC 方面，PBS 是在需求收集、分析和设计活动期间完成的。理想情况下，它是在 PADL 的第 5 层之前完成的，但是有时候会同第 5 层分析并发执行。在一个实际的 PBS 分解中，陈述会被精化和阐明，直到应用程序的解或者服务的陈述中不存在歧义为止。此外，由于 PBS 由命题、谓词、陈述等组成，因此在开发任何代码之前可以使用定理证明方法或模型检测技术来判断应用程序的正确性。这种在代码实现之前检测正确性、含意、声明式语义的能力，当您使用更加复杂的系统和大规模并行多核计算机时，会变得更加关键。

8.3.3 对 PBS 进行编码

领域组件，即应用程序领域的 agent、参与者和对象，应当与 PADL 第 3 层中的 C++应用程序框架、类层次结构、领域类等联合。

对于 PBS 中提到的每个人、地点、事件、思想，应当存在一个对应的 C++组件，该组件实现了概念以及概念之间的联系。这些组件将会是领域组件，而且它们在应用程序架构中直接可见。这是因为应用程序架构之所以被选中，是因为它适合于 PBS。可以在这一层上进行应用程序的正确性和完整性的查看和验证。一旦提供了领域类，它们可以被 TBB、C++并发编程库或类似 STAPL 的框架中的组件所支持。使用 PADL 和 PBS 有助于生成正确、可演化、可高效维护的应用程序。

8.4 小结

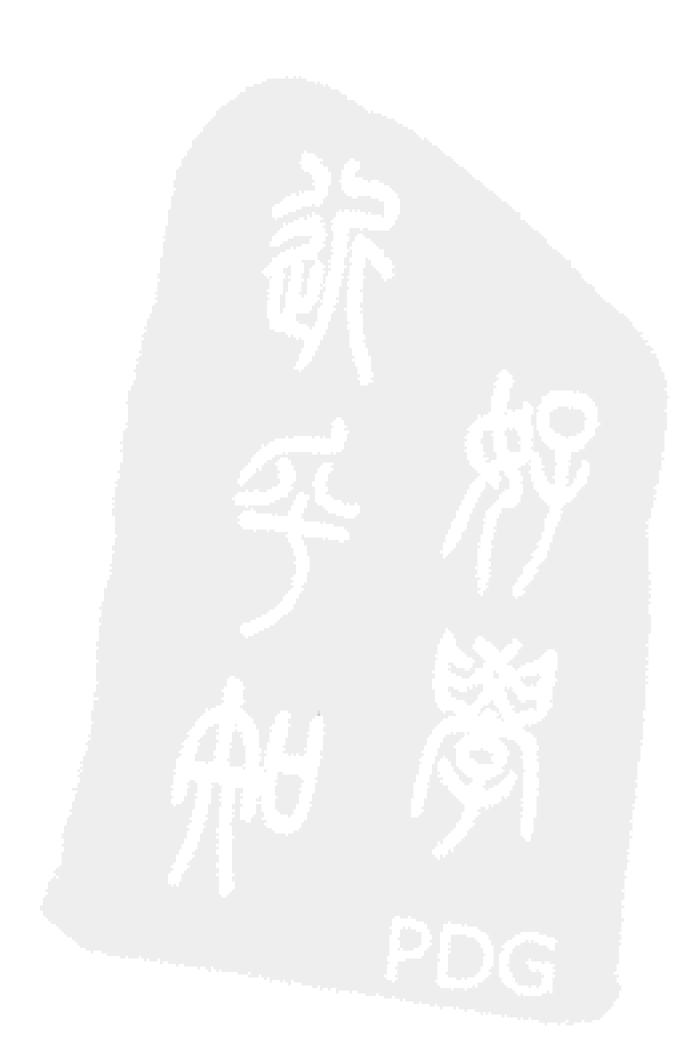
本章解释了当最初的软件开发要求中有着并发需求，或者当解决方案分解中显式要求或隐式要求并行性时，如何进行应用程序设计。我们讨论了两种主要方法。

- 首先，我们引入了并行应用程序设计层(PADL)，这是一个 5 层的分析模型，我们在 CTEST 实验室进行 SDLC 的需求分析、软件设计和分解活动中使用它。我们还使用 PADL 模型来绕过由自底向上方法进行并行编程所导致的大部分的复杂性。本章还展示了处理并行性的自顶向下的架构方法，而不是自底向上、面向任务的命令式方法。还介绍了多 agent 架构和黑板架构，它们是在应用程序层上的两个主要的且被充分了解的并行编程范型。多数成功的软件会经历持续的变化。如果软件有着要求并发和同步的组件，而且在开始时没有选择适当的应用程序架构，则软件的自然演化会非常有挑战性。PADL 分析模型确定多 agent 和黑板架构是两个广为人知且被充分了解的架构，而且能够在软件进化的情况下很好地工作。最终，应用程序的并发会通过底层操作系统原语(如线程和进程)来实现。如果应用程序架构是清晰的、模块化的、可扩展的、能被充分理解的，则成功转换到易于管理的操作系统原语的可能性比较大。另一方面，选择错误或较差的应用程序架构会导致脆弱的、易出错的软件，而且该软件不能够轻易地改变、维护、演化。尽管对于任何种类的计算机应用程序都是如此，但是当软件设计到并行编程、多线程或多进程时，情况更为突出。
- 本章还介绍了应用程序的思想的谓词分解结构(PBS)的概念。应用程序的 PBS 将应用程序分解为一组描述应用程序的断言和工作模式的语句。应用程序的 PBS 包含规则、约束、断言、谓词、命题和公理，它们可以应用到应用程序中的 agent、参与者和对象以及它们之间的关系上。PBS 用来捕获应用程序或软件片段的最具声明式的构造。这种声明式的构造对于有着并行编程或并发要求的软件非常重要。PBS 结构帮助捕获应用程序或软件片段的“含义”。软件片段的含义非常重要，

因为如果您理解了软件，就可以知道如何对它进行扩展和演化，与此同时保证它的可靠性和正确性。如果您不能够理解软件的含义，则软件维护和软件演化是不可能的。

当前的系统正变得越来越大、越来越复杂、集成度越来越高。现在我们正在客户端和服务器端的计算机增加混合多核能力。在这种级别的复杂性下，命令式编程技术无法应对。另一方面，声明式编程技术是可扩展的，而且拥有被设计为应对这种复杂性的模型。声明式解释描述了软件系统是什么以及其含义，而不是描述它们是做什么的以及如何做的。使得您考虑关于在任何指定时刻，agent、参与者和对象之间的关系是怎样的。如果在设计级和领域级没有搞清楚，则在实现级试图对线程或进程进行优化并协调通信或同步是不切实际的。相反，如果在设计级所有的关系都非常清楚，而且很好地理解了工作模式，并且已经实现参与者/对象之间的交互，则映射到线程或进程将会是非常直观的。这样，软件成功部署的机会就会大很多。所有涉及的开发人员都应当对应用程序有很好的理解，这种理解始于软件应用程序的 PADL 分析和 PBS。

下一章将讨论如何使用 UML 标记来对应用程序的并发行为进行建模和记录。本书在前面章节中已经使用了 UML 类图、顺序图和活动图。下一章将讨论一些基本的对类进行建模的 UML 制图技术，逐渐地介绍并发标记以及类间行为，最终介绍有着并行性的系统的架构。



第9章 对要求并发的软件系统进行建模

系统模型是为了研究系统而收集到的信息的主要部分，建立模型的目的是为了让系统的开发人员和维护人员可以更好地理解该系统。当系统被建模后，便可确定实体、属性及系统执行的活动的边界和标识。建模是任何系统的设计过程中的重要工具。开发人员完全理解他们正在开发的系统是至关重要的。建模可以揭示隐藏的并发和可以利用并行的机会。

本章将会为您介绍如何使用 UML 对并发系统进行可视化和建模。我们将从以下 3 个方面讨论对并发系统进行可视化和建模时所使用的制图(diagramming)技术：

- 从结构角度
- 从行为角度
- 从架构角度

注意：

本章中作为示例使用的类、对象、进程、线程和系统是用于说明目的，并不一定反映了实际系统中实际的类、对象或结构。本章不应被视为对 UML 的初级介绍，而是对本书中使用的图的介绍，介绍的重点是用于设计并记录利用并发的系统的 UML 标记。

9.1 统一建模语言

统一建模语言(Unified Modeling Language，UML)是一种图形化语言，用于建模、可视化、设计和记录系统中的工件。UML 是标准化规范语言，用来沟通和建模不同系统的范型，例如，面向对象系统、面向 agent 系统及事件驱动系统。它使用符号和标记来从不同的观点和角度来表示系统中的组件。

UML 之所以被称作“统一的”，是因为它带来 3 种最卓越的建模语言(Grady Booch 的 Approach、Ivar Jacobson 的 Object-Oriented Software Engineering(OOSE)和 James Rumbaugh 的 Object Modeling Technique(OMT))。虽然每种语言都是一个完备的系统，但各自都有各自独特的焦点。每种语言又都有其缺点：它们都不能作为复杂系统的通用建模语言。表 9-1 给出了这些建模方法的简要定义以及这些方法对 UML 标准的贡献。UML 的目标是将这些方法统一成一个定义，从而能够为用户提供一个建模系统，该系统能够：

- 利用面向对象技术捕捉软件系统中概念上的组件以及可执行的组件
- 捕捉从简单到复杂的关键任务(mission-critical)系统

无论这些用户是人或计算机，情况都是如此。在 1995 年早期，UML 的草案初稿发布，同年随后发布了新的版本。1997 年，对象管理组(Object Management Group，OMG)——一个国际性公司联盟——积极促进了面向对象范型的发展，发出了标准建模语言的征求建议书，并提供了 UML 1.0。从那时起，UML 就成为事实上的国际性标准建模语言，并由 OMG 负责其修订和扩展。多年来，许多人为该标准做出了贡献，使得它不仅被面向对象(OO)软件系统的建模所采纳，而且也被并发及分布式系统、工程问题和业务结构及流程所采纳。OMG 在 2007 年发布了 2.0 版本。

表 9-1

UML 语言原语	类　型	描　述
Things	结构的 • 类 • 接口 • 协作 • 用例 • 活动类 • 组件 • 节点	模型中的名词 模型中表示概念上的或实际的元素的静态部分
	行为的 • 交互 • 状态	模型中的动词 模型中表示随时间而发生的行为的动态部分
	分组 • 包	模型中的组织部分 这些是模型中可以发生分解的最高级别
	注释	模型中的解释部分 用于描述模型中元素的注释
Relationships	依赖关系	对一个元素的修改会影响其他元素
	关联关系	元素间结构上的连接(整体-部分)
	泛化关系	子元素是父元素的特化
	实现关系	一个元素满足了另一个元素的契约(contract)
Diagrams	• 类图 • 对象图 • 用例图 • 顺序图 • 协作图 • 状态图 • 活动图 • 组件图 • 部署图	

UML 文法基于 3 个语言原语。

- **Things**：模型中最基本的组件。
- **Relationships**：将 things 关联起来。
- **Diagrams**：定义 things 以及它们之间的 relationships 的集合。

表 9-1 显示了 3 个语言原语如何被分解成不同的组件和类型。可以看到，UML 是一种全面的语言，可以用多种方式来表示一个系统。本章的重点是 UML 语言中为有着并发行为的系统进行建模时可以利用的标记和图。

9.2　对系统的结构进行建模

在对系统的结构进行建模时，重点在于系统的静态部分，如对象、类、它们的属性、服务、组织、组合及系统中的这些部分与系统其他实体间的关系。

9.2.1　类模型

类是面向对象系统中的基本的软件组件。类是包括其属性和行为的构造(construct)的模型。它被用作有着相同的属性和行为的一组事物的定义。类可以对一些概念性的事物、真实世界的物理实体或软件构造进行建模。

- 对某种**概念性的**事物进行建模的类是用于分析或实验目的的过程、概念或想法的按比例表示。它是按比例表示的，是因为创建完整的模型是非常困难的，或者是不可取的。分析可能主要集中在某个特定领域，因此，全尺寸模型(full-scale model)是不必要的。分子模型是类的一个例子。分子的结构和化学键的距离和角度是类的属性，化学反应和过程是模型的行为。这种分子类模仿了真实世界中的分子的特性，用于预测和分析分子行为的目的。
- 类可以对**真实世界的物理实体**、过程、任务或想法进行建模，目的是作为替代品。在这种情况下，模型不是按比例表示的，而是复制现有的实体的所有功能。这种类型的类对真实世界中对应的事物进行建模，因为软件模型可以更有效率、更方便或更有效。例如，类可以对计费系统或计算器建模。calculator 类的属性可以包含显示、计算和输入机制。calculator 类必须能够解析输入、验证输入、执行期望的算术运算，然后显示结果。
- 类可以对**软件构造**进行建模。在这种情况下，模型仅在软件系统中有意义。对软件构造进行建模的一个例子是位图图像类。位图图像类有一个标题、位平面(bit plane)数目、大小和代表图像中每个像素的比特向量。类必须能够对图像进行显示、读取和调整大小。被建模的其他软件构造是数据类型。数据类型，如浮点型、整型和布尔型，都具有属性及一组操作。这些软件构造被用作实用工具和支持类，以对更大的系统进行建模，如计费系统或简单的计算器。它们可被用于各种领域的任何系统中。

正如您所看到的，类可以被用来对软件系统中多种类型的实体进行建模。在建模过程中，能够基于系统的结构、行为或架构视图来确定构造的能力很重要，甚至有必要基于系统中如何使用类来对类的类型进行建模。有一些类在被用作为其他类提供接口策略的蓝图时是有用的，而其他类在用作基类或祖先类时有用，或者某个类只是用于某个特定领域。表 9-2 列出了常见的类的类型。

表 9-2

类	描　述
抽象(Abstract)类	一个为所有后代确定蓝图的基类；对象不能被声明为这种类型
具体(Concrete)类	一个独立的类，表示祖先-后代谱系的末端
接口(Interface)类	更改或增强另一个类或类集合的接口的类
节点(Node)类	提供继承与多态的基础的类，而且不包含纯虚函数
领域(Domain)类	被创建为用来模拟现实或某个领域内的一些实体的类
支持(Support)/实用工具(Utility)类	对各个领域的应用均有用的程序
集合(Collection)与容器(Containers) 类	一组对象的通用载体，具有已经定义的一组操作来对这些对象进行访问
模板(Template)类	类型被参数化的类，具有已经定义的一组操作来访问和操纵对象
数据类型(Datatypes)类	类型与它的操作

例如，假定您在对一个子系统进行建模，该子系统可以确定文本文件中无法识别的单词。子系统从文本文件中提取单词，确定该单词是不是无法识别的，如果是无法识别的，则将该单词存储在一个全局容器内。这个子系统中的一些类如下所示。

- **Word expert**：确定一个单词是否可被识别，若无法被识别，则将单词写入到容器
- **Unrecognized words list**：将给定的字符集从文件中删除
- **Dictionary lexicon**：包含所有可被识别的单词
- **Text file**：包含待识别的单词
- **Main agent**：创建多个 word expert
- **Unrecognized word container**：包含所有无法识别的单词
- **Misspelled and morphology agents**：过滤拼错的单词和来自位于无法识别单词列表中的词形的 agent

每个类可被确定为类的一种类型。

- **实用工具类和支持类**：Filtering agents 和 text file
- **领域类**：Dictionary lexicon 和 word expert
- **容器类**：Unrecognized word container

9.2.2　类的可视化

UML 提供了类的图形化表示。类的表示，即类图标(class icon)，可以显示类的属性、服务和语义。UML 还提供了类的类型的表示，如数据类型类、接口类、模板类和节点类。类的最简单的表示就是包含类名的矩形框。简名(simple name)就是类的名字；路径名(pathname)是以包含该类的包的名字作为前缀的类名。类名是包含任意数目的字母和数字的文本，可以包含除了冒号以外的标点符号，冒号被用来在类名中将包区分开。类名通常是来自被建模的系统的词汇表的名词或名词短语。类名中每个单词的首字母都应大写：

- DictionaryLexicon
- WordExpertAgent
- Filter::MorphologyAgent

1. 类的属性、服务和职责的可视化

类图标可分为 3 个水平区划(compartment)。顶端区划包含类名。接下来的两个区划包含类的属性和为类的用户提供的服务。底端附加的区划可用来描述类的职责。类的职责是用几个短句描述的类的责任。例如，以下是 WordExpertAgent、UnrecognizedWords 和 DictionaryLexicon 类的基本职责。

- WordExpertAgent
  - 确定一个单词是否可被识别
  - 将无法识别的单词写入一个全局容器
- UnrecognizedWords
  - 通过使用 MorphologyAgent 和 MisspelledAgent 来确定无法识别的单词是否是可被识别单词的词形变化或拼写错误
- DictionaryLexicon
  - 包含所有可被识别的单词
  - 包含每个单词的词性、同义词与词形部分
  - 当给出一个单词，如果这个单词在词典中，则返回 TRUE

这些职责可以转变为类的属性和服务。属性是类的被命名的性质，服务或操作描述了类的行为。图 9-1 显示了 DictionaryLexicon 的职责是如何被用来创建类的一些属性和服务的。然后，属性被转换为数据类型和数据结构，服务被转换为方法。属性和服务名的第一个单词的首字母小写，其他单词的首字母大写。

属性、服务和责任区划可分别用 attributes、services 和 responsibilities 来标注，以区分每个区划。如果不显示属性或服务，那么这些区划显示为空。图 9-2 显示了表示一个类的各种方法，以 DictionaryLexicon 类为例。

DictionaryLexicon

- contains all the recognized words
- contains synonyms, parts of speech, and word forms for each word
- when given a word, will return TRUE if the word is in the lexicon

DictionaryLexicon

ATTRIBUTES

recognizedWords
synonyms
partOfSpeech
plurals
wordForms
lexicon

SERVICES

recognizedWords()
synonyms()
plurals()
wordForms()
validateWord()

DictionaryLexicon

ATTRIBUTES

recognizedWords: list<string>
synonyms: list<string>
partOfSpeech: string
plurals: list<string>
wordForms: list <string>
lexicon: map < string, vector <WordInfo> >

SERVICES

recognizedWords(): list <string>
synonyms(Word: string) : list <string>
recognizedWords(Word: string) : list <string>
synonyms(Word: string): list <string>
plurals(Word: string): list <string >
wordForms(Word: string): list <string >
validateWord(Word: string): Boolean

图 9-1

DictionaryLexicon

(a) 最简单的类表示

DictionaryLexicon

ATTRIBUTES

recognizedWords: list<string>
synonyms: list<string>
partOfSpeech: string
plurals: list<string>
wordForms: list <string>
lexicon: map < string, vector<WordInfo> >

SERVICES

recognizedWords(): list <string>
synonyms(Word: string) : list <string>
plurals(Word: string): list <string >
wordForms(Word: string): list <string >
validateWord(Word: string): Boolean
addWord(WordInfo: struct)
removeWord(Word: string)
checkWord(Word: string): Boolean

(b) 显示属性和服务的类表示

DictionaryLexicon

SERVICES

recognizedWords(): list <string>
synonyms(Word: string) : list <string>
plurals(Word: string): list <string >
wordForms(Word: string): list <string >
validateWord(Word: string): Boolean
addWord(WordInfo: struct)
removeWord(Word: string)
checkWord(Word: string): Boolean

(c) 显示空属性区划以及服务的类表示

DictionaryLexicon

Responsibilities

-- contains the recognized words

-- contains the part of speech, synonyms, and word forms for each word

-- when given a word, will return TRUE if word is a recognized word

(d) 显示空属性和空服务区划以及职责的类表示

图 9-2

PDG

- 图 9-2(a)用最简单的表示方法显示类。
- 图 9-2(b)显示了类名和类的属性及服务。
- 图 9-2(c)显示了类名和服务；属性区划为空表示存在属性但未被显示。
- 图 9-2(d)列出了类的职责。

属性区划可以指定数据类型和/或对象属性的默认值(如果有默认值)：

```
word : string
word : string = "Car"
```

WordExpertAgent 和 DictionaryLexicon 类的属性的数据类型和数据结构可被显示为：

```
synonyms : map < string,vector < string > >
synonymIterator : map < string,vector < string > > ::iterator
```

显示方法时可以带有参数和返回类型：

```
synonym( & X : map < string,vector < string > > ) : void
partOfSpeech(string & X) : string
```

synonym( )返回单词的同义词。DictionaryLexicon 是用来对单读领域的字典进行建模的类。每个词的同义词存储在一个 vector 中。map 容器将一个 string(可被识别的单词)映射到包含同义词的 vector。synonym( )返回 void，而 partOfSpeech( )方法以 string 类型返回单词的词性。

2. 使用属性和操作的性质

可以使用性质(property)来描述属性和方法。这些性质帮助描述如何使用属性和方法。属性的性质可以是不变的或者可变的：

- changeable
- addOnly
- frozen

有 4 个性质被用来定义方法：

- isQuery
- sequential
- guarded
- concurrent

表 9-3 列出了这些性质，并对每个性质进行了简要介绍。

表 9-3

属性的性质	描　　述
{changeable}	这种类型的属性在值的修改方面没有限制
{addOnly}	对于多重性大于 1 的属性，可以增加额外的值；一旦创建一个值，则这个值不可以被删除或更改
{frozen}	对象被初始化后属性的值就不可被改变

(续表)

服务的性质	描　述
{isQuery}	当执行完后，对象状态不改变；返回一个值
{sequential}	使用同步来确保对这个方法的顺序访问；对方法的多个并发访问危害对象的完整性
{guarded}	对方法的同步顺序访问内置在对象中；可确保对象的完整性
{concurrent}	允许多个并发访问；可确保对象的完整性

sequential、guarded 和 concurrent 这 3 种性质关注的是涉及并发的方法。例如，可以创建多个 WordExpertAgent 对象，然后将它们分别传递给线程。每个 WordExpertAgent 对象的方法将无法识别的单词写入一个共享容器中。写入共享容器的方法有一个临界区。所谓临界区是这样的代码区域，该区域中的代码访问一个共享的、可修改的对象。当方法运行到其临界区内时，其他访问同一个对象的方法不应当位于它们的临界区中。这些性质(sequential、guarded 和 concurrent)通过明确规定谁负责同步来标记和管理拥有临界区或自身即为临界区的方法。

- **sequential** 性质描述了由方法的调用方(caller)负责同步的并发访问。为了同步对共享对象的访问，调用对象对互斥量进行测试。如果互斥量正在被使用，那么调用对象等待，直到互斥量可用。一旦互斥量可用，调用对象设置互斥量，然后可以执行修改对象的操作。一旦操作结束，互斥量被重置。sequential 操作不能保证对象的完整性。
- **guarded** 性质描述了同步内置在共享对象中的并发访问；访问是顺序的。
- **concurrent** 性质描述了允许同时使用的方法。

性质为 guarded 或 concurrent 的方法保证了共享可修改对象或数据的完整性。图 9-3 显示了 sequential、guarded 和 concurrent 的方法中同步发生的位置。

以下是标记了性质的属性和方法的例子：

属性

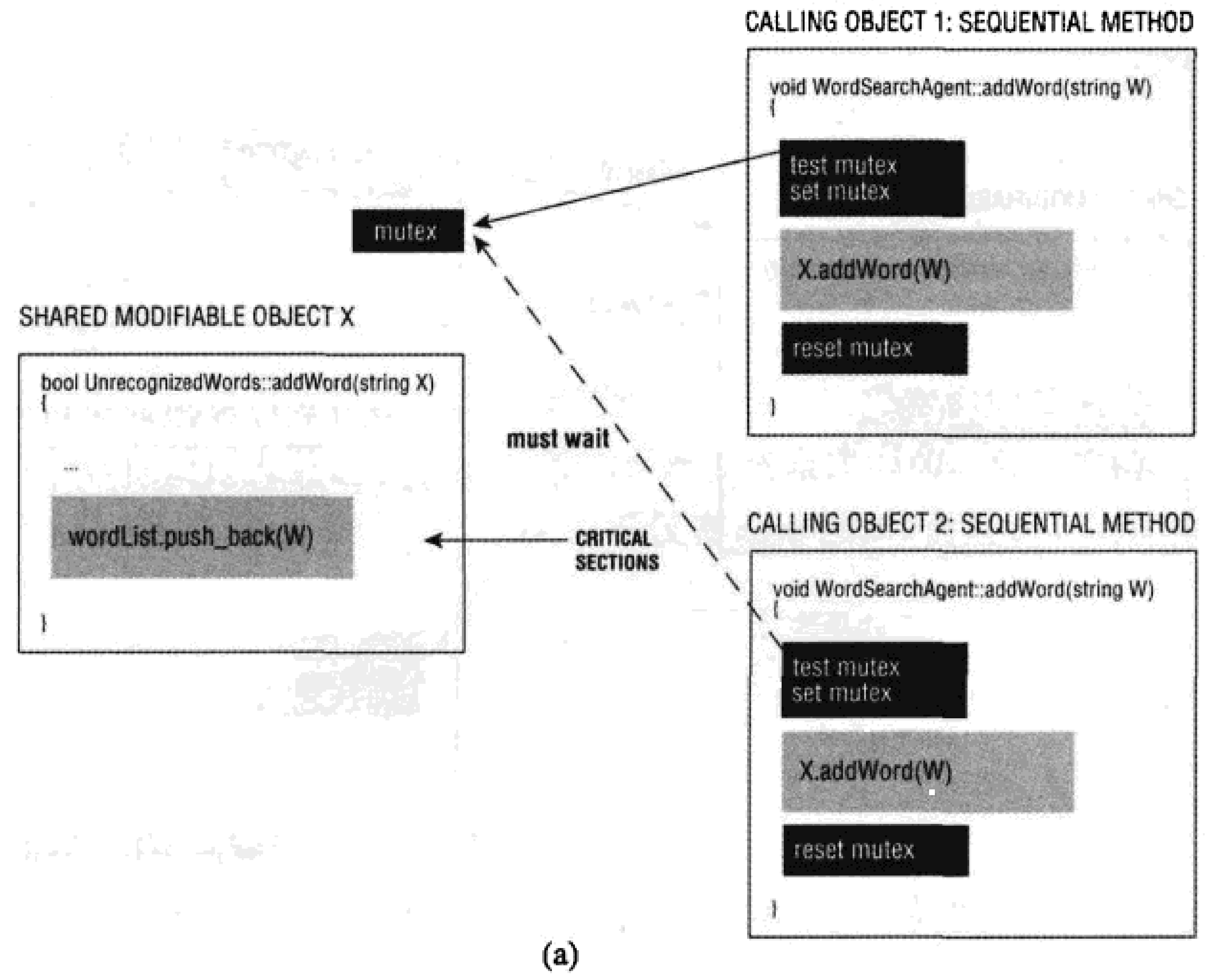
```
wordList : vector < string > {changeable}
```

操作

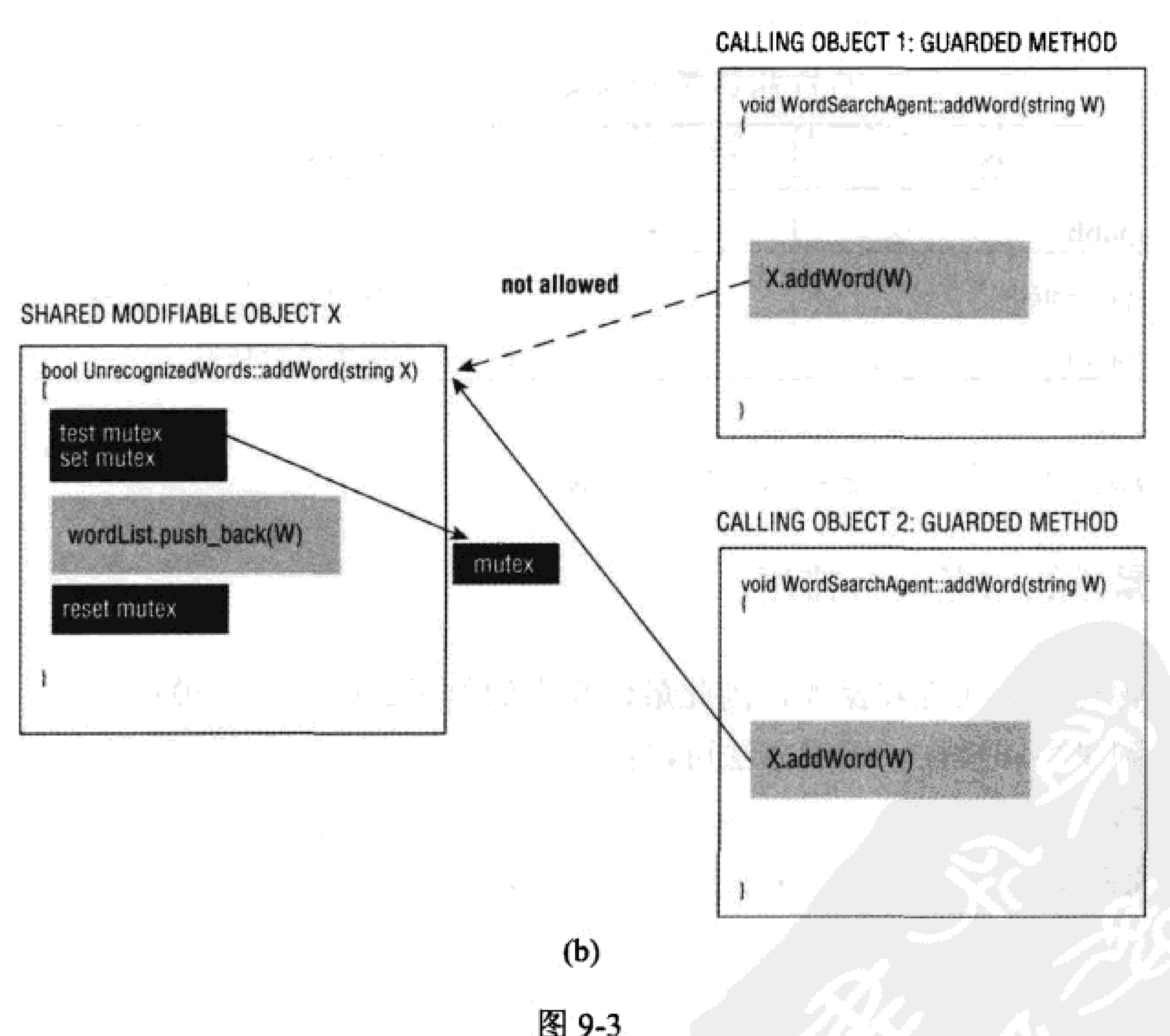
```
addWord(word : string) : boolean {guarded}
nextWord(void) : string {isQuery, concurrent}
```

本例中的属性和操作是由包含所有无法识别单词的 UnrecognizedWords 类定义的。WordSearchAgent 对象向 addWord( )传递一个单词，该方法将单词加入到列表中并返回 TRUE。addWord( )改变了对象的状态。UnrecognizeWords 对象被所有的 WordSearchAgent 对象所共享。因此，addWord( )的性质是 guarded。通过使用互斥量将同步内含到对象中。另一方面，nextWord( )返回列表中的下一个单词。调用这种方法不会改变对象状态；因此，由于操作不改变对象的状态，这种方法可使用 isQuery,concurrent 性质。

可以看到的另一个重要性质是属性和操作的可见性(visibility)。可见性性质描述了谁可以访问属性或调用操作。使用一个字符或符号来表示可见性的级别。可见性映射到 C++和其他语言的访问说明符。表 9-4 列出了访问说明符和可见性符号。

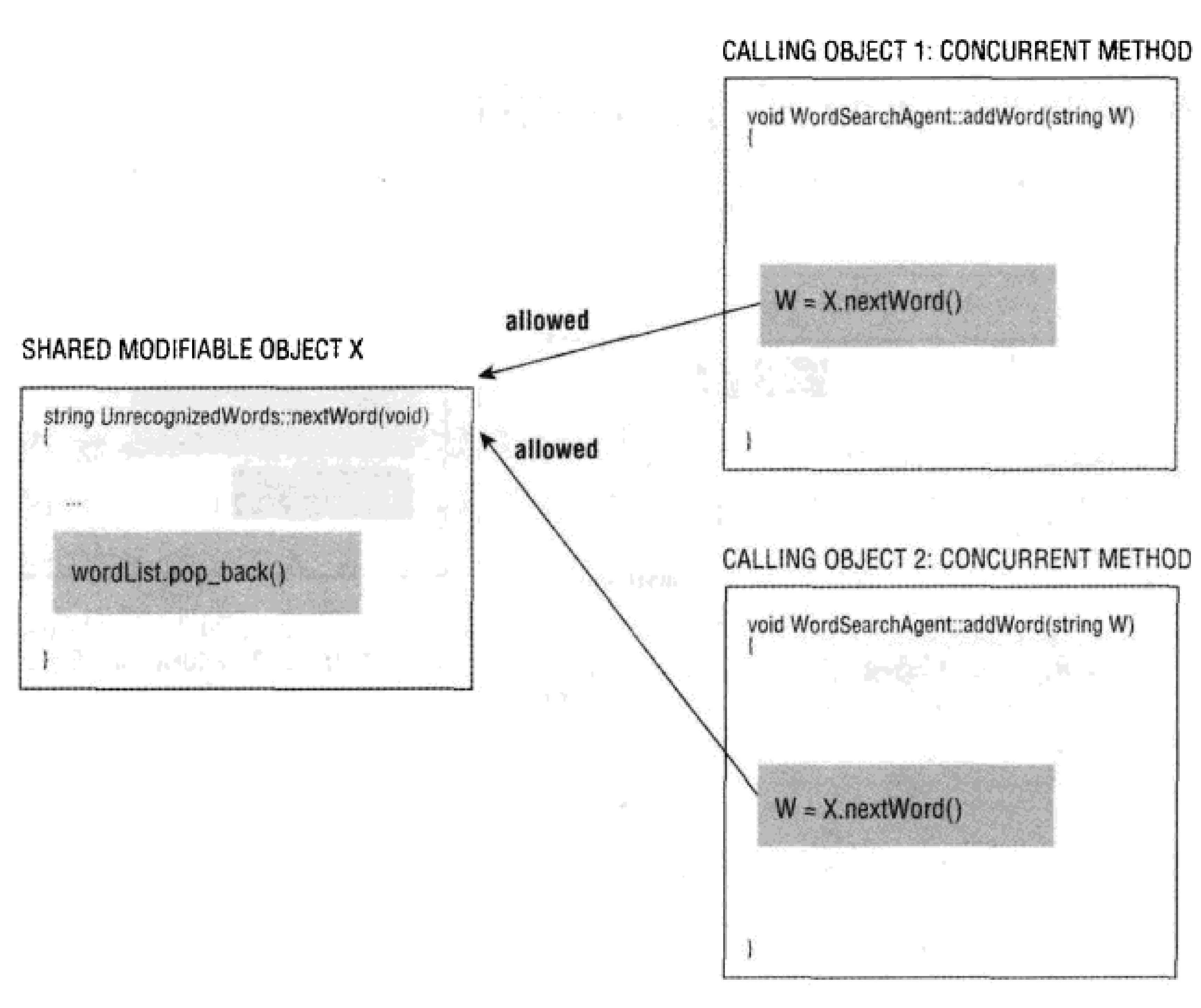


(a)



(b)

图 9-3



(c)

图 9-3(续)

表 9-4

访问说明	可见性符号
public	(+)可被任意访问
protected	(#)类本身与它的后代可访问
private	(-)只有类本身可访问

符号被加在服务名、方法名或属性名之前。

9.2.3 对属性和服务进行排序

有些类含有多种属性和操作，因此最好在它们的区划中对它们进行组织。排序可以帮助确定和找到属性和操作。组织方法可以：

- 通过访问
- 通过类别

通过访问权限来排序对用户很有用。它表达了哪些属性和操作是可被公众访问的。了解哪些成员是被保护的，可帮助需要通过继承来对类进行扩展或特化的用户。可以使用可

见性符号或访问说明符来通过访问组织属性和服务/方法。

基于类别对属性和操作进行排序有助于对类进行建模。通过类别进行组织可以帮助您确定类的基本操作是什么。您是在对一个 nice 类(nice class)进行建模吗？nice 类为类提供常用的功能性。下面是 nice 类应当定义的功能性：

- 默认构造函数
- 复制构造函数
- 赋值运算符
- 等号运算符
- 析构函数

复制构造函数、赋值运算符和析构函数可由编译器为需要它们的类生成，但不对它们进行定义。有些软件设计人员认为非 nice 类具有严格受限的行为。可能的情况下，可重用的类应有一个 nice 接口。最小标准接口定义了不仅有 nice 接口，而且还有附加操作的种类，这些附加操作是：

- 输入和输出
- 哈希函数
- 查询
- 浅复制和深复制操作

尽管如此，对“最小标准接口”甚至是 nice 类仍有争议。例如，一个类可能对一个不需要任何输入或输出操作的对象进行建模。

可以根据领域的语言对属性和服务进行分类。如果您对一个类建模，属性和服务是由正被建模的对象所规定的。例如 DictionaryLexicon 类可能有基于关于词汇、词形、同义词等服务的类别。当您开始判断类需要什么属性和操作时，使用这些类别是很有用的。其他类别可能基于方法或属性的其他性质，例如：

属性

```
static
const
```

操作

```
virtual
pure virtual
friend
query
concurrent
guarded
```

为了显示类别名称，将它们放置在左右双尖括号之间(<<...>>)。

图 9-4 显示了为属性和最小标准接口使用可见性符号，或为操作/服务使用领域类别来为 DictionaryLexicon 类组织操作的不同方式的实例。在图 9-4(a)中，通过为类完成的功能对服务进行分类。图 9-4(b)中，通过类的领域语言(domain language)对服务进行分类。所有

的属性具有私有可见性。

DictionaryLexicon

ATTRIBUTES

- recognizedWords: list<string> {changeable}
- synonyms: list<string> {changeable}
- partOfSpeech: string {changeable}
- plurals: list<string> {changeable}
- wordForms: list <string> {changeable}
- lexicon: map < string, vector<WordInfo> > {changeable}

<<constructor>>
dictionaryLexicon()
dictionaryLexicon(lexicon: map <string, vector<Wordinfo> >

<<query>>
synonyms(Word: string) : list <string>
plurals(Word: string): list <string >
wordForms(Word: string): list <string >
recognizedWords(): list <string>
partOfSpeech(Word: string): string

<<assignment>>
addWord(WordInfo: struct)
synonyms(Word: string, s: list <string>)
plurals(Word: string, p: list <string>
partofSpeech(Word: string, p: string)
...

<<validation>>
validateWord(Word: string): Boolean
checkWord(Word: string): Boolean

(a) 属性根据可见性分类，服务根据功能性分类

DictionaryLexicon

ATTRIBUTES

- recognizedWords: list<string> {changeable}
- synonyms: list<string> {changeable}
- partOfSpeech: string {changeable}
- plurals: list<string> {changeable}
- wordForms: list <string> {changeable}
- lexicon: map < string, vector<WordInfo> > {changeable}

<<constructor>>
dictionaryLexicon()
dictionaryLexicon(lexicon: map <string, vector<Wordinfo> >

<<vocabulary>>
recognizedWords(): list <string>
addWord(WordInfo: struct)
validateWord(Word: string): Boolean
checkWord(Word: string) Boolean

<<word forms>>
synonyms(Word: string, s: list <string>)
synonyms(Word: string): list <string>
plurals(Word: string, p: list <string>
wordForms(Word: string): list <string>

<<part of speech>>
partOfSpeech(Word: string): string
partOfSpeech(Word: string, p: string)

(b) 属性根据可见性分类，服务根据领域语言分类

图 9-4

9.2.4 类的实例的可视化

对象是类的具体实例。对象具有标识，并对属性赋值；这可以通过使用 UML 标记进行描述。对象(类的实例)的最简单描述是使用包含加下划线的对象名的矩形。这被称作类的命名实例(named instance)。类的命名实例在显示时可以有类名，也可以没有类名。

myWordSearch	命名实例
myWordSearch:WordSearchAgent	带有类名的命名实例

由于对象的实际名字可能只有声明它的程序知道，您可能应该在系统文档中表现类的匿名实例(anonymous)，可以有或没有路径名。孤立实例(orphan instance)不显示类名：

:WordSearchAgent	匿名实例
myWordSearch:	孤立实例

类的实例可能还会在自身的区划中显示它们的当前状态、静态性质和动态性质。当对

象动态变化时，对象显示性质的新值。例如，图 9-5 显示了 myWordSearch:WordSearchAgent 对象的属性变化。为显示活动对象或当前对象，使用较粗的线。更具体地说，图 9-5 显示了 myWordSearch 对象的多个版本：

- 图 9-5(a)显示了用于实例的各种标记。
- 图 9-5(b)显示了实例的属性变化，但仅有一个实例是活动对象。
- 图 9-5(c)表示了被称作 multiobjects 的集合，集合中包含类的多个未被初始化的实例。

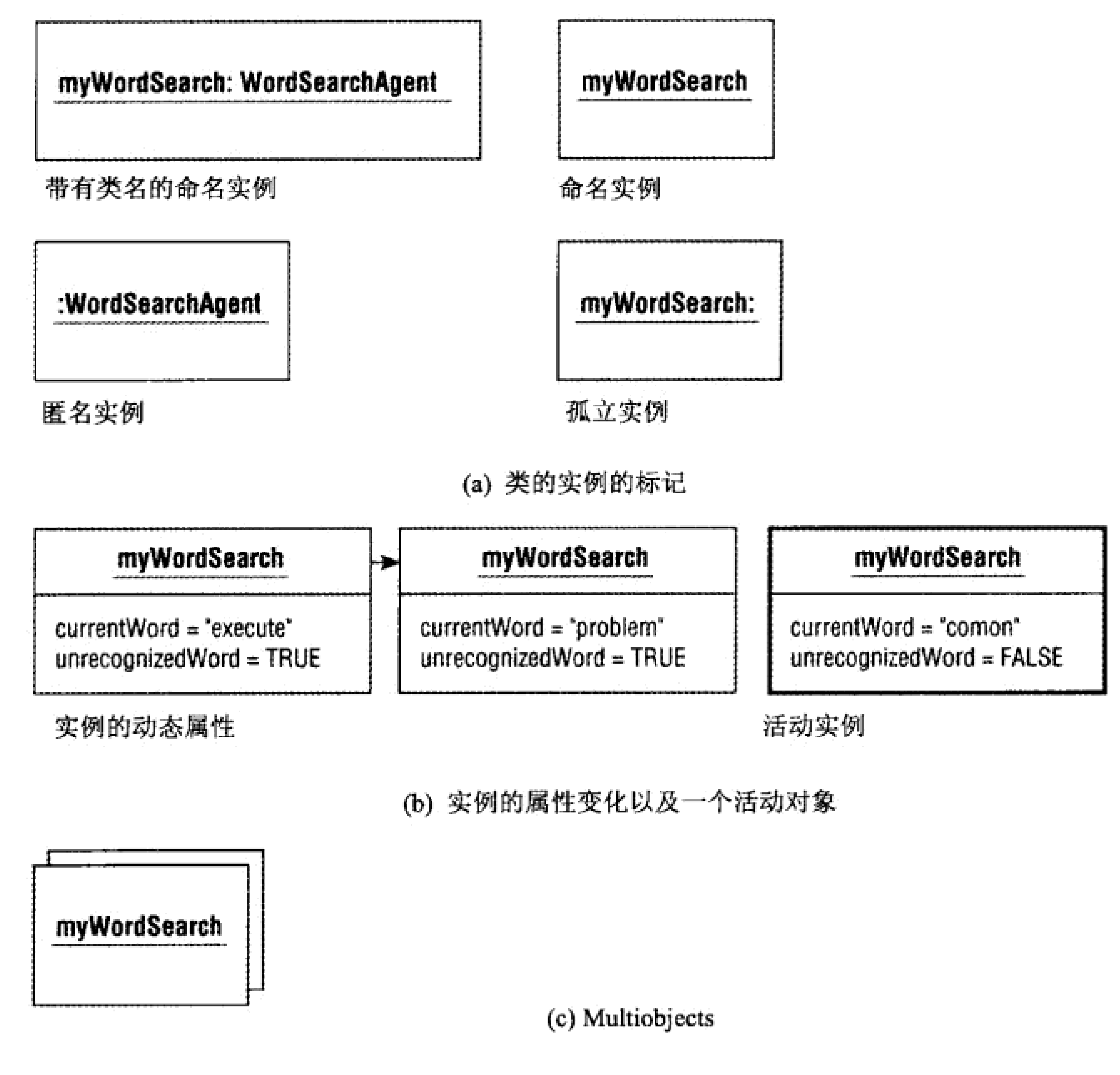


图 9-5

Multiobjects 是显示一个类存在的多个实例的方法。依据类的本质或两个类之间的关系，您可能希望限制某个类的实例的数目。多重性(multiplicity)是一种显示类的实例数目允许范围的规范。类的多重性可以标记在类图标或对象上。多重性放置在图标的右上角。一个类可有零个至无穷个实例。例如，有着零个实例的类是纯抽象类，不能够有显式声明为纯抽象类类型的任何对象。实例的数目可能会存在上界或下界，这也可以在类的图中表示出来。图 9-6 显示了如何表示类的多重性(以及如何在关联的类之间进行表示)。

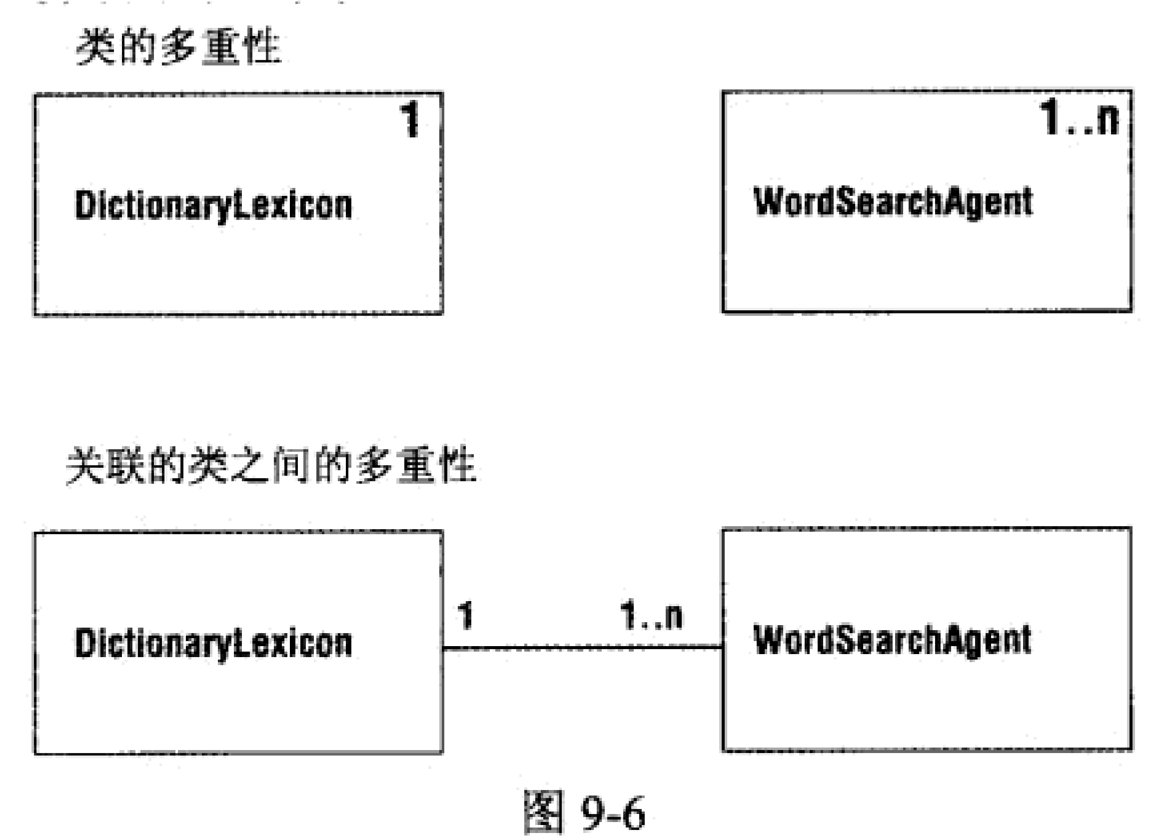


图 9-6

在图 9-6 中，WordSearchAgent 类的多重性为 1..n，意味着在系统中，WordSearchAgent 对象的数目最少是 1，最大是 *n*(取决于可用空间数量)，每个对象包含不同的单词集合。DictionaryLexicon 的多重性为 1。它是一个单例类(singleton class)，意味着在系统中只能存在一个实例。以下关于多重性的符号和含义的更多实例：

1	1 个实例
1..*n*	1 到指定的数字 *n*
1..*	1 到无穷大
0..1	0 到 1
0..*	0 到无穷大
*	无穷大

多重性可以显示在相关联的类之间。在图 9-6 中，存在 1 个 DictionaryLexicon 从 1 到多个 WordSearchAgent 实例的关联。这就意味着可以有多个 WordSearchAgent 在一个 DictionaryLexicon 上进行搜索。

9.2.5 模板类的可视化

模板类(template class)是一种允许将类型作为类的定义中的参数的机制。模板定义了处理传递给它的数据类型的服务。通过使用 template 关键字，可以在 C++中创建参数化的类：

```
template < class Type > classname {...};
```

参数 Type 代表了传递给模板的任意类型。Type 可以是内置数据类型(built-in datatype)或是用户定义的类。当声明了 Type 后，模板就会通过作为参数化类型传递给它的元素进行绑定。例如，Synonym 是包含了 string 对象的 vector 的 map 容器。map 和 vector 是模板类：

```
map < string,vector < string > > Synonym;
```

map 容器将 string 作为键，将包含字符串的 vector 作为值。vector 容器包含了字符串

对象。map 容器可以将任意数据类型映射到任何其他数据类型，vector 容器可以包含任何数据类型：

```
map < int, vector < string > > // maps a number to a vector of strings
map < int, string > >          // maps a number to a string
vector < DictionaryLexicon >   // a vector of dictionary objects
vector < map < int,string > >  // a vector of maps that maps a number to string
```

像所有其他类一样，模板类也使用矩形框来表示，并加上了参数化类型的标记。该标记通过在类图标的右上角的虚线框表示。模板类可以绑定到某种类型，也可以不绑定到某种类型。表示未绑定模板类的记号是用显示大写字母 T 的虚线框，T 表示未绑定的参数化类型。绑定的模板类可用隐式绑定(implicit binding)来表示，隐式绑定是 C++声明和绑定模板类的句法。例如，模板 vector 隐式地同字符串对象绑定：

```
vector <string>
```

这可以在类图标中显示，或者将<string>放置在虚线框中作为模板的类型。另外一种方法是使用依赖构造型(dependency stereotype)bind 和模板对象。构造型通过使用实际命名参数化类型指定对模板类进行实例化的源。这被称作显式绑定(explicit binding)。构造型指示符<<bind>>通过对参数化类型进行实例化对模板类实现精化(refinement)。精化是一个泛称，用来指示已存在事物的更高级别的详情。构造型指示符后面是由模板对象提供的实际命名参数。模板对象与模板类之间有一种依赖关系。模板对象也可看作是模板类的精化。图 9-7 描述了 map 容器的未绑定和绑定的模板类的表示方法。

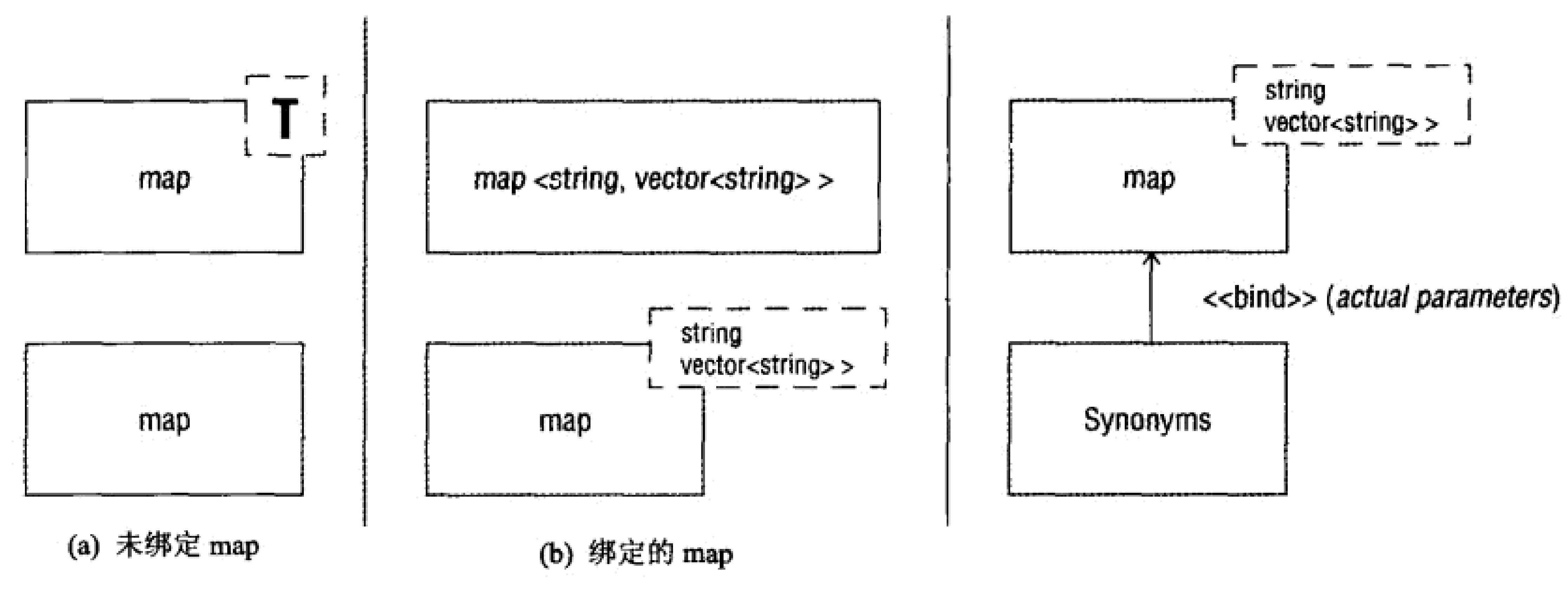


图 9-7

9.2.6 显示类与对象的关系

UML 为类之间的关系提供了 3 种分类。

- **依赖**：两个类间的依赖关系是指独立的类的变化会影响依赖的类。
- **泛化**：两个类间的泛化关系是指其中的一个类是另一个更为特殊的类的一般构造。一般构造是父类或超类，更为特殊的构造是子类。子类继承了父类的性质、属性

和操作，但也可以定义了自己的属性和操作。子类是从父类派生的，可以用作父类的替代。root 类或 base 类是没有父类的类。

- **关联**：关联是指定对象同其他对象连接的结构关系。对象间的关联可以是单向或双向的。当对象间有双向关联时，意味着对象 1 同对象 2 关联，同时对象 2 与对象 1 关联。当对象间是单向关联时，意味着对象 1 与对象 2 关联，但对象 2 不与对象 1 关联。两个元素(类等)间的关联被称为二元关联，而 *n* 个元素间的关联称为 *n* 元关联。

表 9-5 显示了可用于依赖关系的各种构造型。

表 9-5

依赖构造型 源---->目标	描　述
<<bind>>	源使用实际参数实例化模板目标
<<friend>>	目标对源而言是可见的
<<instanceof>>	源是目标的实例；这种依赖被用来定义类和对象间的关系
<<instantiate>>	源创建目标的实例；这种依赖被用来定义类和对象间的关系
<<refine>>	源比目标有着更高的详细程度；这种依赖被用来定义基类与派生类的关系
<<use>>	源依赖于目标的公共接口
<<become>>	目标与源是相同对象，但是处在源对象生命周期的后期；目标可能会处于与源不同的状态
<<call>>	源调用目标的方法
<<copy>>	源是目标的原样但独立的副本
<<access>>	源包被赋予了引用目标包的元素的权利
<<extend>>	目标用例扩展了源用例的行为
<<include>>	源用例能够在由源用例指定的位置包含目标用例的行为

存在多种类型的依赖、泛化与关联。每种关系类别有自己的标记。关系标记是元素间的实线或虚线，可能带有某些类型的箭头。为进一步定义关系，使用了构造型或修饰(adornment)。

这些构造型是进一步描述关系的本质的标签。构造型使用书名号包含的名字来体现，并被放置在元素上部或旁边。例如，在图 9-8 中，构造型简略地显示为：

```
<<bind>>
```

它被放置在指示模板对象到绑定模板类 list 的依赖的箭头旁边。<<bind>>依赖意味着 StrList 是为模板 string 提供参数的 list 类型。UnrecognizedWords 对 StrList 类型有<<instanceOf>>的依赖。WordSearchAgent <<use>> UnrecognizedWords 作为它无法识别的单词的容器。

修饰是添加到元素的基本表示中的文本项或图形项。它们用来记录元素的规范的详细

说明。例如：

- 使用一条连接相同或不同构造的实线来描述关联(association)。当一个构造在结构上与另一个构造相关时，应使用关联关系。
- **导航**(navigation)是关联的一种类型。为描述导航，元素间的线是一条带有指向关联中一个元素的箭头的虚线。
- **依赖**(dependency)由两个元素间的虚有向线(带箭头)来表示，从源指向它所依赖的元素。当一个构造使用另一个构造时，应使用依赖关系。
- **泛化关系**(generalization relationship)由一条指向父类或超类的有大的开放箭头的实有向线来表示。当一个构造从另一构造继承了行为或属性时使用泛化关系。作为子类的构造可能会改变或修改行为与属性。

表 9-6 与表 9-7 分别列出了可应用于泛化和关联的构造型、约束及性质。

表 9-6

泛　化	描　述
构造型	
<<implementation>>	子类继承父类的实现，但是不公开或支持父类的接口
约束	
{complete}	父类的所有子类均被指定，并且不能再派生更多的子类
{incomplete}	父类的所有子类未被指定，可派生其他子类
{disjoint}	父类的对象最多只能以它的一个子类作为其类型
{overlapping}	父类的对象可能以多个子类作为其类型

表 9-7

关　联	描　述
类型 object1 - - -> object2 导航	单向关联，其中 object1 与 object2 关联，但 object2 不与 object1 关联；如果没有箭头，则关联是双向的
part <—◇ whole 聚集	一种包含(整体-局部关系)，其中 part 在它的生命期中不仅仅同一个 whole 关联
part ◀—◆ whole 组合	一种包含(整体-局部关系)，其中 part 在它的生命期中仅同一个 whole 关联
约束	
{implicit}	关系是概念上的
{ordered}	在关联的末端的对象有顺序

(续表)

关　　联	描　　述
性质	
{changeable}	描述两个对象间可增加、删除和更改些什么
{addOnly}	描述可以增加到关联的另一端的对象的连接
{frozen}	描述一旦关联的另一端的对象增加了新的连接，则该连接不可被更改或删除

关联具有另一级别的细节，这些细节可应用于表 9-7 列出的一般关联或构造型。

- **名字**：关联可以有一个用于描述关系本质的名字。可在名字上添加定向三角形(directional triangle)以确保其含义。三角形指向待读取的名字的方向。
- **角色**：角色是关联的近端的类表现给关联的另一端的类的样子。
- **多重性**：多重性标记可用来说明有多少个对象可通过关联来连接。多重性可以显示在关联的两端。

图 9-8 也显示了类间关联关系的实例。DictionaryLexicon 和 WordSearchAgent 有着多重性为 1 到 1..n 的关联。这意味着只有 1 个 DictionaryLexicon 对应 1 到多个 WordSearchAgents。多个 WordSearchAgents 可对 DictionaryLexicon 进行搜索。LexicalEntry 是 DictionaryLexicon 与 Word 间的关联的名字。在这种情况下，LexicalEntry 也是一个具有属性的类。

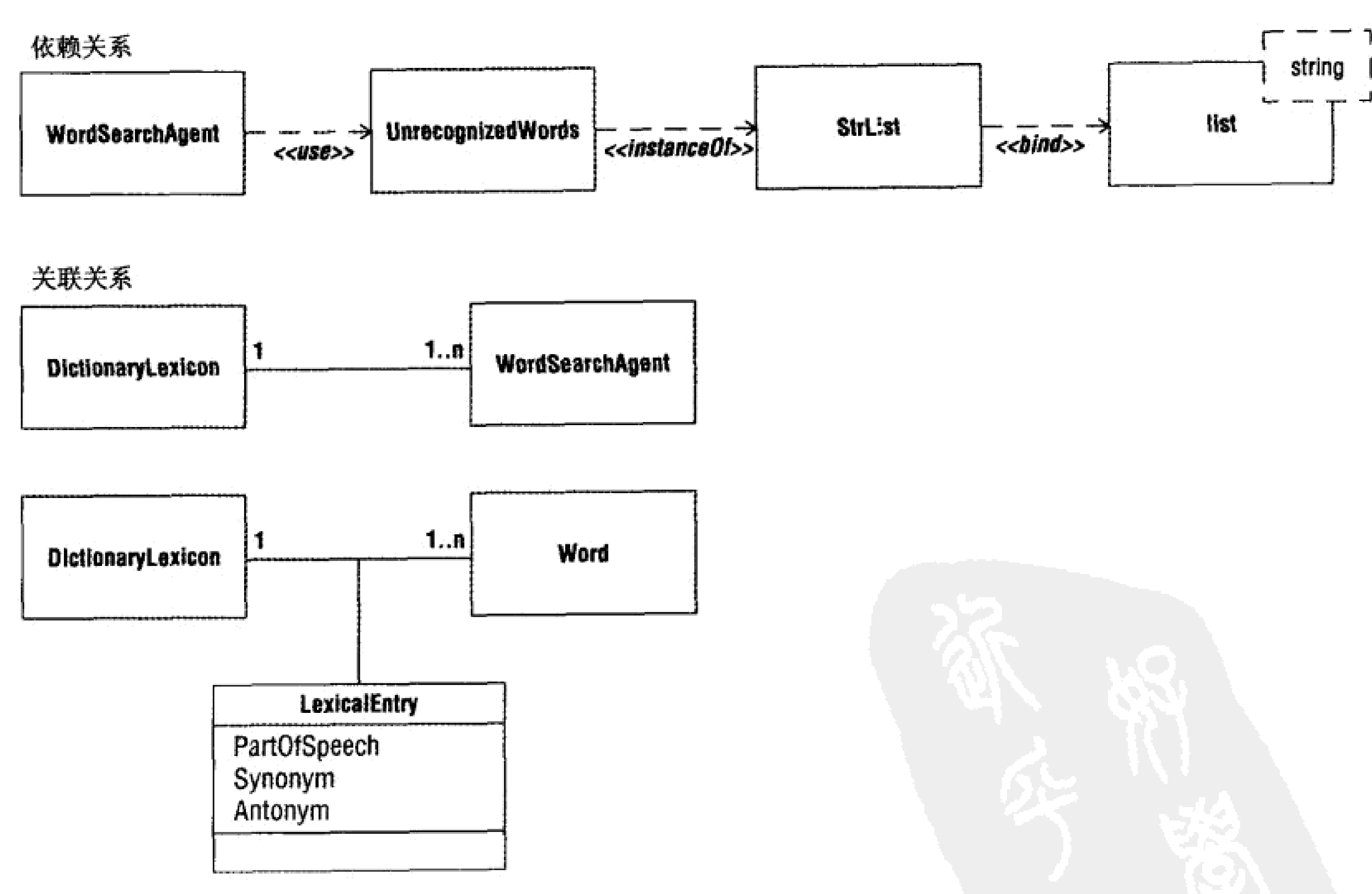


图 9-8

9.2.7　接口类的可视化

类的接口定义了外部世界通过暴露出来的方法和属性同对象的交互。类在设计、实现和使用之后，也许有必要改变类的接口以适应用户。在现实中，改变当前类的接口是不实际的，因为它已经在使用中，而且这样做会破坏已有代码。接口类用来修改另一个已有的类或类的集合的接口。这种修改使得类对一组用户来说更易用、功能性更强、更安全或语义更准确。接口类的例子是 adaptors 容器，它是标准模板库的一部分。adaptors 为 deque、vector 和 list 容器提供了一个新的公有接口。示例 9-1 显示了 stack 类。它用作修改 vector 类的接口类。

示例 9-1

```
//Example 9-1  Using the stack class as an interface class

template < class Container >
class stack{
//...
public:
   typedef Container::value_type value_type;
   typedef Container::size_type size_type;
protected:
   Container c;
public:
   bool empty(void) const {return c.empty();}
   size_type size(void) const {return c.size(); }
   value_type& top(void) {return c.back(); }
   const value_type& top const {return c.back(); }
   void push(const value_type& x) {c.push.back(x); }
   void pop(void) {c.pop.back(); }
};
```

通过指定 Container 的类型声明了 stack。

```
stack<vector<T>>Stack;
```

在这种情形下，Container 是个 vector，但是任何容器，如 deque 和 list，只要定义了如下这些操作：

```
empty()
size()
back()
push.back()
pop.back()
```

就可被用作 stack 接口类的实现类：

```
stack < list < T > > Stack;
stack < deque < T > > Stack;
```

stack 类提供了传统上被栈接受的语义正确的接口：

```
push()
pop()
top()
```

可以使用多种标记来表示一个接口类，各自显示各种级别的细节。图 9-9 显示了接口类的多种标记。

- 在图 9-9(a)中，将 stack 作为接口类来显示。
- 在图 9-9(b)中，在类名上方的类符号中显示了构造型指示符<<interface>>，指出这是一个接口类。这个例子显示了类的属性和方法。字母 I 可放置于接口类的名字以及所有操作之前，以进一步与其他类区分。
- 图 9-9(c)显示了模板类的实现(realization)。可读作“stack 类通过 vector 类实现”。实现用来显示 stack 类与以它作为接口的类之间的关系。实现是类之间的语义关系，在这种关系中，一个类指定契约(接口类)，而另一个类完成它(实现类)。在这里，stack 类指定契约，vector 类完成契约。实现关系用两个类之间的虚线表示，虚线带有一个大的开放箭头指向接口类或指定契约的类。
- 图 9-9(d)显示了接口类与它的实现者之间的关系，使用接口棒棒糖表示法(lollipop notation)来表示。

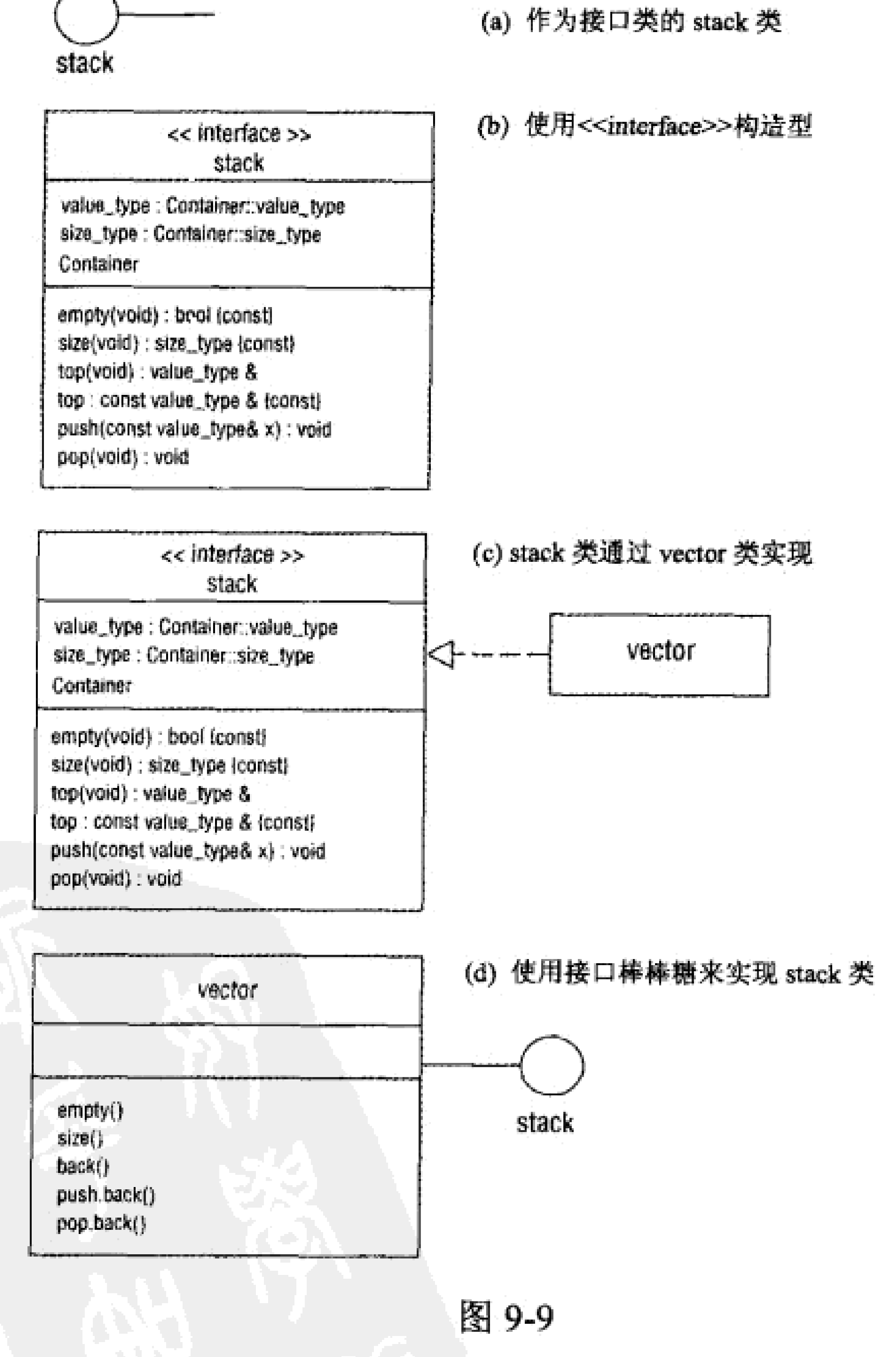


图 9-9

9.2.8　交互式对象的组织

如同您所看到的，类和接口可用作构建块来创建更复杂的类和接口。在并行系统中，可能有许多大而复杂的结构与其他结构合作，这样就创建了类和接口的群，它们共同完成系统的目标。这些元素的集合以及它们之间的交互构成了协作(collaboration)。这些构建块可以包含系统中的结构元素和行为元素。用户提出的执行特定任务的请求可能会涉及很多对象一同工作。这些对象与其他元素一同工作可以完成其他任务。理解了这些，便可清晰看到定义良好的可重用类的益处，即它们可在同一系统中以不同的方式使用，或是可用在完全不同的系统中。协作有两个部分：

- **结构部分**，重点关注协作元素是以何种方式组织和构建的
- **行为部分**，重点关注元素间的交互

图 9-10 显示了基于 agent 的词典系统的结构部分的示例。在该协作中，使用的所有类都有某种类型的依赖关系。例如，MorphologyAgent 和 MisspelledAgent 调用 UnrecognizedWords 中的方法，Lexicon 绑定模板 map，UnrecognizedWords 包含着一个单词列表，该单词列表绑定 list 模板。WordSearchAgent 与 UnrecognizedWords 之间有着 1..n 的关系。协作的结构部分由很多类和接口的任意组合、组件和节点组成。如图 9-10 所示，系统可以包含很多协作。系统中的某个协作是唯一的，但是协作中的元素不是唯一的。一个协作中的元素可以被用在另一个协作中，使用不同的组织并执行不同的功能。在这个特定的协作中，WordSearchAgent 用来查找列表中无法识别的单词。假如将要使用一个不同的 Lexicon，它也可以被用来查找一个特定领域中的单词。

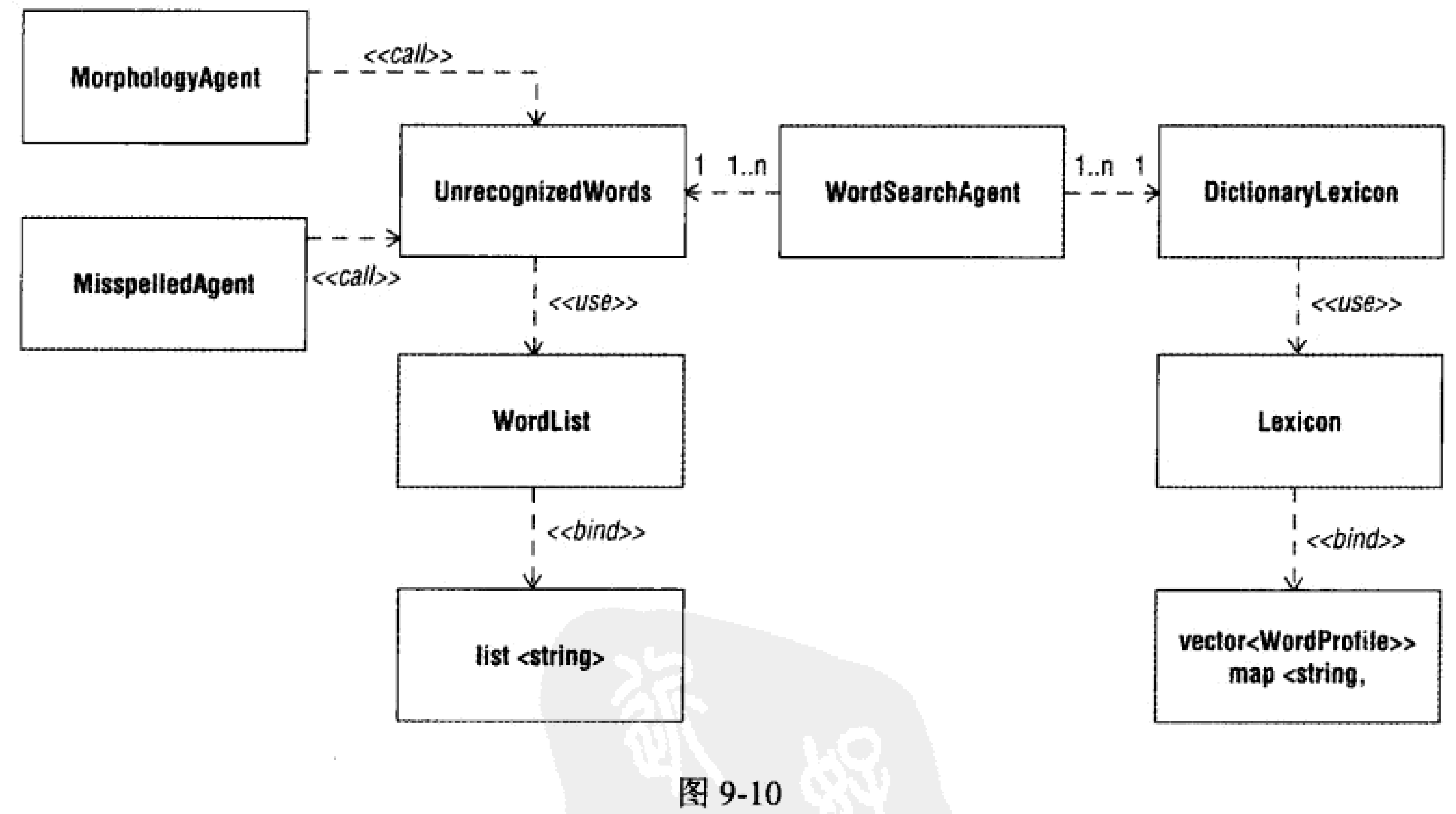


图 9-10

注意：

协作中的行为部分的描述是通过使用交互图来完成的，如顺序图或活动图。这些将在下节讨论。

9.3 UML 与并发行为

系统的行为视图主要关注系统的动态方面。这个视图检查了系统中的元素同系统的其他元素交互时会有怎样的行为。元素之间交互时会发生并发。本节讨论的制图技术是用来对以下进行建模：

- 对象行为的生命期
- 为特定目的一同工作的众多对象的行为
- 集中于动作或动作序列的控制流
- 元素间的同步与通信

9.3.1 协作对象

协作对象(collaborating object)是为了执行特定任务而相互关联的对象，它们并不构成持久的关系。相同的对象可与其他对象关联起来执行其他任务。协作对象可在协作图(collaboration diagram)中表示。协作图包含结构部分和交互部分。我们已经讨论过结构部分，交互部分是由所有参与的对象作为顶点的图。对象间的连接是弧。弧可带有对象间传递的信息、方法调用与构造型指示符等，用来表达关于连接的更多细节。

两个对象间的连接是一个链接(link)。链接是协作的一种类型。当两个对象链接在一起，它们之间可以执行一些动作。动作会导致对象状态的改变，这种改变可以是单方的或是双方的。表 9-8 显示了可能发生的动作的类型以及实例。

表 9-8

create	创建一个对象
destroy	销毁一个对象
call	对象的操作可被对象自身或其他对象调用
return	将一个值返回给对象
send	向对象发送一个信号

当调用任何方法时，可表示出参数与返回值。如果指定，可发生其他动作。

如果接收对象对于调用对象是可见的，则可发生如下动作。可以使用构造型来指定对象为什么是可见的。如表 9-9 所示。

表 9-9

association	由于存在关联(很常见)，因此对象是可见的
parameter	由于对象是传递给调用对象的参数，因此是可见的
local	由于对象对调用对象有着局部作用域，因此是可见的
global	由于对象对调用对象有着全局作用域，因此是可见的
self	对象调用其自身的方法

图中也可以表示其他适合于关联的构造型和修饰。

当方法被调用时，可能会引起其他方法被其他对象所调用。操作执行的顺序可用放在方法之前的序号组合(sequence number combination)与冒号分隔符来显示。序号组合表示了方法是按照什么顺序关联起来的以及操作发生的时间序号。例如，图 9-11 显示了使用序号的协作图。

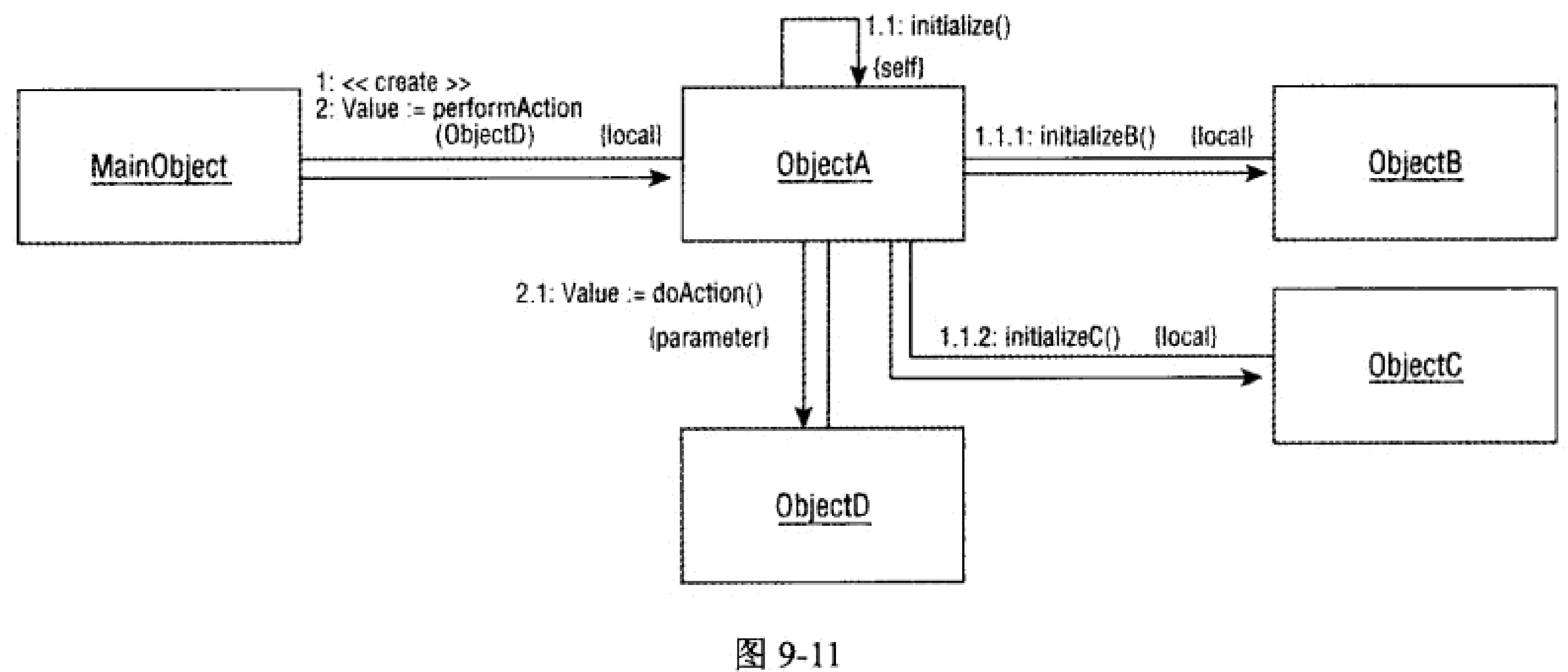


图 9-11

在图 9-11 中，MainObject 顺序地执行两个操作：

```
1: < < create > >
2: Value := performAction(ObjectF)
```

在操作 1 中，MainObject 创建 ObjectA。通过包含(containment)，ObjectA 对于 MainObject 是 local(本地)的。这发起了嵌套控制流中的第一个操作序列。所有属于这个序列的操作使用数字 1 后面跟操作发生的时间序号。序列 1 的第一个操作是：

```
1.1: initialize()
```

ObjectA 调用它自己的操作。这是通过将对象连接到自身并使用{self}构造型指示符来表达的。操作 ObjectA::intitialize( )还引起另一个动作序列的开始：

```
1.1.1: initializeB()
1.1.2: initializeC()
```

其中，调用了对于 ObjectA 是 local 的另外两个对象的初始化方法。操作：

```
2: performAction(ObjectD)
```

是另一个嵌套的序列的开始。ObjectA 调用 ObjectD 的操作：

```
2.1: doAction()
```

ObjectA 之所以可以调用这个操作，是因为 ObjectD 是一个参数(由 MainObject 传递)，就像构造型{parameter}所指示的那样。一个值返回到 ObjectA，而且一个值返回到

MainObject。除了序号组合外，这些嵌套的控制流还通过使用指向序列流的方向的实心箭头得到进一步加强。

9.3.2 使用进程与线程的多任务与多线程

应用程序内部的并发可以通过使用多任务与多线程来实现。多任务允许同时运行多个进程，而多线程则允许单个进程通过使用多个线程来同时执行多个任务。当一个进程被划分成多个任务，而且每个任务通过一个线程来执行时，则称该进程是多线程的。线程是在进程的地址空间内执行的控制流。每个进程至少有一个线程，即主线程。

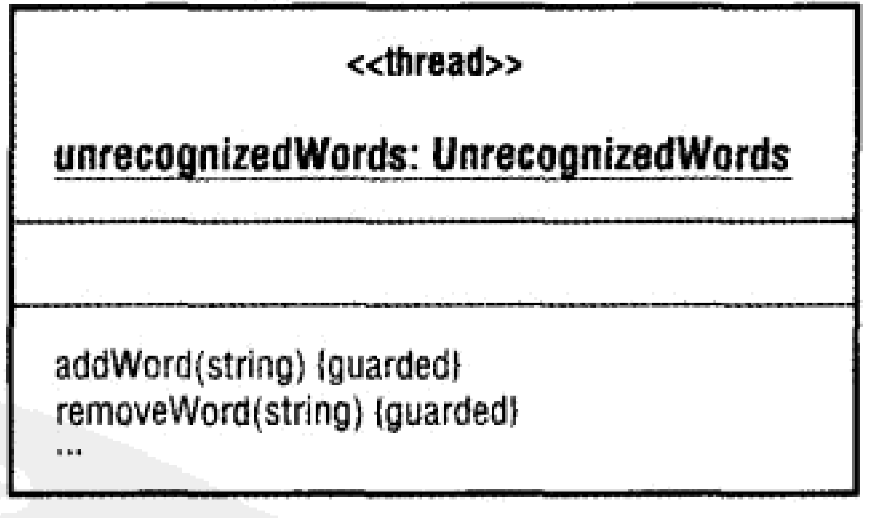
1. 对活动对象进行制图

当使用 UML 时，每个独立的控制流被看作一个活动对象(active object)。活动对象就是拥有一个进程或线程的对象。每个活动对象可以发起控制活动。活动类(active class)，是其对象是活动的类。活动类可被用于对有着相同数据成员和方法的进程组或线程组进行建模。系统中的对象可能与活动对象没有一对一的相关性。当根据对象把程序划分为进程或线程时，对象的方法可以在一个单独的进程或是多个单独的线程上执行。因此，在对这样的对象进行建模时，可以通过多个活动对象来表示。静态对象与动态对象之间的这种关系可以通过使用交互图来表示。线程与进程可直接通过活动对象来表示。

除了用更粗些的线来描绘矩形的边界外，UML 表示活动对象或活动类的方式与表示静态对象的方式相同。还可使用两种构造型：

- process
- thread

可以通过显示这些构造型指示符来表示两种活动对象之间的区别。图 9-12 显示了两个活动对象，均为线程。每个线程执行 unrecognizedWords 与 wordSearchAgent2 的方法。



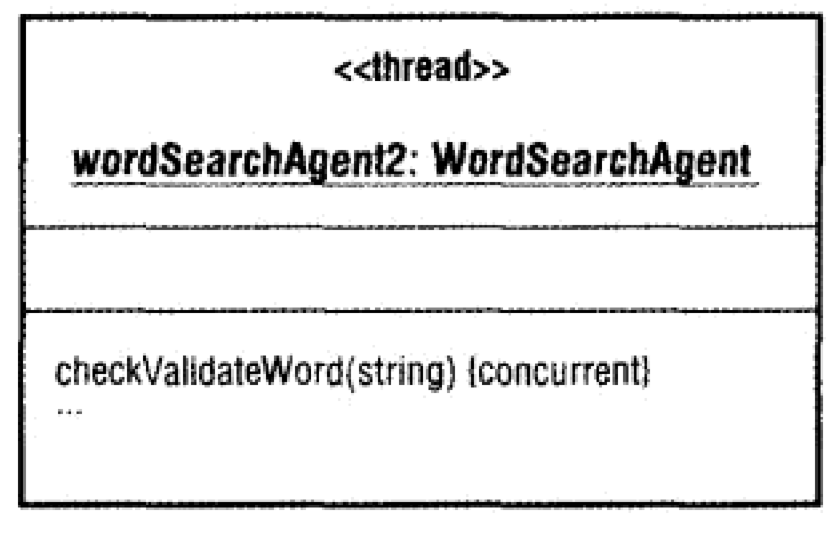


图 9-12

2. 显示多个控制流与通信

在并发系统中，有着多个控制流。每个控制流基于一个控制着活动的进程或线程。这些进程和线程可以在有着多个处理器的单机系统上执行。活动对象或活动类用来表示每个控制流。当创建了活动对象后，就发起了一个独立的控制流。当活动对象被销毁时，控制流终止。在系统上为多控制流建模有助于您对这些控制流进行管理、同步与通信。

在协作图中，序号与实线箭头被用来确定控制流。在由并发系统中的活动对象组成的协作图中，活动对象的名字会在它执行的操作的序号之前出现。活动对象可以调用其他对象中的方法，也可以挂起执行，直到函数返回或能够继续执行。箭头不仅用来显示控制流的方向，也用来表示它的性质。实箭头用来表示同步调用，半尖箭头(half-stick arrowhead)用来表示异步调用。由于多个活动对象可以调用一个对象中的操作，方法的性质：

- sequential
- guarded
- concurrent

可被用来描述方法的同步性质。

图 9-13 显示了多个活动对象的协作。在该图中，这些对象共同产生未知单词的列表。mainAgent 用来记录并协调预备工作及由活动对象问题解决者生成的未知单词的结果列表。

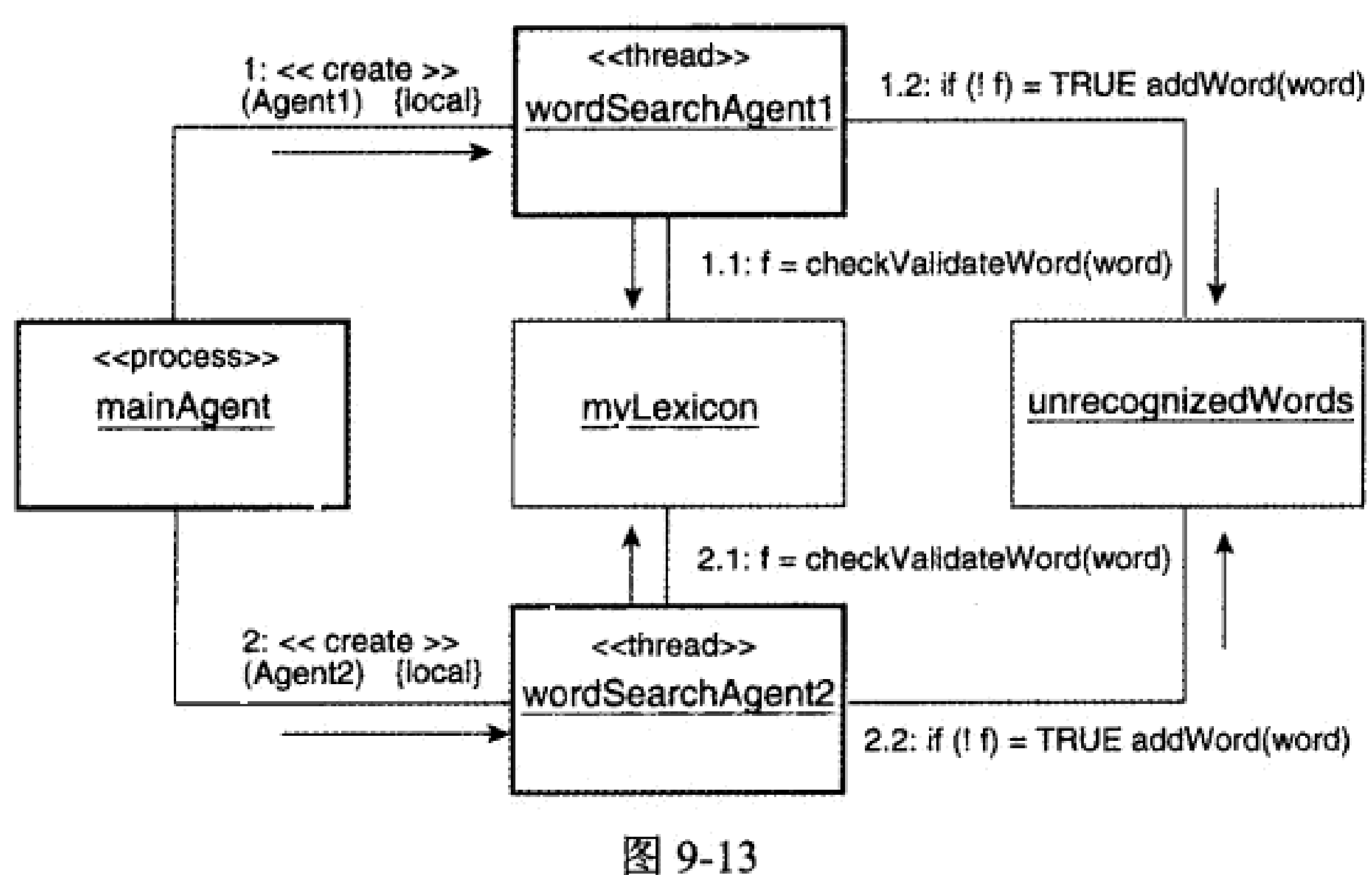


图 9-13

myLexicon 与 unrecognizedWords 对象被两个 agent 并发访问。这两个对象对于本协作中的所有 agent 都是可见的。wordSearchAgents 调用 unrecognizedWords 对象的方法：

```
wordSearchAgent1:unrecognizedWords.addWord(string word)
wordSearchAgent2:unrecognizedWords.addWord(string word)
```

wordSearchAgents 还调用 myLexicon 对象的方法：

```
wordSearchAgent1:myLexicon.validateWord(string & X)
wordSearchAgent2:myLexicon.validateWord(string & X)
```

WordSearchAgents 并发地调用 unrecognizedWords 对象的操作。unrecognizedWords addWord( )具有 guarded 性质，因此同时调用该方法是安全的。myLexicon.validateWord( )不修改对象，同时调用该方法也是安全的。

9.3.3　对象间的消息序列

同协作图集中关注共同工作以执行任务或操作或实现用例的对象间的结构组织和交

互不同，顺序图集中关注方法调用或特定任务、操作或用例中涉及的过程的时间排序。在顺序图中，涉及的每个对象或构造的名字显示在它自己的矩形框中。矩形框放置在沿着图的 x 轴的顶端。您应当仅包含涉及的主要参与者(player)以及最重要的函数调用，因为图会迅速变得过于复杂。对象从左到右排序，从发起动作的对象或过程开始，到最次要的对象或过程。调用沿 y 轴从上到下按时间顺序排列。每个框的底部放置一条垂直线，表示对象的生命线。绘制的实线箭头线从一个对象的生命线到另一个对象的生命线，表示从调用者到接收者的函数调用或方法调用。从接收者回到调用者绘制了尖箭头线，表示从函数或方法返回。每个函数调用最少会标记上函数或方法名。也可以显示参数与控制信息，比如方法在什么条件下会被调用。例如：

```
if(!Unrecognized)
checkTransposes()
```

除非条件为真，否则函数与方法将不会执行。对于需要在一个对象上调用数次的方法，例如从结构体中读取值，会在前面加上一个迭代标记(*)。

图 9-14 显示了 Lexicon 系统中涉及的某些对象的顺序图。为了避免显示复杂的图，我们只显示了部分对象。当您为并发对象或过程使用顺序图时，会使用激活符号(activation symbol)。激活符号是出现在对象的生命线上的矩形，这表示对象或过程是活动的。当对象对另一对象或过程发起调用并且不阻塞时，将使用到它们。它显示对象或过程在继续执行或是活动的。如果一个对象不是活动的，那么会用一条虚线。在图 9-14 中，对象 misspelledAgent 一直是活动的，而 mainAgent 在创建了 misspelledAgent 后变成不活动。一旦创建了 unrecognizedWords 和 myLexicon，它们便不是活动的，直到 misspelledAgent 调用 nextWord( )和 removeWord( )。

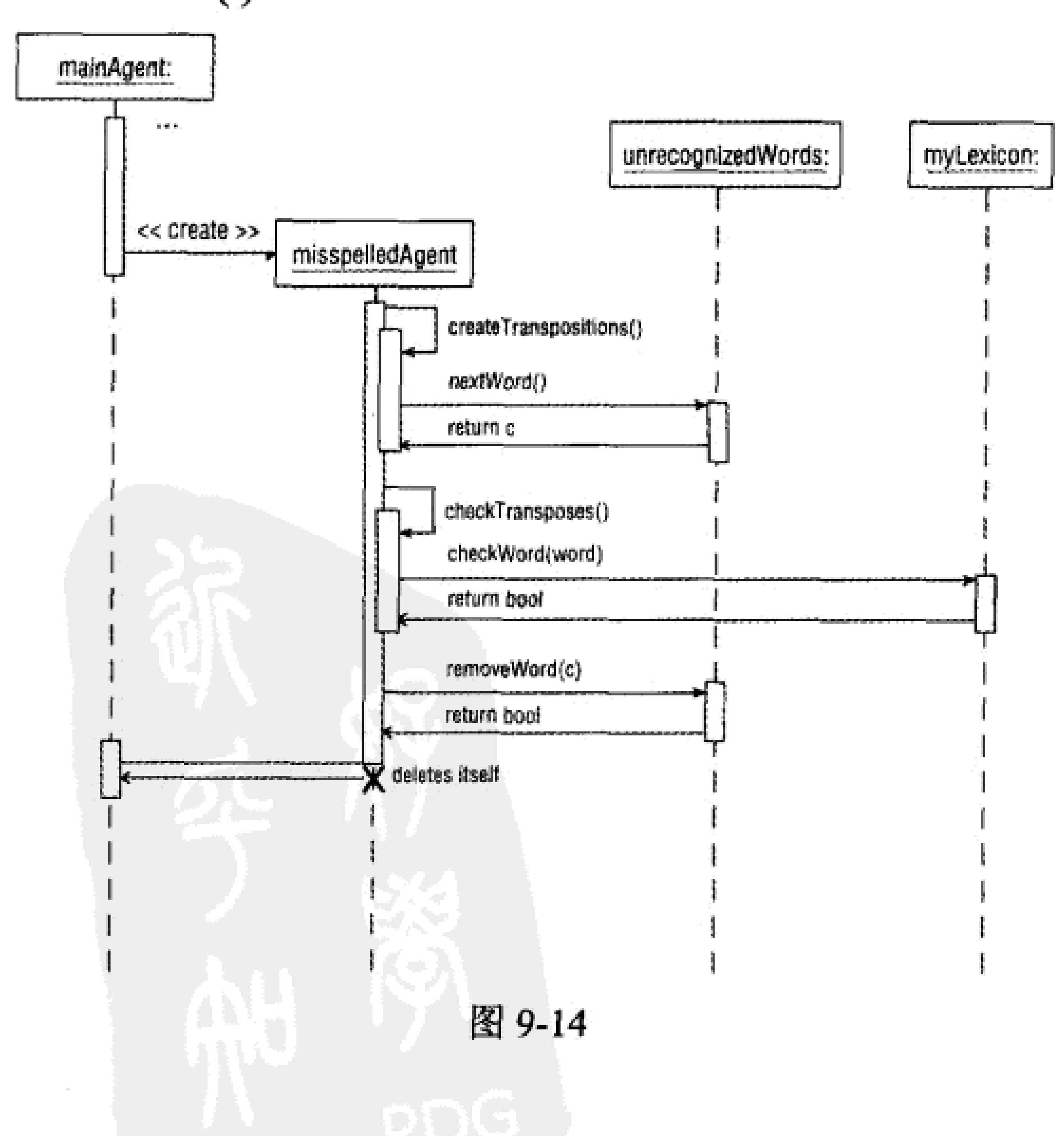


图 9-14

为表示一个对象已经调用它自己的一个方法，应使用自委托(self-delegation)符号。它是激活符号与调用箭头的组合。激活符号与已有的激活符号交迭。带箭头的线从原来的激活符号指向附加的激活符号。在图 9-14 中，当 misspelledAgent 调用它的 createTranspositions( )与 checkTransposes( )方法时，发生自委托。checkTransposes 方法将初始单词的变换传递给 myLexicon，myLexicon 检查这个单词是否在可被识别的单词列表中。这个方法会被迭代调用，直到 Unrecognized 为 FALSE，这意味着变换后的单词被确定为初始单词的正确拼写。如果变换后的单词已经被识别，则初始单词就被删除。

9.3.4　对象的活动

UML 可被用来对特定操作或用例中涉及的对象所执行的活动进行建模。这被称作活动图(activity diagram)。活动图是一个流程图，一步步地显示特定任务中顺序或并发的动作或活动。箭头描绘了图中表示的活动的控制流。协作图强调对象到对象的控制流，顺序图强调按时间顺序的控制流，活动图强调一个动作或活动到另一动作或活动的控制流。动作或活动改变对象的状态或返回一个值。包容动作或活动的是动作状态(action state)或活动状态(activity state)。它们表现了控制流中特定时刻的对象状态。

动作与活动是不同的。

- **动作**在逻辑上不能被分解或被其他动作或事件或中断。动作的实例是创建或销毁对象、调用对象的方法或在过程中调用函数。
- **活动**可被分解成其他活动甚至另一个活动图。活动的例子是程序、用例或过程。活动可以被事件或其他活动或动作所中断。

活动图中的节点是动作或活动，弧是无触发转换(triggerless transition)。无触发转换不需要事件去引起转换发生。当前面的动作或活动完成，转换就会发生。图包含了决策分支、开始、停止以及对多个动作或活动进行 join 或 fork 的同步条(synchronization bar)。动作状态与活动状态以同样的方式表示。为表示动作状态或活动状态，UML 使用用于显示流程图的入口点与出口点的标准流程图符号。这个符号的使用不受发生的动作或活动的类型影响。然而，我们更喜欢使用将输入/输出动作(平行四边形)同处理或转换动作(矩形)区分开的标准流程图符号。作为函数调用、表达式、短语、用例或程序名的动作或活动的描述将显示在使用的动作符号中。除此之外，活动状态还可以显示 entry 动作和/或 exit 动作。entry 动作是发生在进入活动状态时的动作，exit 动作是发生在刚要退出活动状态时的动作。它们分别是在活动状态中执行的第一个和最后一个动作。

一旦动作完成，就会发生转换，下一动作立即发生。转换用从一个状态到下一个状态的带有尖箭头的有向线来表示。指向状态的转换是 inbound，而离开状态的转换是 outbound。在 outbound 转换发生前，如果存在 exit 动作，则执行该动作。在 inbound 转换之后，如果该状态存在 entry 动作，则执行该动作。控制流的开始表示为一个大的实心圆点。

第一个转换从实心圆点出发到图中的第一个状态。活动图中的停止点或停止状态用一个圆内的大的实心圆点来表示。

活动图与流程图类似，具有决策符号。决策符号是具有一个 inbound 转换和两个或更多 outbound 转换的菱形。outbound 转换是守卫条件(guarded condition)，它决定了控制流的路径。守卫条件是简单的布尔表达式。所有的 outbound 转换应覆盖分支的所有可能路径。图 9-15 显示了用于决定是否应当建立一个知识源的决策符号，用于决定是否应当将某个单词加到无法识别的单词列表中。

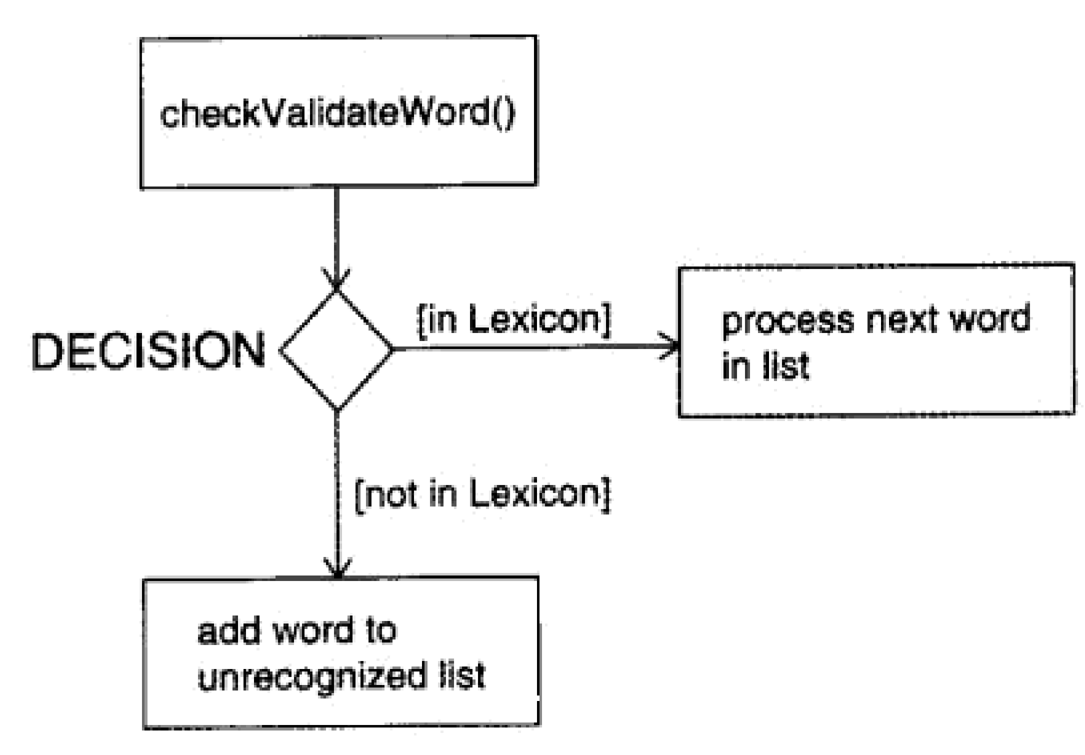


图 9-15

您可能发现一个动作或活动完成后，会存在多个动作或活动的顺序流并发发生。与流程图不同，UML 定义了可用来表示多个控制流并发发生的时刻的符号。同步条被用来显示某路径分叉或分支(fork)成并行路径和并行路径结合(join)的地方。它是一条宽的水平线，可有多个 outbound 转换(forking)或是多个 inbound 转换(joining)。每个转换代表了一条不同的路径。从同步条出去的 outbound 转换表示一个动作或活动的状态导致多个控制流的发生。到达同步条的 inbound 转换表示多条控制流需要被同步。同步条被用来显示该路径等待所有的路径汇集并结合成一条流或路径。

在图 9-16 中，mainAgent 通过创建两个 wordSearchAgents 创建了两条并发控制流。当这些 agent 结束后，它们重新结合成一条控制流，然后 mainAgent 创建 MorphologyAgent。

图被划分成几个被称为泳道(swimlane)的隔离的部分。在每个泳道中，会发生特定对象、组件或用例的动作或活动。泳线(swinline)是将图分成多个部分的垂直线。特定对象、组件或用例的泳道指定了活动的焦点。动作或活动只能在一条泳道中发生。转换与同步条可跨跃一条或多条泳道。同一泳道中或不同泳道相同级别的动作与活动是并发的。图 9-16 显示了带有泳道的活动图。

这个活动图是用来为 mainAgent 涉及的动作序列以及为基于 agent 的词典系统生成无法识别单词列表所涉及的其他对象进行建模。并发在 WordSearchAgents 中发生。同步条位于 WordSearchAgents 的泳道中。当处理完所有单词之后，这些 agent 控制流就会被放弃，然后控制流返回到主线程，即 mainAgent。

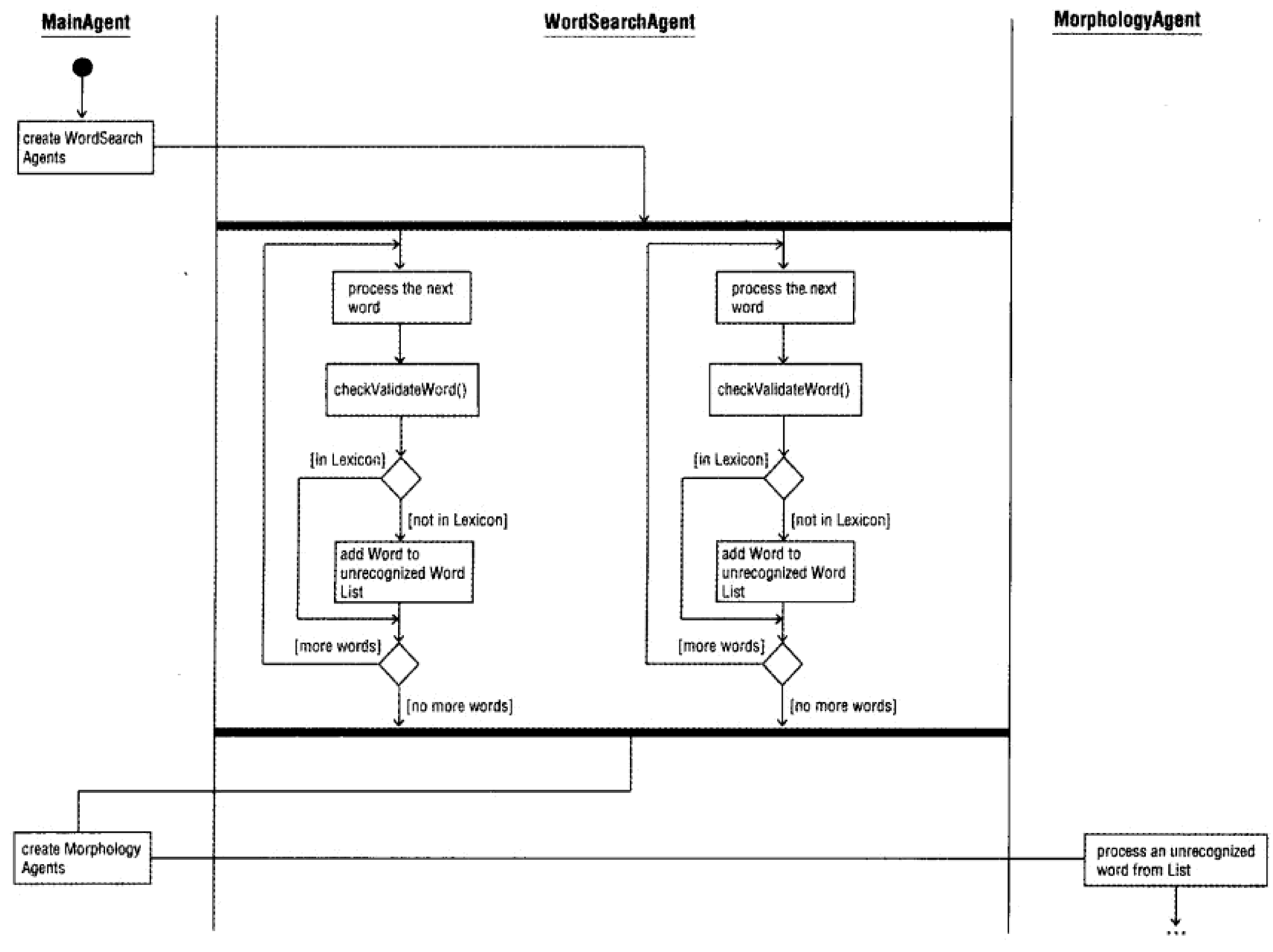


图 9-16

9.3.5　状态机

状态机描述了单个构造的行为，指定了它在生命期中响应内部和外部事件时的转换顺序。这个单个构造可以是系统、用例、或是对象。状态机用来对单个实体的行为建模。实体可以对诸如过程、函数、操作、信号等事件做出响应。实体也可以对时间的流逝做出响应。无论事件何时发生，实体通过执行某些活动或采取一些动作，导致实体状态改变或产生一些制品(artifact)进行响应。执行的动作或活动依赖于实体的当前状态。状态就是在实体的生命期中执行一些动作或对一些事件进行响应之后，实体所处的情形。

状态机可以用表或被称为状态图(state diagram)的有向图来表示。图 9-17 显示了进程状态机的 UML 状态图。

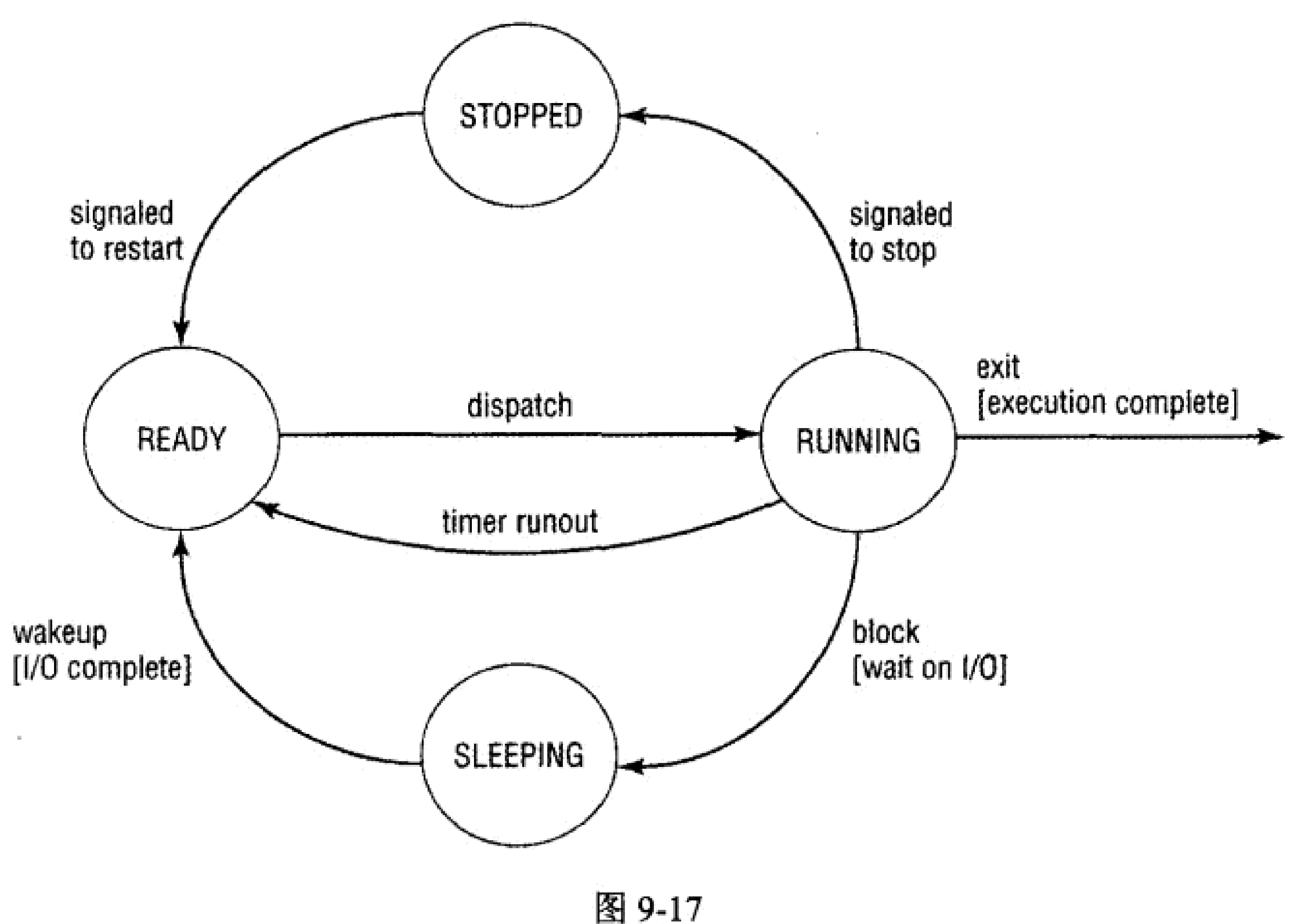


图 9-17

图 9-17 显示了进程在系统中活跃期间所经历的状态。进程可以有 4 种状态：ready、running、sleeping 和 stopped。有 8 种事件可导致进程的 4 种状态。其中 3 种事件只有当条件满足时才会发生。

- 只有在进程请求I/O或是等待某个事件发生时，才会发生block事件。如果发生block事件，则会触发进程从 running 状态到 sleeping 状态的转换。
- 只有当事件发生或 I/O 操作完成时，才会发生 wakeup 事件。如果发生 wakeup 事件，则会触发进程从 sleeping 状态(源状态)到 read 状态(目标状态)的转换。
- 只有当进程执行完所有指令后，才会发生 exit 事件。如果发生 exit 事件，则会触发进程从 running 状态到 sleeping 状态的转换。

其他的事件是外部事件，不受进程的控制。它们因某些外界因素发生，会触发进程从源状态到目标状态的转换。

状态图用来对对象、用例或系统的动态方面进行建模。顺序图、活动图、交互协作图和状态图被用来对活动的系统或对象的行为进行建模。结构上的协作图与类图用来对对象或系统的结构组织进行建模。状态图可以很好地描述对象的行为，不需考虑用例。它们不应被用来描述多个交互或协作的对象的行为。它们应当被用来描述经历很多转换，而且多个事件可导致一个转换发生的对象、系统或用例。这些是对内部和外部事件可以很好地做出反应的构造。

在状态图中，节点是状态，弧是转换。状态通过圆角矩形来表示，其中显示的是状态名。转换就是连接源状态和目标状态的线，该线带有一个尖箭头，指向目标状态。还存在 initial 状态与 final 状态。

- **initial 状态**是状态机默认的起点。用实心黑色圆点来表示该状态，从 initial 状态到状态机的第一个状态存在转换。

- **final 状态**是状态机的结尾状态，指示状态机已完成，或系统、用例、对象已到达生命线的结尾。使用嵌入到圆的实心圆点来表示。

1. 表现状态的各部分

状态有多个部分。表 9-10 列出了状态的各个部分。

表 9-10

状态的各个部分	描　　述
Name	用于同其他状态进行区分的唯一的状态名；状态也可能没有名字
Entry/exit actions	当进入到状态(entry state)或从状态退出(exit state)时执行的动作
Substates	嵌套的状态；子状态是可以被顺序或并发激活的不相交的状态；组合或超状态是包含子状态的状态
Internal transitions	状态内部的转换，其处理不会引起状态改变
Self-transitions	状态内部的转换，其处理不会引起状态改变，但可引发 exit 然后是 entry 动作的执行
Deferred events	当对象处于该状态时会产生的事件列表，但是会对这些事件排队，当对象处于另一状态时，处理这些事件

状态可以简单地通过在状态符号中央显示状态名来表示。如果要在状态符号内显示其他动作，状态名需要在顶部的单独区划中出现。在这个区划下方列出动作和活动，用这种格式显示：

```
label [Guard] / action or activity
```

例如：

```
do / validate(data)
```

do 是当对象处于该状态时要执行的活动的标签。用 data 作为参数来调用 validate(data) 函数。如果动作或活动是对函数或方法的调用，那么可以显示参数。

Guard 是个可被求值为 true 或 false 的表达式。如果条件的求值结果为 true，那么动作或活动会发生。例如：

```
exit [data valid] / send(data)
```

退出动作 send(data)是受控的(guarded)。表达式 data valid 被求值为 true 或 false。在退出该状态时，如果表达式为 true，那么调用函数 send(data)。Guard 总是可选的。

当事件发生时，会发生转换。这将引起对象、系统或用例从一种状态转变为另一种状态。两个转换的发生会不引起对象、系统或用例的状态改变。它们是：

- **Self-transition**：使用自转换时，当特定事件发生时，会触发对象离开当前状态。当退出时，执行 exit 动作(如果存在)，然后执行任何与自转换相关联的动作(如果存在)。对象重入状态，然后执行 entry 动作(如果有)。
- **Internal transition**：使用内部转换时，对象不离开状态，因此不执行 entry 或 exit 动作。

图 9-18 显示了有着 exit 动作和 entry 动作、进行活动并带有内部转换和自转换的状态的一般结构。自转换用一条指回自身的有向线来表示。

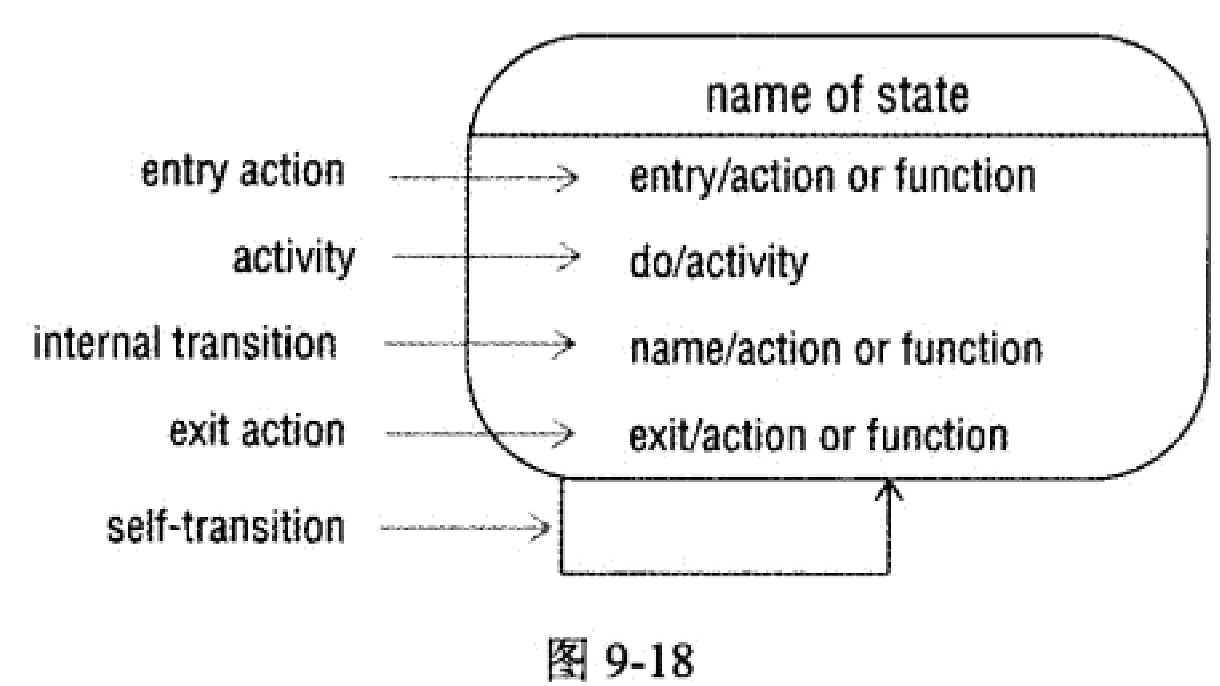


图 9-18

不同状态之间的转换表明它们之间存在关系或路径。在某个状态下，可能会发生某个事件或满足了某个条件，导致对象从一种状态(源状态)转换到另一种状态(目标状态)。事件触发了对象的转换。一个转换可能有多个同时存在的源状态。如果是这样，在转换发生之前，它们会 join(结合)起来。一个转换可能有多个同时存在的目标状态，其中会发生 fork(分支)。表 9-11 列出了转换的各个部分。转换通过从源状态到目标状态的有向线来表示。紧邻转换显示的是事件触发器的名字。和动作及活动类似，导致转换的事件也可以是受控的。转换可以是无触发(tiggerless)的，意味着没有哪些特殊事件的发生会导致转换的发生。退出源状态后，对象立即进行转换并进入到目标状态中。

表 9-11

转换的各个部分	描　述
目标状态	转换发生后对象进入的状态
源状态	对象的原始状态；当转换发生时，对象离开源状态
事件触发器	导致转换发生的事件；转换可能无触发，一旦对象结束源状态内的所有活动时，转换立即发生
守卫条件	与事件触发器相关联的布尔表达式；当表达式的值为真时，发生转换
动作	在转换期间，由对象执行的动作；可以与事件触发器和/或守卫条件关联起来

2. 对并发子状态进行制图

子状态可用来进一步简化对并发系统行为进行建模的描述。子状态是包含在另一个状

态(被称作超状态或复合状态)中的状态。这种表示法意味着状态可进一步被分解成一个或多个子状态。这些子状态可以是顺序的或是并发的。有了并发子状态，每个表现出来的状态机以不同的但并发存在的控制流来表现并行。每个子状态被一条虚线分隔开。对于本章中使用的实例中的 WordSearchAgent 对象，情况便是如此。每个对象处理本地列表中的所有单词。这些对象的状态位于名为 Building the Unrecognized Word List 的超状态中。

每个子状态都被包含在一个独立的区划中。在退出组合状态前，子状态被同步并结合。当一个子状态达到其最终状态时，它等待另一状态达到最终状态，然后这些子状态重新被结合成一个流。图 9-19 显示了基于 agent 的 Lexicon 系统的状态图。

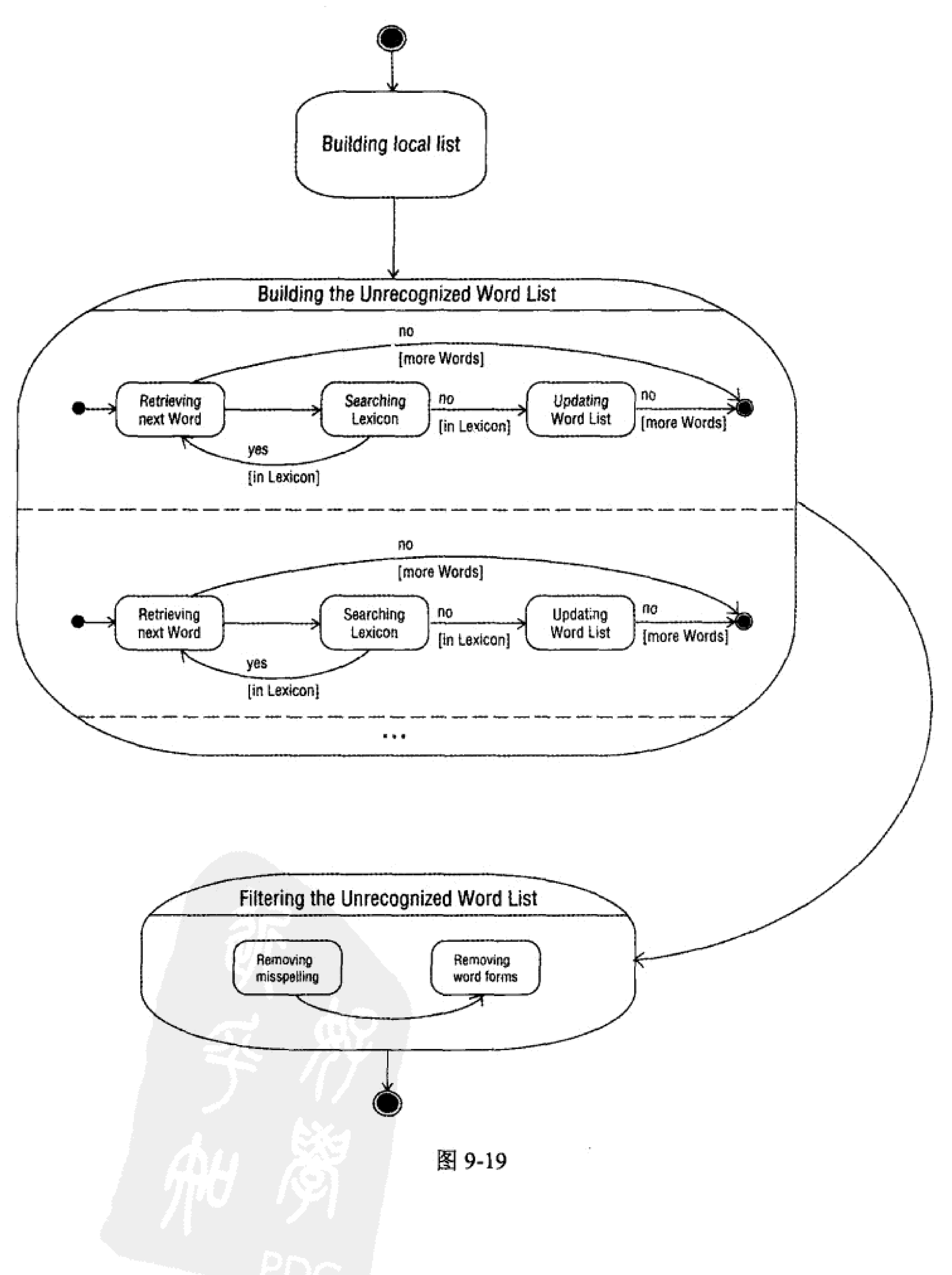


图 9-19

在图 9-19 中，有另一个组合状态叫做 Filtering the Unrecognized Word List。这个组合状态中的子状态是顺序的；它们不被并发执行。首先，删除拼写错误的单词，然后，删除已被识别单词的词形。

9.4 整个系统的可视化

一个系统是由许多元素组成的，包括为了完成某些目的而组织成协作的子系统。它是加入了正常交互的构造的聚合。本章所讨论的制图技术使得开发人员可以从不同观点、不同层次及设计开发系统中的不同控制流来为一个系统建模，从而为系统的设计和开发提供帮助。在本节中，我们讨论对整个系统进行建模和记录，意味着可在最高层次上来描述主要组件或功能元素。本节所讨论的制图技术是用来为系统架构进行建模的。

注意：
尽管这是本章最后一节，但对整个系统进行建模与记录应当是设计和开发一个系统的第一个层次。

当建模与记录系统架构时，系统视图是最高层次。Grady Booch、James Rumbaugh 和 Ivar Jacobson 是这样定义架构的：

注意：
关于软件系统的组织的重大决策的集合，选择组成系统的结构元素与接口，以及这些元素相互协作时所体现的行为，这些结构元素和行为元素组合使它们逐渐合成为更大的子系统，用于指导这个系统组织的架构风格：这些元素以及它们的接口、协作和组合[Booch、Rumbaugh 和 Jacobson，1999]。

对架构的建模与记录将捕捉系统的逻辑和物理元素，以及系统在最高层级的结构和行为。

系统的架构是从一个独特的视图对系统的描述，该视图集中于系统的结构和组织。视图如下：

- **用例**：描述呈现给最终用户的系统行为
- **过程**：描述系统并发与同步机制中使用的进程与线程
- **设计**：描述提供给最终用户的服务与功能
- **实现**：描述创建用于实际系统的组件
- **部署**：描述软件组件以及软件组件在发布系统中执行时所在的节点

可以看出，这些视图重叠且彼此交互。在设计视图中可以使用用例。在实现视图中可以以组件来显示过程。在实现与部署视图中均使用了软件组件。在设计系统架构时，应当建立反映所有这些视图的图。

系统可以被分解成子系统与模块。子系统或模块将被进一步分解为组件、节点、类、

对象和接口。在 UML 中，在文件架构层次上使用子系统或模块叫做包(package)。包可用来将元素组织成组，这个组描述了这些元素的一般用途。包使用左上角带有短小突出部的矩形来表示。包符号包括了包的名字。可通过组合、聚合、依赖和关联关系将系统中的包连接起来。构造型指示符可用来区分不同类型的包。图 9-20 显示了 Lexicon 系统中涉及的包。系统包使用<<system>>指示符来同 List Build 和 Filter 子系统进行区分，这些子系统使用<<subsystem>>指示符。由于它们是子系统，因此它们通过聚合关系与系统联系起来。

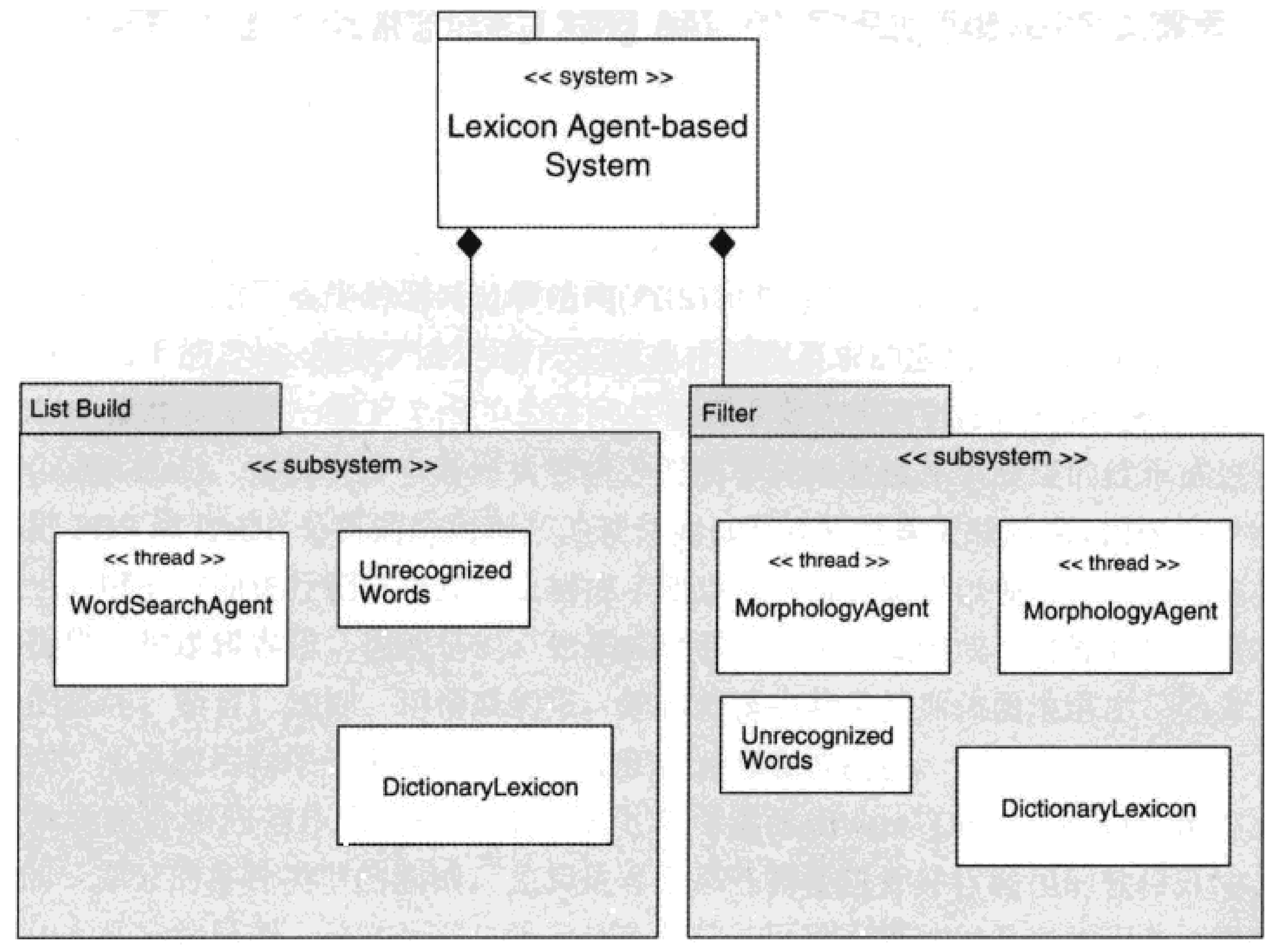


图 9-20

9.5　小结

如前所述，本章讲述了用于设计与记录应用程序并发行为基本 UML 制图与标记技术，但仅仅是对一个非常复杂的主题的简介。介绍了如下要点：

- 系统模型是为了研究系统而收集的信息的主体。文档(documentation)是用于对系统建模的工具。UML(统一建模语言)是由 Grady Booch、James Rumbaugh 和 Ivar Jacobson 创建的用来设计、可视化、建模并记录软件系统结构的图形化标记。它是沟通和建模面向对象系统的事实标准。UML 可从结构和行为的视角来对并发进行建模。
- UML 图可用来为最基本单元、对象、整个系统进行建模。对象是许多 UML 图使用的基本单元。依赖、继承、聚合和组合是对象间可能存在的关系。交互图用来显示对象的行为并识别系统的并发性。对象间可通过通信和调用方法来进行交互。

协作图描述了对象间的交互以共同执行某些特别的任务。顺序图表用来以时间顺序表示对象间的交互。状态图用来描述单个对象在其生命期的动作。

- 当为整个系统建模时，基本单元是包。包可用来表示系统和子系统。包之间可存在关系，例如组合或某些关联类型。

要想对这些技术进行全面了解，我们推荐阅读 *Designing Concurrent, Distributed, and Real-Time Applications with UML*，作者是 Hassan Gomaa(Addison-Wesley,2000)。

下一章将关注多核编程的最后一个方面：测试与逻辑容错。

第10章 并行程序的测试和逻辑容错

第8章介绍了应用程序的谓词分解结构(PBS)和并行应用程序设计层(PADL)分析模型。这些是自上而下的方法，用来产生有着并发或并行编程要求的应用程序的声明式架构。PBS和PADL的最终目标之一就是为应用程序的并发要求建立占有链(chain of possession)或审计追踪(audit trail)，它们引导了如何从解决方案模型转换到操作系统级的线程或进程。本章中将把PBS和PADL分析同应用程序的软件测试及异常处理关联起来。PBS和PADL告诉您应当对哪些方面进行测试以及什么构成了错误或异常。您可以使用异常处理来为声明式架构提供一种逻辑容错。也就是说，如果应用程序由于未知的或不可控制的原因违反了PBS中的陈述、断言、规则、谓词或约束，您希望抛出异常然后体面地退出，因为一旦违反了谓词，那么应用程序的正确性、可靠性和含义都会被损害。

本章还将介绍应当作为任何软件开发生命周期(Software Development Life Cycle，SDLC)的一部分的各种类型的测试，尤其是对于那些需要进行并行编程的软件开发。我们将解释什么是异常处理、异常处理与错误处理之间的区别有哪些以及多线程和并行编程的一些问题。我们还将简要介绍模型检测(model checking)的概念、可能世界语义(possible-world semantics)同并行编程的声明式解释的相关性以及如何将它们应用到测试和异常处理。

具体来讲，本章将讲述以下内容：

- 将PBS和PADL分析同应用软件测试及异常处理关联起来
- 对应当是任何SDLC的一部分的各种类型的测试进行说明
- 定义异常处理以及错误处理与异常处理之间的区别
- 介绍模型检测的概念以及可能世界语义同并行编程的声明式解释的相关性

10.1 能否跳过测试

对软件进行测试的目的是为了确保软件所做的事情恰好是您所希望的。应用程序往往被要求为所需功能的列表。这个列表可以表现为由上百页组成的正式规范，或者可以像雇主口头提出十余条详细要求那样简单。不管对软件的要求列表是如何产生的，测试过程必

PDG

须确保软件能够满足这些要求，并且这些要求能够满足用户的期望。在涉及并行计算机或多处理器的许多场合，用户的期望包括性能提高或特定级别的高性能吞吐能力。当为了满足用户期望而加入多线程或多处理时，软件错误的种类会增加。当软件不按照规范执行时，软件就会处于错误状态，即使违背规范的表现是系统执行速度过慢。需要并行编程的软件非常难以测试与调试。测试与调试位于第 3 章讨论的并行编程的十大挑战之中。测试与调试多线程或多处理程序所特有的一些主要问题是：

- 模拟最小到最大的容积负荷(volume load)
- 在调试期间复制精确的控制流
- 在调试期间复制竞争条件错误
- 复制全系统的进程竞争和线程竞争
- 寻找隐藏的不安全线程函数
- 测试与调试非确定算法
- 证明在软件中不可能出现死锁或数据竞争
- 模拟边界和平均负荷的混合
- 当上百或上千个线程或进程运行时，检测中间结果
- 确定要产生可接受的性能所需要的线程或进程的准确数量

第 8 章中介绍的 PADL 和 PBS 分析建立了初步的并发底层结构(infrastructure)、测试阶段确认(validate)和验证(verify)的实现模型。声明式方法和基于谓词的方法使得它们成为模型检测和测试的更加自动化的形式。声明式的设计带来声明式的实现。声明式的实现使得中等规模到大规模并行程序的测试复杂度位于软件开发者的能力范围之内。无论测试或调试过程的复杂性有多大，其目标是无错且容错地部署软件。测试过程必须找到每个错误和软件缺陷，并将它们去除。

关于程序员道德规范和软件可靠性的公共服务声明

本书中的代码实例很少包含或不包含错误及异常处理。这是因为代码实例仅用于说明目的。我们希望代码实例保持简短，并且不希望将读者的注意力从要展现的概念转移。然而，真实世界的应用程序必须是“防弹的”(bulletproof)。因为作为软件开发者，我们生产应用于医药、制造业、国家安全、交通、金融、教育、科学研究、商业所有领域的应用程序，我们有着伦理和道德上的责任来生产安全、正确、可靠、容错的软件。否则就是玩忽职守。

10.2 测试中必须检查的 5 个并发挑战

第 3 章中的一些并发挑战必须在测试阶段进行检查，并在异常处理程序中进行解决。这些挑战是：

(1) 并行执行的两个或多个任务间的不正确的和不充分的通信

(2) 由两个或多个指令或任务的不安全数据更新造成的数据损坏(data corruption)

(3) 当任务和资源之间存在多对一的比例时的资源竞争

(4) 需要并行执行的单元数无法被接受

(5) 在沟通包含多处理和多线程的软件设计时缺少文档或文档不完整

第 7 章讨论了用于使能并同步并发执行的线程或进程间的通信、数据或设备访问的机制。例如，使用互斥量和信号量控制和防止前面列表中因为挑战 2 而发生的错误。使用定时互斥量控制和防止前面列表中因为挑战 3 中的问题而导致的错误。在许多情况下，文档受到的关注和使用的专用资源都很少，但它是软件部署中最重要的组件之一。就像所有其他事情对于并行编程和多线程一样，文档对于这类应用程序也变得更加关键。测试过程应当验证并确认设计文档与后期制作文档是匹配的。表 10-1 显示在第 7 章讨论过的哪些机制可以用来预防控制和预防前面列表中提到的 5 种挑战。

表 10-1

信号量类型	描　　述
互斥量信号量	用于在代码的临界区中实现互斥现象的机制
读写锁	用来在任务间实现读写访问策略的机制
条件变量	用来在任务间广播某事件已经发生的信号的机制； 当某个任务对事件互斥量加锁后，它会阻塞，直到接收到广播
多条件变量	除包含多个事件或条件以外，与事件互斥量相同

表 10-1 中列出的是底层机制。幸运的是，使用高级组件库的特性，例如线程构建块(Threading Building Blocks，TBB)或是标准 C++并发编程库，能够在测试过程当中避开一些麻烦。这些问题需要在第 8 章中讨论的 PADL 分析模型的第 2 层和第 3 层进行处理。

无论如何，我们首先需要确立一些定义。有一些词汇在测试、错误处理和容错中经常被错误地使用或被过于宽松地使用。表 10-2 包含本章将用到的术语的基本定义。

表 10-2

术　　语	定　　义
缺陷(defect)	软件或软件需求的任何方面的缺点，它导致或可能导致发生一次或多次失效
错误(error)	软件工程师/程序员做出的不适当的决策，会导致软件缺陷
异常处理(exception handling)	管理异常(程序执行中未预料到的情况)的机制，会改变程序/软件的正常地执行流
失效(failure)	由故障导致软件元素的操作发生了不可接受的背离
故障(fault)	由于人的错误导致的软件中的缺陷，在特定条件下运行时导致失效
容错(fault tolerance)	使得软件能够在由故障(缺陷)导致的失效中存活并恢复的特性，这些故障(缺陷)是由于人的错误引入到软件当中的
可靠性(reliability)	在特定条件下的规定时段内，软件执行要求的功能的能力

由于表 10-2 中的一些术语，如错误、失效、故障，会以很多方式大量使用，因此我们提供了它们在本章中如何使用的简单定义。容错的一种度量方式是软件能够最小化失效的影响的程度。实现容错软件是所有工程师努力的主要目标之一。但是，容错软件同经过反复测试的软件之间的区别经常会被错误理解和模糊。有时，软件验证、软件确认与异常处理的责任和活动经常被错误地交换。为了达到使用 C++异常处理器制帮助完成逻辑容错的软件的目的，必须首先清楚异常处理在事物发展过程中的适用情况。

10.3　失效：缺陷与故障导致的结果

失效发生在软件操作的运行时。硬件、软件或人工操作中的缺陷导致失效。如果软件没有运行，则它不会遇到缺陷。尽管这是一个显而易见的陈述，但它对于理解测试阶段与异常处理阶段在职责和活动之间的区别非常重要。理想情况下，软件缺陷会在检测阶段去除，对硬件缺陷也是如此。对于人工操作中的缺陷，我们希望可通过培训和经验将它们去除，但说易行难。为简化问题，我们将主要讨论软件缺陷。

10.3.1　基本的测试类型

表 10-3 描述了在软件投入运行之前应当进行的 7 种基本测试类型。

表 10-3

测 试 类 型	描　　述
单元测试	要求每次针对一个组件或单元进行测试；单元可能是指软件模块、模块集合、函数、过程、对象、算法，或在某些情况下是指计算机程序
压力测试	被设计为将系统的组件推动到其极限，有时候甚至超出其极限；压力测试包括边界条件测试；边界条件测试的结果，有助于确定软件组件或系统组成处于边界条件下时会发生什么事
集成测试	用来测试组件的组装；组件合并成逻辑组，每个逻辑组作为一个单元进行测试；组可以进行的测试类型与单元进行的测试类型相同；随着每个组件被加入到组装中，必须测试的元素的数量成组合式增长
回归测试	用来重测已经改变的模块；回归测试确保对组件的改动不会导致它失去任何功能性
运行测试	用来在系统全面运行的情况下对系统进行检测；这个测试将软件组件放到完整系统负荷的现场环境下进行测试；组件在单元测试、集成测试和压力测试中经历的测试通常也可以用作运行测试；运行测试还用来确定组件在一个完全陌生环境中的运行情况将会怎样
规范测试	用作软件验证过程的一部分；针对最初的规范对组件进行审计；这个规范指出系统中包含哪些组件以及这些组件之间有怎样的关系
验收测试	被模块、组件、系统的终端用户用来确定执行情况；验收测试是软件确认过程的一部分

当您对软件进行表 10-3 中的类型的测试时，您发现缺陷并将它们从软件中去除。在测试过程中发现并去除的缺陷越多，软件在运行时遇到的缺陷就越少。运行时遇到的缺陷会导致软件失效。软件失效将产生异常条件(软件在异常条件下运行)。异常条件要求异常处理。因此，需要在以下两者间采取平衡措施：

- 测试阶段期间的缺陷排除
- 异常处理期间的缺陷存活

10.3.2　缺陷排除与缺陷存活

尽管您可以选择更赞成缺陷存活而不是缺陷排除，但问题是异常处理代码可能会非常复杂，以至于又在软件中引入缺陷。这样，异常处理成了失效的源头，而不是提供一种机制来帮助实现容错。优先选择缺陷存活而不是缺陷排除降低了软件正常运转的可能性。全面而彻底的检测会去除缺陷，从而减轻对异常处理的压力。值得一提的是异常处理不会以独立的代码段的形式存在。它们存在于整体软件架构的环境中。软件中的容错之旅常常是从达到以下认识开始的：

- 再多的异常处理也无法拯救有缺陷或不适当的软件架构
- 软件容错同软件架构的质量直接相关
- 异常处理架构不能取代各个测试阶段

为使得关于异常处理的讨论清晰明确，理解异常处理架构作为整体存在于软件架构的环境中很重要。这就是说，异常是由 PBS 和 PADL 分析来确定的。解决方案模型包含 PBS。如果应用程序架构的 PBS 有无法避免、不可控制、无法解释的背离，则存在异常。因此，异常是通过明显地铰接在一起的架构来定义的。如果软件架构不适当、不完整、考虑不足，那么任何进行事后异常处理的企图都是高度可疑的。另外，如果在测试阶段走了捷径(如不完全的压力检测、不完全的集成测试、不完全的白盒测试等)，那么必须永远地加入异常处理代码并使它的复杂度持续增加，最终降低了软件的容错性能减少了应用程序的声明式架构的优势。另一方面，如果软件架构是合理的，并且异常处理架构与 PBS 和 PADL 的第 3 层、第 4 层和第 5 层兼容且一致，那么并行程序就可以实现高度容错了。如果您在达成上下文失效复原(context failure resilience)的目标时，理解软件应用程序架构和测试所发挥的作用，那么显然您会优先选择缺陷排除，而不是缺陷存活。缺陷排除在测试中发生。

10.4　如何对并行程序实现缺陷排除

首先，我们应指出测试应当伴随 SDLC 中每个主要活动，从需求收集活动到软件维护活动。但是，涉及多线程或多处理的程序在测试阶段需要进行更多的工作。因此，需要在测试计划中利用 PADL 分析和 PBS 分解。将并行程序的测试目标分解为回答 3 个基本问题：

(1) 设计模型和 PBS 能正确并完全地表示解决方案模型吗(假定解决方案模型能够解决最初的问题)？

(2) 实现模型正确地映射到设计模型(PADL 的第 4 层和第 5 层)和 PBS 了吗？

(3) 实现模型中所有的并发挑战都得到解决了吗？

传统上，测试并行程序的大部分工作是用在回答第 3 个问题。这通常发生在命令式、自底向上的并行编程方法中。然而在声明式方法中，首先需要回答第一个问题。这是最重要的测试。如果这个测试失败，就没有理由去继续执行任何进一步的测试。第 8 章已经给出了这些模型的简单版本，但是在发布的代码中，这些模型应当包含大量的细节。如果设计模型足够详细准确、PBS 是完整的，而且它们都能准确地映射到最初的解决问题中，这样就处于良好的状态中。软件应用程序的可扩缩性或演化的机会是由应用程序架构和并发底层结构所决定的。在回答第一个问题的过程中，我们测试了两者的质量。如果第二个问题的答案是肯定的，那么该应用程序是可靠的。因此，尽管传统上认为对问题 3 的回答是最重要的，但此时它的重要性要次于问题 1 和问题 2 的回答。标准软件工程测试技术被用来回答这些问题。我们只需要使用 PADL 和 PBS 来明确表达这 3 个基本问题。测试将验证应用程序符合 PADL 和 PBS。

为搞清楚上述工作原理，可以查看标准测试阶段前面的处理流程。

10.4.1 问题陈述

回忆第 8 章中对游戏场景的精化：我的忠实助手已经交给我一个 6 字符编码。编码中的字符可以重复出现。然而，编码只可包含数字 0～9 和字符 a～z。您的任务是猜测助手给了我什么编码。在游戏中，设定蜂鸣器两分钟后响起。如果您在两分钟内猜到了该编码，那么您就赢了。如果我的忠实助手确定您在 15 秒的时间间隔中做了超出 N 次的不正确猜测，那么他会交给我新的编码，并保证该编码位于您已经做过的猜测之中。

10.4.2 简单策略和粗解决方案模型

根据问题陈述，您设计出如下的简单策略：您恰好在一个文件中有着可以从中进行选择的 4 496 388 个所有可能编码。简单的办法是对文件进行穷举搜索，将文件提供的每个编码作为一次猜测。但由于有着两分钟的时间约束，您无法确定是否有足够的时间。因此，您的策略是找出一种方法来同时对文件的多个部分同时进行搜索。您决定将最初的文件分成 8 份，并对它们进行并发搜索。因此，您应能够在对文件的 1/8 进行搜索所需的时间内猜测出正确的编码。同时您还做了一个预防，如果您能够查找所有 8 个文件，在未能准确猜出而且也未超时的前提下，您将会把文件分成 64 份，并进行重试。如果您失败了，但仍有时间，将会把文件分成 128 份，并进行重试，依此类推。

10.4.3 使用 PADL 第 5 层的修正的解决方案模型

既然您已经确定并行化在解决方案模型中是有用的，现在您将 PADL 分析作为 SDLC 的一部分。PADL 的第 5 层涉及为问题模型和解决方案模型确定适当的应用程序架构。初

看上去，多 agent 架构最适合于解决方案模型。因此，在多 agent 应用程序架构的上下文中精化解决方案模型。

1. 修正的 agent 模型

从问题的陈述中可以很容易看出最初要面对 3 个 agent。如果根据 agent 对游戏进行重新描述，则为：Agent A 将编码提供给 Agent B。Agent C 试着猜测 Agent B 的编码。如果 Agent C 在 15 秒内提出过多的猜测，Agent A 就会给 Agent B 一个新的编码，并确保该编码是 Agent C 已提出过的。如果 Agent C 遍历了所有的可能但仍未赢得比赛，并且 Agent C 仍有剩余时间，Agent C 将以更快的速度进行相同的猜测。Agent C 意识到在给定的时间期限内生成足够的猜测以确保成功，它需要得到帮助。因此，Agent C 征募一队 agent 来帮助提出猜测。对全部的可能性每完成一趟遍历，Agent C 会征募更大的一队 agent 来帮助提出猜测。

2. agent 的并发模型

在修正的 agent 模型中，您已经确定需要使用的并发模型是 boss-worker 模型、对等(peer-to-peer)、单程序多数据(SPMD)、单指令多数据(SIMD)和互斥读互斥写(EREW)。您在这里使用 SPMD/SIMD，是由于 agent 对不同的数据集使用相同的搜索技术。EREW 适用于 agent 同 boss 的通信。boss-worker 模型适用于猜测者 agent 与它的帮助者之间的关系。对等模型适用于编码所有者与忠实助手之间的关系。

10.4.4　agent 解决方案模型的 PBS

分解 1：如果您的猜测是正确且及时的，您就会在游戏中获胜。

分解 2：如果猜测是由 6 字符编码组成且编码只包含字符(a～z,0～9)的组合，考虑到重复是允许的，而且该编码是我的 agent 递交给我的，则称猜测是正确的。

分解 3：如果您的猜测是正确的，并且是在 2 分钟之内产生的，那么您的猜测是及时的。

分解 4：如果有足够多的 agent 进行搜索，那么对编码进行穷举搜索将会成功。

分解 5：*N* 个 agent 足以在 2 分钟内从四百万个编码的样本中找到正确的编码。

分解 6：如果编码每 15 秒改变一次，需要 4*N* 个代理 2 分钟内从四百万个编码的样本中找到正确的编码。

1. PBS 的声明式实现

程序清单 10-1 遵从 PBS 的声明式语义。

程序清单 10-1

```
// Listing 10-1   A declarative implementation of the guess_it program.
```

```
 1  #include "posix_process.h"
 2  #include "posix_queue.h"
 3  #include "valid_code.h"
 4
 5
 6  char **av;
 7  char **env;
 8
 9
10  int main(int argc,char *argv[],char *envp[])
11  {
12
13     valid_code  ValidCode;
14     ValidCode.determination("ofind_code");
15     cout <<  (ValidCode() ? "you win" : "you lose)";
16  }
```

这样，在第 13 行声明了 ValidCode 谓词。这个谓词用来表示如下陈述：

```
This is the code the trusted agent handed you.
```

在第 14 行，您通过调用谓词 ValidCode( )来检测这个陈述。ValidCode( )谓词产生 4 个进程，每个进程又会依次产生两个线程。因此，实际上 ValidCode( )谓词是使用并行编程来实现的。然而，它的实现被封装了起来。程序清单 10-2 显示了 valid_code 谓词类的声明。

程序清单 10-2

```
//Listing 10-2  Declaration of the valid_code predicate class.


 1  #ifndef __VALID_CODE_H
 2  #define __VALID_CODE_H
 3  using namespace std;
 4
 5  #include <string>
 6  class valid_code{
 7  private:
 8     string Code;
 9     float  TimeFrame;
10     string Determination;
11     bool InTime;
12  public:
13     bool operator()(void);
14     void determination(string X);
15  };
16
17  #endif
```

在 C++中，谓词是有着返回布尔值的 operator( )方法的类。应使用 C++谓词来模拟 PBS 中谓词的概念。由于谓词是一个 C++类，因此可以与容器及算法共同使用。程序清单 10-3

显示了谓词类的定义。

程序清单 10-3

```
// Listing 10-3 Definition of the valid_code predicate.

 1  #include "valid_code.h"
 2  #include "posix_process.h"
 3  #include "posix_queue.h"
 4
 5  extern char **av;
 6  extern char **env;
 7
 8
 9  bool valid_code::operator()(void)
10  {
11     int Status;
12     int N;
13     string Result;
14     posix_process Child[4];
15     for(N = 0; N < 2; N++)
16     {
17        Child[N].binary(Determination);
18        Child[N].arguments(av);
19        Child[N].environment(env);
20        Child[N].run();
21        Child[N].pwait(Status);
22     }
23     posix_queue PosixQueue("queue_name");
24     PosixQueue.receive(Result);
25     if((Result == Code) && InTime){
26        return(true);
27     }
28     return(false);
29  }
30
31
32  void valid_code::determination(string X)
33  {
34     Determination = X;
35  }
```

下面是程序清单 10-1、程序清单 10-2、程序清单 10-3 和程序清单 10-4 的程序概要，其中程序清单 10-4 将会在本章稍后部分出现。

程序概要 10-1

程序名：

pguess_it (程序清单 10-1)

描述：

这个程序是个“猜谜(guess it)”游戏。如果您在 2 分钟内猜对了，您就在游戏中获胜。如果猜测是由 6 字符编码组成且编码只包含字符(a～z,0～9)的组合，并考虑到重复是允许的，则猜测是正确的。

ValidCode 谓词的声明是为了表达如下陈述：

```
This is the code the trusted agent handed you.
```

这个陈述是通过调用 ValidCode( )谓词来检测的。ValidCode( )谓词产生 4 个进程，每个进程依次产生两个线程。它的实现是被封装的。

必需的库：

```
rt
```

所需的其他源文件：

pguess_it.cc (见程序清单 10-1)、posix_process.cc (见程序清单 10-4)、valid_code.cc (见程序清单 10-3)和 posix_queue.cc

注意：

由于长度过长，posix_queue.cc 在本书中并未完整地列出，可在 http://www.wrox.com 中下载到 posix_queue.cc 的完整代码，以及书中其他示例代码。

必需的用户定义头文件：

posix_process.h (见程序清单 5-3)、valid_code.h (见程序清单 10-2)和 posix_queue.h (见程序清单 7-3)

编译和链接指令：

```
c++ -o pguess_it pguess_it.cc valid_code.cc posix_process.cc posix_queue.cc
-lrt
```

测试环境：

Linux Kernel 2.6

Solaris 10、gcc 3.4.3 和 gcc 3.4.6

处理器：

Multicore Opteron、UltraSparc T1 和 Cell Processor

注释：

无

2. 如何知道代码有效

您使用了标准软件工程测试技术，但是是在 PADL、PBS 和本章前面提到的 3 个基本问题的上下文下使用的。为了回答这个问题，您需要了解将 PBS 作为成功的基本量尺，标准软件工程测试可以进行到什么程度。

10.5　什么是标准软件工程测试

就像我们之前已经说明的，有很多种类型的软件测试：软件设计阶段完成的测试、软件实现阶段完成的测试、安装与运行测试、可用性测试、验收测试等，它们决定了客户对交付的系统是否满意。由于当前软件系统的复杂性，以及它们的经济价值与社会价值，软件测试已经成为高度专业化领域。电气与电子工程师协会(Institute of Electrical and Electronics Engineers，IEEE)为一整套测试活动发布了标准。IEEE，作为一个非营利组织，是在推进科技进步方面全球领先的专业协会。IEEE 发布的两个重要测试标准文档是：

- IEEE Std 1012，为软件验证与确认提供指导
- IEEE Std 1008，适用于单元测试

注意：

每个软件开发组均应备有这些文件。如果没有，可以从网站 http://www.standards.ieee.org 下载。

本章对 Std 1008 与 Std 1012 所涵盖的问题做了简要概览。由于测试是任何 SDLC 的基本活动之一，每个 SDLC 中都应包含 Std 1008 与 Std 1012。如果坚持这些标准，那么这些标准中的指导和建议能够大幅提高最终交付给用户的软件的质量。从 Std 1008 与 Std 1012 收集到的要求与规范可以帮助您回答以下两个问题：

(1) 我是否在构建正确的软件？

(2) 我是否在正确地构建软件？

这两个问题的答案可处理软件的验证与确认中的问题。

10.5.1　软件验证与确认

进行软件验证与确认是为了在表 10-3 中提到的各种类型的测试期间中去除缺陷。当您

进行软件确认时，您是在回答第一个问题，而软件验证则回答第二个问题。确认是针对软件规范对软件特性进行审计。根据 IEEE Std1012:

> 软件验证与确认(V&V)是系统工程的技术学科。软件 V&V 的目的是为了帮助开发机构确保软件在软件生命周期中的质量。软件 V&V 过程决定了为指定的活动开发的产品是否符合该活动的需求，以及软件是否满足了预期用途和用户需要。这个判定包括了对软件产品和过程的评估、分析、鉴定、审查、检查和测试。软件 V&V 与软件开发同步进行，并不是位于软件开发结束时。

在 SDLC 的需求和分析活动中，生成软件规范。出于本书所讨论的多核应用程序设计的目的，规范来自 PADL 分析以及 PBS 的设计模型和实现模型。当您执行软件验证时，是在检查软件是否符合这些规范。当您从事于确定软件是否实际执行用户希望的工作时，您是在回答第二个问题。验证是实现“软件的正确(software right)”，而确认是实现“正确的软件(right software)”。软件测试过程的大部分可以被描述为验证过程或确认过程。最终，在 SDLC 中必须进行表 10-3 中所有类型的测试。

除了表 10-3 列出的 7 种测试类型，还有 6 种重要的错误与并行编程相关，需要您引起注意。表 10-4 列出了常见的并行编程错误以及它们的描述。

表 10-4

常见并行编程错误	描　　述
死锁	任务等待不会发生的事件
优先级倒置	发生在较低优先级的任务阻塞了较高优先级任务的执行时，这种情况在使用同步变量或对资源进行竞争时发生
性能降级	发生在系统的性能在响应性、执行时间、结果计算等方面降低或降级时
无限期推迟	发生在当其他任务受到注意并分配到资源时，系统无限期地拖延了任务的调度
互斥量耗尽	发生在系统达到能够创建的互斥量的最大数目时
线程耗尽	发生在系统达到能够分配的线程的最大数目时

10.5.2　代码不能正常工作该怎么办

例如，执行了最基本的单元测试后，假设您发现程序清单 10-1 中的程序未通过单元测试。程序不能正常工作。程序不能正常工作意味着什么？这个程序失败的原因是它不符合 PBS 的规范。这意味着程序使 PBS 中的一条或多条陈述为假。由于 PBS 目的是捕捉并发底层结构的含义，您知道程序中的并行性将会在某一位置上失败，因为它没有正确地实现 PBS 中的断言、陈述和谓词。特别是 PBS 的第 6 条。

分解 6：如果编码每 15 秒改变一次，需要 4*N* 个代理 2 分钟内从四百万个编码的样本中找到正确的编码。

回顾程序清单 10-1 中第 13 行～第 15 行引用的用户定义谓词 valid_code( )。它说明搜索是否成功。但为了使搜索成功，PBS 中所有的断言都必须被保持。如果仔细看一下程序清单 10-3 中第 9 行～第 30 行的用于 valid_code 的谓词 operator( )，您会发现产生了 4 个 posix_process。程序清单 10-4 给出了用户定义的 process_interface 类的定义。

程序清单 10-4

```
// Listing 10-4  Definitions for user-defined posix_process interface
// class.

 1  #include "posix_process.h"
 2  #include <sys/wait.h>
 3
 4
 5  posix_process::posix_process(string Path,char **av,char **env)
 6  {
 7
 8     argv = av;
 9     envp = env;
10     ProgramPath = Path;
11     posix_spawnattr_init(&SpawnAttr);
12     posix_spawn_file_actions_init(&FileActions);
13
14
15  }
16
17  posix_process::posix_process(string Path,char **av,char **env,
                                 posix_spawnattr_t  X,
                                 posix_spawn_file_actions_t Y)
18  {
19     argv = av;
20     envp = env;
21     ProgramPath = Path;
22     SpawnAttr = X;
23     FileActions = Y;
24     posix_spawnattr_init(&SpawnAttr);
25     posix_spawn_file_actions_init(&FileActions);
26
27
28
29  }
30
31  void posix_process::run(void)
32  {
33
34     posix_spawn(&Pid,ProgramPath.c_str(),&FileActions,
                  &SpawnAttr,argv,envp);
35
36
```

```
37  }
38
39  void posix_process::pwait(int &X)
40  {
41
42     wait(&X);
43  }
```

程序清单 10-4 中的类是为 POSIX API posix_spawn( )适配接口的接口类。posix_process 类用来产生两个 ofind_code 的副本。ofind_code 依次产生两个线程。问题是最终您仅得到 8 个连续的 agent(4 进程×2 线程)开展工作。因此，如果编码被改变，PBS 中的第 6 条要求您将 agent 的数目加到 4 倍，这不是由 valid_code 谓词解决的。在这种情况下，您可以简单地将每个 agent 映射到一个线程。如果应用程序中的谓词与 PBS 不一致，那么您将不会通过单元测试。

这种将 PBS 同 C++中的谓词匹配起来是转向声明式风格的并行编程的基础的一部分。这是一个细微的但非常强大的想法。您将软件验证与确认(V&V)应用到 PBS 的映射，然后 C++谓词将并行编程的声明式风格扩展到 V&V。V&V 的其他重要目的是：

- 推动软件错误的早期检测与改正
- 增强对过程和产品风险的深入管理
- 支持软件生命周期处理，以确保符合程序性能、进度和预算需求

如果根据 PBS 对谓词进行审计，可以在软件部署之前较早地发现并发问题。实际上，PADL 和 PBS 的主要目的之一就是在书写任何代码之前处理并发需求的全部复杂性。

PADL 和 PBS 的后果之一就是它们支持测试的声明式模型，如模型检测和可能世界分析(将会在本章稍后部分简要介绍)。这些模型可在 SDLC 的早期部署。在软件开发工作的初期，就必须对多线程和多处理程序的复杂性和完整性进行管理。测试过程必须反映出这一点。请看来自 IEEE Std 1012 的如下内容：

> 在规划 V&V 过程时，软件的完整性级别通常在开发过程的早期指定，更好的做法是在系统需求、分析及架构设计活动中指定。软件的完整性级别可被指定给软件需求、功能、功能组或软件组件及子系统。被指定的软件完整性级别可能会随着软件的演变而变化。开发机构选择的设计、编码、程序上的、技术上的实现特性可以提升或降低软件的危险程度和指定给软件的相关软件完整性级别。

V&V 在单元测试级别上非常有效(如果被使用)，如同 ValidCode( )谓词在单元测试级别上失败的例子所说明的。单元测试过程由共被分为 8 个基本活动的 3 个阶段组成：

(1) 进行测试计划编制
   a. 规划总体方法、资源和进度
   b. 确定要进行测试的特性
   c. 精化总体规划

(2) 获取测试集
   a. 设计测试集

b. 实现精化后的计划和设计

(3) 测量测试单元

a. 执行测试过程

b. 检查终止

c. 评估测试投入和单元

我们建议您在 PADL 模型中的第 4 层分析结束及 PBS 完成后，开始将这些基本活动集成起来。表 10-5 中列出了应当在第 4 层分析的测试阶段定位的其他常见错误类型。

表 10-5

错误的种类	描　述
用户接口错误	用户接口的设计、功能或性能方面的错误；用户接口可能会在通信、命令结构或输出等方面失败；程序可能会无法完成用户期望的任务或执行任务时非常笨拙
边界相关的错误	程序的边界是指使得程序改变其行为的任何事；错误地描述边界条件或未能检测到程序的限制都会产生错误
计算错误	计算期间发生的错误；这些错误包括曲解公式或丢失精度；它还包括由于使用不正确的算法导致的计算上的错误
初始和后续的状态	初次使用程序时发生的错误；每次程序重新启动时都会发生这些错误
竞争条件	当一个线程或进程在期望的线程或进程之前执行时发生的错误
控制流错误	当程序执行错误的后续步骤时发生的错误
错误处理	当处理错误时产生的错误；程序可能无法检测或预料可能的错误，并且无法合理地改正这些错误
处理或解释数据中的错误	当数据在模块、程序、线程、进程间来回传递时，被曲解、毁坏或丢失时发生的错误
负荷状态	当程序超载时发生的错误；程序在长期执行很多工作或突然完成大量工作时可能会发生错误；程序可能会在内存不足、与其他程序或例程共享的资源耗尽时失败；程序到达其限制时被执行的状态如何？当程序超出其限制时会执行得多差？
硬件	当程序不能识别硬件故障或无法从硬件故障中恢复时发生的错误；程序可能无法识别从设备返回的错误编码，或者可能试图访问不存在的或已经被使用中的设备；程序也可能向设备发送错误的数据
源和版本控制	当使用不正确版本的程序时发生的错误；有可能包含错误的较旧版本的子过程被链接到程序的最新版本
文档	当使用错误的文档时发生的错误
测试错误	在软件测试期间发生的错误

使用应用程序的声明式架构，加上将 PBS 和 PADL 映射到单元测试阶段的早期，可以处理并行编程开发工作中的最重要的挑战。这项技术被使用 C++异常处理的应用程序所支持，该技术被称作逻辑容错。

10.5.3 什么是逻辑容错

您可以使用 C++异常处理工具和异常类来加强 PADL 和 PBS 的语义。通过继承扩展异常类和错误类，有助于测试过程捕捉应用程序 PBS 的 C++谓词实现中不合逻辑的情况。我们将这个过程称作对应用程序加入逻辑容错。

C++异常处理工具的基础如何被使用对软件的架构至少有两个重要的影响：

- 软件架构中的控制流会被 throw 机制改变。
- 使用的异常类引入新的类型，而且每种类型有其自身的语义。

从问题域到知道如何将系统带领到一致状态的其他区域的控制转移，以及了解异常抛出的语义，使得您可以开始研究逻辑容错的目的。异常抛出的语义描述了异常条件是什么以及建议应当做些什么。控制的转移将您带到实现异常策略的代码。异常策略被设计为使得软件对缺陷和系统失效具有弹性。在 C++中，catch( )机制既可以直接实现异常策略，也可以创建对象并调用实现异常策略的函数：

```
catch(some_exception){

    //Execute exceptions strategy

}
```

1. 异常处理程序

catch{}块被称作异常处理程序。C++程序可包含多个异常处理程序。每个异常处理程序同一种或多种类型关联(取决于异常的类层次结构)。异常处理程序的 3 个基本功能是：

- 登记它可以处理的异常类型
- 对发生了什么异常进行记录或以某种方式记入日志(有时这需要通知)
- 执行适当的异常处理策略

异常处理策略有许多种。终止模型(termination model)中的异常处理策略的主要目的是将软件带回到一致的状态，从而使软件可以继续在某个可接受的级别下运行。表 10-6 包含了常用的异常策略。

表 10-6

异 常 策 略	描　　述
资源再分配和回收	试图： ● 再分配内存 ● 关闭文件 ● 释放互斥量 ● 关闭信号量 ● 释放内存 ● 查找文件 ● 关闭进程 ● 改变引发问题的进程的安全性
事务或数据回滚	撤销未完成事务的步骤，将数据回滚到数据有效的某个检查点
操作重试	重试一次操作： ● 使用最初的资源 ● 使用改变后的资源 ● 在一段时间间隔之后 ● 在附加条件被满足之后
冗余和故障恢复	将处理移交给其他线程或与当前进程并行运行的进程
通知外部援助	请求从其他软件 agent、拟人化用户或其他系统获得援助

使用哪种异常处理策略将对软件架构产生很大的影响。这就意味着必须在软件设计阶段就将异常处理策略包括进来。在接近并行编程的声明式解释中，您正在向着逻辑模型移动。最终，您希望将不合逻辑的模型或不合理的程序行为视作异常。因此，异常处理策略从 PADL 的第 5 层和 PBS 中产生。它是软件架构的基本部分。如果总体软件架构是脆弱的，那么异常处理策略注定会失败。

异常抛出的语义与实现的异常策略紧密相关。在软件架构的上下文中理解异常的语义同决定在异常处理过程中将控制转移到哪里同样重要。C++标准定义了多个有着自身语义的内置异常类。图 10-1 给出了 C++异常类的类关系图。

这些异常类可通过继承得到扩展。C++也支持用户定义的异常类。

标准 C++类库有 9 种异常类，被分成两组：运行时错误组和逻辑错误组。运行时错误组代表了难以预防的错误，而逻辑错误组代表了理论上可以避免的错误。

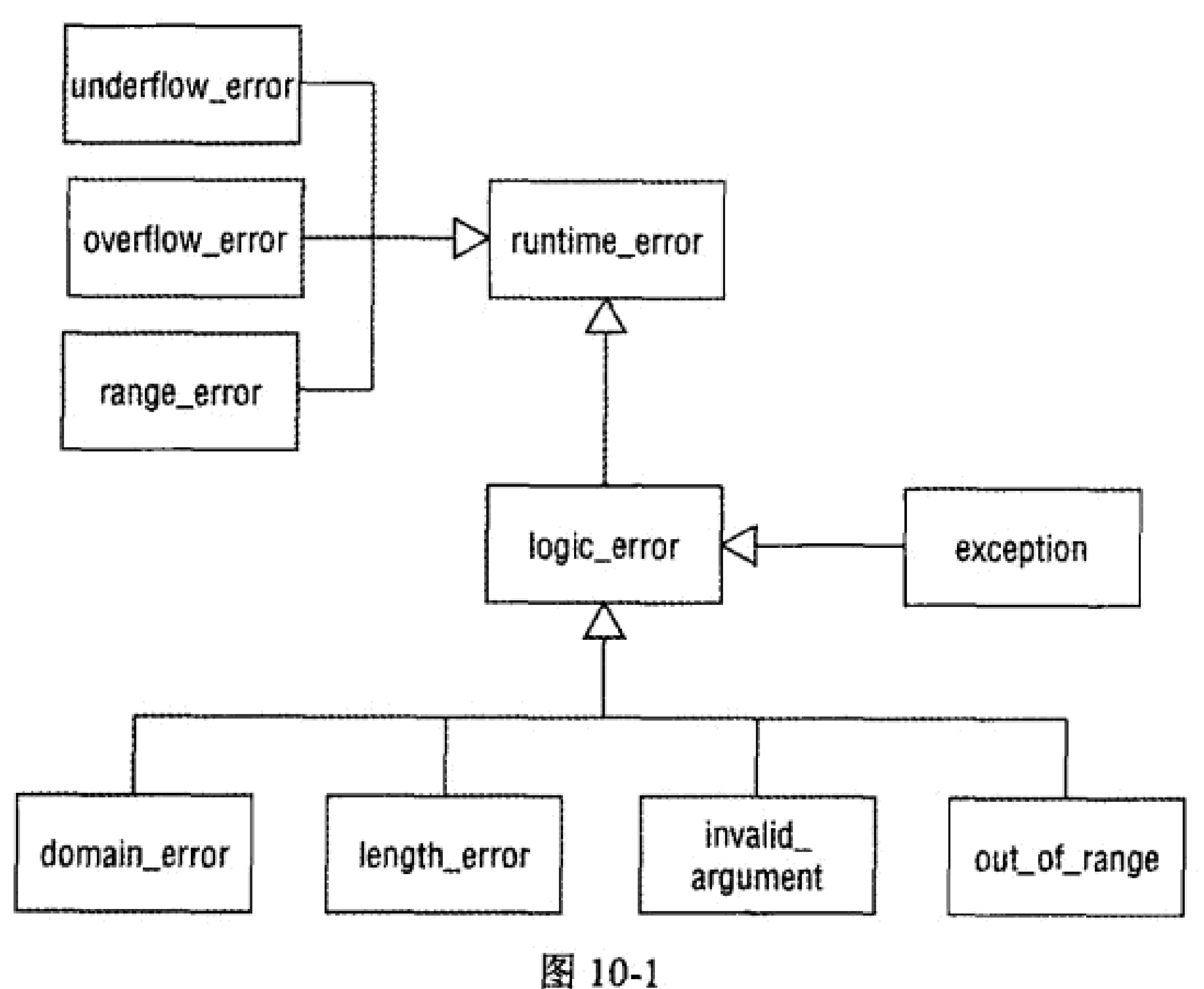


图 10-1

runtime_error 类

runtime_error 类系列是从 exception 类派生来的。3 个类从 runtime_error 类派生得到：range_error、overflow_error 和 underflow_error。runtime_error 类报告内部的计算或算术错误。runtime_error 类从异常类祖先继承得到主要的功能。what( )方法、赋值运算符 operator=( )和异常处理类的构造函数为 runtime_error 类提供了能力。runtime_error 类提供了一个异常框架和架构蓝图。

logic_error 类

logic_error 类系列是从 exception 类派生来的。实际上，logic_error 类系列的大部分功能也是从 exception 类继承来的。exception 类包含了 what( )方法，用来报告给用户所抛出的错误的描述。每个 logic_error 类系列中的类包含一个构造函数，用来裁剪针对该类的消息。

与 runtime_error 类类似，这些类是被设计为专用的。除非用户为这些类增加一些功能，否则它们只报告错误及其类型，其他什么都不能做。因此，9 个通用异常类没有提供纠正动作或错误处理。看一下基本异常类在没有专门化时是如何工作的。示例 10-1 显示了如何抛出一个 exception 对象和 logic_error 对象。

示例 10-1

```
// Example 10-1  Throwing an exception object and a logic_error object.

try{
    exception X;
    throw(X);
}
```

```
catch(const exception &X)
{
   cout << X.what() << endl;
}

try{
   logic_error Logic("Logic Mistake");
   throw(Logic);
}

catch(const exception &X)
{
   cout << X.what() << endl;
}
```

基本的异常类只有构造、析构、赋值、复制和简单的报告功能。它们不包含纠正已经发生的故障的能力。异常类的 what( )方法返回的错误消息将由传递给 logic_error 对象的构造函数的字符串确定。在示例 10-1 中，传递给构造函数的 Logic Mistake 字符串是由 catch block( )中的 what( )返回的。

派生新的异常类

异常类可以照原样的方式来使用，也就是它们仅用来报告描述已经发生的错误的错误消息。然而，这对于异常处理技术而言本质上是无用的。仅知道异常是什么并不能增加软件的可靠性。异常类层次结构的真正价值在于它为设计人员和开发人员提供的结构上的线路图。异常类提供了开发人员可以进行特化的基本错误类型。在运行时环境中发生的很多异常既可放置到 logic_error 类系列中，也可以放置到 runtime_error 类系列中。为了了解如何特化 exception 类，您可以将 runtime_error 作为一个例子。如前所述，runtime_error 类是 exception 类的后代。可通过继承来特化 runtime_error 类。例如：

```
class  concurrent_ file_access_exception : public runtime_error{
protected:
   //...
   int ErrorNumber;
   string DetailedExplanation;
   string FileName;
   //...
public:
   virtual int takeCorrectiveAction(void)
   string detailedExplanation(void);
   //...
};
```

这里，concurrent_file_access_exception 继承了 runtime_error，并通过增加许多数据成员和成员函数使其特化。特别地，增加了 takeCorrectiveAction( )。这个方法可用来帮助异常处理程序执行它的恢复和纠正工作。file_access_exception 对象知道如何识别死锁以及如何打破死锁。死锁是并行编程的主要挑战之一。它还有专门的逻辑来处理能破坏文件的病

毒。它还有着专门的知识来处理意外中断的文件传输。这些情况都能够引发运行时异常。您可将 concurrent_file_access_exception 对象和 C++的 throw、catch 和 try 共同使用。例如：

```
try{
    //...
    fileProcessingOperation();
    //...
}

catch(concurrent_file_access_exception  &E)
{
    cerr << E.what() << endl;
    cerr << E.detailedExplanation() << endl;
    E.takeCorrectiveAction();
    // Handler Take Additional Corrective Action
    //...
}
```

保护异常类不发生异常

当一些软件组件遇到软件或硬件异常时，会抛出异常对象。但注意，异常对象自身不抛出异常。这有很多含义。在多线程或多处理环境中设计处理程序时需特别注意。若异常处理过于复杂，以至于潜在导致其他异常的产生，那么就需重新设计异常处理并尽可能做简化。如果异常处理代码能够产生异常，那么异常处理器制会没有必要地变得很复杂。因此，多数异常类中包含了空的 throw( )方法。

```
// Class declaration for exception class

class exception {
public:
   exception() throw() {}
   exception(const exception&) throw() {}
   exception& operator=(const exception&) throw() {return *this;}
   virtual ~exception() throw() {}
   virtual const char* what() const throw();
};
```

注意，throw( )的声明中参数为空。空的参数说明该方法不能够抛出异常。如果方法试图抛出异常，将会产生一个编译时错误消息。如果基类无法抛出异常，那么相应的方法在任何派生类中都将无法抛出异常。

2. 实现逻辑容错的简单策略

在最简单的形式下，实现模型为应用程序的PBS架构中的每个谓词创建一个C++谓词。此时，将为每个谓词创建一个谓词异常类(可能从 logic_error 类派生)。如果因为一些未知的和不可控的原因，关于某个谓词的假设、断言或命题是矛盾的，那么就抛出该谓词的异常类。在多数情况下，这意味着应用程序将体面地退出。这时应当退出，因为如果 PBS 中

的一个或多个谓词是矛盾的，那么应用程序不具有相同的含义，已经违背了并发基本结构的完整性。谓词异常类担当着基于逻辑的不变式的角色，用它来保证 PBS 与应用程序的一致性。显然，谓词异常处理程序知道当它的异常被抛出时应当做些什么。

例如，如果 guess_it 程序确定没有足够的 agent 来生成足够的猜测以赢得游戏，是否应当抛出 valid_code 异常？或者缺乏足够的 agent 是一个普通的软件错误，因此应当由错误处理而不是异常处理来解决？答案是缺少足够的 agent 在本例中不是普通的软件错误，因为软件的基本操作是获取足够的 agent 去及时地猜测编码。若无法获取足够的 agent，任务将失败。单元测试过程应当暴露它，谓词异常处理程序用来辅助单元测试的质量。所以，逻辑容错在本例中完成两个重要的任务。

(1) 使应用程序体面地退出(或到其他可接受的状态)。对于多 agent 和黑板架构，这包括：

- 将 agent 和知识源解散或退出
- 归还黑板资源和通信线路
- 释放对任何共享数据、同步或通信组件的占用

(2) 阻止应用程序在不合理的状态中继续运行，从而防止应用程序犯更多的错误。

特别地，逻辑容错的第二个任务是帮助构建和加强基于声明式的架构。

3. 测试与逻辑容错

典型情况下，异常处理器制被用来防止程序简单地崩溃。C++异常处理器制支持终止模型(termination model)。终止模型中提倡的是体面地退出。但是，经常会在某些灾难性事件发生时终止应用程序。尽管在没有严格按照 PBS 时，多线程或多处理应用程序有可能继续运行，但是您应将任何对 PBS 的背离视为灾难性的，因为应用程序的逻辑结论(含义)是其谓词、命题、公理和陈述的总和。这些谓词构成了应用程序的逻辑参数。如果断言或谓词之一的结果为 false，那么应用程序在该位置是不合理的。您希望在测试阶段发现所有不合理的行为，并使用逻辑容错语义来帮助实现这一目的。C++谓词和谓词异常的联合使用是达到声明式并行编程方法的重要部分。尽管目的是在测试阶段找出所有的缺陷，但这并不总是可以实现的。所以，在 C++异常处理的典型用法基础上增加了逻辑容错。逻辑容错的目的是不允许程序有任何后果是在 PBS 给出的后果之外的。

10.5.4　谓词异常和可能世界

您可能会回想起在 guess_it 游戏中，只有当特定时间间隔内达到猜测数目时，持有编码的 agent 才会改变编码。在这个场景中，尝试猜测编码的 agent 面临两个可能世界：一个世界中的编码没有改变，另一个世界中的编码发生改变。PBS 对这些情形有着如下的分解。

分解 5：N 个 agent 足以在 2 分钟内从四百万个编码的样本中找到正确的编码。

分解 6：如果编码每 15 秒改变一次，需要 $4N$ 个代理 2 分钟内从四百万个编码的样本中找到正确的编码。

ValidCode( )谓词未通过单元测试，因为它没有为分解 6 做准备。需要注意的是，如果 agent 非常幸运地在 15 秒之内猜对了代码，那么程序将会显得处于好的工作状态。但是当您依据 PBS 对应用程序进行评估时，必须为 PBS 中的每个谓词提供一个测试案例。每个谓词引入了一个或多个可能世界。可能世界的概念是从逻辑领域拿来的。可能世界被用来表达什么必然为真、可能为真或偶尔为真。可能世界模型背后的直观想法是在事件的真实(或当前)状态之外，还有其他多种可能的事件状态，或“世界”[Fagin et al.，1995]。可能世界是 agent 认为可能的结果，在本例中，就是编码被改变(世界 1)或编码未被改变(世界 2)。应用程序的 PBS 清楚地定义了 agent 或知识源将要在什么样的可能世界中运行。对于 agent 的每个可能的世界，都有可接受的代码集合供 agent 或知识源执行。另一方面，对于不可能的世界，不存在可接受的代码供 agent 执行。不可能世界产生谓词异常。

PBS 中的谓词引入了应用程序或 agent 所处的可能世界是什么。谓词异常类代表了进入到不可能世界的场景。例如，对于编码每 15 秒改变一次，但没有新的 agent 加入以适应这种改变，这在 PBS 中是一个不可能世界。正是这个问题导致 ValidCode( )谓词失败。您可以在单元测试中运用模型检测来发现并行程序中这类软件缺陷。

10.5.5 什么是模型检测

尽管关于模型检测的讨论超出了本书的范围，但是我们在这里对它做一下介绍，这样一旦您准备好了处理大且复杂的并行程序或大规模多线程应用程序，您能够对使用的工具有一些认识。本书给出了对于多核和多线程编程的一些更具挑战性的问题。但本书仅仅做了简要介绍。我们介绍了向着并行化的逻辑模型发展的声明式并行编程技术。逻辑模型将最终帮助开发者应对大规模并行多核计算机。模型检测技术可用来自动化具有声明式架构的多线程或并行程序的检测。模型检测是用来验证有限状态并发系统的技术[Huth，Ryan，2004]。模型检测用来判断给出的模型是否是期望的模型。在本章中作为例子的 PBS 方法，给出了应用程序的逻辑模型，该模型可以被形式化并用于模型检测工具。您可能对了解 Kripke 架构被用于可能世界和模型检测的形式化表示感兴趣[Meyer，Van der Hoek，2004]。

10.6 小结

测试软件的目的是为了确保软件做了您希望的事情，而且您希望的事情正是软件所做的。应用程序往往被要求为所需功能的列表，而且无论软件的这些需求列表是如何生成的，测试过程必须确保软件满足需求，而且这些需求满足用户的期望。

对于涉及并行计算机或多处理器的许多情况，用户的期望包括性能提高或者是某种级别的高性能吞吐量，而且当为了达到用户的期望而加入多线程或多处理时，您遇到的软件错误的种类会增加。当软件不按照规范执行时，软件处于错误之中，即使是由于系统执行得过慢而违反规范时也是如此。

本章讨论了使用异常处理为声明式架构提供逻辑容错。也就是说，如果应用程序由于

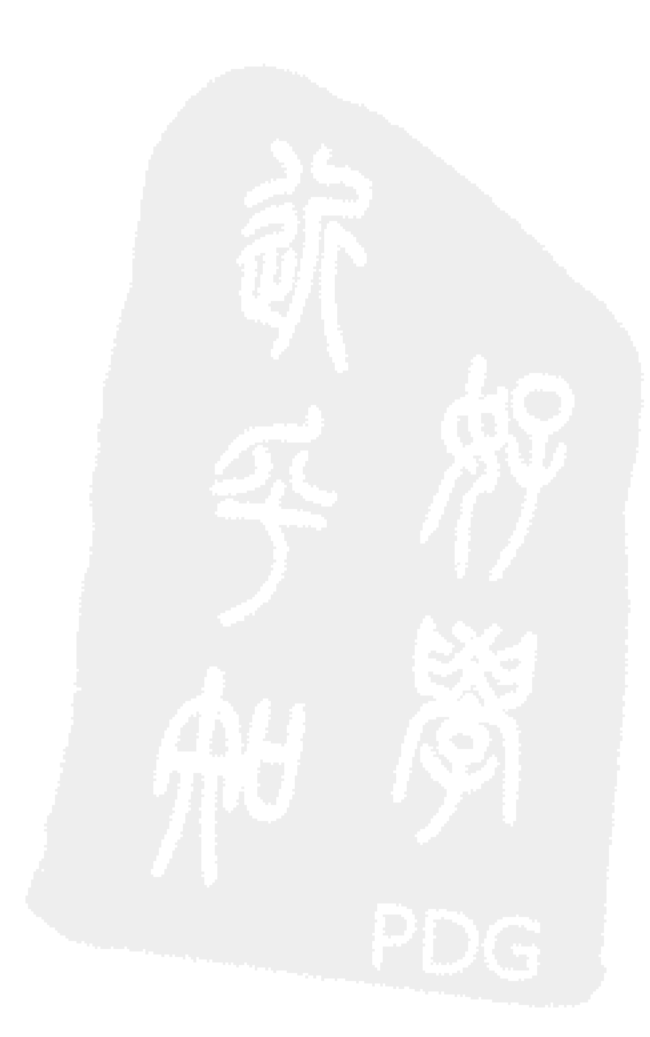
未知的和不可控的原因违反了来自 PBS 的陈述、断言、规则、谓词或约束，您希望抛出异常并体面地退出，因为一旦谓词被违反，则应用程序的正确性、可靠性和意义都会受到危害。软件中的容错之旅常常是从达到以下认识开始的：

- 再多的异常处理也无法拯救有缺陷或不适当的软件架构
- 软件容错同软件架构的质量直接相关
- 异常处理架构不能取代各个测试阶段

在努力达到并行编程的声明式解释过程中，我们在尽力使您进一步地向着逻辑模型发展。最终，您希望将非逻辑模型或不合理的程序行为都视为异常。因此，异常处理策略需要从 PADL 的第 5 层和 PBS 中实现，并成为软件架构的基本部分。用户定义的 C++谓词形成了应用程序的逻辑参数。如果断言或谓词之一证明为 false，那么应用程序在该位置是不合理的。应用程序的 PBS 清晰地定义了 agent 或知识源运行时所在的可能世界。对于 agent 的每种可能世界，都有可接受的代码供 agent 或知识源来执行。

要求并行编程的软件会非常难以测试和调试，这就是为什么对于有着并发需求的软件，测试和调试成为关键挑战的原因。在理念上，本章为您提供了如何在并行编程中解决某些测试和调试的挑战的基础，正像本书为您对多处理器和多线程架构进行编程打下日常基础，恰当的应用程序设计和开发成为了人们当前关注的主流。我们祝愿您在今后的多核程序项目上取得成功。

最后，我们重复本章开头所提到的观点：作为软件开发者，我们开发应用于医药、制造业、国家安全、交通、金融、教育、科学研究、商业所有领域的应用程序，我们有着伦理和道德上的责任来开发安全、正确、可靠、容错的软件。否则就是玩忽职守。



并发设计使用的 UML

本附录提供了本书中所使用的 UML 图的快速参考。UML(Unified Modeling Language，统一建模语言)是用来设计、可视化、建模和记录软件系统中的制品(artifact)的图形符号。它是面向对象系统进行沟通和建模的事实上的标准。建模语言从不同的角度以及不同的关注焦点，使用符号来表示软件系统中的制品。尽管本书中还使用了其他图形符号和制品，但本附录提供了一种快速方法使得读者能够在对软件系统编写文档时，熟悉基本的 UML 符号。

下面的网站是 UML 文档方面的一些在线资源以及 PDF 格式的快速参考：

- http://www.uml.org
- http://www.acmesoffware.com/acme/default.asp
- http://www.oio.de/uml-1-4-reference.htm
- http://www.quantum-leaps.com/resources/UML_Reference.pdf

A.1 类图和对象图

类图和对象图是在对面向对象系统进行建模时最常使用的图。类图可用于表示系统中的每种类型的类，包括：

- 模板类
- 接口类

类图可以包括类的细节(例如，属性和服务)。类图和对象图可以显示数据类型、变量值、函数的返回类型。对象图可以显示对象名称。两种类型的图都可以描述系统中使用的类或对象的数目，以及类之间和对象之间的关系。

图 A-1 示范了表示类和对象的各种方式，包括活动类和活动对象。类图可以根据需要显示细节，包括属性、服务、初始值、返回类型和访问权限。活动类或活动对象使用较粗的线。

图 A-2 显示了类和对象的多个实例。可以使用多重性符号来图形化显示多个实例。

图 A-3 示例了表示绑定或未绑定模板类或参数化类的方式。

图 A-4 显示了表示接口类的方法。接口类可以通过使用棒棒糖符号来表示，或者表示为显示<<interface>>构造型的常规类。也可以描述接口类同类的实现之间的关系。

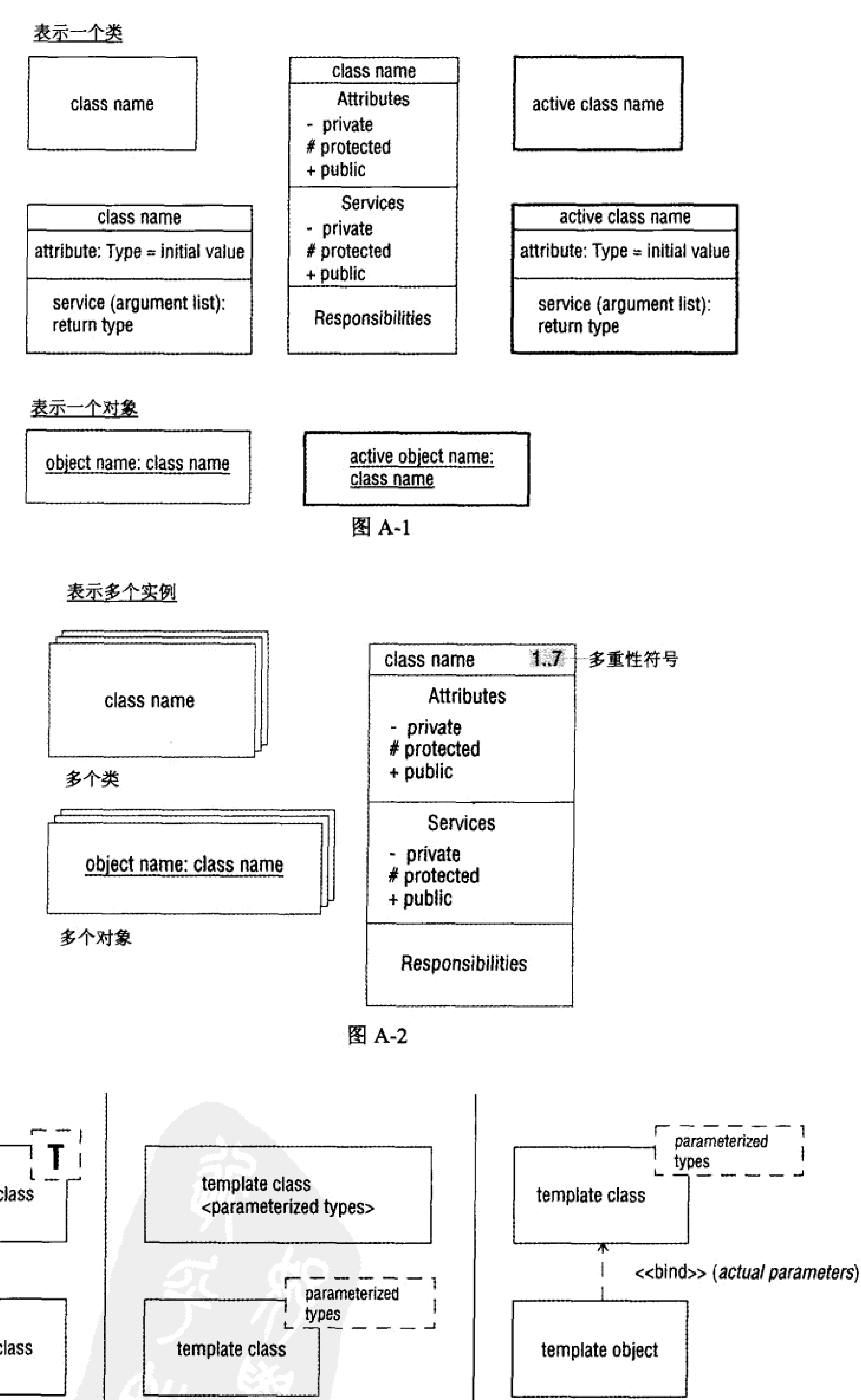


图 A-1

图 A-2

图 A-3

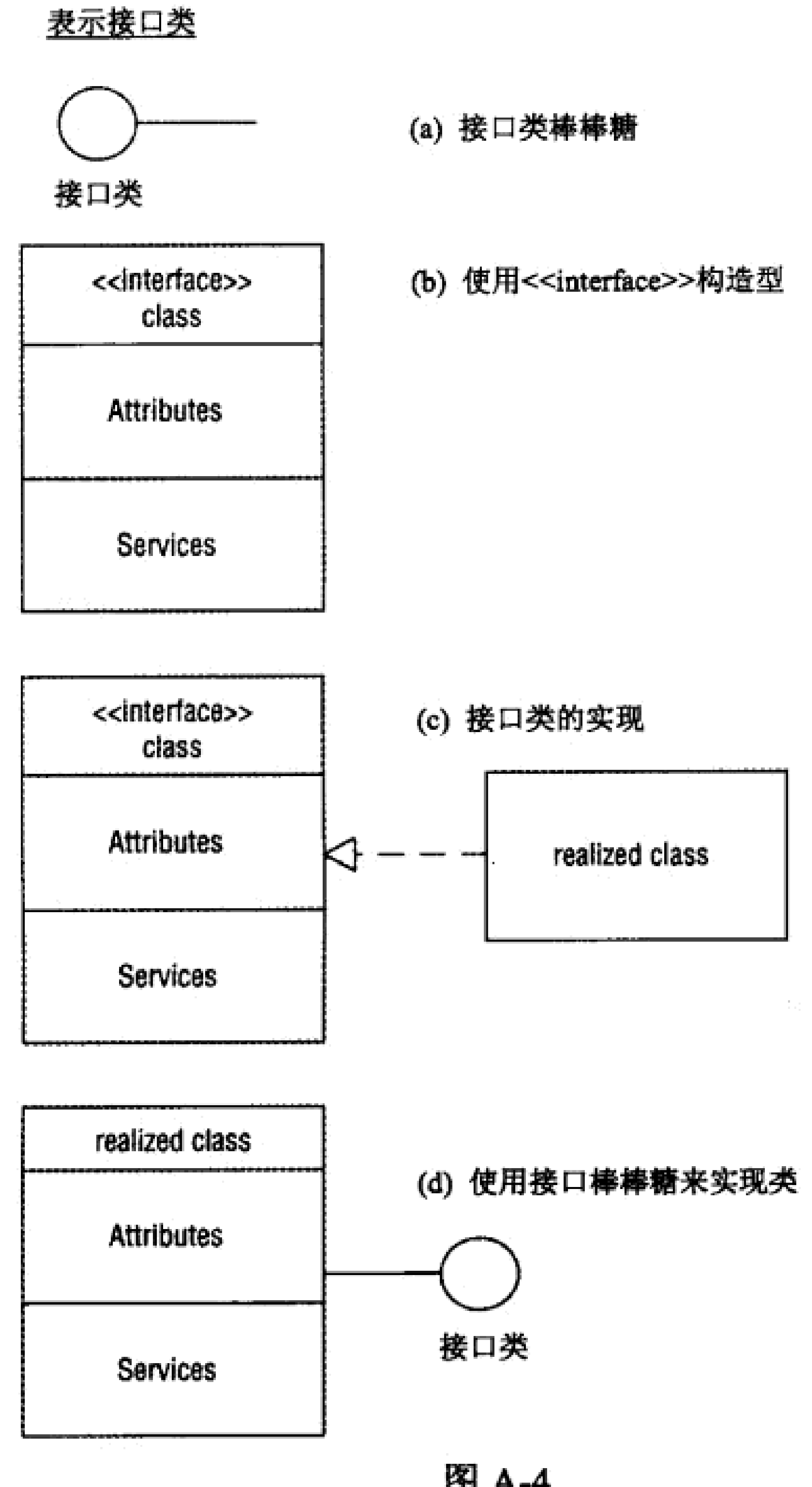


图 A-4

图 A-5 显示了表示单继承和多重继承的方式。当关系中涉及多个类时，可以使用两种到达目标的风格：共享的和分离的。

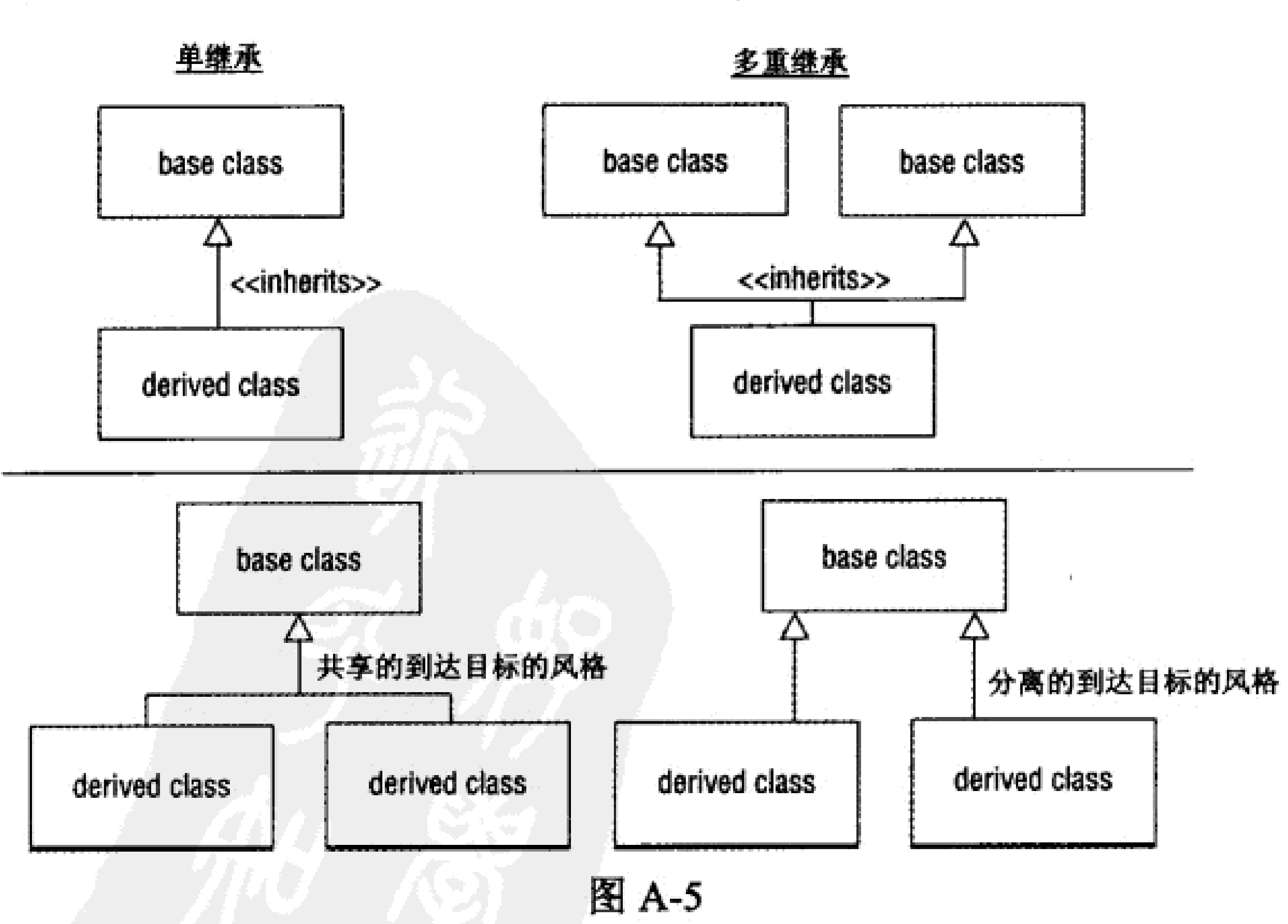


图 A-5

- 使用共享的到达目标的风格时，将多个类结合成一个指向目标类的继承符号。
- 使用分离的到达目标的风格时，每个类都有着自己的继承符号。

图 A-6 示范了可以在类图中描述的多种关系的示例。可以使用多重性符号来显示在类和对象间的实例的数目。

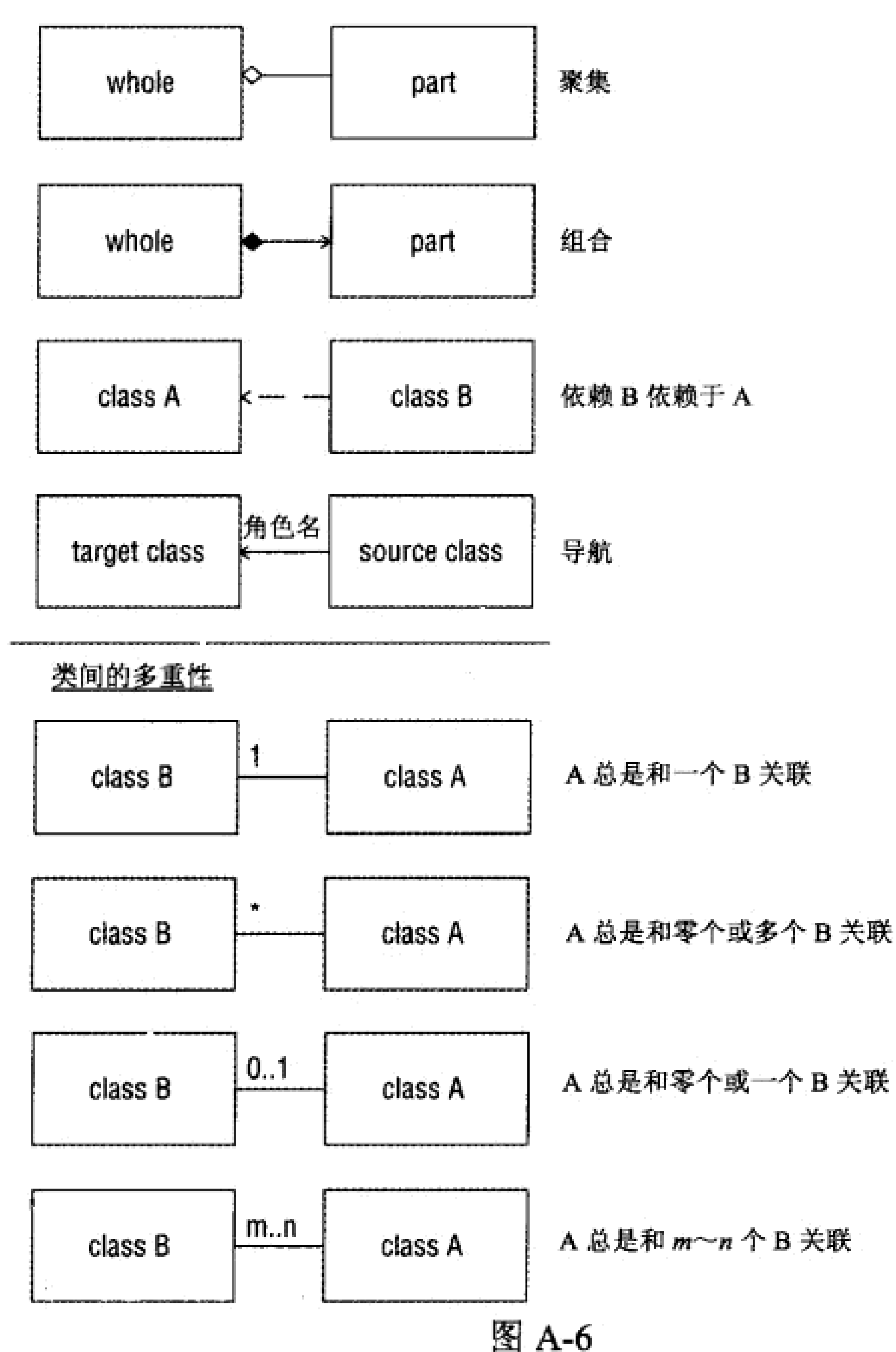


图 A-6

A.2 交互图

交互图显示对象之间的交互。它们由一组对象、对象间的关系和对象间交换的消息组成。交互图包括协作图、顺序图和活动图。

A.2.1 协作图

协作图用来显示一组对象共同工作以完成某些任务。系统中的协作是一组对象之间暂时性的合作。协作图可以描述协作的组织或协作的结构。这涉及显示集合中所有的对象、对象的连接、对象之间发送和接收的消息。

图 A-7 是显示系统中协作的组织以及协作中对象的结构关系的协作图。

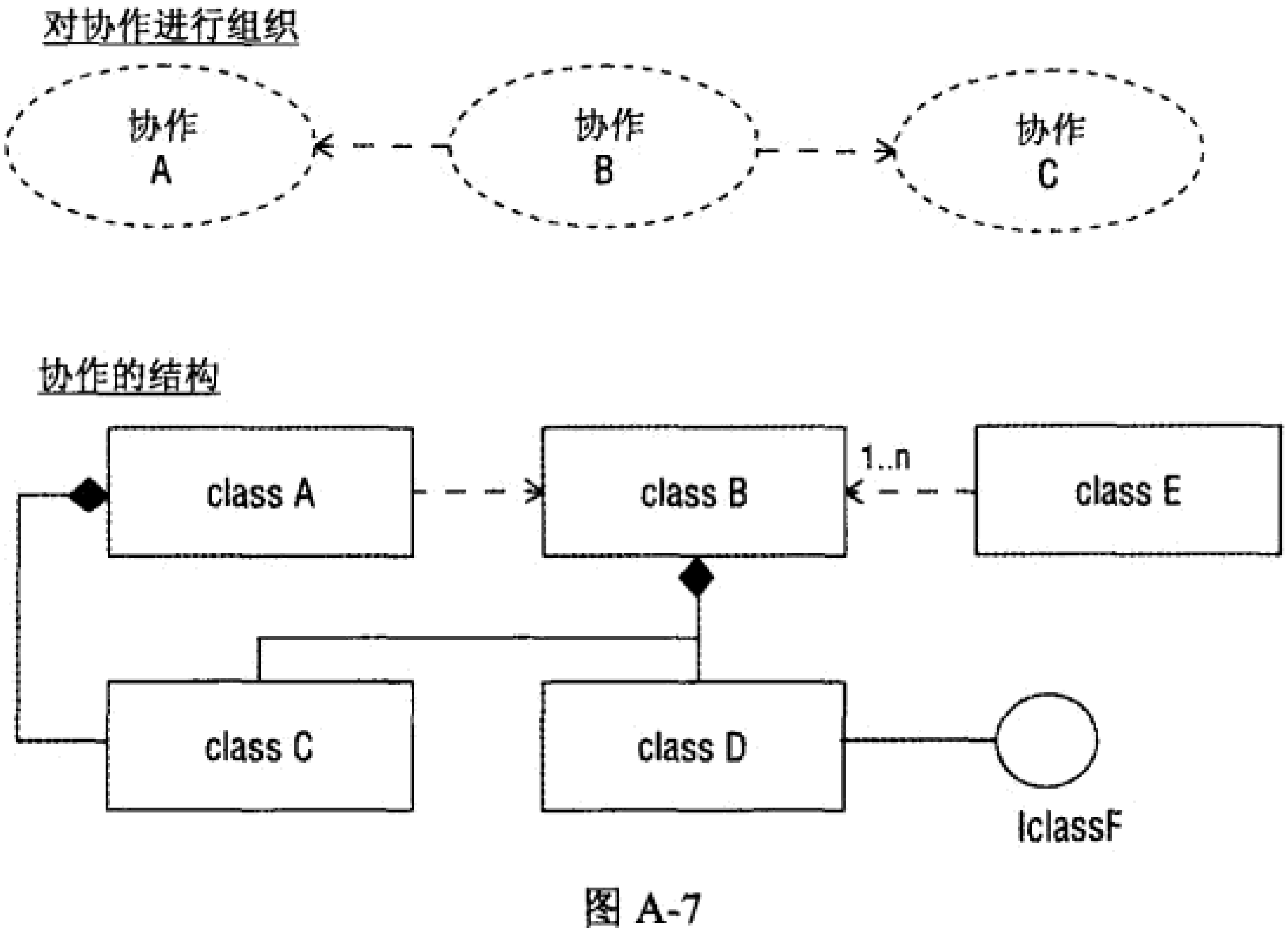


图 A-7

A.2.2 顺序图

顺序图用来强调系统中的对象发送和接收的消息的时间顺序。

在图 A-8 中，使用顺序图来强调对象间传递的消息的时间顺序。活动对象放置在图的 x 轴的上方。对象间传递的消息放置在图的 y 轴。图可以描述同步和异步的消息传递。消息的时间顺序是通过沿着 y 轴自上而下阅读消息来说明的。

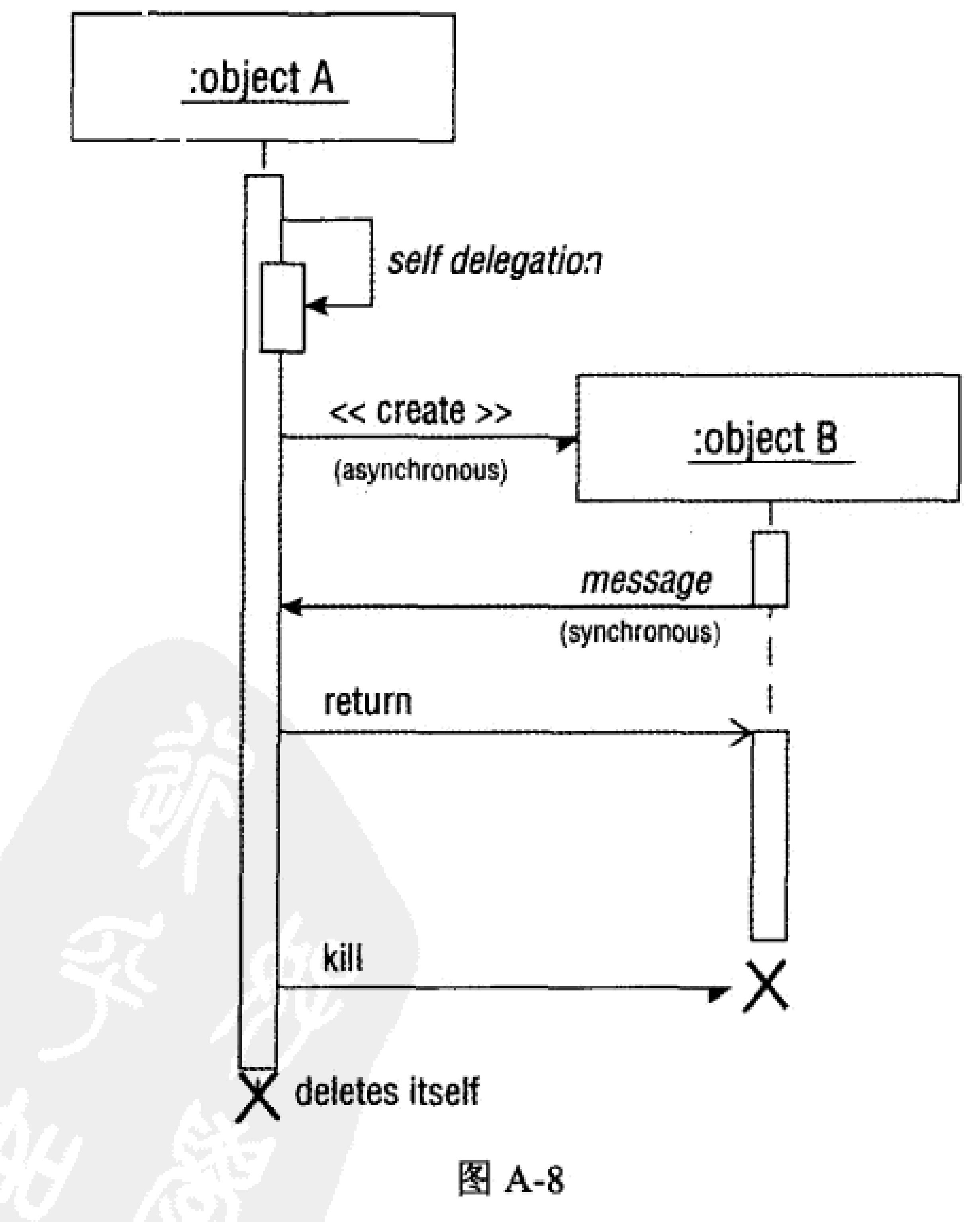


图 A-8

A.2.3 活动图

活动图显示从一个活动到另一个活动的控制流。活动是由对象执行的动作。动作包括处理输入/输出、创建或销毁对象和执行计算。活动图类似于流程图。

在图 A-9 中，活动图显示了当从一个对象的控制焦点移动到另一个对象的控制焦点时对象的动作。该图描述了分支成多个控制流(并发)以及控制流通过同步条结合。泳道(swimlane)用来显示哪个对象在执行该动作。转换可能会跨泳道。同步条也可能跨泳道，表示不同对象中的多个控制流并发地执行动作。

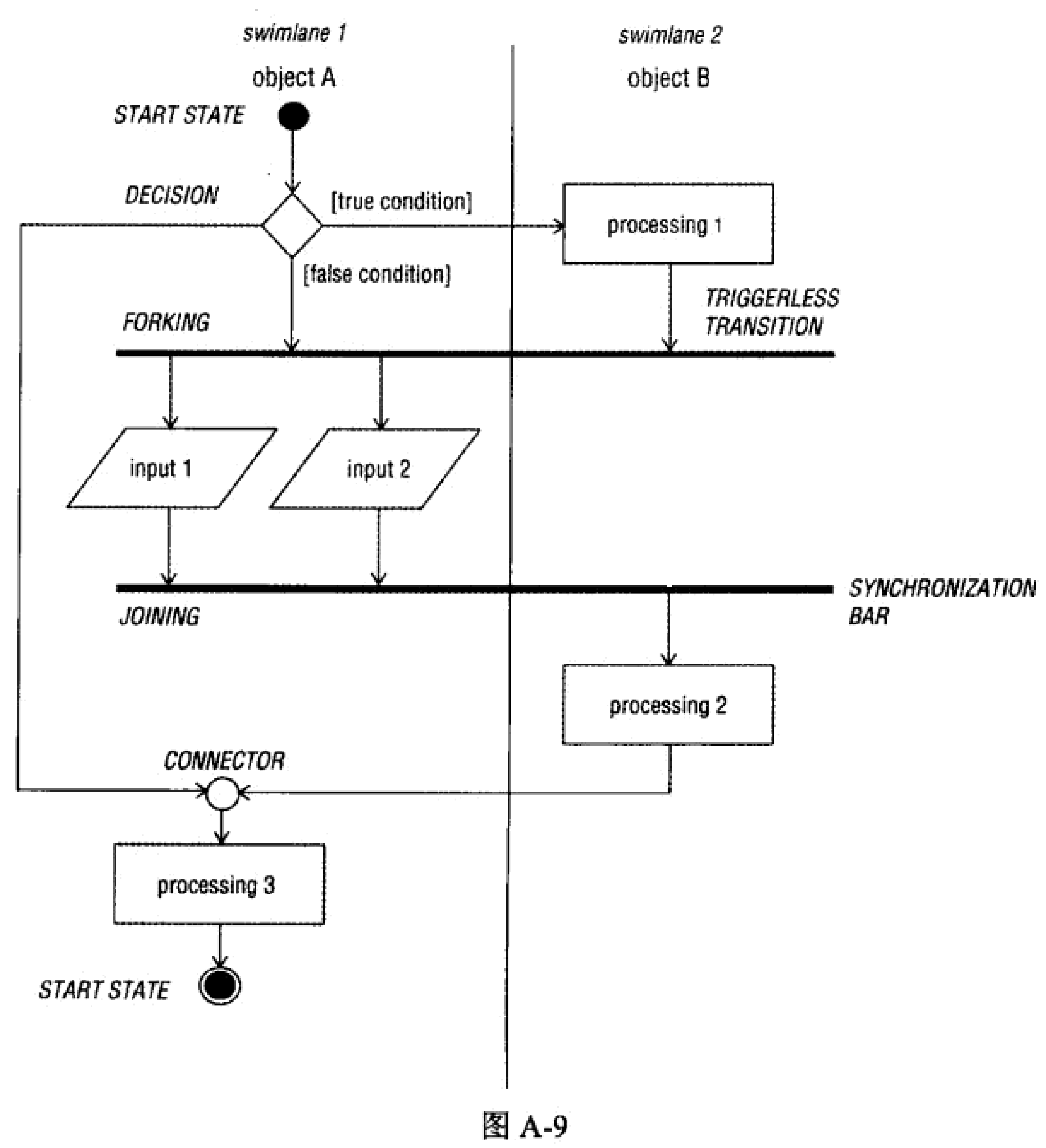


图 A-9

A.3 状态图

状态图用来强调对象的状态以及到这些状态的转换。状态是对象在其生命期的某个时刻所处的情形。对象可以在其生命期中转换到很多不同的状态。如果某些条件满足、某些动作被执行、或某些事件发生，则对象转换到某个状态中。

图 A-10 中的状态图显示了一个对象在其生命期中的状态及转换。状态图有着初始状态和最终状态。状态由几部分组成。状态还可以是其他状态的组合甚至是另一个状态图。在一个实体中并行执行的子状态被称为并发子状态。

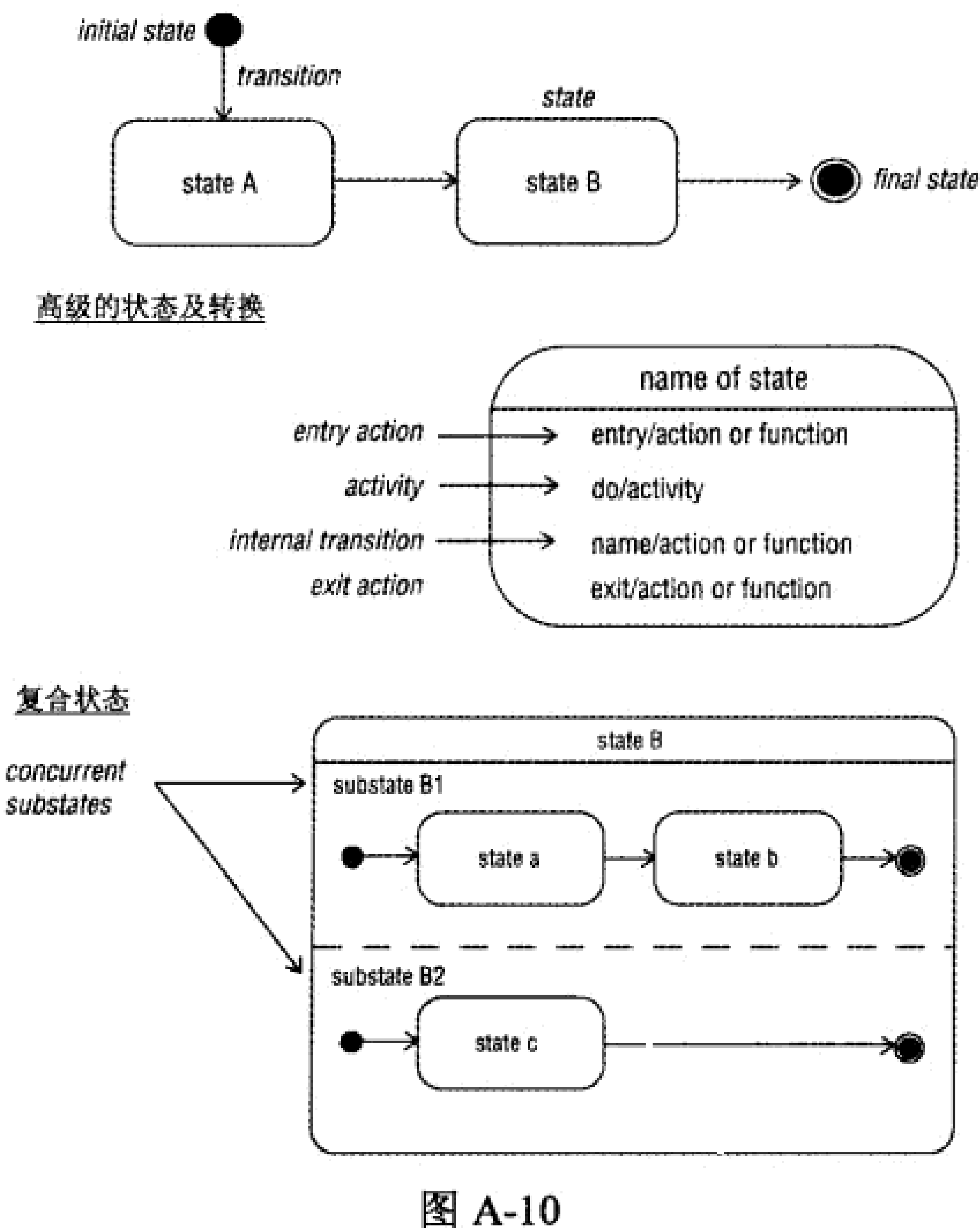


图 A-10

A.4　包图

包图用来将实体组织成组。

图 A-11 显示了如何使用包图来组织系统中的元素。可以使用<<system>>和<<subsystem>>构造型。如果包中包含其他实体，则左部的短小突出部可以保存包的名字。

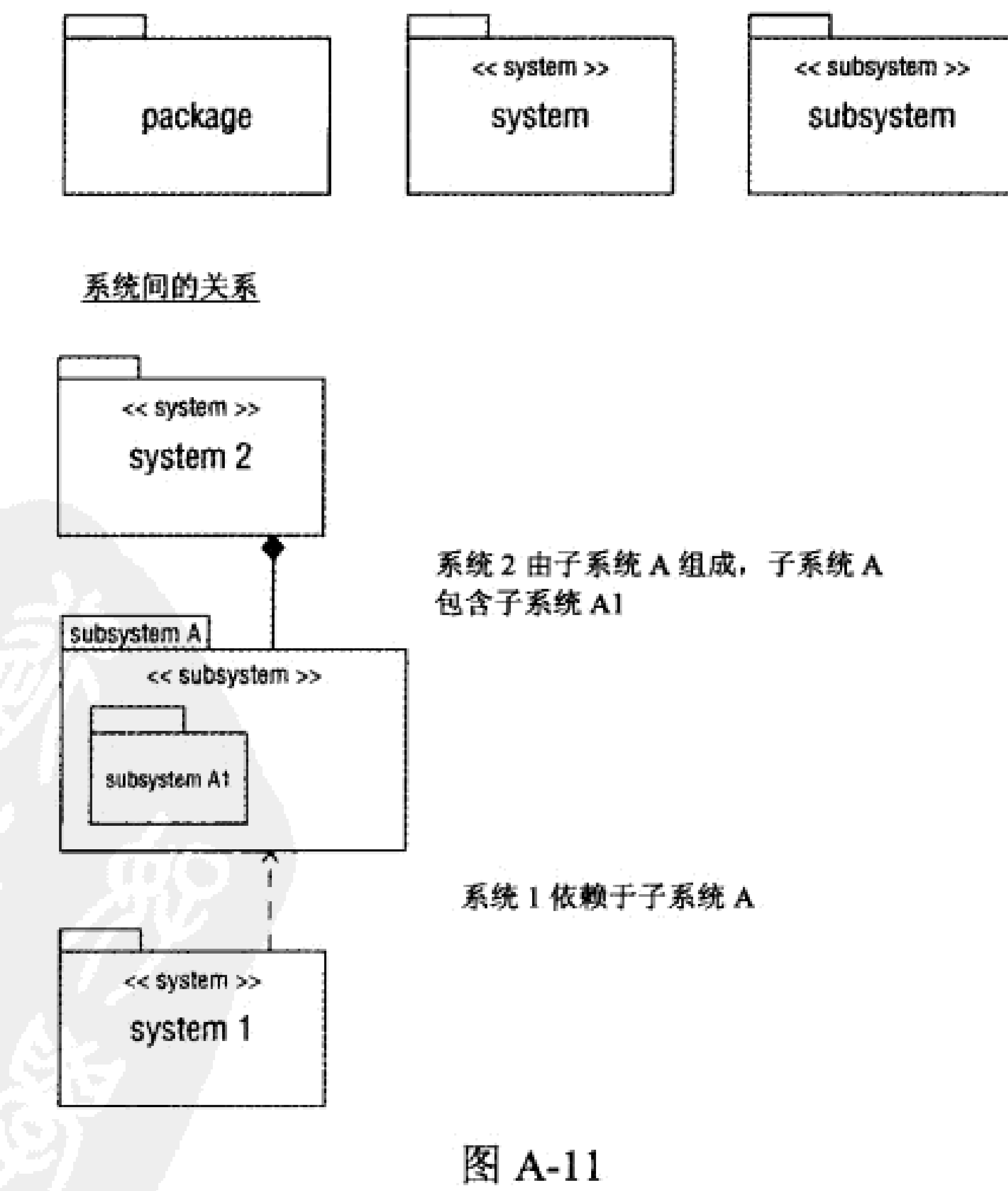


图 A-11

并发模型

本附录提供本书中所使用的并发模型的快速参考。并发模型决定了并发任务如何将工作委派给任务以及如何完成通信。模型还可以提供结构和方法，用来帮助您决定访问策略。这里，我们编译了并发模型，并为每个模型提供了简短的定义和图示。

B.1 进程间通信和线程间通信

进程间的通信(IPC)是由存在于两个进程地址空间外部的机制来执行的。

线程间的通信(ITC)发生在通信的线程所在的进程的地址空间中。

在图 B-1 中，可以看到 IPC 位于两个进程的地址空间的外部。IPC 允许相关或不相关的进程之间进行通信和数据传递。ITC 位于进程的地址空间内部。ITC 机制允许相同进程的多个线程访问全局数据和变量、参数、文件句柄。

图 B-1 显示了 IPC 和 ITC 的示例。

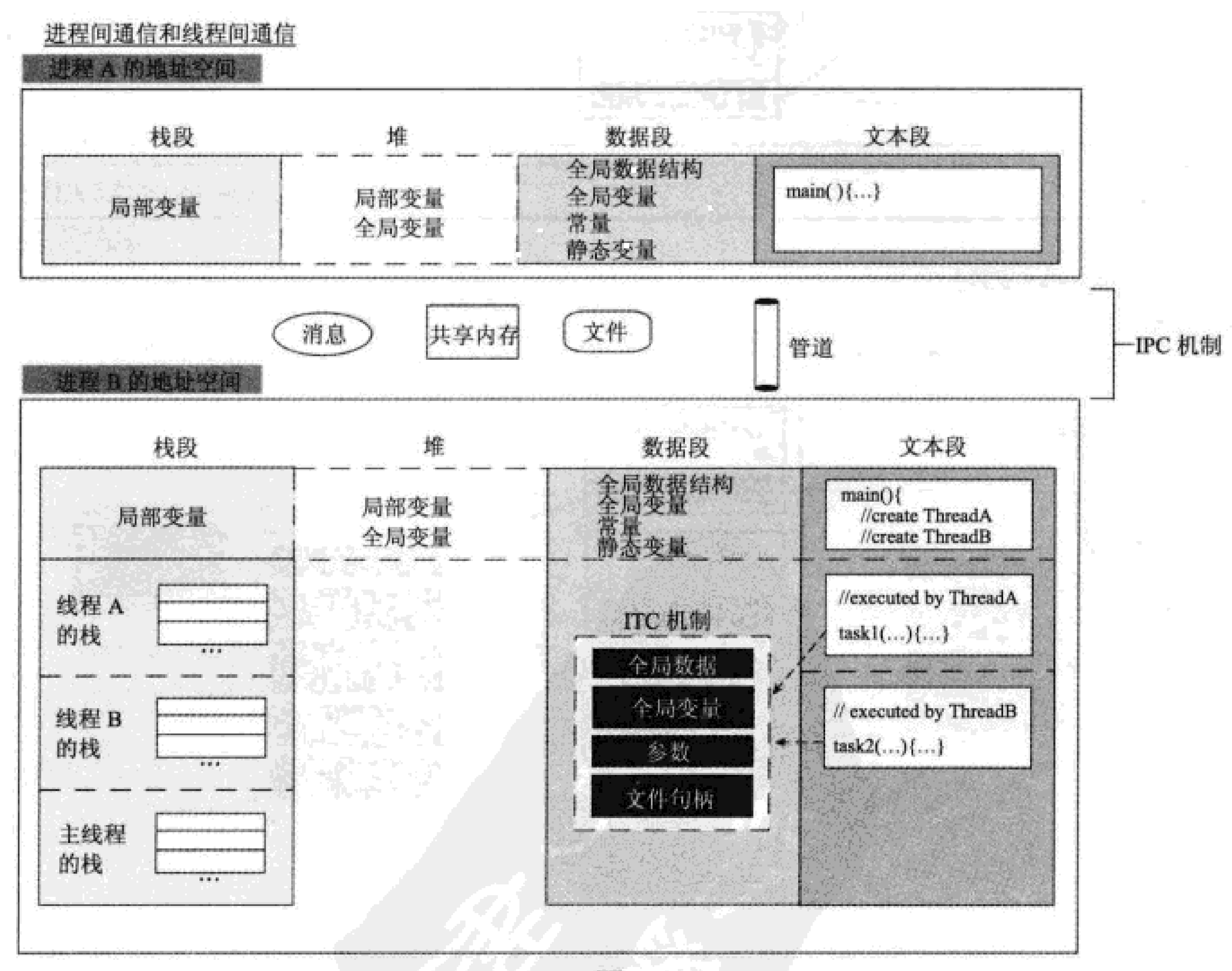


图 B-1

B.2 使用线程的 boss/worker 方法 1

boss 线程为每个任务创建一个新的线程。然后这些线程可以同时执行它们的任务。线程是活动的。线程间通信(ITC)用来在线程间进行通信并同步对共享资源、数据、对象的访问。图 B-2 的顶部给出了一个示例。

B.3 使用进程的 boss/worker 方法 1

boss 进程为每个任务创建一个新的进程。然后这些进程可以同时执行它们的任务。进程是活动的。进程间通信(IPC)用来在进程间进行通信并同步对共享资源、数据、对象的访问。图 B-2 的底部给出了一个示例。

使用线程的 boss/worker 方法 1
进程 A
<<Boss Thread>>
TASK _1
TASK _2
TASK _3
...
TASK _N
<<pthread_create>>
<<Thread 1>>
do
TASK_1
<<Thread 2>>
do
TASK_2
<<Thread 3>>
do
TASK_3
...
<<Thread N>>
do
TASK_N
<<uses>>
ITC
全局数据
全局变量
参数
文件句柄
共享数据/对象

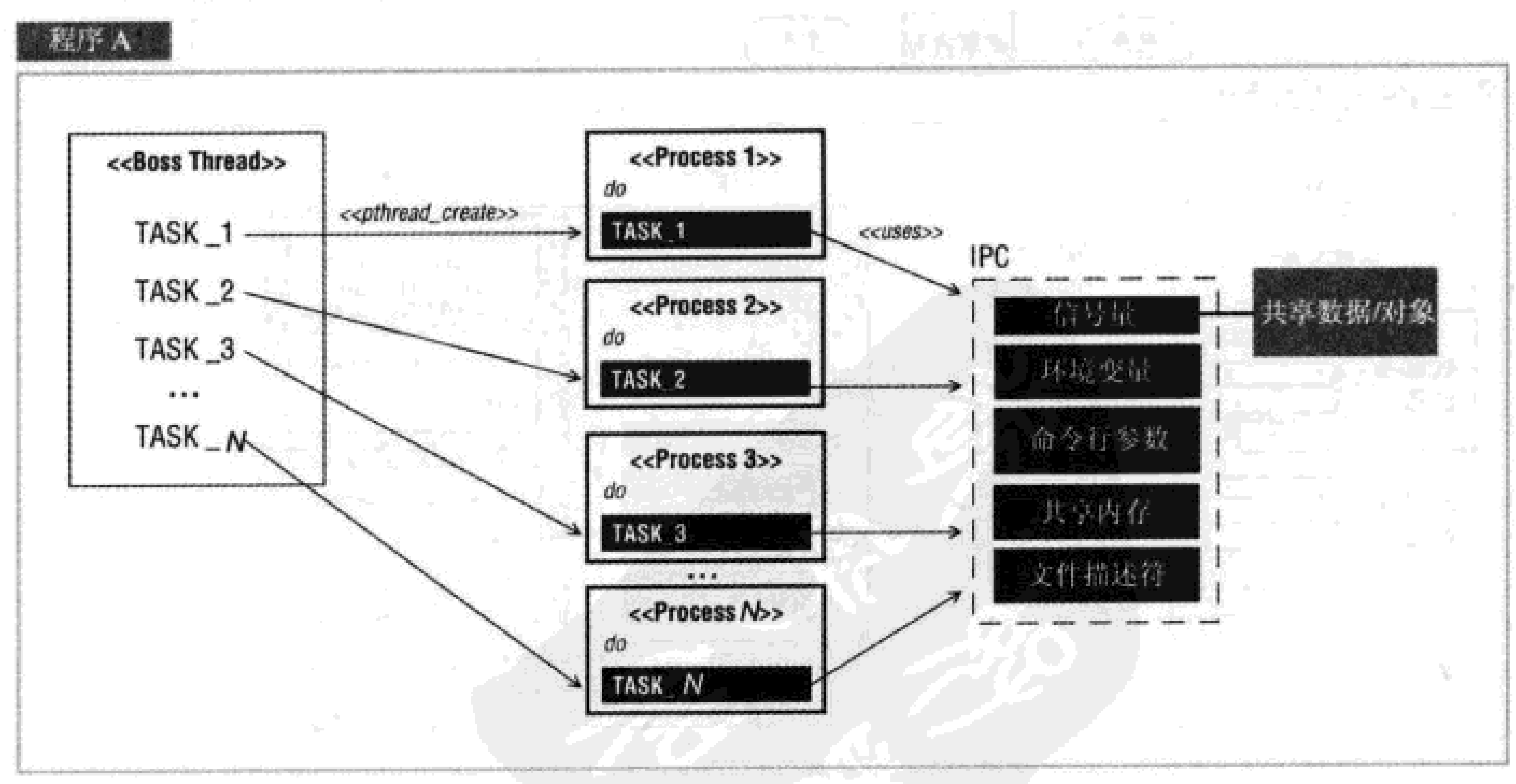


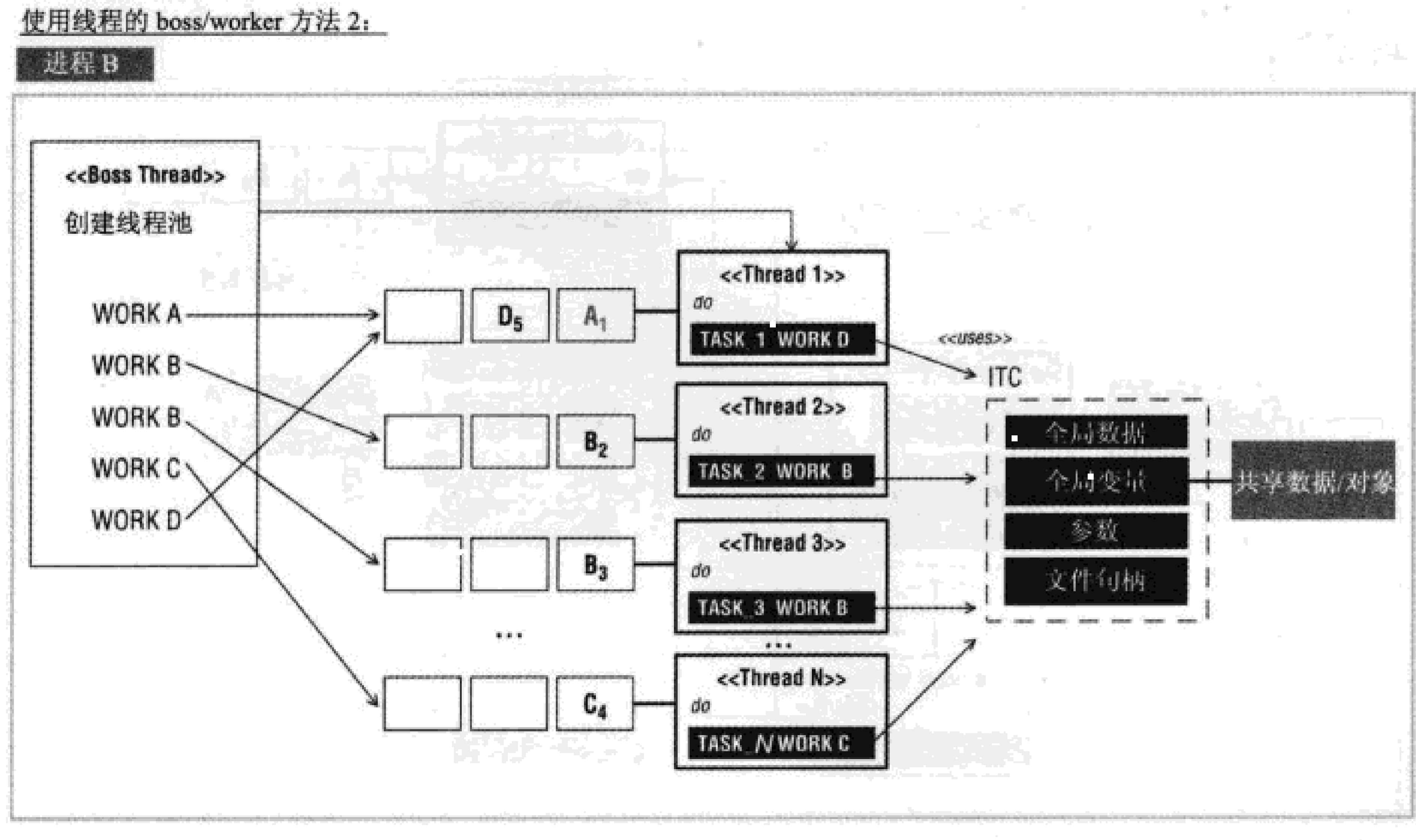
图 B-2

B.4 使用线程的 boss/worker 方法 2

boss 线程创建一个线程池，其中每个线程有着自身的数据队列。对于共享数据或对象，使用 ITC 同步访问。图 B-3 的顶部给出了一个示例。

B.5 使用线程的 boss/worker 方法 3

boss 线程创建一个线程池，这些线程处理来自共享数据队列的数据。每个线程将会处理一种类型的数据。对于共享数据或对象，使用 ITC 同步访问。图 B-3 的底部给出了一个示例。



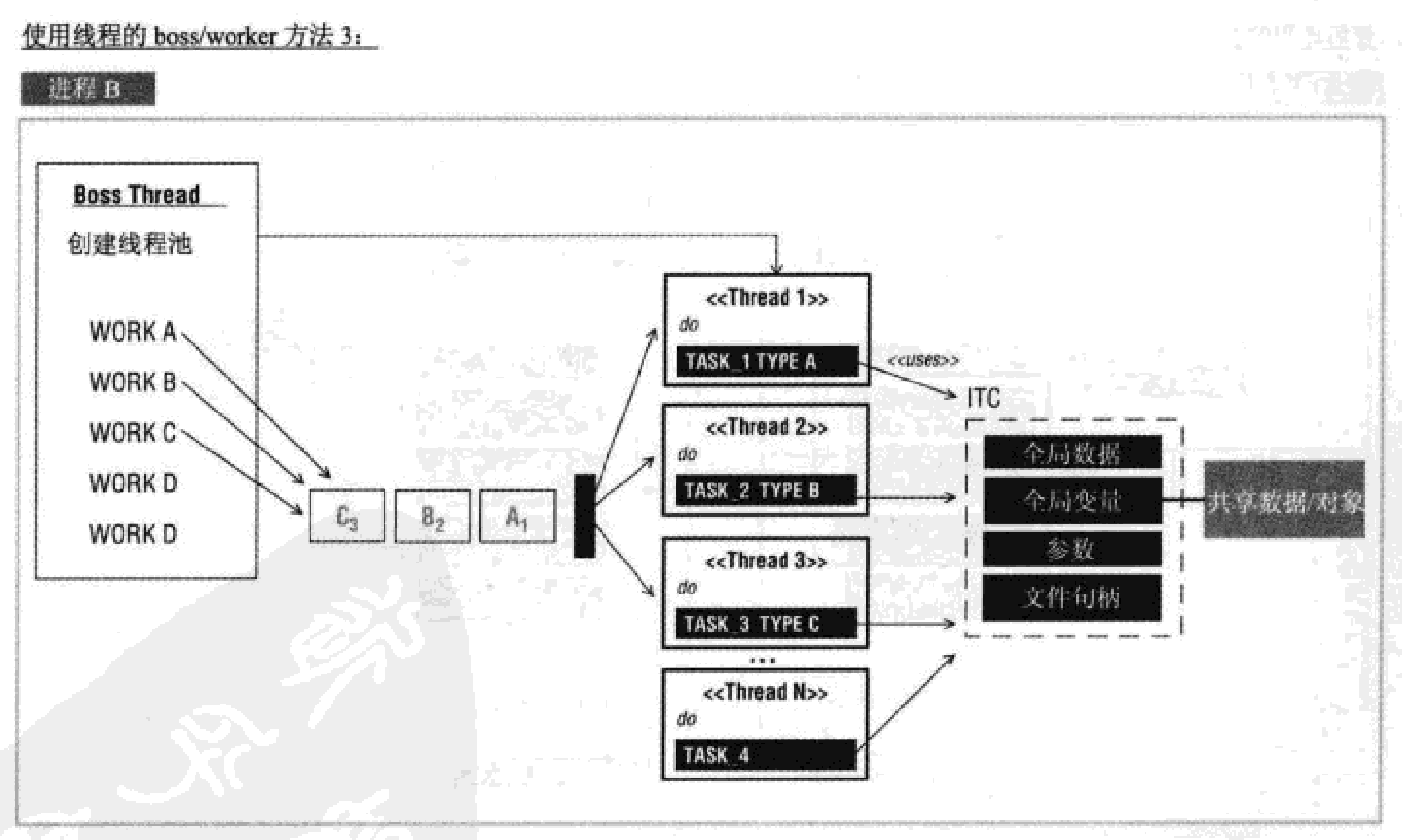


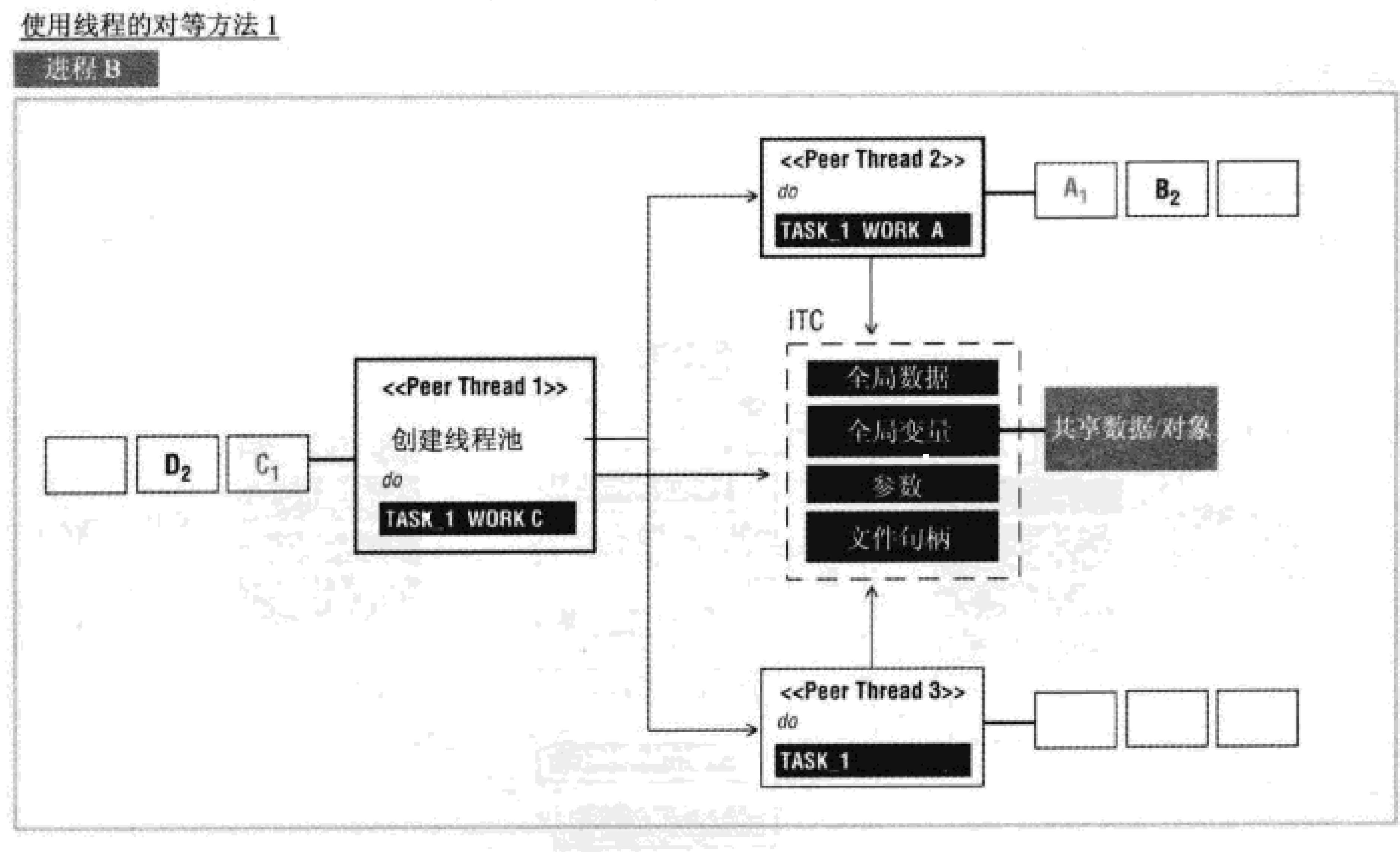
图 B-3

B.6 使用线程的对等方法 1

对等线程创建对等线程池，其中每个线程有着自己的数据队列。对于共享数据或对象，使用 ITC 来同步访问。图 B-4 的顶部给出了一个示例。

B.7 使用进程的对等方法 1

对等进程产生对等进程池，其中每个进程有着自己的数据队列。对于共享数据或对象，使用 IPC 来同步访问。图 B-4 的底部给出了一个示例。



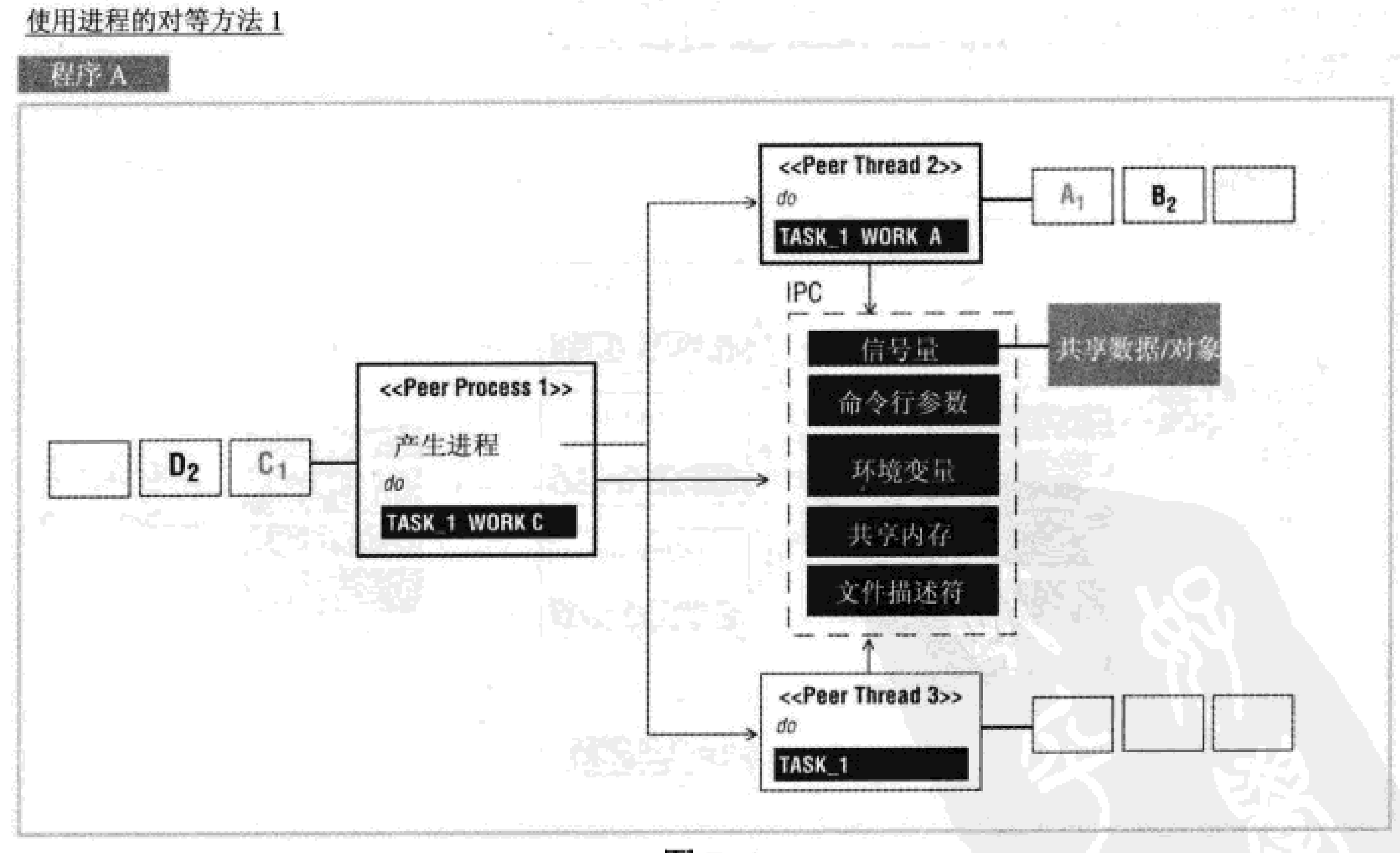


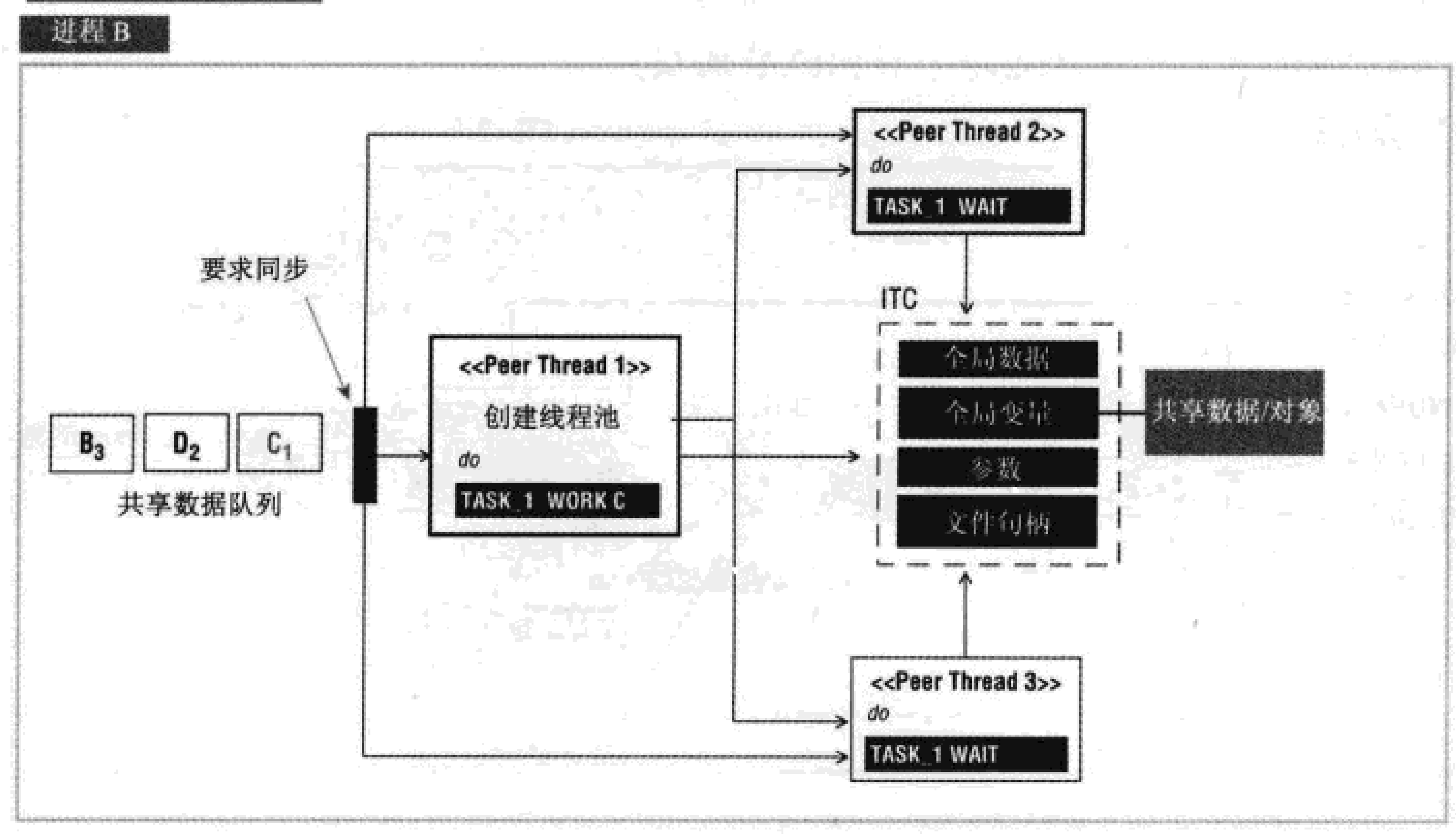
图 B-4

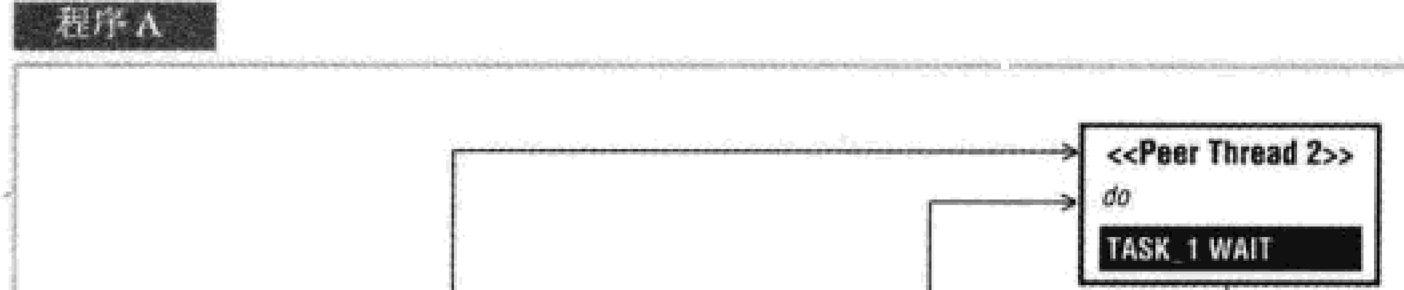
B.8 使用线程的对等方法 2

对等线程创建对等线程池，其中的线程共享一个数据队列。对于共享数据或对象，使用 ITC 来同步访问。图 B-5 的顶部给出了一个示例。

B.9 使用进程的对等方法 2

对等进程产生对等进程池，其中的进程共享一个数据队列。对于共享数据或对象，使用 IPC 来同步访问。图 B-5 的底部给出了一个示例。





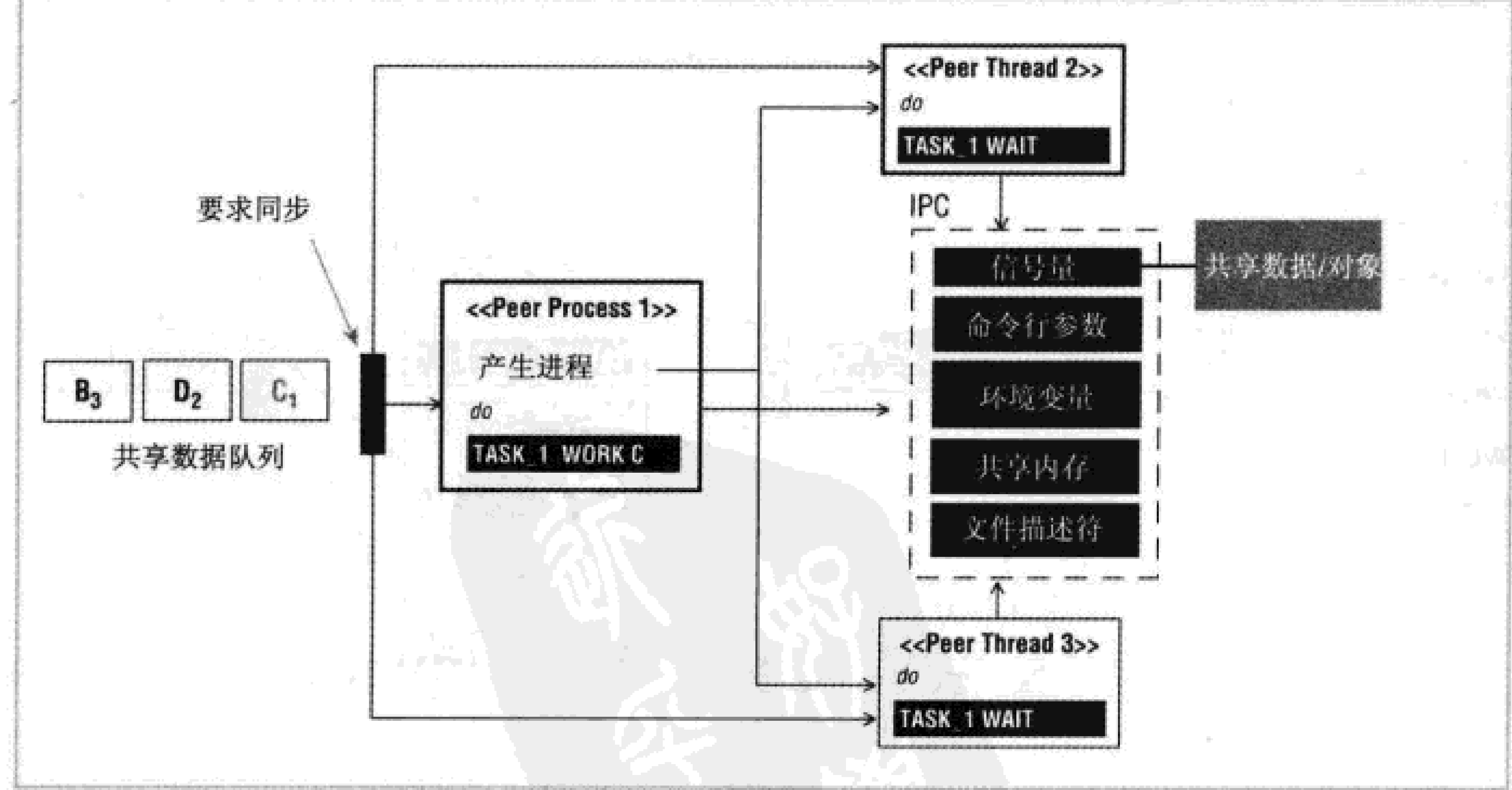


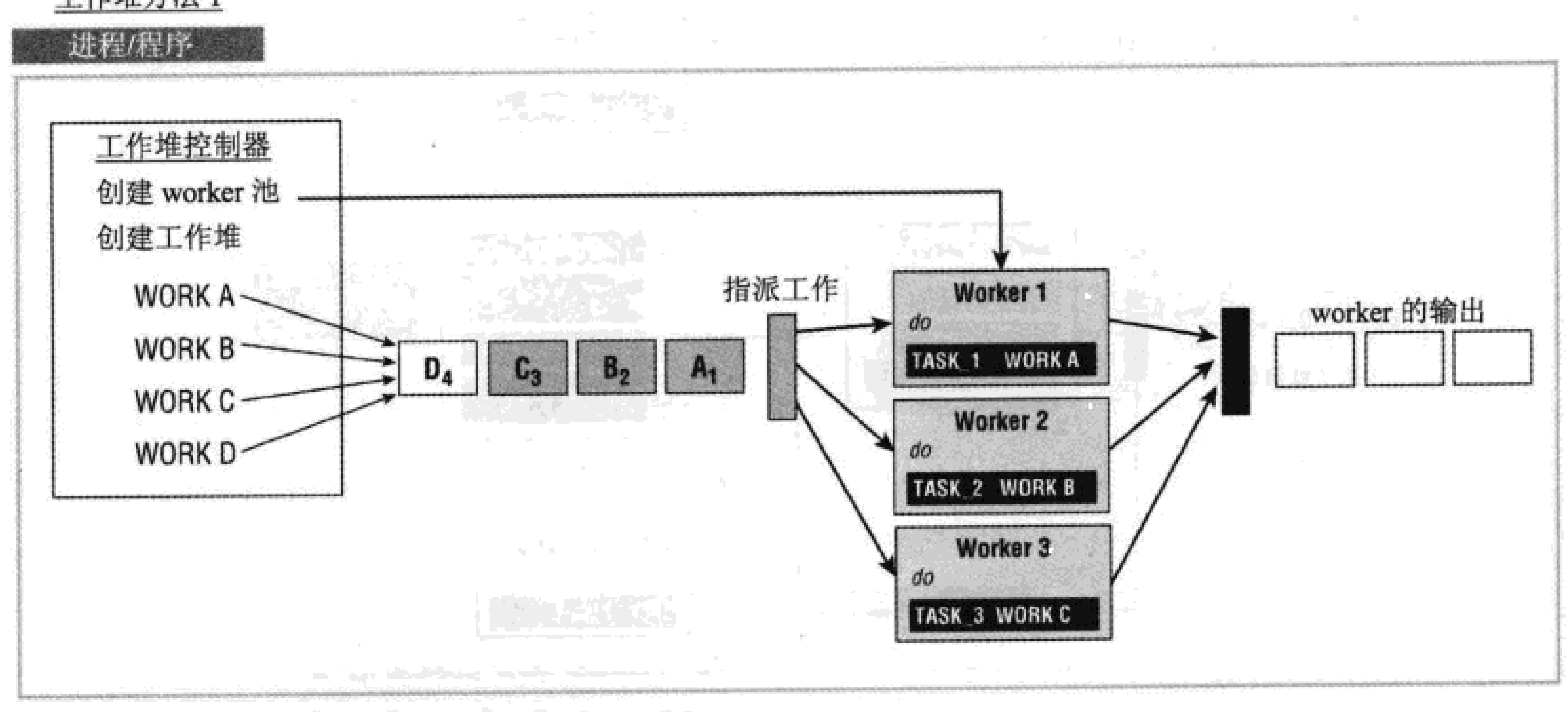
图 B-5

B.10 工作堆方法 1

工作堆(workpile)方法要求多个 worker 处理来自工作堆(共享的数据队列)的数据。控制及创建一个 worker 池并创建一个工作堆。控制器通过将工作指派给 worker 来管理工作堆。然后 worker 将它们的工作结果保存到要求同步的输出队列。图 B-6 的顶部给出了一个实例。

B.11 工作堆方法 2

对于工作堆方法 2，线程也可以产生可以放置到工作堆的工作。图 B-6 的底部给出了一个实例。



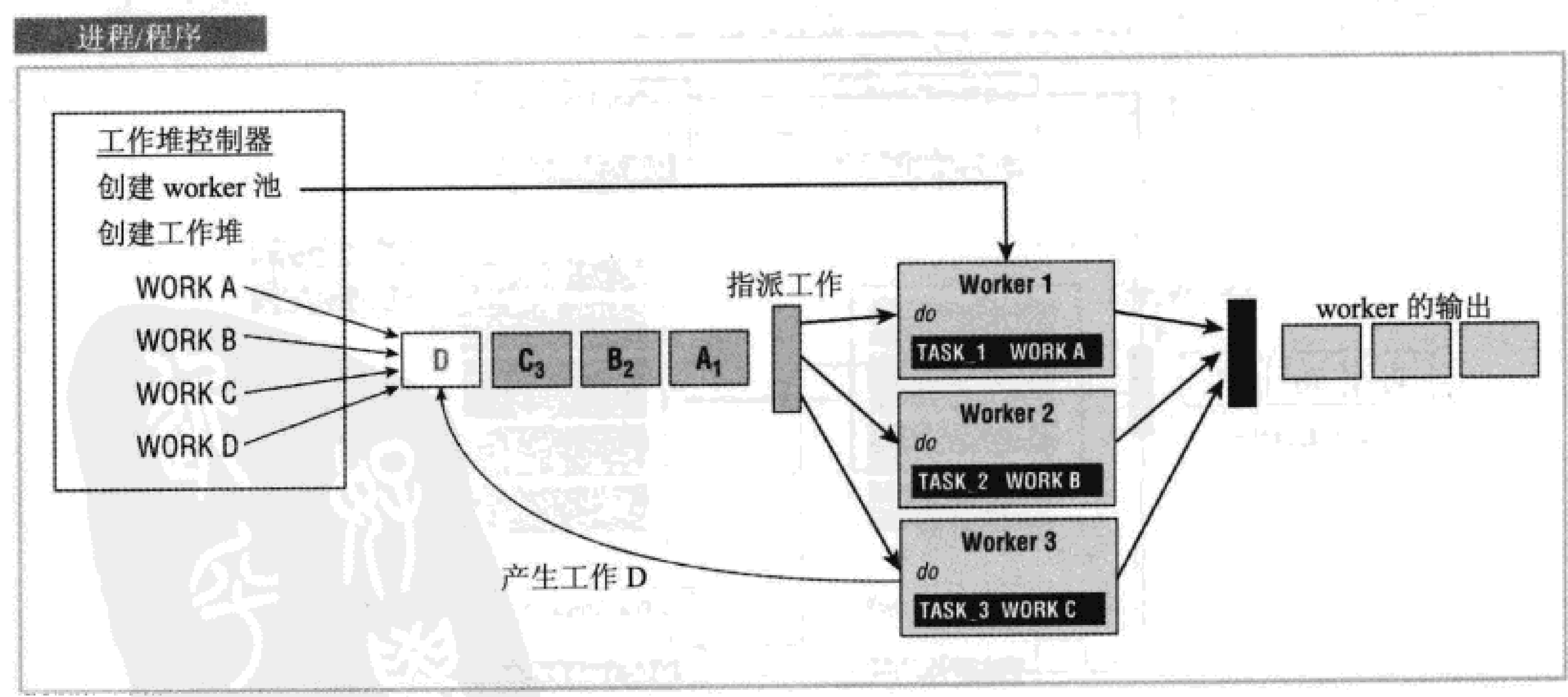


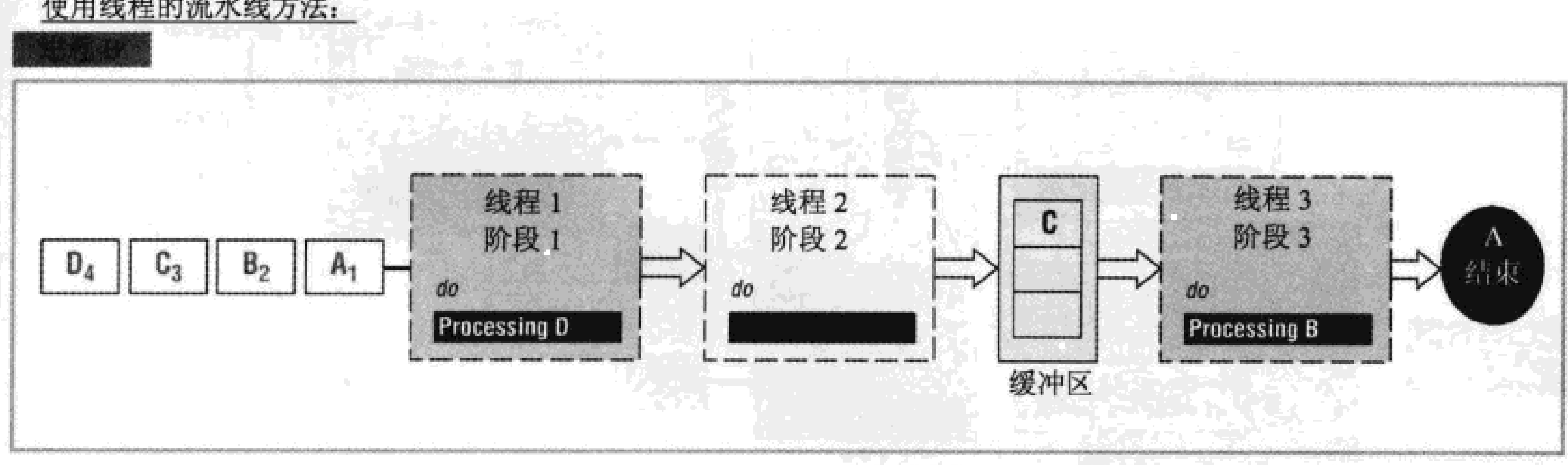
图 B-6

B.12　使用线程的流水线方法

流水线中的每个线程从队列中对输入进行处理。一旦一个线程已经完成处理后，将结果传递给流水线中的下一个线程。每个线程代表流水线中的一个阶段，用于处理中间结果。这样就允许同时处理多个输入。最后一个线程产生流水线的最终结果。如果一个线程比前面的线程要慢，则可以在线程之间使用缓冲区。图 B-7 的顶部给出了一个示例。

B.13　使用线程的生产者/消费者方法 1

生产者线程产生将被消费者线程消耗的数据。数据保存在内存块中，该内存块中的数据被消费者线程和生产者线程共享。将数据保存到内存中以及从内存中提取数据是需要同步的。图 B-7 的底部给出了一个示例。



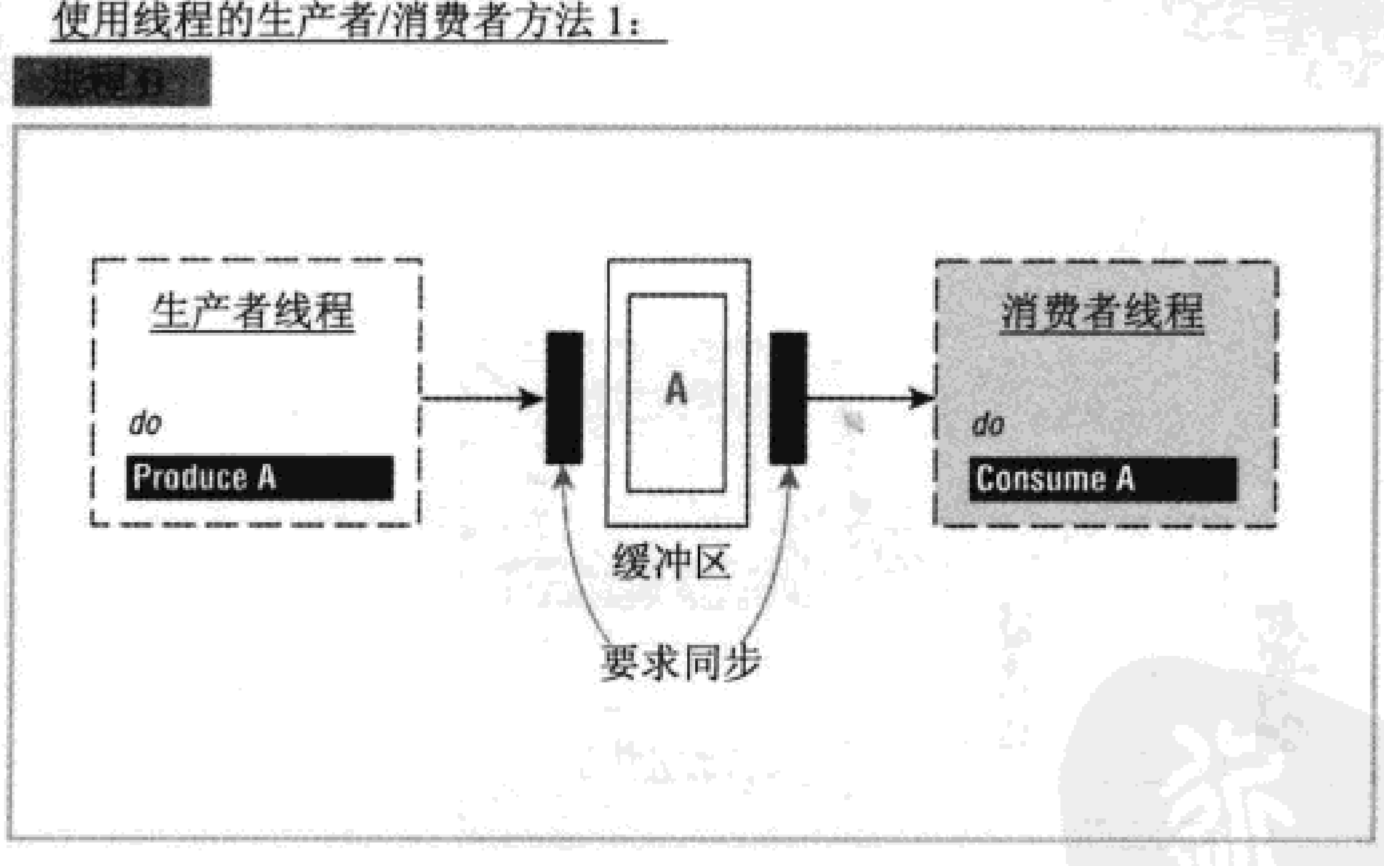


图 B-7

B.14　使用线程的生产者/消费者方法 2

多个生产者线程产生将被一个消费者线程消耗的数据。数据保存在被这些线程共享的

内存块中。将数据保存到内存中以及从内存中提取数据是需要同步的。图 B-8 的顶部给出了一个示例。

B.15 使用线程的生产者/消费者方法 3

多个生产者线程产生将被多个消费者线程消耗的数据。数据保存在被这些线程共享的内存块中。将数据保存到内存中以及从内存中提取数据是需要同步的。图 B-8 的底部给出了一个示例。

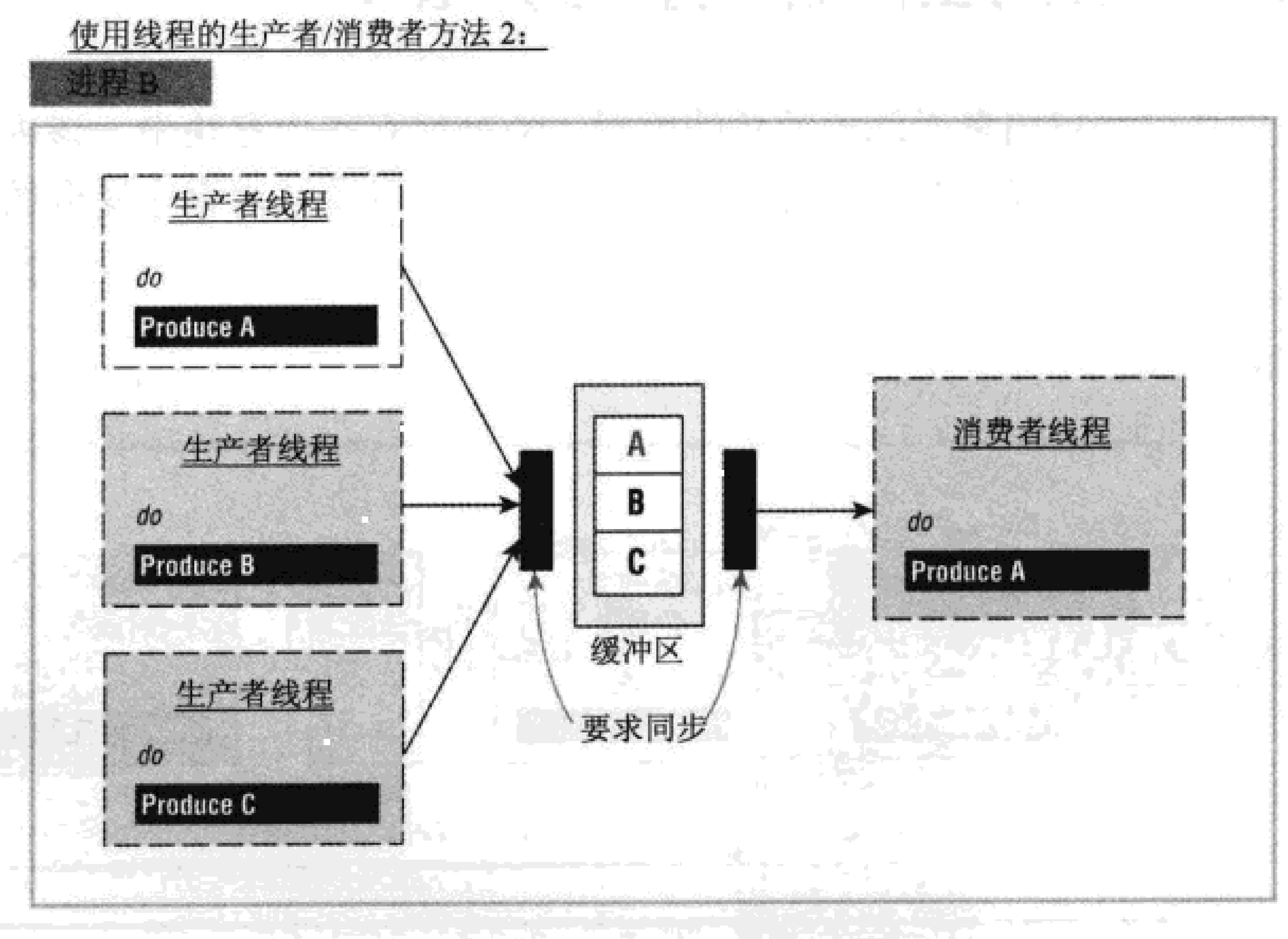


图 B-8

B.16 监管程序方法

监管程序(monitor)包含了同步对连续可重用共享资源、数据或对象的访问的数据和方法。worker 必须进入到监管程序并得到对一个条件变量的访问权限，才能够访问共享数据。图 B-9 给出了一个示例。

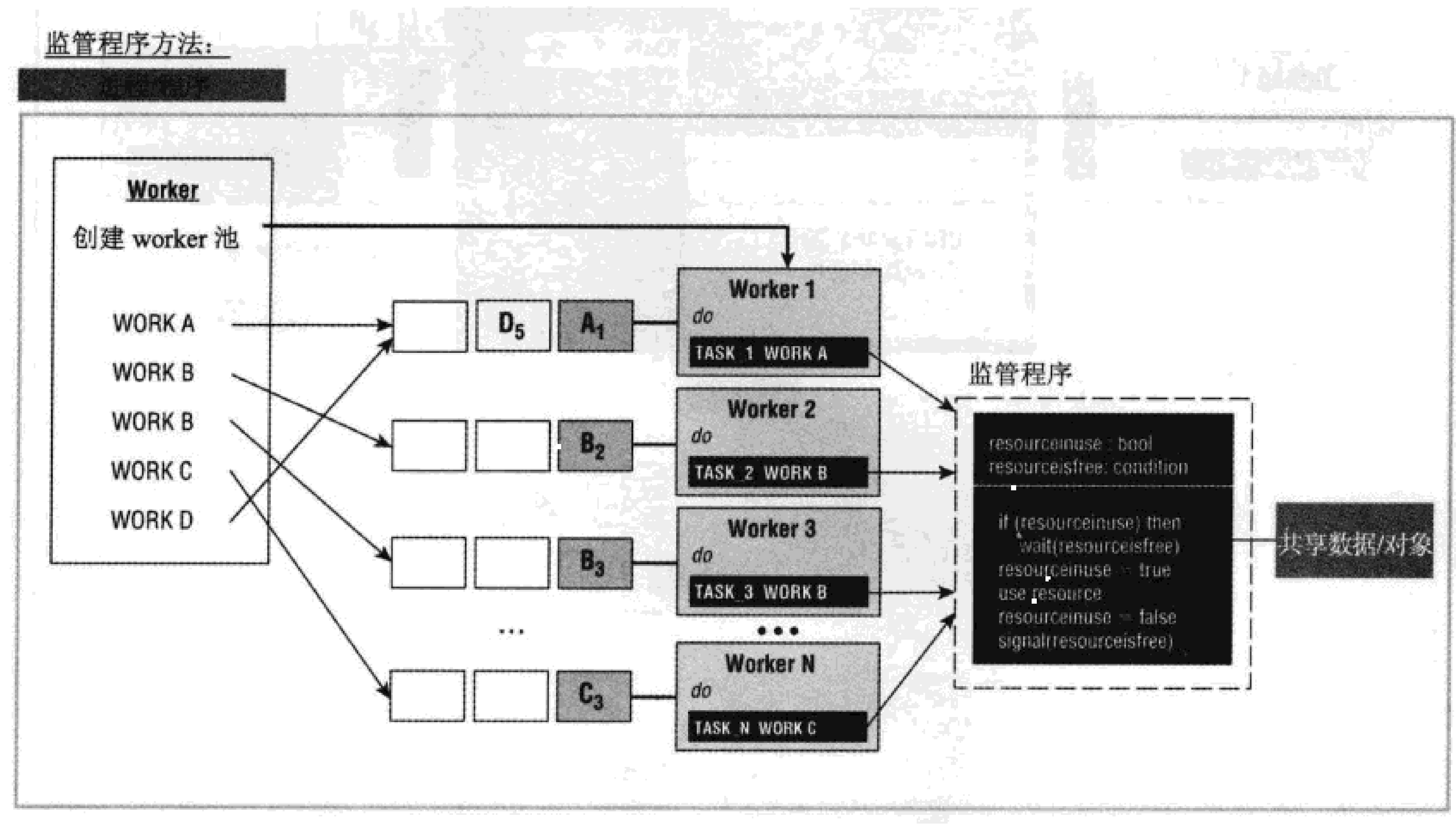


图 B-9

B.17 使用线程的黑板方法

黑板(blackboard)是一个集中的对象，每个线程都可以访问它。对黑板的访问必须进行同步。每个线程可以张贴初步的结果或数据。线程还可以对结果或数据进行处理以创建部分解。图 B-10 给出了一个示例。

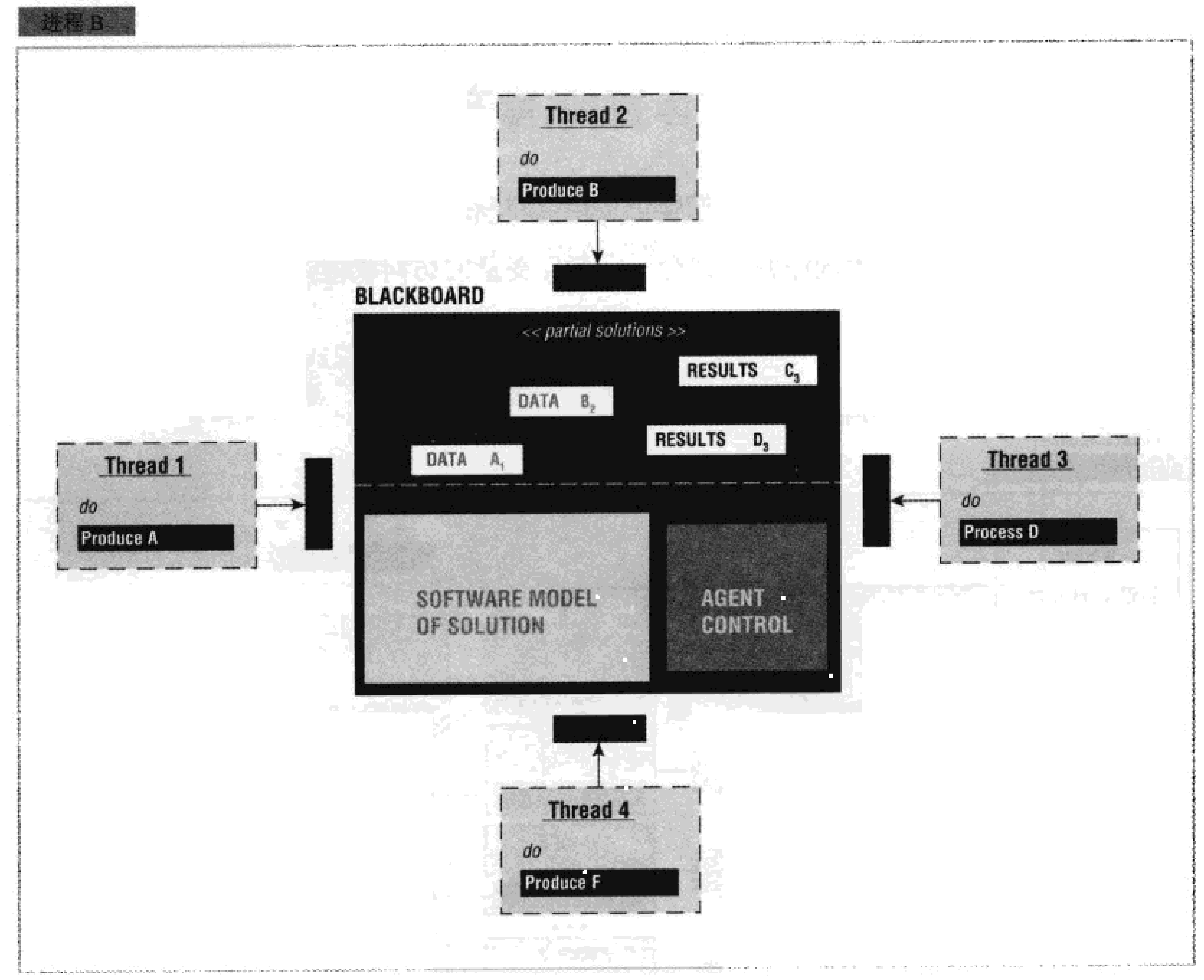


图 B-10

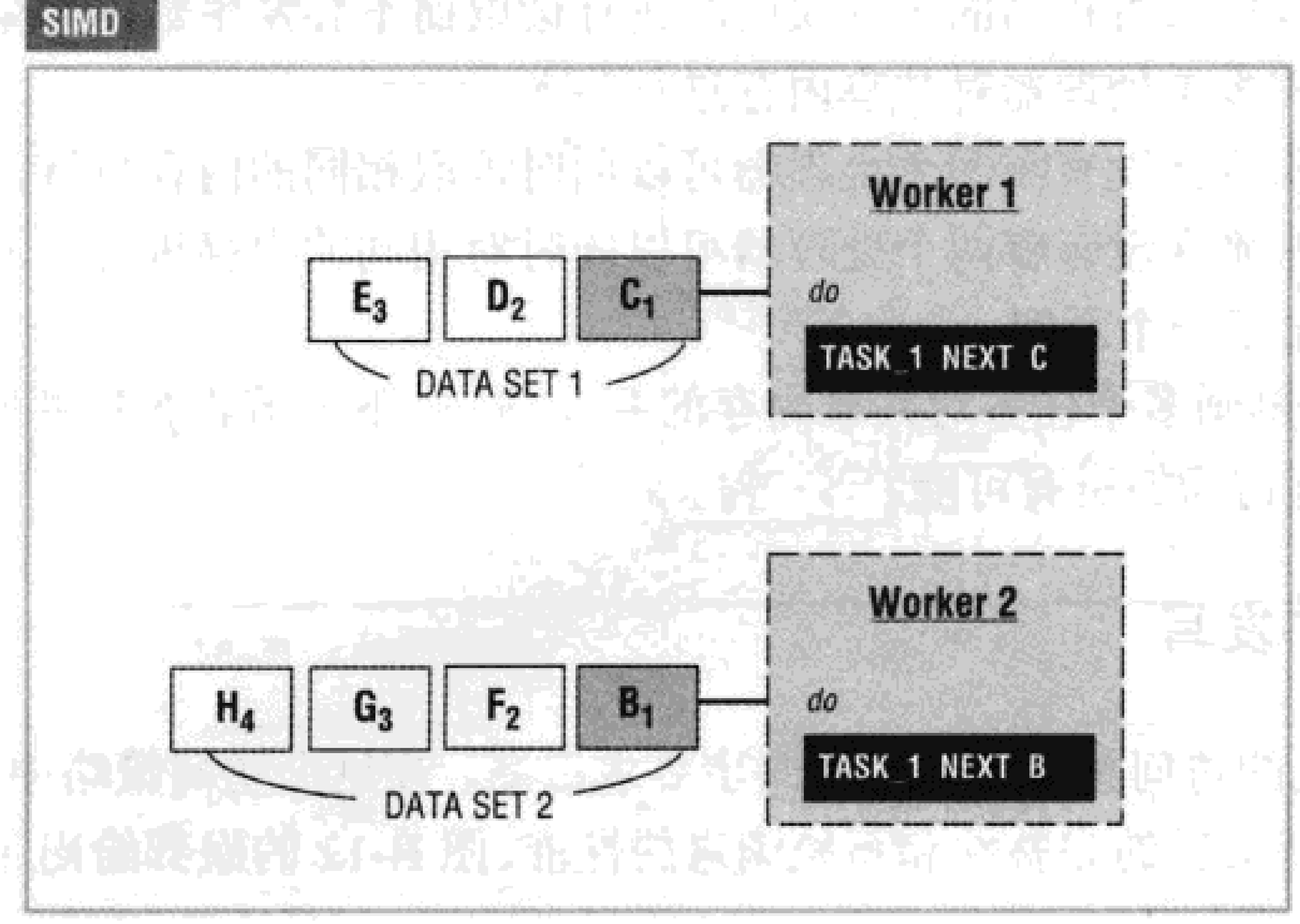
B.18 数据级并行性：SIMD 方法

单指令多数据(Single Instruction Multiple Data，SIMD)意味着一条指令在不同的数据或数据集上执行。线程或进程可以执行相同的任务或者是不同的线程/进程可以执行相同的任务。图 B-11 的顶部给出了一个示例。

B.19 数据级并行性：MIMD 方法

多指令多数据(Multiple Instruction Multiple Data，MIMD)意味着多条指令在不同的数据或数据集上执行。多个线程或进程用来执行不同的任务。图 B-11 的底部给出了一个示例。

数据级并行性：SIMD 方法



数据级并行性：MIMD 方法

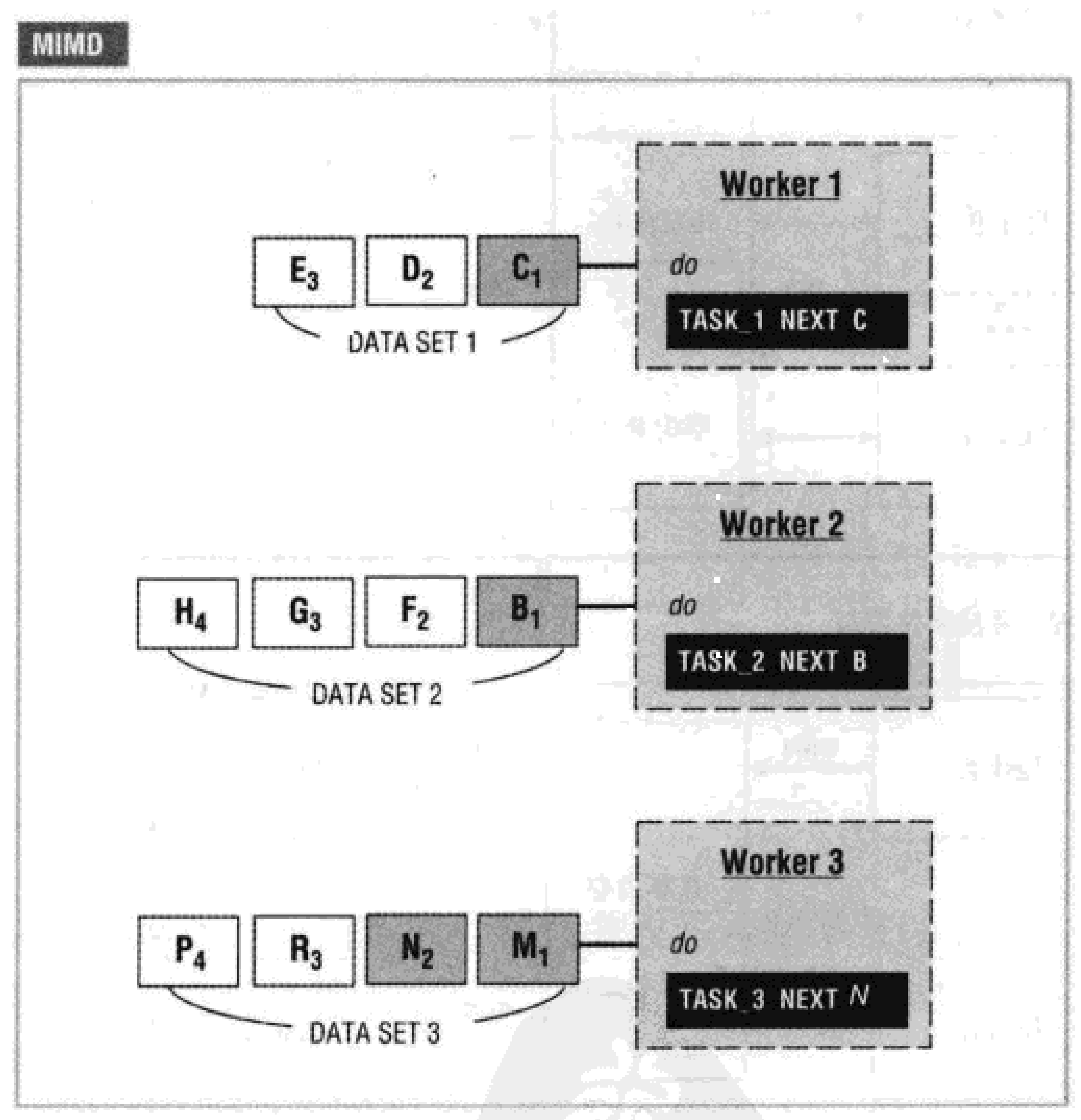


图 B-11

B.20　PRAM 模型

并行随机存取计算机(Parallel Random Access Machine，PRAM)是多个处理器共享全局内存的理想模型。所有的处理器都可以同时对共享的全局数据进行读取或写入的访问。PARM

模型可用于对共享全局内存进行同时访问的任务。有 4 种算法可用于访问共享全局内存：

(1) 并发读算法可以用来对相同的内存块同时进行读取而不会发生数据污染。

(2) 并发写算法允许多个处理器对共享内存进行写入。

(3) 互斥读算法用来确保没有哪两个处理器可以同时读取相同的内存位置。

(4) 互斥写算法用来确保没有哪两个处理器可以同时对相同的内存进行写入。

图 B-12 的顶部给出了一个示例。

PARM 模型可用来刻画多个任务对共享内存的并发访问。下一节将解释并发和互斥读-写算法，它结合了读-写访问的所有可能。

CRCW——并发读并发写

CRCW 是最难实现的访问策略，并且对同步要求最高。在这个访问策略中，允许无限制的访问，但是必须维持数据的完整性和系统满意的性能。图 B-12 的底部给出了一个示例。

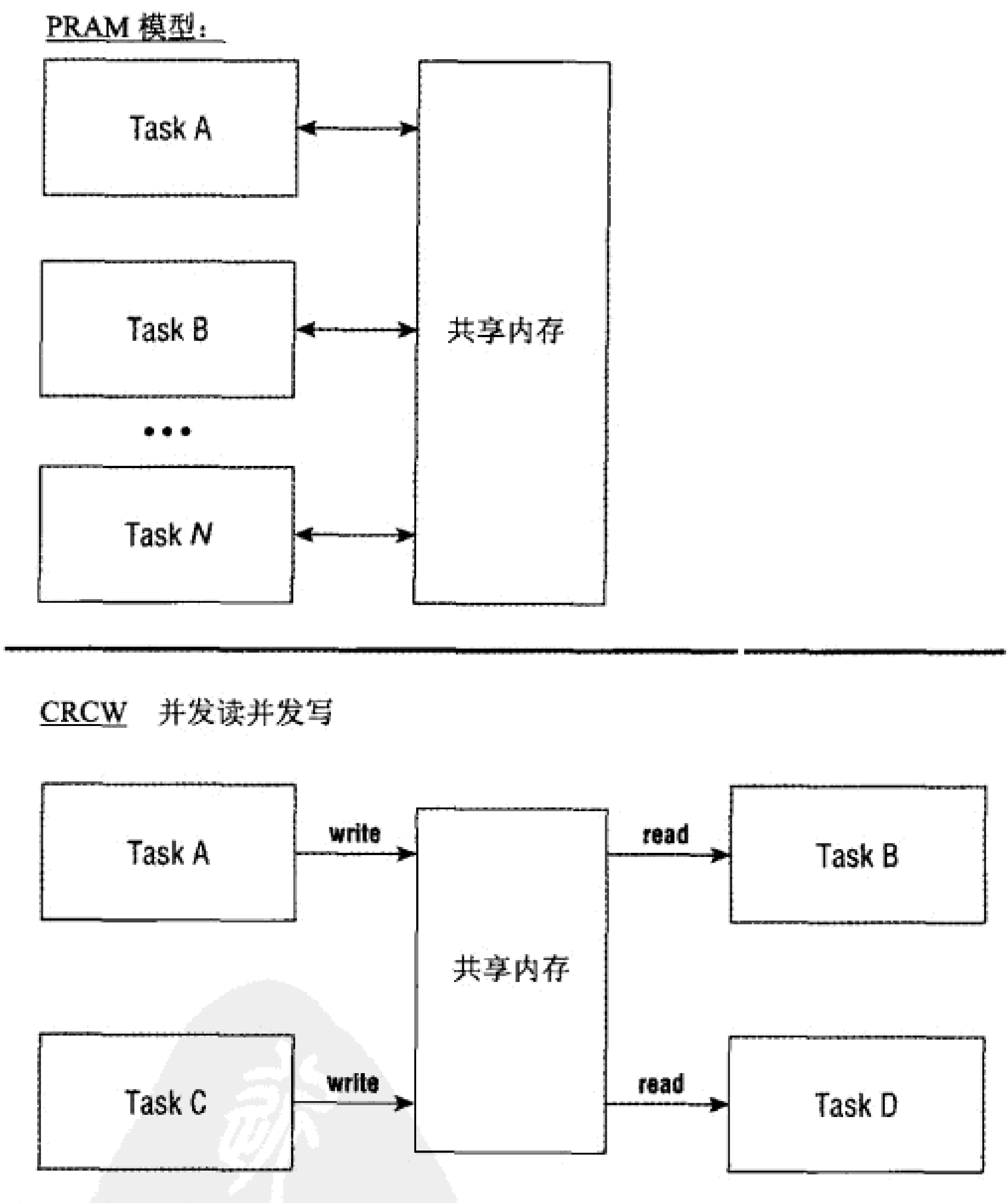


图 B-12

EREW——互斥读互斥写

EREW 意味着对共享内存的访问被线性化，每次只有一个任务能够访问共享内存，无论这个访问是读取或写入。EREW 访问策略的一个示例是 producer-consumer。这是最为严格的访问策略。参见图 B-13 的顶部。

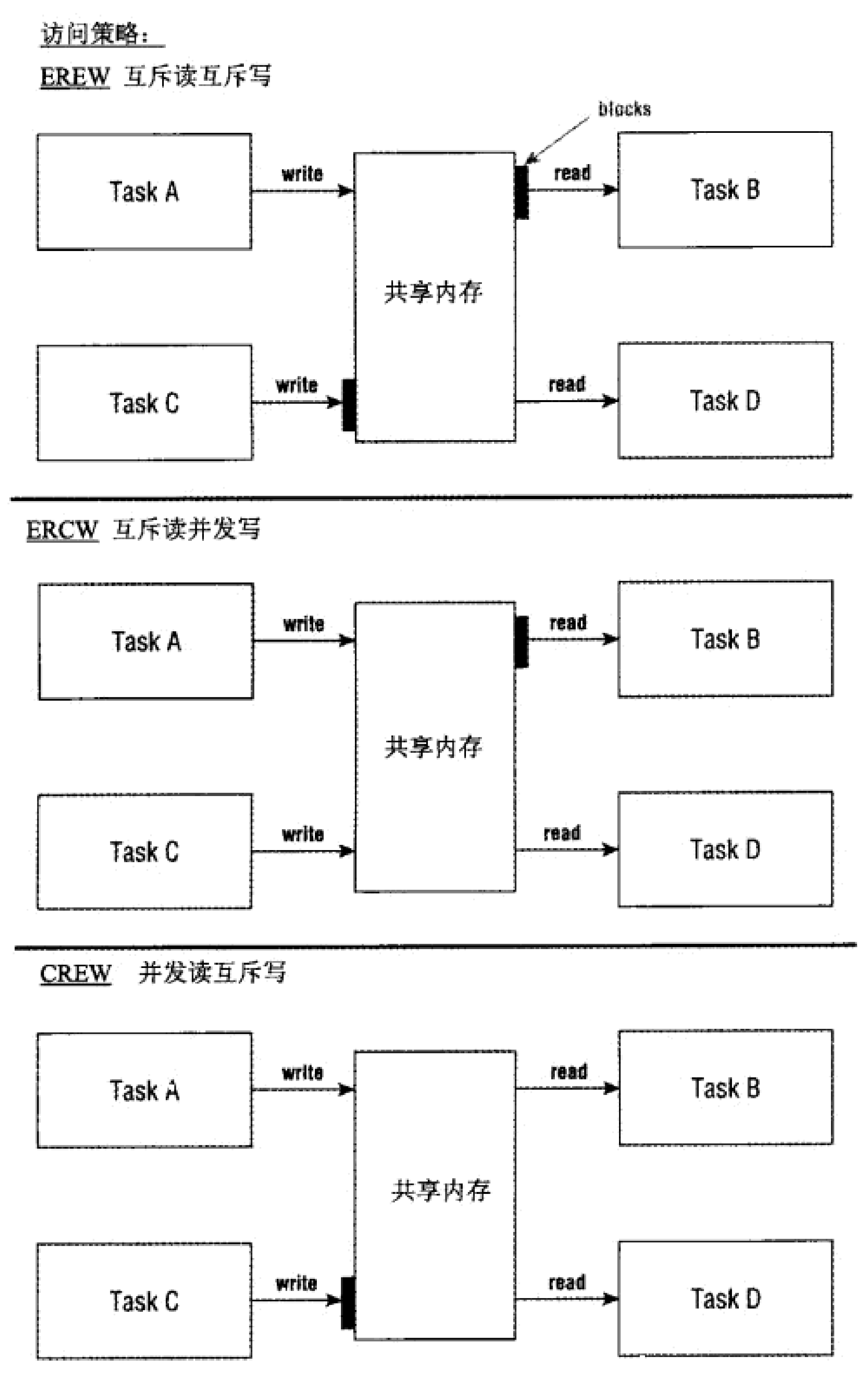


图 B-13

ERCW——互斥读并发写

ERCW 访问策略允许对共享内存进行并发写入和排他读取。对并发写入共享内存的任务的数目没有限制，但是这种策略只允许一个任务读取共享内存。这种访问策略一般不会被实现。参见图 B-13 的中部。

CREW——并发读互斥写

CREW 访问策略允许对共享内存的并发读取和互斥写入。对同时读取共享内存的任务的数目没有限制，但是策略只允许一个任务对共享内存进行写入。并发读取可以在互斥写入发生的同时发生。使用这种访问策略时，在其它任务进行写入期间，每个读取任务可能会读到不同的值。下一个读取共享内存的任务将会看到于其他任务所看到的数据不同的数据。参见图 B-13 的底部。

附 C 录

线程管理的 POSIX 标准

本附录包含来自 POSIX Standard for Thread Management 的部分内容。POSIX 是全世界所接受的开放操作系统接口标准。它由 IEEE 完成并被 ISO 和 ANSI 所认同。对 POSIX 标准的支持确保系统之间的代码移植性，在商业应用和政府合同中正逐渐占据统治地位。POSIX 标准是跨平台多核开发最广泛可用的方法。它同高级线程库兼容，例如 Solaris Threads、Intel Thread Building Block 和新的 Standard C++0x。POSIX 标准内容丰富，包含上千页。为了方便起见，我们节选了用于线程管理的标准。这些部分中包含的 API 函数要么在本书中提及过，要么就同开发多核应用程序有关。

下面的内容是经 IEEE 允许，对 IEEE std. 1003.1-2001，IEEE standard for Information Technology-Portable Operating System Interface(POSIX), Copyright 2001 的重印。

名称

pthread_atfork——注册 fork 处理程序。

调用形式

THR

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```

描述

pthread_atfork( )函数将在调用 fork( )的线程的上下文中，声明在 fork( )之前和之后会被调用的 fork 处理程序。应当在 fork( )处理开始之前调用 fork 处理程序 prepare。父进程中应当在 fork( )处理完成之后调用 fork 处理程序 parent。子进程中应当在 fork( )处理完成之后调用 fork 处理程序 child。如果在这 3 个位置没有需要进行的处理，那么对应的 fork 处理程序地址可以设置为 NULL。

pthread_atfork( )调用的顺序非常重要。parent 和 child 这两个 fork 处理程序应当按照它们通过调用 pthread_atfork()来建立的顺序进行调用。fork 处理程序 prepare 应当以相反的顺序调用。

返回值

若成功完成，pthread_atfork( )应当返回 0 值；否则，应当返回错误号以指示该错误。

错误

如果发生以下情况，则 pthread_atfork( )将失败：

[ENOMEM] 没有足够的表空间来记录 fork 处理程序地址。

pthread_atfork( )函数不应当返回错误代码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

在多线程程序中，关于 fork( )的语义至少有两个严重的问题。一个问题同互斥量覆盖的状态(如内存)有关。考虑如下情形：一个线程已经对一个互斥量加锁，而且该互斥量覆盖的状态在另一个线程调用 fork( )期间是不一致的。在子线程中，互斥量处于加锁状态(被一个不存在的线程加锁，这样就永远不会被解锁)。令子线程重新初始化互斥量不够让人满意，因为这种方法没有解决如何修正或处理子线程中的不一致。

建议使用 fork( )的程序在子进程中随后很快地调用 exec 函数，这样就会重置所有状态。同时，在此期间，应当确保只有简短的异步信号安全(async-signal-safe)的库例程出现。

不幸的是，这种解决方案没有解决多线程库的需要。应用程序可能不知道正在使用一个多线程库，因此它们随意地在 fork( )和 exec 调用之间调用任意数量的库例程，正如它们已经做的。实际上，它们可能是尚存的单线程程序，因此无法被指望遵守线程库所提出的新的限制。

另一方面，假使之后又在子进程中被重入，多线程库需要有一种方法保护 fork( )其内部状态。这个问题在多线程 I/O 库中尤为明显，该库几乎肯定会在 fork( )和 exec 调用之间被唤起，以影响 I/O 重定向。解决方案可能要求在 fork( )期间对互斥量变量加锁，或者可能要求在 fork( )处理完成之后必须重置子进程中的状态。

pthread_atfork( )函数为多线程库提供了一种方式来保护自身不受调用 fork( )的应用程序的影响，并且为多线程应用程序提供了一种标准机制，保护它们不受库例程或应用程序自身中的 fork( )调用的影响。

期望的用法是 prepare 处理程序请求所有的互斥信号量锁，然后另外两个 fork 处理程序释放它们。

例如，应用程序可以提供一个 prepare 例程，它请求所有必需的互斥量，库维护并提供 child 和 parent 例程，它们释放这些互斥量，这样来确保子进程得到库状态的一致快照(而且没有哪个信号量被忽视)。或者，某些库可能能够只提供 child 例程，它将库中的互斥量以及所有关联的状态重新初始化到一些已知的值(例如，当映像最初被执行时是怎样的)。

当调用 fork( )时，子进程中只复制了调用线程。同步变量在子进程中仍然保持与父进程调用 fork( )时的相同的状态。这样，互斥量锁可能被不在子进程中存在的线程所持有，所有相关联的状态可能都是不一致的。父进程可以通过 pthread_atfork( ) 使用显式编码获得并释放对于子进程非常关键的锁，从而避免这个问题。此外，在子进程中需要将关键线

程重新创建并重新初始化到适当的状态(也是通过 pthread_atfork( ))。

高层的包可能要求在调用低层的包时对自身的数据结构获得锁。在这种场景中，为 fork 处理程序调用指定的顺序使用一条简单的初始化规则来避免包发生死锁：在为自己调用 pthread_atfork( )函数之前，初始化它依赖的所有的包。

未来方向

无。

参见

atexit( )、fork( )、the Base Definitions volume of IEEE std 1003.1-2001 和**<sys/types.h>**。

变更历史

首次发布于 Issue 5。来自 POSIX Threads Extension。

应用了 IEEE PASC Interpretation 1003.1c #4。

Issue 6

将 pthread_atfork( )函数标注为 Threads 选项的一部分。

将<pthread.h>头文件加入到调用形式中。

名称

pthread_attr_destroy 和 pthread_attr_init——销毁和初始化线程属性对象

调用形式

THR

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```

描述

pthread_attr_destroy( )函数将销毁一个线程属性对象。实现可能会令 pthread_attr_destroy( )将 attr 设置成一个由实现所定义的无效值。使用 pthread_attr_init( )可以对已经销毁的 attr 属性对象进行重新初始化；否则引用被销毁的对象的结果未定义。

pthread_attr_init( )函数在对线程属性对象 attr 进行初始化时，会使用给定实现中为 attr 所使用的所有属性提供的默认值进行初始化。

pthread_create( )所使用的作为结果的属性对象(可能会通过设置了个别属性值进行修改)定义了所创建线程的属性。一个属性对象可以用于对 pthread_create( )的多个同时调用。如果在调用 pthread_attr_init()时，指定一个已经初始化的 attr 属性对象，其结果是未定义的。

返回值

成功完成时，pthread_attr_destroy( )和 pthread_attr_init( )均会返回 0；否则，就会返回一个错误号以指示所出现的错误。

错误

如果出现了下面的情况，pthread_attr_init( )函数将会失败：

[ENOMEM]　没有足够的内存来初始化线程属性对象。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

属性对象是作为一种机制提供给线程、互斥量和条件变量的，以便在无需修改函数本身的情况下支持这些领域中未来的标准化。

属性对象为线程的可配置方面提供了清晰的隔离。例如，stack size 是线程的一个重要属性，但是它无法以一种可移植的方式表示。在对线程化程序进行移植时，往往需要对栈大小进行调整。使用属性对象就可以将所有的修改隔离在一处，而无需在线程创建的各个

实例中进行。

属性对象可以用于设置具有相似属性的线程“类”；例如：“具有大的栈和高优先级的线程”或者“具有最小栈的线程”。可以在某处定义这些类，然后无论在何处只要需要创建这类线程，可以引用它们。这样，对“类”决策的修改也就相当直观，并且无需对每个pthread_create( )调用进行详细的分析。

属性对象被定义为作为对可扩展性的辅助的非透明类型。如果这些对象已经被指定为结构，添加新属性就会在扩展属性对象时迫使对所有的多线程程序进行重新编译；如果不同的程序组件是由不同的供货商提供的，那么这可能无法实现。

此外，非透明属性对象提供了改进性能的机会。参数有效性在属性设置时检查一次即可，无需每次创建线程时都去进行检查。在很多实现中，往往都会对那些创建代价高的内核对象进行缓存。非透明属性对象提供了有效的机制来检测缓存的对象何时会由于属性变化而变得无效。

由于对于给定的非透明类型，未必会定义赋值，因此无法以可移植的方式定义由实现定义的默认值。该问题的解决方案为：允许属性对象初始化函数对属性对象进行动态初始化，从而具体的实现就可以动态地提供默认值。

作为所提供属性的一种建议性选择，下面给出了一些建议。

(1) 保持向初始化例程(pthread_create( )、pthread_mutex_init( )和 pthread_cond_init( ))进行参数传递的风格：对标志进行按位或操作来构成参数。出于可扩展性的考虑，含有标志的参数应该是一种非透明类型。如果在参数中没有设置任何标志，那么就会用默认特征来创建对象。具体实现可以指定由实现定义的标志值及其相关行为。

2. 如果有必要对互斥量和条件变量做进一步特化，实现可以指定对 pthread_mutex_t 及 pthread_cond_t 对象(而不是对属性对象)的额外过程。

这种解决方案的困难在于：

(1) 如果要使用显式编码的按位或操作来将位设置到位向量属性对象中，位掩码就不是非透明的。如果选项集超出 int 的长度，应用程序员就需要知道每一位的具体位置。如果位是通过封装来设置或者读取的(即：get 和 set 函数)，那么位掩码就仅仅是属性对象的一种实现，并且不应该暴露给编程人员。

(2) 很多属性不是 Boolean 或者很小的整数值。例如，调度策略可能使用 3 位或 4 位来表示，但是优先级需要 5 位或者更多位，因此就会占用整数为 16 位的机器里可能的 16 位中的至少 8 位。鉴于此，位掩码只能比较合理地控制特定属性是否被设置了，但是无法作为值本身的库。值需要作为函数参数来指定(这样是不可扩展的)，或通过设置结构域来实现(这样就不是非透明的)，要么通过 set 和 get 函数来实现(这样就使得掩码成了属性对象的一种冗余)。

由于栈本身就是机器相关的，因此 stack size 被定义为可选属性。在一些实现中可能会无法改变栈的大小，而另一些则有可能无需这样做，因为栈页可能是不连续的并且可以按需分配和释放。

属性机制已经在很大程度上为扩展性进行了设计。为了不影响到二进制兼容性，未来对属性机制或者定义在 IEEE Std 1003.1-2001 该卷中的任何属性对象的扩展都必须要谨慎地对待。

属性对象，即使是通过诸如 malloc()之类的动态分配函数进行分配的，在编译时也可能已经大小固定了。这意味着某种(带有对 pthread_attr_t 的扩展)具体实现中的 pthread_create( )函数就无法查看二进制应用程序认为有效的区域之外的区域了。这就暗示着如果要容纳不同方向的扩展(可能由不同供货商进行)，那么具体的实现应该在属性对象中维护一个 size 域，以及可能的版本信息。

未来方向

无。

参见

pthread_attr_getstackaddr( )、pthread_attr_getstacksize( )、pthread_attr_getdetachstate( )、pthread_create( )、the Base Definitions Volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，引入的目的在于同 POSIX Threads Extension 相一致。

Issue 6

pthread_attr_destroy( )和 pthread_attr_init( )函数被标记为 Threads 选项的一部分。

应用了 IEEE PASC Interpretation 1003.1#107，对一个已经初始化了的线程属性对象进行初始化的结果是未定义的。

名称

pthread_attr_getdetachstate 和 pthread_attr_setdetachstate——获取和设置 detachstate 属性。

调用形式

THR

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
    int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

描述

detachstate 属性控制着线程是否以一种分离状态来创建。如果线程是以分离状态创建的，那么 pthread_detach( )或者 pthread_join( )函数使用新创建的线程的 ID 就是一种错误。

pthread_attr_getdetachstate( )和 pthread_attr_setdetachstate( )函数将分别在 attr 对象中获取和设置 detachstate 属性。

对于 pthread_attr_getdetachstate( )，dctachstate 将会被设置成 PTHREAD_CREATE_DETACHED 或 PTHREAD_CREATE_JOINABLE。

对于 pthread_attr_setdetachstate( )，应用程序会将 detachstate 设置成 PTHREAD_CREATE_DETACHED 或 PTHREAD_CREATE_JOINABLE。

PTHREAD_CREATE_DETACHED 会导致所有使用 attr 创建的线程处于分离状态，而使用 PTHREAD_CREATE_JOINABLE 则会导致所有使用 attr 创建的线程处于可结合状态。detachstate 属性的默认值为 PTHREAD_CREATE_JOINABLE。

返回值

若成功完成，pthread_attr_getdetachstate( )和 pthread_attr_setdetachstate( )函数都会返回 0；否则，就会返回一个错误号以表示所产生的错误。

如果成功，pthread_attr_getdetachstate( )函数就会将 detachstate 属性的值保存到 detachstate。

错误

如果出现了下面的情况，pthread_attr_setdetachstate( )函数就会失败：

[EINVAL]　detachstate 的值无效。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getstackaddr( )、pthread_attr_getstacksize( )、pthread_create( )、the Base Definitions volume of IEEE STD 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的在于与 POSIX Threads Extension 一致。

Issue 6

pthread_attr_setdetachstate( )和 pthread_attr_getdetachstate( )函数被标记为 Threads 选项的一部分。

对“描述”进行了更新，对应用程序需求应避开使用“必须(must)”。

名称

pthread_attr_getguardsize 和 pthread_attr_setguardsize——获取和设置线程 guardsize 属性。

调用形式

XSI

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
    size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

描述

pthread_attr_getguardsize( )函数会获得 attr 对象中的 guardsize 属性。该属性将会在 guardsize 参数中返回。

pthread_attr_setguardsize( )函数会设置 attr 对象中的 guardsize 属性。该属性的新值从 guardsize 参数获得。如果 guardsize 为 0，那么就不会为使用 attr 创建的线程提供 guard 区域。如果 guardsize 大于 0，那么就会为使用 attr 创建的每个线程提供大小至少为 guardsize 的 guard 区域。

guardsize 属性控制所创建线程的栈的 guard 区域的大小。guardsize 属性为避免栈指针溢出提供了保护。如果线程的栈是使用 guard 保护创建的，那么具体实现就会在栈的溢出端分配额外的内存作为防止栈指针溢出的缓冲区。如果应用程序溢出到了这个缓冲区中，那么就会导致错误(该错误可能在发送给线程的 SIGSEGV 信号中)。

符合的实现可能会将 guardsize 中的值集中到可配置系统变量{PAGESIZE}(参见<sys/mman.h>)的倍数。如果实现将 guardsize 的值集中到{PAGESIZE}的倍数，那么当指定 attr 调用 pthread_attr_getguardsize( )时，就会在 guardsize 参数中保存之前的 pthread_attr_setguardsize( )函数调用所指定的 guard size。

guardsize 的默认值为{PAGESIZE}字节，{PAGESIZE}的实际大小由具体实现定义。

如果已经设置了 stackaddr 或者 stack 属性(即：调用者分配和管理它自己的线程栈)，那么 guardsize 属性就会被忽略掉，并且实现也不会提供任何保护。在这种情况下，对栈溢出的管理以及栈本身的分配、管理将会由应用程序来负责。

返回值

如果成功，pthread_attr_getguardsize( )和 pthread_attr_setguardsize( )函数都会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_attr_getguardsize( )和 pthread_attr_setguardsize( )函数将会失败：

[EINVAL]　属性 attr 无效。

[EINVAL]　参数 guardsize 无效。

这两个函数均不会返回错误码[EINTR]。

示例
无。

应用程序使用
无。

基本原理
之所以为应用程序提供 guardsize 属性，有下面两个原因：

(1) 溢出保护会潜在地导致系统资源的浪费。创建了大量线程并且知道其线程永远不会发生栈溢出的应用程序，可通过关闭 guard area 来节约系统资源。

(2) 如果线程在栈上分配大的数据结构，可能就会需要大的 guard area 来检测栈溢出。

未来方向
无。

参见
The Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sys/mman.h>。

变更历史
首次发布于 Issue 5。

Issue 6
在“错误”部分，删除了第三个[EINVAL]错误情形，因为实际上它已经被第二个错误情形所包括了。

为了与 ISO/IEC 9899:1999 标准匹配，为 pthread_attr_getguardsize( )原型加入了 restrict 关键字。

名称

pthread_attr_getinheritsched 和 pthread_attr_setinheritsched——获取和设置 inheritsched 属性(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
    int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

描述

pthread_attr_getinheritsched( )和 pthread_attr_setinheritsched( )函数，会分别获取和设置 attr 参数中的 inheritsched 属性。

当 pthread_create( )使用属性对象时，将会根据 inheritsched 属性来判断如何设置所创建线程的其他调度属性。

PTHREAD_INHERIT_SCHED

这个值说明所创建线程的调度属性将从创建线程继承，并且这个 attr 参数中的调度属性将会被忽略。

PTHREAD_EXPLIXIT_SCHED

这个值说明所创建线程的调度属性将会被设置为该属性对象中对应的值。

PTHREAD_INHERIT_SCHED 和 PTHREAD_EXPLICIT_SCHED 符号在<pthread.h>头文件中进行了定义。

下面这些由 IEEE Std 1003.1-2001 所定义的线程调度属性会受到 inheritsched 属性的影响：调度策略(schedpolicy)、调度参数(schedparam)和调度竞争范围(contentionscope)。

返回值

如果调用成功，pthread_attr_getinheritsched( )和 pthread_attr_setinheritsched( )函数将会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_attr_setinheritsched( )函数可能会失败：

[EINVAL]　inheritsched 的值无效。

[ENOTSUP]　试图将属性设置为一种不被支持的值。

这两个函数均不会返回错误码[EINTR]。

实例

无。

应用程序使用

设置了这些属性之后，就可以使用 pthread_create( )函数用指定的属性创建线程。这些例程的使用不会影响到当前的运行线程。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getscope( )、pthread_attr_getschedpolicy( )、pthread_attr_getschedparam( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 相一致。

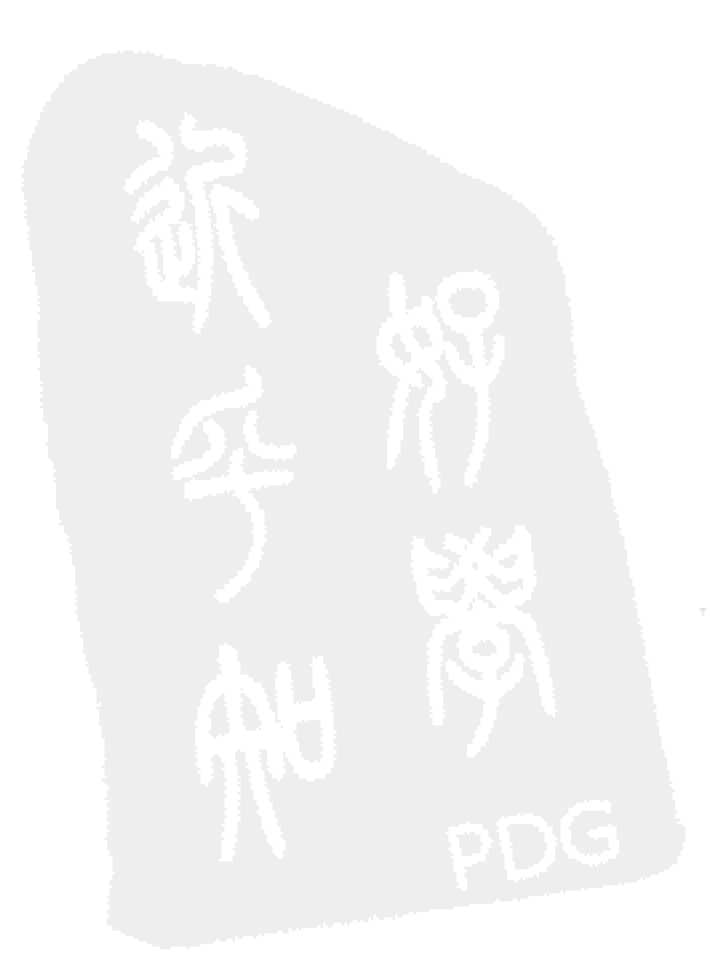
被标记为 Realtime Threads Feature Group 的一部分。

Issue 6

pthread_attr_getinheritsched( )和 pthread_attr_setinheritsched( )函数被标记为 Threads 和 Thread Execution Scheduling 选项的一部分。

删除了[ENOSYS]错误条件，原因在于如果某种实现不支持 Thread Execution Scheduling 选项，那么，就无需提供 stub。

为了与 ISO/IEC 9899:1999 标准相一致，在 pthread_attr_getinheritsched( )原型中加入了 restrict 关键字。



名称

pthread_attr_getschedparam 和 pthread_attr_setschedparam——获取和设置 schedparam 属性。

调用形式

THR

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
    struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

描述

pthread_attr_getschedparam( )和 pthread_attr_setschedparam( )函数分别获取和设置 attr 参数中的调度参数属性。param 结构的内容定义于<sched.h>头文件中。对于 SCHED_FIFO 和 SCHED_RR 策略，param 要求的唯一成员是 sched_priority。

对于 SCHED_SPORADIC 策略，param 结构所要求的成员为 sched_priority、sched_ss_low_priority、sched_ss_repl_period、sched_ss_init_budget 和 sched_ss_max_repl。指定的 sched_ss_repl_period 必须要大于或等于指定的 sched_ss_init_budget 的值，这样函数才能成功；否则函数就会失败。sched_ss_max_repl 的值要位于范围[1.{SS_REPL_MAX}]之内，这样函数才会成功；否则函数将失败。

返回值

如果成功，pthread_attr_getschedparam( )和 pthread_attr_setschedparam( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_attr_setschedparam( )函数可能会失败：

[EINVAL] param 的值无效。

[ENOTSUP] 试图将属性设置为不被支持的值。

这两个函数均不会返回错误码[EINTR]。

实例

无。

应用程序使用

在设置了这些属性之后，就可以使用所指定的属性调用 pthread_create( )函数来创建线程。使用这些例程不会影响当前运行的线程。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getscope( )、pthread_attr_getinheritsched( )、pthread_attr_getschedpolicy( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 相一致。

Issue 6

pthread_attr_getschedparam( )和 pthread_attr_setschedparam( )函数被标记为 Threads 选项的一部分。

为了与 IEEE Std 1003.1.d-1999 一致而添加了 SCHED_SPORADIC 调度策略。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_attr_getschedparam( )和 pthread_attr_setschedparam( )的原型中加入了 restrict 关键字。

名称

pthread_attr_getschedpolicy 和 pthread_attr_setschedpolicy——获取和设置 schedpolicy 属性(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
    int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

描述

pthread_attr_getschedpolicy( )和 pthread_attr_setschedpolicy( )函数分别获取和设置 attr 参数中的 schedpolicy 属性。

所支持的 policy 值包括：SCHED_FIFO、SCHED_RR 以及 SCHED_OTHER，它们均定义在<sched.h>头文件中。当一些调度策略为 SCHED_FIFO、SCHED_RR 或者 SCHED_SPORADIC 的执行线程在等待一个互斥量时，则当该互斥量解锁时，那么它们将会按照优先级的顺序获得该互斥量。

返回值

如果调用成功，pthread_attr_getschedpolicy( )和 pthread_attr_setschedpolicy( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_attr_setschedpolicy( )函数可能会失败：

[EINVAL]　policy 的值无效。

[ENOTSUP] 试图将属性设置为不被支持的值。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

在设置了这些属性之后，就可以调用 pthread_create( )函数使用指定的属性创建线程。使用这些例程不会影响到当前运行的线程。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getscope( )、pthread_attr_getinheritsched( )、pthread_attr_getschedparam( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

被标记为实时线程特征组(Realtime Threads Feature Group)的一部分。

Issue 6

pthread_attr_getschedpolicy( )和 pthread_attr_setschedpolicy( )函数被标记为 Threads 和 Thread Execution Scheduling 选项的一部分。

删除了[ENOSYS]错误条件，原因在于如果某种实现不支持 Thread Execution Scheduling 选项，那么就无需提供 stub。

为了与 IEEE Std 1003.1.d-1999 一致，添加了 SCHED_SPORADIC 调度策略。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_attr_getschedpolicy( )原型中加入了 restrict 关键字。

名称

pthread_attr_getscope 和 pthread_attr_setscope——获取和设置 contentionscope 属性(实时线程)。

调用形式

THR TPS

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
    int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

描述

pthread_attr_getscope( )和 pthread_attr_setscope( )函数分别获取和设置 attr 参数中的 contentionscope 属性。

contentionscope 属性的值可以是 PTHREAD_SCOPE_SYSTEM，意味着系统调度竞争范围；或者是 PTHREAD_SCOPE_PROCESS，意味着进程调度竞争范围。符号 PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS 均在<sched.h>头文件中定义。

返回值

如果成功，pthread_attr_getscope( )和 pthread_attr_setscope( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_attr_setscope( )函数可能就会失败：

[EINVAL]　contentionscope 的值无效。

[ENOTSUP] 试图将属性设置为不被支持的值。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

在设置了这些属性之后，就可以调用 pthread_create( )函数使用指定的属性创建线程。使用这些例程不会影响到当前运行的线程。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getinheritsched( )、pthread_attr_getschedpolicy( )、pthread_attr_getschedparam( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

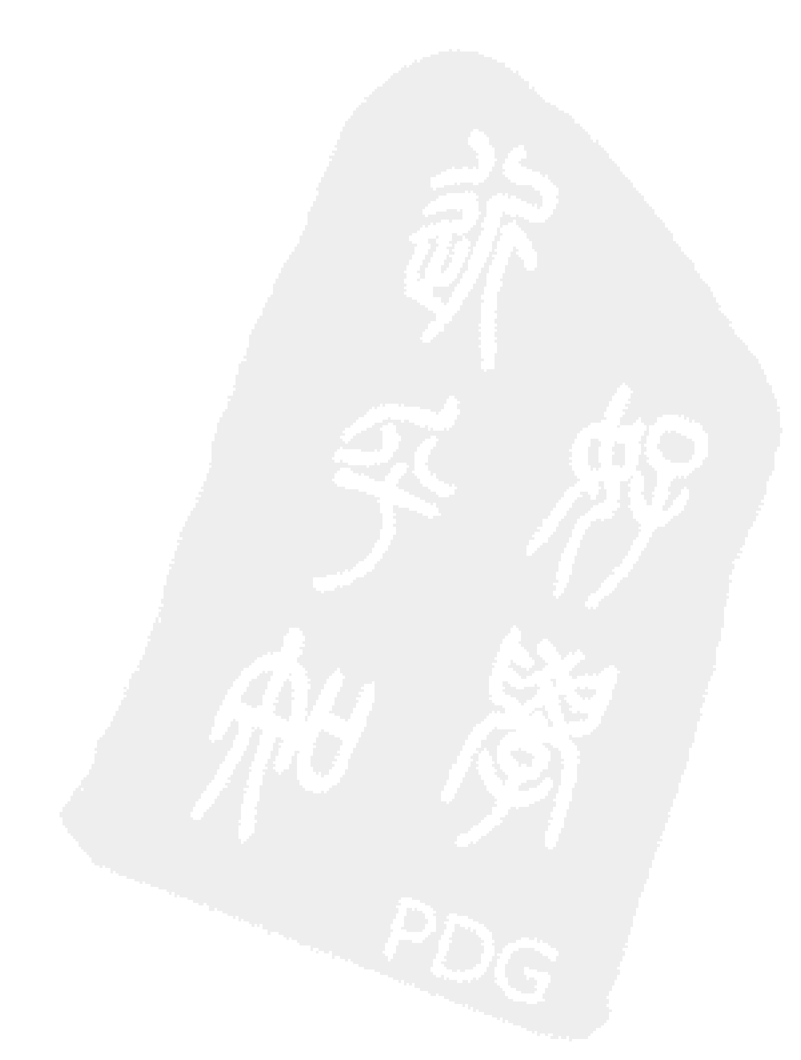
被标记为实时线程特征组(Realtime Threads Feature Group)的一部分。

Issue 6

pthread_attr_getscope( )和 pthread_attr_setscope( )函数被标记为 Threads 和 Thread Execution Scheduling 选项的一部分。

删除了[ENOSYS]错误条件，原因在于如果某种实现不支持 Thread Execution Scheduling 选项，那么就无需提供 stub。

为了与 ISO/IEC 9899:1999 标准相一致，在 pthread_attr_getschedpolicy( )原型中加入了 restrict 关键字。



名称

pthread_attr_getstack 和 pthread_attr_setstack——获取和设置 stack 属性。

调用形式

THR
```
#include <pthread.h>
```
TSA TSS
```
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
    void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

描述

pthread_attr_getstack( )和 pthread_attr_setstack( )函数分别获取和设置 attr 参数中的线程来创建 stack 属性 stackaddr 和 stacksize。

stack 属性指定了被创建的线程的栈所使用的存储区域。存储区域的基(最低可寻址字节)将是 stackaddr，存储区域的大小将为 stacksize 字节。stacksize 最少为{PTHREAD_STACK_MIN}。stackaddr 应当进行适当的对齐才能用作栈。例如，如果(stackaddr & 0x7)不为 0，则 pthread_attr_setstack( )可能会失败，并报[EINVAL]的错误。在由 stackaddr 和 stacksize 所描述的栈内的所有的页，都应可以被线程读取和写入。

返回值

如果成功，这些函数返回 0；否则，返回一个错误号以指示所产生的错误。

如果成功，pthread_attr_getstack( )函数将会把栈属性值保存在 stackaddr 和 stacksize 中。

错误

如果出现了下面的情况，pthread_attr_setstack( )函数将会失败：

[EINVAL]　stacksize 的值小于{PTHREAD_STACK_MIN}或者超出实现所定义的限制。

如果出现下面的情况，pthread_attr_getstack( )函数将可能会失败：

[EINVAL]　stackaddr 的值没有采用适当的对齐，或者{stackaddr+stacksize}缺少适当的对齐。

[EACCES]　stackaddr 和 stacksize 描述的栈页，对于线程不是既可读取又可写入的。

这两个函数均不会返回错误码[EINTR]。

实例

无。

应用程序使用

这些函数适合被应用程序用于线程的栈必须放置在内存中某些特定区域的环境下。

尽管通过在指定的栈区域外提供受保护的页，能够检测栈溢出，但是这不能够达到可移植。实现中可以自由地将线程的初始栈指针设置在指定区域的任何位置来适应机器的栈指针行为以及空间分配要求。此外，在某些特定架构下(例如 IA-64)，“溢出”可能意味着区域中分配两个单独的栈指针可能会在区域的中部发生重叠。

基本原理

无。

未来方向

无。

参见

pthread_attr_init()、pthread_attr_setdetackstate()、pthread_attr_setstacksize()、pthread_create()、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 6，作为 XSI 扩展提出，并被 IEEE PASE Interpretation 1003.1 #101 加入到 BASE。

名称

pthread_attr_getstackaddr 和 pthread_attr_setstackaddr——获取和设置 stackaddr 属性。

调用形式

THR TSA
OB

```
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
    void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

描述

pthread_attr_getstackaddr( )和 pthread_attr_setstackaddr( )函数分别获取和设置 attr 参数中的 stackaddr 属性。

stackaddr 属性指定了被创建线程的栈所使用的存储区位置。存储区的大小至少为{PTHREAD_STACK_MIN}。

返回值

如果成功完成，pthread_attr_getstackaddr( )和 pthread_attr_setstackaddr( )函数返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果成功完成，pthread_attr_getstackaddr( )将会把 stackaddr 属性的值保存到 stackaddr 中。

错误

没有定义错误。

这些函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

stackaddr 属性的规范存在几点不明确的地方，使得这些接口不能够实现可移植。将一个地址参数描述为“栈”并没有在地址同该地址暗示的“栈”之间指定特定的关系。例如，该地址可能会被当作将用作栈的缓冲区的低内存地址，或者可能会被用作新的线程的初始栈指针的寄存器值。对于计算机来讲，它们是不相同，一种是栈从低内存向高内存“向上生长”，另一种则是首先通过“入栈”操作将值保存到内存中，然后增加栈指针寄存器。此外，在栈从高地址内存向低地址内存“向下生长”的计算机中，将地址解释为“低内存”地址要求对栈的预期大小进行判断。IEEE Std 1003.1-2001 已经引入了新的接口 pthread_attr_setstack( )和 pthread_attr_getstack( )来解决这些不确定性。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getinheritsched( )、pthread_attr_getstack( )、pthread_attr_getstacksize( )、pthread_attr_setstack( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<limits.h>和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_attr_getstackaddr( )和 pthread_attr_setstackaddr( )函数被标记为 Threads 和 Thread Stack Address Attribute 选项的一部分。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_attr_getschedpolicy( )原型中加入了 restrict 关键字。

这些函数已经被标记为过时的。

名称

pthread_attr_getstacksize 和 pthread_attr_setstacksize——获取和设置 stacksize 属性。

调用形式

```
THR TSA #include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
    size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

描述

pthread_attr_getstacksize( )和 pthread_attr_setstacksize( )函数分别获取和设置 attr 参数中的 stacksize 属性。

stackaddr 属性定义了为被创建线程的栈分配的最小栈大小(以字节为单位)。

返回值

如果成功完成，pthread_attr_getstacksize( )和 pthread_attr_setstacksize( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

如果成功完成，pthread_attr_getstacksize( )将会把 stacksize 属性的值保存到 stacksize 中。

错误

如果发生如下情况，pthread_attr_setstacksize( )函数将会失败：

[EINVAL]　stacksize 的值小于{PTHREAD_STACK_MIN}或超过系统使用的限制。

这些函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_attr_destroy( )、pthread_attr_getstacksize( )、pthread_attr_getdetachstate( )、pthread_create( )、the Base Definitions volume of IEEE Std 1003.1-2001、<limits.h>和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_attr_getstacksize( )和 pthread_attr_setstacksize( )函数被标记为 Threads 和 Thread Stack Address Attribute 选项的一部分。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_attr_getschedpolicy( )原型中加入了 restrict 关键字。

名称

pthread_attr_init——初始化线程属性对象。

调用形式

THR

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

描述

参见 pthread_attr_destroy( )。

名称

pthread_attr_setdetachstate——设置 detachstate 属性。

调用形式

THR

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

描述

参见 pthread_attr_getdetachstate( )。

名称

pthread_attr_setguardsize——设置线程 guardsize 属性。

调用形式

XSI

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

描述

参见 pthread_attr_getguardsize( )。

名称

pthread_attr_setinherisched——设置 inheritsched 属性(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

描述

参见 pthread_attr_getinheritsched( )。

名称

pthread_attr_setschedparam——设置 schedparam 属性。

调用形式

THR

```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

描述

参见 pthread_attr_getschedparam( )。

名称

pthread_attr_setschedpolicy——设置 schedpolicy 属性(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

描述

参见 pthread_attr_getschedpolicy( )。

名称

pthread_attr_setscope——设置 contentionscope 属性(实时线程)。

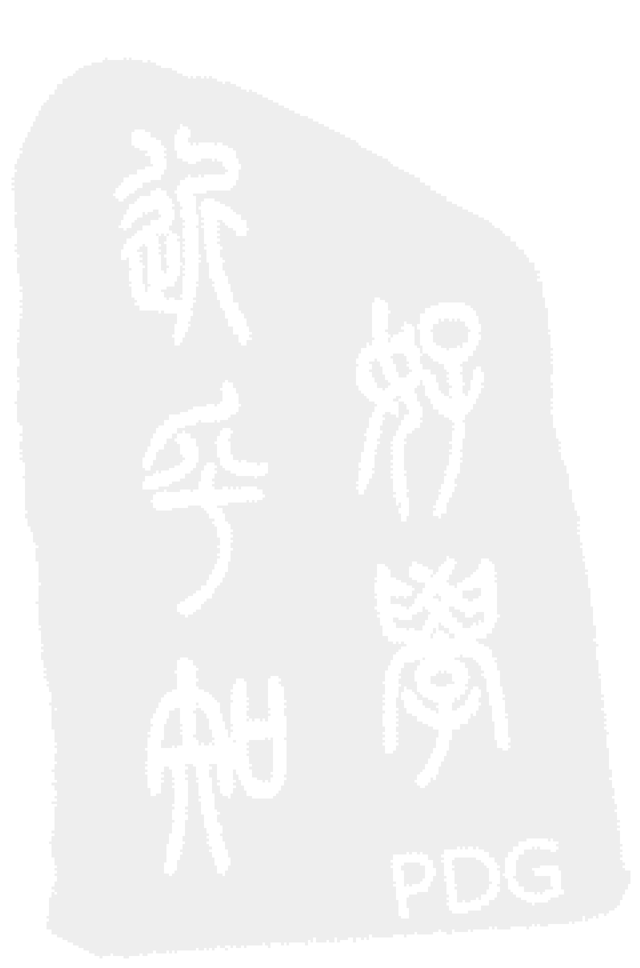
调用形式

THR TPS

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

描述

参见 pthread_attr_getscope( )。



名称

pthread_attr_setstack——设置 stack 属性。

调用形式

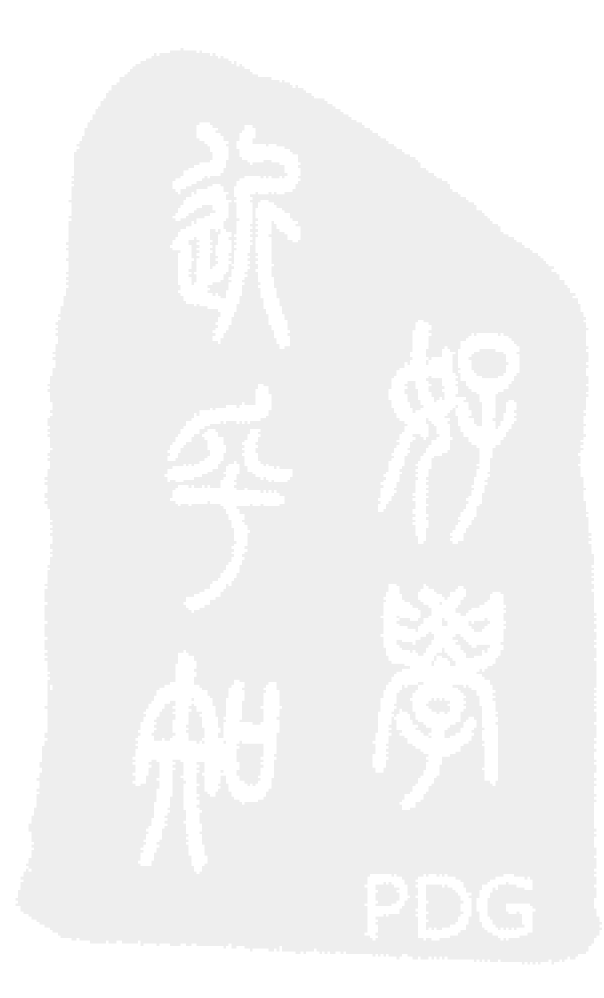
XSI

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

描述

参见 pthread_attr_getstack( )。



名称

pthread_attr_setstackaddr——设置 stackaddr 属性。

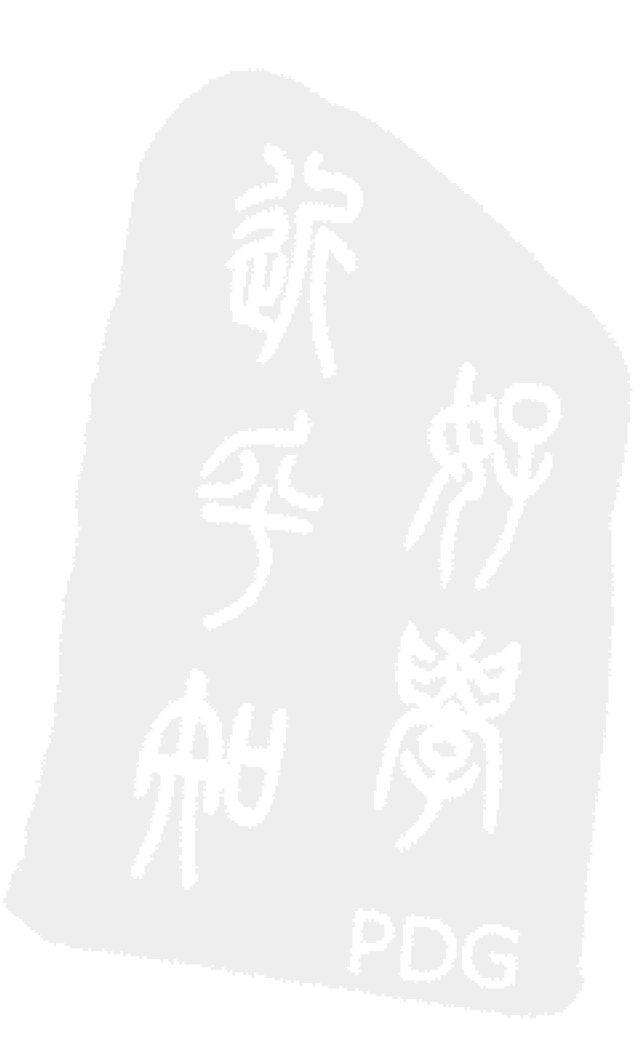
调用形式

THR TSA
OB

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

描述

参见 pthread_attr_getstackaddr( )。



名称

pthread_attr_setstacksize——设置 stacksize 属性。

调用形式

THR TSA
```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

描述

参见 pthread_attr_getstacksize( )。

名称

pthread_cancel——取消线程的执行。

调用形式

THR

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

描述

pthread_cancel( )函数请求取消 thread。目标线程的可取消状态及类型决定了取消何时会生效。当取消被执行时，就会调用 thread 的取消清理处理程序。当最后一个取消清理处理程序返回时，就会对 thread 调用线程特定的数据析构函数。当最后一个析构函数返回时，thread 就会被终止。

目标线程中的取消处理将与 pthread_cancel( )的调用线程一起异步运行。

返回值

如果调用成功，pthread_cancel( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_cancel( )函数可能会失败：

[ESRCH]　根据指定的线程 ID 找不到任何对应线程。

pthread_cancel( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

可以考虑使用两个函数之一向线程发送取消通知。其中一个函数将定义在发送时具有取消语义的新信号 SIGCANCEL；另一函数定义新的 pthread_cancel( )函数来触发取消语义。

使用新信号的优势在于，当试图发送取消通知的信号时所使用的发送标准同很多信号发送标准是一致的。确实，在很多实现中都使用了特殊信号来实现取消。另一方面，除了 pthread_kill( )函数之外就没有任何其他的函数使用这个信号了，并且所发送的取消信号的行为不同于任何先前已经定义的信号。

使用特殊函数的好处包括承认由于相似的发送标准而对该信号加以定义、承认这是取消请求和信号之间唯一的共同行为。此外，取消发送机制也可以不必作为一个信号来实现。还存在强于或等同于信号机制的语言异常机制，但如果发送机制明显接近于信号的话，那些机制潜在地会被遮蔽。

最后，考虑到将新信号与已有的信号函数一起有这么多地例外，这样做会有误导性，

就采用了一个专用函数来解决这个问题。这个函数进行了仔细地定义，以提供期望功能强于信号的实现。专用函数还意味着在实现中，不必通过信号来实现取消。

未来方向

无。

参见

pthread_exit( )、pthread_cond_timedwait( )、pthread_join( )、pthread_setcancelstate( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包括进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_cancel( )函数被标记为 Threads 选项的一部分。

名称

pthread_cond_broadcast 和 pthread_cond_signal——广播或者用信号发送某一条件。

调用形式

THR

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

描述

这两个函数会对在某条件变量上阻塞的线程解除阻塞。

pthread_cond_broadcast( )函数会对所有目前阻塞在指定条件变量 *cond* 的线程解除阻塞。

pthread_cond_signal( )函数会为至少一个受指定条件变量 *cond* 阻塞的线程(如果有线程阻塞在 *cond* 上的话)解除阻塞。

如果有多个线程阻塞在某个条件变量上，那么调度策略就会决定对这些线程解除阻塞的顺序。当线程由于 pthread_cond_broadcast( )或者 pthread_cond_signal( )而解除了阻塞，并从 pthread_cond_wait( )或者 pthread_cond_timedwait( )的调用返回时，线程就会获得其调用 pthread_cond_wait( )或者 pthread_cond_timedwait( )时所使用的互斥量。根据调度策略，解除阻塞的线程会对互斥量展开竞争，就如同每个线程都调用了 pthread_mutex_lock( )。

无论线程当前是否拥有所需的互斥量(该互斥量已经被调用 pthread_cond_wait( )或 pthread_cond_timedwait( )的线程在等待时同条件变量关联)，都可以调用 pthread_cond_broadcast( )或 pthread_cond_signal( )函数；不过，如果要求调度行为可预测，那么调用 pthread_cond_broadcast( )或 pthread_cond_signal( )的线程就会对该互斥量加锁。

如果当前没有线程阻塞于 cond，那么 pthread_cond_broadcast( )和 pthread_cond_signal( )函数就不会产生任何影响。

返回值

如果成功，pthread_cond_broadcast( )和 pthread_cond_signal( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_cond_broadcast( )和 pthread_cond_signal( )函数可能会失败：

[EINVAL] *cond* 的值没有指向一个已初始化的条件变量。

这些函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无论何时，只要共享变量的状态是以这样一种方式改变的，即多个线程可以继续执行其任

务，那么就使用 pthread_cond_broadcast( )函数。考虑单生产者/多消费者(single producer/multiple consumer)问题，在该问题中，生产者会将多项产品插入到一个列表中，该列表只能由消费者每次访问一项产品。通过调用 pthread_cond_broadcast( )函数，生产者就可以向所有可能正在等待的消费者发出通知，从而应用程序就可以在多处理器上获得更大的吞吐量。此外，pthread_cond_broadcast( )使得读-写锁易于实现。为了在写操作者释放其写锁时，可以唤醒所有等待中的读操作者，就需要调用 pthread_cond_broadcast( )函数。最后，两阶段提交算法也可以使用广播函数来通知所有即将进行事务提交的线程。

在以异步方式调用的信号处理程序中使用 pthread_cond_signal( )并不安全。即便这种使用是安全的，在测试 Boolean pthread_cond_wait( )时仍然存在着竞争，并且无法有效地消除。

互斥量和条件变量并不适合于从运行在信号处理程序中的代码发送信号来释放一个等待中的线程。

基本原理

通过条件信号实现多个唤醒

在多处理器系统中，pthread_cond_signal( )的实现可能无法避免对多个阻塞在某条件变量上的线程解除阻塞。比如，考虑下面给出的对 pthread_cond_broadcast( )和 pthread_cond_signal( )的部分实现，它由两个线程按照给定顺序执行。其中一个线程试图等待条件变量，而另一个线程正在并发地执行 pthread_cond_signal( )，与此同时第三个线程已经在等待了。

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

效果是作为 pthread_cond_signal( )调用的结果，多个线程可以从对 pthread_cond_wait( )或者 pthread_cond_timedwait( )的调用中返回。这种效果被称作“假唤醒”。注意这种情况是进行自动修正的，即被唤醒的线程数是有限的；例如，在这一系列事件阻塞的之后，下一个调用 pthread_cond_wait( )的线程。

尽管这种问题是可以解决的，但是为了极少会发生的一种边缘情况而舍弃效率显然是

不可接受的，尤其是不管怎样都要去检查与条件变量相关联的谓词。对该问题的纠正会没有必要地在这个基本构建块中降低更高层同步操作的并发度。

允许假唤醒发生的附加收益是应用程序被迫在条件等待附近加入谓词测试循环的代码。这也使得应用程序能够容纳很多条件广播或相同条件变量上的多个信号，它们可以在应用程序的其他部分编码。因此使得应用程序更加健壮。因此，IEEE Std 1003.1-2001 显式地记录了可能发生伪唤醒。

未来方向

无。

参见

pthread_cond_destroy( )、pthread_cond_timedwait( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_cond_broadcast( )和 pthread_cond_signal( )函数被标记为 Threads 选项的一部分。

添加了“应用程序使用”部分。

名称

pthread_cond_destroy 和 pthread_cond_init——销毁和初始化条件变量。

调用形式

THR

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

描述

pthread_cond_destroy( )函数会销毁由 *cond* 所指定的条件变量；实际上这个对象会变成未初始化的。实现可能会导致 pthread_cond_destroy( )将 *cond* 所指向的对象设置成无效值。可以使用 pthread_cond_init( )对已经被销毁的条件变量对象重新初始化；否则在对象被销毁之后，对其进行引用的结果是未定义的。

销毁当前没有任何线程受其阻塞的已初始化的条件变量是安全的。试图销毁一个当前尚有其他线程受其阻塞的条件变量会出现未定义的行为。

pthread_cond_init( )函数会用 attr 指向的属性对 *cond* 指向的条件变量进行初始化。如果 attr 为 NULL，那么就会使用默认的条件变量属性；其结果与将一个默认条件变量属性对象的地址传递过去相同。每当成功进行初始化，条件变量的状态就会变成已初始化。

只有 *cond* 本身可被用于执行同步。如果在对 pthread_cond_wait( )、pthread_cond_timedwait( )、pthread_cond_signal( )、pthread_cond_broadcast( )和 pthread_cond_destroy( )的调用中引用了 *cond* 的副本，其结果是未定义的。

试图对一个已初始化的条件变量进行初始化会导致未定义的行为产生。

在适合于使用默认条件变量属性的情况下，可以使用 PTHREAD_COND_INITIALIZER 宏对静态分配的条件变量进行初始化。除了不会进行错误检查之外，其效果等同于使用指定为 NULL 的参数 attr 调用 pthread_cond_init( )而进行的动态初始化。

返回值

如果成功，pthread_cond_destroy( )和 pthread_cond_init( )函数都会返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果实现了[EBUSY]和[EINVAL]错误检查的话，看上去就好像在函数处理之初就立即执行了它们一样，并且会在修改 *cond* 所指定的条件变量状态之前返回一个错误。

错误

如果出现了下面的情况，pthread_cond_destroy( )函数可能会失败：

[EBUSY]实现检测到在 cond 所引用的对象正处于被引用之中(比如，正为另一个线程中的 pthread_cond_wait( )或者 pthread_cond_timedwait( )所使用)，出现了试图对其进行销毁的尝试。

[EINVAL] *cond* 指定的值无效。

如果出现了下述情况，pthread_cond_init( )函数将会失败：

[EAGAIN] 系统缺少必需的资源(内存之外的其他资源)初始化另一个条件变量。

[ENOMEM] 缺乏足够的内存对条件变量进行初始化。

如果出现了下面的情况，pthread_cond_init( )函数可能将会失败：

[EBUSY]实现检测到了试图对 cond 所引用的一个先前已初始化的、并且尚未销毁的对象进行重新初始化的尝试。

[EINVAL] attr 所指定的值无效。

这两个函数均不会返回错误码[EINTR]。

示例

当所有阻塞于某条件变量的线程都被唤醒之后，就可以立即销毁该条件变量了。例如，请看下面的代码：

```
struct list {
    pthread_mutex_t lm;
    ...
}
struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
}
/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;
    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}
delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0;  /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&rp->notbusy);
    free(ep);
}
```

在这个例子中，一旦所有等待条件变量的线程(A 行)都被唤醒了，那么就可以立即释放条件变量及其列表成员(B 行)了，因为互斥量和代码可以确保没有其他的线程能够触及到即将被销毁的成员。

应用程序使用
无。

基本原理
见 pthread_mutex_init( )；相似的基本原理也可以应用于条件变量。

未来方向
无。

参见
pthread_cond_broadcast( )、pthread_cond_signal( )、pthread_cond_timedwait( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史
首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6
pthread_cond_destroy( )及 pthread_cond_init( )函数被标记为 Threads 选项的一部分。

应用了 IEEE PASC Interpretation 1003.1c #34，更新了“描述”部分。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_cond_init( )原型中加入了 restrict 关键字。

名称

pthread_cond_signal——为某情形发送信号。

调用形式

THR

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

描述

参见 pthread_cond_broadcast( )。

名称

pthread_cond_timedwait 和 pthread_cond_wait——等待一个条件。

调用形式

THR

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

描述

pthread_cond_timedwait( )和 pthread_cond_wait( )函数会在一个条件变量上阻塞。调用时需要使用被调用线程加锁的 mutex，否则会导致未定义的行为。

这两个函数原子性地释放 mutex，并导致调用线程阻塞在条件变量 *cond* 上；这里的"原子性地"意味着"相对于另一个线程依次对互斥量及条件变量的访问而言是原子性的"。也就是说，如果在即将阻塞的线程已经释放了互斥量之后，另一个线程能够获得它，那么在这个线程中后续地对 pthread_cond_broadcast( )或者 pthread_cond_signal( )调用，其行为看上去就好像该即将阻塞的线程已经阻塞一样。

当成功返回时，互斥量会被加锁，并且会被调用线程所拥有。

在使用条件变量的时候，总会有一个 Boolean 谓词，它涉及共享的变量，该变量会同每个条件等待关联，如果线程应当继续，则该谓词为真。来自于 pthread_cond_timedwait( )或者 pthread_cond_wait( )的假唤醒可能会大量地发生。由于 pthread_cond_timedwait( )或者 pthread_cond_wait( )的返回值并没有给出任何相关于该谓词的值的提示，所以在每次返回之时都应当对谓词进行重新评估。

对相同条件变量上并发的 pthread_cond_timedwait( )和 pthread_cond_wait( )操作使用多个互斥量，其效果是未定义的；也就是说，当某个线程在等待某个条件变量时，该条件变量就会与唯一的互斥量绑定，并且当等待返回时，这种(动态的)绑定就会结束。

条件等待(无论是否是定时的)就是一个取消点。当线程的可取消性使能状态被设置为 PTHREAD_CANCEL_DEFERRED 时，在条件等待时对一次取消请求进行响应的副效应是在调用第一个取消清理处理程序之前重新获得互斥量(从实际效果上来看)。这种效果看上去就好像是线程被解除了阻塞一样，并且允许执行到 pthread_cond_timedwait( )或者 pthread_cond_wait( )调用的返回处，只是在该点会注意到取消请求，然后就启动线程取消活动(包括调用取消清理处理程序)，而不是返回到 pthread_cond_timedwait( )或者 pthread_cond_wait( )的调用者。

当阻塞于 pthread_cond_timedwait( )或者 pthread_cond_wait( )调用中的线程由于取消操作而解除了阻塞时，如果还有其他的线程阻塞于条件变量，则不会消耗并发地指向条件变量的条件信号。

pthread_cond_timedwait( )函数等价于 pthread_cond_wait( )，唯一的不同在于如果在条件 cond 发送信号或者广播之前，由 abstime 所指定的绝对时间已经过了(也就是说，系统时

间等于或者超过了 abstime)，或者如果 abstime 所指定的绝对时间已经在调用之时过了，那么就会返回一个错误。

如果支持 Clock Selection 选项的话，条件变量就会拥有一个 clock 属性，以指定在度量参数 abstime 所指定的时间时将会用到的时钟。当此类超时发生时，pthread_cond_timedwait( )不管怎样都要释放并重新获得 mutex 所引用的互斥量。pthread_cond_timedwait( )函数也是一处取消点。

如果给等待条件变量的线程发送了一个信号，那么在从信号处理程序返回之时，该线程将会继续等待条件变量，看上去就好像是并未受到中断一样；或者如果是一次假唤醒，就会返回 0。

返回值

除了[ETIMEDOUT]情况之外，所有这些错误检查看上去都好像是在函数处理之初就立即执行了它们一样，并且在修改 mutex 所指定的互斥量状态或者 *cond* 所指定的条件变量之前，将会导致一次错误返回。

若成功完成，就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_cond_timedwait( )函数将会失败：

[ETIMEDOUT] 通过 abstime 指定给 pthread_cond_timedwait( )的时间已经过去。

如果出现了下面情况，pthread_cond_timedwait( )和 pthread_cond_wait( )函数可能会失败：

[EINVAL] cond、mutex、或者 abstime 指定的值无效。

[EINVAL] 在相同的条件变量上为并发的 pthread_cond_timedwait( )或者 pthread_cond_wait( )操作提供了不同的互斥量。

[EPERM] 在调用之时互斥量没有为当前的线程所拥有。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

条件等待语义

注意下面这一点是很重要的：当 pthread_cond_timedwait( )和 pthread_cond_wait( )无错返回时，相关联的谓词可能仍然为假。类似地，当 pthread_cond_timedwait( )以超时错误返回时，相关的谓词也可能为真，这是由于超时过期与谓词状态修改之间不可避免的竞争而导致的。

在某些实现中，尤其是在多处理器系统中，当对不同处理器上的条件变量同步发送信号时，可能有时会导致多个线程被唤醒。

总而言之，只要条件等待返回，线程都需要重新计算与条件等待相关联的谓词，以便判断是可以安全地继续执行、再次等待，还是声明一次超时。从等待中的返回并不意味着相关联的谓词为真或为假。

因此建议将条件等待置于一个等效于“while 循环”的结构中对谓词进行检查。

定时等待语义

出于两方面的考虑在指定超时参数时选择了绝对时间度量：首先，在指定了绝对时间的函数之上很容易实现相对时间度量，但是在指定了相对超时的函数之上指定一个绝对超时就会出现一个与之相关的竞争条件。例如，假定 clock_gettime( )会返回当前的时间，并且 cond_relative_timed_wait( )使用了相对超时：

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

如果在第一条语句和最后一条语句之间线程被抢占了，那么该线程被阻塞的时间就太长了。如果使用了绝对超时，阻塞就无关紧要了。如果在一次循环中(比如包含了一次条件等待的循环)多次使用了定时操作，那么使用绝对超时将无需进行重新计算。

对于系统时钟被操作者偶尔提前的情况，期望实现在处理在此期间超时的定时等待就好像时间真的过去了一样。

取消和条件等待

条件等待，无论是否被定时，都是一次取消点。即函数 pthread_cond_timedwait( )或者 pthread_cond_wait( )均注意未决(或者并发)取消请求的位置。其原因在于在这些地方无限等待是不可能的——即便程序完全正确，不论等待什么样的事件，也可能永远不会发生；例如，一些等待的输入数据可能永远也不会被送过来。通过将条件等待设成一处取消点，虽然它仍然可能会被困在某个无穷等待中，但是能够取消掉这样的线程并执行其取消清理处理程序。

如果某个线程阻塞于某个条件变量，则此时发起取消请求的副作用是在调用任何取消清理处理程序之前，要重新获得互斥量。这样做的目的在于确保执行取消清理处理程序时的状态能够与条件等待函数调用前后的关键代码的状态相同。在从诸如 Ada 或者 C++之类的语言(在这些语言中，可能会选择将取消映射到某种语言异常)转到 POSIX 线程接口时，这条规则也是要求的。该规则可以确保每个对临界区进行保护的异常处理程序总是能够安全地依赖于相关互斥量已经被加锁的事实，而不必关心在临界区的什么位置发生的异常。没有这条规则，关于锁的异常处理程序就没有一条统一的规则可以遵循了，从而就会导致编程上的麻烦。

因此，既然需要给出一些语句以声明一次等待过程中发送取消时的锁状态，就需要选择一种定义以便应用程序编程可以简便而且无误。

在某个线程被某个条件变量阻塞期间，当处理取消请求时，就会要求实现来确保线程不会消耗任何这样的条件信号，即该信号指向的条件变量是有任何其他线程正在等待的条件变量。之所以指定这条规则是为了避免死锁情形，如果没有独立地处理这两个独立的请求(一个作用于线程，而另一个作用于条件变量)，则可能会发生死锁。

互斥量和条件变量的性能

我们会希望互斥量只为较少的指令加锁。出于程序员们对避免执行较长串行区域(这样做将会降低总体有效并行性)的渴望，这一实践几乎自动受到支持。

在使用互斥量和条件变量时，人们会试图确保经常发生的情况是对互斥量加锁、访问共享数据和对互斥量解锁。在一个条件变量上等待应当是一种相对较为罕见的情况。例如，在实现读-写锁时，获取读锁的代码往往只需要将读操作者的数目加一(在互斥现象之下)然后返回。只有已经存在着一个活跃的写入线程时，调用线程才会在条件变量上等待。因此，同步操作的效率是与互斥量的加锁/解锁代价绑定在一起的，而不是与条件等待绑定在一起。注意在一般情况下没有上下文切换。

这并不是说条件等待的效率就不重要了。由于每次 Ada 集合(rendezvous)都至少需要进行一次上下文切换，因此在条件变量上等待的效率也很重要。在条件变量上等待的开销应当略大于进行上下文切换的最低开销加上对互斥量进行加锁和解锁操作所需要的时间。

互斥量和条件变量的特性

曾经有人提出过这样的建议：将获取和释放互斥量的部分从条件等待中分离出去。之所以会拒绝掉这项建议，原因在于实际上正是操作本身所具有的这种结合性的特性便利了实时实现。这些实现可以原子地在条件变量和互斥量间移动高优先级的线程，并且这种移动对调用者而言是透明的。这样就可以避免额外的上下文切换，并且在发送信号给等待中的线程时，提供了对互斥量进行更具确定性的获取能力。如此，就可以将公平性和优先级问题直接交给调度规则来处理了。而且，当前的条件等待操作与现有做法是匹配的。

互斥量和条件变量的调度行为

试图通过指定顺序规则来与调度策略进行交互的同步原语被认为是不合适的。在互斥量和条件变量上等待的多个线程，选择它们继续执行的顺序应当取决于调度策略而不应当按照某种固定的次序(例如，FIFO 或者优先级)。这样，调度策略决定了究竟应该唤醒和允许继续执行哪个线程。

定时条件等待

pthread_cond_timedwait( )函数允许应用程序在给定数量的时间之后放弃对某个特定条件的等待。下面给出的这个例子演示了它的用法：

```
(void) pthread_mutex_lock(&t.mn);
        t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

如果将超时参数设置为绝对时间，就无需在每次程序检查阻塞谓词时对其进行重新计算。如果超时是相对的，则必须在每次调用之前进行重新计算。这样将会特别困难，因为

这种代码需要考虑在谓词为真或超时之前，由于额外的广播或者由条件变量上所发送的信号而导致出现额外唤醒。

未来方向

无。

参见

pthread_cond_signal( )、pthread_cond_broadcast( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_cond_timedwait( )和 pthread_cond_wait( )函数被标记为 Threads 选项的一部分。

应用了 Open Group Corrigendum U021/9，修正了 pthread_cond_wait( )函数的原型。

为了与 IEEE 1003.1j-2000 标准一致，对“描述”部分进行了更新：添加了 Clock Selection 选项的语义。

为了响应 IEEE PASC Interpretation 1003.1c #28，为“错误”部分添加了[EPERM]的一种条件。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_cond_timedwait( )和 pthread_cond_wait( )原型中加入了 restrict 关键字。

名称

pthread_condattr_destroy 和 pthread_condattr_init——销毁和初始化条件变量属性对象。

调用形式

THR

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

描述

pthread_condattr_destroy( )函数会销毁一个条件变量属性对象；从实际效果上，这个对象会变成未初始化的。实现可能会导致 pthread_condattr_destroy( )将 attr 所引用的对象设置成无效值。可以使用 pthread_condattr_init( )对一个已经销毁的属性对象进行重新初始化；在属性对象已经销毁之后，对其进行引用的结果是未定义的。

pthread_condattr_init( )函数会使用由实现为全部属性定义的默认值来对条件变量属性对象 attr 进行初始化。

如果使用一个已初始化的 attr 属性对象调用 pthread_condattr_init()，其结果是未定义的。

当一个条件变量属性对象已经用于初始化一个或者多个条件变量之后，任何会影响到属性对象的函数(包括析构函数)都不会影响到任何先前已经初始化过的条件变量。

IEEE Std 1003.1-2001 的这一卷要求提供两种属性：clock 属性和 process-shared 属性。

其他的属性，它们的默认值以及用于获取和设置这些属性值的相关函数名都是由具体的实现定义的。

返回值

如果成功，pthread_condattr_destroy( )和 pthread_condattr_init( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_condattr_destroy( )函数可能会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，pthread_condattr_init( )函数就会失败：

[ENOMEM] 初始化条件变量属性对象时内存不足。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

请参阅 pthread_attr_init( )和 pthread_mutex_init( )。

为条件变量定义了一个 process-shared 属性，对该属性进行定义的原因与对互斥量进行相应属性的定义的原因相同。

未来方向

无。

参见

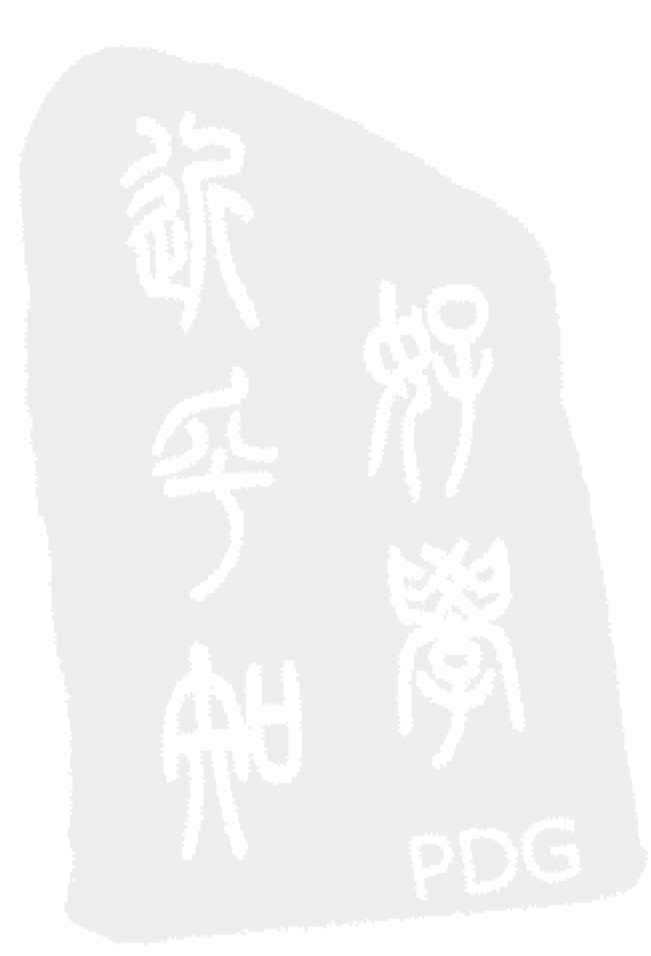
pthread_attr_destroy( )、pthread_cond_destroy( )、pthread_condattr_getpshared( )、pthread_create( )、pthread_mutex_destroy( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包括进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_condattr_destroy( )和 pthread_condattr_init( )函数被标记为 Threads 选项的一部分。



名称

pthread_condattr_getclock 和 pthread_condattr_setclock——获取和设置 clock selection 条件变量属性(高级实时)。

调用形式

THR CS
```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

描述

pthread_condattr_getclock( )函数会从 attr 指向的属性对象中得到 clock 属性的值。pthread_condattr_setclock( )函数将设置 attr 指向的已初始化属性对象的 clock 属性。如果使用指向 CPU-time 时钟的 clock_id 参数调用 pthread_condattr_setclock( )，则调用会失败。

clock 属性是被 pthread_cond_timedwait( )用来度量超时服务的时钟的 ID。clock 属性的默认值会指向系统时钟。

返回值

如果成功，pthread_condattr_getclock( )函数返回 0，并将 attr 的时钟属性的值保存到由 clock_id 参数指向的对象。否则，返回一个错误号以指示所产生的错误。

如果成功，pthread_condattr_setclock( )函数将返回 0，否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这些函数可能会失败：

[EINVAL]　attr 指定的值无效。

如果出现了下面的情况，pthread_condattr_setclock( )函数就可能会失败：

[EINVAL]　由 clock_id 指定的值不代表一个已知的时钟，或者是一个 CPU-time 时钟。

这些函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

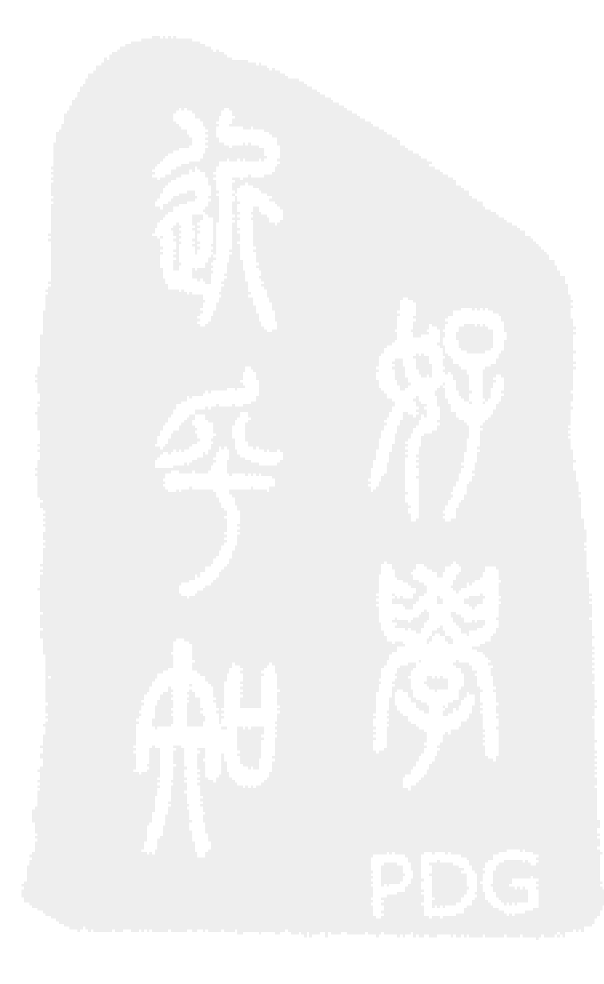
无。

基本原理

无。

未来方向

无。

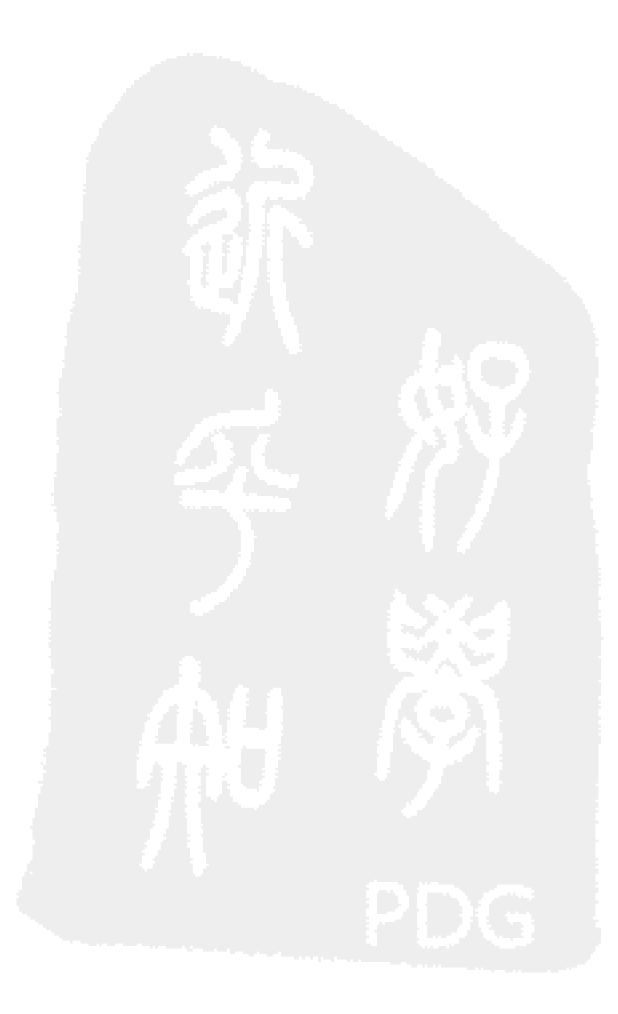


参见

pthread_cond_destroy( )、pthread_cond_timedwait( )、pthread_condattr_destroy( )、pthread_condattr_getshared( )、pthread_condattr_init( )、pthread_condattr_setshared( )、pthread_create( )、pthread_mutex_init( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 6，来自 IEEE std 1003.1j-2000。



名称

pthread_condattr_getpshared 和 pthread_condattr_setpshared——获取和设置 process-shared 条件变量属性。

调用形式

THR TSH
```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
    int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```

描述

pthread_condattr_getpshared( )函数会从 attr 所指向的已初始化属性对象中获得 process-shared 属性的值。pthread_condattr_setpshared( )函数会在 attr 所指向的一个已初始化属性对象中设置 process-shared 属性。

为了允许能够访问到条件变量分配空间中的任何线程可以对条件变量进行操作，应该将 process-shared 属性设置成 PTHREAD_PROCESS_SHARED，即便分配给该条件变量的内存区是为多个进程所共享的。如果 process-shared 属性为 PTHREAD_PROCESS_PRIVATE，那么能够对条件变量进行操作的线程将仅限于那些与对该条件变量进行初始化的线程位于相同进程中的线程；如果不同进程中的线程试图对这样一种条件变量进行操作，其行为是未定义的。process-shared 属性的默认值为 PTHREAD_PROCESS_PRIVATE。

返回值

如果成功，pthread_condattr_setpshared( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果成功，pthread_condattr_getpshared( )函数返回 0，并将 attr 的 process-shared 属性保存在 pshared 参数所指向的对象中；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_condattr_getpshared( )和 pthread_condattr_setpshared( )函数可能会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，pthread_condattr_setpshared( )函数可能就会失败：

[EINVAL] 为属性所指定的新值超出了该属性取值的合法范围。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_create( )、pthread_cond_destroy( )、pthread_condattr_destroy( )、pthread_mutex_destroy()、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_condattr_getpshared( )和 pthread_condattr_setpshared( )函数被标记为 Threads 和 Threads Process-Shared Synchronization 选项的一部分。

为了与 ISO/IEC 9899:1999 标准相一致，在 pthread_condattr_getpshared( )原型中加入了 restrict 关键字。

名称

pthread_condattr_init——初始化条件变量属性对象。

调用形式

THR

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
```

描述

参见 pthread_condattr_destroy( )。

名称

pthread_condattr_setclock——设置 clock selection 条件变量属性。

调用形式

THR CS

```
#include <pthread.h>

int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

描述

参见 pthread_condattr_getclock( )。

名称

pthread_condattr_setshared——设置 process-shared 条件变量属性。

调用形式

THR TSH

```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```

描述

参见 pthread_condattr_getshared( )。

名称

pthread_create——线程创建。

调用形式

THR

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

描述

pthread_create( )函数将使用 attr 所指定的属性，在进程中创建新的线程。如果 attr 为 NULL，就会使用默认的属性。如果后来对 attr 所指定的属性进行了修改，线程的属性也不会受到影响。成功完成时，pthread_create( )会将所创建线程的 ID 保存到 thread 所指向的位置。

所创建的线程会执行以 arg 为唯一参数的 start_routine。如果 start_routine 返回了，其效果就好像是以 start_routine 的返回值作为退出状态，对 pthread_exit( )进行了一次隐式调用。注意最初调用 main( )的线程与此不同。当它从 main( )返回时，其效果就好像使用 main( )的返回值作为退出状态，对 exit( )进行一次隐式调用一样。

对新线程的信号状态将会做如下的初始化：

- 信号的掩码将会继承自创建线程。
- 新线程的未决信号集为一个空集。

浮点环境将会继承自创建线程。

如果 pthread_create( )失败，则不会创建出任何新线程，并且 thread 所指向的地址的内容也是未定义的。

如果定义了_POSIX_THREAD_CPUTIME，那么新线程就会拥有一个可访问的 CPU 时间时钟，并且该时钟的初始值将设置为 0。

返回值

如果成功，pthread_create( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_create( )函数就会失败：

[EAGAIN] 系统缺乏足够的资源来创建另一个线程，或者将超出系统所强加的一个进程中可以存在的线程总数的限制{PTHREAD_THREADS_MAX}。

[EINVAL] attr 指定的值无效。

[EPERM] 调用者不具备适当的权限来设置所要求的调度参数或者调度策略。

pthread_create( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

对 pthread_create( )曾经提议过另一种选择，即定义两个分离的操作——创建和启动。在某些应用程序中会发现这种行为会更加自然。特别地，Ada 就将一个任务的“创建”与“激活”进行了分离。

由于许多原因，标准开发人员拒绝了这种分离的操作：

- 要求启动一个线程的调用数目会从一个增加到两个，从而对无需额外同步的应用程序施加了额外的负担。不过，第二个调用可以通过一个附加的启动状态属性而避免。
- 可以引入一个额外的状态：“创建但未启动”。这样就需要标准来指定当目标线程尚未启动执行时，线程操作的具体行为。
- 对于那些需要此类行为的应用程序，使用当前所提供的功能是可能模拟出这两个分离的步骤来的。通过等待一个由启动操作来发送信号的条件变量，就可以对 start_routine( )进行同步了。

Ada 的实现者可以选择在 Ada 程序中下面两处地方中的任意一处创建线程：创建任务对象时，或者激活任务(通常是在一个 begin 的位置处)时。如果采用了第一种方法，那么 start_routine( )就需要等待一个条件变量以接收开始“激活”的命令。而第二种方法则无需此类条件变量或者额外的同步。在这两种方法中，如果所创建的任务对象是为了保存集合点队列或者其他类似的结构，那么它们都会需要创建一个分离的 Ada 任务控制块。

对前面这种模型的一种扩展是允许线程状态在创建和启动之间进行修改。这样就会允许清除线程属性对象了。由于下面的原因，这种提议也被拒绝了：

- 应当能够为线程设置线程属性对象中的所有状态。这就要求对修改线程属性的函数进行定义，从而就不能减少建立线程所要求的函数调用的数目。实际上，对于使用相同属性创建所有线程的应用程序而言，建立线程所要求的函数调用的数目会急剧增加。对线程属性对象的使用则允许应用程序构造一个属性设置函数调用集。否则，就需要为每个线程的创建构造其属性设置函数调用集。
- 根据实现的架构，用于设置线程状态的函数可能会要求进行内核调用，或者出于其他的实现原因，而无法作为宏加以实现，这样就增加了线程创建的开销。
- 应用程序就会丧失掉按类隔离线程的能力。

在所提议的另一种可选方案中，使用了一种类似于进程创建的模型，比如 thread fork。fork 语义会提供更大的灵活性，并且仅仅通过执行一次线程 fork 后立即为线程调用适当的 start routine。不过，这种选择具有下面这些问题：

- 对于很多实现而言，需要复制调用线程的整个栈，因为很多架构都无法判定调用帧的大小。

- 由于至少需要复制栈的某些部分，效率就会被降低，即便在大多数情况下，线程永远都不会需要所复制的上下文，因为它仅仅会调用所需的启动例程。

未来方向

无。

参见

fork( )、pthread_exit( )、pthread_join( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_create( )函数被标记为 Threads 选项的一部分。

下面对 POSIX 上实现的新要求来源于和 Single UNIX Specification 一致的需要：

- 增加了[EPERM]强制错误条件

  为了与 IEEE Std 1003.1d-1999 一致，增加了线程 CPU-time 时钟语义。

  为了与 ISO/IEC 9899:1999 标准一致，在 pthread_create( )原型中加入了 restrict 关键字。

  对“描述”部分进行了更新，从而显式说明浮点环境是从创建线程继承的。

名称

pthread_detach——分离线程。

调用形式

THR

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

描述

pthread_detach( )函数会向实现进行如下指示：当该线程终止时可以将线程 thread 的存储区收回。如果 thread 尚未终止，pthread_detach( )不会导致其终止。对相同的目标线程进行多次 pthread_detach( )调用，其结果是未指定的。

返回值

如果函数调用成功，pthread_detach( )就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_detach( )函数将会失败：

[EINVAL] 实现已经检测到 thread 所指定的值没有指向一个可结合的线程。

[ESRCH] 找不到任何可以与给定线程 ID 相对应的线程。

pthread_detach( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

为了能够收回与线程相关联的存储区，最终会对每个创建线程调用 pthread_join( )或者 pthread_detach( )函数。

已经有人建议过 detach 函数是不必要的；提供一个 detachstate 线程创建属性就足够了，因为人们永远都不需要动态地去分离一个线程。不过，在下面两种情况下，就会产生相应的需求。

(1) 在 pthread_join( )的一个取消处理程序中，为了能够分离 pthread_join( )正在等待的线程，具备这样一个 pthread_detach( )函数几乎是必需的。如果没有这个函数，那么就需要让处理程序执行另一次 pthread_join( )以便试图对该线程进行分离，而这样做既会将取消处理延迟一个没有定界的时段，又会引入对 pthread_join( )的一次新调用，并且这次调用本身可能还会需要一个取消处理程序。在这种情况下，动态的分离几乎是必需的。

(2) 为了能够分离出“initial 线程”(在建立服务器线程的进程中可能会用到)。

未来方向

无。

参见

pthread_join( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_detach( )函数被标记为 Threads 选项的一部分。

名称

pthread_equal——比较线程 ID。

调用形式

THR
```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

描述

这个函数将比较线程 ID t1 和 t2。

返回值

如果 t1 和 t2 相等，则 pthread_equal( )函数将返回非零值；否则，将返回 0。

如果 t1 或 t2 不是有效的线程 ID，那么函数的行为未被定义。

错误

没有定义错误。

pthread_equal( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

实现可能会选择将线程 ID 定义为结构体。这样会比定义为 int 更具灵活性和健壮性。例如，线程 ID 可以包含序号，从而允许检测“悬挂 ID”(已经被分离的线程 ID 的副本)。由于 C 语言不支持结构体类型的比较，因此提供了 pthread_equal( )函数来比较线程 ID。

未来方向

无。

参见

pthread_create( )、pthread_self( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_equal( )函数被标记为 Threads 选项的一部分。

名称

pthread_exit——线程终止。

调用形式

THR

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

描述

pthread_exit( )函数会终止调用线程，并且使得 value_ptr 值可用于任何与终止线程成功结合的线程。所有被推入但是尚未被弹出的取消清理处理程序都会按照它们被推入的相反次序被弹出，然后加以执行。在执行完所有的取消清理处理程序之后，如果线程还持有一些特定于该线程的数据，那么就会以一种未指定的次序调用适当的析构函数。线程的终止并不会释放任何应用程序可见的进程资源，包括(但不局限于)互斥量和文件描述符，它也不会执行任何进程级清理动作，包括(但不局限于)调用任何可能存在的 atexit( )例程。

当一个线程(只要它不是首次调用 main( )的线程)从用于创建它的启动例程返回时，就会出现一次对 pthread_exit( )的隐式调用。函数的返回值将会用作线程的退出状态。

如果从取消清理处理程序或者析构函数中(这个析构函数是作为一次显式或者隐式的 pthread_exit( )调用的结果而被调用的)调用 pthread_exit( )，那么，pthread_exit( )的行为是未定义的。

在一个线程已经终止之后，对线程的本地(auto)变量进行访问，其结果是未定义的。因此，就不应该将退出线程的本地变量的指针作为 pthread_exit( )的 value_ptr 参数值。

在最后一个线程已经终止之后，进程就会以退出状态 0 退出。这种行为看上去就好像是实现在线程终止之时，以 0 作为参数调用 exit( )。

返回值

pthread_exit( )函数无法返回到其调用者。

错误

没有定义任何错误。

示例

无。

应用程序使用

无。

基本原理

线程终止的正常机制为：从启动线程的 pthread_create( )调用中所指定的例程返回。pthread_exit( )函数提供了一种不必要求从该线程的启动例程返回而终止一个线程的能力，从而提供了一个类似于 exit( )的函数。

无论线程终止的方法怎样，任何已经被推入但尚未被弹出的取消清理处理程序都会被

执行，并且任何已有的线程所特定的数据的析构函数也都会被加以执行。在 IEEE Std 1003.1-2001 的这一卷中，要求取消清理处理程序依次被弹出和调用。在执行完所有的取消清理处理程序之后，线程所特定的数据析构函数就会被调用，对于存在于线程中的为具体线程所特定的每项数据而言，其调用的顺序是未指定的。由于取消清理处理程序可能会依赖于线程所特定的数据，这种顺序其实是有必要的。

由于状态的意义是由应用程序所决定的(除了线程已经被取消的时候，在这种情况下其状态为 PTHREAD_CANCELED)，实现并不知道合法的状态值是什么，这也就是没有进行任何地址错误检查的原因所在。

未来方向

无。

参见

exit( )、pthread_create( )、pthread_join( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_exit( )函数被标记为 Threads 选项的一部分。

名称

pthread_getconcurrency 和 pthread_setconcurrency——获取和设置并发级别。

调用形式

XSI

```
#include <pthread.h>

int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

描述

在一个进程中未被绑定的线程可能会也可能不会被要求同时处于激活状态。在默认情况下，线程实现确保足够数目的线程是处于激活状态的，以便进程能够继续向前执行。尽管这种做法会节约系统的资源，但是可能不会获得最高效的并发级别。

pthread_setconcurrency( )函数允许一个应用程序将其所期望的并发级别 new_level，通知给线程的实现。作为该次函数调用的结果而由实现所提供的实际并发级别是未指定的。

如果 new_level 为 0，就会导致实现根据其自身的判断来维持并发级别，看上去就好像是从未调用过 pthread_setconcurrency( )一样。

pthread_getconcurrency( )函数会返回前一次 pthread_setconcurrency( )函数调用所设置的取值。如果先前没有调用 pthread_setconcurrency( )函数，那么该函数就会返回 0，以指示由实现维护并发级别。

对 pthread_setconcurrency( )的一次调用会将它所期望的并发级别告知具体实现。不过，具体的实现会使用它作为一种提示，而不是一种要求。

如果某种实现不支持位于几个内核调度实体之上的用户线程复用的话，那么出于源代码兼容性的考虑，它们都会提供 pthread_setconcurrency( )和 pthread_getconcurrency( )函数，但是实际上在调用这两个函数的时候并不会产生任何效果。为了保持函数的语义，在调用 pthread_setconcurrency( )函数时，也对 new_level 参数加以保存，以便后续 pthread_getconcurrency( )调用能够返回相同的数值。

返回值

如果成功，pthread_setconcurrency( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

pthread_getconcurrency( )函数总是会返回前面一次在 pthread_setconcurrency( )调用中所设置的并发级别。如果从未调用过 pthread_setconcurrency( )函数，pthread_getconcurrency( )就会返回 0。

错误

如果出现下面的情况，pthread_setconcurrency( )函数就会失败：

[EINVAL] new_level 指定的值无效。

[EAGAIN] new_level 指定的值将导致超出系统资源。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

使用这两个函数会改变应用程序所依赖的底层并发状态。建议库开发人员不要使用 pthread_getconcurrency( )和 pthread_setconcurrency( )函数，因为它们的使用可能会与应用程序对这些函数的使用冲突。

基本原理

无。

未来方向

无。

参见

The Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

名称

pthread_getcpuclockid——访问线程 CPU-time 时钟(高级实时线程)。

调用形式

THR TCT
```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

描述

pthread_getcpuclockid( )函数将在 clock_id 中返回由 thread_id 指定的线程的 CPU-time 时钟的时钟 id，前提是 thread_id 指定的线程存在。

返回值

如果成功完成，pthread_getcpuclockid( )返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现下面的情况，pthread_setconcurrency( )函数就可能会失败：

[ESRCH] thread_id 指定的值不代表一个存在的线程。

示例

无。

应用程序使用

pthread_getcpuclockid( )函数是 Thread CPU-Time Clocks 选项的一部分，所有的实现都需要提供。

基本原理

无。

未来方向

无。

参见

clock_getcpuclockid( )、clock_getres( )、timer_create( )、The Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<time.h>。

变更历史

首次发布于 Issue 6。来自于 IEEE Std 1003.1d-1999。

在调用形式中，不再需要包含<sys/types.h>。

名称

pthread_getschedparam 和 pthread_setschedparam——动态线程调度参数访问(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
    struct sched_param *restrict param);
int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

描述

pthread_getschedparam( )和 pthread_setschedparam( )函数将分别获取和设置将要被提取和设置的多线程进程中的单个线程的调度策略和调度参数。对于 SCHED_FIFO 和 SCHED_RR 策略，sched_param 数据结构唯一要求的成员为优先级 sched_priority；而对于 SCHED_OTHER，受影响的调度参数由实现定义。

pthread_getschedparam( )函数会从线程 ID 为 thread 的线程中提取调度策略和调度参数，并分别将它们保存在 policy 和 param 中。由 pthread_getschedparam( )所返回的优先级的值，将是最近一次影响到目标线程的由 pthread_setschedparam( )、pthread_setschedprio( )、或者 pthread_create( )调用指定的值。它不会反映出由于优先级继承性或者上限设置函数而对优先级进行的任何临时性调整。pthread_setschedparam( )函数分别用 policy 和 param 所提供的策略及相关参数，对线程 ID 为 thread 的线程进行调度策略和相关调度参数的设置。

policy 参数的可能值有 SCHED_OTHER、SCHED_FIFO 及 SCHED_RR。SCHED_OTHER 策略的调度参数由实现定义。SCHED_FIFO 和 SCHED_RR 策略将只有一个单独的调度参数 priority。

如果定义了_POSIX_THREAD_SPORADIC_SERVER，那么 policy 参数的值可能为 SCHED_SPORADIC，除非在调用 pthread_setschedparam( )函数时，调度策略不是 SCHED_SPORADIC，是否支持该函数取决于实现；换句话说，实现不必允许应用程序能够动态地将调度策略修改为 SCHED_SPORADIC。偶尔发生的服务器调度策略的相关调度参数包括：sched_ss_low_priority、sched_ss_repl_period、sched_ss_init_budget、sched_priority、以及 sched_ss_max_repl。所给定的 sched_ss_repl_period 值要大于或等于所给定的 sched_ss_init_budget，这样函数才会成功；否则，函数调用就会失败。sched_ss_max_repl 的取值应当位于[1，SS_REPL_MAX]的含括范围之内，这样函数才能成功；否则，函数就会失败。

如果 pthread_setschedparam( )函数失败，那么目标线程的调度参数就不会被修改。

返回值

如果成功，pthread_getschedparam( )和 pthread_setschedparam( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现下面情况，pthread_getschedparam( )函数可能失败：

[ESRCH] 由 thread 指定的值没有指向已存在线程。

如果出现下面情况，pthread_setschedparam( )函数可能失败：

[EINVAL] policy 或者与调度策略 policy 相关联的调度参数之一指定的值无效。

[ENOTSUP] 试图将策略或者调度参数设置成一个不被支持的值。

[ENOTSUP]试图将调度策略动态修改成 SCHED_SPORADIC，并且实现不支持这种修改。

[EPERM] 调用者不具备对给定线程进行调度参数或者调度策略设置的相应权限。

[EPERM] 实现不允许应用程序将参数之一修改为指定的值。

[ESRCH]由 thread 指定的值没有指向任何已存在的线程。

这些函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_setschedprio( )、pthread_attr_getparam( )、pthread_getscheduler( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<sched.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 相一致。

Issue 6

pthread_getschedparam( )和 pthread_setschedparam( )函数被标记为 Threads 及 Thread Execution Scheduling 选项的一部分。

删除了[ENOSYS]错误条件，因为如果实现不支持 Thread Execution Scheduling 选项，那么就无需提供 stub。

应用了 Open Group Corrigendum U026/2，修正了 pthread_setschedparam( )函数的原型，其第 2 个参数的类型成为 int。

为了与 IEEE Std 1003.1.d-1999 一致，增加了 SCHED_SPORADIC 调度策略。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_getschedparam( )原型中加入了 restrict 关键字。

应用了 Open Group Corrigendum U047/1。

应用了 IEEE PASC Interpretation 1003.1 #96，注意优先级的值也可以通过调用 pthread_setschedprio( )函数来进行设置。

名称

pthread_getspecific 和 pthread_setspecific——管理线程特定数据。

调用形式

THR

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

描述

函数 pthread_getspecific( )将返回代表调用线程绑定到指定的 key 的当前值。

函数 pthread_setspecific( )将线程特定的 value 同之前调用 pthread_key_create( )所得到的 key 关联起来。不同线程可能将不同的值绑定到相同的 key。这些值通常是指向被调用线程保留的动态分配内存块的指针。

使用不是从 pthread_key_create( )得到的 key 值，或当 key 已经通过 pthread_key_delete( )删除，调用 pthread_getspecific( )或 pthread_setspecific( )的效果未被定义。

线程特定数据的析构函数可以调用 pthread_getspecific( )和 pthread_setspecific( )。为已经销毁的线程特定数据 key 调用 pthread_getspecific( )将会返回 NULL 值，除非值被对 pthread_setspecific( )的调用改变(在析构函数启动之后)。从线程特定数据析构例程调用 pthread_setspecific( )可能会导致丢失存储(至少在 PTHREAD_DESTRUCTOR_ITERATIONS 尝试析构之后)或无限循环。

两个函数都被实现为宏。

返回值

函数 pthread_getspecific( )将返回同指定的 key 关联的线程特定数据值。如果没有现成特定数据值同 key 关联，则会返回 NULL 值。

如果成功，pthread_setspecific( )将返回 0，否则，将会返回一个错误号来指示错误。

错误

pthread_getspecific( )不返回错误。

如果发生如下情况，则 pthread_setspecific( )将失败：

[ENOMEM]　没有足够的内存将值同关键字关联。

如果发生以下情况，则 pthread_setspecific( )可能会失败：

[EINVAL]　关键字的值无效。

这些函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

pthread_getspecific( )的性能和易用性对于依赖于维护线程特定数据中的状态的函数非常关键。由于不要求通过它来检测错误，而且由于它能够检测出的错误只是使用无效关键字，因此 pthread_getspecific( )的功能被设计为倾向于速度和简明性，而不是报告错误。

未来方向

无。

参见

pthread_key_create( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

函数 pthread_getspecific( )和 pthread_getspecific( )被标记为 Threads 选项的一部分。

应用了 IEEE PASC Interpretation 1003.1c #3 (Part 6)，更新了描述。

名称

pthread_join——等待线程终止。

调用形式

THR

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

描述

pthread_join( )函数将挂起调用线程的执行至目标 thread 终止，除非目标 thread 已经终止。当以非 NULL 的 value_prt 作为参数对 pthread_join( )的调用成功返回时，通过终止线程传递给 pthread_exit( )的值就应当位于 value_ptr 所指向的位置并且可用了。当 pthread_join( )成功返回，目标线程就已经被终止。对指定相同目标线程的 pthread_join( )进行多个同时调用的结果是未定义的。如果调用 pthread_join( )的线程被取消，那么目标线程就不会被分离。

已经退出但是仍然保持未结合状态的线程是否会被计入{PTHREAD_THREADS_MAX}是未指定的。

返回值

如果成功，pthread_join( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_join( )函数就会失败：

[EINVAL]　实现检测到 thread 指定的值没有指向一个可结合的线程。

[ESRCH]　没有找到任何可以与指定的线程 ID 对应的线程。

如果出现了下面的情况，pthread_join( )函数有可能会失败：

[EDEADLK]检测到死锁或者 thread 的值指向调用线程。

pthread_join( )函数不会返回错误码[EINTR]。

示例

下面给出了创建和删除一个线程的示例：

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

int main(void)
{
    int        ar[1000000];
```

```
    pthread_t  th1, th2;
    subarray   sb1, sb2;

    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);

    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```

应用程序使用

无。

基本原理

pthread_join( )函数确实很方便，在多线程应用程序中已经证明了其用处。如果没有提供该函数，通过将额外状态作为参数的一部分传递给 start_routine( )，程序员的确也可以对这个函数进行模拟。终止线程将设置一个标志以指示终止，并将作为该状态的一部分的一个条件进行广播；结合线程会在该条件变量上进行等待。尽管这样一种技术可以允许线程等待更为复杂的情况(例如，等待多个线程终止)，但是在单个线程终止上进行等待被认为是广泛有用的。而且，将 pthread_join( )包括进来毫无疑问地会将程序员从这种复杂的等待编程中解脱出来。因此，尽管 pthread_join( )不是一个原语，但是，将其包括进 IEEE Std 1003.1-2001 是相当有价值的。

pthread_join( )函数提供了一种允许应用程序等待线程终止的简单机制。在线程终止之后，应用程序就可以选择清理这个线程所使用的资源。例如，在 pthread_join( )返回之后，就可以收回任何给应用程序提供的栈存储区。

对于每个创建时将 detachstate 属性设置为 PTHREAD_CREATE_JOINABLE 的线程而言，它们最终都会调用到 pthread_join( )或者 pthread_detach( )函数，以便能够收回与线程相关联的存储区域。

出于下面的原因，pthread_join( )和取消之间的交互是定义良好的：

- pthread_join( )函数，类似于其他的非可安全异步取消(non-async-cancel-safe)的函数，只能使用延迟可取消类型进行调用。
- 在禁用可取消状态下，取消是无法发生的。

因此，只需考虑默认的可取消状态。正如所指出的那样，要么 pthread_join( )调用被取消，要么成功，但两者不能同时发生。这里的区别对于应用程序而言是很明显的，因为要么一个取消处理程序被运行，要么 pthread_join( )返回。在这里不存在任何的竞争条件，因为 pthread_join( )是在延迟可取消状态下调用的。

未来方向

无。

参见

pthread_create( )、wait( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_join( )函数被标记为 Threads 选项的一部分。

名称

pthread_kill——向线程发送一个信号。

调用形式

THR
```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

描述

pthread_kill( )函数请求向指定的线程发送一个信号。

在 kill( )中，如果 sig 为 0，则会执行错误检测，但是不会实际发送信号。

返回值

若成功完成，则函数会返回 0。否则，函数会返回一个错误号。如果 pthread_kill( )函数失败，则不会发送信号。

错误

如果发生以下情况，则 pthread_kill( )函数将失败：

[ESRCH] 根据给定的线程 ID 无法找到指定的对应线程。

[EINVAL] 参数 sig 的值是无效的或不被支持的信号编号。

pthread_kill( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

函数 pthread_kill( )提供了一种在调用进程中将信号异步导向线程的机制。这可以被一个线程用于将信号的广播发送改变为发送到一组线程。

注意 pthread_kill( )只会使得信号在指定线程的上下文中被处理；信号动作(终止或停止)会影响进程整体。

基本原理

无。

未来方向

无。

参见

kill( )、pthread_self( )、raise( )、the Base Definitions volume of IEEE Std 1003.1-2001 和 <signal.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_kill( )函数被标记为 Threads 选项的一部分。

加入了“应用程序使用”部分。

名称

pthread_mutex_destroy 和 pthread_mutex_init——销毁和初始化互斥量。

调用形式

THR

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

描述

pthread_mutex_destroy( )函数将销毁由 mutex 所指向的互斥量对象；从实际效果上这个对象就会变成未初始化的。实现可能会导致 pthread_mutex_destroy( )将 mutex 所指向的对象设置成无效值。可以使用 pthread_mutex_init( )对一个已经销毁的互斥量对象进行重新初始化；在互斥量对象已经销毁之后对其进行引用，其结果是未定义的。

销毁一个未加锁且已经初始化的互斥量是安全的。试图销毁一个已加锁的互斥量将会导致未定义的行为发生。

pthread_mutex_init( )函数会使用 attr 所指定的属性值对 mutex 所指向的互斥量进行初始化。如果 attr 为 NULL，那么就会使用默认的互斥量属性；其效果与将默认的互斥量属性对象的地址传递过去是相同的。每当成功地完成了一次初始化操作，互斥量的状态就会变成已初始化未加锁的状态。

只有 mutex 本身可以被用于执行同步。如果在对 pthread_mutex_lock( )、pthread_mutex_trylock( )、pthread_mutex_unlock( )和 pthread_mutex_destroy( )的调用中引用了 mutex 的副本，其结果是未定义的。

试图初始化一个已经初始化的互斥量，会导致未定义行为发生。

在适合于使用默认互斥量属性的情况下，就可以使用 PTHREAD_MUTEX_INITIALIZER 宏来对静态分配的互斥量进行初始化。其效果等效于调用 attr 参数为 NULL 的 pthread_mutex_init( )函数进行动态初始化，唯一的不同在于不会执行任何的错误检查。

返回值

如果成功，pthread_mutex_destroy( )和 pthread_mutex_init( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果实现了[EBUSY]和[EINVAL]错误检查，其执行看上去就好像是在函数处理之初它们就被立即执行了一样，并且会导致在对 mutex 所指定的互斥量进行修改之前返回一个错误。

错误

如果出现下面情况，pthread_mutex_destroy( )函数可能会失败：

[EBUSY]实现已经检测到当 mutex 所指向的对象被加锁或被另外的线程引用时(例如，正被用于 pthread_cond_timedwait( )或者 pthread_cond_wait( ))，试图对对象进行销毁。

[EINVAL]　mutex 指定的值无效。

如果出现下面情况，pthread_mutex_init( )函数将会失败：

[EAGAIN]　系统缺乏必要的资源(内存之外的资源)对另一个互斥量进行初始化。

[ENOMEM]　用于初始化互斥量的内存不足。

[EPERM]　调用者不具备执行该项操作的特权。

如果出现下面情况，pthread_mutex_init( )函数可能失败：

[EBUSY]实现已经检测到试图对 mutex 所指向的一个先前已初始化的、但尚未被销毁的互斥量对象进行重新初始化的尝试。

[EINVAL]　attr 指定的值无效。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

其他的实现可能

IEEE Std 1003.1-2001 的这一卷支持几种可选的互斥量实现。实现可以将锁直接保存在类型为 pthread_mutex_t 的对象中。或者实现可以将锁保存在堆中，并且在互斥量对象中只保存一个指针、句柄或者唯一的 ID 号。两种实现都各有优点，或可能依赖于特定的硬件配置。因此编写可移植的代码就不会因受到这种选择的影响，IEEE Std 1003.1-2001 的这一卷没有为这种类型定义赋值或者相等操作，它使用了“初始化”术语来强化(更具限制性)锁实际上可以驻留在互斥量对象本身中的概念。

请注意这样就可以避免对互斥量或者条件变量类型的过多说明，并且促成了类型的非透明性。

实现允许但并不要求让 pthread_mutex_destroy( )在互斥量中存入一个非法值。这样做将有助于检测到试图对已销毁的互斥量进行加锁(或者其他引用)的错误程序。

错误检查和所支持的性能之间的权衡

很多错误检查都是可选的，以便具体的实现能够根据特定应用程序以及执行环境的需要，在性能和错误检查程度之间进行权衡。作为一条常用规则，由系统所导致的错误或者情形(例如内存不足)总是需要进行报告的，但是由于错误编写的应用程序而导致的错误(例如，无法提供足够的同步以避免互斥量在使用中被销毁)则可以定为可选的。

这样就可能会出现大量的实现。例如，针对应用程序调试的实现可能会实现所有的错误检查，而在嵌入式计算机中的性能限制比较严格的条件下运行的单一的、可证明正确的应用程序的实现，可能就只会实现最少的检查。甚至可以提供两个版本的实现，类似于编译器所提供的两个选项：完整检查但是速度较慢的版本；有限检查但是速度较快的版本。禁止此种可选性将是对用户的一种不友好。

通过将“未定义行为”的使用仅限于包含错误的(编写很糟糕的)应用程序可能会出现的情况，以及将资源不可用类型的错误进行强制定义，IEEE Std 1003.1-2001 的这一卷确保了一个遵循相同标准的应用程序在大量实现之间是可移植的，同时又不会强迫所有的实现增加对一个正确的程序永远都不可能会出现的错误情况进行检查的开销。

为何没有定义任何限制

曾经考虑过为互斥量和条件变量的最大数目定义符号，但是后来又被否决了，原因在于这些对象的数目可能会动态地改变。而且，在很多实现中都将这些对象置于应用程序的内存中了；这样，就没有显式最大值。

互斥量和条件变量的静态初始化函数

如果对静态分配的同步对象进行静态初始化，那么带有私有静态同步变量的模块就可以避免运行时的初始化测试和开销了。而且，这样也简化了自初始化模块的编程。而在 C 库中这种模块又是很常见的：由于各种原因，设计需要进行自初始化而不是调用一个显式的模块初始化函数。下面给出了静态初始化的一个范例。

没有静态初始化，一个进行自初始化的例程 foo( )看上去将会类似于：

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

如果使用静态初始化，那么相同的例程将会如下编码：

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

请注意：静态初始化既消除了在 pthread_once( )中进行初始化测试的需要，又无需取出&foo_mutex 的值以获取要传递给 pthread_mutex_lock( )或者 pthread_mutex_unlock( )的地址。

因此，在所有的系统上用 C 编写的代码对静态对象进行初始化都是相当容易的，并且在很多类型的系统上(即(整个)同步对象均可以保存在应用程序内存中的那些系统)，其速度也是相当快的。

但是，对于那些需要将互斥量分配在专有内存区域之外的机器而言，可能就会出现锁性能问题了。实际上此类机器往往不得不让互斥量以及可能的条件变量包含指向实际硬件

锁的指针。要让静态初始化能够在此类机器上运转起来，pthread_mutex_lock( )也要测试指针是否指向了已分配的实际锁。如果没有的话，pthread_mutex_lock( )就要在使用前对其进行初始化。在加载程序之时就可以对此类资源进行保留，从而就不会增加互斥量加锁和条件变量等待以指示完全初始化失败的返回代码了。

pthread_mutex_lock( )这种运行时的测试初次看上去可能像是一项额外的工作；其实，需要这种额外的测试来检查一下指针是否已经初始化了。实际上，在大部分机器上，这项工作可以这样来实现：取指针，测试其取值是否为 0，如果已经初始化了，就可以使用这个指针了。尽管测试看上去似乎增加了额外的工作量，但对一个寄存器进行测试的额外开销通常是可以忽略掉的，因为实际上并没有任何的额外内存引用。随着越来越多的机器提供了 cache，实际的开销就在于内存引用，而不在于指令执行。

另外一种选择是，根据机器的架构，通常有办法消除在最为重要的情况下的所有开销，即在进行过初始化的锁上的锁操作开销。具体的做法是将多余的开销转移到频度较低的操作初始化上。由于离线互斥量分配也意味着一个地址必须在取内容之后才能够找到实际的锁，所以一种较为广泛应用的技术就是让静态初始化为该地址保存一个假值，尤其是，一个会导致机器错误发生的地址。当初次尝试对该互斥量加锁时出现了这样一种错误情况，就可以完成有效性检查了，然后就可以用实际锁的正确地址对其进行填充了。由于它们不会再“出错”了，所以后继的锁操作就不会引入任何的额外开销。这仅仅是一项可用于支持静态初始化的技术，而不会对锁获取的性能产生反面影响。毫无疑问，还存在着其他高度机器相关的技术。

因此，进行离线互斥量分配的机器花在锁操作上的开销，就类似于进行隐式初始化的模块，这种情况对进行完全在线的互斥量分配是一种改进。这样，在线情况的速度就会快一些，而离线的情况也不会太差。

对于这种机器而言，除了锁性能方面的问题之外，还有一项顾虑：在试图完成对静态分配的互斥量进行的初始化时，线程有可能会串行竞争初始化锁。(此类完成工作往往会包括如下操作：获得一个内部锁、分配一个数据结构、在互斥量中保存一个到该数据结构的指针以及释放这个内部锁。)首先，在很多实现中都会通过对互斥量地址进行散列操作而降低这种串行化；其次，此类串行化只会出现有限的次数。尤其是，它至多会发生同步对象被静态分配的次数。仍然可以使用 pthread_mutex_init( )或者 pthread_cond_wait( )来对动态分配的对象进行初始化。

最后，对于某些具体实现上的应用程序而言，如果上述用于离线分配的优化技术没有产生什么显著的性能改进的话，应用程序不妨使用相应的 pthread_*_init( )函数来对所有的同步对象进行显式初始化，这样就完全避免了静态初始化，而且所有的实现也都支持这种做法。具体的实现也可以将这些权衡写入文档，并给出对于这种特定的实现，哪种初始化技术更为高效的建议。

销毁互斥量

一个互斥量在解锁之后就可以立即将其销毁了。例如，可以考虑下面的代码：

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};

obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
(A)     pthread_mutex_destroy(&op->om);
(B)     free(op);
    } else
(C)     pthread_mutex_unlock(&op->om);
}
```

在这个例子中，对 obj 进行了引用计数，并且无论何时只要释放了对该对象的引用，就会调用 obj_done( )函数。要求具体的实现允许在对象解锁之后(C 行)就立即将其销毁和释放并且有可能解除映射(A 行和 B 行)。

未来方向

无。

参见

pthread_mutex_getprioceiling( )、pthread_mutex_lock( )、pthread_mutex_timedlock( )、pthread_mutexattr_getpshared( )、the Base Definitions volume of IEEE Std 1003.1-2001 和 <pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_mutex_destroy( )和 pthread_mutex_init( )函数被标记为 Threads 选项的一部分。

为了与 IEEE Std 1003.1d-1999 一致，在“参见”部分中添加了 pthread_mutex_timedlock( )函数。

应用了 IEEE PASC Interpretation 1003.1c #34，更新了“描述”部分。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutex_init( )原型中加入了 restrict 关键字。

名称

pthread_mutex_getprioceiling 和 pthread_mutex_setprioceiling——获取和设置一个互斥量的优先级上限(实时线程)。

调用形式

THR TPP
```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
    int *restrict prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

描述

pthread_mutex_getprioceiling()函数将返回互斥量的当前优先级上限。

pthread_mutex_setprioceiling()函数要么会对互斥量加锁(如果该锁没有被加锁的话)，要么会阻塞直至它能够成功地锁住该互斥量，然后就会修改互斥量的优先级上限并释放该锁。当修改成功时，就会在 old_ceiling 中返回优先级上限的先前值。对互斥量进行加锁的过程不需要遵循优先级保护协议的规则。

如果 pthread_mutex_setprioceiling( )函数失败，互斥量优先级上限将不会被修改。

返回值

如果调用成功，pthread_mutex_getprioceiling( )和 pthread_mutex_setprioceiling( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现下面情况，pthread_mutex_getprioceiling( )和 pthread_mutex_setprioceiling( )函数可能失败：

[EINVAL] prioceiling 所要求的优先级超出了范围。

[EINVAL] mutex 指定的值没有指向当前已存在的互斥量。

[EPERM] 调用者不具备执行该项操作的特权。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_mutex_destroy( )、pthread_mutex_lock( )、pthread_mutex_timedlock( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含的目的是为了能够与 POSIX Threads Extension 一致。

被标记为 Realtime Threads Feature Group 的一部分。

Issue 6

pthread_mutex_getprioceiling( )和 pthread_mutex_setprioceiling( )函数被标记为 Threads 和 Thread Priority Protection 选项的一部分。

删除了[ENOSYS]错误条件，原因在于：如果某种具体的实现不支持 Thread Execution Scheduling 选项，那么就无需提供 stub。

删除了指示不支持互斥量优先级上限协议的[ENOSYS]错误。这是因为：如果实现提供了这些函数(不管是否定义了_POSIX_PTHREAD_PRIO_PROTECT)，那么它们就必须按照“描述”中的方式运作，因此互斥量的优先级上限协议是受到支持的。

为了与 IEEE Std 1003.1d-1999 一致，在“参见”部分中添加了 pthread_mutex_timedlock( )函数。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutex_getprioceiling( )和 pthread_mutex_setprioceiling( )原型中加入了 restrict 关键字。

名称

pthread_mutex_int——初始化互斥量。

调用形式

THR

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

描述

参见 pthread_mutex_destroy( )。

名称

pthread_mutex_lock、pthread_mutex_trylock 和 pthread_mutex_unlock——对互斥量进行加锁和解锁。

调用形式

THR

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

描述

由 mutex 所指向的互斥量对象将通过调用 pthread_mutex_lock( )来加锁。如果该互斥量已被加锁，那么调用线程就会阻塞直至互斥量可用。这个操作返回时，mutex 所引用的互斥量对象就会处于加锁状态，并且以调用线程为其拥有者。

如果互斥量类型为 PTHREAD_MUTEX_NORMAL，就不会提供死锁检查。试图对互斥量进行重新加锁将会导致死锁。如果一个线程试图对一个尚未加锁或者目前处于解锁状态的互斥量进行解锁操作，就会导致未定义行为出现。

如果互斥量的类型为 PTHREAD_MUTEX_ERRORCHECK，那么，将提供错误检测。如果一个线程试图对一个已经加了锁的互斥量进行重新加锁，那么就会返回一个错误。如果一个线程试图对一个尚未加锁或者目前处于解锁状态的互斥量进行解锁操作，也会返回一个错误。

如果互斥量的类型为 PTHREAD_MUTEX_RECURSIVE，那么互斥量就会维护一个锁数的概念。每当线程第一次成功地获得了互斥量，锁数就会被置为 1。每当一个线程对该互斥量进行重新加锁时，锁数就会加 1。每当有线程对该互斥量进行解锁操作，锁计数就会减 1。当锁计数到达 0 时，互斥量就可以为其他线程所获取了。如果一个线程试图对一个尚未加锁或者目前处于解锁状态的互斥量进行解锁操作，那么就会返回一个错误。

如果互斥量的类型为 PTHREAD_MUTEX_DEFAULT，那么试图对互斥量进行递归加锁的尝试就会导致未定义行为的出现。如果试图对不是由调用线程加锁的互斥量进行解锁操作，也会导致未定义行为的产生。如果试图对尚未被加锁的互斥量进行解锁操作，也会导致未定义行为的产生。

pthread_mutex_trylock( )函数等效于 pthread_mutex_lock( )，唯一的不同在于：如果 mutex 所指向的互斥量对象目前被(任何线程，包括当前线程)加锁，那么调用就会立即返回。如果互斥量的类型为 PTHREAD_MUTEX_RECURSIVE，并且互斥量目前正为调用线程所拥有，那么互斥量的计数就会加 1，pthread_mutex_trylock( )函数将会立即成功返回。

pthread_mutex_unlock( )函数将会释放 mutex 所指向的互斥量对象。释放互斥量的方式取决于互斥量的类型属性。如果在调用 pthread_mutex_unlock( )时，还有线程阻塞在 mutex 所指向的互斥量对象上的话，那么就会导致互斥量变成可用状态，调度策略将会判断究竟该由哪个线程来获取该互斥量。

(如果使用的是 PTHREAD_MUTEX_RECURSIVE 互斥量，那么当锁计数到达 0，并且

调用线程不再拥有对该互斥量的任何加锁时，该互斥量就变成可用的了。)

如果给一个等待互斥量的线程发送了一个信号，那么当信号从处理程序返回之时它将继续等待该互斥量，看上去就好像没有被中断过一样。

返回值

如果成功，pthread_mutex_lock( )和 pthread_mutex_unlock( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

如果获取到了 mutex 所指向的互斥量对象上的锁，那么 pthread_mutex_trylock()函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutex_lock( )和 pthread_mutex_unlock( )函数将会失败：

[EINVAL] 在创建 mutex 时协议属性的取值为 PTHREAD_PRIO_PROTECT，并且调用线程的优先级高于互斥量当前的优先级上限。

如果出现了下面的情况，pthread_mutex_trylock( )函数会失败：

[EBUSY] 无法得到互斥量，因为它已经被加锁了。

如果出现了下述情况，pthread_mutex_lock( )、pthread_mutex_trylock( )和 pthread_mutex_unlock( )函数可能会失败：

[EINVAL] mutex 所指定的值没有指向已初始化的互斥量对象。

[EAGAIN] 无法得到互斥量，因为已经超出了 mutex 递归锁的最大数目。

如果出现了下面的情况，pthread_mutex_lock( )函数可能会失败：

[EDEADLOCK] 当前线程已经拥有该互斥量。

如果出现了下面的情况，pthread_mutex_unlock( )函数可能会失败：

[EPERM] 当前线程不拥有该互斥量。

这 3 个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

互斥量对象的目的在于作为一种底层原语，在此之上可以构建其他线程同步函数。因此，互斥量的实现应当尽可能地高效，并且这一点对于接口的可用特征也具有分枝影响。

互斥量函数以及互斥量属性的特定默认设置已经受到了如下期望的推动：能够得到不排除互斥量加锁、解锁的快速内联实现。

例如，为了避免在基本的机制中所要求的开销大于绝对的需要，双锁上的死锁是一种显式允许的行为。(用户通过其他机制可以很容易地构建出更加“友好的”、可以进行死锁检测或者允许同一个线程多次加锁的互斥量。例如，可以使用 pthread_self( )记录互斥量的拥有者关系。)具体的实现也可以选择提供诸如特殊互斥量属性之类的扩展特征。

由于仅仅在线程即将被阻塞时，才需要对大部分属性进行检查，所以属性的使用不会减缓(常见的)互斥量加锁操作。

同样，尽管大家可能期望能够提取到互斥量拥有者的线程 ID，但是这种做法要求每次对互斥量加锁时保存当前线程的 ID，从而就会引入无法接受的开销。类似的讨论也适用于 mutex_tryunlock 操作。

未来方向

无。

参见

pthread_mutex_destroy( )、pthread_mutex_timedlock( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

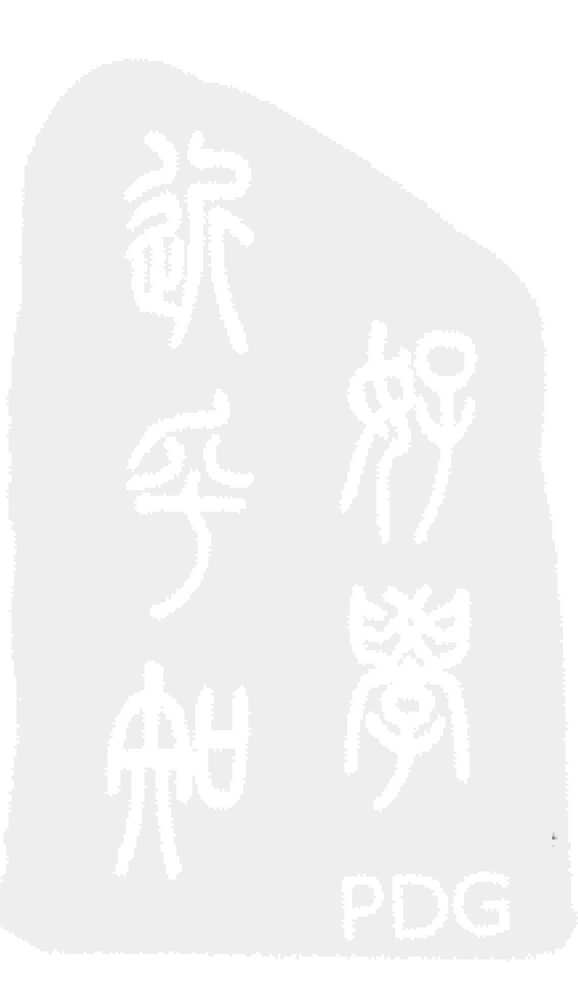
首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_mutex_lock( )、pthread_mutex_trylock( )和 pthread_mutex_unlock( )函数被标记为 Threads 选项的一部分。

下面 POSIX 上实现的新要求来源于与 Single UNIX Specification 一致的需要：定义了试图重新对一个互斥量加锁的行为。

为了与 IEEE Std 1003.1d-1999 一致，在“参见”部分中添加了 pthread_mutex_timedlock( ) 函数。



名称

pthread_mutex_setprioceiling——设置互斥量的优先级上限(实时线程)。

调用形式

THR TPP
```
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

描述

参见 pthread_mutex_getprioceiling( )。

名称

pthread_mutex_timedlock——对互斥量进行加锁(高级实时)。

调用形式

THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abs_timeout);
```

描述

pthread_mutex_timedlock( )函数会对 mutex 所指向的互斥量对象进行加锁。如果互斥量已经被加了锁，那么调用线程将会阻塞直至互斥量变为可用，正如 pthread_mutex_lock( )函数中的情况。如果不等待另一个线程解锁互斥量，而无法对互斥量加锁，那么当指定的超时时间到达时这种等待就会终止。

当 abs_timeout 所指定的绝对时间到达时，正如超时所基于的时钟所测量的那样(也就是说，当该时钟的取值等于或者超过 abs_timeout 时)，或者在调用之时 abs_timeout 所指定的绝对时间就已经过了，就会超时。

如果支持 Timers 选项的话，那么超时所基于的时钟将是 CLOCK_REALTIME 时钟；如果不支持 Timers 选项的话，那么超时所基于的时钟将是 time( )函数所返回的系统时钟。

超时的精度与其所基于的时钟的精度是一样的。在<time.h>中定义了 timespec 数据类型。

如果能够立即对互斥量加锁的话，那么该函数永远也不会出现超时失败。如果能够立即对互斥量加锁的话，那么无需检查 abs_timeout 参数的有效性。

由于优先级的继承规则(对于使用 PRIO_INHERIT 协议进行初始化的互斥量)，如果因为超时而导致一个定时互斥量等待被终止，那么就会对互斥量拥有者的优先级进行调整，以反映该线程不在等待互斥量的线程之列这一事实。

返回值

如果调用成功，pthread_mutex_timedlock( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutex_timedlock( )函数就会失败：

[EINVAL] 在创建 mutex 时协议属性的值为 PTHREAD_PRIO_PROTECT，并且调用线程的优先级高于互斥量当前的优先级上限。

[EINVAL] 进程或者线程可能会阻塞，并且 abs_timeout 参数指定了一个小于 0 或者大于等于 10 亿纳秒的域。

[ETIMEDOUT] 在指定时间超过之前，无法对互斥量加锁。

如果出现了下面的情况，pthread_mutex_timedlock( )函数可能会失败：

[EINVAL] mutex 指定的值没有指向一个已初始化的互斥量对象。

[EAGAIN]　无法得到互斥量，因为已经超出了 mutex 递归锁允许的最大数目。

[EDEADLOCK]　当前的线程已经拥有了该互斥量。

这个函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

pthread_mutex_timedlock( )函数是 Threads 和 Timeouts 选项的一部分，并且无需在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

pthread_mutex_destroy( )、pthread_mutex_lock( )、pthread_mutex_trylock( )、time( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<time.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

名称

pthread_mutex_trylock 和 pthread_mutex_unlock——对互斥量加锁与解锁。

调用形式

THR

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

描述

参见 pthread_mutex_lock( )。

名称

pthread_mutexattr_destroy 和 pthread_mutexattr_init——销毁和初始化互斥量属性对象。

调用形式

THR

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

描述

pthread_mutexattr_destroy( )函数会销毁一个互斥量属性对象；实际上，这个对象就会变成未初始化的。实现可能会导致 pthread_mutexattr_destroy( )将 attr 指向的对象设置成一种无效值。可以使用 pthread_mutexattr_init( )对一个已经销毁了的属性对象进行重新初始化；在属性对象已经销毁之后对其进行引用，其结果是未定义的。

pthread_mutexattr_init( )函数会使用具体实现为全部属性所定义的默认值对互斥量属性对象 attr 进行初始化。

如果使用一个已初始化的 attr 属性对象调用 pthread_mutexattr_init( )，其结果是未定义的。

在互斥量属性对象已被用于初始化一个或者多个互斥量之后，影响属性对象的任何函数(包括析构函数)都不会影响任何先前已被初始化的互斥量。

返回值

如果调用成功，pthread_mutexattr_destroy( )和 pthread_mutexattr_init( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutexattr_destroy( )函数可能会失败：

[EINVAL]　attr 指定的值无效。

如果出现了下面的情况，pthread_mutexattr_init( )函数会失败：

[ENOMEM]　初始化互斥量属性对象时内存不足。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

有关属性的一般性的解释可参见 pthread_attr_init( )。属性对象允许实现进行有用的扩展，并且允许在不修改已有函数的情况下，对 IEEE Std 1003.1-2001 的这一卷进行扩展。因此，它们提供了 IEEE Std 1003.1-2001 的这一卷的未来可扩展性，并且降低了对那些还没有被广泛实现或者理解的语义进行贸然标准化的诱惑。

已经被讨论的可能会增加的互斥量属性有：spin_only、limited_spin、no_spin、recursive

以及 metered。(对后面几个属性可能的意义稍做解释：recursive 互斥量允许锁的当前拥有者对锁进行多次的重新加锁；metered 互斥量可以透明地记录队列长度、等待时间等)。由于来自于共享实现和使用经验的结果还没有对这些属性的有用性达成一致的看法，所以在 IEEE Std 1003.1-2001 的这一卷中尚未对它们加以指定。不过，互斥量属性对象提供了对这些概念进行测试以便将来可能标准化的一种可能。

互斥量属性和性能

为了保证对互斥量属性默认取值的定义能够让使用默认值初始化的互斥量具有足够简单的语义，从而加锁和解锁的完成能够等效于测试-设置(test-and-set)指令(可能再加上几条其他的基本指令)，在设计时进行了谨慎的考虑。

如果互斥量具有非默认属性，至少存在一种实现方法可用来降低加锁时的测试开销。实现可以采用的此类方法之一(并且可以完全透明地遵循 POSIX 应用程序的标准)为：对以非默认属性值进行初始化的互斥量进行秘密预加锁。后来任何试图对此类互斥量进行加锁的尝试都会导致实现分枝到一条“慢道”上，看上去就好像是互斥量是不可用的一样、然后，在“慢道”上，实现就可以做些“实际的工作”，即对使用非默认值的互斥量进行加锁。底层的解锁操作要更为复杂一些，因为这种实现从来不希望释放掉对此类互斥量的预加锁。这就表明了，取决于具体的硬件，是可能进行一定的优化的，以便能够最为高效地处理那些大家认为“使用频度最高”的互斥量属性。

进程共享内存和同步

IEEE Std 1003.1-2001 的这一卷中内存映射函数的存在使得应用程序能够在为多个进程所访问(因此，也就可以为多个进程的线程所访问)的内存区域中进行同步对象的分配。

为了能够允许这种使用，而同时又能够保持常规使用情况(也就是在单进程中的使用)下的高效性，所以就定义了一个 process_shared 属性。

如果某种实现支持_POSIX_THREAD_PROCESS_SHARED 选项，那么就可以使用 process_shared 属性来指示互斥量或者条件变量可以为多个进程的线程所访问。

为 process_shared 属性选择了 PTHREAD_PROCESS_PRIVATE 的默认设置，以便默认地以最为高效的形式创建这些同步对象。

对于使用 PTHREAD_PROCESS_PRIVATE process_shared 属性初始化的同步变量，只有那些对它们进行初始化的进程中的线程才能对它们进行操作。而对于使用 PTHREAD_PROCESS_SHARED process_shared 属性初始化的同步变量，只要一个进程对其具有访问权限，那么该进程中的任何线程就都可以操作这个同步变量了。尤其是，这些进程的存在可能会超出初始化进程的生存期。例如，下面的这段代码就在一个映射文件中实现了一个可为许多进程所使用的、简单的计数信号量：

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```

```
void semaphore_post(semaphore_t *semap);
void semaphore_wait(semaphore_t *semap);
void semaphore_close(semaphore_t *semap);

/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
int fd;
    semaphore_t *semap;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;

    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
    semap->count = 0;
    return (semap);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;

    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    return (semap);
}
```

```
void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semapx->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}
```

下面这段代码中有 3 个独立的进程，它们创建、发布和等待文件/tmp/semaphore 中的信号量。一旦该文件被创建，即便发布和等待程序没有对信号量进行初始化，它们也可以对计数信号量进行增减(如果需要的话，会采取等待和唤醒)操作。

```
/* create.c */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}

/* post */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;
```

```
        semap = semaphore_open("/tmp/semaphore");
        if (semap == NULL)
            exit(1);
        semaphore_post(semap);
        semaphore_close(semap);
        return (0);
    }
    /* wait */
    #include "pthread.h"
    #include "sem.h"
    int
    main()
    {
        semaphore_t *semap;
        semap = semaphore_open("/tmp/semaphore");
        if (semap == NULL)
            exit(1);
        semaphore_wait(semap);
        semaphore_close(semap);
        return (0);
    }
```

未来方向

无。

参见

pthread_cond_destroy( )、pthread_create( )、pthread_mutex_destroy( )、pthread_mutexattr_destroy( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_mutexattr_destroy( )和 pthread_mutexattr_init( )函数被标记为 Threads 选项的一部分。

应用了 IEEE PASC Interpretation 1003.1c #27，更新了“错误”部分。

名称

pthread_mutexattr_getprioceiling 和 pthread_mutexattr_setprioceiling——获取和设置互斥量属性对象的优先级上限属性(实时线程)。

调用形式

THR TPP
```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
    restrict attr, int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```

描述

pthread_mutexattr_getprioceiling( )和 pthread_mutexattr_setprioceiling( )函数分别获取和设置 attr 所指向的互斥量属性对象的优先级上限属性，这个互斥量属性对象是先前由 pthread_mutexattr_init( )函数所创建的。

prioceiling 属性包含已被初始化的互斥量的优先级上限，其值位于 SCHED_FIFO 所定义的优先级最大范围之内。

prioceiling 属性定义了被初始化互斥量的优先级上限，这也是能够执行受互斥量保护的临界区必须具备的最低优先级。为了避免优先级的翻转，应该将互斥量的优先级上限设置为一个大于等于可以对该互斥量进行加锁的所有线程的最高优先级。prioceiling 属性的取值要位于 SCHED_FIFO 策略所定义的优先级最大范围之内。

返回值

如果调用成功，pthread_mutexattr_getprioceiling( )和 pthread_mutexattr_setprioceiling( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现下面情况，pthread_mutexattr_getprioceiling( )和 pthread_mutexattr_setprioceiling( )函数可能会失败：

[EINVAL] attr 或者 prioceiling 指定的值无效。

[EPERM] 调用者不具备执行该项操作的特权。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_cond_destroy( )、pthread_create( )、pthread_mutex_destroy( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来 Issue 5 的目的是为了能够与 POSIX Threads Extension 一致。

被标记为 Realtime Threads Feature Group 的一部分。

Issue 6

pthread_mutexattr_getprioceiling( )和 pthread_mutexattr_setprioceiling( )函数被标记为 Threads 和 Thread Priority Protection 选项的一部分。

删除了[ENOSYS]错误条件，原因在于如果某种实现不支持 Thread Execution Scheduling 选项，那么就无需提供 stub 了。

删除了[ENOTSUP]错误条件，因为这些函数没有 protocol 参数。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutexattr_getprioceiling( )原型中加入了 restrict 关键字。

名称

pthread_mutexattr_getprotocol 和 pthread_mutexattr_setprotocol——获取和设置互斥量属性对象的协议属性(实时线程)。

调用形式

THR
```
#include <pthread.h>
```
TPP|TPI
```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
    restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

描述

pthread_mutexattr_getprotocol( )和 pthread_mutexattr_setprotocol( )函数，将分别获取和设置先前由 pthread_mutexattr_init( )函数所创建的 attr 所指向的互斥量属性对象的协议属性。

protocol 属性定义了应用互斥量时要遵循的协议，protocol 的值可以为下面 3 种值之一：

PTHREAD_PRIO_NONE

PTHREAD_PRIO_INHERIT

PTHREAD_PRIO_PROTECT

它们均定义于<**pthread.h**>头文件中。

当一个线程拥有了 PTHREAD_PRIO_NONE protocol 属性时，其优先级和调度将不会受到其互斥量拥有者关系的影响。

当由于拥有一个或者多个带有 PTHREAD_PRIO_INHERIT protocol 属性的互斥量，而导致一个线程阻塞了更高优先级的线程时，它将以自己的优先级和那些等待该线程拥有的、以本协议初始化的任何一个互斥量的线程中所具有的最高优先级中较高的那一个来执行。

当一个线程拥有一个或者多个以 PTHREAD_PRIO_PROTECT 协议进行初始化的互斥量时，它将在自身的优先级和以该线程所拥有的、以这种属性初始化的所有互斥量的最高优先级上限中较高的那个优先级上执行，不论有没有其他线程受到这些互斥量的阻塞。

当一个线程持有以 PTHREAD_PRIO_INHERIT 或者 PTHREAD_PRIO_PROTECT 协议属性初始化的互斥量时，如果其原始的优先级被修改了，例如被一个 sched_setparam( )调用修改了，那么不会将它移到其优先级调度队列的尾部。同理，当一个线程解锁了一个以 PTHREAD_PRIO_INHERIT 或者 PTHREAD_PRIO_PROTECT 协议属性初始化的互斥量时，如果其原始的优先级被修改了，也不会将其移动到其优先级调度队列的尾部。

如果一个线程同时拥有多个以不同协议初始化的互斥量，它将会以从这些协议中得到的优先级中最高的优先级执行。

当一个线程对 pthread_mutex_lock( )函数发出了一次调用时，互斥量会以 PTHREAD_PRIO_INHERIT 作为协议属性值进行初始化，当调用线程由于互斥量为另一个线程所拥有而受到阻塞，只要拥有者线程继续拥有这个互斥量，它就会继承调用线程的优先级。实现会将其执行优先级更新为其所赋优先级及所有继承属性中的最大值。此外，如果拥有者线程本身阻塞在另一个互斥量上，按照递归的形式，相同的优先级继承效果将会传播到这个互斥量的拥有者线程。

返回值

如果调用成功，pthread_mutexattr_getprotocol( )和 pthread_mutexattr_setprotocol( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutexattr_setprotocol( )函数会失败：

[ENOTSUP] protocol 指定的值是一个未受支持的值。

如果出现了下述情况，pthread_mutexattr_getprotocol( )和 pthread_mutexattr_setprotocol( )函数可能会失败：

[EINVAL]　attr 或者 protocol 指定的值无效。

[EPERM]　调用者不具备执行该项操作的特权。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_cond_destroy( )、pthread_create( )、pthread_mutex_destroy( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

被标记为 Realtime Threads Feature Group 的一部分。

Issue 6

pthread_mutexattr_getprotocol( )和 pthread_mutexattr_setprotocol( )函数被标记为 Threads 选项以及 Thread Priority Protection 及 Thread Priority Inheritance 两选项之一的一部分。

删除了[ENOSYS]错误条件，原因在于：如果某种具体的实现不支持 Thread Priority Protection 或者 Thread Priority Inheritance 选项，那么就无需提供 stub 了。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutexattr_getprotocol( )原型中加入了 restrict 关键字。

名称

pthread_mutexattr_getpshared 和 pthread_mutexattr_setpshared——获取和设置进程共享属性。

调用形式

THR TSH
```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
    restrict attr, int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

描述

pthread_mutexattr_getpshared( )函数会从 attr 所指向的属性对象获得 process_shared 属性的值。pthread_mutexattr_setpshared( )函数会在 attr 所指向的已初始化属性对象中设置 process_shared 属性。

如果将 process_shared 属性设置成了 PTHREAD_PROCESS_SHARED，那么对为互斥量分配的内存区具有访问权限的任何线程都可以对该互斥量进行操作，即便互斥量被分配在了被多个进程共享的内存区域。如果 process_shared 属性为 PTHREAD_PROCESS_PRIVATE，那么只有那些与对互斥量进行初始化的线程处在相同进程中的线程才能对互斥量进行操作；如果不同进程中的线程试图对此类互斥量进行操作，其行为是未定义的。该属性的默认值为 PTHREAD_PROCESS_PRIVATE。

返回值

当成功完成，pthread_mutexattr_setpshared( )返回 0；否则，返回错误号以指示所产生的错误。

当成功完成，pthread_mutexattr_getpshared( )返回 0，并将 attr 的 process_shared 属性值保存到 pshared 参数指向的对象中去。否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutexattr_getpshared( )和 pthread_mutexattr_setpshared( )函数可能会失败：

[EINVAL]　attr 指定的值无效。

如果出现了下面的情况，pthread_mutexattr_setpshared( )函数可能会失败：

[EINVAL]　为属性所指定的新值超出了该属性合法取值的范围。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_cond_destroy( )、pthread_create( )、pthread_mutex_destroy( )、pthread_mutexattr_destroy( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的是为了能够与 POSIX Threads Extension 一致。

Issue 6

pthread_mutexattr_getpshared( )和 pthread_mutexattr_setpshared( )函数被标记为 Threads 和 Thread Process_Shared Synchronization 选项的一部分。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutexattr_getpshared( )原型中加入了 restrict 关键字。

名称

pthread_mutexattr_gettype 和 pthread_mutexattr_settype——获取和设置互斥量类型属性。

调用形式

XSI

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

描述

pthread_mutexattr_gettype( )和 pthread_mutexattr_settype( )函数分别获取和设置互斥量的 type 属性。这个属性在这些函数的 type 参数中进行设置。type 属性的默认值为 PTHREAD_MUTEX_DEFAULT。

在互斥量属性的 type 属性中包含互斥量的类型信息。有效的互斥量类型包括：

PTHREAD_MUTEX_NORMAL

这种类型的互斥量不会检测死锁。一个事先没有对互斥量进行解锁操作而试图对互斥量进行重新加锁的线程会导致死锁。试图对由另一个线程所加锁的互斥量进行解锁操作，将会导致未定义行为。试图对一个已被解锁的互斥量进行解锁操作，也会导致未定义行为。

PTHREAD_MUTEX_ERRORCHECK

这种类型的互斥量提供了错误检查。一个事先没有对互斥量进行解锁操作而试图对互斥量进行重新加锁的线程会导致错误的产生。试图对由不同线程所加锁的互斥量进行解锁操作，将会返回一个错误。试图对一个已被解锁的互斥量进行解锁操作，也会返回一个错误。

PTHREAD_MUTEX_RECURSIVE

一个事先没有对互斥量进行解锁操作而试图对互斥量进行重新加锁的线程会成功地对互斥量加锁。对于类型为 PTHREAD_MUTEX_NORMAL 的互斥量中会发生的重新加锁死锁情况，不会发生在这种类型的互斥量上。这种互斥量上的多锁将会要求相同次数的解锁，以便在另一个线程能够获得该锁之前释放掉它。试图对由另一个线程加锁的互斥量进行解锁操作，将会返回一个错误。试图对一个已被解锁的互斥量进行解锁操作，也会返回一个错误。

PTHREAD_MUTEX_DEFAULT

试图递归地对这种类型的互斥量进行加锁操作，将会导致未定义的行为。试图对非调用线程所加锁的互斥量进行解锁操作，将会导致未定义行为。试图对这种类型的一个尚未加锁的互斥量进行解锁操作，将会导致未定义行为。具体的实现可能会将这种互斥量映射为其他互斥量类型中的一种。

返回值

若成功完成，pthread_mutexattr_gettype( )函数返回 0，并将 attr 属性的 type 值保存到 type 参数指向的对象中；否则，返回一个错误号以指示所产生的错误。

若成功，pthread_mutexattr_settype( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_mutexattr_settype( )函数将会失败：

[EINVAL]　type 指定的值无效。

如果出现了下面的情况，pthread_mutexattr_gettype( )和 pthread_mutexattr_settype( )函数可能会失败：

[EINVAL]　attr 指定的值无效。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

建议应用程序不要和条件变量一起使用 PTHREAD_MUTEX_RECURSIVE 互斥量，因为为 pthread_cond_timedwait( )或者 pthread_cond_wait( )执行的隐式解锁操作可能不会实际地释放掉互斥量(如果它已经被多次加锁的话)。如果这种情况发生了，那么就不会有任何其他的线程能够满足断言所要求的条件。

基本原理

无。

未来方向

无。

参见

pthread_cond_timedwait( )、pthread_cond_wait( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

应用了 Open Group Corrigendum U033/3。更新了 pthread_mutexattr_gettype( )的“调用形式”部分，从而使它第一个参数的类型为 const pthread_mutexattr_t*。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_mutexattr_gettype( )原型中加入了 restrict 关键字。

名称

pthread_mutexattr_init——初始化互斥量属性对象。

调用形式

THR

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

描述

参见 pthread_mutexattr_destroy( )。

名称

pthread_mutexattr_setprioceiling——设置互斥量属性对象的prioceiling属性(实时线程)。

调用形式

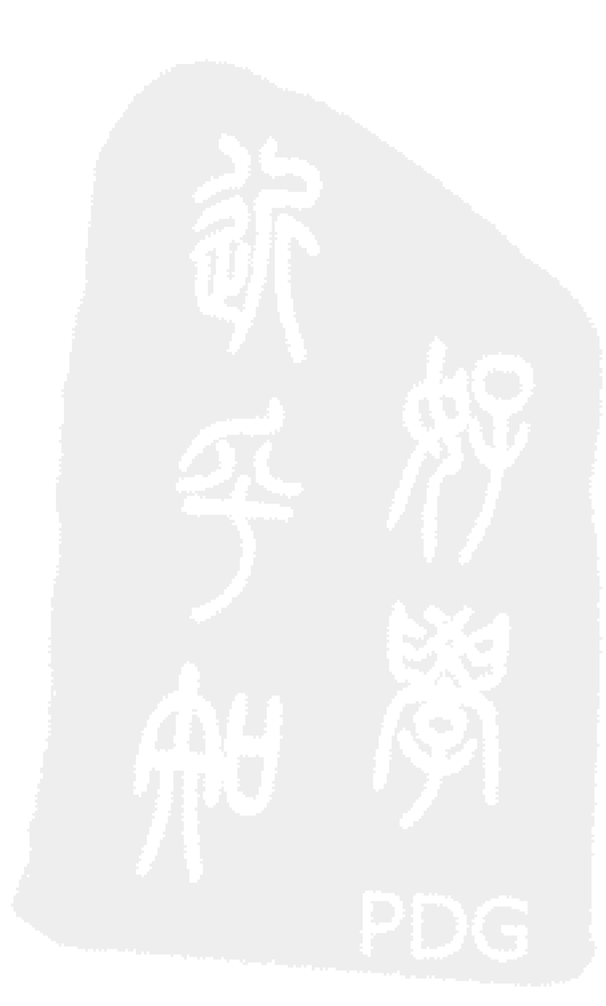
THR TPP

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```

描述

参见 pthread_mutexattr_getprioceiling( )。



名称

pthread_mutexattr_setprotocal——设置互斥量属性对象的 protocal 属性(实时线程)。

调用形式

THR

```
#include <pthread.h>
```

TPP|TPI

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

描述

参见 pthread_mutexattr_getprotocal( )。

名称

pthread_mutexattr_setpshared——设置 process-shared 属性。

调用形式

THR TSH

```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

描述

参见 pthread_mutexattr_getshared( )。

名称

pthread_mutexattr_settype——设置互斥量 type 属性。

调用形式

XSI

```
#include <pthread.h>
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

描述

参见 pthread_mutexattr_gettype( )。

名称

pthread_once——动态包初始化。

调用形式

THR

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

描述

如果进程中任何线程首次使用给定的 once_control 参数调用 pthread_once( )函数，就会调用不带参数的 init_routine。而使用相同 once_control 参数对 pthread_once( )的后继调用都不会调用 init_routine。在从 pthread_once( )返回之时，init_routine 就已经完成了。once_control 参数将会决定是否已经调用了相关的初始化例程。

pthread_once( )函数并不是一个取消点。不过，如果 init_routine 为一个取消点，并且已经被取消，那么，对 once_control 的影响就好像是从来没有调用过 pthread_once( )一样。

在<pthread.h>头文件中定义了 PTHREAD_ONCE_INIT 常量。

如果 once_control 具有自动存储期或者没有被 PTHREAD_ONCE_INIT 初始化，那么 pthread_once( )的行为将是未定义的。

返回值

每当函数成功完成，pthread_once( )就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_once( )函数可能会失败：

[EINVAL]　如果 once_control 或者 init_routine 中任何一个无效。

pthread_once( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

某些 C 库是为动态初始化设计的。也就是说，当调用该库的第一个过程时，就会执行对该库的全局初始化。在单线程的程序中，这通常会通过使用静态变量来实现，并且会在例程调用的入口检查该静态变量的值，具体做法如下所示：

```
static int random_is_initialized = 0;
extern int initialize_random();

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* Operations performed after initialization. */
}
```

为了在多线程程序中保持相同的结构，就需要新的原语。否则，就需要在库的任何使用之前显式地调用库导出初始化函数来完成库初始化。

对于多线程进程中的动态库初始化，一个简单的初始化标志并不够用；还需要对这个标志进行保护以免被同步调用库的多个线程所修改。而保护标志就需要使用互斥量；不过，互斥量必须在使用之前加以初始化。而要确保只对互斥量进行一次初始化，就会要求对这个问题提供一个递归性的解决方案。

pthread_once( )的使用不仅提供了一种受实现保证的动态初始化方法，而且也对多线程实时系统的可靠构建提供了帮助。据此，前面的例子就变成了下面的样子：

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern int initialize_random();

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* Operations performed after initialization. */
}
```

请注意，pthread_once_t 不能是一个数组，因为一些编译器不接受这样的构造&<array_name>。

未来方向

无。

参见

The Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

修改历史

首次发布于 Issue 5，包含进来的目的在于能够与 POSIX Threads Extension 一致。

Issue 6

pthread_once( )函数被标记为 Threads 选项的一部分。

添加了[EINVAL]错误，用它来指示两个参数之一无效情况下可能会出现的失败。

名称

pthread_rwlock_destroy 和 pthread_rwlock_init——销毁和初始化读-写锁对象。

调用形式

THR

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```

描述

pthread_rwlock_destroy( )函数会销毁由 rwlock 指向的读-写锁对象，并释放该锁所使用的全部资源。之后再使用该锁的结果是未定义的，直至另一次对 pthread_rwlock_init( )的调用对该锁进行重新初始化。实现可以让 pthread_rwlock_destroy( )对 rwlock 所指向的对象设置一个无效值。如果在没有任何线程持有 rwlock 的时候调用了 pthread_rwlock_destroy( )，其结果是未定义的。试图销毁一个未初始化的读-写锁将会导致未定义的行为。

pthread_rwlock_init( )函数将会对使用 rwlock 所指向的读-写锁时所需的任何资源进行分配，并用 attr 所指向的属性将该锁初始化为一种未加锁状态。如果 attr 为 NULL，那么就会使用默认的读-写锁，其效果与传递默认读-写锁属性对象的地址相同。一旦对锁进行了初始化，就可以任意次数地使用它而无需重新初始化。如果在指定一个已经初始化了的读-写锁的情况下调用 pthread_rwlock_init( )函数，其结果是未定义的。如果在使用一个读-写锁的时候没有首先对其进行初始化，其结果也是未定义的。

如果 pthread_rwlock_init( )函数失败，rwlock 将不会被初始化，并且 rwlock 的内容是未定义的。

只有 rwlock 指向的对象才可以被用于执行初始化。如果在调用 pthread_rwlock_destroy( )、pthread_rwlock_rdlock( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_tryrdlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_unlock( )或者 pthread_rwlock_wrlock( )的时候指向了该对象的副本，其结果是未定义的。

返回值

如果成功，pthread_rwlock_destroy( )和 pthread_rwlock_init( )函数都会返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果实现了[EBUSY]和[EINVAL]错误检查，那么其效果看上去就好像是在函数处理之初就立即执行了它们，并且在修改 rwlock 锁指定的读-写锁状态之前就导致了一个错误返回。

错误

如果出现了下面的情况，pthread_rwlock_destroy( )函数可能会失败：

[EBUSY]当 rwlock 所指向的对象尚处于加锁状态时，实现就已经检测到了试图销毁该对象的尝试。

[EINVAL] rwlock 指定的值无效。

如果出现了下面的情况，pthread_rwlock_init( )函数将会失败：

[EAGAIN]　系统缺乏对另一个读-写锁进行初始化所需的(内存之外的)资源。

[ENOMEM]　对读-写锁进行初始化时内存不足。

[EPERM]　调用者不具备执行该项操作的特权。

如果出现了下面的情况，pthread_rwlock_init( )函数可能会失败：

[EBUSY]实现检测到了试图对 rwlock 所指向的对象进行重新初始化的尝试，这是一个先前已经初始化过了，但是尚未被销毁的读-写锁。

[EINVAL]　attr 指定的值无效。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_rdlock( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_tryrdlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_unlock( )、pthread_rwlock_wrlock( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。初始化宏也从“调用形式”部分中删除了。
- 对“描述”部分的更新如下：
  - 它显式地提到了对读-写锁对象初始化时的资源分配。
  - 添加了一段说明读-写锁对象的副本可能会无法使用的文字。
- 在 pthread_rwlock_init( )的 ERRORS 部分添加了一个[EINVAL]错误，用于指示 rwlock 的取值无效。
- 对“参见”部分进行了更新。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_rwlock_init( )原型中加入了 restrict 关键字。

名称

pthread_rwlock_rdlock 和 pthread_rwlock_tryrdlock——获取和设置用于进行读操作的读-写锁对象。

调用形式

THR
```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

描述

pthread_rwlock_rdlock( )函数将读锁应用于 rwlock 所指向的读-写锁。如果没有写操作者持有该锁，并且也没有任何写操作者阻塞在该锁上，那么调用线程就会获得这个读锁。

如果实现支持 Threads Execution Scheduling 选项，并且锁中所涉及的线程执行在 SCHED_FIFO 或者 SCHED_RR 调度策略之下，那么倘若有一个写操作者持有该锁或者如果优先级更高或者同级的写操作者阻塞在该锁上，调用线程就不会获取到该锁；否则，调用线程就会获取到该锁。

如果实现支持 Threads Execution Scheduling 选项，并且锁中所涉及的线程执行在 SCHED_SPORADIC 调度策略之下，那么倘若有一个写操作者持有该锁或者如果优先级更高或者同级的写操作者阻塞在该锁上，调用线程就不会获取到该锁；否则，调用线程就会获取到该锁。

如果具体实现不支持 Thread Execution Scheduling 选项，那么当有写操作者持有该锁或者有写操作者阻塞在该锁上时，调用线程是否会获取到该锁将是由实现来定义的。如果有一个写操作者持有该锁，调用线程将无法获取到该读锁。如果没有获取到读锁，那么调用线程将会一直阻塞到它能够得到这把锁为止。如果调用线程在发出该调用时已经持有了一个写锁，那么可能线程将会死锁。

一个线程可以持有多个 rwlock 上的并发读锁(也就是说，成功地调用了 pthread_rwlock_rdlock( ) *n* 次)。如果这样的话，应用程序就要确保线程也执行了相匹配次数的解锁(也就是说，它也调用了 pthread_rwlock_unlock( ) *n* 次)。

某个具体实现保证能够应用于读-写锁上的可以同时使用的读锁的最大数目是由实现来定义的。如果超过了这个最大数目的话，pthread_rwlock_rdlock( )函数可能会失败。

与 pthread_rwlock_rdlock( )函数一样，pthread_rwlock_tryrdlock( )函数也会应用一把读锁，不同之处在于如果 pthread_rwlock_rdlock( )调用失败，则会阻塞调用线程，而 pthread_rwlock_tryrdlock( )函数无论如何都不会阻塞；它要么会获得锁，要么会失败并立即返回。

如果使用了一个未初始化的读-写锁对这两个函数中的任何一个进行了调用，其结果是未定义的。

如果向一个等待读-写锁进行读操作的线程发送了一个信号，那么在从信号处理程序返回之时，它会继续等待这把读-写锁以便进行读操作，看上去就好像并未被中断过一样。

返回值

如果调用成功，pthread_rwlock_rdlock( )函数会返回 0；否则，返回一个错误号以指示所产生的错误。

如果得到了 rwlock 所引用的读-写锁对象上的读锁，pthread_rwlock_tryrdlock( )函数会返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_tryrdlock( )函数将会失败：

[EBUSY]无法得到读-写锁进行读操作，因为有个写操作者持有该锁或者有个具有一定优先级的写操作者阻塞在该锁上。

如果出现了下面的情况，pthread_rwlock_rdlock( )和 pthread_rwlock_tryrdlock( )函数可能会失败：

[EINVAL]　rwlock 指定的值没有引用到一个已初始化了的读-写锁对象。

[EAGAIN]　无法获得读锁，因为已经超过了 rwlock 读锁的最大数目。

如果出现了下面的情况，pthread_rwlock_rdlock( )函数可能会失败：

[EDEADLOCK]　当前线程已经为了写入占有了这个读-写锁了。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

正如在 IEEE Std 1003.1-2001 Base Definitions 卷，3.285 部分，Priority Inversion 中所讨论的那样，使用这两个函数的应用程序可能会遭受优先级翻转。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_unlock( )、pthread_rwlock_wrlock()、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。
- 对“描述”部分的更新如下：
  - 说明了写操作者会优先于读操作者的条件。
  - 对 pthread_rwlock_tryrdlock( )函数的失败情况进行了阐明。
  - 添加了一段说明读锁最大数目的文字。
- 在“错误”部分，对[EBUSY]进行了修改：考虑了写优先级，并将[EDEADLOCK]从 pthread_rwlock_tryrdlock( )的错误表中删除了。
- 对“参见”部分进行了更新。

名称

pthread_rwlock_timedrdlock——为了读取对一个读-写锁加锁。

调用形式

THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

描述

与 pthread_rwlock_rdlock( )函数相似，pthread_rwlock_timedrdlock( )函数会向 rwlock 所引用的读-写锁应用一个读锁。不过，如果不等待其他线程解锁就无法得到该锁时，当指定的时间超时之后，等待就会被终止。当 abs_timeout 所指定的绝对时间过了的时候，正如超时所基于的时钟所测量的一样(也就是说，当这个时钟的取值等于或者超过了 abs_timeout 时)，或者如果在调用之时 abs_timeout 所指定的绝对时间就已经过了的话，超时就过期了。

如果支持 Timers 选项的话，那么超时所基于的时钟为 CLOCK_REALTIME 时钟；如果不支持 Timers 选项的话，那么超时所基于的时钟为 time( )函数所返回的系统时钟。超时的精度与它所基于的时钟是相同的。在<time.h>头文件中定义了 timespec 数据类型。如果该锁能够立即获得的话，那么无论怎样这个函数都不会超时失败。如果立即获得了该锁的话，就无需去检查 abs_timeout 参数的有效性。

如果向一个由于 pthread_rwlock_timedrdlock( )调用而阻塞在一个读-写锁上的线程发送了一个会导致信号处理程序执行的信号，那么当线程从信号处理程序返回之时，它会继续等待该锁，看上去就好像它从未被中断过一样。

如果在发出调用之时，调用线程已经持有了 rwlock 上的一个写锁，那么调用线程就会死锁。如果使用一个未被初始化的读-写锁调用这个函数，其结果是未定义的。

返回值

如果获得了 rwlock 所引用的读-写锁上的读锁，那么 pthread_rwlock_timedrdlock( )函数会返回 0；否则，会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_timedrdlock( )函数将会失败：

[ETIMEDOUT]　在指定的超时过期之前无法得到锁。

如果出现了下面的情况，pthread_rwlock_timedrdlock( )函数可能会失败：

[EAGAIN]　无法得到读锁，因为已经超过了该锁所允许的读锁的最大数目。

[EDEADLOCK] 调用线程已经持有了 rwlock 上的写锁。

[EINVAL] rwlock 指定的值没有指向一个已初始化的读-写锁对象，或者 abs_timeout 纳秒的取值小于 0 或者大于等于 10 亿。

这个函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

正如在 IEEE Std 1003.1-2001 Base Definitions 卷，3.285 部分，Priority Inversion 中所讨论的那样，使用这两个函数的应用程序可能会遭受优先级翻转。

pthread_rwlock_timedrdlock( )函数为 Threads 和 Timeouts 选项的一部分，无需在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlock_rdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_tryrdlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_unlock( )、pthread_rwlock_wrlock( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<time.h>。

变更历史

首次发布于 Issue 6，来源于 IEEE Std 1003.1j-2000。

名称

pthread_rwlock_timedwrlock——为了写入对读-写锁加锁。

调用形式

THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

描述

与 pthread_rwlock_wrlock( )函数相似，pthread_rwlock_timedwrlock( )函数会向 rwlock 所引用的读-写锁应用一个写锁。不过，如果不等待其他线程解锁就无法得到该锁时，那么当指定的时间过期之后，这种等待就会被终止。当 abs_timeout 所指定的绝对时间过了的时候，正如超时所基于的时钟所测量的一样(也就是说，当这个时钟的取值等于或者超过了 abs_timeout 时)，或者如果在调用之时 abs_timeout 所指定的绝对时间就已经过了的话，超时就过期了。

如果支持 Timers 选项的话，那么超时所基于的时钟为 CLOCK_REALTIME 时钟；如果不支持 Timers 选项的话，那么超时所基于的时钟为 time( )函数所返回的系统时钟。超时的精度与它所基于的时钟的精度是相同的。在<time.h>头文件中定义了 timespec 数据类型。如果能够立即获得该锁的话，那么无论怎样这个函数都不会超时失败。如果立即获得了该锁的话，就无需去检查 abs_timeout 参数的有效性。

如果向一个由于 pthread_rwlock_timedwrlock( )调用而阻塞在一个读-写锁上的线程发送了一个会导致信号处理程序执行的信号，那么当线程从信号处理程序返回之时，它会继续等待该锁，看上去就好像它从未被中断过一样。

如果在发出调用之时，调用线程已经持有了这个读-写锁，那么调用线程就会死锁。如果使用一个未被初始化的读-写锁调用这个函数，其结果是未定义的。

返回值

如果获得了 rwlock 所引用的读-写锁上的写锁，那么 pthread_rwlock_timedwrlock( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_timedwrlock( )函数将会失败：

[ETIMEDOUT]　在指定的超时过期之前无法得到锁。

如果出现了下面的情况，pthread_rwlock_timedwrlock( )函数可能会失败：

[EDEADLOCK] 调用线程已经持有了 rwlock 锁。

[EINVAL] rwlock 指定的值没有指向一个已初始化的读-写锁对象，或者 abs_timeout 纳秒域的取值小于 0 或者大于等于 10 亿。

这个函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

正如在 IEEE Std 1003.1-2001 Base Definitions 卷，3.285 部分，Priority Inversion 中所讨论的那样，使用这两个函数的应用程序可能会遭受优先级翻转。

pthread_rwlock_timedwrlock( )函数为 Threads 和 Timeouts 选项的一部分，并且无需在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlock_rdlock( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_tryrdlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_unlock( )、pthread_rwlock_wrlock( )、the Base Definitions volume of IEEE Std 1003.1-2001、<pthread.h>和<time.h>。

变更历史

首次发布于 Issue 6，来源于 IEEE Std 1003.1j-2000。

名称

pthread_rwlock_tryrdlock——为读操作对一个读-写锁加锁。

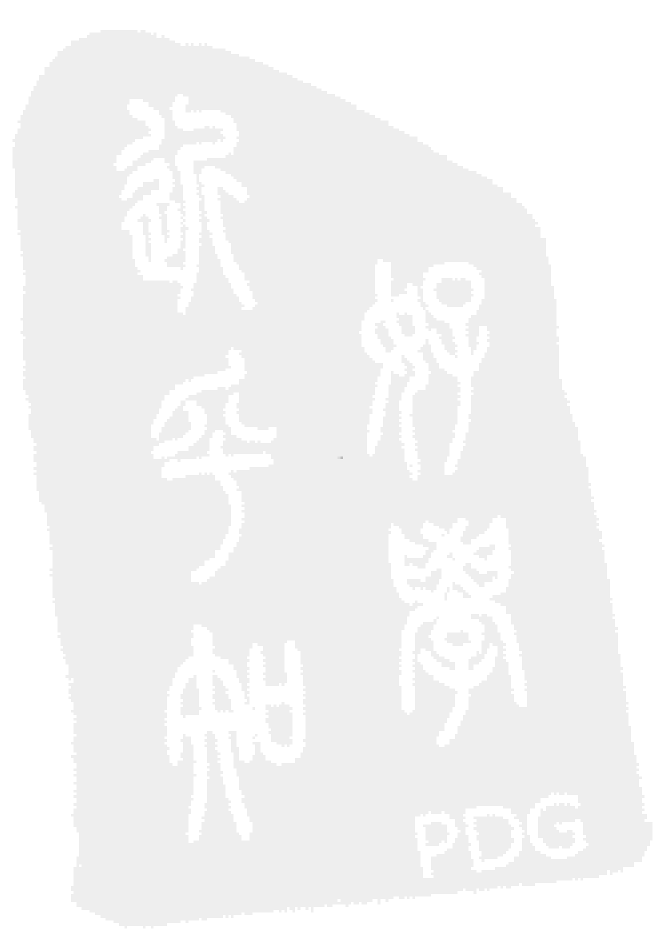
调用形式

THR

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

描述

参见 pthread_rwlock_rdlock( )。



名称

pthread_rwlock_trywrlock 和 pthread_rwlock_wrlock——获取和设置用于进行写操作的读-写锁对象。

调用形式

THR

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

描述

与 pthread_rwlock_wrlock( )函数相似，pthread_rwlock_trywrlock( )函数将会应用一个写锁，不同之处在于：如果当前已经有线程持有 rwlock 了(用于读操作或者写操作)，那么这个函数就会失败。

pthread_rwlock_wrlock( )函数会对 rwlock 所引用的读-写锁应用一个写锁。如果没有任何其他的线程(读操作者或者写操作者)持有读-写锁 rwlock，那么调用线程就会获得这个写锁。如果在函数调用之时，调用线程已经持有了这个读-写锁(不论是读锁还是写锁)，那么该线程可能就会死锁。

为避免“写饥饿”情况的出现，实现可能会在读写操作上优先考虑写操作。

如果使用了一个未初始化的读-写锁对这两个函数中的任何一个进行调用，其结果是未定义的。

如果向一个等待读-写锁进行写操作的线程发送了一个信号，那么在从信号处理程序返回之时，它会继续等待这把读-写锁以便进行写操作，看上去就好像并未被中断过一样。

返回值

如果得到了 rwlock 所引用的读-写锁对象上的写锁，那么 pthread_rwlock_trywrlock( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

如果调用成功，pthread_rwlock_wrlock( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_trywrlock( )函数将会失败：

[EBUSY] 无法得到读-写锁进行写操作，因为该锁已经被锁住进行读操作或者写操作了。

如果出现了下面的情况，pthread_rwlock_trywrlock( )和 pthread_rwlock_wrlock( )函数可能会失败：

[EINVAL] rwlock 指定的值没有指向一个已初始化了的读-写锁对象。

如果出现了下面的情况，pthread_rwlock_wrlock( )函数可能会失败：

[EDEADLK] 当前线程已经占有了这个读-写锁进行写操作或者读操作了。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

正如在 IEEE Std 1003.1-2001 Base Definitions 卷，3.285 部分，Priority Inversion 中所讨论的那样，使用这两个函数的应用程序可能会遭受优先级翻转。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlock_rdlock( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_tryrdlock( )、pthread_rwlock_unlock( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

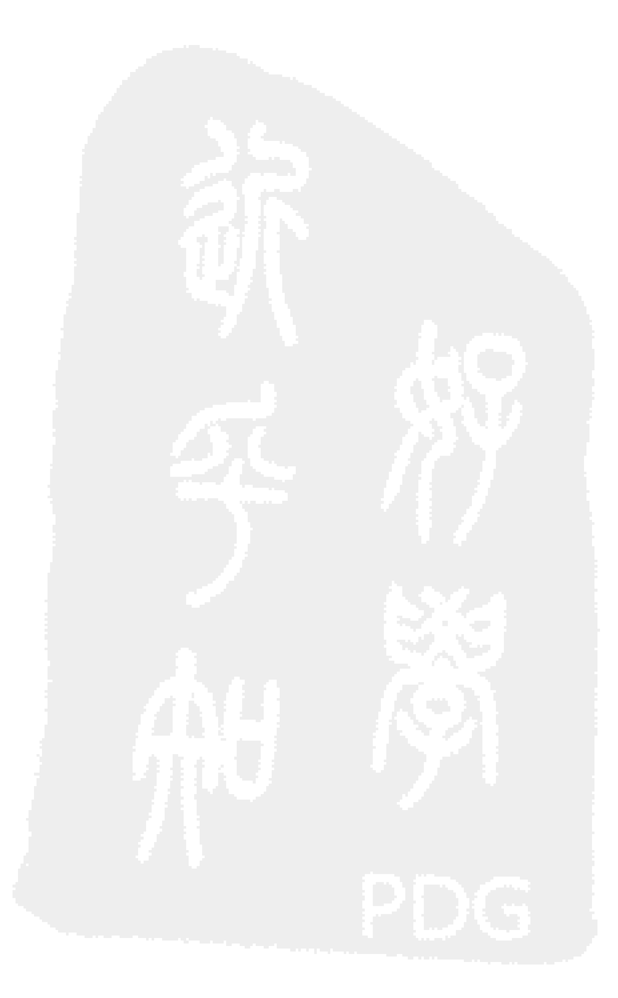
变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。
- 将[EDEADLK]从 pthread_rwlock_trywrlock( )的错误表中删除了。
- 对“参见”部分进行了更新。



名称

pthread_rwlock_unlock——对一个读-写锁对象进行解锁。

调用形式

THR
```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

描述

pthread_rwlock_unlock( )函数会释放 rwlock 指向的读-写锁上的一次加锁。如果调用线程没有持有该锁，那么其结果将是未定义的。

如果调用该函数来释放读-写锁对象上的一个读锁，并且当前在该读-写锁对象上还持有其他的读锁，那么这个读-写锁对象将会保持在读加锁状态。如果这个函数释放掉了某个读-写锁对象上的最后一个读锁，那么该读-写锁对象就会被置于没有任何拥有者的未加锁状态。

如果调用该函数来释放这个读-写锁对象上的一个写锁，那么读-写锁对象就会被置于未加锁状态。

如果在某个读-写锁可用时，还有其他的线程阻塞在它上面，那么调度策略将会决定该由哪(几)个线程来获得该锁。如果支持 Thread Execution Scheduling 选项，并且等待该锁的线程执行在 SCHED_FIFO、SCHED_RR 或者 SCHED_SPORADIC 调度策略之下，那么当锁可用时，这些等待线程就会按照优先级的顺序获得该锁。对于优先级相同的线程，写锁要先于读锁。如果不支持 Thread Execution Scheduling 选项的话，写锁是否要优先于读锁将会由具体的实现来定义。

如果使用一个未初始化的读-写锁对象调用了这个函数，其结果将是未定义的。

返回值

如果成功，pthread_rwlock_unlock( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_unlock( )函数可能会失败：

[EINVAL]　rwlock 指定的值没有指向一个已初始化了的读-写锁对象。

[EPERM]　当前的线程没有持有该读-写锁上的锁。

pthread_rwlock_unlock( )函数不会返回[EINTR]错误码。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlock_rdlock( )、pthread_rwlock_timedrdlock( )、pthread_rwlock_timedwrlock( )、pthread_rwlock_trywrlock( )、pthread_rwlock_wrlock( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 相一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。
- 对“描述”部分的更新如下：
  - 说明了写操作者会优先于读操作者的条件。
  - 删除了读-写锁拥有者的概念。
- 对“参见”部分进行了更新。

名称

pthread_rwlock_wrlock——为写入对读-写锁加锁。

调用形式

THR

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

描述

参见 pthread_rwlock_trywrlock( )。

名称

pthread_rwlockattr_destroy 和 pthread_rwlockattr_init——销毁和初始化读-写锁属性对象。

调用形式

THR
```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

描述

pthread_rwlockattr_destroy( )函数会销毁一个读-写锁属性对象。可以使用 pthread_rwlockattr_init( )函数重新初始化已销毁的 attr 对象；否则，在读-写锁属性对象被销毁之后，引用该对象的结果是未定义的。实现可以让 pthread_rwlockattr_destroy( )对 attr 所指向的对象设置成一个无效值。

pthread_rwlockattr_init( )函数将使用实现所定义的全部属性的默认值初始化读-写锁属性对象 attr。

如果使用已被初始化的 attr 属性对象调用 pthread_rwlockattr_init( )，结果是未定义的。

已经使用读-写锁属性对象初始化一个或多个读写锁之后，影响属性对象的任何函数(包括析构函数)都不会影响任何先前已被初始化的读-写锁。

返回值

如果成功，pthread_rwlockattr_destroy( )和 pthread_rwlockattr_init( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_destroy( )函数可能失败：

[EINVAL]　attr 指定的值无效。

如果出现了下面的情况，pthread_rwlock_init( )函数将会失败：

[ENOMEM]　对读-写锁属性对象进行初始化的内存不足。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlockattr_getpshared( )、pthread_rwlockattr_setpshared( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 相一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。初始化宏也从 SYNOPSIS 部分中删除了。
- 对“参见”部分进行了更新。

名称

pthread_rwlockattr_getpshared 和 pthread_rwlockattr_setpshared——获取和设置读-写锁属性对象的进程共享属性。

调用形式

THR TSH
```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
    restrict attr, int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```

描述

pthread_rwlockattr_getpshared( )函数将从 attr 所指向的已初始化的属性对象中获得 process-shared 属性的取值。pthread_rwlockattr_setpshared( )函数将在 attr 所指向的已初始化属性对象中设置其 process-shared 属性。

为了允许能够访问到为读-写锁分配的空间的任何线程都可以对读-写锁进行操作，应该将 process-shared 属性设置成 PTHREAD_PROCESS_SHARED，即便分配给该读-写锁的内存区是为多个进程所共享的。如果 process-shared 属性为 PTHREAD_PROCESS_PRIVATE，那么能够对读-写锁进行操作的线程将仅限于：那些与对该锁进行初始化的线程位于相同进程中的线程；如果不同进程中的线程试图对这样一种读-写锁进行操作的话，其行为是未定义的。process-shared 属性的默认取值为 PTHREAD_PROCESS_PRIVATE。

额外的属性、它们的默认值、以及用于获取和设置这些属性值的相关函数名都是由实现定义的。

返回值

若成功完成，pthread_rwlockattr_getpshared( )函数返回 0，并将 attr 的 process-shared 属性保存在 pshared 参数所引用的对象中；否则，就会返回一个错误号以指示所产生的错误。

如果成功，pthread_rwlockattr_setpshared( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_rwlock_getpshared( )和 pthread_rwlockattr_setpshared( )函数可能会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，pthread_rwlockattr_setpshared( )函数可能会失败：

[EINVAL] 为属性所指定的新值超出了该属性合法取值的范围。

这两个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

参见

pthread_rwlock_destroy( )、pthread_rwlockattr_destroy( )、thread_rwlockattr_init( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5。

Issue 6

为了与 IEEE Std 1003.1j-2000 相一致，进行了下面的修改：

- “调用形式”部分中的边缘代码(margin code)被修改到了 THR 中，以指示该项功能现在已经是 Threads 选项的一部分了(先前，它是 IEEE Std 1003.1j-2000 Read-Write Locks 选项的一部分，也是 XSI 扩展的一部分)。初始化宏也从“调用形式”部分中删除了。
- 在“描述”部分中提到了额外的属性都是由具体实现所定义的。
- 对“参见”部分进行了更新。

为了与 ISO/IEC 9899:1999 标准一致，在 pthread_rwlockattr_getpshared( )原型中加入了 restrict 关键字。

名称

pthread_rwlockattr_init——初始化读-写锁属性对象。

调用形式

XSI

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

描述

参见 pthread_rwlockattr_destroy( )。

名称

pthread_rwlockattr_setpshared——设置读-写锁属性对象的 process-shared 属性。

调用形式

XSI

```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```

描述

参见 pthread_rwlockattr_getpshared( )。

PDG

名称
pthread_self——获得调用线程的 ID。

调用形式
THR
```
#include <pthread.h>
pthread_t pthread_self(void);
```

描述
pthread_self( )函数会返回调用线程的线程 ID。

返回值
请参见“描述”部分。

错误
没有定义任何错误。

pthread_self( )函数不会返回错误码[EINTR]。

示例
无。

应用程序使用
无。

基本原理
pthread_self( )函数所提供的功能类似于 getpid( )函数为进程所提供的功能，并且这两者的原理也是相同的：创建调用没有向所创建的线程提供线程 ID。

未来方向
无

参见
pthread_create( )、pthread_equal( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史
首次发布于 Issue 5，包含进来的目的在于能够与 POSIX Threads Extension 一致。

Issue 6
pthread_self( )函数被标记为 Threads 选项的一部分。

名称

pthread_setcancelstate、pthread_setcanceltype 和 pthread_testcancel——设置可取消性状态。

调用形式

THR

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

描述

pthread_setcancelstate( )函数自动地将调用线程的可取消性状态设置成所给定的 state，并将先前的可取消性状态返回到 oldstate 所指向的地址处。state 的合法取值有：PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DISABLE。

pthread_setcanceltype( )函数自动地将调用线程的可取消性类型设置成所给定的 type，并将先前的可取消性类型返回到 oldtype 所指向的地址处。type 的合法取值有：PTHREAD_CANCEL_DEFERRED 和 PTHREAD_CANCEL_ASYNCHRONOUS。

任何新创建的线程，包括首次调用 main( )的线程，它们的可取消性状态和类型分别为 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_ASYNCHRONOUS。

pthread_testcancel( )函数在调用线程中将创建一个取消点。如果禁用了可取消性的话，pthread_testcancel( )函数将不会产生任何作用。

返回值

如果调用成功，pthread_setcancelstate( )和 pthread_setcanceltype( )函数将返回 0；否则返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_setcancelstate( )函数可能会失败：

[EINVAL] 所指定的状态不是 PTHREAD_CANCEL_ENABLE 或 PTHREAD_CANCEL_DISABLE。

如果出现了下面的情况，pthread_setcanceltype( )函数可能会失败：

[EINVAL]所指定的类型不是 PTHREAD_CANCEL_DEFERRED 或 PTHREAD_CANCEL_ASYNCHRONOUS。

这 3 个函数均不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

pthread_setcancelstate( )和 pthread_setcanceltype( )函数控制着一个线程的异步可取消点。为了能够以一种模块化的方式使用取消控制，就需要遵循一些规则。

可以将对象看作是过程的泛化。它是作为一个单元而编写的一系列过程和全局变量，并会由不为对象所知的客户加以调用。对象可能依赖于其他的对象。

首先，只应该在对象的入口处禁用可取消性，在这里永远都不要显式地使能可取消性。当从一个对象退出时，总是应当将可取消性状态恢复为其在对象入口处的取值。

这一点是从模块性讨论而得出的结论：一个对象的客户(或者使用该对象的一个对象的客户)禁用可取消性，这是因为如果线程在执行一系列动作时被取消，客户不希望考虑具体的清理工作。如果在这样一种状态下调用了一个对象，并且它对可取消性是使能的，这样就会给这个线程留下一个未决的取消请求，从而线程就被取消掉了，也就违背了客户对其加以禁用的初衷了。

其次，当进入一个对象时，可以将其可取消性类型显式地设置为 deferred 或者 asynchronous。但是对于可取消性状态，当从一个对象退出时，应当总是将其可取消性类型恢复为其在对象入口处的取值。

最后，在异步可取消线程中只能调用可安全取消的函数。

未来方向

无。

参见

pthread_cancel( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 5，包含进来的目的在于能够与 POSIX Threads Extension 对应。

Issue 6

pthread_setcancelstate( )、pthread_setcanceltype( )和 pthread_testcancel( )被标记为 Threads 选项的一部分。

名称

pthread_setconcurrency——设置并发级别。

调用形式

XSI

```
#include <pthread.h>
int pthread_setconcurrency(int new_level);
```

描述

参见 pthread_getconcurrency( )。

名称

pthread_setschedparam——动态线程调度参数访问(实时线程)。

调用形式

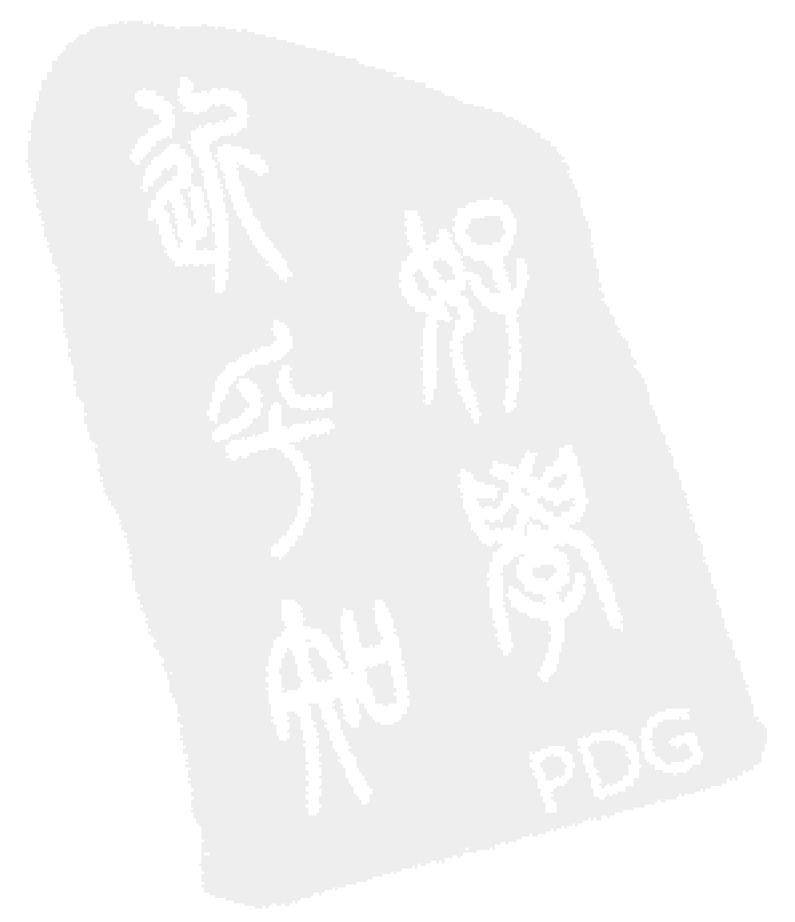
THR TPS

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

描述

参见 pthread_getschedparam( )。



名称

pthread_setschedprio——动态线程调度参数访问(实时线程)。

调用形式

THR TPS
```
#include <pthread.h>
int pthread_setschedprio(pthread_t thread, int prio);
```

描述

pthread_setschedprio( )函数会将线程 ID 为 thread 的线程的调度优先级设置为 prio 的给定值。关于这个函数将如何影响线程新优先级所属线程列表中的线程顺序的说明，请参阅“调度策略”部分。

如果 pthread_setschedprio( )函数失败了，将不会对目标线程的调度优先级进行修改。

返回值

如果调用成功，那么 pthread_setschedprio( )函数就会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，pthread_setschedprio( )函数可能会失败：

[EINVAL] 对于给定线程的调度策略，prio 的值无效。

[ENOTSUP] 试图将优先级设置为一种不受支持的值。

[EPERM] 调用者不具备对给定线程进行调度策略的恰当权限。

[EPERM] 实现不允许应用程序将优先级修改为所指定的值。

[ESRCH] thread 指定的值没有指向一个已存在的线程。

pthread_setschedprio( )函数不会返回错误码[EINTR]。

示例

无。

应用程序使用

无。

基本原理

pthread_setschedprio( )函数提供了一种可以临时性地提升优先级，然后将其降下来，而又不会对其他同优先级的线程产生不希望出现的副效应的方法。如果应用程序想要实现自己的绑定优先级翻转的策略，例如优先级继承或者优先级上限，这将是非常必要的。如果某种实现不支持 Thread Priority Protection 或者 Thread Priority Inheritance 选项的话，这种功能将会尤为重要，但是即便实现支持了这些选项，如果将优先级继承绑定到了其他资源，例如信号量，那么也会需要这种功能。

标准开发人员认为：尽管从思想上来讲可能会更倾向于通过修改 pthread_setschedparam( )的规范来解决这个问题，但是要进行这样一种修改已经为时过晚，因为这样可能会需要修

改一些实现。所以，也就引入了这个新的函数。

未来方向

无。

参见

调度策略、pthread_attr_getschedparam( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<pthread.h>。

变更历史

首次发布于 Issue 6，引入 Issue 6 的目的在于响应 IEEE PASC Interpretation 1003.1 #96。

名称

pthread_setspecific——线程特定数据的管理。

调用形式

THR

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
```

描述

参见 pthread_getspecific( )。

名称

pthread_testcancel——设置可取消性状态。

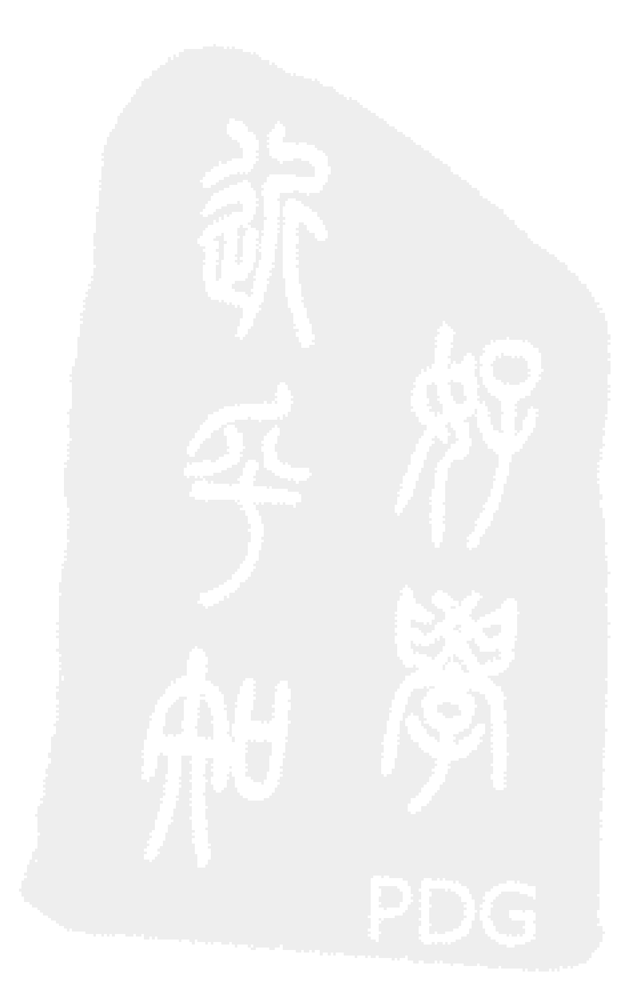
调用形式

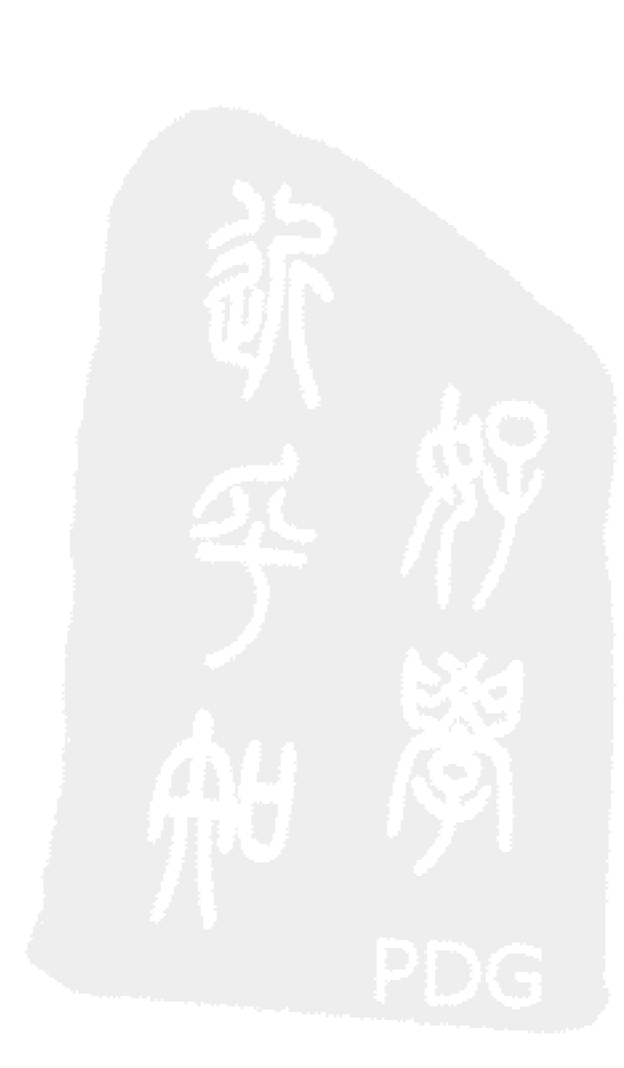
THR

```
#include <pthread.h>
void pthread_testcancel(void);
```

描述

参见 pthread_setcancelstate( )。





进程管理的 POSIX 标准

本附录包含来自 POSIX Standard for Process Management 的部分内容。POSIX 是被全世界所接受的开放操作系统接口标准。它由 IEEE 完成并被 ISO 和 ANSI 所认同。对 POSIX 标准的支持确保系统之间的代码移植性，在商业应用和政府合同中正逐渐占据统治地位。POSIX 标准是跨平台多核开发最广泛可用的方法。它同高级库兼容，例如 STAPL 和新的 Standard C++0x。POSIX 标准内容丰富，包含上千页。为了方便起见，我们节选了用于进程管理的标准。这些部分中包含的 API 函数要么在本书中提及过，要么就同开发多核应用程序有关。

下面的内容是经 IEEE 允许，对 IEEE std. 1003.1-2001，IEEE standard for Information Technology-Portable Operating System Interface(POSIX), Copyright 2001 的重印。

名称

posix_spawn 和 posix_spawnp——产生一个进程(高级实时)。

调用形式

SPN

```
#include <spawn.h>
```

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char * const envp[restrict]);
```

描述

posix_spawn( )和 posix_spawnp( )函数会从指定的进程映像创建一个新的进程(子进程)。新的进程映像是通过正规可执行文件调用新进程映像文件来构建的。

如果这个调用导致某个 C 程序的执行，那么该 C 程序应当如同被如下的 C 语言函数调用所激活：

```
int main(int argc, char *argv[]);
```

其中的 argc 为参数的数目，argv 为一个字符串指针数组，其中的指针指向参数本身。此外，下面的这个变量：

```
extern char **environ;
```

应当被初始化为一个指向字符串指针数组的指针，该数组中的指针指向环境字符串。

参数 argv 是一个字符串指针数组，数组中的指针指向以 null 结尾的字符串。该数组的最后一个成员为一个空指针，并且不会计入到 argc。这些字符串构成了新进程映像可用的参数列表。argv[0]中的指针应该指向一个同 posix_spawn( )或 posix_spawnp( )函数所启动的进程映像相关联的文件名。

参数 envp 为一个字符串指针数组，数组中的指针指向以 null 结尾的字符串。这些字符串构成了新进程映像的环境。环境数组是以一个空指针结尾的。

子进程的参数列表和环境列表结合起来的可用字节数为{ARG_MAX}。实现应当在系统文档(请参见 the Base Definitions volume of IEEE Std 1003.1-2001，Chapter 2，Conformance)中说明任何的列表开销，例如长度字、null 结束符、指针、对齐字节等，是否包括在总数中。

posix_spawn( )的 path 参数是用于标识要执行的新进程映像的路径名。

posix_spawnp( )的 file 参数会用于构建一个标识新的进程映像文件的路径名。如果在 file 参数中包含了一个斜杠字符，那么 file 参数就会用作新进程映像文件的路径名。否则，该文件的路径前缀就会通过搜索作为环境变量传递的 PATH 目录得到(请参见 the Base Definitions volume of IEEE Std 1003.1-2001，Chapter 8，Environment Variables)。如果没有

定义该环境变量，那么搜索的结果就会由具体实现定义。

如果 file_actions 是一个空指针，那么除了那些设置了 close-on-exec 标志 FD_CLOEXEC(参见 fcntl( ))的文件描述符外，在调用进程中打开的文件描述符在子进程中仍然会保持为打开状态。对于那些保持为打开状态的文件描述符，相应的打开文件描述的所有属性，包括文件锁(请参见 fcntl( ))，都会保持不变。

如果 file_actions 不为空，那么在子进程中打开的文件描述符是那些由于被 file_actions 指向的 spawn 文件动作对象修改而在调用进程中打开，同时在 spawn 文件动作被处理之后 FD_CLOEXEC 仍然保持打开的文件描述符。处理 spawn 文件动作的实际顺序为：

(1) 子进程的打开文件描述符的集合最初同调用进程打开的文件描述符集合相同。相应的打开文件描述的所有属性，包括文件锁(请参见 fcntl( ))，都保持不变。

(2) 信号掩码、信号默认动作、子进程有效用户 ID 和组 ID 都会根据 attrp 所指向的属性对象进行更改。

(3) 由 spawn 文件动作对象指定的文件动作会按照它们加入到 spawn 文件动作对象时的顺序执行。

(4) 任何设置有 FD_CLOEXEC 标志(请参见 fcntl( ))的文件描述符都会被关闭。

spawn 属性对象类型 posix_spawnattr_t 在<spawn.h>中定义。它将至少包含下面定义的属性。

如果在 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETPGROUP 标志，并且该对象的 spawn-pgroup 属性非零，那么就会将子进程的进程组设置为 attrp 所指向的对象的 spawn-pgroup 属性所指定的值。

作为一个特例，如果在 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETPGROUP 标志，并且该对象的 spawn-pgroup 属性为零，那么子进程就会位于一个新的进程组中，该进程组的 ID 等于它的进程 ID。

如果 attrp 指向的对象的 spawn-flags 属性中没有设置 POSIX_SPAWN_SETPGROUP 标志，那么新的子进程会继承父进程的进程组。

如果 attrp 所指向的对象的 spawn-flags 属性设置了 POSIX_SPAWN_SETSCHEDPARAM 标志，但没有设置 POSIX_SPAWN_SETSCHEDULER 标志，那么新的进程映像最初将会拥有调用进程的调度策略，调度参数由 attrp 指向的对象的 spawn-schedparam 属性指定。

如果 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETSCHEDULER 标志(不论是否设置了 POSIX_SPAWN_SETSCHEDPARAM 标志)，那么新的进程映像最初具有的调度策略是在 attrp 指向的对象的 spawn-schedpolicy 属性中指定的，而所使用的调度参数是在该对象的 spawn-schedparam 属性中指定的。

attrp 所指向的对象的 spawn-flags 属性有一个 POSIX_SPAWN_RESETIDS 标志，它控制着子进程的有效用户 ID。如果没有设置该标志，那么子进程将会继承父进程的有效用户 ID。如果设置了该标志，那么就会将子进程的有效用户 ID 重置为父进程的真实用户 ID。在这两种情况下，如果新进程映像文件的 set-user-ID 模式位被设置，那么在新的进程映像开始执行之前，子进程的有效用户 ID 将会变成文件的属主 ID。

attrp 指向的对象的 spawn-flags 属性中的这个 POSIX_SPAWN_RESETIDS 标志还控制着子进程的有效组 ID。如果没有设置该标志，那么子进程将会继承父进程的有效组 ID。

如果设置了该标志，那么就会将子进程的有效组 ID 重置为父进程的真实组 ID。在这两种情况下，如果新进程映像文件的 set-group-ID 模式位被设置，那么在新的进程映像开始执行之前，子进程的有效组 ID 将会变成文件的组 ID。

如果 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETSIGMASK 标志，那么子进程最初将会有着 attrp 指向的对象中的 spawn_sigmask 属性指定的信号掩码。

如果 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETSIGDEF 标志，那么同一个对象的 spawn_sigdefault 属性指定的信号在子进程中会设置为它们的默认动作。而那些被设置为在父进程中的默认动作的信号将会用被设置为在子进程中的默认动作。

被设置成由调用进程捕获的信号将会被设置为子进程中的默认动作。

除了 SIGCHLD 信号外，父进程设置为忽略的信号，子进程也会将其设置为忽略，除非在 attrp 指向的对象的 spawn-flags 属性中设置的 POSIX_SPAWN_SETSIGDEF 标志另外指定，并且在 attrp 指向的对象的 spawn-sigdefault 属性中也对这些信号加以说明。

如果将 SIGCHLD 信号设置成被调用进程忽略，那么将不指定子进程会将 SIGCHLD 信号设置成忽略还是设置成默认动作，除非在 attrp 指向的对象的 spawn-flags 属性中设置的 POSIX_SPAWN_SETSIGDEF 标志对信号另外指定，并且在 attrp 指向的对象的 spawn-sigdefault 属性对 SIGCHLD 信号加以说明。

如果 attrp 指针为空，那么就会使用默认值。

所有的进程属性，除了上述受到 attrp 指向的对象中的属性设置影响以及受到 file_actions 中指定的文件描述符操作影响的之外，都会出现在新的进程映像中，看上去就好像是调用了 fork( )创建了一个子进程，然后子进程调用了 exec 函数系列中的一个成员来执行这个新的进程映像。

在调用 posix_spawn( )和 posix_spawnp( )时，是否运行 fork 处理程序由具体的实现定义。

返回值

若成功完成，posix_spawn( )和 posix_spawnp( )会向父进程返回子进程的进程 ID，该 ID 值保存在非空 pid 参数所指向的变量中，并且返回零作为函数的返回值。否则，将不会创建任何子进程，非空 pid 参数所指向的变量的值未指定，并且返回一个错误号作为函数的返回值，以指示出现错误。如果 pid 参数为空指针，则子进程的进程 ID 不会返回给调用者。

错误

如果出现了下述情况，posix_spawn( )和 posix_spawnp( )函数可能将会失败：

[EINVAL]　file_actions 或 attrp 指定的值无效。

如果在调用进程从 posix_spawn( )和 posix_spawnp( )成功返回后出现了这种错误，子进程可能会以退出状态 127 退出。

如果由于任何会导致 fork( )或者 exec 函数系列之一失败的原因而导致 posix_spawn( )或者 posix_spawnp( )失败，那么所返回的错误值将和 fork( )及 exec 描述的一样(或者，如果在调用进程成功返回后出现了错误，那么子进程将会以退出状态 127 退出)。

如果 attrp 指向的对象的 spawn-flags 属性设置了 POSIX_SPAWN_SETPGROUP 标志，并且在更改子进程的进程组时，posix_spawn( )或者 posix_spawnp( )失败，那么所返回的错误值将和 setpgid( )中所描述的一样(或者，如果在调用进程成功地返回后出现了错误，那么

子进程将会以退出状态 127 退出)。

如果 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETSCHEDPARAM 标志，但是没有设置 POSIX_SPAWN_SETSCHEDULER 标志，那么如果由于任何可能会导致 sched_setparam( )调用失败的原因，而导致 posix_spawn( )或者 posix_spawnp( )失败，那么所返回的错误值将会和 sched_setparam( )中所描述的一样(或者，如果在调用进程成功地返回后出现了错误，子进程将会以退出状态 127 退出)。

如果 attrp 指向的对象的 spawn-flags 属性中设置了 POSIX_SPAWN_SETSCHEDULER 标志，并且如果由于任何会导致 sched_setscheduler( )调用失败的原因而导致 posix_spawn( )或者 posix_spawnp( )调用失败，那么所返回的错误值将会和 sched_setscheduler( )中所描述的一样(或者，如果在调用进程成功地返回后出现了错误，子进程将会以退出状态 127 退出)。

如果 file_actions 参数非空，并且指定要执行 close、dup2 或者 open 动作中的任何一个，这时如果由于任何会导致 close( )、dup2( )及 open( )函数调用失败的原因而导致 posix_spawn( )或者 posix_spawnp( )调用失败，那么所返回的错误值将会相应地和 close( )、dup2( )及 open( )中所描述的一样(或者，如果在调用进程成功地返回之后出现了错误，子进程将会以退出状态 127 退出)。除了 open( )所描述的错误之外，一次打开文件动作，本身就可能导致出现 close( )或者 dup2( )中所描述的错误。

示例

无。

应用程序使用

这两个函数为 Spawn 选项的一部分，并且无需在所有的实现上提供。

基本原理

引入 posix_spawn( )函数及其近亲 posix_spawnp( )的目的在于克服下面这些意识到的使用 fork( )时的困难：在没有换页或者动态地址转换的情况下，很难或者不可能实现 fork( )函数。

- 换页通常对于实时环境而言太慢了。
- 动态地址转换并非在所有 POSIX 有用的环境中可用。
- 进程太有用，以至于无法在没有地址转换或者其他 MMU 服务时，简单地将它从 POSIX 中去掉。

因此，POSIX 需要能够在没有地址转换或者其他 MMU 服务的前提下高效实现的进程创建和文件执行原语。

posix_spawn( )函数是作为一个库例程实现的，但是 posix_spawn( )和 posix_spawnp( )都被设计为内核操作。同时，尽管它们可以是很多 fork( )/exec 对的高效替代，但是引入它们的目的是为那些使用 fork( )存在困难的系统提供有用的进程创建原语，而不在于为 fork( )/exec 提供顺便的替代。

posix_spawn( )和 posix_spawnp( )在这个角度所起的作用，影响了它们的 API 设计。它并没有试图提供 fork( )/exec 的全部功能，即允许在子进程的创建和新的进程映像执行之间执行由用户所指定的任何操作，任何试图到达这种级别的尝试都需要提供一种编程语言作

为参数。相反，posix_spawn( )和 posix_spawnp( )都是进程创建原语，类似于 Start_Process 及 Start_Process_Search Ada 语言绑定包 POSIX_Process_Primitives，也类似于在很多非 UNIX 但具有一些 POSIX 特有附件的操作系统中的那些函数。

为达到其覆盖目标，posix_spawn( )和 posix_spawnp( )可以控制 6 种类型的继承：文件描述符、进程组 ID、用户和组 ID、信号掩码、调度和在父进程中被忽略的每个信号在子进程中是继续保持忽略还是将其设置为子进程中的默认动作。

为了让一个独立书写的子进程映像能够访问到由父进程所打开的甚至所创建的或者读取的数据流，同时不必通过特殊编码来了解将要使用哪个父文件和文件描述符，对文件描述符能够加以控制是必须的。为了控制子进程任务控制与父进程任务控制之间的关联方式，要求能够控制进程的组 ID。

对信号掩码及信号默认动作的控制足以支持 system( )的实现。尽管对 system( )的支持并不是 posix_spawn( )和 posix_spawnp( )的显式目标之一，但是在范围覆盖目标的“至少 50%”中是包括这一点的。

这里的意图是在于跨越 fork( )的普通文件描述符继承、指定的 spawn 文件动作的后继效果、以及跨越 exec 系列函数之一的普通文件描述符继承应当能够完全指定打开文件的继承关系。当子进程映像开始执行时，实现不需要对打开的文件描述符集合做出任何决策，这些决策已经由调用者做出了，并通过打开文件描述符集合、调用时指定的 FD_CLOEXEC 标志和调用中的 spawn 文件动作进行了表述。我们已经得到保证，在 POSIX Start_Process Ada 原语已经在库中实现的情况下，这种控制文件描述符继承关系的方法非常容易实现。

尽管我们能够找出 posix_spawn( )和 posix_spawnp( )的一些问题，但是没有哪种解决方案能够引入更少的问题。对子进程属性中那些没有通过 attrp 或者 file_actions 参数指定的环境修改必须在父进程中完成，并且由于父进程往往需要保存其上下文，与 fork( )/exec 实现类似功能相比，这种方法的代价更大。临时修改一个多线程的进程的环境也很复杂，因为所有的线程都必须就何时修改环境是安全的达成一致。不过，这种代价只有那些使用了额外功能的 posix_spawn( )和 posix_spawnp( )调用才会产生。由于大量的修改不是常见的情况，尤其不可能出现在时间紧迫的代码中，因此将大部分环境控制置于 posix_spawn( )和 posix_spawnp( )之外是合适的设计。

posix_spawn( )和 posix_spawnp( )函数并不具备 fork( )/exec 的全部功能，这是预料之中的。fork( )函数是一种极为强大的操作。我们并没有期望在无需任何特殊硬件需求的前提下，以简单而快速的函数复制其功能。值得指出的是：posix_spawn( )和 posix_spawnp( )非常类似于很多非 UNIX 系统中的进程创建操作。

要求

posix_spawn( )和 posix_spawnp( )的要求为：

- 在没有 MMU 或者非常规硬件的条件下，它们必须是可实现的。
- 必须与已有的 POSIX 标准相兼容。

额外的目标有：

- 它们应该被高效实现。
- 它们应该能够替代至少 50%的 fork( )典型执行。

- 具有 posix_spawn( )和 posix_spawnp( )而没有 fork( )的系统应该是有用的，至少对于实时应用的确如此。
- 具有 fork( )和 exec 系列函数的系统应当能够以库例程的形式实现 posix_spawn( )和 posix_spawnp( )。

两种语义

POSIX 的 exec 具有几个功能大致相同的调用序列。这些似乎是出于与现有做法兼容的目的。由于 posix_spawn*( )函数的现有做法与 POSIX 大不相同，我们感觉简明性重于兼容性。因此，posix_spawn*( )函数只有两个名字。

posix_spawn( )和 posix_spawnp( )的参数列表没有不同；只不过 posix_spawnp( )对第二个参数的解释要比 posix_spawn( )更精巧。

与 POSIX.5(Ada)的兼容性

来自 POSIX_Process_Primitives 包(该包为 Ada 语言对 POSIX.1 的绑定)的 Start_Process 和 Start_Search 以同 posix_spawn( )和 posix_spawnp( )类似的方式对 fork( )及 exec 的功能进行封装。最初，出于保持简明的目的，标准开发人员将 posix_spawn( )和 posix_spawnp( )的功能限制为 Start_Process 和 Start_Search 的功能的子集，不支持一些非默认的能力。不过，基于投票组提高或去除文件描述符映射的建议，以及一位 Ada Language Bindings 工作组成员的意见，标准开发人员决定 posix_spawn( )和 posix_spawnp( )的功能应该足够强大以实现 Start_Process 和 Start_Search。这里的基本原理是如果 Ada 语言对这样一种原语的绑定已经被认可为一种 IEEE 标准了，那么就没有理由不去认可一个功能相当的 C 语言绑定。posix_spawn( )和 posix_spawnp( )所提供的能力中，仅有 3 种是 Start_Process 和 Start_Search 未提供的，它们是选择性地指定子进程的进程组 ID、在子进程中将信号集合重置为默认处理，以及子进程的调度策略和参数。

如果要使用 posix_spawn( )实现 Ada 语言对 Start_Process 的绑定，该绑定就需要将一个空的信号掩码以及父进程的环境显式地传递给 posix_spawn( )，只要 Start_Process 的调用者允许将这些参数设置为默认值。这样要求的原因在于 posix_spawn( )没有提供这种默认形式。Start_Process 这种能够在执行过程中屏蔽用户指定信号的能力是 Ada 语言绑定所独有的一种功能，所以必须在绑定中与调用 posix_spawn( )进行分离的处理。

进程组

进程组继承域可以用来将子进程加入一个已有的进程组中。通过将 attrp 指向的对象的 spawn_pgroup 属性赋零值，setpgid( )机制就会将子进程放置到一个新的进程组中。

线程

如果没有 posix_spawn( )和 posix_spawnp( )函数，没有地址转换的系统仍可以使用线程来提供并发抽象。在很多情况下，线程创建就足够了，但这并不总是一种好的替代。与线程创建相比，posix_spawn( )和 posix_spawnp( )函数要“重”很多。进程具有几个线程所不具备的重要属性。即使没有地址转换，进程还会具有 base-and-bound 内存保护。每个进程有一个包括安全属性、文件功能以及强大的调度属性的进程环境。进程对非一致性存储体

系结构多处理器的抽象要好于线程，并且对于非紧密链接的活动，使用起来要更为方便。

posix_spawn( )和 posix_spawnp( )函数可能不会为所有配置带来多进程的支持。为了支持多进程，进程创建并不是对操作系统支持多进程提出的唯一要求。在某些环境下，支持多进程的总成本可能相当高。已有实践表明，对多进程的支持是不常见的，而在一些“小内核”中对多线程的支持则是常见的。因此，很有可能会继续维持操作系统只有一个进程。

异步错误通知

用库来实现 posix_spawn( )和 posix_spawnp( )，可能无法在创建子进程前检测到所有可能会出现的错误。IEEE Std 1003.1-2001 提供了一种通过特殊的退出状态对错误进行提示的机制，该状态值是从无法成功完成 spawn 操作的子进程返回的，并且可以使用 wait( )及 waitpid( )返回的状态值来检测。

stat_val 接口以及用于解释它的宏并不非常适合返回 API 错误的目的，但它们却是唯一的库实现的方法。因此，在 posix_spawn( )和 posix_spawnp( )函数已经成功返回之后，某种实现可能会导致子进程在产生进程过程中检测到的任何错误都会以退出状态 127 而退出。

标准开发人员曾经建议使用两个额外的宏来解释 stat_val。第一个为 WIFSPAWNFAIL，它会检测到一种状态，该状态表示子进程是由于在 posix_spawn( )或 posix_spawnp( )操作过程中而不是在实际的子进程映像执行过程中检测到了错误而退出的；第二个宏为 WSPAWNERRNO，如果 WIFSPAWNFAIL 的取值表示一个失败，那么就可以使用这个宏来提取相应的错误值了。不幸的是，投票组对此表示了强烈的反对，因为这种实现将会导致 posix_spawn( )或 posix_spawnp( )的库实现依赖于对 waitpid( )的内核修改，让 waitpid( )能够在 stat_val 中嵌入特殊信息以指示 spawn 失败。

要区分 spawn 错误与其他可能会由任意进程映像所返回的错误，IEEE Std 1003.1-2001 所保证的可为等待中的父进程访问到的子进程退出状态中的 8 比特是不够的。并没有要求退出状态的其他位在 stat_val 中是可见的，因此这些宏在库一级是无法严格实现的。对于此类 spawn 错误保留一个 127 的退出状态，这种做法与 system( )和 popen( )在信号失败时对该值的使用是一致的，在函数返回之后 snell 可以执行之前，如果这些操作出现了失败，system( )和 popen( )也会使用该值。因此，127 号退出状态并没有唯一地标识这类错误，也没有对失败的本质提供任何详细的信息。注意对 posix_spawn( )或 posix_spawnp( )的内核实现允许(也很提倡)将任何可能会出现的错误作为函数值返回，从而向父进程提供更为详细的失败信息。

这样，没有任何专门的宏可用于分离异步的 posix_spawn( )或 posix_spawnp( )错误。而实际上，在子进程上下文中，在子进程映像开始执行前，由 posix_spawn( )或 posix_spawnp( )操作检测到的错误，是通过将子进程的退出状态设置成 127 来报告的。调用进程可以对 wait( )或者 waitpid( )函数保存的 stat_val 使用 WIFEXITED 和 WEXITSTATUS 宏来检测 spawn 失败，这样就可以扩展到子进程映像退出时的其他状态值了(在父进程能够结论性地断定子进程映像已经开始执行之前)，而且这些值与退出状态值 127 不同。

未来方向

无。

参见

alarm( )、chmod( )、close( )、dup( )、exec、exit( )、fcntl( )、forx( )、kill、open( )、posix_spawn_file_actions_addclose( )、posix_spawn_file_actions_adddup2( )、posix_spawn_file_actions_addopen( )、posix_spawn_file_actions_destroy( )、<REFERENCE UNDEFINED>(posix_spawn_file_actions_init)、posix_spawnattr_destroy( )、posix_spawnattr_init( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getflags( )、posix_spawnattr_getpgroup( )、posix_spawnattr_getschedparam( )、posix_spawnattr_getschedpolicy( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault( )、posix_spawnattr_setflags( )、posix_spawnattr_setpgroup( )、posix_spawnattr_setschedparam( )、posix_spawnattr_setschedpolicy( )、posix_spawnattr_setsigmask( )、sched_setparam( )、sched_setscheduler( )、setpgid( )、setuid( )、stat( )、times( )、wait( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

应用了 IEEE PASC Interpretation 1003.1 #103，请注意对信号默认动作以及第二步的信号掩码进行了修改。

应用了 IEEE PASC Interpretation 1003.1 #132。

PDG

名称

posix_spawn_file_actions_addclose 和 posix_spawn_file_actions_addopen——向 spawn 文件动作对象添加关闭或者打开动作(高级实时)。

调用形式

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
    file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
    restrict file_actions, int fildes,
    const char *restrict path, int oflag, mode_t mode);
```

描述

这两个函数会向 spawn 文件动作对象添加或者删除一个关闭或打开动作。

spawn 文件动作对象的类型为 posix_spawn_file_actions_t(定义于<spawn.h>中)，使用它的目的在于为 posix_spawn( )或 posix_spawnp( )操作指定一系列执行动作，从而在给出了父进程已打开文件描述符集的情况下，能够让子进程找到打开文件描述符集。在 IEEE Std 1003.1-2001 中并没有为 posix_spawn_file_actions_t 类型定义比较或者赋值操作符。

在将一个 spawn 文件动作对象传递给 posix_spawn( )或 posix_spawnp( )时，该对象会指定调用进程中的打开文件描述符集合是如何被转换为所产生进程的潜在打开文件描述符集合的。这种转换应当看上去似乎所指定的动作序列确实被执行了一次一样，其执行环境就在所产生进程(先于新进程映像的执行)的上下文中，其执行的顺序就按照动作添加到对象中的次序；此外，当新的进程映像执行时，凡是设置有 FD_CLOEXEC 标志的(来自于这个新集合的)任何文件描述符都会被关闭(请参阅 posix_spwan( ))。

posix_spawn_file_actions_addclose( )函数会向 file_actions 指向的对象添加一个 close 动作，从而导致在使用该文件动作对象产生一个新进程时，文件描述符 fildes 会被关闭(就好像是调用了 close(fildes)一样)。

posix_spawn_file_actions_addopen( )函数会向 file_actions 指向的对象添加一个 open 动作，从而导致在使用该文件动作对象产生一个新进程时，以 path 命名的文件会被打开(就好像调用了 open(path，oflag，mode)函数一样，并且如果所返回的文件描述符不是 fildes，就会将其修改为 fildes)。如果 fildes 已经是一个打开文件描述符了，那么在新文件被打开之前就会将其关闭。

由 path 所描述的字符串会被 posix_spawn_file_actions_addopen( )函数所复制。

返回值

每当成功完成，这两个函数都会返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这两个函数将会失败：

[EBADF] fildes 指定的值为负值或者大于等于{OPEN_MAX}。

如果出现了下述情况，这两个函数可能会失败：

[EINVAL] file_actions 指定的值无效。

[ENOMEM] 向 spawn 文件动作对象进行添加时内存不足。

如果在调用之时无法执行所指定的操作，那么不会认为是传递给这两个函数用以指定一个文件描述符的 fildes 参数的错误。所有此类错误都会在后来 posix_spawn( )或 posix_spawnp( )操作过程中使用相关文件动作对象时检测到。

示例

无。

应用程序使用

这两个函数均为 Spawn 选项的一部分，并且不需要在所有的实现上提供。

基本原理

一个 spawn 文件动作对象可以被初始化为包含 close( )、dup2( )及 open( )操作的一个有序序列，它们被 posix_spawn( )或 posix_spawnp( )使用，以便在调用 posix_spawn( )或 posix_spawnp( )时，产生进程可以的从父进程打开文件描述符集合继承来的打开文件描述符集合。曾经有人建议只需 close( )和 dup2( )两个操作就足以对文件描述符重新安排了，并且对于那些产生进程需要打开来使用的文件，可以按下面两种方法进行处理：可以让调用进程在调用 posix_spawn( )或 posix_spawnp( )之前把它们打开(之后再关闭它们)，也可以(在 argv 中)将文件名传递给所产生的进程从而让它自己打开。标准开发人员的推荐做法是在实践中，应用程序可以使用这两种方法中的任何一种，因为这样应用程序就总是可以得到失败的打开操作的详细错误状态了。不过，标准的开发人员感觉允许一个 spawn 文件动作对象指定打开操作仍然是恰当的，因为：

(1) 这样就与 POSIX.5(Ada)的功能一致了。

(2) 这样就可以支持一些 POSIX 程序中常常会使用到的、从 shell 进行调用的 I/O 重定向范型了。当此类程序为一个子进程时，可能就无法将其设计成由自己打开文件。

(3) 这样就允许那些如果由父进程来执行可能会失败或者与文件的属主/访问权限相冲突的文件打开了。

关于上面提到的第二点，注意 spawn 打开文件动作为 posix_spawn( )和 posix_spawnp( )提供的功能与 shell 重定向操作符为 system 所提供的功能相同，只是它没有干涉到 shell 的执行。例如：

```
system ("myprog <file1 3<file2");
```

关于上面所提到的第三点，注意如果调用进程需要为所产生的进程打开一个或者多个文件，但是没有充足的空闲文件描述符，那么就有必要使用打开动作了，以便在其他(那些在父进程中必须保持为打开状态的)文件描述符已经被关闭之后，在子进程的上下文中能够进行 open 操作。

此外，如果一个父进程是从一个 set-user-id 模式位被设置的文件执行的，并且在产生属性中设置了 POSIX_SPAWN_RESETIDS 标志，那么在父进程中所创建的文件将会(很有

可能不正确地)将父进程的有效用户 ID 设置成自己的属主，而在 posix_spawn( )或 posix_spawnp( )过程中通过 open()动作创建的文件则有可能会成功地打开一个实际用户应当没有权限打开的文件，或者无法打开一个实际用户应当具有权限打开的文件。

文件描述符映射

标准开发人员最初曾经建议使用一个数组来指定映射，该映射将子进程文件描述符映射回父进程文件描述符。投票组指出：在没有提供一个或者多个空闲文件描述符记录项(可能是不可用的)的情况下，在 posix_spawn( )或 posix_spawnp( )的库实现中对文件描述符进行任意重新洗牌是不可能的。这样一种数组要求具体的实现开发一个复杂的策略来达到期望的映射，同时不会不经意地在错误的时间关闭了错误的文件描述符。

Ada Language Bindings 工作小组的一位成员指出，在已获认可的 POSIX 进程原语的 Ada Language 的 Start_Process 系列中，使用了一种调用者指定的文件动作集，以一种灵活的方式对正常的 fork( )/exec 文件描述符继承性语义进行了修改，但是此类问题并不存在，因为用于判断如何达到最终的文件描述符映射完全是应用程序的任务。而且，尽管文件动作接口初看上去挺可怕，实际上以库或者内核的形式实现都是非常简单的。

未来方向

无。

参见

close( )、dup( )、open( )、posix_spawn( )、posix_spawn_file_actions_adddup2( )、posix_spawn_file_actions_destroy( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

应用了 IEEE PASC Interpretation 1003.1 #105，在描述中添加了一条注释，path 所指向的字符串由 posix_spawn_file_actions_addopen( )函数复制。

名称

posix_spawn_file_actions_adddup2——向 spawn 文件动作对象添加 dup2 动作(高级实时)。

调用形式

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
    file_actions, int fildes, int newfildes);
```

描述

posix_spawn_file_actions_adddup2( )函数会向 file_actions 指向的对象添加一个 dup2( )动作，从而当使用该文件动作对象产生一个新的进程时，文件描述符 fildes 被复制为 newfildes(就好像调用了 dup2(fildes, newfildes)一样)。

spawn 文件动作对象的定义与 posix_spawn_file_actions_addclose( )中的定义相同。

返回值

每当成功完成，posix_spawn_file_actions_adddup2( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下述情况，posix_spawn_file_actions_adddup2( )函数就会失败：

[EBADF]　fildes 或者 newfildes 指定的值为负数或者大于等于{OPEN_MAX}。

[ENOMEM]　向 spawn 文件动作对象进行动作添加时内存不足。

如果出现了下面的情况，posix_spawn_file_actions_adddup2( )函数可能会失败：

[EINVAL]　file_actions 指定的值无效。

如果在调用之时无法执行所指定的操作，那么不会认为是传递给 posix_spawn_file_actions_adddup2( )函数的用以指定一个文件描述符的 fildes 参数的错误。所有的此类错误都会在后来 posix_spawn( )或 posix_spawnp( )的操作过程中使用相关文件动作对象时检测到。

示例

无。

应用程序使用

posix_spawn_file_actions_adddup2( )函数是 Spawn 选项的一部分，并且不需要在所有的实现提供。

基本原理

参见 posix_spawn_file_actions_addclose( )中的“基本原理”部分。

未来方向

无。

参见

dup( )、posix_spawn( )、posix_spawn_file_actions_addclose( )、posix_spawn_file_actions_destroy( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和 <spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

应用了 IEEE PASC Interpretation 1003.1 #104，请注意[EBADF]错误除了可以应用于参数 fildes 外还可以应用于 newfildes。

名称

posix_spawn_file_actions_addopen——将 open 动作加到 spawn 文件动作对象(高级实时)。

调用形式

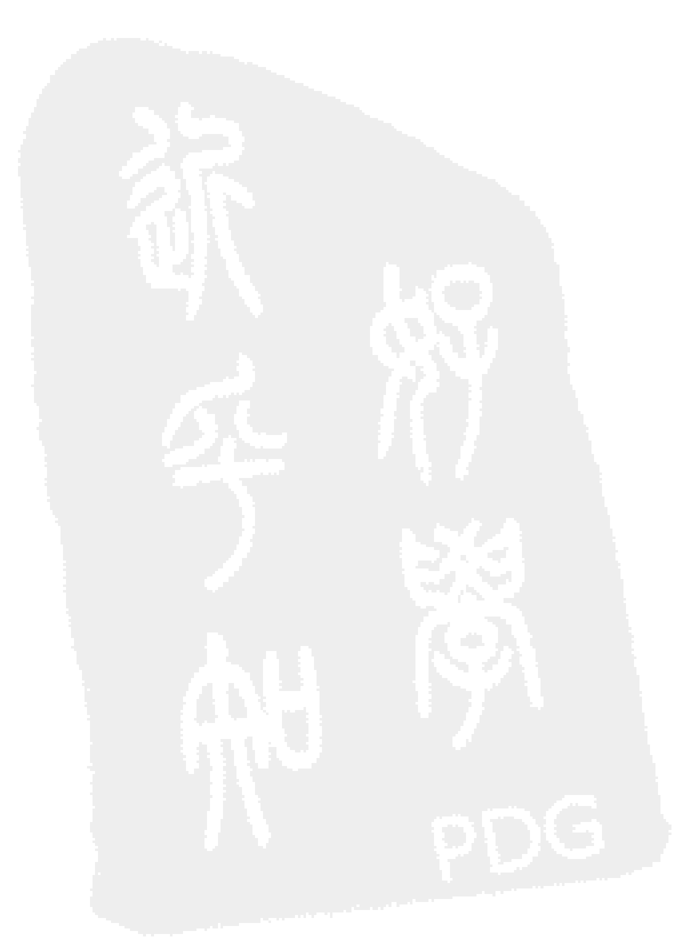
SPN

```
#include <spawn.h>

int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
    restrict file_actions, int fildes,
    const char *restrict path, int oflag, mode_t mode);
```

描述

参见 posix_spawn_file_actions_addclose( )。



名称

posix_spawn_file_actions_destroy 和 posix_spawn_file_actions_init—— 销毁和初始化 spawn 文件动作对象(高级实时)。

调用形式

SPN
```
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
    file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```

描述

posix_spawn_file_actions_destroy( )函数会销毁 file_actions 指向的对象；实际上这个对象就会变成未初始化的。实现可以让 posix_spawn_file_actions_destroy( )将 file_actions 所指向的对象设置成一个无效值。使用 posix_spawn_file_actions_init( )则可以将一个已经被销毁的 spawn 文件动作对象重新初始化；在销毁之后对该对象进行再次引用的结果是未定义的。

posix_spawn_file_actions_init( )函数会将 file_actions 所指向对象初始化为不包含 posix_spawn( )或者 posix_spawnp( )要执行的任何文件动作。

spawn 文件动作对象的定义与 posix_spawn_file_actions_addclose( )中的定义相同。

对一个已经被初始化了的 spawn 文件动作对象进行初始化的结果是未定义的。

返回值

每当成功完成，这两个函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，posix_spawn_file_actions_init( )函数就会失败：

[ENOMEM]　对 spawn 文件动作对象进行初始化时内存不足。

如果出现了下面的情况，posix_spawn_file_actions_destroy( )函数可能会失败：

[EINVAL]　file_actions 指定的值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 选项的一部分，不需要在所有的实现上提供。

基本原理

参见 posix_spawn_file_actions_addclose( )中的“基本原理”部分。

未来方向

无。

参见

posix_spawn( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

在“调用形式”中，不再需要包含<sys/types.h>。

名称

posix_spawnattr_destroy 和 posix_spawnattr_init——销毁或者初始化 spawn 属性对象(高级实时)。

调用形式

SPN

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

描述

posix_spawnattr_destroy( )函数会销毁一个 spawn 属性对象。使用 posix_spawnattr_init( )可以对一个已经销毁了的 attr 属性对象进行重新初始化；在销毁之后对该对象进行再次引用的结果是未定义的。实现可以让 posix_spawnattr_destroy( )将 attr 指向的对象设置成一个无效值。

posix_spawnattr_init( )函数会为所有单个属性用实现所使用的默认值对 spawn 属性对象 attr 进行初始化。如果指定一个已经被初始化了的 attr 属性对象调用 posix_spawnattr_init，结果是未定义的。

spawn 属性对象的类型为 posix_spawnattr_t(定义于<spawn.h>中)，并且可以使用它来指定跨越 spawn 操作的进程属性继承性。在 IEEE Std 1003.1-2001 中并没有为 posix_spawnattr_t 类型定义比较或者赋值操作符。

每种实现都应将其所使用的各个属性及其默认值写入文档，除非这些默认值已经在 IEEE Std 1003.1-2001 中进行了定义。没有被 IEEE STD 1003.1-2001 所定义的属性，它们的默认值以及设置或者获取这些属性取值的相关函数是由具体的实现定义的。

可以使用所生成的 spawn 属性对象(可能会通过设置一些单个属性值来修改)来变更 posix_spawn( )或者 posix_spawnp( )的行为。在 spawn 属性对象被 posix_spawn( )或者 posix_spawnp( )调用以产生一个进程之后，影响属性对象的任何函数(包括析构函数)都不会影响到任何以这种方式已经产生的进程。

返回值

每当成功完成，posix_spawnattr_destroy( )和 posix_spawnattr_init( )均返回 0；否则，返回一个错误号以指示所出现的错误。

错误

如果出现了下面的情况，posix_spawnattr_init( )函数将会失败：

[ENOMEM]　对 spawn 属性对象进行初始化时内存不足。

如果出现了下面的情况，posix_spawnattr_destroy( )函数可能会失败：

[EINVAL]　file_actions 指定的值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 选项的一部分，并且不需要在所有的实现上提供。

基本原理

在 IEEE Std 1003.1-2001 中所提出的最初的 spawn 接口将某些属性定义为一种结构，这些属性指定跨越 spawn 操作的进程属性的继承性。为了能够在它们恰当的选择下分离可选的单个属性(即：spawn_schedparam 和 spawn_schedpolicy 属性依赖于 Process Scheduling 选项)，同时也出于可扩展性及与更新的 POSIX 接口相一致的考虑，将属性接口修改成一种非透明的数据类型。现在这个接口的组成为：代表 spawn 属性对象的 posix_spawnattr_t 类型、用于初始化或者销毁属性对象的相关函数、以及用于设置或者获取单个属性的函数。尽管与最初的结构相比，新的面向对象的接口要稍显冗长，但是它的使用简单，更具扩展性，并且易于实现。

未来方向

无。

参见

posix_spawn( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getflags( )、posix_spawnattr_getpgroup( )、posix_spawnattr_getschedparam( )、posix_spawnattr_getschedpolicy( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault( )、posix_spawnattr_setflags( )、posix_spawnattr_setpgroup( )、posix_spawnattr_setsigmask( )、posix_spawnattr_setschedpolicy( )、posix_spawnattr_setschedparam( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<**spawn.h**>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

应用了 IEEE PASC Interpretation 1003.1 #106，请注意：对一个已经初始化了的 spawn 属性对象进行初始化的结果是未定义的。

名称

posix_spawnattr_getflags 和 posix_spawnattr_setflags——获取和设置一个 spawn 属性对象的 spawn-flags 属性(高级实时)。

调用形式

SPN
```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
    short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

描述

posix_spawnattr_getflags( )函数会从 attr 指向的属性对象得到 spawn-flags 的值。

posix_spawnattr_setflags( )函数会设置 attr 指向的已初始化的属性对象的 spawn-flags 属性。

可以使用 spawn-flags 属性来指示调用 posix_spawn( )或者 posix_spawnp( )时子进程映像中哪些进程属性会被改变，它是 0 个或多个下列标志进行按位 OR：

POSIX_SPAWN_RESETIDS
POSIX_SPAWN_SETPGROUP
POSIX_SPAWN_SETSIGDEF
POSIX_SPAWN_SETSIGMASK
PS POSIX_SPAWN_SETSCHEDPARAM
POSIX_SPAWN_SETSCHEDULER

这些标志都定义在<spawn.h>中。该属性的默认值设置就好像没有设置任何标志。

返回值

每当成功完成，posix_spawnattr_getflags( )返回 0，并将 attr 的 spawn-flags 属性保存到 flags 参数指向的对象中；否则，就会返回一个错误号以指示所产生的错误。

每当成功完成，posix_spawnattr_setflags( )返回 0；否则，就会返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这两个函数就会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，posix_spawnattr_setflags( )函数可能会失败：

[EINVAL] 正在设置的属性值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 选项的一部分，并且不需要在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

posix_spawn( )、posix_spawnattr_destroy( )、posix_spawnattr_init( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getpgroup( )、posix_spawnattr_getschedparam( )、posix_spawnattr_getschedpolicy( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault()、posix_spawnattr_setpgroup( )、posix_spawnattr_setschedparam( )、posix_spawnattr_setschedpolicy( )、posix_spawnattr_setsigmask( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

名称

posix_spawnattr_getpgroup 和 posix_spawnattr_setpgroup——获取和设置一个 spawn 属性对象的 spawn-pgroup 标志(高级实时)。

调用形式

SPN
```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
    pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

描述

posix_spawnattr_getpgroup( )函数会从 attr 指向的属性对象得到 spawn-pgroup 属性的值。

posix_spawnattr_setpgroup( )函数会在一个 attr 指向的已初始化的属性对象中设置 spawn-pgroup 属性。

spawn-pgroup 属性代表了在产生操作中新进程映像将要加入的进程组(如果在 spawn-pgroup 属性中设置了 POSIX_SPAWN_SETPGROUP 标志)。该属性的默认值为 0。

返回值

每当成功完成，posix_spawnattr_getpgroup( )就会返回 0，并将 attr 的 spawn-pgroup 属性保存到 pgroup 参数指向的对象中；否则，返回一个错误号以指示所产生的错误。

每当成功完成，posix_spawnattr_setpgroup( )就会返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这两个函数就可能会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，posix_spawnattr_setpgroup( )函数可能会失败：

[EINVAL] 正在设置的属性值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 选项的一部分，并且不需要在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

posix_spawn( )、posix_spawnattr_destroy( )、posix_spawnattr_init( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getflags( )、posix_spawnattr_getschedparam( )、posix_spawnattr_getschedpolicy( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault( )、posix_spawnattr_setflags( )、posix_spawnattr_setschedparam( )、posix_spawnattr_setschedpolicy( )、posix_spawnattr_setsigmask( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001 和 <spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

名称

posix_spawnattr_getschedparam 和 posix_spawnattr_setschedparam——获取和设置 spawn 属性对象的 spawn-schedparam 标志(高级实时)。

调用形式

SPN PS
```
#include <spawn.h>
#include <sched.h>
```
```
int posix_spawnattr_getschedparam(const posix_spawnattr_t *
    restrict attr, struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

描述

posix_spawnattr_getschedparam( )函数从 attr 指向的属性对象得到 spawn-schedparam 属性的值。

posix_spawnattr_setschedparam( )函数会在 attr 指向的已初始化的属性对象中设置 spawn-schedparam 属性。

spawn-schedparam 属性代表了在产生操作中将要赋给新进程映像的调度参数(如果在 spawn-flags 属性中设置了 POSIX_SPAWN_SETSCHEDPARAM 标志)。没有指定该属性的默认值。

返回值

每当成功完成，posix_spawnattr_getschedparam( )返回 0，并将 attr 的 spawn_schedparam 属性保存到 schedparam 参数指向的对象中；否则，返回一个错误号以指示所产生的错误。

每当成功完成，posix_spawnattr_setschedparam( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这两个函数可能就会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，posix_spawnattr_setschedparam( )函数可能会失败：

[EINVAL] 正在设置的属性值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 和 Process Scheduling 选项的一部分，不需要在所有的实现上提供。

基本原理

无。

未来方向

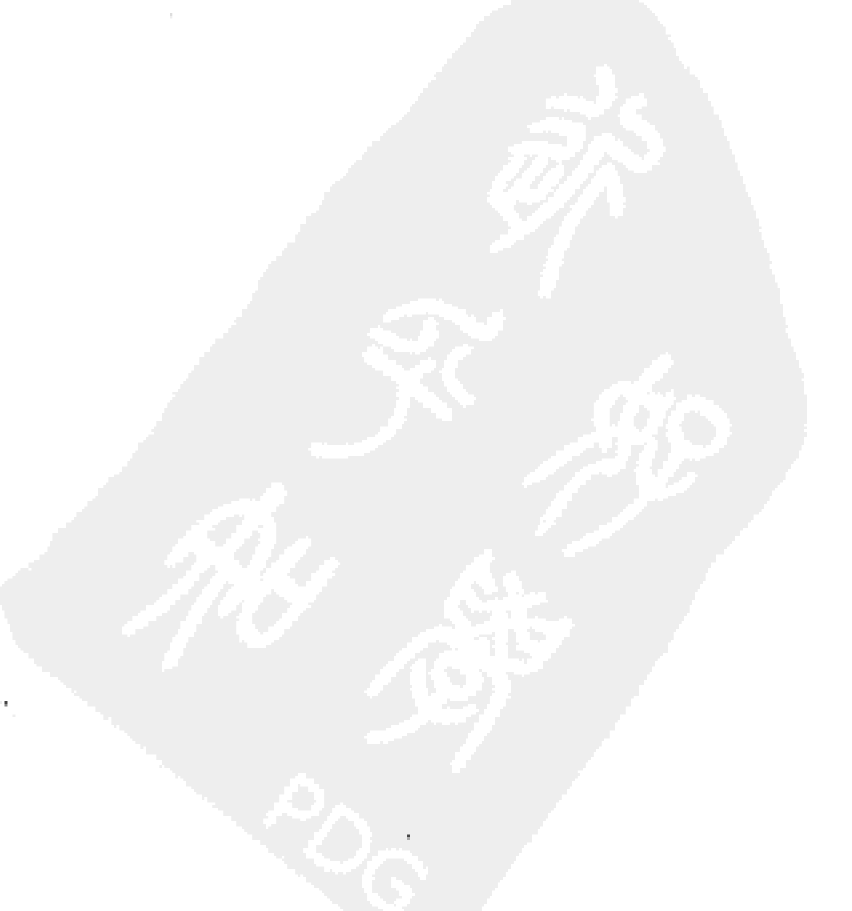
无。

参见

posix_spawn( )、posix_spawnattr_destroy( )、posix_spawnattr_init( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getflags( )、posix_spawnattr_getpgroup( )、posix_spawnattr_getschedpolicy( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault( )、posix_spawnattr_setflags( )、posix_spawnattr_setpgroup( )、posix_spawnattr_setschedpolicy( )、posix_spawnattr_setsigmask( )、posix_spawnp( )、the Base Definitions volume of IEEE Std 1003.1-2001、<sched.h>和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。



名称

posix_spawnattr_getschedpolicy 和 posix_spawnattr_setschedpolicy——获取和设置一个spawn 属性对象的 spawn-schedpolicy 标志(高级实时)。

调用形式

SPN PS

```
#include <spawn.h>
#include <sched.h>
```

```
int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
    restrict attr, int *restrict schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```

描述

posix_spawnattr_getschedpolicy( )函数从 attr 指向的属性对象中得到 spawn-schedpolicy 属性的值。

posix_spawnattr_setschedpolicy( )函数在 attr 指向的已初始化的属性对象中设置 spawn-schedpolicy 属性。

spawn-schedpolicy 属性代表了在产生操作中将要赋给新进程映像的调度策略(如果在 spawn-flags 属性中设置了 POSIX_SPAWN_SETSCHEDPOLICY 标志)。没有指定该属性的默认值。

返回值

每当成功完成，posix_spawnattr_getschedpolicy( )返回 0，并将 attr 的 spawn-schedpolicy 属性保存到 schedpolicy 参数指向的对象中；否则，返回一个错误号以指示所产生的错误。

每当成功完成，posix_spawnattr_setschedpolicy( )函数返回 0；否则，返回一个错误号以指示所产生的错误。

错误

如果出现了下面的情况，这两个函数可能就会失败：

[EINVAL] attr 指定的值无效。

如果出现了下面的情况，posix_spawnattr_setschedpolicy( )函数可能会失败：

[EINVAL] 正在设置的属性值无效。

示例

无。

应用程序使用

这两个函数为 Spawn 和 Process Scheduling 选项的一部分，不需要在所有的实现上提供。

基本原理

无。

未来方向

无。

参见

posix_spawn( )、posix_spawnattr_destroy( )、posix_spawnattr_init( )、posix_spawnattr_getsigdefault( )、posix_spawnattr_getflags( )、posix_spawnattr_getpgroup( )、posix_spawnattr_getschedparam( )、posix_spawnattr_getsigmask( )、posix_spawnattr_setsigdefault( )、posix_spawnattr_setflags( )、posix_spawnattr_setpgroup( )、posix_spawnattr_setschedparam( )、posix_spawnattr_setsigmask( )、posix_spawnp( )、the Base Definitions volume of IEEE STD 1003.1-2001、<sched.h>和<spawn.h>。

变更历史

首次发布于 Issue 6。来源于 IEEE Std 1003.1d-1999。

PDG