第4章 ACE轻量级OS并发机制的OO封装

Douglas C. Schmidt

摘 要

本论文描述ACE面向对象的线程封装C++类库的设计，该类库使程序员与Solaris线程、POSIX pthreads及Win32线程之间的差异相屏蔽；并从最终用户和内部设计的视角来展现其体系结构，并讨论了关键的设计和实现问题。读者将获得对总体设计方法，以及在多种软件质量因素，如质量、可移植性及可扩展性之间所做的权衡的理解。

4.1 介绍

某些类型的分布式应用通过使用并发模式执行任务来从中获益。对于多处理器平台上的网络服务器，并发特别有助于改善性能和简化编程。对于服务器应用，使用线程来并发地处理多客户请求常常比下面的设计方法要更为方便和更不易出错：

- 在传输层接口对请求进行人工的序列化；
- 在内部对请求进行排队，并依次处理它们；
- 为每一客户请求派生一个重量级的进程。

本论文描述ACE自适配通信环境[1]中包含的C++类库。ACE封装并增强了由Solaris 2.x线程[2]、POSIX Pthreads[3]及Win32线程[4]所提供的轻量级并发机制。

在本论文中介绍的材料面对的是那些有兴趣了解线程的面向对象（OO）并发编程的战略和战术的技术人员。读者被假定熟悉一般的OO设计和编程技术（比如设计模式[5]、应用构架[6]、模块性、信息隐藏和对象建模[7]）、OO表示法（比如OMT[8]），基本的的C++编程语言特性（比如类、继承、动态绑定和参数化类型[9]）、基本的UNIX系统编程概念（比如进程管理、虚拟内存和进程间通信[10]），以及网络术语（比如客户/服务器体系结构[11]、RPC[12]、CORBA[13]和TCP/IP[14, 15]）。

一般而言，理解本论文并不需要对并发有深入的了解；特别地，也不需要对Solaris/POSIX/Win32多线程和同步机制有深入的了解。对并发编程和多线程的综述在4.3介绍，在其中定义了关键的术语，并概述了多种用于在Solaris 2.x、POSIX pthreads和Win32线程上进行并发编程的可选机制。

本论文被组织如下：4.2给出对ACE OS线程封装库的目标的综述，并概述该库的组件的面向对象体系结构。4.3一般性地介绍并发编程的相关背景材料，并特别介绍了Solaris多线程模型。4.4介绍一种激发了ACE线程封装库的设计的最终用户视点，并聚焦于从并发客户/服务器应用中精选的一个使用实

例。4.5详细描述ACE线程封装库的公共接口和内部设计。4.6介绍了若干例子，演示在4.5中定义的OO组件。最后，4.7给出结束语。

4.2 ACE OO并发机制综述

4.2.1 总体目标

与前几代SunOS相比，现代操作系统（比如Solaris、OSF/1、Windows NT和OS/2）的一种显著特性是其集成的对内核级和用户级多线程及同步的支持。但是，现有的与这些操作系统一起发布的多线程和同步机制都是用C写成的相对低级的API。混合使用C++类和低级C API来开发应用给开发者造成了不可接受的负担。在单个应用中混合这两种风格将导致面向对象和过程编程之间的“阻抗”失配。这样一种混合的编程风格让人迷惑，并会带来慢性的维护问题。

为避免让每个开发者实现他们自己特别的OS线程机制的C++包装，ACE提供了一组在此论文中描述的面向对象的并发组件。这些ACE组件为并发编程提供了可移植和可扩展的接口。该接口简化了用于开发客户和服务器的线程管理和同步机制。它已被移植到POSIX pthreads标准的许多试验版本[3]、Solaris线程[2]、Microsoft Win32线程[4]，以及VxWorks tasks。

4.2.1.1 总体要求

与封装和简化OS线程机制的并发底层的目的相结合，ACE OO线程封装类库正在被开发以响应下列常见的应用要求：

- *简化程序设计*：通过允许多个应用任务使用传统的同步编程抽象独立地执行（比如CORBA远地方法请求）来实现；
- *透明地改善性能*：通过使用像SPARCcenter 1000和2000共享内存对称多处理器这样的硬件平台的并行处理能力来实现；
- *显式地改善性能*：通过减少数据拷贝，以及通过重叠的通信计算来实现；
- *改善可感知的响应时间*：对于交互式应用（比如用户接口或是网络管理应用），通过将分离的线程与应用中的不同任务或服务相关联来实现。

4.2.1.2 设计目标

ACE OO线程类库被开发用以实现下列设计目标：

- 通过使开发者能够在他们的并发应用中始终一致地使用C++和OO，促进编程风格的一致性。
- 改善底层并发机制的可移植性和可复用性。
- 减少为使应用线程安全化所需做出的强制性变动。

- 排除或减少发生那些微妙的同步错误的潜在可能性。
- 增强抽象和模块性，而又不牺牲性能。

4.2.2 ACE OO线程封装组件的体系化综述

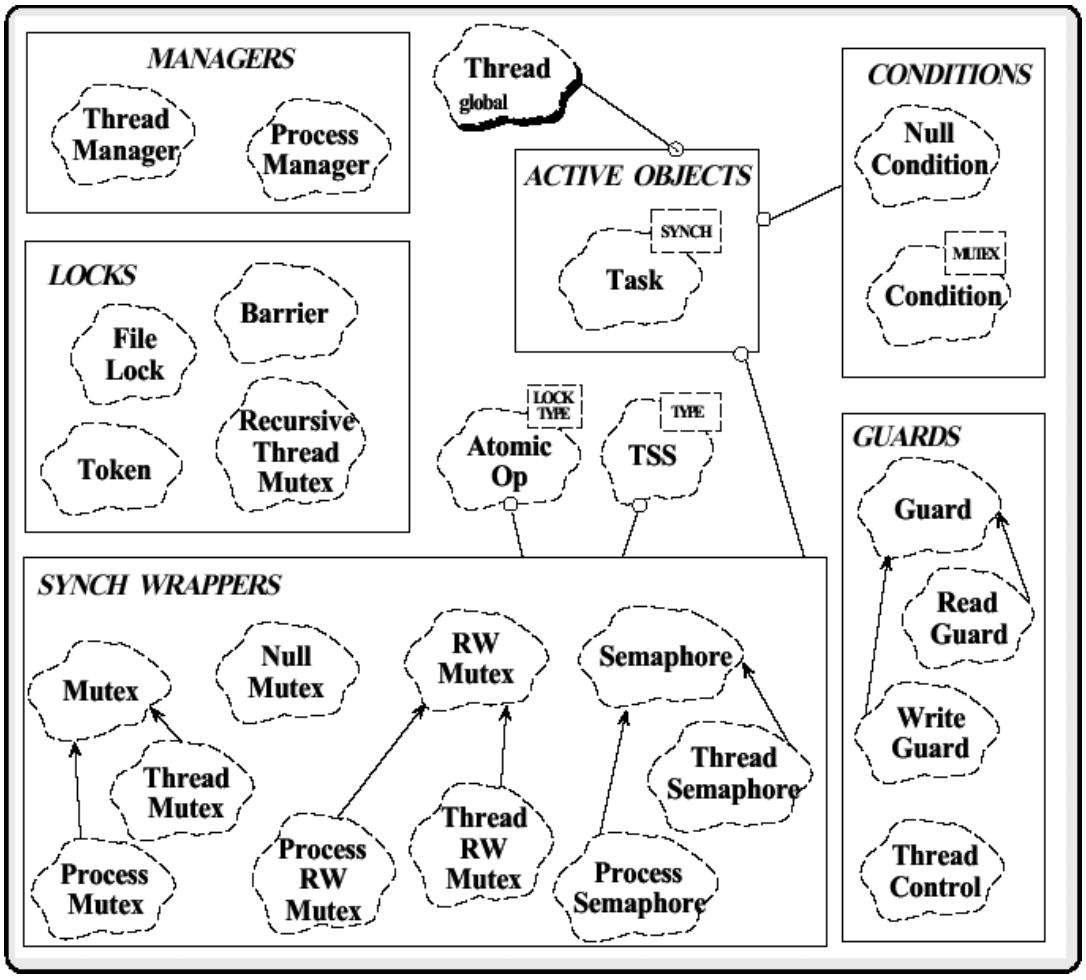

图4-1 ACE OO线程封装组件的对象模型

图4-1中的Booch对象模型演示了ACE线程封装类库中的组件。这些组件包括下面描述的C++类和类属。

4.2.2.1 ACE锁（Lock）类属

- Mutex、Thread_Mutex和Process_Mutex：这些类提供简单而高效的机制来序列化对共享资源（比如共享内存中的文件或对象）的访问。它们封装了Solaris、POSIX和Win32同步变量（分别是mutex_t、pthread_mutex_t和HANDLE）；在4.5.1.1中描述。
- RW_Mutex、RM_Thread_Mutex、RW_Process_Mutex：这些类序列化对共享资源的访问，其中的内容较少变动，而是更多地用于搜索。它们封装了Solaris rwlock_t同步变量（POSIX pthreads和Win32线程实现使用其他机制）；在4.5.1.3中描述。
- Semaphore、Thread_Semaphore、Process_Semaphore：这些类实现了Dijkstra的“计数信号量”抽象（一种用于序列化多线程控制的通用机制）。它们封装了Solaris sema_t同步变量

（POSIX pthreads和Win32线程实现使用其他机制）；在4.5.1.2中描述。
- Null_Mutex：Null_Mutex类提供一种零开销的锁接口实现，被其他C++封装用于同步。此类在4.5.1.5中描述。
- Token（令牌）：Token类提供一种比Mutex更为通用的同步机制。例如，它实现了“递归互斥体”语义，拥有令牌的线程可以重新获取它，而不会导致死锁。此外，当其他线程释放令牌时，阻塞在该令牌上的线程以严格的FIFO（先进先出）的顺序被服务（相反，Mutex并不严格地强制实行一种获取顺序）。该类在4.5.1.6中描述。
- Recursive_Thread_Mutex：通过允许嵌套调用acquire方法（只要拥有该锁的线程也是重新获取它的线程）， Recursive_Thread_Mutex扩展了缺省的Solaris线程互斥体语义。它与上面概述的Thread_Mutex类一起工作；在4.5.1.4中描述。

4.2.2.2 ACE守卫（Guard）类属

- Guard、Write_Guard和Read_Guard：这些类确保在进入和退出一个C++代码块时分别被自动获取和释放锁。它们在4.5.2.1中描述。
- Thread_Control：Thread_Control类与Thread_Manager类相结合，用于在线程的发起函数中自动进行优雅的终止和清扫活动。该类在4.5.2.2中描述。

4.2.2.3 ACE条件（Condition）类属

- Condition：Condition类用于在涉及共享数据的条件表达式的状态发生变化时进行阻塞。它封装了Solaris和POSIX pthreads cond_t同步变量（Win32线程使用其他机制实现）；在4.5.3.1中描述。
- Null_Condition：Null_Condition类提供零开销的Condition接口实现，用于单线程应用。它在4.5.3.2中描述。

4.2.2.4 ACE管理器（Manager）类属

- Thread_Manager：Thread_Manager类含有一套机制来对相互协作以实现集体行为的成组线程进行管理。该类在4.5.4.1中描述。
- Thread_Spawn：Thread_Spawn提供一种标准工具，对为并发地处理来自客户的请求所进行的线程创建进行管理。该类在4.5.4.2中描述。

4.2.2.5 ACE主动对象（Active Object）类属

- Task（任务）：Task类是ACE中用于定义*主动对象*[16, 17]的中心机制。这些主动对象将输入输出消息排队，并在分离的线程控制中执行用户定义的消息处理服务。该类在4.5.5.1中描述。

4.2.2.6 杂项ACE并发类

- Thread：Thread类封装Solaris线程、POSIX Pthreads和Win32线程族的线程创建、终止和管理例程。该类在4.5.6.1中描述
- Atomic_Op：Atomic_Op类将同步特性透明地参数化进基本的算术操作。该类在4.5.6.2中描述。
- Barrier（栅栏）：Barrier类实现“栅栏同步”，对于许多类型的并行科学应用特别有用。该类在4.5.6.3中描述。
- TSS：TSS类允许“物理上”线程专有的对象被“逻辑地”当作程序的全局对象进行访问。该类在4.5.6.4中描述。

4.3 并发编程和多线程的背景

大多数UNIX系统程序员都熟悉传统的进程管理系统调用（比如fork、exec、wait和exit）。但是，他们关于正在形成中的UNIX多线程和同步机制（比如Solaris线程[2]、POSIX pthreads[3]，或是Win32线程[4]）的经验却较少。这一部分将给出对并发编程和Solaris线程的相关背景材料的综述。对并发编程和Solaris/POSIX/Win32线程的更为详细的讨论见[2, 18, 19, 3, 4]。

4.3.1 进程和线程

*进程*是使程序指令得以执行的一组资源。这些资源包括虚拟内存、I/O描述符、运行时栈、信号处理器、用户和组id，以及访问控制令牌。在早期的UNIX系统上（比如SunOS 4.x），进程是“单线程”的。在UNIX中，单线程程序中的操作通常是同步的，因为控制总是在程序（也就是，用户代码）中，或是在操作系统中（经由系统调用）。在某种程度上，传统UNIX进程的单线程特性简化了编程，因为没有程序员显式地进行干预，进程不会与其他进程相互干扰。

但是，使用单线程进程，有许多应用很难开发（特别是网络服务器）。例如，单线程网络文件服务器不能长期阻塞以处理一个客户请求，因为其他客户的响应性会受到损害。有若干常用方法可用以避免阻塞在单线程服务器中：

- *事件多路分离器/分派器*：方法之一是开发一种事件多路分离器/分派器（比如面向对象的反应堆构架[20]）。该技术被广泛用于在单线程用户接口构架中管理多个输入设备。主事件多路分离器/分派器检测到来的事件，将其分离到适当的事件处理器，然后分派与该事件处理器相关联的应用特有的回调方法。

  该方法的主要缺点是持续时间长的会话必须被开发为有限状态机。当状态的数目增加时该方法将变得相当笨拙。此外，因为只能使用非阻塞的操作，很难通过像“I/O流” 这样的技术、或是数据和指令缓存中的本地引用方案来改善性能。

- *用户级协同例程*（User-level co-routines）：另一种方法是开发一个非占先的（non-preemptive）用户级协同例程包，显式地存储和恢复上下文信息。这使得任务能够挂起它们的执行，直到另一个协同例程在后面将它们唤醒。Windows 3.1和Mac System 7 OS上的多任务机制是广泛可用的使用这一方法的系统。

一般而言，要正确地使用协同例程很复杂，因为开发者必须通过周期性地显式派生线程控制来人工地进行任务占先。而且，每个任务必须只执行相对较短的时间。否则，客户将会觉察请求正在被顺序地、而非并发地处理。协同例程的另一局限是，如果OS在任务引发页错误时阻塞进程中所有的任务，应用的性能可能会下降。此外，单个任务（例如，进入了一个无限循环）的失败可能会挂起整个进程。

- *多进程*：降低单线程UNIX进程的复杂性的另一种方法是使用fork和exec系统调用提供的粗粒度多进程能力。fork派生一个分离的子进程，与父进程并发地执行任务。相互分离的进行可以通过使用像共享内存和内存映射文件这样的机制来进行直接的协作。在本地主机上，共享内存是一种比消息传递更为迅捷的IPC方法，因为它避免了显式的数据拷贝。

  但是，fork和exec的开销和不灵活使得动态的进程请求对于许多应用来说都极为昂贵和复杂。例如，对于持续时间短的服务（比如解析IP地址的以太网号 、从网络文件服务器获取磁盘块，或是在SNMP MIB中设置属性）来说，进程管理开销就太过度了。而且，使用fork和exec很难对调度和进程优先级进行细粒度的控制。此外，在共享内存段中共享C++对象的进程必须对虚表指针的位置做出不可移植的假定。

多线程机制提供了更为优雅，有时也更为高效的方法来克服上述的传统并发进程技术的局限。线程是在进程的上下文中执行的单序列的指令步骤。除了指令指针，线程还包括其他的一些资源，比如函数启用记录的运行时栈、一组通用寄存器，以及线程专有的数据。

传统的工作站操作系统（比如UNIX的一些变种[2, 21, 22]和Windows NT[4]）支持多进程（每一个进程包含1或多个线程）的并发执行，每个进程可包含1或多个线程。进程充当被保护的单元、和在单独的硬件保护地址空间中进行资源分配的单元。线程充当在进程地址空间中运行的执行单元，该线程与0或多个线程共享此地址空间。

4.3.2 基于线程的并发编程的好处

因为如下原因，在相互分离的线程、而不是进程中实现执行多任务的并发应用常常是有益的：

- *线程创建*：不像生成新进程，派生一个新线程不需要（1）复制父地址空间内存，（2）设置新的内核数据结构，以及（3）消耗一个额外的进程槽，以在大的应用中执行子任务。
- *上下文切换*：线程维护最小限度的状态信息，因而降低了上下文切换的开销，因为只须存储和取回少量状态信息。特别地，线程间的上下文切换消耗的时间比UNIX重量级进程间的上下文切换要少。这是由于在同一进程中的线程间切换时，无需改变TLB虚地址映射。而且，严格地在用户级运行的线程不会带来任何上下文切换开销。
- *同步*：当调度和执行应用的线程时，可能不需要在内核模式和用户模式之间进行切换。进程同步比起线程同步要更为昂贵一些。例如，进行同步的实体常常不是全局的、而是局部的实体。全局同步总是涉及到内核，而应用线程所使用的局部（或“进程内”）同步则可能无需内核的干预。
- *数据拷贝*：分离的线程间通过共享内存进行的通信常常比在分离的进程间使用IPC消息传递要快得多，因为前者避免了显式的数据拷贝的开销。例如，相互协作的数据库服务经常引用驻留内存的公用数据结构，通过线程来实现这些服务可能要更为简单和高效。一般而言，在线程间使用进程的共享地址空间进行通信通常比在进程间使用共享内存机制（比如系统V共享内存或内存映射文件）要更为容易和高效。

4.3.3 Solaris上的多进程和多线程综述

这一部分总结Solaris 2.x提供的多进程（MP）和多线程（MT）机制的相关背景材料。其他的线程模型和实现（比如SGI、Sequent、OSF/1和Windows NT）的细节有所不同，但基本的概念都是非常类似的。

相对来说，传统的UNIX进程是一种“重量级”的实体，其中含有一个单线程控制。相反，Solaris上可用的基于线程的并发机制要更为成熟、灵活和高效（在适当使用时）。如图4-2所示，Solaris MP/MT体系结构在两个层面上运作（内核空间和用户空间），并含有以下4种组件：

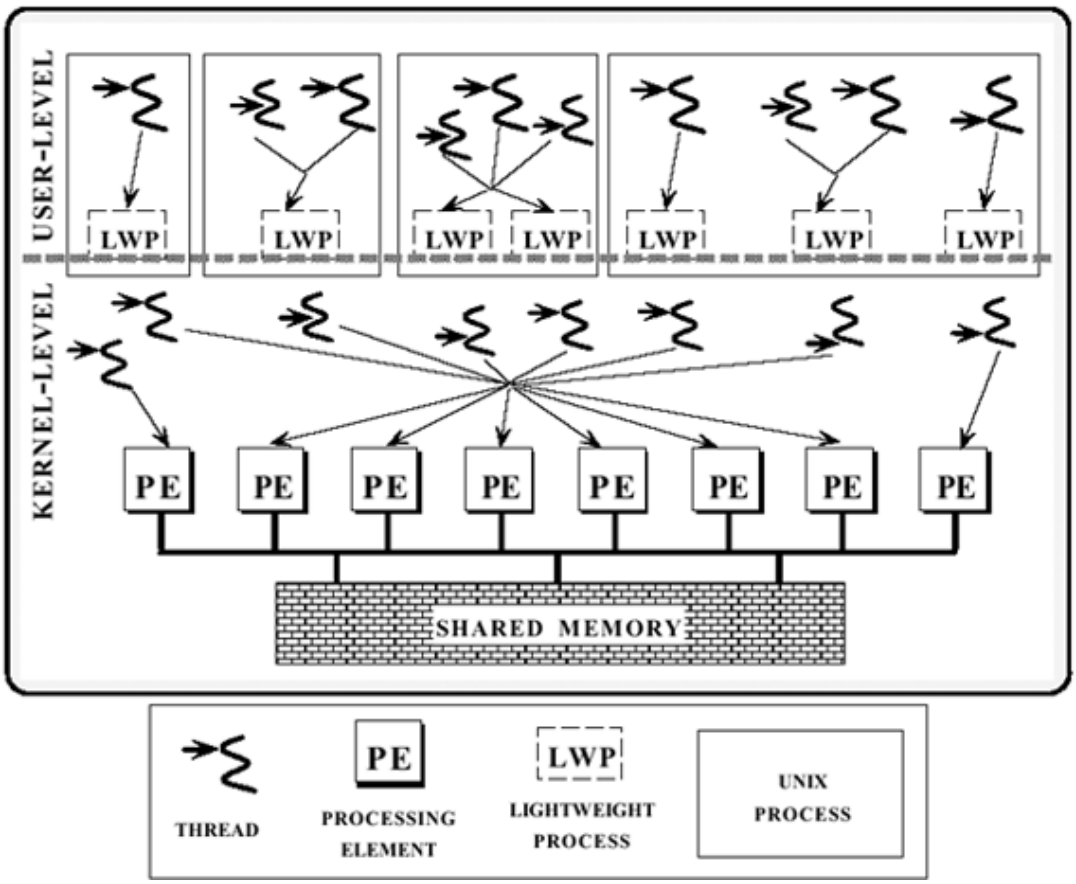

图4-2 Solairs 2.x多进程和多线程体系结构

- *处理单元（Processing element）*：这是执行用户级和内核级指令的CPU。Sun MP/MT模型的语义意图能同时用于单处理器和共享内存的硬件上的对称多处理器。
- *内核线程（Kernel thread）*:这是处理单元(PE)在内核空间中调度和执行的基本实体。OS内核为每一内核线程维护一个小数据结构和栈。内核线程间的上下文切换相对较快，因为它不要求改变虚拟内存信息。
- *轻量级进程（Lightweight process，或LWP）*：它们与内核线程相关联。在Solaris 2.x中，一个UNIX进程不再是一个线程控制，而是在其中含有一或多个LWP。在LWP和它的内核线程间存在着1对1的映射。Solaris中的内核级调度器使用LWP（因而也包括内核进程）来调度应用任务。LWP含有数量相对较多的状态（比如寄存器数据、账务和特征信息、虚拟内存地址范围和定时器）。因而，LWP间的上下文切换相对较慢。

  对于分时调度器类别（缺省），调度器通过“*占先*”（preemption）将可用的PE在多个活动的LWP间进行划分。通过这种技术，每个LWP运行一段有限的时间（通常为10毫秒）。当

前LWP的时间片到期后，OS调度器选择另一个可用的LWP，执行一次上下文切换，并将被占先的LWP放置到一个队列中。内核使用若干标准（比如优先级、资源可用性、调度类别，等等）来调度LWP。在分时调度器类别中，没有固定的LWP执行顺序。

- *应用线程*：每个LWP可被认为是一个“虚拟PE”，在其上应用线程被一个用户级的线程库调度和多路复用。每个应用线程与其他线程一起共享它的进程地址空间，尽管它拥有唯一的栈和寄存器组。应用线程还可以派生出其他应用线程。在进程中，每个这样的应用线程都独立地执行（尽管取决于硬件、并非必然地并发执行）

  Solaris 2.x提供一种多层的并发模型，允许使用下面的两种模式来派生和调度应用线程：

1. 绑定线程（Bound thread）：它被1对1地映射到LWP和内核线程。绑定线程允许独立任务在多PE上并行执行。因而，如果两个应用线程运行在分离的LWP上（因而也是分离的内核线程上），它们也就可以并行地执行（假定它们运行在多处理器上，或使用异步I/O）。而且，应用线程可以执行阻塞的系统调用并处理页错误，而不会阻止其他应用线程的运行。

   重新调度绑定线程需要一次内核级上下文切换。同样地，绑定线程上的同步操作也需要OS内核的干预。当应用被设计利用在硬件平台上可用的并行性优点时，绑定线程最为有用。因为每个绑定线程都要求分配内核资源，分配大量绑定线程可能导致效率低下。

2. *非绑定线程：*它们被一个线程运行时库以多对多的方式在一或多个LWP和内核线程上多路复用。该用户级库实现了一个非占先的协作式多任务并发模型。在使内核干预最少化的同时，它调度、分派，并挂起非绑定线程。与使用绑定到LWP的应用线程相比较，非绑定应用线程的派生、上下文切换和同步所需的开销较小。

   取决于应用和/或库与一个进程相关联的内核线程的数目，可以在多PE上并行执行一或多个非绑定线程。因为每个非绑定线程并不分配内核资源，有可能分配数量相当大的非绑定线程，而不会显著地降低性能。

4.3.4 并发编程的挑战

在多处理器上，可以在分离的多个PE上并行地运行多于一个的LWP。在单处理器上，在任何时刻只能有一个活动的LWP。不管是怎样的硬件平台，程序员必须确保对共享资源（比如文件、数据库记录、网络设备、终端，或共享内存）的访问是依次进行的，以防止“*竞争状态*”（race condition）。竞争状态在两个或多个并发LWP的执行顺序会导致不可预测和错误的结果时发生（比如数据库记录被留在了不一致状态）。竞争状态可使用4.3.5描述的Solaris 2.x同步机制来加以排除。这些机制序列化对共享资源的代码临界区的访问。

除了并发控制的挑战，当使用多线程（而不是多进程，或单线程的反应式事件循环）实现并发应用时还会出现以下的限制：

- *健壮性*：在单进程地址空间中通过线程执行所有任务可能会降低应用的健壮性。该问题之所以发生，是因为在同一进程地址空间中的分离线程彼此间并没有受到保护。为了降低上下文切换和同步开销，硬件内存管理单元（MMU）对线程的保护很少，或者根本没有。

因为线程不受保护，进程中一个有缺陷的服务可能破坏它与在进程的其他线程中运行的服务共享的全局数据结构。于是这就可能导致不正确的结果，毁坏整个进程，致使网络服务器无限期地挂起，等等。一个相关的问题是在某个线程中调用的某些UNIX系统调用可能对整个进程产生不希望产生的副作用。例如，exit系统调用具有销毁进程中所有线程的副作用（应使用thr_exit来终止当前线程）。

- *访问特权*：多线程的另一局限是一个进程中的所有线程共享同样的用户id和访问文件和其他受保护资源的特权。因此，为防止对未授权资源意外或故意的访问，那些将安全机制建立在进程所有权之上的网络服务（比如Internet ftp和telnet服务）通常都在分离的进程中实现。
- *性能*：一种常见的误解是认为使应用多线程化将自动提高性能。其实在许多环境中，多线程并不会提高性能。例如，单处理器[19]上专事计算的应用并不会从多线程中获益，因为计算不会涉及通信。另外，细粒度的锁定会导致高昂的同步开销[23, 24]。这也阻止了应用对并行处理的优点进行全面利用。

在有些环境中多线程可以显著地提高性能。例如，通过在多处理器平台上运行，一个多线程的面向连接的应用网关可以从中获益。同样地，在单处理器上，专事I/O的应用也可以从多线程中获益，因为计算涉及到通信和磁盘操作。

4.3.5 Solaris 2.x同步和线程机制综述

这一部分概述并演示在Solaris 2.x、POSIX pthreads和Win32线程中可用的同步和线程机制。在这些系统中，线程在单进程地址空间中共享若干资源（比如打开的文件、信号处理器，以及全局内存）。因此，它们必须利用*同步机制*来协调对共享数据的访问，以避免发生4.3.4所讨论的竞争状态。为演示对同步机制的需要，考虑下面的C++代码段：

```
typedef u_long COUNTER;

COUNTER request_count; // At file scope


void *run_svc (Queue<Message> *q)

{

Message *mb; // Message buffer


while (q->dequeue (mb)) > 0)

{

       // Keep track of number of requests

       ++request_count;


       // Identify request and

       // perform service processing here...

}
```

```
return 0;

}
```

该代码形成了一个网络看守（比如用于医学成像的分布式数据库，或分布式文件服务器）的主事件循环部分。在代码中，主事件循环等待消息从客户到达。当消息到达时，主线程通过dequeue方法将它从消息队列中移除。然后取决于接收到的消息的类型，线程执行某种处理（例如，图像数据库查询、文件更新，等等）。request_count变量追踪到来的客户请求的数量。该信息可用于更新SNMP MIB中的属性。

只要run_svc在单线程控制中执行，上面所示的代码工作良好。但是，当run_svc由运行在不同的PE上的多线程控制同时执行时，在许多多处理器平台上将会产生不正确的结果。这里的问题是这些代码并非是“线程安全”的，因为针对全局变量request_count的自增操作含有一个竞争状态。因而，不同的线程可能会增加存储在它们自己的PE数据缓存中的request_count变量的陈旧版本。

这一现象可通过执行下面的例4-1中的C++代码来演示，运行环境为一台运行Solaris 2.x操作系统的共享内存多处理器。Solaris 2.x允许多线程控制在共享内存多处理器上并行执行。下面所示的例子是上面演示的网络看守的简化版本：

例4-1

```
typedef u_long COUNTER;

static COUNTER request_count; // At file scope


void *run_svc (int iterations)

{

for (int i = 0; i < iterations; i++)

       ++request_count; // Count # of requests


return (void *) iterations;

}


typedef void *(*THR_FUNC)(void *);


// Main driver function for the

// multi-threaded server.


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

{

int n_threads = argc > 1 ? atoi (argv[1]) : 4;

int n_iterations = argc > 2 ? atoi (argv[2]) : 1000000;
```

```
thread_t t_id;


// Divide iterations evenly among threads.
int iterations = n_iterations / n_threads;


// Spawn off N threads to run in parallel.
for (int i = 0; i < n_threads; i++)
       thr_create (0, 0, THR_FUNC (&run_svc),
                                    (void *) iterations,
                             THR_BOUND | THR_SUSPENDED,
                             &t_id);


// Resume all suspended threads
// (threads id's are contiguous...)
for (i = 0; i < n_threads; i++)
       thr_continue (t_id--);


// Wait for all threads to exit.
int status;


while (thr_join (0, &t_id, (void **) &status) == 0)
       cout << "thread id = " << t_id
              << ", status = " << status << endl;


cout << n_iterations << " = iterations\n"
       << request_count << " = request count"
       << endl;


return 0;
}
```

Solaris thr_create线程库例程被调用n_thread次，以派生n个新线程控制。在此例中，每个新创建的线程都传递iterations的值给run_svc函数，作为它唯一的参数，并执行之。该值使得run_svc例程循环n_iterations/n_threads次。

每个线程都使用THR_BOUND和THR_SUSPENDED标志来派生。THR_BOUND通知Solaris线程运行时库将该线程绑定到专用的LWP。每个LWP可以在一个多处理器系统中的单独的PE上并行运行。THR_SUSPENDED标志创建“挂起”状态的线程，确保在调用thr_continue恢复（resume）测试之前，所有线程被完全地初

始化。thr_continue函数是一个Solaris线程库例程，可恢复挂起线程的执行。注意此例利用了Solaris是以升序连续分配线程id的事实。

一旦所有线程都已被恢复，thr_join例程就阻塞主线程的执行。thr_join与UNIX wait系统调用相类似棗它获取退出线程的状态。thr_join将“收割”线程，并返回0，直到运行run_svc的线程全部退出为止。当所有其他的线程退出后，主线程打印出iterations的总数，以及request_count的最后值，然后退出程序。

将此代码编译成可执行的a.out文件，并以1 个线程循环10,000,000次的方式运行它，得到如下结果：

```
% a.out 1 10000000

thread id = 4, status = 1000000

10000000 = iterations

10000000 = request count
```

该结果正如所愿。但是，当以4个线程循环10,000,000次的方式在4个PE的机器上运行时，程序打印出：

```
% a.out 4 10000000

thread id = 5, status = 1000000

thread id = 7, status = 1000000

thread id = 6, status = 1000000

thread id = 4, status = 1000000

10000000 = iterations

5000000 = request count
```

显然，有什么出错了，因为全局变量request_count的值只是循环总数的一半。这里的问题是变量request_count的自增没有被正确地序列化。

一般而言，在不提供“*强有序缓存一致性模型*”（strong sequential order cache consistency model）的共享内存多处理器平台上并行执行时，run_svc会产生不正确的结果。为增强性能，许多共享内存多处理器采用“*弱有序缓存一致性语义*”（weakly-ordered cache consistency semantics）。例如，SPARC多处理器的V.8和V.9家族同时提供“*总存储序*”（total store order）和“*部分存储序*”（partial store order）内存缓存一致性语义。对于总存储序语义，对正在被其他PE上的线程访问的变量的读取和同时进行的对同一变量的写也许不会被序列化。同样地，对于部分存储序语义，写操作与写操作也可能不会被序列化。在任一情形中，由于多PE间的缓存延迟，需要对内存位置进行不止一次装载和存储的表达式（比如foo++或i = i - 10）可能会产生不一致的结果。为确保线程间共享的变量的读写被正确更新，程序员必须人工地保证对这些变量的变动成为全局可见的。

在“*总存储序*”或“*部分存储序*”共享内存多处理器上强制实现强有序的一种常用技术是使用同步机制来保护request_count变量的增长。Solaris 2.x提供若干种同步机制。本论文描述Solaris 2.x提供的四种主要同步机制的C++包装：*互斥体*、*读者/作者锁*、*计数信号量*，以及*条件变量*[19]。ACE含有封装这四种Solaris 2.x同步机制（分别为mutex_t、rwlock_t、sema_t，以及cond_t）的C++包装。在

4.3的余下部分我们将概述Solaris同步机制的行为。4.4演示C++包装的使用，用以简化常见的同步变量的使用，并改善程序的可靠性。

4.3.5.1 互斥锁

互斥锁（通常称为“互斥体”或“二元信号量”）用于保护多线程控制并发访问的共享资源的完整性。互斥体通过定义临界区来序列化多线程控制的执行，在临界区中每一时刻只有一个线程在执行它的代码。互斥体简单（例如，只有拥有该互斥体的线程可以释放它）而高效（时间和空间）。

像Solaris 2.x这样的操作系统中的互斥体变量的操作通过可适配回旋锁（spin-locks）来实现。回旋锁通过使用一条原子的硬件指令来确保互斥。对于特定类型的短暂资源争用（像上面的例4-1演示的全局request_count变量的自增），它是简单而高效的同步机制。可适配回旋锁使用原子的硬件指令来轮询指定的内存位置，直到下面的情况之一发生[2]：

- 该位置的值被当前拥有该锁的线程改变。这表示锁已被释放，并可以被回旋的线程获取。
- 持有该锁的线程进入睡眠。此时，回旋线程也让自己进入睡眠，以避免不必要的轮询。

在多处理器上，回旋锁带来的开销相对较小。基于硬件的轮询不会导致系统总线上的争用，因为它仅仅影响在互斥体上回旋的线程的本地PE缓存。

互斥体的一种简单而高效的类型是“非递归”互斥体。非递归互斥体不允许当前拥有互斥体的线程在释放它之前重新获取它。否则，将会立即发生死锁。Solaris 2.x和POSIX pthreads通过mutex_t数据类型和相关的mutex_lock和mutex_unlock函数来实现非递归互斥体。

POSIX pthreads和Win32线程都实现了递归和非递归互斥体（其他类型的互斥体在4.4.4讨论）。如4.5.1.4所描述的，ACE OO线程封装库提供互斥体C++封装，以可移植地实现非递归互斥体语义。非递归互斥体在ACE Recursive_Thread_Mutex类中可移植地实现。

4.3.5.2 读者/作者锁

读者/作者锁与互斥体相类似。例如，获取读者/作者锁的线程也必须释放它。多个线程可同时获取一个读者/作者锁用于读，但只有一个线程可以获取该锁用于写。当互斥体保护的资源用于读远比用于写要频繁时，读者/作者互斥体有助于改善并发的执行。

Solaris 2.x通过它的rwlock_t类型来支持读者/作者互斥体。无论是POSIX pthreads，还是Win32线程都没有本地的读者/作者锁支持。如4.5.1.3所描述的，ACE线程库提供了一个叫作RW_Mutex的类，在C++封装类中可移植地实现了读者/作者锁的语义。读者/作者锁的Solaris和ACE实现都将优先选择权给作者。因而，如果有多个读者和一个作者在锁上等待，作者将会首先获取它。

4.3.5.3 计数信号量

在概念上，计数信号量是可以原子地增减的整数。如果线程试图减少一个值为零的信号量的值，它就会阻塞，直到另一个线程增加该信号量的值。

计数信号量用于追踪共享程序状态的变化。它们记录某种特定事件的发生。因为信号量维护状态，它们允许线程根据该状态来作决定，即使事件是发生在过去。

信号量比互斥体效率要低，因为它们保持额外的状态，并使用休眠锁，而不是回旋锁。但是，它们要更为通用，因为它们无需被最初获取它们的同一线程获取和释放。这使得它们能够用于异步的执行上下文中（比如信号处理器）。

Solaris 2.x通过它的sema_t类型支持信号量。Win32将信号量作为HANDLE来支持。POSIX pthreads没有本地的信号量支持。如4.5.1.2所描述的，ACE线程库提供一个叫作Semaphore的类来可移植地在C++包装类中实现信号量语义。

4.3.5.4 条件变量

条件变量提供风格与互斥体、读者/作者锁和计数信号量不同的锁定机制。当持有锁的线程在临界区执行代码时，这三种机制让协作线程进行等待。相反，条件变量通常被一个线程用于使自己等待，直到一个涉及共享数据的条件表达式到达特定的状态。当另外的协作线程指示共享数据的状态已发生变化，调度器就唤醒一个在该条件变量上挂起的线程。于是新唤醒的线程重新对它的条件表达式进行求值，如果共享数据已到达合适状态，就恢复处理。

被条件变量等待的条件表达式可以任意地复杂。一般而言，与其他同步机制相比，条件变量允许更为复杂的调度决策。条件变量同步使用休眠锁来实现，休眠锁触发上下文切换，并允许另一线程执行，直到锁被获取。如4.3.5.1所描述的，互斥体使用可适配回旋锁来实现。如果线程必须长时间地等待某特定条件实现，回旋锁就会消耗过多的资源。

对于涉及条件表达式语义的情况，条件变量比信号量或互斥体要更为有用。在这种情况下，等待线程必须阻塞到特定的涉及共享状态的条件表达式变为真（例如，表不再为空和网络流控制减轻）。在这种情况下，不需要维护事件历史。因而，条件表达式不记录它们什么时候被“置位”。如果没有正确地使用，这可能会导致“丢失的苏醒”（lost wakeup）问题[19]。

Solaris 2.x和POSIX pthreads通过cond_t类型支持条件变量。本地的Win32 API不支持条件变量。如4.5.3.1所描述的，ACE线程库提供一个叫作Condition的类来可移植地在C++包装类中实现条件变量语义。

4.3.6 进程 vs. 线程同步语义

为增加灵活性和改善性能，Solaris 2.x提供两种风格的同步语义，并为下列两种情况而优化：（1）在同一进程中执行的线程（也就是，进程内序列化）以及（2）在相互分离的进程中执行的线程（也就是，进程间序列化）。在Solaris 2.x中，同步机制的*_init函数的USYNC_THREAD标志创建为单个进程中的线程而优化的变量。同样地，USYNC_PROCESS标志创建跨进程有效的同步变量。后一种类型的同步机制更为通用，尽管略为低效，如果所有线程是在单一进程中运行的话。

4.3.7 互斥体例子

下面的代码演示怎样将Solaris互斥体变量用于解决我们先前考察的request_count的自增序列化问题：

例4-2

```
typedef u_long COUNTER;

// At file scope

static COUNTER request_count;

// mutex protecting request_count (initialized to zero).

static mutex_t m;


void *run_svc (void *)

{

for (int i = 0; i < iterations; i++)

{

       mutex_lock (&m); // Acquire lock

       ++request_count; // Count # of requests

       mutex_unlock (&m); // Release lock

       }


return (void *) iterations;

}
```

在上面的代码中，m是mutex_t类型的全局变量。在Solaris中，凡是被置零的同步变量都使用它的缺省语义来初始化。例如，mutex_t变量m总是被初始化成从*未锁定*状态开始。因此，当mutex_lock第一次被调用时，它将获得该锁的所有权。任何其他想要获取该锁的线程都必须等待（例如，通过“回旋”），直到锁的属主释放m。

上面所示的例4-2解决了原来的同步问题，但它仍具有以下缺陷：

- *不优雅和不一致*：代码混合了C函数和C++代码，以及不同的标识符命名习惯。使用混合的编程风格让人困惑，还可能带来维护问题。
- *强制性*：该方案要求改变源代码。在开发大型软件系统时，人工地进行这样的变动会导致维护问题，如果所有变动没有一致地进行的话。
- *不可移植*：该代码只能与Solaris 2.x同步机制一起工作。特别地，移植该代码以使用POSIX pthreads和Windows NT线程将需要改变锁定代码。
- *易错*：程序员很容易忘记调用mutex_unlock。这会使其他正试图获取该锁的线程饿死。而且，如果锁的属主试图重新获取它已经拥有的互斥体，就会发生死锁。

程序员还可能忘记初始化互斥体变量。如上面所提到的，在Solaris 2.x中，被置零的mutex_t变量被隐含地初始化。但是，并不能保证在动态分配的结构或类中作为域被分配的mutex_t变量也被隐含地初始化。而且，其他线程机制（比如POSIX pthreads和Windows NT线程）并没有这样的初始化策略，所有同步对象都必须显式地初始化。

在4.4我们将检查通过改善Solaris同步机制的功能、可移植性和健壮性，C++包装的使用是怎样帮助克服这些问题的。

4.4 通过OO和C++简化并发编程

这一部分检查一个使用实例，以演示在C++包装中封装Solaris并发机制的优点。该使用实例描述了一个基于生产系统的有代表性的情景[25]。紧跟4.5对库接口的介绍，在4.6中还有ACE OO线程封装类库的其他例子。

通过归纳系统开发中发生的实际问题的解决方案，许多有用的C++类已逐渐发展起来。但是在类的接口和实现稳定后，随着时间的过去，这样的反复对类进行归纳的过程已不再被强调。这让人遗憾，因为要进入面向对象设计和C++的主要障碍是（1）学习并使*怎样*识别和描述类和对象的过程内在化，以及（2）理解何时以及怎样应用（或不应用）像模板、继承、动态绑定和重载这样的C++特性来简化和归纳他们的程序。

为努力把握C++类设计演变的动力特性，下面的部分演示逐步应用面向对象技术和C++习语、以解决一个惊人地微妙的问题的过程。该问题发生在开发并发分布式应用族的过程中，该应用族在单处理器和多处理器平台上都能高效地执行。通过使用模板和重载来透明地将同步机制参数化进并发应用，这一部分集中考察那些涉及对已有代码进行归纳的步骤。其基础代码基于在[1, 26, 20]中描述的自适配通信环境（ACE）构架中的组件。

此例子检查若干C++的语言特性，它们可以更为优雅地解决4.3.5.1提出的序列化问题。如在那里所描述的，原来的方案既不优雅、不可移植、易错，并且还要求强制性地改变源代码。这一部分演示一种进化的、在先前反复的设计演变中所产生的洞见之上构建的C++方案。

4.4.1 初始C++方案

解决原来问题的一种更为优雅一点的方案是通过C++ Thread_Mutex包装来封装现有的Solaris mutex_t操作，如下所示：

```
class Thread_Mutex

{

public:

Thread_Mutex (void)

{
```

```
        mutex_init (&lock_, USYNC_THREAD, 0);
}

?Thread_Mutex (void)
{
        mutex_destroy (&lock_);
}

int acquire (void)
{
        return mutex_lock (&lock_);
}

int release (void)
{
        return mutex_unlock (&lock_);
}

private:
// Solaris 2.x serialization mechanism.
mutex_t lock_;
};
```

给互斥机制定义C++包装接口的一个优点是应用代码现在变得更为可移植了。例如，下面是Thread_Mutex类接口的实现，它基于Windows NT WIN32 API[4]中的机制之上：

```
class Thread_Mutex
{
public:
Thread_Mutex (void)
{
        InitializeCriticalSection (&lock_);
}

?Thread_Mutex (void)
{
```

```
        DeleteCriticalSection (&lock_);
}

int acquire (void)
{
        EnterCriticalSection (&lock_); return 0;
}

int release (void)
{
        LeaveCriticalSection (&lock_); return 0;
}

private:
// Win32 serialization mechanism.
CRITICAL_SECTION lock_;
};
```

使用Thread_Mutex C++包装类使原来的代码变得更为清晰，改善了可移植性，并且确保了Thread_Mutex对象被定义时，初始化工作自动地进行。如下面的代码段所示：

例4-3

```
typedef u_long COUNTER;
// At file scope.
static COUNTER request_count;
// Thread_Mutex protecting request_count.
static Thread_Mutex m;

void *run_svc (void *)
{
for (int i = 0; i < iterations; i++)
{
        m.acquire ();
```

```
        // Count # of requests.
        ++request_count;

        m.release ();
}

return (void *) iterations;
}
```

但是，C++封装方法并没有解决所有在4.3.5.1所标出的问题。特别地，它没有解决忘记释放互斥体的问题（它还是需要程序员的人工干预）。此外，使用Thread_Mutex也还是需要对原来的非线程安全的代码进行强制性的改变。

4.4.2 另一种C++方案

确保锁被自动释放的一种直截了当的方法是使用C++类构造器和析构器语义。下面的工具类使用这些语言构造来自动进行互斥体的获取和释放：

```
class Guard
{
public:
Guard (const Thread_Mutex &m): lock_ (m)
{
        lock_.acquire ();
}

?Guard (void)
{
        lock_.release ();
}

private:
const Thread_Mutex &lock_;
}
```

Guard定义了一“块”代码，在其上一个Thread_Mutex被自动获取，并在退出代码块时自动释放。它采用了一种通常称为“作为资源获取的构造器楦魑□试词头诺奈龋蛊鳖?/FONT>[9, 27, 7]的C++习语。

如上面的代码所示，当Guard类的对象被创建时，它的构造器自动获取Thread_Mutex对象上的锁。同样地，Guard类的析构器在对象出作用域时自动解锁Thread_Mutex对象。

注意Guard类的数据成员lock_是Thread_Mutex对象的一个引用。这避免了在Guard对象的构造器和析构器每次执行时创建和销毁底层Solaris mutex_t变量的开销。

通过对代码作出轻微的变动，我们现在保证了Thread_Mutex被自动获取和释放：

例4-4

```
typedef u_long COUNTER;

// At file scope.

static COUNTER request_count;

// Thread_Mutex protecting request_count.

static Thread_Mutex m;


void *run_svc (void *)

{

for (int i = 0; i < iterations; i++)

{

       {

              // Automatically acquire the mutex.

              Guard monitor (m);


              ++request_count;

              // Automatically release the mutex.

       }


       // Remainder of service processing omitted.

}

}
```

但是，该方案还是没有解决强制性改变代码的问题。而且，在Guard周围增加额外的‘{’和‘}’花括号分隔符块既不优雅又容易出错。进行维护的程序员可能会误认为花括号并不重要并将它们去除，产生出下面的代码：

```
for (int i = 0; i < iterations; i++)

{

Guard monitor (m);



++request_count;

// Remainder of service processing omitted.

}
```

遗憾的是，这样的“花括号省略”有副作用：它通过序列化主事件循环、消除了应用中所有并发执行。因此，所有应该在那段代码区中并行执行的计算都会被不必要地序列化。

4.4.3 改良的C++方案

要以一种透明的、非强制的和高效的方式解决现存的问题，需要使用两种另外的C++特性：参数化类型和操作符重载。我们使用这些特性来提供一个称为Atomic_Op的模板类，其部分代码显示如下（完整的接口见4.5.6.2）：

```
template <class TYPE>

class Atomic_Op

{

public:

Atomic_Op (void) { count_ = 0; }

Atomic_Op (TYPE c) { count_ = c; }

TYPE operator++ (void)

{

       Guard monitor (lock_);

       return ++count_;

}

operator TYPE ()

{

       Guard monitor_ (lock_);

       return count_;

}

// Other arithmetic operations omitted...
```

```
private:

Thread_Mutex lock_;

TYPE count_;

};
```

Atomic_Op类重新定义了普通的针对内建数据类型的算术操作符（比如++、--、+=，等等），以使这些操作符原子地工作。一般而言，由于C++模板的“延期实例化”语义，任何定义了基本算术操作符的类都将与Atomic_Op类一起工作。

因为Atomic_Op类使用了Thread_Mutex的互斥特性，针对Atomic_Op的实例化对象的算术运算现在在多处理器上工作*正常*。而且，C++特性（比如模板和操作符重载）还允许这样的技术在多处理器上*透明地*工作。此外，Atomic_Op中的所有方法操作都被定义为内联函数。因此，一个C++优化编译器将生成代码确保Atomic_Op的运行时性能不会低于直接调用mutex_lock和mutex_unlock函数。

使用Atomic_Op类，我们现在可以编写下面的代码，几乎等同于原来的非线程安全代码（实际上，只是改变了COUNTER的类型定义）：

例4-5

```
typedef Atomic_Op<u_long> COUNTER;

// At file scope

static COUNTER request_count;


void *run_svc (void *)

{

for (int i = 0; i < iterations; i++)

{

       // Actually calls Atomic_Op::operator++()

       ++request_count;

}

}
```

通过结合C++构造器/析构器习语（以自动获取和释放Thread_Mutex）和模板及重载的使用，我们生成了一种既简单又非常有表现力的参数化类抽象。该抽象可在无数需要原子操作的类型族上正确而原子地运作。例如，要为其他算术类型提供同样的线程安全功能，我们只需简单地实例化Atomic_Op模板类的新对象：

```
Atomic_Op<double> atomic_double;
```

```
Atomic_Op<Complex> atomic_complex;
```

4.4.4 通过参数化互斥机制类型扩展Atomic_Op

尽管上面描述的Atomic_op和Guard类的设计产生了正确和透明的线程安全程序，还是存在着足资改进的空间。特别地，注意Thread_Mutex数据成员的类型是被硬编码进Atomic_Op类的。既然在C++中可以使用模板，这样的设计决策就是一种不必要的、可以轻易克服的限制。解决方案就是参数化Guard，并增加另一种类型参数给模板类Atomic_Op，如下所示：

```
template <class LOCK>
class Guard
{
// Basically the same as before...
private:
// new data member change.
const LOCK &lock_;
};

template <class LOCK, class TYPE>
class Atomic_Op
{
TYPE operator++ (void)
{
       Guard<LOCK> monitor (lock_);
       return ++count_;
}
// ...

private:
LOCK lock_; // new data member
TYPE count_;
};
```

使用这个新类，我们可以在源代码的开始处作出下面的简单变动：

```
typedef Atomic_Op <Thread_Mutex, u_long> COUNTER;

// At file scope.

COUNTER request_count;


// ... same as before
```

4.4.5 设计原理和性能问题

在作出上面描述的变动之前，很值得分析一下使用模板来参数化程序所用的互斥机制类型的动机是否是有益的。毕竟，仅仅因为C++支持模板，并不意味着它们在所有的环境中就都有用。事实上，通过模板来参数化和归纳问题空间，而又缺少清晰而足够的理由，可能会增加理解和复用C++类的难度。

Atomic_Op类中模板的使用引发了若干问题。第一个是“所有增加的抽象的运行时性能代价是什么？”第二个问题是“除了模板，为什么不使用继承和动态绑定来强调统一的互斥体接口并共享通用的代码？”第三是“程序的同步属性不是因为使用模板和重载而变得不明晰了吗？”若干这样的问题是相关的，本节将讨论涉及不同设计选择的折衷。

4.4.5.1 性能

为Atomic_Op类选用模板的主要原因涉及运行时效率。在模板实例化的过程中，一旦优化C++编译器将其展开，额外的运行时开销很少，或根本不存在。相反，继承和动态绑定会带来用以分派虚方法调用的运行时开销。

图4-3演示例4-2到例4-5中使用的互斥技术所展现的性能。该图描述处理1千万次迭代所需的秒数，迭代次数被划分为2.5百万次迭代/线程。测试例子使用Sun C++ 3.0.1编译器的-O4优化级编译。每个测试在另外一台闲置的4 PE Sun SPARCserver 690MP上执行10次。结果被平均，以减少虚假的偏差量（此数量被证明是微不足道的）。

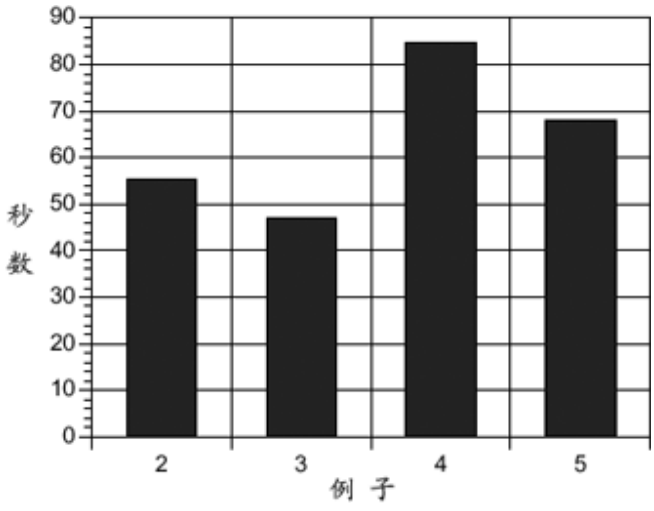

图4-3 处理10,000,000次迭代所需秒数

例4-2直接使用Solaris mutex_t函数。例4-3使用C++ Thread_Mutex类包装接口。令人惊讶的是，该实现始终都优于例4-1，而例4-1是直接调用底层Solaris互斥体函数。例4-4在一对花括号块中使用Guard助手类，以确保Thread_Mutex被自动释放。该版本所需的执行时间最多。最后，例4-5使用Atomic_Op模板类，比直接使用Solaris互斥体函数的只稍稍低效一点。更为强势的优化C++编译器可能还会减少这些结果的偏差量。

例子	微秒/操作	比率
例4-2	2.76	1
例4-3	2.35	0.85
例4-4	4.24	1.54
例4-5	3.39	1.29

表4-1 不同例子的序列化时间

表4-1显示从例4-2到例4-5每次互斥操作所花费的微秒数。注意每次循环迭代需要2次互斥体操作（一次获取锁，一次释放锁）。例4-2被用作基准值，因为它直接使用底层Solaris原语。例4-3到例4-5的第三列通过将它们的值除以例4-2的值来进行规格化。

4.4.5.2 可移植性

参数化互斥机制类型的一个动机是增强跨平台可移植性。模板使形式的参数类名“Thread_Mutex”与实际用于提供互斥的类的名字去耦合。这对于已经使用符号Thread_Mutex来表示现有类型或功能的平台来说是有用的。在移植到这样的平台时，通过使用模板，不需要对Atomic_Op类源代码进行任何改变。

但是，更为有趣的动机来自于对下述事实的观察：一个人可能实际上想使用若干不同风格的互斥体语义（在同一程序中，或跨越一个相关程序族）。每个这样的互斥风格共享同一基本的获取和释放协议，但它们持有拥有不同的序列化和性能属性。4.5.1.1介绍许多已在实践中被证明为有用的互斥机制。

4.4.5.3 透明性

一种*反对*使用模板来参数化同步的意见认为透明度隐藏了程序的互斥语义。这被认为是“臭虫”还是“特性”取决于开发者是否认为并发和同步应被集成进程序中。对于包含基本的“积木”组件（比如4.5.1.1描述的Map_Manager）的类库，常常希望允许同步语义被参数化，因为这使得开发者能够*精确地*控制和指定他们想要的并发语义。这一策略的替换方案有：（1）如果使用多线程，就不使用类库（这显然限制了功能），（2）在库的外面完成锁定（这可能是低效或不安全的），或者（3）将锁定策略硬编码进库的实现（这也是不灵活和潜在地低效的）。所有这些方案都是违背面向对象软件系统中的复用原则的。

4.4.5.4 折衷评估

选择适当的设计策略来开发支持并发的类库依赖于若干因素。例如，某些库用户可能会欢迎简单的、看上去隐藏了并发控制机制的接口。相反，其他库用户可能愿意接受更复杂的接口，以获取额外的控制和更高的效率。一种分层的类库设计方法也许可以同时满足这两种用户。使用这样的设计方法，类库的最底层输出大多数或全部的参数化类型作为模板参数。较高层则提供合理的缺省类型值，并提供易于使用的应用开发者编程接口。

最近被ANSI C++委员会采纳的新的“缺省模板参数”特性便利了同时满足两种类型库用户的类库的开发。该特性允许库开发者指定常用的缺省类型作为模板类和函数定义的参数。例如，下面对模板类Atomic_Op的修改为之提供了常用的缺省模板参数：

```
template <class LOCK = Thread_Mutex,

       class TYPE = u_long>

class Atomic_Op

{

// Same as before

};



// ...



#if defined (MT_SAFE)

// Default is Thread_Mutex and u_long.

typedef Atomic_Op<> COUNTER;

#else

// Don't serialize.

typedef Atomic_Op<Null_Mutex> COUNTER;

#endif /* MT_SAFE */
```

```
COUNTER request_count;
```

由于将并发结合进应用所带来的复杂性，C++模板对于减少多余的开发工作相当有用。但是，就像任何其他的语言特性一样，也有可能误用模板，不必要地使系统的设计和实现变得复杂化（不用说还会增加编译和链接时间）。在决定是否使用参数化类型时，可以使用一种启发式方法：留意何时现有代码将要被复制，而所做修改仅涉及数据类型。如果有另一种合理的情况，需要只是类型不同的第三种版本，这就说明可能值得对原始代码进行归纳来使用模板。

4.5 公共接口和内部设计

这一部分描述ACE OO线程封装库中组件的公共接口和相关的内部设计情况。ACE组件被划分为下面的组：

- *低级C++线程API*：为底层的OS线程和同步API提供低级C++封装。低级C++线程API包含在被称为OS的类中。该类封装了各种版本的UNIX以及WIN32之间的所有差异。ACE中的其他组件被编写为只使用该类中的方法，从而使得将ACE移植到新的平台上变得更为容易。
- *高级C++线程库*：允许多线程应用被编写为使用更高级的C++特性，像构造器/析构器和模板。高级C++线程API根据低级的ACE OS类来编写。它们被划分为三个组：

  - 锁定机制（在4.5.1描述）的C++包装。
  - 本地线程函数（在4.5.6.1描述）的C++包装。
  - 更高级的线程管理类（在4.5.4.1描述）。

ACE OO线程组件的一个使用实例已在4.4中介绍。这一部分的余下部分对ACE的公共接口、功能和内部设计进行更为全面的讨论。在适当的地方，这一部分描述该C++包装类的私有部分，以演示包装是如何映射到Solaris 2.x线程和同步机制上的。POSIX pthreads和Win32包装的实现是类似的。

4.5.1 ACE锁类属

ACE C++包装给4.3.5描述的Solaris、POSIX和Win32 OS同步机制提供了一种可移植、线程安全和面向对象的接口。下面的条目概述了这些ACE C++封装的主要好处：

- **提高正确性**：通过使作为C++类和结构中的域出现的同步对象的初始化自动化，以及通过保证锁被自动获取和释放来实现。
- **统一的同步接口**：所有线程和同步的C++包装都为获取和释放多种类型的锁提供了统一的接口。特别地，ACE锁类属中的所有组件都支持四种通用方法：acquire、try_acquire、release和remove。这种统一性使得开发者有可能使用锁类作为类型参数，来与其他的ACE同步组件（比如在4.5.2.1、4.5.6.2和4.5.1.4定义的那些组件）联合使用。

- **更为直观的错误报告**：Solaris 2.x和POSIX pthreads同步函数使用一种不那么标准的机制来将错误返回给调用者。相反，ACE包装使用一种更为标准的方法：如果发生失败就返回-1，并设置errno来指示失败的原因。
- **简化常见使用模式**：包装简化了低级线程和同步机制的常见使用模式。所示代码通过使用mutex_t和cond_t的ACE C++包装实现一个简单版本的Dijkstra计数信号量（也就是，P和V分别等价于acquire和release）演示了这一点。

```
class Semaphore

{

public:

Semaphore (int initial_value)

       : count_nonzero_ (lock_)

{

       // Automatically acquire lock.

       Guard<Thread_Mutex> monitor (lock_);



       count_ = initial_value;

       // Automatically release the lock

}



// Block the thread until the semaphore

// count becomes greater than 0,

// then decrement it.

void acquire (void)

{

       // Automatically acquire lock

       Guard<Thread_Mutex> monitor (lock_);



       // Wait until semaphore is available.

       while (count_ == 0)

              count_nonzero_.wait ();



       count_ = count_ - 1;

       // Automatically release the lock

}



// Increment the semaphore, potentially
```

```
// unblocking a waiting thread.

void release (void)

{

       // Automatically acquire lock

       Guard<Thread_Mutex> monitor (lock_);



       // Allow waiter to continue.

       if (count_ == 0)

              count_nonzero_.signal ();



       count_ = count_ + 1;

       // Automatically release the lock

}



private:

Thread_Mutex lock_;

Condition<Thread_Mutex> count_nonzero_;

u_int count_;

};
```

注意Condition对象count_nonzero_的构造器是怎样将Thread_Mutex对象lock_和Condition对象绑定在一起的。这简化了Condition::wait调用接口。相反，本地的Solaris和pthreads cond_t cond_wait接口要求每次调用wait时都传递一个互斥体作为参数。

Solaris 2.x和Win32提供一种内建的计数信号量实现（见4.3.5.3的讨论）。但是，POSIX pthreads[3]线程库*没有*包含信号量。因此，上面所示的类既演示了ACE C++包装的使用，又为ACE线程封装库中POSIX pthreads的可移植Semaphore实现提供了文档。

4.5.1.1 互斥体类

ACE互斥体包装提供了一种简单而高效的机制来序列化对共享资源的访问。它们封装Solaris和POSIX pthreads mutex_t同步变量，以及Win32的基于HANDLE的互斥体实现。Mutex的类定义如下所示：

```
class Mutex

{

public:
```

```
// Initialize the mutex.

Mutex (int type = USYNC_THREAD);


// Implicitly destroy the mutex.

?Mutex (void);


// Explicitly destroy the mutex.

int remove (void);


// Acquire lock ownership (wait

// for lock to be released).

int acquire (void) const;


// Conditionally acquire lock

// (i.e., don't wait for lock

// to be released).

int try_acquire (void) const;


// Release lock and unblock

// the next waiting thread.

int release (void) const;


private:

mutex_t lock_;

// Type of synchronization lock.

};
```

在ACE中，线程可以通过调用Mutex对象的acquire方法来进入临界区。任何对该方法的调用都将会阻塞，直到当前拥有该锁的线程离开它的临界区。要离开临界区，线程调用它当前拥有的Mutex对象的release方法。调用release使得另一个阻塞在该互斥体上的线程能够进入它的临界区。

Thread_Mutex和Process_Mutex类继承自Mutex，并使用它的构造器来创建适当类型的互斥体，如下：

```
class Thread_Mutex : public Mutex

{
```

```
public:

Thread_Mutex (void): Mutex (USYNC_THREAD);

};


class Process_Mutex : public Mutex

{

public:

Thread_Mutex (void): Mutex (USYNC_PROCESS);

};
```

这些调用被映射到适当的底层API上，以分别创建线程和进程专用的互斥体。特别地，Thread_Mutex的Win32实现使用更为高效、但却不那么强大的CRITICAL_SECTION实现，而Process_Mutex实现则使用较为低效、但却更为强大的Win32互斥体HANDLE。

4.5.1.2 信号量类

ACE信号量包装类实现Dijkstra的“计数信号量”抽象，这是一种用于序列化多个线程控制的通用机制。它们封装Solaris sema_t同步变量。Semaphore类接口如下所示：

```
class Semaphore

{

public:

// Initialize the semaphore,

// with default value of "count".

Semaphore (u_int count,

                           int type = USYNC_THREAD,

              void * = 0);


// Implicitly destroy the semaphore.

?Semaphore (void);


// Explicitly destroy the semaphore.

int remove (void);


// Block the thread until the semaphore count
```

```
// becomes greater than 0, then decrement it.

int acquire (void) const;


// Conditionally decrement the semaphore if

// count greater than 0 (i.e., won't block).

int try_acquire (void) const;


// Increment the semaphore, potentially

// unblocking a waiting thread.

int release (void) const;


private:

sema_t semaphore_;

};
```

Thread_Semaphore和Process_Semaphore类继承自Semaphore，并使用它的构造器来创建适当类型的信号量，如下所示：

```
class Thread_Semaphore : public Semaphore

{

public:

Thread_Semaphore (void): Semaphore (USYNC_THREAD);

};


class Process_Semaphore : public Semaphore

{

public:

Thread_Semaphore (void): Semaphore (USYNC_PROCESS);

};
```

4.5.1.3 RW_Mutex类

ACE读者/作者包装序列化对这样一种资源的访问：其内容被搜索要多于被变动。它们封装rwlock_t同步变量，这种变量在Solaris上在本地实现，而在Win32和Pthreads上则由ACE模拟。RW_Mutex接口如下所示：

```
class RW_Mutex

{

public:

// Initialize a readers/writer lock.

RW_Mutex (int type = USYNC_THREAD,

              void *arg = 0);


// Implicitly destroy a readers/writer lock.

?RW_Mutex (void);


// Explicitly destroy a readers/writer lock.

int remove (void);


// Acquire a read lock, but

// block if a writer hold the lock.

int acquire_read (void) const;


// Acquire a write lock, but

// block if any readers or a

// writer hold the lock.

int acquire_write (void) const;


// Conditionally acquire a read lock

// (i.e., won't block).

int try_acquire_read (void) const;


// Conditionally acquire a write lock

// (i.e., won't block).

int try_acquire_write (void) const;


// Unlock a readers/writer lock.

int release (void) const;


private:

rwlock_t lock_;
```

```
};
```

注意POSIX Pthreads和Win32线程并不提供rwlock_t类型。为确保代码的可移植性，ACE提供了一种基于现有低级同步机制（比如互斥体和条件变量）的RW_Mutex实现。另外，ACE还提供RW_Thread_Mutex和RW_Process_Mutex实现。

4.5.1.4 Recursive_Thread_Mutex类

Recursive_Thread_Mutex扩展缺省的Solaris非递归锁定语义。它允许嵌套调用acquire方法，只要拥有该锁的线程也是重获取它的线程。它与Thread_Mutex类一起工作。

缺省地，Solaris提供非递归互斥体。这些语义在某些环境中太过受限。因此，ACE通过Recursive_Thread_Mutex类为递归锁提供支持。递归锁对于回调驱动的C++构架[28, 20]特别有用，在其中构架的事件循环执行对用户定义的代码的回调。因为用户定义的代码有可能随后经由一个方法入口重入构架代码，递归锁对于防止在回调过程中，在构架所持有的锁上发生死锁十分有用。

下面的C++类为Solaris 2.x同步机制实现递归锁语义（注意POSIX Pthreads和Win32在它们的本地线程库中提供递归锁）：

```
class Recursive_Thread_Mutex

{

public:

// Initialize a recursive mutex.

Recursive_Thread_Mutex (const char *name = 0

                                        void *arg = 0);


// Implicitly release a recursive mutex.

?Recursive_Thread_Mutex (void);


// Explicitly release a recursive mutex.

int remove (void);


// Acquire a recursive mutex (will increment

// the nesting level and not deadmutex if

// owner of the mutex calls this method more

// than once).

int acquire (void) const;
```

```
// Conditionally acquire a recursive mutex
// (i.e., won't block).
int try_acquire (void) const;

// Releases a recursive mutex (will not
// release mutex until nesting level == 0).
int release (void) const;

thread_t get_thread_id (void);
// Return the id of the thread that currently
// owns the mutex.

int get_nesting_level (void);
// Return the nesting level of the recursion.
// When a thread has acquired the mutex for the
// first time, the nesting level == 1. The nesting
// level is incremented every time the thread
// acquires the mutex recursively.

private:
void set_nesting_level (int d);
void set_thread_id (thread_t t);

Thread_Mutex nesting_mutex_;
// Guards the state of the nesting level
// and thread id.

Condition<Thread_Mutex> lock_available_;
// This is the condition variable that actually
// suspends other waiting threads until the
// mutex is available.

int nesting_level_;
// Current nesting level of the recursion.

thread_t owner_id_;
```

```
// Current owner of the lock.
};
```

下面的代码演示Recursive_Thread_Mutex类中的方法的实现：

```
Recursive_Thread_Mutex::Recursive_Thread_Mutex
(const char *name, void *arg)
: nesting_level_ (0),
owner_id_ (0),
nesting_mutex (name, arg),
lock_available_ (nesting_mutex_, name, arg)
{
}

// Acquire a recursive lock (will increment
// the nesting level and not deadlock if
// owner of lock calls method more than once).
int Recursive_Thread_Mutex::acquire (void) const
{
thread_t t_id = Thread::self ();

Thread_Mutex_Guard mon (nesting_mutex_);
// If there’s no contention, just
// grab the lock immediately.
if (nesting_level_ == 0)
{
       set_thread_id (t_id);
       nesting_level_ = 1;
}
// If we already own the lock,
// then increment the nesting level
// and proceed.
else if (t_id == owner_id_)
       nesting_level_++;
else
```

```
{
	// Wait until the nesting level has dropped to
	// zero, at which point we can acquire the lock.
	while (nesting_level_ > 0)
		lock_available_.wait ();

	set_thread_id (t_id);
	nesting_level_ = 1;
}

return 0;
}

// Releases a recursive lock.
int Recursive_Thread_Mutex::release (void) const
{
thread_t t_id = Thread::self ();

// Automatically acquire mutex.
Thread_Mutex_Guard mon (nesting_mutex_);

nesting_level_--;
if (nesting_level_ == 0)
	// Inform waiters that the lock is free.
	lock_available_.signal ();

return 0;
}
```

下面是基于4.4介绍的Atomic_Op COUNTER的变种的一个Recursive_Thread_Mutex的例子。在例中，Atomic_Op在单线程中被多次递归的函数调用：

```
// Counter is a recursive lock.
typedef Atomic_Op<Recursive_Thread_Mutex>
COUNTER;
// Keep track of the recursion depth.
```

```
static COUNTER recursion_depth;


int factorial (int n)

{

if (n <= 1)

{

       cout << "recursion depth = "

              << recursion_depth << endl;

       return n;

}

else

{

       // First call acquires lock, subsequent

       // calls increment nesting level.

       recursion_depth++;

       return factorial (n - 1) * n;

}

}
```

当recursion_depth计数器增长时，Recursive_Thread_Mutex的使用防止了死锁的发生。尽管这演示了递归锁的行为，它并非是一个非常令人信服的例子。在多线程中执行factorial的程序可能会产生不可预知的结果，因为recursion_depth是一个全局变量，可能会被多个线程控制连续地修改！在这种情况下，一种更为适当的（并且更低廉的）锁定策略将使用4.5.6.4描述的线程专有存储模式[29]。

4.5.1.5 Null_Mutex类

Null_Mutex类提供一种零开销的通用锁定接口的实现，该接口与其他用于线程和同步的C++包装共享。Null_Mutex的接口和极其简单的实现如下所示：

```
class Null_Mutex

{

public:

Null_Mutex (void) {}

?Null_Mutex (void) {}

int remove (void) { return 0; }
```

```
int acquire (void) const { return 0; }

int try_acquire (void) const { return 0; }

int release (void) const { return 0; }

};
```

如上面的代码所示，Null_Mutex类将acquire和release方法实现为“空操作”内联函数，编译优化器将把它们完全清除掉。4.6演示Null_Mutex的使用。

4.5.1.6令牌（Token）类

该类提供了一种比Mutex更为通用的同步机制。例如，它实现了“递归互斥体”语义，拥有该令牌的线程可以重新获取它，而不会死锁。此外，当其他线程释放Token时，阻塞并等待该Token的线程严格地按照FIFO（先进先出）的顺序被服务。相反，Mutex并不严格地实施一种获取顺序。

Token类的接口如下所示：

```
class Token

{

public:

// Initialization and termination.

Token (const char *name = 0, void * = 0);

?Token (void);


// Acquire the token, sleeping until it is

// obtained or until <timeout> expires.

// If some other thread currently holds

// the token then <sleep_hook> is called

// before our thread goes to sleep.

int acquire (void (*sleep_hook)(void *),

                          void *arg = 0,

                          Time_Value *timeout = 0);


// This behaves just like the previous

// <acquire> method, except that it

// invokes the virtual function called

// <sleep_hook> that can be overridden
```

```
// by a subclass of Token.
int acquire (Time_Value *timeout = 0);

// This should be overridden by a subclass
// to define the appropriate behavior before
// <acquire> goes to sleep. By default,
// this is a no-op...
virtual void sleep_hook (void);

// An optimized method that efficiently
// reacquires the token if no other threads
// are waiting. This is useful for if you
// don't want to degrad the quality of
// service if there are other threads
// waiting to get the token.
int renew (int requeue_position = 0,
                            Time_Value *timeout = 0);

// Become interface-compliant with other
// lock mechanisms (implements a
// non-blocking <acquire>).
int tryacquire (void);

// Shuts down the Token instance.
int remove (void);

// Relinquish the token. If there are any
// waiters then the next one in line gets it.
int release (void);

// Return the number of threads that are
// currently waiting to get the token.
int waiters (void);

// Return the id of the current thread that
// owns the token.
```

```
thread_t current_owner (void);

};
```

4.5.2 ACE守卫（Guard）类属

4.5.2.1 Guard类

与C一级的互斥体API相比较，4.5.1.1描述的Mutex包装为同步多线程控制提供了一种优雅的接口。但是，Mutex潜在地容易出错，因为程序员有可能忘记调用release方法（如4.3.7所示）。这可能由于程序员的疏忽或是C++异常的发生而发生。

因此，为改善应用的健壮性，ACE同步机制有效地利用C++类构造器和析构器的语义来确保Mutex锁被自动获取和释放。ACE提供了一个称为Guard、Write_Guard和Read_Guard的类族，确保在进入和退出C++代码块时分别自动获取和释放锁。

Guard类是最基本的守卫机制，定义如下：

```
template <class LOCK>

class Guard

{

public:

// Implicitly and automatically acquire (or try

// to acquire) the lock.

Guard (LOCK &l, int block = 1): lock_ (&l)

{

       result_ = block ? acquire () : tryacquire ();

}


// Implicitly release the lock.

?Guard (void)

{

       if (result_ != -1)

              lock_.release ();

}


// 1 if locked, 0 if can't acquire lock

// (errno will contain the reason for this).
```

```
int locked (void)
{
       return result_ != -1;
}

// Explicitly release the lock.
int remove (void)
{
       return lock_->remove ();
}

// Explicitly acquire the lock.
int acquire (void)
{
       return lock_->acquire ();
}

// Conditionally acquire the lock (i.e., won't block).
int tryacquire (void)
{
       return lock_->tryacquire ();
}

// Explicitly release the lock.
int release (void)
{
       return lock_->release ();
}

private:
// Pointer to the LOCK we're guarding.
LOCK *lock_;

// Tracks if acquired the lock or failed.
int result_;
};
```

Guard类的对象定义一“块”代码，在其上锁被自动获取，并在退出块时自动释放。注意这种机制也能为Mutex、RW_Mutex和Semaphore同步封装工作。这演示了使用C++包装的另一个好处：通过改编不必要地互不兼容的接口（比如Solaris 2.x信号量和互斥体），这些封装促进了接口的一致性。

缺省地，上面所示的Guard类构造器将会阻塞，直到锁被获取。会有这样的情况，程序必须使用非阻塞的acquire调用（例如，防止死锁）。因此，可以传给ACE Guard的构造器第二个参数，指示它使用锁的try_acquire方法，而不是acquire。随后调用者可以使用Guard的locked方法来原子地测试实际上锁是否已被获取。

Read_Guard和Write_Guard类有着与Guard类相同的接口。但是，它们的acquire方法分别对锁进行读和写。

4.5.2.2 Thread_Control类

Thread_Control类用于与Thread_Manager类相结合，以使在线程的发起函数中的优雅的线程终止和清扫活动自动化。例如，Thread_Control的构造器存储状态信息。当最初用于调用线程的函数终止时，该信息自动地从相关联的Thread_Manager中移除该线程。该技术能正确地工作，不管（1）函数经由哪一条路径被执行，以及（2）是否有异常被扔出。就这一点来说，Thread_Control类的行为与4.5.2.1介绍的Guard类相类似。

Thread_Control类的接口给出如下：

```
class Thread_Control

{

public:

// Initialize the thread control object.

// If INSERT != 0, then register the thread

// with the Thread_Manager.

Thread_Control (Thread_Manager *, int add = 0);


// Implicitly kill the thread on exit and

// remove it from its associated Thread_Manager.

?Thread_Control (void);


// Explicitly kill the thread on exit and

// remove it from its associated Thread_Manager.

void *exit (void *status);


// Set the exit status (and return status).
```

```
void *set_status (void *status);


// Get the current exit status.

void *get_status (void);

};
```

4.5.3 ACE条件（Condition）类属

4.5.3.1 Condition类

Condition类用于在涉及共享数据的条件表达式的状态发生变化时进行阻塞。它封装Solaris线程和POSIX pthreads cond_t同步变量。Condition类接口给出如下：

```
template <class MUTEX>

class Condition

{

public:

// Initialize the condition variable.

Condition (const MUTEX &m,

                           int type = USYNC_THREAD,

                    void *arg = 0);


// Implicitly destroy the condition variable.

?Condition (void);


// Explicitly destroy the condition variable.

int remove (void);


// Block on condition, or until absolute

// time-of-day has elapsed. If abstime

// == 0 use blocking wait().

int wait (Time_Value *abstime = 0) const;


// Signal one waiting thread.

int signal (void) const;
```

```
// Signal *all* waiting threads.

int broadcast (void) const;


private:

cond_t cond_;

// Reference to mutex lock.

const MUTEX &mutex_;

};
```

注意Win32不提供条件变量抽象。因此，ACE线程库使用其他像信号量和互斥体这样的ACE组件来实现条件变量。

4.5.3.2 Null_Condition类

Null_Condition类是上面描述的Condition接口的一种零开销实现。它的方法全都实现为空操作。这对于根本不需要互斥的情况很有用（例如，一个特定的程序或服务将*总是*在单线程控制中运行，并且/或者不与其他线程争夺对共享资源的访问）。使用Null *类的原因是允许应用参数化它们所需的同步类型，而*不*需要改动应用代码。Null_Condition类接口给出如下：

```
template <class MUTEX>

class Null_Condition

{

public:

Null_Condition (const MUTEX &m,

                                int type = 0,

                          void *arg = 0) {}


?Null_Condition (void) {}


int remove (void) { return 0; }


int wait (Time_Value *abstime = 0) const

{

       errno = ETIME; return -1;

}
```

```
int signal (void) const

{

       errno = ETIME; return -1;

}


int broadcast (void) const

{

       errno = ETIME; return -1;

}

};
```

Null_Condition类在“精神”上与4.5.1.5描述的Null_Mutex类是一样的。

4.5.4 ACE线程管理器类属

4.5.4.1 Thread_Manager类

Thread_Manager类含有一组机制来对进行协作、以实现集体动作的成组线程进行管理。例如，Thread_Manager类提供的机制（比如suspend_all和resume_all）允许任意数量的参与线程被原子地挂起或恢复。Thread_Manager类还将应用与不同风格的多线程机制（比如Solaris、POSIX和Win32）间的许多不兼容特性屏蔽开来。

Thread_Manager类的接口演示如下：

```
class Thread_Manager

{

public:

// Initialize the thread manager.

Thread_Manager (int size);


// Implicitly destroy thread manager.

?Thread_Manager (void);


// Initialize the manager with room

// for SIZE threads.
```

```
int open (int size = DEFAULT_SIZE);

// Release all resources.
int close (void);

// Create a new thread.
int spawn (THR_FUNC,
                          long, thread_t * = 0,
           void *stack = 0,
           size_t stack_size = 0);

// Create N new threads.
int spawn_n (int n, THR_FUNC,
                          void *args, long flags);

// Clean up when a thread exits.
void *exit (void *status);

// Blocks until there are no
// more threads running.
void wait (void);

// Resume all stopped threads.
int resume_all (void);

// Suspend all threads.
int suspend_all (void);

// Send signum to all stopped threads.
int kill_all (int signal);

private:
// ...
};
```

4.5.4.2 Thread_Spawn类

Thread_Spawn类提供的一种标准机制管理线程的创建，以并发地处理来自客户的请求。该类作为“线程工厂”，接受来自客户的连接，并“按需”派生线程，以运行由用户提供的服务处理器（SVC_HANDLER）指定的服务。

Thread_Spawn类的接口给出如下：

```
template <class SVC_HANDLER,
       class PEER_ACCEPTOR,
       class PEER_ADDR>
class Thread_Spawn
: public Acceptor <SVC_HANDLER,
       PEER_ACCEPTOR,
PEER_ADDR>
{
public:
// = Initialization methods.
Thread_Spawn (Thread_Manager *tm, Reactor *);
virtual int open (const PEER_ADDR &sia, Reactor *);

protected:
virtual int handle_input (int fd);
// Template method that accepts connection
// and spawns a thread.

virtual int handle_close (int fd, Reactor_Mask);
// Called when this factory is closed down.

virtual SVC_HANDLER *make_svc_handler (void);
// Factory method that creates an appropriate
// SVC_HANDLER *.

virtual int thr_flags (void);
// Returns the flags used to spawn a thread.
};
```

注意此类是怎样从ACE Acceptor类继承的，后者是一种通用工厂，用于被动地连接客户和创建服务处理器[30]。

4.5.5 ACE主动对象（Active Object）类属

4.5.5.1 Task（任务）类

Task类是ACE中用于创建用户定义的*主动对象*[16]和*被动对象*（它们处理应用消息）的中心机制。ACE Task可进行以下活动：

- 可被动态链接；
- 可用作I/O操作的端点；
- 可与多个线程控制相关联（也就是，成为所谓的“主动对象”）；
- 可在队列中存储消息，用于后续处理；
- 可执行用户定义的服务。

Task抽象类定义的接口被派生类继承和实现，以提供应用特有的功能。它之所以是一个抽象类，是因为它的接口定义了一些在下面描述的纯虚方法（open、close、put和svc）。通过使Stream类属提供的与应用无关的组件与继承并使用这些组件的应用特有的子类去耦合，将任务定义为抽象类增强了复用。同样地，纯虚函数的使用允许C++编译器确保任务的子类遵从它提供以下功能的义务：

- *初始化和终止方法*：派生自Task的子类必须实现open和close方法，它们执行应用特有的Task初始化和终止活动。这些活动通常分配和释放资源，比如连接控制块、I/O句柄和同步锁。

  Task可与Module（模块）一起或是分开定义和使用。当与模块一起使用时，任务被成对地存储：一个Task子类处理读端的自下游发到该模块层的消息，另一个处理写端的自上游发到该模块层的消息。

  当一个模块被插入流，或从流中被移除时，它的读端和写端的任务子类的open和close方法分别自动地被ASX构架调用。

- *应用特有的处理方法*：除了open和close方法，Task的子类还必须定义put和svc方法。这些方法在消息上执行应用特有的处理功能。例如，当消息到达流的头或尾时，作为调用流中的每个任务的put和/或svc方法的结果，它们被“护送”通过一系列互连的任务。
  put方法在流中某层的一个Task传递消息给另一层中相邻的Task时被调用。put方法相对于它的调用者*同步地*运行，也就是，它从起先调用它的put方法的Task那里借用线程控制。该线程控制通常从应用进程“自下而上”、从处理I/O设备中断[31]的线程池“自上而下”发起，或是由事件分派机制（比如在面向连接的传输协议模块中、用于触发重发的定时器驱动的呼出队列）在流内部发起。

  如果一个ACE Task作为*被动对象*执行（也就是，它总是从调用者那里借用线程控制），Task::put方法就是进入该Task的入口，并用作Task在其中执行它的工作的上下文。相

反，如果一个ACE Task作为*主动对象*执行，Task::svc方法就被用于相对于其他Task*异步地*执行应用特有的处理。不像put，svc方法并不直接从相邻的Task那里调用。相反，它被与它从属的Task相关联的一个分离的线程调用。该线程为Task的svc方法提供执行上下文和线程控制。svc方法运行一个事件循环，持续地等待消息到达Task的Message Queue（消息队列，见下一条目）。

在put或svc方法的实现中，消息可以通过任务的put_next实用方法传递给流中相邻的Task。put_next调用驻留在相邻层中的下一个Task的put方法。这个对put的调用可以从调用者那里借用线程控制，并立即处理消息（也就是，图4-4（1）演示的同步处理方法）。相反地，put方法可以将消息入队，并将处理推迟给它的在分离的线程控制中执行的svc方法（也就是，图4-4（2）演示的异步处理方法）。如在[1]中所讨论的，选择特定的处理方法对性能和编程的容易程度有着显著的影响。

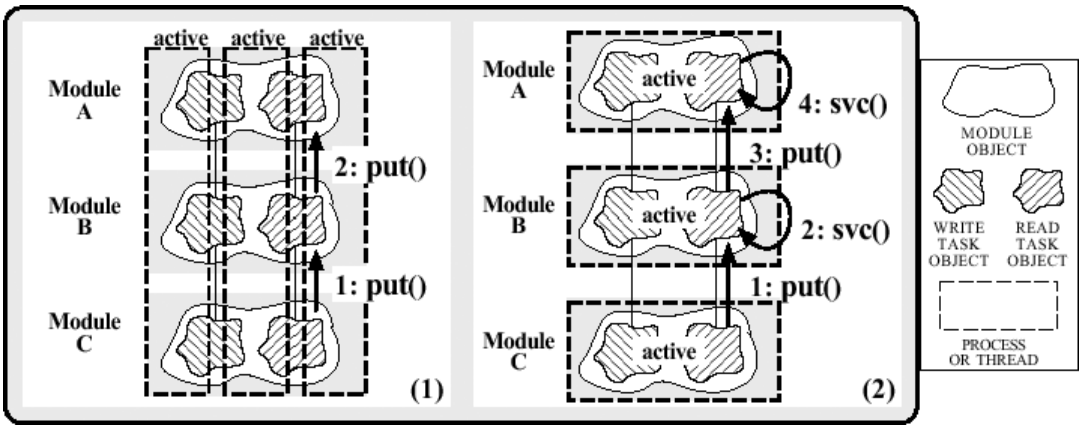

图4-4 调用put和svc方法的可选方案

- *消息排队机制*：除了open、close、put和svc纯虚方法接口，每个任务还含有一个Message Queue（消息队列）。Message Queue是ACE中的标准组件，用于在Task间传递消息。而且，当Task作为主动对象执行时，它的Message Queue用于缓存一系列数据消息和控制消息，以在svc方法中作后续处理。在消息到达时，svc方法使消息出队，并执行Task子类的应用特有的处理工作。

  Message Queue中可以出现两种类型的消息：简单的和复合的。简单消息含有单个Message Block，而复合消息含有多个链接在一起的Message Block。复合消息通常由一个*控制*块和紧随的一或多个*数据*块组成。控制块含有“簿记”信息（比如目的地址和长度域），而数据块含有消息的实际内容。通过传递消息指针而不是拷贝数据，在Task间传递Message Block的开销被降到了最低。

  Message Queue含有一对高低水位标变量，用于在流中的相邻Module间实现层到层的流控制。高水位标指示消息队列在进行流控制之前所愿意缓存的消息字节数。低水位标指示在先前已进行流控制的Message Queue不再被视为满的“水位”。

Task类的接口提供如下：

```
template <class SYNCH>
```

```
class Task : public Service_Object
{
public:
// Initialization/termination methods.
Task (Thread_Manager *thr_mgr = 0,
Message_Queue<SYNCH> *mp = 0);
virtual int open (void *flags = 0) = 0;
virtual int close (u_long = 0) = 0;

// Transfer msg into the queue to handle
// immediate processing.
virtual int put (Message_Block *, Time_Value *tv = 0) = 0;

// Run by a daemon thread to handle
// deferred processing.
virtual int svc (void) = 0;

protected:
// Turn the task into an active object..
int activate (long flags);

// Routine that runs the service routine
// as a daemon thread.
static void *svc_run (Task<SYNCH> *);

// Tests whether a message can be enqueue
// without blocking.
int can_put (Message_Block *);

// Insert message into the message list.
int putq (Message_Block *, Time_Value * = 0);

// Extract the first message from the list.
int getq (Message_Block *&, Time_Value * = 0);

// Return a message to the queue.
```

```
int ungetq (Message_Block *, Time_Value * = 0);

// Transfer message to the adjacent Task
// in a Stream.
int put_next (Message_Block *, Time_Value * = 0);

// Turn the message back around.
int reply (Message_Block *, Time_Value * = 0);

// Task utility routines to identify names.
const char *name (void) const;
Task<SYNCH> *sibling (void);
Module<SYNCH> *module (void) const;

// Check if queue is a reader.
int is_reader (void);

// Check if queue is a writer.
int is_writer (void);

// Special routines corresponding to
// certain message types.
int flush (u_long flag);

// Manipulate watermarks.
void water_marks (IO_Cntl_Msg::IO_Cntl_Cmds, size_t);
};
```

4.5.6 杂项ACE并发类

4.5.6.1 Thread类工具

Thread类工具在C++包装中封装Solaris、POSIX和Win32族的线程创建、终止和管理例程族。该类提供了一种映射到Solaris线程、POSIX Pthreads和Win32线程的通用接口。

Thread类的接口提供如下：

```
typedef void *(*THR_FUNC)(void *);
class Thread
{
public:
// Spawn N new threads, which execute
// "func" with argument "arg".
static int spawn_n (size_t n, THR_FUNC func,
                                    void *arg, long flags,
                                    void *stack = 0,
                                    size_t stack_size = 0);

// Spawn a new thread, which executes
// "func" with argument "arg".
static int spawn (THR_FUNC,
                                    void *arg, long,
                                    thread_t * = 0,
                                    void *stack = 0,
                                    size_t stack_size = 0,
                                    hthread_t *t_handle = 0);

       // Wait for one or more threads to exit.
static int join (hthread_t, hthread_t *, void **);

// Suspend the execution of a thread.
static int suspend (hthread_t);

// Continue the execution of a
// previously suspended thread.
static int resume (hthread_t);

       // Send signal signum to the thread.
static int kill (thread_t, int signum);
```

```
// Return the unique ID of the thread.
static thread_t self (void);

// Yield the thread to another.
static void yield (void);

// Exit current thread, returning "status".
static void exit (void *status);

// Set LWP concurrency level of the process.
static int setconcurrency (int new_level);

// Get LWP concurrency level of the process.
static int getconcurrency (void);

static int sigsetmask (int how,
                       const sigset_t
                       *set,
                       sigset_t *oset =
                       0);
// Change and/or examine calling thread's
// signal mask.

static int keycreate (thread_key_t *keyp,
                      void (*)(void
                      *value));
// Allocates a <keyp> that is used to
// identify data that is specific to each
// thread in the process. The key is global
// to all threads in the process.

static int setspecific (thread_key_t key, void *value);
// Bind value to the thread-specific data
// key, <key>, for the calling thread.

static int getspecific (thread_key_t key, void **valuep);
// Stores the current value bound to <key>
```

```
// for the calling thread into the location
// pointed to by <valuep>.
};
```

4.5.6.2 Atomic_Op类

Atomic_Op类将同步透明地参数化进基本的算术运算中。

```
template <class LOCK, class TYPE>
class Atomic_Op
{
public:
// Initialize count_ to 0.
Atomic_Op (void);

// Initialize count_ to c.
Atomic_Op (TYPE c);

// Atomically increment count_.
TYPE operator++ (void);

// Atomically increment count_ by inc.
TYPE operator+= (const TYPE inc);

// Atomically decrement count_.
TYPE operator-- (void);

// Atomically decrement count_ by dec.
TYPE operator-= (const TYPE dec);

// Atomically compare count_ with rhs.
TYPE operator== (const TYPE rhs);

// Atomically check if count_ >= rhs.
TYPE operator>= (const TYPE rhs);
```

```
// Atomically check if count_ > rhs.
TYPE operator> (const TYPE rhs);

// Atomically check if count_ <= rhs.
TYPE operator<= (const TYPE rhs);

// Atomically check if count_ < rhs.
TYPE operator< (const TYPE rhs);

// Atomically assign rhs to count_.
void operator= (const TYPE rhs);

// Atomically return count_.
operator TYPE ();

private:
LOCK lock_;
TYPE count_;
};
```

4.5.6.3 Barrier（栅栏）类

Barrier类实现“栅栏同步”，它对于许多类型的并行科学应用特别有用。该类允许count数量的线程同步它们的完成（所谓的“栅栏同步”）。它的实现使用了“子栅栏生成编号”方案来避免过度的开销，并确保所有线程都正确地离开栅栏。

```
class Barrier
{
public:
// Initialize the barrier to
// synchronize count threads.
Barrier (u_int count,
              int type = USYNC_THREAD,
              void *arg = 0);
```

```
// Block the caller until all count threads

// have called wait() and then allow all

// the caller threads to continue in parallel.

int wait (void);

};
```

4.5.6.4 TSS类

TSS类允许“物理上”线程专有的（也就是，线程私有的）对象被“逻辑地”当作程序的全局对象进行访问。该类所基于的底层“线程专有存储”模式在[29]中描述。

下面是ACE TSS类的公共接口：

```
template <class TYPE>

class TSS

{

public:

// If caller passes a non-NULL ts_obj *

// this is used to initialize the

// thread-specific value. Thus, calls

// to operator->() will return this value.

TSS (TYPE *ts_obj = 0);


// Get the thread-specific object for the key

// associated with this object. Returns 0

// if the data has never been initialized,

// otherwise returns a pointer to the data.

TYPE *ts_object (void);


// Use a "smart pointer" to obtain the

// thread-specific object associated with

// the key.

TYPE *operator-> ();

};
```

4.6 使用ACE OO线程封装库

这一部分介绍若干例子，演示ACE线程库中的关键特性的使用。请查阅4.5描述的接口，以确定ACE并发组件的行为。

4.6.1 用于消息多路分离的映射管理器

选择一种有着适当语义的互斥机制常常取决于类被使用的“上下文”。下面的例子演示通用ACE工具包[32]中的“映射管理器”的接口和实现。此组件通常用于在网络服务器中将外部的标识符（比如端口号或连接id）映射到内部的标识符（比如指向卫星输出链路开始流控制后用于存储消息的队列的指针）。Map_Manager的部分接口和实现显示如下：

```
template <class EXT_ID,
          class INT_ID,
          class LOCK>
class Map_Manager
{
public:
// Associate EXT_ID with the INT_ID.
int bind (EXT_ID ext_id, const INT_ID *int_id)
{
          Write_Guard<LOCK> monitor (lock_);
          // ...
}

// Break any association of EXT_ID.
int unbind (EXT_ID ext_id)
{
          Write_Guard<LOCK> monitor (lock_);
          // ...
}

// Locate INT_ID associated with EXT_ID and
// pass out parameter via INT_ID.
// If found return 0, else -1.
```

```
int find (EXT_ID ext_id, INT_ID &int_id)

{

       Read_Guard<LOCK> monitor (lock_);

       if (locate_entry (ext_id, int_id)

              // ext_id is successfully located.

              return 0;

       else

              return -1;

}


private:

LOCK lock_;

// ...

};
```

该方法的一个好处是无论哪条执行路径退出方法，lock_都会被释放。例如，if/else语句的任何一条分支从find方法中返回，lock_都会被正确释放。此外，如果在locateEntry助手方法的处理过程中有异常发生，“作为资源获取/释放的构造器/析构器”习语也会正确地释放lock_。这是有用的，因为C++异常处理机制被设计为在从异常被扔出的块中退出时、调用所有必需的析构器。注意我们使用了获取和释放lock_的显式调用来编写find的定义，也就是：

```
int find (EXT_ID ext_id, INT_ID &int_id)

{

lock_.acquire ();


if (locateEntry (ext_id, int_id)

{

       // ext_id is successfully located.

       lock_.release ();

       return 0;

}

else

{

       lock_.release ();

       return -1;

}

}
```

find方法的逻辑更不自然，空间的使用效率也低下。此外，也不能保证有异常从locateEntry方法中扔出时，lock_会被释放。

Map_Manager模板类用于实例化的LOCK类型取决于锁被使用时程序代码中的并行的特定结构。例如，在某些情况下，这样声明是有用的：

```
typedef Map_Manager <Addr, TCB, Mutex>MAP_MANAGER;
```

并使得所有对find、bind和unbind的调用自动地被序列化。在其他的一些情况下，可以通过使用Null_Mutex类来关闭同步，而不用改变已有的库代码：

```
typedef Map_Manager <Addr, TCB, Null_Mutex>MAP_MANAGER;
```

还有一种情况，对find的调用可能*远*比对bind或unbind的调用频繁。在这样的情况下，使用RW_Mutex读者/作者锁可能是有意义的：

```
typedef Map_Manager <Addr, TCB, RW_Mutex>MAP_MANAGER;
```

通过使用C++包装和模板，我们可以创建一个高度可移植、平台无关的互斥类接口，而无需对我们所用的不同同步机制强加任何语法约束。通过使用模板来参数化锁定类型，无需改动应用代码、或只需很少一点改动就可以适应新的同步语义。但是，像往常一样，选择适当的同步机制需要通过剖面和经验的衡量进行指导。

4.6.2 线程安全的消息排队机制

该例子演示ACE Condition包装（4.5.1.1）和ACE Mutex包装（4.5.3.1）的使用。代码从Message_Queue类中摘录，该类包含在4.5.5.1描述的Task类中。Message_Queue可被同步策略类型参数化，以获取所期望的并发控制级。缺省地，并发控制级是“线程安全”，如ACE Synch.h文件中的MT_Synch类所定义的：

```
class MT_Synch

{

public:

typedef Condition<Mutex> CONDITION;

typedef Mutex MUTEX;

};、
```

如果MT_Synch被用于实例化Message_Queue，所有的公共方法都将是线程安全的，但同时也带来相应的开销。相反，如果Null_Synch类用于实例化Message_Queue，所有公共方法都不是线程安全的，同时也就没有额外的开销。Null_Synch也在Synch.h中定义，如下所示：

```
class Null_Synch
{
public:
typedef Null_Condition<Null_Mutex> CONDITION;
typedef Null_Mutex MUTEX;
};
```

在4.3.5开始处的run_svc函数中有Message_Queue的使用实例。Message_Queue在系统V STREAM[33]和BSD UNIX[34]提供的消息排队和缓冲区管理机制的基础上构建。

ACE Message_Queue由一或多个通过prev_和next_指针链接在一起的Message_Block组成。这样的结构可以高效地操作任意大的消息，而不会导致巨大的内存拷贝开销。

```
// The contents of a message are represented
// internally by a Message_Block.
class Message_Block
{
public:
Message_Block (size_t size,
                          Message_Type type = MB_DATA,
                          Message_Block *cont = 0,
                          char *data = 0);
// ...
};
```

Message_Queue是一种线程安全的消息排队机制。注意C++“traits”习语的使用将Condition和Mutex类型合并进了单个模板参数。

```
template <class SYNCH = MT_Synch>
class Message_Queue
{
public:
```

```
// Default high and low water marks.
enum
{
       // 0 is the low water mark.
       DEFAULT_LWM = 0,

       // 1 K is the high water mark.
       DEFAULT_HWM = 4096,

       // Message queue was active
       // before activate() or deactivate().
       WAS_ACTIVE = 1,

       // Message queue was inactive
       // before activate() or deactivate().
       WAS_INACTIVE = 2
};

// Initialize a Message_Queue.
Message_Queue (size_t hwm = DEFAULT_HWM,
                           size_t lwm = DEFAULT_LWM);

// Destroy a Message_Queue.
?Message_Queue (void);

/* Checks if queue is full/empty. */
int is_full (void) const;
int is_empty (void) const;

// Enqueue and dequeue a Message_Block *.
int enqueue_tail (Message_Block *new_item,
                                  Time_Value *tv = 0);
       int enqueue_head (Message_Block *new_item,
                                        Time_Value *tv =
                                        0);
       int dequeue_head (Message_Block *&first_item,
```

```
                                Time_Value *tv = 0);

// Deactivate the queue and wakeup all threads
// waiting on the queue so they can continue.
int deactivate (void);

// Reactivate the queue so that threads can
// enqueue and dequeue messages again.
int activate (void);

private:
// Routines that actually do the enqueueing
// and dequeueing (assumes locks are held).
int enqueue_tail_i (Message_Block *);
int enqueue_head_i (Message_Block *);
int enqueue_head_i (Message_Block *&);

// Check the boundary conditions.
int is_empty_i (void) const;
int is_full_i (void) const;

// Implement activate() and deactivate()
// methods (assumes locks are held).
int deactivate_i (void);
int activate_i (void);

// Pointer to head of Message_Block list.
Message_Block *head_;

// Pointer to tail of Message_Block list.
Message_Block *tail_;

// Lowest number before unblocking occurs.
int low_water_mark_;

// Greatest number of bytes before blocking.
```

```
int high_water_mark_;


// Current number of bytes in the queue.

int cur_bytes_;


// Current number of messages in the queue.

int cur_count_;


// Indicates that the queue is inactive.

int deactivated_;


// C++ wrapper synchronization primitives

// for controlling concurrent access.

SYNCH::MUTEX lock_;

SYNCH::CONDITION notempty_;

SYNCH::CONDITION notfull_;

};
```

Message_Queue类的实现显示如下。Message_Queue的构造器创建一个空的消息列表，并初始化Condition对象。注意Mutex的lock_被它的缺省构造器自动创建。

```
template <class SYNCH>

Message_Queue::Message_Queue (size_t hwm,

                                                size_t
                                                lwm)

: notfull_ (lock_), notempty_ (lock_)

{

// ...

}
```

下面的代码检查队列是否为“空”（也就是，没有消息在其中）或“满”（也就是，在其中的字节的数目多于high_water_mark）。注意这些方法（像下面的其他方法一样）怎样利用一种模式，公共方法藉此来获取锁，而私有方法假定锁已被持有。

```
template <class SYNCH> int

Message_Queue<SYNCH>::is_empty_i (void) const

{
```

```
return cur_bytes_ <= 0 && cur_count_ <= 0;

}


template <class SYNCH> int

Message_Queue<SYNCH>::is_full_i (void) const

{

return cur_bytes_ > high_water_mark_;

}


template <class SYNCH> int

Message_Queue<SYNCH>::is_empty (void) const

{

Guard<SYNCH::MUTEX> monitor (lock_);

return is_empty_i ();

}


template <class SYNCH> int

Message_Queue<SYNCH>::is_full (void) const

{

Guard<SYNCH::MUTEX> monitor (lock_);

return full ();

}
```

下面的方法用于启用和停用Message_Queue。deactivate方法停用队列，并唤醒所有等待在该队列上的线程，以使它们继续。没有消息被从队列中移除。其他任何被调用的方法都立即返回-1，且errno == ESHUTDOWN，直到队列被再次激活。这些信息允许调用者检测状态的变化。

```
template <class SYNCH> int

Message_Queue<SYNCH>::deactivate (void)

{

Guard<SYNCH::MUTEX> m (lock_);

return deactivate_i ();

}


template <class SYNCH> int

Message_Queue<SYNCH>::deactivate_i (void)
```

```
{

int current_status = deactivated_ ? WAS_INACTIVE : WAS_ACTIVE;


// Wake up all the waiters.

notempty_.broadcast ();

notfull_.broadcast ();


deactivated_ = 1;

return current_status;

}
```

activate方法重新启用队列，以使线程能够再次让消息入队或出队。如果队列在调用前是不活动的，该方法返回WAS_INACTIVE；如果队列在调用前是活动的，就返回WAS_ACTIVE。

```
template <class SYNCH> int

Message_Queue<SYNCH>::activate (void)

{

Guard<SYNCH::MUTEX> m (lock_);

return activate_i ();

}


template <class SYNCH> int

Message_Queue<SYNCH>::activate_i (void)

{

int current_status =

       deactivated_ ? WAS_INACTIVE : WAS_ACTIVE;

deactivated_ = 0;

return current_status;

}
```

enqueue_head方法在队列的前面插入一个新条目。像其他的入队和出队方法一样，如果tv参数为NULL，调用者将阻塞直到可以继续执行。否则，调用者将阻塞，等待*tv所指定的数量的时间。但是当队列被关闭、有信号发生，或tv中指定的时间过去了，阻塞的调用都会返回，且errno == EWOULDBLOCK。

```
template <class SYNCH> int
```

```
Message_Queue<SYNCH>::enqueue_head
(Message_Block *new_item, Time_Value *tv)
{
Guard<SYNCH::MUTEX> monitor (lock_);

if (deactivated_)
{
       errno = ESHUTDOWN;
       return -1;
}

// Wait while the queue is full.
while (is_full_i ())
{
       // Release the lock_ and wait for
       // timeout, signal, or space becoming
       // available in the list.
       if (notfull_.wait (tv) == -1)
       {
              if (errno == ETIME)
                            errno = EWOULDBLOCK;
                            return -1;
       }

       if (deactivated_)
       {
              errno = ESHUTDOWN;
              return -1;
       }
}

// Actually enqueue the message at the
// head of the list.
enqueue_head_i (new_item);

// Tell any blocked threads that the
```

```
// queue has a new item!

notempty_.signal ();


return 0;

}
```

注意当队列先前为空时，这个方法是怎样只发送信号给notempty_条件对象的。通过减少不必要的信号发送所导致的上下文切换的数量，这样的优化使性能得到了提高。其他两个入队和出队方法也执行类似的优化。

enqueue_tail方法在队列的末尾插入新条目。它返回的是队列中条目的数量。

```
template <class SYNCH> int

Message_Queue<SYNCH>::enqueue_tail

(Message_Block *new_item, Time_Value *tv)

{

Guard<SYNCH::MUTEX> monitor (lock_);

if (deactivated_)

{

       errno = ESHUTDOWN;

       return -1;

}


// Wait while the queue is full.

while (is_full_i ())

{

       // Release the lock_ and wait for

       // timeout, signal, or space becoming

       // available in the list.

       if (notfull_.wait (tv) == -1)

       {

              if (errno == ETIME)

                            errno = EWOULDBLOCK;

              return -1;

       }


       if (deactivated_)
```

```
        {
                errno = ESHUTDOWN;
                return -1;
        }
}

// Actually enqueue the message at
// the end of the list.
enqueue_tail_i (new_item);

// Tell any blocked threads that
// the queue has a new item!
notempty_.signal ();

return 0;
}
```

dequeue_head方法移除队列中最前面的条目，并将它传回给调用者。该方法返回队列中所余条目的数目。

```
template <class SYNCH> int
Message_Queue<SYNCH>::dequeue_head
(Message_Block *&first_item, Time_Value *tv)
{
Guard<SYNCH::MUTEX> monitor (lock);

// Wait while the queue is empty.
while (is_empty_i ())
{
        // Release the lock_ and wait for
        // timeout, signal, or a new message
        // being placed in the list.
        if (notempty_.wait (tv) == -1)
        {
                if (errno == ETIME)
                        errno = EWOULDBLOCK;
```

```
                return -1;
        }

        if (deactivated_)
        {
                errno = ESHUTDOWN;
                return -1;
        }
}

// Actually dequeue the first message.
dequeue_head_i (first_item);

// Tell any blocked threads that
// the queue is no longer full.
notfull_.signal ();

return 0;
}
```

下面的代码演示一个使用Message_Queue的经典的“有界缓冲区”的ACE实现。程序使用两个线程来并发地将stdin拷贝到stdout。图4-5演示并发运行的ACE组件间的关系。生产者线程从stdin流中读取数据，创建消息，然后将消息放入Message_Queue。消费者线程使消息出队并将它写到stdout。为节省空间，省略了大多数错误检查。

```
#include "Message_Queue.h"
#include "Thread_Manager.h"

typedef Message_Queue<MT_Synch> MT_Message_Queue;

// Global thread manager.
static Thread_Manager thr_mgr;
```

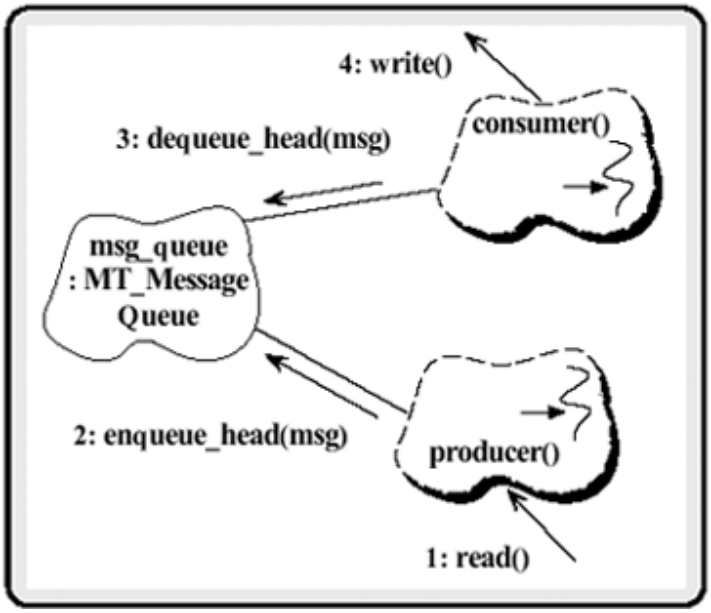

图4-5 并发生产者和消费者

生产者从stdin中将数据读取到消息中，然后为消费者排队此消息。没有更多数据可读时，一个NULL指针就被放到队列中。该指针用于通知消费者何时退出。

```
static void *producer (MT_Message_Queue *msg_queue)

{

// Insert thread into thr_mgr.

Thread_Control tc (&thr_mgr);

char buf[BUFSIZ];


for (int n; ; )

{

       // Allocate a new message.

       Message_Block *mb = new Message_Block (BUFSIZ);


       n = read (0, mb->rd_ptr (), mb->size ());

       if (n <= 0)

       {

              // Shutdown message to the other

              // thread and exit.

              mb->length (0);

              msg_queue->enqueue_tail (mb);
```

```
        }

        // Send the message to the other thread.
        else
        {
                mb->wr_ptr (n);
                msg_queue->enqueue_tail (mb);
        }
}

// The destructor of Thread_Control removes
// the exiting thread from the
// Thread_Manager automatically.
return 0;
}
```

consumer从Message_Queue中取出一个消息，将它写到stderr流中，并删除此消息。producer发送一个NULL指针，通知消费者停止读取并退出。

```
static void *consumer(MT_Message_Queue *msg_queue)
{
Message_Block *mb = 0;

// Insert thread into thr_mgr.
Thread_Control tc (&thr_mgr);
int result = 0;

// Keep looping, reading a message out
// of the queue, until we timeout or get a
// message with a length == 0, which signals
// us to quit.
for (;;)
{
        result = msg_queue->dequeue_head (mb);
        if (result == -1)
                return -1;
```

```
        int length = mb->length ();
        if (length > 0)
                ::write (1, mb->rd_ptr (), length);

        delete mb;

        if (length == 0)
                break;
}

// The destructor of Thread_Control removes
// the exiting thread from the
// Thread_Manager automatically.
return 0;
}
```

main函数派生两个线程来运行producer和consumer函数，以并行地拷贝stdin到stdout。

```
int main (int argc, char *argv[])
{
// Use the thread-safe instantiation
// of Message_Queue.
Message_Queue msg_queue;

thr_mgr.spawn (THR_FUNC (producer),
                            (void *) &msg_queue,
                            THR_NEW_LWP | THR_DETACHED);
thr_mgr.spawn (THR_FUNC (consumer),
                            (void *) &msg_queue,
                            THR_NEW_LWP | THR_DETACHED);

// Wait for producer/consumer threads to exit.
thr_mgr.wait ();
return 0;
```

```
}
```

4.6.3 并发网络数据库服务器

下面的例子演示一个使用ACE线程管理组件开发的并发网络数据库服务器。客户请求触发服务器根据它们唯一的数字ID查找“雇员”。如果找到匹配项，就将名字返回给客户。

每个给服务器的客户请求都并行地运行。这个例子演示了Thread_Manager和Thread_Control类的使用。此外，它还演示了socket的ACE C++包装类的使用[35]。

下面所示的代码意在简化例子，并没有表现一个高度健壮和高效的实现是如何开发的。例如，产品实现会设置所派生的绑定线程数目的上限，以避免消耗大量的内核资源。此外，产品实现还可能会明确地使用一个更为成熟的数据库方案。

```
#include "SOCK_Acceptor.h"

#include "Thread_Manager.h"


// Per-process thread manager.

Thread_Manager thr_mgr;


// Function called when a new thread is created.

// This function is passed a connected client

// SOCK_Stream, which it uses to receive a

// database lookup request from a client.

static void *lookup_name (ACE_HANDLE handle)

{

// Local thread control object.

Thread_Control tc (&thr_mgr);


enum

{

       // Maximum line we'll read from a client.

       MAXLINE = 255,


       // Maximum size of employee name.

       EMPNAMELEN = 512

};
```

```
// Simple read-only database.
static struct
{
	int emp_id;
	const char emp_name[EMPNAMELEN];
} emp_db[] =
	{
		123, "John Wayne Bobbit",
		124, "Cindy Crawford",
		125, "O. J. Simpson",
		126, "Bill Clinton",
		127, "Rush Limbaugh",
		128, "Michael Jackson",
		129, "George Burns",
		0, ""
	};

SOCK_Stream new_stream;
char recvline[MAXLINE];
char sendline[MAXLINE];

new_stream.set_handle (handle);

ssize_t n = new_stream.recv (recvline, MAXLINE);
int emp_id = atoi (recvline);
	int found = 0;

for (int index = 0; found == 0 && emp_db[index].emp_id; index++)
	if (emp_id == emp_db[index].emp_id)
	{
		found = 1;
		n = sprintf (sendline, "%s", emp_db[index].emp_name);
	}

if (found == 0)
	n = sprintf (sendline, "%s", "ERROR");
```

```
new_stream.send_n (sendline, n + 1);
new_stream.close ();

// The destructor of Thread_Control removes the
// exiting thread from the Thread_Manager
// automatically.
return 0;
}

// Default port number.
static const int default_port = 5000;

int main (int argc, char *argv[])
{
// Port number of server.
u_short port = argc > 1 ? atoi (argv[1]) : default_port;

// Internet address of server.
INET_Addr addr (port);

// Passive-mode listener object.
SOCK_Acceptor server (addr);

SOCK_Stream new_stream;

// Wait for a connection from a client
// (this illustrates a concurrent server).
for (;;)
{
       // Accept a connection from a client.
       server.accept (new_stream);

       // Spawn off a thread-per client request.
       thr_mgr.spawn (THR_FUNC (lookup_name),
```

```
                                (void *) new_stream.get_handle
                                (),

                                THR_BOUND | THR_DETACHED);

}


// NOTREACHED

return 0;

}
```

4.7 结束语

本论文描述了ACE中提供的面向对象线程封装库。ACE线程类库为开发者提供了若干好处：

- 通过使开发者能够在他们的并发应用中始终使用C++和OO，改善了编程风格的一致性。
- 减少了为使应用线程安全化所进行的强制性变动的数量。例如，库中的工具类（比如Atomic_Op、Mutex、RW_Mutex、Semaphore和Condition）改善了底层OS特有的并发机制的可移植性和复用性。
- 消除或使发生微妙的同步错误的可能性降至了最低。若干ACE线程库类（比如Guard和Thread_Control）确保即使发生异常，资源也会被适当地分配和释放。
- 增强了抽象和模块性，而*不用*牺牲性能。使用C++语言特性（比如内联函数和模板）确保了ACE OO线程库所提供的额外功能不会显著地降低效率。

ACE OO线程封装库已在许多商业项目中被使用。这些产品包括Ericsson EOS族电信交换监控应用、Bellcore ATM交换管理软件、Motorola Iridium全球个人通信系统的网络管理子系统和核心基础构造子系统，以及在柯达和西门子的企业级电子医学成像系统。

参考文献

[1] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[2] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalin-giah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, “Beyond Multiprocessing... Multithreading the SunOS Kernel,” in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[3] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[4] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[6] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[7] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[9] Bjarne Stroustrup, *The C++ Programming Language, 2nd Edition*. Addison-Wesley, 1991.

[10] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applications*.Englewood Cliffs, NJ: Prentice Hall, 1992.

[12] Sun Microsystems, *Open Network Computing: Transport Independent RPC*, June 1995.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[14] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[15] W. R. Stevens, *TCP/IP Illustrated, Volume* 1. Reading, Massachusetts: Addison Wesley, 1993.

[16] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[17] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.

[18] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[19] Sun Microsystems, Inc., Mountain View, CA, *SunOS 5.3 Guide to Multi-Thread Programming*, Part number: 801-3176-10 ed., May 1993.

[20] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[21] A. Garg, "Parallel STREAMS: a Multi-Process Implementation," in *Proceedings of the Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

[22] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[23] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[24] Mats Bjorkman and Per Gunningberg, "Locking Strategies in Multiprocessor Implementations of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (San Francisco, California), ACM, 1993.

[25] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*,vol.2, no. 1, 1996.

[26] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the 2nd C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[27] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[28] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

[29] T. Harrison and D. C. Schmidt, "Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent Programs," in *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed Systems*, ACM, October 1995.

[30] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

[31] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64 - 76, January 1991.

[32] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," in *Proceedings of the 12th Annual Sun Users Group Conference*, (San Francisco, CA), pp. 214 - 225, SUG, June 1994.

[33] UNIX Software Operations, *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, 1990.

[34] W. R. Stevens, *TCP/IP Illustrated, Volume* 2. Reading, Massachusetts: Addison Wesley, 1993.

[35] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.