《 UNIX 高级教程系统技术内幕 》
(OCR by djq@SMTH ftp://166.111.136.159/incoming/UNIX_Book)

第 3 章 线程和轻量级进程

3.1 简 介

传统上的进程模型有两个严重的局限性。首先，许多应用程序都想并发地执行那些彼此间独立的任务，但是它们必须要共享一个公共的地址空间和其他的资源，这类应用的例子包括服务器上的数据库管理服务器，事务处理监测程序(monitors)，以及中间层和高层的网络协议。这些进程本质上是并行的，所以需要支持并行的编程模型。传统的 UNIX 系统强行的把这些应用中独立的任务串行化或者是设计一些糟糕的,而且效率很低的机制来管理这些操作。

其次，传统的进程不能很好地利用多处理器体系结构。因为一个进程在某个时刻只能使用一个处理器。一个应用必须创建许多个进程，并把它们分配到所有那些可用的处理器上去执行。这些进程应该找到一种方法来共享内存和其他资源，并且实现互相间的同步。

现在的一些 UNIX 变体系统是通过在操作系统中提供大量的原语来支持并发处理,从而解决这些问题的。由于缺少标准的术语，使得描述和比较各式各样的进程间的并行机制变得很困难。每一个 UNIX 变体都有它自己的术语系统，包括内核线程，用户线程，内核支持的用户线程，C-threads 和 Pthreads 以及轻量级进程等。本章解释这些术语，阐述基本的抽象概念，并且描述一些重要的 UNIX 变体中所提出的方法，最后，评价这些机制的优缺点。首先我们从线程的必要性和优点开始。

3.1.1 动机

许多程序必须执行一些独立的不需要串行化的任务。比如，一个数据库服务器应该能监听和处理大量的客户请求。因为这些请求不需要按照一个待定的顺序来得到服务，所以它们可以被当作独立的执行体来看待，原则上它们是可以并行运行的。如果系统提供了子任务可以并发执行的机制，那么这些应用程序可以执行得更好。

在传统的 UNIX 系统中，这些程序使用多个进程。许多关键的服务器应用程序有一个监听进程在不停地运行，等待客户请求到来。当一个请求到达时，这个监听进程创建(fork)一个新的进程为这个请求服务。因为对请求进行服务经常包括一些 I/O 操作，它可能阻塞进程。这种方法甚至在单处理器的系统中也能获得一些并发性。

其次，让我们考虑一个特定的应用程序，计算一个数组中不同项的值，项之间互相独立。这个应用程序能够为数组中每一个元素创建一个不同进程，并把每个进程分配到一个不同的计算机上去执行，或者在多处理器系统中分配到不同的 CPU 上执行而获得真正的并行性。甚至在一个单处理器机器的系统中，也需要把一个住务分解成多个进程。如果一个进程由于 I/O 操作或是发生页错误而必须阻塞时，另一个进程可以被调度向前执行。另一个例子是，UNIX 系统中的 make 工具允许用户并行的编译几个文件，每个文件都使用一个独立的进程。

在一个应用程序中使用多个进程有着一些明显的缺点。创建这些进程增加了一些基本的开销，因为 fork 是一个花销很大的系统调用(甚至在支持写拷贝共享地址空间的系统中)。因为每个进程都有它自己的地址空间，它必须使用进程问通信的手段如消息传递或者共享内存。要把这些进程分配到不同的机器或处理器上去运行，及在进程之间传递信息，等待进程的完成，收集结果等都需要额外的开销。最后，UNIX 系统中没有对共享某些资源给出合适的框架，例如网络连接。这种模型只有在并发带来的好处超过创建和管理多个进程所付出的代价时才是可行的。

这些例子都说明了进程抽象概念的不充分之处，以及我们迫切需要一种能很好的适合于并行计算的方法。我们现在能够建立一个独立的计算单元的概念模型，这些计算单元是一个应用程序全部处理工作的一个部分。这些单元之间的交互相对是很少的，因此需要很少的同

步。一个应用程序可能含一个或多个这种的单元。线程这种抽象概念就代表了一个单独的计算单元。传统的 UNIX 进程是单线程的,这意味着所有的计算都被串行化在同一个单元之中了。

本章中介绍的机制解决了进程模型所存在的局限性。同样这种机制也有它自身的缺点，我们将在本章的后面部分进行讨论。

3.l.2 多线程和多处理器

对干并行的计算密集型(computing-bound)的应用,当多线程系统与多处理器体系结构结合起来时，体现出来的优点十分明显。通过在不同的处理器上执行每一个线程，一个应用可以获得真正的并行性。如果线程的数目大于处理器的数目，那么就需要这些线程复用那些可用的处理器①。理想状况下，一个应用程序的 n 个线程在 n 个处理器上运行，并且完成工作的时间是单线程方式程序所需时间的 1/n。实际上，创建、管理和同步线程的花销以及多处理器操作系统的开销使之大大地低于理想的比率。

如图 3-1 所示，一组单线程的进程在一个单处理器机系统上执行的情况。系统分配给每一个进程一小段时间(时间片)去运行，当时间片用完后系统切换到下一个进程去运行。利用这种方法可以模拟并发的实现。在这个例子中，开始的时候，三个进程在一个客户服务器应用的服务器上运行。服务器为每一个活动的客户创建一个新的进程。这些进程拥有几乎相同的地址空间，并且通过使用进程间通信机制来共享信息。而在其下面的两个进程执行另一个服务器应用。

图 3-2 指示了在多线程系统中两个服务器的执行情况。每个服务器以一个单独进程的形式运行，多个线程共享一个地址空间。线程间上下文切换是由内核线程库或是用户级线程库来处理，选择哪一种方式取决于操作系统。对于两种情况，这个例子都显示出线程的一些优点，对每一个应用程序，消除多个近于相同的地址空间就会减轻内存管理子系统的负担(甚至现在使用写拷贝共享的系统也必须要为每一个进程管理独立的地址变换映射),因为一个应用程序的所有线程共享一个公共的地址空间，它们可以使用效率很高的同时开销很小的线程间通信和同步机制。

这种方法的优点是很明显的。一个单线程的进程并不用保护它自己的数据来防止其他进程的访问，多线程进程就要必须考虑在它地址空间中的每一个对象。如果多于一个线程能一

图 3-1 传统 UNIX 系统——使用单线程进程的单处理机系统

图 3-2 单处理机系统中的多线程进程

访问这个对象，它们必须使用一些同步方法来防止数据被破坏。

如图 3-3 所示，两个多线程的进程在一个多处理器机上运行。每一个进程的所有线程都共享相同的地址空间，但是它们的每一个线程运行在不同的处理器上。因此它们都是并发执行的。这显著地改善了系统的性能，但是也使同步问题变得很复杂。

图 3-3 多处理机系统中的多线程进程

尽管这两种方法结合得很好，但是它们当中的单独一个也可以很好的运行。一个多处理器的系统对单线程的应用程序同样也是十分有用的，因为几个进程可以并行地执行。同样地，多线程应用程序的优点也十分显著，甚至在单处理器的机器上。当一个线程必须在 I/O 操作或其他资源上阻塞的时候，另一个线程能够被调度去运行，这样应用程序就可以继续向前执行。线程这个抽象概念代表一个程序的本质上的并发性，比用来把软件设计映射到多处理器硬件体系结构上更适合。

3.1，3 并发和并行

为了理解不同类型的线程抽象概念，我们必须首先分清并发和并行[Blac 90]。一个多处理器应用程序的并行是指达到并行执行的实际程度，因此它受到应用程序可以使用的物理处理器的数目的限制。应用程序的并发是指应用程序在无处理器数目限制的情况所能达到的最大并行性。它取决了应用程序是怎样编写的，反映出在正确的资源可用的情况下，有多少个

线程可以同时地执行。

并发性可以在系统级或应用程序级上提供。内核提供的系统并发性是通过识别一个进程中的多个线程(也叫做主动线程)并独立地调度它们。接下来这些线程复用那些可用的处理器。如果一个线程阻塞在一个事件或资源上，内核能够调度另一个线程，因此一个应用程序甚至在单处理器的机器上也能得到系统并发性带来的优点。

应用程序也可以通过用户级的线程库提供并发性。这类用户线程，或叫合作例程(也叫做被动线程)是不能被内核识别的，它们必须由应用程序本身来调度和管理它们。因为如果这些线程不能在实际上并行执行，将不能提供真正的并发性或并行性。但是它却提供了对于并发应用的一个更加自然的编程模型。通过使用不阻塞(non-blocked)的系统调用，一个应用程序能够同时维护几个运行的进程间的交互、用户线程通过从每个线程的局部变量(在那个线程的堆栈中)而不是从全局的状态表中获得这些交互状态来简化编程。

每一个并发模型都有自己的极限值。线程用来作为组织工具和利用多处理器。内核线程工具允许在多处理器上并行执行，但是它不适合构造用户级的应用程序。比如，一个服务器应用程序可能需要创建数以千计的线程，其中的每一个对应于一个客户。内核线程的运行要消耗宝贵的资源如物理内存(因为许多实现要求线程结构是驻留内存的)，因此这对于一个程序是没用好处的。相反地，一个纯粹的用户级工具只对构造应用程序有用，然而不允许代码的并行执行。

许多系统实现一个二元并发模型，它把系统并发和用户并发结合在一起。内核识别出一个进程中的多个线程，而线程库增加了一些内核不可见的用户线程。用户线程允许在一个程序中并发例程之间的同步，而不需要使用系统调用，因此在多线程内核的系统中，用户线程也是需要的。进一步地说，减少内核大小和内核要完成的责任总是一个好主意，把线程支持功能分为内核部分和线程库是和这种策略一致的。

3.2 基本抽象概念

一个进程是一个复合的实体，可以分为两个部分——一线程的集合和资源集合。线程是一个动态的对象，它表示进程中的一个控制点，并且执行一系列的指令。资源包括地址空间，打开的文件，用户凭证和配额等等。这些资源为进程中所有线程所共享。此外每一个线程有它自己私有对象，比如程序计数器，堆栈和寄存器上下文。传统的 UNIX 进程有一个单独的控制线程。在多线程系统中对这个概念进行了扩展，允许在一个进程中有多于一个的控制线程。

在进程抽象概念中，把资源的所有权集中化有一些缺点。考虑一个服务器应用程序代替远端的客户执行文件操作。为了保证和文件访问权限的一致性、服务器在对请求进行服务时需要假定客户的身份。为了做到这一点，服务器是用超级用户特权安装的，它可以调用 setuid，setgid 和 setqroups 来改变它的用户证书来暂时匹配客户的信息。让这个服务器以多线程形式构造可以增加并发性，但是同时也将引起安全问题。因为进程拥有一单独的证书集合，它每次只能扮作一个客户。因此，服务器必须强制地串行化(单线程)所有的系统调用去检查安全性。

有几种不同类型的线程，每个都有不同的属性和使用方法。在本节中我们讨论二种重要类型的线程——内核线程，轻量级进程和用户线程。

3.2.1 内核线程

一个内核线程不需要和一个用户进程联系起来。它的创建和撤销是由内核的内部需求决定的，用来负责执行一个指定的函数。它共享内核正文字段和全局数据，具有它自己的内核堆栈。它能够被单独地调度并能使用标准的内核同步机制，如 sleep()和 wakeup()等等。

内核线程对于执行异步 I/O 操作非常有用。内核可以简单的创建一个新的线程来处理这种请求，而不是提供一种特殊的机制，这些请求被这些线程同步处理，而对于内核的其他部分出现异步。内核线程也用来处理中断，在 3.6.5 小节中将会详细地讨论。

内核线程的创建和使用开销并不是很大。它们使用的唯一资源就是内核堆栈和在它们不运行时用来保存寄存器上下文的一个区域，(我们也需一些数据结构来保存调度和同步的信息)。内核线程间的上下文切换是很快的，因为内存映射不用重新刷新。

内核线程不是一个新的概念。系统进程如页面管理进程(pagedaemon)在传统的 UNIX 内核中，它在功能上等同于内核线程。守护进程如 nfsd(网络文件系统服务器进程)在用户级启动，但是一旦启动，就完全在内核中运行。当它们进入到内核态后，用户态上下文就不需要了。它们也等同于内核线程。因为在传统的系统中缺少一种单独的抽象概念来代表内核线程，这些进程和那种传统进程所有的不必要的“垃圾”——proc 结构和 user 结构等纠缠在一起。多线程内核允许这些守护进程作为内核线程来简单实现。

3.2.2 轻量级进程

一个轻量级进程(LWP)是一个内核支持的用户线程。它是一个在内核线程基础上的高层抽象，因此内核在支持轻量级线程之前必须支持内核线程。每个进程可能有一个或多个轻量级进程，每一个都由一个单独的内核线程来支持(图 3-4)。轻量级进程被独立调度并且共享地址空间和进程中的其他资源。它们可以对 I/O 或其他资源进行系统调用，同时也能在 I/O 操作或资源访问的时候被阻塞。在一个多处理器的系统中，一个进程能真正享受并行所带来的好处，是因为每一个轻量级线程都能被分配到一个不同的处理器上运行。再者由于等待 I/O 操作完成或资源被阻塞的只是单个的轻量级进程而不是整个进程，在单处理器环境中其优势也很明显。

图 3-4 轻量级进程

除了内核堆栈和寄存器，轻量级进程也需维护一些用户状态。这主要包括用户寄存器上下文，当轻量级进程被抢占时这些内容必须被保存、尽管每一个轻量级进程都和一个内核线程相联系，但是一些内核线程是专门处理系统任务而不支持轻量级进程。

当每一个线程是相当独立的，并且不经常与其他线程交互时，多线程的进程是十分有利的。用户代码是可以完全被抢占的，所有的轻量级进程在一个进程中共享一个公共的地址空间，如果任何数据可以被多个轻量级并发地访问，这种访问必须被同步以保证数据的一致性。因此内核需要提供一些方法来锁定共享变量，如果一个轻量级进程想访问一个被锁住的数据时它将被阻塞。这些加锁的方法，如互斥(mutex)锁，信号量和条件变量，将在第 7 章中深入讨论。

能够认识到轻量级进程的局限性是很重要的，大部分的轻量级进程的操作，如创建，释放和同步都需要系统调用。系统调用相对来说是开销很大的操作，因为每个系统调用都需要在两个模式间切换，一个是在调用执行时，从用户态到内核态，另一个是在调用完成时返回到用户态。在每一个模式切换时，轻量级进程都要跨越一个保护边界。内核必须把系统调用的参数从用户空间拷贝到内核空间，并且校验它们来防止恶意进程的破坏。同样地，在从系统调用返回到用户态时，内核一定要把数据拷贝回用户空间。

当轻量级进程频繁访问共享数据时，同步的开销可能会抵销任何性能上的优点。在多数的多处理机系统中提供了锁机制，如果锁没有被其他线程占有，一个线程就可以在用户级上获得锁[Mule 93]。如果一个线程企图拥有一个当前不可用的资源时，它可能执行一个忙等待(busy-wait)(在资源释放之前一直空循环)，而不需要内核的参与。忙等待方法对于那些被暂时占有的资源还是可行的，然而在其他的情况下，必须阻塞这个线程。阻塞一个线程的操作是需要内核参与的，因此开销极大。

每一个轻量级进程都要花费相当多的内核资源，包括内核堆栈的物理内存。因此一个系统不能支持大量的轻量级进程。进一步地讲，因为系统有一个单独的轻量级进程的实现，它必须要通用，能支持大多数合理的应用。它因此也必须负担许多应用不需要的垃圾。轻量级进程对于使用大量线程，或是那些经常创建和撤销线程的应用来说是不适合的。最后轻量级

进程必须由内核来调度。对于那些必须经常把控制从一个线程转移到另一个线程的应用，如果它们使用轻量级进程，那么就不是很容易实现。轻量级进程也带来一些公平问题——一个用户能够通过创建大量的轻量级进程而垄断处理器的使用。

总之，尽管内核提供了创建同步和管理轻量级进程的机制，但是需要开发人员能够合理地使用这些机制。许多应用程序通过使用用户级的线程方法能很好地完成，这些方法我们将在下一节讨论。

注意；术语轻量级进程是从 SVR4/MP 和 Solaris 2.x 的命名系统中借用的，有时它会引起混淆，因为 SunOS 4.X[Kepe 85]使用术语“轻量级进程”是指用户级的线程(在下节中有介绍)。但是，在本书中，我们一致地使用“轻量级进程”是指内核支持的用户级线程。在一些系统中还使用了术语“虚拟处理机”，它本质上是和轻量级进程相同的。

3.2.3 用户线程

用户线程是完全在用户级上提供的抽象概念，而无需内核知道它们的存在，这通过使用像 Mach 的 C-threads 和 POSIX 的 pthreads 等线程库程序包可以成功的实现。这些库提供所有的创建、同步、调度和管理线程的函数，而不用内核的特殊帮助。线程间的交互不包含内核的参与，因此速度非常快②。图 3-5(a)描述了这种状态。

图 3-5(b)把用户线程和轻量级进程结合起来，创建了一个非常强大的编程环境。内核来识别、调度和管理轻量级进程。一个用户级的库复用在轻量级进程上的用户线程，并且提供线程的调度、上下文切换和同步等方法，而不涉及到内核。实际上，库充当它所控制的线程的微内核。

用户线程的实现是可能的，因为用户级线程的上下文可以在没有内核下顶的情况下被保存和恢复。每一个用户线程有它自已的用户堆栈，一块用来保存用户级寄存器上下文以及例如信号屏蔽等状态信息的内存区域。库通过保存当前线程的堆栈和寄存器内容，载入新调度线程的那些内容来实现用户线程间的调度和上下文切换。

内核仍然负责进程的切换，因为只有内核具有修改内存管理寄存器的权力。用户线程不是真正地可以调度的实体，内核没有保留它们的一点信息，内核只是简单地调度它们下面的进程或轻量级进程，这些进程再使用库函数来调度它们的线程。当进程或轻量级进程被抢占时，它们的线程也被抢占。同样如果一个用户线程发生了一个阻塞的系统调用，它也会阻塞它下面的轻量级进程。如果进程只有一个轻量级进程(或者如果用户线程是在一个单线程的系统中实现的)，那么它所有的线程都要被阻塞。

库也提供同步对象来保护共享的数据结构。这样的一个对象通常由一种类型的锁变量(如信号量)和阻塞在其上的线程队列组成。线程在访问数据结构之前必须获得锁。如果这

图 3-5 用户级线程实现

个对象已经被锁住了，库就会通过把这个线程链接到它的阻塞队列中来阻塞这个线程、同时，库把控制权传递给另一个线程。

现代 UNIX 系统提供了异步 I/O 机制，它允许进程执行 I/O 操作而不用阻塞。举个例子，SVR4 系统对任何流设备提供一个 IO_SETSIG ioctl 操作。(在第 17 章中将介绍流)。随后的对流的读或者写操作就简单地放到队列中而无阻塞返回。当 I/O 操作完成后，这个进程就会被一个 SIGPOLL 信号通知。

异步 I/O 是一个非常有用的手段，因为它允许一个进程在等待 I/O 操作完成的同时执行其他任务。但是，它带来了一个非常复杂的编程模型。人们希望把异步限制到操作系统级上，而给应用程序一个同步的编程环境。线程库通过提供一个同步接口来做到这一点，而在内部使用异步机制。和调用线程有关的每一个请求都是同步的，这个线程在 I/O 操作完成前都一直被阻塞，但是作为进程来说仍然是向前运行发展的，因为库引起了异步操作，同时也调度另一个用户线程去运行，当 I/O 操作完成时，库重新调度被阻塞的线程。

用户线程有几个明显的优点，它以一个非常自然的方式来对向 Windows 那样的应用进行编程，用户线程也通过把异步操作的复杂性隐藏在线程库中来提供同步编程规范。这就足以说明用户线程是十分有用的，甚至在那种缺少任何支持线程的内核的系统中，一个系统可以提供几个线程库，每一个部适用一个不同类的应用。

用户线程的最大优点就是性能，用广线程是轻量级的，只有当它们附着在轻量级线程上时才消耗内核资源，它们这种性能的获得是由于功能的实现是在用户级上，而不是使用系统调用，这样就避免了陷入处理和跨越保护边界时移动参数和数据的开销。一个非常有用的概念是关键线程大小[Bita 95]，它指的是一个线程用作一个单独实体时必须做的工作量，这个大小取决于创建和使用一个线程的开销。对于用户线程，它的重要部分的大小一般在几百条指令左右，通过编译器的辅助可以减少到少于 100 条。用户线程需要非常少的时间去创建，撤销和同步。表 3-1 比较了在 SPARC 工作站 2 上[Sun 93]内核进程、轻量级进程和用户线程的不同操作的所表现的执行时间。

表 3-1 在 SPARC 工作站 2 上的用户线程、LWP 和进程操作的执行时

	产生时间(miu s)	使用信号量的同步时间(miu s)
用户线程	52	66
LWP	350	390
进程	1700	200

另一方面，用户线程也存在一些局限性，这主要是由于内核和线程库之间信息的完全分离造成的。因为内核不知道用户线程的情况，这样它就不能使用它的保护机制来保护用户线程不被其他的线程破坏。每一个进程拥有自己的地址空间，内核保护它们防止其他非授权的进程的访问。用户线程就不享受这种保护，它们在公共地址空间中操作。从线程互相操作的要求出发，线程库必须提供同步的方法。

这种分开的调度模型带来了许多其他的问题，线程库调度用户线程，内核调度底层的进程或轻量级进程，两者都不需要知道对方在做些什么，比如，内核可能抢占一个轻量级进程，而此时它的用户线程拥有一个自旋锁(spin lock)，如果另一个轻量级进程上的用户线程想得到这个锁，它将一直忙等待直到锁的拥有者恢复运行。同样地，因为内核不知道用户线程的相对优先级，它可能抢占一个运行高优先级用户线程的轻量级进程，而去调度运行低优先级用户线程的轻量级进程。

在一些情况下，用户级的同步机制可能运行得不正常。所有的可执行线程最终都能被调度，许多应用程序都是在这个假设的基础上编写的。当每一个用户线程都绑定在一个单独的轻量级进程时，该假设是正确的，但当用户线程复用少量的轻量级进程时，该假设可能不成立。因为当轻量级线程的用户线程在某个系统调用上阻塞的时候，它可能也阻塞在内核，这使得一个进程中没有可运行的轻量级进程，这种情况甚至在存在可运行的线程和可用的处理器时也可能发生。异步 I/O 机制有助于减少这类问题的发生。

最后，由于没有内核显式地支持，用户线程可能改善其并发性，但是不能增加其并行性，甚至在一个多处理器的系统中，一个轻量级进程也是不能并行执行的。

本节解释了三个经常使用的用户线程抽象概念，内核线程是基本对象，它对应用程序是不可见的。轻量级进程是用户可见的线程，它们在内核线程的基础上是可以为内核识别的。用户线程是高层的对象，它对内核是不可见的。如果系统支持，用户线程可以使用轻量级进程，或者它们可以在标准的 UNIX 进程中实现而不需要内核的支持。轻量级进程和用户线程都有主要的缺点而限制了它们的可用性。3.5 节将介绍一个在调度器调用的基础上的新的框架，它将解决存在的大多数问题。但是首先我们要仔细看看与轻量级进程及用户线程设计有关的事项。

3.3 轻量级进程设计--要考虑的问题

有几个因素影响轻量级进程的设计。最低限度对于单线程的情况下，最主要的是需要正确的保留 UNIX 语义。这意味着包含一个单独的轻量级进程的进程必须表现出和一个传统的 UNIX 进程一样的行为特性(再次指出轻量级进程指的是内核支持的用户线程，而不是 SunOS 4.0 中的轻量级进程，它是纯粹用户级对象)。

有几个方面的 UNIX 概念不能很容易的对应到一一个多线程的系统中。以下几节将讨论这些问题，并且给出可能的解决方法。

3.3.1 fork 的语义

在多线程环境中的系统调用有一些和通常不一样的含义。许多系统调用都必须要重新设计，这些调用包括处理进程创建、地址空间管理或者是每个进程对资源的操作(如打开文件)等等。有两个重要的指导原则。首先，在单线程的情况下，系统调用必须满足传统的 UNIX 语义。第二，多线程进程调用的时候，系统调用应该表现出一个很合理的方式，并且和单线程的语义非常相似。下面，我们看一些非常重要的，由干多线程设计而受影响的系统调用。

在传统的 UNIX 系统中，系统调用 fork 创建一个新的进程。这个新的进程几乎就是它的父进程的一个克隆。差别就是那些必须的用来区分父进程和子进程的信息。系统调用 fork 的语义对于单线程进程是非常清楚的，在多线程的情况下，就存在一个选择，是复制父进程的所有轻量级进程，还是复制调用 fork 的那一个。

假设系统调用 fork 只是复制调用的那个轻量级进程到新进程中，无疑这是非常有效的。对于子进程能够很快地通过调用 exec 执行另一个程序的情况也是一个很好的模型。这个接口存在几个问题[powe 91]。轻量级进程经常用来支持用户级线程库。这种线程库通过使用用户空间的数据结构代表每个用户线程。如果 fork 复制的只是调用的轻量级进程，新的进程将包含用户级线程，而不映射到任何轻量级进程上。更进一步地讲，当子进程想获得那些被其他进程中线程拥有的锁，将会发生死锁。因为线程库常常会创建一些编程者没有注意到的隐藏线程，这是无法避免的。

另一方面，假设 fork 复制父进程中所有的轻量级进程。在系统调用 fork 是用来“克隆”整个父进程而不是去执行另一个程序的情况下，这 种方法非常有用。但是它也存在许多问题。父进程中的一个轻量级进程可能在系统调用中被阻塞。那么在子进程中它的状态将无法定义。一种可能就是让这种调用返回一个状态码 EINTR(系统调用被中断)，允许轻量级进程在必要的时候重新执行。还有就是一个轻量级进程可能打开一个网络连接，如果在子进程中关闭这个网络连接，那么可能引起意外的消息被发送到远端节点。一些轻量级进程可能止在管理一个外部的共享数据结构，如果 fork“克隆”这个轻量级进程就可能造成这些共享数据结构的破坏。

出为两个方案都不能正确地处理所有的情况,许多系统通过提供两个变体的 fork 折衷方案。一种是复制整个进程，另一种只复制单个的线程。对于后一种情况，系统定义了一组安全函数。可能在子进程执行 exec 前调用这些函数。另一种解决方案就是允许进程注册一个或多个 fork 处理程序，它们是在父进程或子进程中执行，但是在 fork 前还是 fork 后执行是在注册过程中指定的。

3.3.2 其他的系统调用

在多线程的系统中，除了 fork 之外，许多其他的系统调用也必须被修改才能正确地工作。进程中所有的轻量级进程共享一组公共的文件描述符。如果一个轻量级进程关闭当前另一个轻量级进程读或写的丈件，那么将引起冲突。文件偏移指针(offset pointer)也是通过文件描述符进行共享的，因此由一个轻量级进程执行的 lseek 将会影响其他的轻量级进程的操作。图 3-6 解释了这个问题。轻量级进程 L1 想从文件中从偏移 OFF1 开始读数据，就调用 lseek，并且跟随一个读操作。假设在这两个调用之间，轻量级进程 L2 对同一个文件调用一个 lseek，指定一个不同的偏移。这将引起 L1 读到错误的数据。实际中应用程序可以直接

使用一些文件锁协议来解决这个问题。或者是内核能够提供机制利用原子地执行随机的 I/O 操作(3.6.6 小节)。

图 3-6 并发访问同一文件的间题

同样地，进程有一个单独的当前工作目录和一个单独的用户证书结构。因为用户证书可以在任意时间发生变化，内核必须能够原子地对他们取样，每个系统调用都需要一次。

进程的所有轻量级进程共享公共的地址空间,并且可能通过如 mmap 和 brk 等系统调用并发地来管理这个地址空间。这些系统调用必须在线程级上是安全的，从而保证它们不破坏地址空间。编程者必须小心地串行化这些操作，否则结果不可想象。

3.3.3 信号传递和处理

在 UNIX 系统中，信号的传递和处理是由进程完成的。然而在一个多线程的系统中，系统必须决定进程的那一个轻量级进程来处理这个信号，这种问题同样也发生在用户线程上。因为一旦内核传递一个信号到一个轻量级进程上，线程库能够直接把它传给指定的线程。这里有几种可能的对信号的处理：

1.把信号发送到每一个线程。

2.指定进程中的主线程接收所有的信号。

3.发送信号绘任意一个线程。

4.使用启发式方法来决定信号请求的是哪个线程。

5.创建一个新线程来处理每一个信号。

第 1 种情况开销极大，而且它和大多数止常的信号应用不兼容。然而，它在某些情况中是很有用的。比如，当一个用户在终端上按下了 Ctrl-Z，他可能希望挂起进程中所有的线程。第 2 种情况中会出现不公正对待线程的现象，这和许多的现代方法不兼容，而且同那些经常和多线程内核的对称多处理器系统不兼容。第 5 种方法只是在一些指定的情况下才可能有用。

剩下的两种可能方法的选择取决于信号产生的本质。一些信号，如 SIGSEGV(段错误)和 SIGILL(非法操作)是由线程本身引起的，把这类信号传递给引起它发生的线程是十分有意义的。其他信号，如 SIGTSTP(从终端产生的停止信号)和 SIGINT(中断信号)是由外部事件产生的，在逻辑上不能把它们同任何特定的线程联系起来。

另一个相关的问题是信号的处理和信号的屏蔽。所有的线程一定要共享一组公共的信号处理程序吗？或者它们中的每一个可以定义自己的信号处理程序吗？尽管后一种方法功能强大而且十分灵活，但是它给每一个线程增加了相当大的开销，这削弱了多线程进程的主要目的。同样，这种方法也不适合信号屏蔽。信号通常要屏蔽起来，来保护临界区的代码不被破坏。因此，每个线程应能指定自己的信号掩码。每个线程的信号掩码的开销是相对很低的，而且是能够接受的。

3.3.4 可视性

决定在多大程度上一个轻量级进程可以被外部进程所见是很重要的。勿庸置疑，内核可以看见轻量级进程，并且能够独立地调用它们。在大多数系统中都不允许进程知道或同另一个进程的指定轻量级进程交互。

但是，在一个进程内，轻量级进程之间需要互相知道。因此在许多系统中提供了一些特殊的系统调用，它允许轻量级进程向同一个进程中的其他指定的轻量级进程发送消息。

3.3.5 堆栈增长

当一个 UNIX 进程的堆栈溢出时，就会发生一个段违例错误。内核发现这个错误是发生在堆栈段，它就会自动地扩大这个堆栈③而不是向进程发信号。

一个多线程进程有几个堆栈，每个用户线程有一个。这些线程是由线程库在用户级分配的。因此内核再想扩大堆栈就不合理了，因为在用户线程库中的堆栈分配器会和内核的操作

发生冲突。

因此在一个多线程的系统中，内核不知道用户堆栈的使用情况④。系统中可能没有特定的堆栈区，用户可能从堆区分配数据区域的一部分作为堆栈。通常的情况下，一个线程能够指定它所需要的堆栈大小，线程库通过在堆栈顶分配一个写保护的页来防止堆栈溢出。如果堆栈溢出时，就会引起一个写保护错误，内核通过向合适的线程发送一个 SIGSEGV 信号来响应。扩展堆栈或是以另一种方式处理溢出就是线程的责任了⑤。

3.4 用户级线程库

设计用户级线程库必须要解决两个主要的问题--线程库呈现给用户的是什么样的编程接口，和它如何通过操作系统提供的原语来实现。目前已经有许多不同的线程库存在，如 Chorus[Arma 90]，Topaz[Vand 88]以及 Mach 的 C-threads [Coop 90」。近来，P1003.4aIEEE POSIX 标准组织实现了几个线程库的初步模型叫做 pthreads[IEEE 94」。现代的 UNIX 版本努力支持 pthreads 接口来和这个标准兼容(见 3.8.3 小节)。

3.4.1 编程接口

一个线程库提供的接口必须包括几个重要的方面。它必须能提供在线程上的大量操作，如：

- 创建和中止线程
- 挂起和恢复执行线程
- 对单个线程分配优先级
- 线程调度和上下文切换
- 通过如信号鲑和互斥锁等工具完成的同步活动
- 线程间发送消息

因为在用户态和内核态之间来回切换的开销很大，线程库应该尽量减少内核的参与。因此线程库应尽可能多的提供工具。通常内核没有用户线程的有关信息，但是线程库能使用系统调用来实现它的一些功能。这是有重要意义的，比如，线程优先级和内核调度优先级没有什么关系，内核调度优先级是分配给底层的进程或轻量级进程的。而线程优先级就是一个进程的相对优先级，被线程调度器使用来调度线程的运行。

3.4.2 线程库的实现

线程库的实现依赖于内核为多线程提供的工具。许多线程库是在传统的 UNIX 的内核上实现的，传统的 UNIX 内核对线程不提供特殊的支持，线程库起一个微内核的作用，维护每一个线程的状态信息，并在用户级处理所有的线程操作。实际上，线程库串行化了所有的处理，并且在一定程度上通过使用系统的异步 I/O 手段实现了并发性。

现代许多系统中，内核通过使用轻量级进程支持多线程进程。在这种情况下，用户线程库有几种可选择的实现方法；

- 让每一个用户线程绑定在不同的轻量级进程上。但是这要使用更多的内核资源，和提供更少的附加值。它要求内核参与所有的线程同步和线程调度。
- 让用户线程复用少量的轻量级进程集合。这种方法消耗非常少的内核资源，因此非常有效、进程中所有的线程大略等同时，这种方法工作得尤其好。但是这种方法不能提供一种容易的方式保证资源能够分配到特定的线程。
- 允许在进程中混合着有界线程和无界线程。这样应用程序能充分地利用系统的并发性和并行性。通过增加底层的轻量级进程的调度优先级或是给底层的轻量级进程对处理器的独占权来优先处理有界线程。

线程库中包含调度算法来选择用户线程运行。它维护每个线程的状态和优先级，这些信息和底层轻量级进程的状态及优先级没有关系。考查图 3-7 中所示的例子，6 个用户线程复

用 2 个轻量级进程，线程库调度一个线程到轻量级进程上运行。这些线程(u5 和 u6)都处在可执行状态，但是它们底层的轻量级进程可能在系统调用中被阻塞，或是被抢占而等待被调度。

图 3-7 用户线程状态

当一个线程(在图 3-7 中如 u1 或 u2)想获得一个被其他线程占有的同步对象时，它将进入到阻塞状态。当这个锁被释放时，线程库解除对这个线程的阻塞，并把它放入调度队列。线程调度器依据优先级和 LWP 机构从队列中选择一个线程。这种机制和内核的资源等待调度算法十分相似。线程库充当它管理的线程的一个微内核。

[Doep 87]，[Muel 93]和[Powe 91]中更详细的讨论了用户线程。

3.5 调度器调用

在 3.3 节和 3.4 节中描述了轻量级进程和用户线程的优缺点。两个模型都不是十全十美的。应用程序的开发人员希望能同时得到性能优点和用户线程的灵活性。可是用户线程缺少与内核的集成，它不能完成轻量级进程的功能。[Ande 91]描述了一个新的线程体系结构，这种结构结合了两种模型的优点。操作系统研究机构接受了这种框架结构，并且一些供应商如 SGI 等采用这种结构来实现商用线程[Bita 95]。

这种结构的基本原则就是用户线程与内核紧密结合起来。内核负责处理器分配，线程库负责调度，线程库把那些影响处理器分配的事件通知给内核。它既可以请求额外的处理器，也可以放弃对处理器的拥有。内核完全控制处理器的分配，它可以任意地抢占一个处理器并把它分配给其他的进程。

当给线程库分配一些处理器时，它就完全控制哪个线程能被调度到处理器上运行。如果内核取走了一个处理器，线程库会得到通知，它会重新正确地分配线程在处理器上的运行。如果线程在内核中阻塞，进程并不失去处理器。内核通知线程库，线程库就会立即调度另外的用户线程在处理器上运行。

这种方法的实现需要两个新的抽象概念——一个是向上调用(upcall)，另一个是调度器调用。向上调用是指内核对线程库的调用。调度器调用是指可以用来执行用户线程的可执行的上下文。它和轻量级进程相似，并且有自己的内核和用户堆栈。内核做向上调用时，它把一个调用传给线程库，线程库使用这个调用来处理这个事件，运行一个新的线程或是唤起另一个系统调用。内核并不按时间分割在一个处理器上的调用。住何时刻，一个进程对于分配给它的处理器只有一个调用。

调度器调用框架结构的一个显著的特点就是它对阻塞的处理。当一个用户线程在内核中阻塞时，内核创建一个新的调用和对线程库的一个新的向上调用。线程库在旧的调用中保存线程的状态，并且通知内核旧的调用还可以重新使用。接下来，线程库调度在新的调用上的另一个用户线程。当阻塞操作结束时，内核作为一个向上调用来把这个事件通知给线程库。这个向上调用需要一个新的调用。内核可以分配一个新的处理器来运行这个调用或者是抢占这个进程的当前调用。后一种情况下，向上调用通知线程库两个事件：第一，最初的线程可以恢复执行。第二，在那个处理器上运行的线程已被抢占。线程库把这两个线程都放入就绪队列，接下来决定首先调度哪一个。

调度器调用有许多优点,因为许多操作都不需要内核的参与。它运行的速度特别快,[Ande 91]中描述的 Benchmarks 示出了在调用基础上的线程库执行情况和使用其他用户线程库的对比。因为内核把阻塞线程和抢占处理器等事件通知给线程库，线程库就可以做十分灵活的调度和同步，从而避免死锁的发生和不正确的语义。比如，内核占据一个处理器，它的当前线程持有一个自旋锁(spin lock)。线程库可切换这个线程到另一个处理器上，在释放自旋锁之前这个线程将一直在那里运行。

本章的余下部分将介绍在 Solaris，SVR4，Mach 以及 Digital UNIX 系统中线程的实现。

3.6 Solaris 和 SVR4 的多线程处理

Sun Microsystem 在 Solaris 2.x 中引入了内核级支持的线程。UNIX 系统实验室在 SVR4.2/MP 中采纳了 Solaris 的线程设计。这种体系结构在内核和用户级提供了各式各样大量的原语，允许开发功能强大的应用程序。

Scans 支持内核线程、轻量级进程和用户线程。一个用户进程可能有几百个线程，真实地映射了程序本质上的并行性。线程库的多个线程将会复用少量的轻量级进程。用户可以控制轻量级进程的数目，从而能最高效率的利用系统资源，也可以把一些线程绑定到单个轻量级进程上(见 3.6.3 小节)。

3.6.1 内核线程

在 Solaris 中内核线程是一个基础的轻量级的对象，它可以独立地被调度和分配到系统的处理器上去运行。内核线程不用和进程联系起来，它可以通过内核执行特定的函数来创建、运行和释放。这样，内核就不用在内核线程切换时重新映射地址空间[Kepp 91]。因此，内核线程的上下文切换要比一个进程的上下文切换花费少得多。

内核线程使用的资源仅仅是一个小的数据结构和一个堆栈。内核线程的数据结构包含下列信息：

- 保存内核寄存器的拷贝值。
- 优先级和调度信息。
- 指向把线程放人调度器队列的指针，或者是线程阻塞时，把线程放入资源等待队列的指针。
- 堆栈指针。
- 指向相关的 LWP 和 proc 结构的指针(如果线程没有绑定到一个轻量级进程上，指针为空)。
- 维护一个进程中所有线程队列的指针和维护系统中所有线程队列的指针。
- 如果有相关的轻量级进程，那么还包括相关轻量级进程的信息(见 3.6.2 小节)。

Solaris 的内核是内核线程的集合。这些内核线程中的一些运行轻量级进程，另外一些执行内部的内核函数。内核线程是可以完全被抢占的，它们可以属于系统中的任何一个调度类(见 5.5 节)，包括实时类。它们使用特殊版本的同步原语(信号量，条件变量等)来防止优先级逆转。低优先级的线程锁住了高优先级线程需要的资源，因此阻止了高优先级线程的向前执行。这些问题将在 5.6 节中介绍。

内核线程用来处理异步事件，如磁盘延迟写操作，流服务过程和调出队列(见 5.2.1 小节)。内核把每一个事件同一个优先级联系起来(通过设定线程的优先级)，这样内核就能够恰当的调度它们。内核线程也用来支持轻量级进程，每一个轻量级进程都连着一个内核线程(尽管不是所有的内核线程有一个轻量级进程)。

3.6.2 轻量级进程的实现

轻量级进程在一个单独的进程中提供多线程控制。这些轻量级进程被独立地调度，可以并行在多处理器上运行。每一个轻量级进程都被绑定到它自己的内核线程上，而且在它的整个生命期内这种绑定都有效。

传统的 proc 和 user 结构是不能充分地表示多线程进程的。在这些对象中的数据必须被分成每个进程的信息和多个轻量级进程的信息。Solaris 用 proc 结构保存所有进程的数据，包括在传统的 u 区的和进程相关的部分。

一个新的 LWP 结构保存上下文中的每个轻量级进程的部分信息。它包括以下的信息：

- 用户级寄存器保存的值(当轻量级进程下运行时)。

· 系统调用的参数、结果和错误代码。
· 信号处理信息。
· 资源使用情况和统计数据。
· 虚拟时间报警。
· 指向内核线程的指针。
· 指向 proc 结构的指什。

lwp 结构可以和轻量级进程一起被换出内存，因此那些一定不能换出的信息如信号掩码。必须要保存在相关的线程结构中。在 sparc 的实现中预留全局寄存器 %g7 来保存指向当前线程的指针。因此可以允许快速访问当前的轻量级进程和进程。

对轻量级进程(以及内核线程)可用的同步原语有互斥锁，条件变量，计数信号量和读写锁。这些方法将在第 7 章介绍。每一个同步工具能表现出不同的行为特性。比如，一个线程想获得被另一个线程占有的互斥锁，它可能忙等待或是阻塞等待互斥锁被释放。一个同步对象在初始化的时候，调用者必须指定他所希望的是哪一种行为。

进程中的所有轻量级进程共享一组公共的信号处理程序。但是每一个信号处理程序可能拥有自己的信号掩码，来决定忽略或阻塞哪一个信号。每一个轻量级进程也可以指定自己的替换堆栈来处理信号。信号被分成两类——陷入和中断。陷入是由轻量级进程本身产生的同步信号(如 SIGSEGR,SIGFPE 和 SIGSYS)。这些陷入信号总是传给产生信号的 LWP。中断信号(如 SIGSTOP 和 SIGINT)可以被传递到任意没有屏蔽信号的轻量级进程。

轻量级进程没有全局的名字空间，因此对于其他的进程是不可见的。进程不能直接发送信号到另一个进程中指定的轻量级进程，或者知道发给它消息的是哪一个轻量级进程。

用户线程是通过线程库实现的。它们可以在没有内核参与下创建、释放以及管理。线程库提供同步和调度的方法。这样进程可以使用大量的线程而不消耗内核资源，而且省去大量的系统调用的开销。尽管 Solaris 是在轻量级进程之上实现用户线程的。但是线程库隐藏了这些细节。大多数应用程序开发者仅仅和用户线程打交道。

缺省的情况下，线程库为进程创建一个轻量级进程池，并且在此之上复用所有的用户线程。这个池的大小取决于处理器的数目和用户线程的数目。应用程序可以对缺省情况重载并指定创建的轻量级进程的数目。它也可以要求系统为每个特定的线程指定一个轻量级进程。因此一个进程可能有两种类型的用户线程——绑定到轻量级进程上的线程和那些没有绑定的共享公共轻量级进程池的用户线程(见图 3-8)。

通过在少量的 LWP 上复用很多的线程可以以较低的代价实现并发。例如，在一个窗口系统中，每一个对象(窗口，对话框，菜单，图标等)都可以用一个线程代表。在某一时刻只有很少几个窗口是活动的，只有这些线程需要 LWP。LWP 的数目决定了应用程序可以获得的最大的并行度(假设至少我们有相同数目的处理器人这个数目还限制了进程在某时刻最多的阻塞操作。

有些时候线程比 LWP 多是不利的。例如，当计算两个二维数组的内积时，我们可以为结果数组的每一个元素分配一个线程。如果处理器的数目很少，这种方法可能就反而效率不高了，究其原因就是线程库花费了很多的时间来完成线程切换。如果为结果乘积数组的每一行分配一个线程，并将其与一个 LWP 绑定在一起可能会更有效。

在涉及时间关键性处理的应用中，同时包含绑定的和非绑定的线程是非常有用的。时间

图 3-8 Solaris 2.x 中的进程抽象表示

关键性处理可以绑定到分配厂实时调度优先级的 LWP 上完成，而其他线程负责其他低优先级的后台处理。在前面的那个窗口系统例子中，用实时线程来响应用户的鼠标移动事件，这样就可以立即在显示器上反映出鼠标的移动了。

因为线程库在线程间上下文切换时浪费了大量的时间。如果对乘积数组的每行创建一个

线程.并把每个线程绑定到一个轻量级进程上，效率就会大大地提高。

如果处理对时间要求很严格的话，应用程序中有绑定的和没绑定的用户线程是很有意义的。这些临界时间的处理可以由绑定到轻量级进程上的用户线程来处理，系统分配给它们实时的调度优先级，而其他的用户线程则负责低优先级的后台处理。在前面的窗口系统的例子中，一个实时的线程可以被分配去响应鼠标移动，因为这些鼠标移动必须在显示器上立即反映出来。

3.6.4 用户线程的实现

每一个用广线程必须维护以下的状态信息：

·线程 ID 它允许进程中的线程互相通过信号来通信等等。

·保有的寄存器状态 包括程序计数器和堆栈指针。

·用户堆栈 线程可能有由线程库分配的堆栈。内核不知道这些堆栈。

·信号掩码 每个线程有自己的信号掩码。当一个信号到来时，线程库可以通过检查线程的信号掩码把信号发送到合适的线程。

·优先级 用户线程使用相关进程的优先级，由线程调度器使用。内核并不知道这个优先级，因为它只是调度下面的轻量级进程。

·线程局部存储 线程允许有一些私有存储(由线程库来管理)，来支持 C 线程库接口的再版本[IEEE 94]。举个例子，许多 C 线程库例程返回一个错误代码到一个叫做 errno 的全局变量中。如果多个线程并发的调用这个例程将出现混乱的现象。为了避免这个问题的发生，多线程的线程库把 errno 放在线程的局部存储中[powe 91]。

线程使用线程库提供的同步方法，它们与内核中的同步方法相似(信号量，条件变量等)。通过简单地使用放在共享内存中同步变量，Solaris 允许在不同进程中的线程相互间同步。这些变更也可以放在文件中，通过使用 mmap 映射文件来访问这些变量。这样就允许同步对象的生命期超过创建它们的进程，并可以用它们来对不同进程中的线程进行同步。

3.6.5 中断处理

中断处理程序通常管理那些被内核其余部分共享的数据，这就需要内核同步对共享数据的访问。在传统的 UNIX 系统中，执行访问共享数据的操作时。通过提高中断优先级(ipl)来阻塞相关的中断，从而达到同步。通常地，被保护的对象是不可能被中断访问的。比如，在一个睡眠队列上的锁必须被保护不受中断，尽管大多数的中断不会访问那个队列。

这个模型有许多缺点。在许多系统中，升高或降低中断优先级是花费极高的，并且需要几个命令。中断是重要的而且紧急的事件。如果阻塞它们将会在很多方面降低系统的性能。多处理器系统中，这些问题变得更加严重。内核必须保留更多的对象，并且通常必须在所有处理器上中断阻塞。

Solaris 取代了传统的中断和同步模型，它采用一个新的实现[Eykh 92，Klei 95]旨在改善性能。待别是对于多处理器系统。首先，它并不使用中断优先级来保护中断，而是使用各种内核同步对象，如互斥锁和信号量。其次，它使用内核线程来处理中断，这些中断线程可以在高层创建，并且分配一个优先级，这个优先级高于所有其他类型的线程的优先级。它们使用和其他线程一样的同步原语，因此当它们需要的资源被其他线程占有时，它们也会阻塞。内核只是在非常少的异常情况下才阻塞中断，比如，当获得了保护睡眠队列的互斥锁。

尽管创建一个内核线程的开销是很小的，但是如果为每一个中断都创建一个新的线程，那么开销就太大了。内核维护一个中断线程池，它们是预分配的，而且部分是初始化的。缺省情况下，这个池为每个 CPU 的每个中断级别包含一个线程，加上一个为时钟处理用的系统范围的线程。因为每个线程需要大约 8K 字节空间来存储堆栈和线程数据，所以这个池使用了相当大的内存空间。在内存紧缺的系统中，最好能减少池中的线程数目，因为所有的中断不可能同时活动。

图 3.9 说明广在 Solaris 中中断的处理情形。当线程们接收到一个中断时，它在处理器 P1 上执行。中断处理程序首先提高中断优先级来防止同级或更低级的中断的发生(保留 UNIX 语义)。接下来它从池中分配一个中断线程 T2，并切换到线程 T2 上。T2 执行时 T1 在忙等待。也就是线程 T1 不能到另一个 CPU 上运行。线程 T2 执行完返回时，切换回线程 T1，它将恢复执行。

中断线程 T2 的运行不需要完全地初始化。这意味着它不是一个完全的线程，而且它不能被调度。仅当有理由去阻塞时这个线程完成初始化，这时，它保存自己的状态，变成一个独立的线程，而且能够在 CPU 上执行。如果线程 T2 发生阻塞，它把控制权还给线程 T1，这样 T1 就不用忙等待了。这种方式下，完全的线程初始化开销限制在中断线程必须阻塞的情况。以线程方式实现中断增加了一些开销(在 Sparc 上约 40 个指令)。然而另一方面，它避免了对每一个同步对象阻塞的中断，每次要节省 12 条指令。因为同步操作比中断发生频繁得多，只要中断阻塞不是太频繁，这种方法最终还是改善了系统的性能。

3.6.6 系统调用处理

在 Solaris 中，系统调用 fork 复制父进程的所有轻量级进程到子进程中。系统调用中的每一个轻量级进程都返回一个 EINTR 错误。Solaris 也提供一个新的系统调用 fork1，它

图 3-9 使用线程处理中断

fork 相似，区别是它只复制调用它的轻量级进程，当子进程期望在不久就执行一个新程序时，fork1 非常有用。

Solaris 2.3 解决了对文件并发执行任意 I/O 操作的问题(见 3.3.2 小节)。它增加了系统调用 pread 和 pwrite,，这些调用把寻址偏移量作为一个参数。不幸的是，Solaris 2.3 不提供等价的系统调用来代替 readv 和 writev，它们执行散-合 I/O 操作(见 8.2.5 小节)。

总之，Solaris 在这种两层模型中提供了一组编程接口。同时有用户线程和轻量级进程使我们能分清什么是编程人员能看见的，什么是操作系统提供的。程序开发人员可以只使用线程来写应用程序，之后通过管理底层的轻量级进程来优化程序，使之能提供应用所需的真正的并发性。

3.7 Mach 中的线程

Mach 从一开始就设计为多线程处理的操作系统。它支持内核中的线程和用户级线程库中的线程。它提供额外的机制来控制在多处理器系统中为线程分配处理器。Mach 在编程接口级上提供完全的 4.3BSD UNIX 语义，包括所有系统调用和库⑦。本节将要描述 Mach 中的线程的实现。3.8 节讨论在 Digital UNIX 中的 UNIX 接口，它是从 Mach 继承来的。3.9 节介绍一种新的机制叫做连续性，它是在 Mach 3.0 中引进的。

3.7.1 Mach 的抽象概念一任务和线程

Mach 内核中两个最基本的抽象概念是任务和线程[Teva 87」。任务是一个静态对象，它由地址空间和叫做端口权力的系统资源集合组成(见 6.4.1 小节)。任务本身不是一个可执行的实体，它仅仅是一个环境，一个或多个线程可以在其中运行。

线程是基本的执行单元，它在任务的上下文中运行。任务可以包含零个或多个线程，它们都共享任务的资源、每一个线程有一个内核堆栈，用来处理系统调用，它也有自己的计算状态(程序计数器，堆栈指针通用寄存器等等)，处理器可以独立地调度线程。属于用户任务的线程等价于轻量级进程，纯内核线程属于内核任务。

Mach 也支持处理器集合概念。在 5.7.1 小节中我们将做进一步的讨论。系统中的可用处理器可以被分成没有重叠的处理器集合。任务和线程能被分配到任何处理器集合(许多处理器集合的操作需要超级用户特权)。这样就允许多处理器中的一些 CPU 专门为一个或多个指定的任务服务，保证了高优先级任务的资源要求。

Mach 中用 task 结构表示任务，它包含下列信息：

·地址映射指针，它描述了任务的虚拟地址空间。

·任务中所有线程队列的头指什。

·任务分配到的处理器集合的指针。

·指向 utask 结构的指针(见 3.8.1 小节)。

·端口及其他有关 IPC 的信息(见 6.4 节)。

任务占有的资源为所有线程共享，每一个线程由一个 thread 结构来描述，它包含下列信息：

·把线程放人调度器或等待队列的链接。

·指向 task 结构的指针和它所属的处理器集合的指什。

·把线程放入相同任务中所有线程队列的链接，把线程放入相同处理器集合的所有线程队列的链接。

·指向保存它的寄存器上下文的进程控制块的指针。

·指向它的内核堆栈的指针。

·调度状态(可执行的，挂起，阻塞等等)。

·调度信息如优先级，调度策略和 CPU 使用情况等等。

·指向相关的 uthread 和 utask 结构的指针(见 3.8.1 小节)。

·特定线程的 IPC 信息(见 6.4.1 小节)。

任务和线程起着互补的作用。任务占有资源，包括地址空间。线程执行代码。传统的 UNIX 进程由一个包含单个线程的住务组成。多线程的进程由一个任务和若干线程组成。

Mach 提供一组系统调用来管理任务和线程。系统调用 task_create，task_terminate，task_suspend 和 task_resume 对任务操作。系统调用 thread_create，thread_terminate，thread_suspend 和 thread_resume 对线程操作。这些系统调用有着明显的含义。此外，thread_status 和 thread_mutate 允许读和修改线程的寄存器状态，系统调用 task_threads 返回任务中的所有线程队列。

3.7.2 Mach 的 C-threads

Mach 提供 C-threads 线程库，它提供一个易于使用的接口来创建和管理线程。举个例子，函数：

```
cthread_t  cthread_fork(void * ( * func )(), void * arg);
```

它创建一个新的线程并调用函数 func()。一个线程能调用：

```
void * cthread_join(cthread_t T);
```

来挂起自身一直到线程 T 中止。调用者接收 T 的高层函数返回值，或者是线程 T 显式地调用 cthread_exit()的状态码。

C-threads 线程库提供了互斥和条件变量等同步机制。它也提供了函数 cthread_yield()来让内核调度另一个线程去运行。这个函数仅对下面将要讨论的合作例程的实现是必需的。

有三种方法来实现 C-threads 线程库，应用可以选择最适合自己需求的一种：

·基于协同例程 在单线程任务(UNIX 进程)之上复用用户线程。这些线程是不可抢占的，只有在同步过程中,线程库才会切换到另一个线程(当前进程一定阻塞在一个互斥锁或信号量上)。此外，它靠线程调用 cthread_yield()来防止其他线程饿死。这种实现方法非常有利于调试，因为线程上下文切换的顺序是重复的。

·基于线程 每一个 C-thread 使用一个不同的 Mach 线程。这些线程是被抢占调度的，并且在多处理器上可以并行执行。这是一种缺省的实现方式。这种方法使用在 C-threads 程序的生产版本中。

·基于任务 每个 C-thread 使用一个 Mach 任务(UNIX 进程)，并且使用 Mach 的虚拟内存原语在与其他线程共享内存。这种方法只使用在需要特殊的共享内存语义的情况下。

3.8 Digital UNIX

Digital UNIX 操作系统，以前称为 DSF/1，是建立在 Mach 2.5 内核上的。从一个应用程序开发人员的角度看，它提供一个完全的 UNIX 编程接口。在内部，许多 UNIX 特征都是使用 Mach 原语来实现的，这项工作是基于 Mach 的 4.3BSD 兼容层，由开放软件组织(OSF)扩展，和 SVR3 及 SVR5 也兼容。这件方法对它的设计有深远影响。

Digital UNIX 提供了一组方法扩展了进程抽象概念[OSF 93]。多线程进程被内核和 Posix 兼容的线程库所支持。UNIX 的进程是在 Mach 的抽象概念任务和线程之上实现的。

3.8.1 UNIX 接口

尽管任务和线程充分地提供了 Mach 程序执行接口，但是它们不能充分地描述一个 UNIX 进程。进程提供的一些方法在 Mach 中没有对等的方法，如用户证书，打开文件描述符，信号处理和进程组。再进一步地讲，为了避免重新编写 UNIX 接口，代码是从 Mach 2.5 中 4.3 BSD 兼容层开始移植，而它又是从 4.3BSD 实现中移植来的。同样地，许多设备驱动程序是从 Digital 的 ULTRIX 移植来的，它也是基于 BSD 的。这些移植的代码对 proc 和 user 结构做了大量的引用，因此应该保留这些接口。

在最初的形式中保留 proc 和 user 结构存在着两个问题。首先它们的一些功能由 task 和 thread 结构提供。其次，这些结构不能充分代表多线程进程。比如，在传统的 u 区中包含进程控制块，它保存进程的寄存器上下文值。多线程的情况下，每个线程都有自己的寄存器上下文。因此一定要修改这些结构。

u 区被两个对象取代——单独的一个 utask 结构代表整个任务，和 uthread 结构代表任务中的每一个线程。这些结构不是保存在进程中的固定地址，同时也不随进程的换出而被换出。

utask 结构包含下列信息：

- 指向当前目录和根目录 v 节点的指针。
- 指向 proc 结构的指针。
- 信号处理程序数组和有关信号的字段。
- 打开文件描述符表。
- 缺省文件创建掩码(mask)。
- 资源使用情况，配额和统计信息。

如果线程打开一个文件，文件描述符被任务中所有的线程所共享。同样地，它们都有一个公共的当前工作目录。结构 uthread 描述了 UNIX 进程的每线程资源。它包括：

- 指向保存的用户级寄存器的指针。
- 路径名遍历域。
- 当前的和挂起的信号。
- 特定线程的信号处理函数。

为了减轻移植的困难，对老的 u 区字段的引用转换为对 utask 或者 uthread 结构字段的引用。实现转换的宏如下：

```
#define u_cmask utask->uu_cmask
#define u_pcb   uthread->uu_pcb
```

结构 proc 大部分保留下来，几乎没什么变化。但是它的大多数功能现在由 task 和 thread 结构来提供。这样，尽管由于历史原因被保留，proc 结构的许多字段没有被使用。比如，因为 Digital UNIX 单独调度线程，所以和调度及优先级有关的字段就不需要了。Digital UNIX 的 proc 结构包括以下信息：

- 把结构放到已分配的、僵尸或自由进程链表的链接。

·信号掩码。

·指向证书结构的指针。

·标识符和层次信息——PID，父进程 PID，指向父进程的指针，指向兄弟进程的指针，指向子进程的指针等等。

·进程组和会话信息。

·调度字段(未用)。

·当退出时保存状态和资源使用情况的字段。

·指向 task 和 utask 结构的指针，以及指向第一个线程的指针。

图 3-10 描述了 Mach 和 UNIX 数据结构之间的关系。结构 task 维护线程的链接的链表。结构 task 指向结构 utask，并且每个 thread 结构指向相应的 nthread 结构。proc 结构包括指向结构 task，utask 和第一个线程的指针。utask 结构向回指向 proc 结构，同时线程向回指向 task 和 utask 结构。这就允许快速访问所有结构。

图 3-10 Digital UNIX 的任务和线程数据结构

不是所有的线程都有用户上下文。一些线程可能是由内核直接创建来执行一些系统功能如页替换。这些线程和内核任务联系起来，它们没有用户地址空间。内核任务和线程没有相关的 utask，uthread 和 proc 结构。

3.8.2 系统调用和信号

在 Digital UNIX 中，系统调用 fork 创建新的进程有一个单独的线程，它是开始调用 fork 的线程的一个克隆。在 Digital UNIX 中没有可替换的调用能复制所有线程。

像在 Solaris 中一样，信号被划分为同步信号(或陷入)和异步信号(或中断)。陷入信号被传递到引起它的线程，中断信号被传递给任意线程。但是与 Solaris 不同的是，进程中的所有线程共享一组单独的信号掩码，它们存储在 Proc 结构中。允许每一线程对同步信号声明自己的一组处理程序，但是对于异步信号它们共享一组公共的处理程序。

3.8.3 Pthreads 线程库

pthreads 线程库对线程提供一个用户级 POSIX 兼容的编程接口。与 Mach 的系统调用相比较，这个编程接口要容易得多。线程库把每一个 Mach 线程同 pthreads 线程库联系起来。从功能上看，pthreads 线程库同 C-threads 线程库或其他线程库相似，但是它们实现了一个标准接口。

pthreads 线程库实现了由 POSIX 标准定义的异步 I/O 函数。比如，线程调用 POSIX 函数 aioread()时，线程库创建一个新的线程来同步地执行读操作。读操作完成时，内核唤醒被阻塞的线程，这个线程通过发送信号通知调用线程操作的完成。这个过程在图 3-11 中说明。

pthreads 线程库提供了完善的编程接口，包括信号处理和调度函数以及同步原底线程之间的同步可以在用户级实现，但是当一个轻量级进程需要阻塞时，必须要内核参与。

Digital 也提供了一个专门的 cma_threads 线程库，这个线程库提供一些额外特征[DEC 94]。使用这种线程库的程序可以在 Digital 的 VMS 和 Windows/NT 平台工作，但是不能在其他的 UNIX 系统上工作。

图 3-11 通过创建分离线程实现异步 I/O

3.9 Mach 3.0 的续体

尽管内核线程要比进程小一些，但是它要消耗大量的内核内存。这主要是为内核线程的堆栈、内核堆栈典型地要消耗至少 4K 字节的内存，它占一个线程使用的内核空间的 90%。一个有大量(数以百计的)线程的系统中，这种开销变得很大，而且降低了系统性能。一种解决办法就是在 Mach 线程或轻量级线程上复用用户线程,这样就避免了每个线程一个内核堆栈的

要求。但是这种方法有自身的缺点：用户线程不能被独立地调度，因此不能提供同级的并发性。进一步地讲，因为内核线程不能跨越保护边界，任务必须至少包含一个内核线程，产生在许多活动任务的系统中的一些问题。在本节我们将看 Mach 3.0 是怎样通过一个内核方法叫做续体(continuations)的来解决这些问题的。

3.9.1 编程模型

UNIX 系统使用进程模型来编写程序。每个线程有一个内核堆栈，线程在系统调用或发生异常而陷入到内核态时，就会使用内核准栈。线程阻塞在内核时，内核堆栈包含它的执行状态，其中包括它的调用顺序和自动变色这个方法的优点就是简单，因为内核线程不用显式的来保存任何状态。主要缺点就是使用了大量的内存。

一些操作系统如 Quicksilver [Hask 88]和 V [Cher 88]使用一个中断编程模型。内核把系统调用和异常当作中断，每个处理器有一个内核堆栈，对所有的内核操作都使用这个内核堆栈。因此，线程需要在内核中阻塞时，它首先必须显式地保存它的状态到某个地方。下一次运行时，内核使用这些保存的信息来恢复线程的状态。

中断模型的主要好处就是通过拥有一个单独的内核堆栈节省厂内存，主要的缺点就是线程对每次的阻塞操作必须显式地保存它的状态。这个模型难于编程，因为那些必须要保存的信息可能会跨过模块的边界，因此如果线程阻塞在职套很深的过程中，必须要判断在调用链中所有函数需要什么状态信息。

线程在什么条件下必须阻塞决定了哪一个模型更合适。线程在一个调用链的深处阻塞，使用进程模型更适合。然而，如果线程阻塞有非常少的状态需要保存，那么中断模型的效率是很高的。比如，许多服务器程序等待客户请求时在内核中多次阻塞。这样的程序没有很多的状态要在内核中保存，因而可以很容易的消除它的堆栈。

Mach 3.0 的续体方法结合了两种模型的优点，并且允许内核按照环境来选择阻塞方法。现在我们看一下它的设计和实现。

3.9.2 使用续体

Mach 使用函数 thread_block()函数来阻塞线程。Mach3.0 修改了这个函数使之能够接受多数，新的语法是：

```
Thread_block(void * (contfn)());
```

其中 contfn()是下次线程运行时要调用的续体函数、传递一个空指针参数表示需要传统的阻塞方式。这样，线程能够任意地选择使用续体。

线程使用续体时，它首先要保存那些在恢复执行时可能需要的状态。为了这个目的，结构 thread 中保留了一个 28 字节的区域、内核阻塞线程并且重新获得堆栈。线程重新恢复执行时，内核给它一个新的堆线，同时调用续体函数，这个函数恢复它以前保存的状态。为了完成这个过程，要求 continuaton 函数和调用函数对被保存了什么状态和在哪里保存的有清楚的了解。

下面的例子说明了续体的使用。例子 3-1 使用传统的方法来阻塞线程。

实例 3-1 没有使用续体来阻塞线程

```
syscall_l(argl)
{
    ...
    thread_block();
    f2(arg);
    return;
}
f2(arg)
```

```
    {
        ...
        return;
    }
```

例子 3-2 示出怎么使用续体来阻塞线程。 第

实例 3-2 使用续体来阻塞线程

```
syscall_1(arg1)
{
        ...
        save arg1  and any other state information;
        thread_block(f2);
        /* not raached */
}
f2()
{
        restore argl and any other state information;
        ...
        thread_syscall_return(status);
}
```

注意到当使用参数调用 thread_block()时，它并不返回到调用者。线程恢复执行时，内核把控制权传递给 f2() 函数 thread_syscall_return()用来从系统调用返回到用户级。整个过程对用户是透明的，用户所看到的只是从系统调用一个同步返回。

阻塞线程时，只有少数的状态需要保存时，内核才使用续体。比如，一个最通常的阻塞操作发生在页错误处理过程中，传统的实现中，处理程序发出一个读磁盘请求，在读操作完成之前一直阻塞。此时内核只是简单地把线程返回到用户级，这个应用就能重新恢复执行。这项工作必须在读操作完成之后才能处理,并且它需要非常少的保存的状态(可能要读入指向页面的指针，更新内存映射数据)，这个例子能很好的说明续体的用处。

3.9.3 优化

续体的最直接的优点就是它减少了系统中的内核堆栈的数目。对续体也有一些重要的优化。假设在上下文切换过程中，内核发现新旧两个线程都使用续体，老的线程已经消除了它的内核堆栈，新的线程没有内核堆栈，那么内核可以直接把内核堆栈从老的线程传递给新的线程。如图 3-12 所示，除了能够节省分配一个新的内核堆钱的开销外，因为相同的内存被重新使用，同时也能提高高速缓存和 TLB 的命中率(见 13.3.1 小节)。

Mach 的 IPC(进程间通信)实现(见 6.4 节)可以在更深程度优化。消息传递包含有两个步骤——客户通过系统调用 mach_msg 发送消息并等待应答和服务器使用系统调用 mach_msg 发送应答并等待下一个请求。消息发送到一个端口或从一个端口接收消息。端口是一个保护的消息队列。发送和接收是互相独立的，如果接收者没有就绪，内核就把消息放人端口的消息队列中。

当接收者在等待消息时.可以通过使用续体来优化消息传送。如果发送方看见接收方正在等待，它就直接把内核堆栈递交给接收方并且使用一个续体 mach_msg_continue()来阻塞自身。接收线程使用发送者的内核堆栈，那里包含发送消息的全部信息。这就避免了把消息放人队列和从队列中取出来的开销，并且加速了消息的传送。当服务器应答时，它将把它的内核堆栈还给客户线程，并让它以相同的形式重新执行。

图 3-12 使用续体处理堆栈

3.9.4 分析

在 Mach 中，续体被证明有非常高的效率。因为它们的使用是可选择的，它不用改变全部的编程模型，它们的使用可以被很快的扩展。在很大程度上，续体降低了对内核内存的需求。性能测量[Dray 91]表明，平均情况下，系统每个处理器只需要 2.002 个内核堆栈，并且每线程的内核空间需求从 4664 字节减少到 690 字节。

因为 Mach 3.0 是一个微内核，它提供的接口很小，并且抽象概念的数量也很少，所以它特别适合续体的使用。特别是，UNIX 兼容代码已经从内核中移去。Mach 3.0 中，UNIX 是由用户级的服务器实现的[Golu 90]。这样内核能够仅仅在 60 个左右的地方阻塞，而且大约 99% 的阻塞发生在仅仅 6 个“热点”上、相比之下，在传统的 UNIX 系统中，内核可能在几百个地方阻塞，而没有什么真正的热点。

3.10 小结

我们已经看到了设计多线程系统的多种方法、有许多种类型的线程原语，系统可以用一个或多个线程原语来创造一个丰富的并发编程环境。线程可能由内核支持，也可能由用户线程库支持或者由双方一起支持。

应用程序开发人员也必须选择正确的内核和用户方法。他们面临的一个问题就是，每个操作系统供应商提供一组不同的系统调用来创建和管理线程。这样编写出的多线程代码很难高效利用系统资源而且难有很好的移植性。POSIX 1003.4a标准定义了线程库函数.但是没有定义内核的接口和实现。

3.11 练习

1.对于下列这些应用，试讨论轻量级进程，用户线程或其他编程模式哪种更合适：

(a)分布式名字服务的服务器部分。

(b)窗口系统，如 X 服务器。

(c)运行在多处理器上的科学计算应用，执行许多并行计算。

(d)工具 make，尽可能并行地编译文件。

2.在什么情况下一个应用程序使用多进程比 LWP 或用户线程效果要好？

3.为什么每一个 LWP 都需要一个独立的内核栈？系统可以通过仅仅在 LWP 调用系统调用时才分配一个内核栈来达到减少资源耗费的目的吗？

4.proc 结构和 u 区包括了进程的属性和资源。在多线程系统中，进程的所有 LWP 共享它们的每个域，哪些必须是每个 LWP 私有的？

5.假设一个 LWP 正在调用 fork,此时同一进程的另一个 LWP 调用了 exit。如果系统通过 fork 复制了进程所有的 LWP，结果会怎样？如果 fork 仅仅复制一个 LWP 又会怎样？

6.如果系统中仅有一个系统调用原子地完成 fork 和 exec，多线程系统中的 fork 问题还会存在吗？

7.3.3.2 节介绍了单一共享集的问题，如文件描述符和当前目录。为什么这些资源不能由每个 LWP 或每个用户线程私有？[Bart 88]更进一步探讨了这个问题。

8.标准库中为每个进程定义了一个称为 errno 的变量，它存放最近一次系统调用的错误状态字。在多线程进程中，这产生了什么问题了这些问题可以怎样解决？

9.许多系统都将库分为线程安全的和线程非安全的。对于多线程应用来说，是什么造成函数是非安全的？

10.利用线程来运行中断处理函数的缺点有哪些？

11.让内核完成对 LWP 的调度有哪些缺点？

12.设计一个接口，通过这个接口用户可以控制哪个 LWP 被首先调度。这样会造成哪些问题？

13.比较 Solaris 和 Digital UNIX 系统的多线程原语。它们各自的优点都有哪些 ？

3.12 参考文献

①这就是说要每一个可用的处理器都服务于可运行的线程。当然，一个处理器在某一时刻只能运行一个线程。
②许多线程库需要内核为异步 I/O 提供必要的设施。
③直到达到一个配置上限。在 SVR 4 中，RLIMIT_NOFILE 限制了堆栈的大小。这个值包括一个硬限和软限。通过系统调用 getrlimit 获得这些值。系统调用 setrlimit 可以降低硬限，在不超过硬限的前提下降低或提升软限。
④某些多线程系统，诸如 SVR4.2/MP，提供用户线程堆栈自动增长设施。
⑤当然，由于正常的堆栈已经没有空间来处理这个信号，它必须在特殊的堆栈上处理。现代 UNIX 系统可以让应用程序在另一个堆栈上进行信号处理(见 4.5 节)。
⑦Mach 2.5 的实现本身是在内核内部提供了 4.2BSD 的功能。Mach 3.0 则是通过一个应用级的服务器程序实现了这些功能。