科技开发研究

分布式操作系统中线程包实现方法的对比研究

王凤岭

(南宁职业技术学院 计算机与信息工程系,广西 南宁 530003)

[摘 要] 内核级线程是微内核操作系统的基本调度单位，较好地支持了细粒度的并行计算，但在支持用户分布式并发模型上还有许多缺点，而用户级线程是在核心线程的支持下建立的更高层次的用户级调度单位，能较好地支持用户程序的并发执行。文章重点分析了分布式操作系统中线程包的两种实现方法，并对比研究了它们的优缺点。

[关键词] 分布式操作系统;内核线程;用户线程 虚拟处理机

[中图分类号] TP316.4 [文献标识码] A [文章编号] 1009-3621(2004)04-0076-05

Comparing analysis and Study on the ways to implement a threads package in Distributed system

WANG Feng - ling

(Computer & Information Engineering Department,
Nanning polytechnic, Nanning, Guangxi, 530003)

Abstract: Kernel - level threads is a basic scheduling unit on micro kernel operating system, and supports fine - grained parallel computing, but it has many shortcomings in supporting user - concurrency models. On the basis of kernel - level threads, user - level threads are the higher layer user - scheduling unit than kernel - level. They support concurrence execution of user programs. The paper focuses on discussing the ways to implement a threads package, and comparing & studding their shortcomings and virtues.

Key words: distributed operating system, Kernel - level threads, user - level threads, virtual processor

进程是资源管理的最小单位，每个进程有它自己的程序计数器、堆栈、寄存器和地址空间，不同的进程间互不干扰，它们通过通信原语(如:信号量、管程、消息等)进行通信。一个进程可能有一个或多个线程，这多个线程可共享一个地址空间并可以并行运行。

[收稿日期] 2004 - 08 - 09

[作者简介] 王凤岭(1972 -)，男，内蒙古乌兰察布人，南宁职业技术学院计算机与信息工程系主任、讲师。

线程(threads)也称为线索，是进程中的一个指令执行实体，是被系统独立调度和分派的基本单位，也是程序执行的最小单位。通常，进程管理着资源（如CPU、内存、文件等），而将线程分配到某个CPU上执行。线程自己基本上不拥有系统资源，只拥有一点儿在运行中必不可少的资源（如程序计数器、一组寄存器和堆栈）。因为每个线程的执行环境很小，负担很小，因此被称为一种轻量级进程(Lightweight process)。线程作为系统调度的基本单位，它的执行环境除了自身堆栈、寄存器、优先级等私有资源外，它与同属于一个进程的其它线程共享其所属进程所拥有的全部资源，同一个进程的每个线程按顺序分时执行，但在分布式多处理机系统中它们可以并行执行。比如，如果进程运行在分布式并行处理机器上，它就可以同时使用多个CPU来执行各个线程，达到最大程度的并行，以提高效率；同时，即使是在单CPU的机器上，采用多线程模型来设计程序，正如当年采用多进程模型代替单进程模型一样，使设计更简洁、功能更完备，程序的执行效率也更高，例如采用多个线程响应多个输入，而此时多线程模型所实现的功能实际上也可以用多进程模型来实现，而与后者相比，线程的上下文切换开销就比进程要小的多了，从语义上来说，同时响应多个输入这样的功能，实际上就是共享了除CPU以外的所有资源的。像传统的进程一样，线程也处于几种状态之一：运行、阻塞、就绪或结束。

在操作系统的设计和实现上，从进程演化出线程，最主要的目的就是使顺序和并行执行相结合，更好的支持分布式处理系统，以及减小（进程/线程）上下文切换开销，从而得到更大的系统吞吐量和更高的效率，这也是引入线程模型的意义所在。

针对线程模型的两大意义，分别开发出了内核级线程和用户级线程两种线程模型，当然当前还有一些新的研究成果，即它们的“混合体”。而分类的标准主要是线程的调度者在核内还是在核外。前者更利于并发使用多处理器的资源，而后者则更多考虑的是上下文切换开销。

在线程机制的具体实现上，可以在操作系统内核上实现线程，也可以在核外实现，后者显然要求核内至少实现了进程，而前者则一般要求在核内同时也支持进程。核心级线程模型显然要求前者的支持，而用户级线程模型则不一定基于后者实现。这种差异，正如前所述，是两种分类方式的标准不同带来的。

1. 线程包的实现

从实现方式上划分，线程有两种模型：“用户级线程”和“内核级线程”。

用户级线程指不需要内核支持而在用户程序中实现的线程，这种线程甚至在象DOS这样的操作系统中也可实现，但线程的调度需要用户程序来完成，这有些类似Windows 3.x的协作式多任务。另外一种则需要内核的参与，由内核完成线程的调度。这两种模型各有其好处和缺点。例如，对于用户线程不需要额外的内核开支，但是当一个线程因为I/O而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会；而内核线程则没有各个限制，但却占用了更多的系统开支。

对于这两种实现方法各有其长处和不足，目前还存在一些争议。显然，还会出现一些“混合”的方法。接下来将讨论这些方法以及它们的优缺点，并进行对比研究。

1.1 在用户空间中实现线程包

这种方法是将线程包完全放到用户空间中去，内核对此一无所知。当然这里所涉及的内核是管理普通的、单线程的进程，这种方法的好处是：

1.1.1 用户级的线程包能够在不支持线程的操作系统中实现。例如：UNIX并不支持线程，但已经有了为它而写的各种各样的用户空间的线程包。

1.1.2 所有这些的实现都有相同的结构，如图1所示。用户级线程包的实现方法，实现线程切换比使用内核陷阱至少快一个数量级。

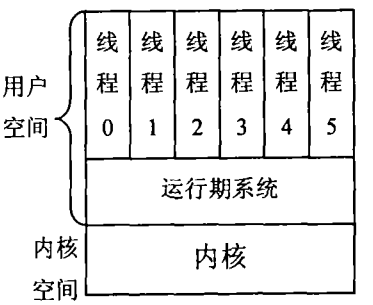

图1　用户级线程包

它们的执行过程是这样的：线程运行在运行期系统（runtime system）的上层，运行期系统是一些管理线程的过程集，当一个线程执行了一个系统调用时就进入休眠，对信号量或互斥体执行一个操作，或其他什么原因而挂起，它都调用一个运行期系统过程，这个过程检查线程是否必须挂起。如果是，它将该线程的寄存器存储到一个表中，然后寻找另一个未阻塞的线程运行，将新线程所保存的值重新装入到它自己的机器寄存器中。一旦切换了堆栈指针和程序计数器的内容，新的线程就自动进入运行状态。如果机器有一条指令用来保存所有寄存器且有另一条指令用来装入所有寄存器的值时，整个线程切换可用有限的几条指令来完成。

1.1.3 用户级线程包允许每一个进程有自己定制的调度算法。对某些具体的应用，不用担心它在不合适的时刻被停止，如：有垃圾收集器线程的应用，这就是一个很好的例子。

1.1.4 在用户空间中实现的线程包，它们的可扩展性也很好。由于内核级线程总是需要内核中的许多表和堆栈空间，如果存在大量的线程将产生问题。

当然，在用户空间中实现线程包，仍存在一些主要的不足：

(1)首先是阻塞调用是怎样实现的问题。假设线程从空管道中读数据或其他的一些操作将导致阻塞的操作。让线程发出这样的系统调用是不可行的，因为这样将终止所有线程。引入多线程的主要目的是允许每一个线程使用阻塞调用，但又要阻止已阻塞的线程去影响其他的线程。运行阻塞系统调用，不能实现这个目的。

(2) 可以将系统调用全部变为非阻塞系统调用，这对于操作系统的改变是微不足道的。例如：读一个空管道将会失败问题。

(3) 对于用户级线程的一个争论恰恰是通过事先确定一个调用是否引起阻塞，它可运行在现存操作系统中。比如，在 UNIX 版本中，存在一个调用 SELECT 方法，可以告诉调用者管道是否为空等，再来改变 READ 的语义执行，但这将需要修改许多用户程序。需注意的是，重写部分系统调用库的方法不仅效率低，而且不太方便。

(4) 与阻塞系统调用类似的问题是页面错，如果一个线程产生页面错时，内核甚至并不知道这些线程的存在，自然会阻塞整个进程直到所需要的页面取出为止。

(5) 用户级线程包的另一问题是如果一个线程开始运行，进程中的其他线程除非在第一个线程自愿释放 CPU 后它才能运行。在单一进程中，没有时钟中断，也不可能实现轮转法调度，除非一个线程自愿进入运行期系统，否则没有任何机会让调度程序调度它。

同步领域中无时钟中断是致命的。在分布式中有一种很普遍的现象。一个线程初始化一个动作时，另一线程必须响应，且连续进行循环检测看响应是否发生。这种条件叫旋转锁（spinlock）或忙等待。这种方法在期望得到快速响应时很有用，同时使用同步信号量的花费又很高。如果线程能领依靠时钟中断在几个毫秒的时间内自动调度，这种方法工作得很好。然而，如果线程在阻塞前一直运行，这种方法容易产生死锁。

解决线程一直运行的一个办法是让运行期系统每秒得到一个时钟信号中断，来使它得到控制权，这样会使程序太粗糙和混乱。更高频率的周期性时钟中断并不总是可能的。即使可能，总的费用也很大。而且，线程需要的时钟中断可能与运行期系统使用时钟相互干扰。

(6) 另一个也可能是对用户级线程构成最大的威胁是编程人员通常想在线程经常阻塞的应用中使用多线程，例如多线程文件服务器。这些线程不停地进行系统调用。一旦产生陷阱进入内核执行系统调用，如果旧线程阻塞了，内核要进行线程切换将很困难，并且将使内核不停地检查系统调用是否安全。对于那些本质上完全受限于 CPU 和几乎不阻塞的应用程序，有多线程又有什么意义？没有人会认真地提出使用多线程来计算前 N 个质数或用它来下棋，因为这样做什么好处也得不到。

2. 在内核中实现线程

当内核知道并管理线程时，则不再需要运行期系统，而当一个线程想去创建一个新线程或撤消已存在的线程时，它发出一个内核调用，由它完成创建和回收工作，这种机制是在内核中实现线程，如图 2 所示。

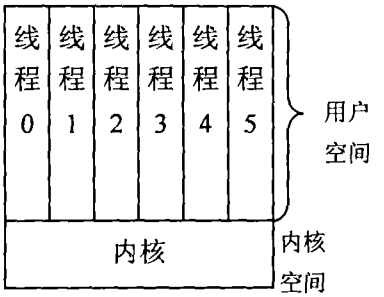

图2 由内核管理的线程包

为了管理所有线程，在内核中每个进程都有一张表，每个线程在表中有一个入口(线程表)。每个入口保存着线程的寄存器、状态、优先权和其他信息。这些信息和用户级线程一样，但它现在是在内核中，而不是在用户空间中。这些信息类似于传统内核中维护的每一个单线程进程的信息。相对于用户级线程来说，在内核中实现线程也有许多优点和不足的地方，归纳如下：

(1) 所有可能阻塞线程的调用，如线程间使用信号量同步，都是按系统调用来实现的，但是这将比运行期系统过程开销更大。当一个线程阻塞时，内核根据它的选择，可运行同一进程中的其他线程(如果有已准备好的)，或者另一个进程的线程。在用户级线程中，运行期系统持续运行自己进程中的线程直到内核收回处理机(CPU)，或者没有已就绪的线程可运行了为止。

(2) 由于在内核中创建和撤消线程需更高的花费，所以有些系统采用环境补偿的方法再循环它们的线程。当线程撤消时，将它标记为不能运行，但它的内核数据结构却不受影响。稍后当它作为一个新线程创建时，原有的线程被重新激活，这样，节省了一些开销。对用户级线程来说线程再循环也是可能的。但是，由于线程管理的开销较小，因此没有理由要这样做。

(3) 内核线程不需要任何新的非阻塞系统调用，当使用旋转锁(spin lock)时，它们也不导致死锁。而且，如果一个进程中的线程产生了页面错，内核在等待来自磁盘或网络的所需页面时，能很容易地运行另一线程。它们的主要缺点是系统调用耗费很大。因此如果线程操作(创建，删除，同步等)很频繁，将引起更多的系统开销。

另外，除了用户线程和内核线程所特有的问题外，它们还有一些相同的问题。例如，许多库函数是不可重入的。例如，可用编程为在网络上发送一条消息，它首先在一个固定的缓冲区中组装要发送的消息，然后一个时钟中断强迫将该线程切换为另一个线程，那个线程又马上用自己的消息重写缓冲区，这时会发生什么情况呢？类似的，当一个系统调用完成后，发生了线程切换，可能改变前一线程读出的错误状态。同样，内存分配过程，可随意地访问临界表而不影响设置和使用受保护的临界区，因为它们是为单线程环境设计的，而在那种环境下根本没必要，像 UNIX 中的 malloc 命令就是这样，而要有效地解决所有这些问题就意味着重写整个库。

另一解决办法是给每个过程提供一个 jacket(外壳)，当过程被启动时，它锁住一个全局信号量或互斥体。通过这种方法，在库中一次仅有一线程是激活的。这样，实际上整个库成为了一个大的监视程序。信号也是一个难题，假设某线程想捕获一个特殊信号(例如：用户单击 DEL 键)，而另一线程想让这信号终止进程。可能出现这种情况：如果一个或多个线程运行标准库过程，而其他一些是用户写的。显然，这些线程的愿望不相容。通常，在单线程环境中管理信号是很困难的，而在多线程环境下也不容易。信号，是典型的每进程概念，而不是每线程的概念，特别是当内核不知道线程的存在时。

在目前的商用系统中，通常都将两者结合起来使用，既提供核心线程以满足分布式处理系统的需要，也支持用线程库的方式在用户空间中实现另一套线程的机制，此时一个核心线程同时成为多个用户态线程的调度者。正如很多技术一样，“混合”通常都能带来更高的效率，但同时也带来更大的实现难度，出于”简单”的设计思路，比如 Linux 从一开始就没有实现混合模型的计划，但它在实现上采用了另一种思路的“混合”。比如，调度者行为。它的目标是模拟内核线程的功能，像在用户空间内实现的线程包一样有更好的性能和有更大的灵活性。

尽管调度者行为解决了怎样将控制权由进程中的阻塞线程传递给非阻塞线程的问题，但同时它产生了一个新问题：被中断的线程在挂起时可能正在执行对信号量的操作，在这种情况下，这个线程在就绪队列中可能正持有一把锁。如果由up-

call 启动的运行期系统企图自己得到这把锁，目的是将新的就绪线程放放队列中，它就可能失败并随之产生死锁，这种问题能通过跟踪线程什么时候在或不在临界区来解决，但这种解决方法太复杂并且很不高明。另一个调度者行为的缺陷是它主要依靠upcall，upcall 这个概念与任何一个分层系统的内在结构相违背。通常，第 n 层提供第 n + 1 层可调用的某种服务，但是第 n 层是不能调用第 n + 1 层中的过程。

当核内既支持进程也支持线程时，就可以实现线程 - 进程的"多对多"模型，即一个进程的某个线程由核内调度，而同时它也可以作为用户级线程池的调度者，选择合适的用户级线程在其空间中运行。这就是前面提到的"混合"线程模型，既可满足多处理机系统的需要，也可以最大限度的减小调度开销。绝大多数商业操作系统（如 Digital Unix、Solaris、Irix）都采用的这种能够完全实现 POSIX1003.1c 标准的线程模型。在核外实现的线程又可以分为"一对一"、"多对一"两种模型，前者用一个核心进程（也许是轻量进程）对应一个线程，将线程调度等同于进程调度，交给核心完成，而后者则完全在核外实现多线程，调度也在用户态完成。后者就是前面提到的单纯的用户级线程模型的实现方式，显然，这种核外的线程调度器实际上只需要完成线程运行栈的切换，调度开销非常小，但同时因为核心信号（无论是同步的还是异步的）都是以进程为单位的，因而无法定位到线程，所以这种实现方式不能用于多处理器系统，而这个需求正变得越来越大，因此，在现实中，纯用户级线程的实现，除算法研究目的以外，几乎已经消失了。

3. 小结

线程作为并发处理的一种实现方式，成为了许多并行程序设计的解决方法，其可以在分布式操作系统内核和用户地址空间中实现，即内核级线程和用户级线程。

内核级线程是微内核操作系统的基本调度单位，较好地支持了细粒度的并行计算，向不同用户提供了较低层次的系统调用，从而可以设计出不同的多线程应用程序，但存在移植性、灵活性、完备性差，操作不方便、效率低，核心进程所提供的调度策略不能满足不同用户的需求等缺点。从现实的性能上来看，从根本上弱于用户级线程，而用户级管理线程并行，是获得高效并行计算效率的基本要素，但由于现代多处理器操作系统缺乏对于用户级线程的内核支持，所以在将用户级线程与其他系统服务整合在一起时，也遇到一系列问题，有待更进一步地研究和探讨。

[参考文献]

[1] 陆丽娜等译．分布式操作系统[M]．北京：电子工业出版社，1999.

[2] AGHA, G. Actors: A Model of Concurrent Computation in Distributed Systems[J]. MIT Press, Cambridge, Mass. 1996.

[3] BARNES, J. AND HUT, P. A hierarchical O(N log N) force - calculation algorithm[M]. Nature 324 (1996), 446 - 449.

[4] BLACK, D. Scheduling support for concurrency and parallelism in the Mach operating system[J]. IEEE Comput. Mag. 23, 5 (May 1990), 35 - 43.

[5] CHERITON, D. The V distributed system[M]. Com - mun. ACM. 31, 3 (Mar. 1998), 314 - 333.

[6] DRAVES, R. AND COOPER, E. C Threads. Tech. Rep[J]. CMU - CS - 88 - 154, School of Computer Sci ence, Carnegie Mellon Univ. June 1998.

[7] 徐永森，陈俊良．JAVA 应用程序设计和开发环境[M]．南京：南京大学出版社，1998.

[8] SCHROEDER, M. AND BURROWS, M. Performance of Firefly RPC[J]. ACM Trans. Comput. Syst. 8, 1 (Feb. 1990), 1 - 17.

[9] Andrew S. Tanenbaum . Distributed Operating Systems [M]. 北京：清华大学出版社，1998.

[责任编辑：黄桂婵]
[责任校对：曾广春]