VTOS:一个支持多核的微内核操作系统设计与实现

景香博,陈 江,钱振江

(南京大学 计算机科学与技术系 软件新技术国家重点实验室,江苏 南京 210093)

摘 要:现在处理器的发展已经进入了一个新的时代,继承了几十乃至上百个核心的处理器已经出现。这在大大提升了硬件处理能力的同时,也给软件设计,尤其是操作系统设计带来了很大困难。为了提高系统的可扩展性,操作系统开发人员需要花费大量的精力来进行严谨而有效率的同步设计。由于宏内核自身的复杂性,在进行同步设计时很困难。阐述了一个微内核的多核同步设计方案,由此可以发现在设计多核操作系统时微内核系统先天的巨大优势。

关键词:多核;微内核;操作系统;可扩展性;同步

中图分类号:TP316 **文献标识码**:A **文章编号**:1672-7800(2012)010-0125-05

0 引言

近10年来,传统的单指令流处理器性能提升遇到了瓶颈,因为即使采用了流水线等技术,指令序列所能达到的并发程度是有限的,而减小晶体管、提升主频的方法也因为严重的功耗和散热问题而遇到了障碍。由于这些原因,处理器性能提升速度明显变慢。据统计,在20世纪90年代,处理器性能平均每年提升60%,而从2000年到2004年性能提升速度下降到40%,2004年更是只有20%的提升。在这种情况下,单核处理器的发展已经达到一个极限,多核技术应运而生。

多核处理器通过在一个芯片上集成多个处理核心,多个核心并行的工作来提升总体性能。Knight指出,处理

参考文献:

[1] 张清雅.基于Web Service技术的考勤管理系统的分析与实现[J].福建电脑,2010(1).

[2] 尚俊杰.网络程序设计——ASP[M].北京:清华大学出版社,2009.

[3] 顼宇峰,马军.ASP网络编程从入门到精通[M].北京:清华大学出版社,2006.

[4] 张永瑞.基于ASP.NET的自适应考试系统的设计与实现[D].南京:南京理工大学,2010.

(责任编辑:孙 娟)

Design and Realization of ASP of Oneline Learning Based on B/S

Abstract: Along with the computer the rapid development of Internet technology, Internet has penetrated into all aspects of life, and the masses of people living environment and way of life has a great influence in the field of education, the application is more widely. New ways of teaching to change the traditional teaching mode, this paper introduces the design of ASP based on B/S mode technology of online learning system, breaking the limitations of traditional classroom teaching mode, form a kind of interactive, open unrestricted teaching mode.

Key Words: ASP system; online learning; teaching model; database

基金项目:863计划重大项目(2011AA01A202);国家自然科学基金项目(61021062);江苏省"六大人才高峰"高层次人才项目(2011-DZXX-035)

作者简介:景香博(1986-),男,南京大学计算机科学与技术系硕士研究生,研究方向为操作系统安全;陈江(1988-),男,南京大学计算机科学与技术系硕士研究生,研究方向为操作系统安全;钱振江(1982-),男,博士,软件新技术国家重点实验室、南京大学计算机科学与技术系讲师,中国计算机学会会员,研究方向为计算机安全与形式化验证、嵌入式系统。

器主频每提高 400MHz，功耗就上升约 60%，因此当主频比较高了之后，使用多个低功耗的核心带来的性能提升明显要高于提升主频。与多处理器系统相比，多核处理器优势明显：同一芯片上的多个核心互联线路极短，有利于降低通信延迟，提高数据传送带宽。随着多核技术的成功应用，以及高端应用的强大需求，多核技术快速发展。早在 2009 年，Tilera 公司推出的 TELA 系列处理器已经在一个芯片上集成了 100 个核心。而 Intel 公司也推出了 80 核的芯片。处理器即将进入众核时代。

多核技术的快速发展，大大提升了硬件处理能力。然而此时，也给操作系统设计与实现带来了很大的挑战。为了有效地利用多核处理器，操作系统必须能够在多个核心上同时运行内核服务。这时，就必须采取一些同步机制来保护共享的数据结构，以避免并行的访问导致数据结构遭到破坏。在核心很多时，同步粒度的设计对于整体性能影响很大，因此系统的可扩展性是一个重要的指标。根据系统的不同，内核中可能包含很多数据结构，繁杂的同步很可能导致死锁。考虑到这些因素，要实现一个可扩展性良好的操作系统内核可以说非常困难。

本文首先简要介绍当前主流的多核操作系统设计方法，然后介绍了在 VTOS 中采用的多核设计方案，分析了其作为一个微内核系统在多核设计中的优势。最后，本文给出了系统设计中尚需要进一步解决的问题。

1 主流多核操作系统设计方案

在多核操作系统中，硬件决定了系统必然是并行的。因此，即使关闭处理器对中断的响应，并且在没有异常发生的情况下，依然可能发生对数据的并发访问。为了协调多个处理器上内核对共享数据的访问，当前操作系统采取的方法主要包括：非对称结构设计、基于锁的设计、无锁设计、虚拟化设计。

最常见的非对称设计就是主从结构设计。在主从结构设计中，只有一个主处理器在运行内核代码，其它从处理器只执行用户态应用程序。在一个典型实现中，主处理器和从处理器各有一个队列。当从处理器上的应用程序需要内核服务时，从处理器首先进入内核态，然后将当前进程迁移到主处理器的队列上运行。主从结构的设计很简单，而且可以在任意数目的处理器上运行。但是因为内核在任一时刻只能为一个进程服务，因此当应用程序很多时，内核会成为系统瓶颈，可扩展性较差。

基于锁的设计可以分为大内核锁、粗粒度锁、细粒度锁。采用大内核锁的系统中，使用一个自旋锁来保护整个内核，以避免并发访问。粗粒度的实现仅使用少数几个锁，每个锁保护内核的很大一部分，或者就是一个子系统。细粒度的实现使用很多的锁，主要用于保护每个数据结构。大内核锁使得所有的内核操作串行执行，因此实现起来很简单。然而，同主从结构设计一样，大内核锁的可扩展性非常差。核心数目越多，对大内核锁的争用就会越激烈，多核的性能优势很难发挥出来。许多操作系统最初引入 SMP 支持的时候，采用的就是大内核锁的设计，例如 Linux 2.0，FreeBSD 等，后来逐步替换为其它粒度细一些的锁机制。微内核系统 QNX Neutrino 也是采用了大内核锁，其设计者认为微内核的操作非常快，而主要的操作系统功能都实现在用户态服务进程中，因此服务进程的并发比微内核要重要的多。

对于性能要求较高的操作系统，尤其是宏内核操作系统来说，大内核锁的设计导致的性能瓶颈很难接受。因此，许多采用大内核锁的系统后来逐步开始使用细粒度的锁。在 Linux 2.2 内核中，系统开始采用粗粒度的锁，各个子系统，例如文件系统、声卡驱动，开始使用各自独立的锁，从而使得各个子系统可以并发运行。在这个阶段，锁保护的依然不是各个数据结构，而是代码，避免代码被并发执行，而不是避免数据结构被并发访问。

从 Linux 2.4 内核开始，系统采用了细粒度的锁，并在 2.6 内核中进一步细化。细粒度的锁保护具体每个共享数据结构以避免并发访问。然而，大内核锁也并没有完全弃用，部分代码依然靠大内核锁保护。

除了基于自旋锁的机制，在 Linux 2.6 内核中还引入了新的顺序锁和 RCU 机制。顺序锁通过引入一个计数器，每次对数据结构进行写操作时，在写操作之前和之后都要对计数器进行加一操作。读者通过判断读之前和读之后计数器的值来判断读到的值是否有效。当然写操作还是需要自旋锁互斥的。RCU 机制则完全不使用锁，并且允许多个读者和写者并发执行。在 RCU 机制下，有一个指针指向当前数据结构。当读者读一个数据结构时，会获取当前该数据结构的指针，通过该指针访问数据结构。但是写者需要做很多事情来保证一致性。写者要更新数据结构时，首先获取数据结构的一个副本，然后修改这个副本，最后将指针指向修改后的副本。但是，在写操作完成之前开始访问数据结构的读者访问的仍是之前的副本，因此该副本不能被丢弃，直至该副本的读者都完成读操作。

还有一个方案就是引入虚拟化机制，将多核处理器虚拟化成多个单核的计算机系统，在每个系统上运行一个操作系统实例。这种方案的例子有 vmware、Xen 等系统。但是虚拟监控器本身依然要通过其它手段来支持多核。

1.1 系统设计面临的困难

如前所述，主从设计的可扩展性非常差，因此在此不对其进行讨论。当前主要操作系统都是基于锁机制来进行同步的，下面以 Linux 为例主要介绍加锁机制所面临的问题。

据统计，在 Linux 2.0.40 内核中包含了 17 个 BKL（大内核锁）调用；在 Linux 2.4.30 内核中包含了 101 个 BKL，329 个自旋锁，121 个读写锁调用；而在 Linux 2.6.11.7内核中包含了 101 个 BKL，1717 个自旋锁，349 个读写锁，56 个顺序锁，14 个 RCU 调用。

因为粗粒度的锁即使在 4 个核心的处理器上可扩展

性也并不好，Linux 内核开发者们使用各种手段分析哪些锁造成了性能瓶颈，并通过细化锁、使用 CPU 本地数据等方法来解决这些问题。从图 1 可以看出，这些方法大大提高了系统的可扩展性。

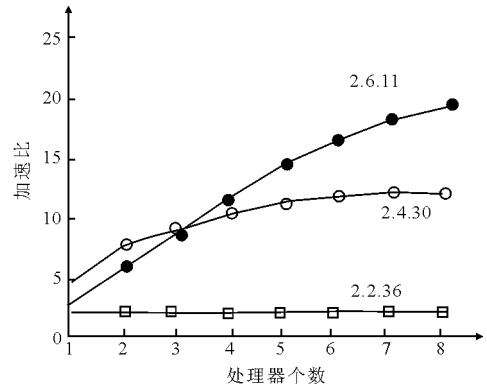

图 1　Linux 的可扩展性

随着处理器技术快速发展，核心数目越来越多，Linux 内核开发者们耗费了很大精力，做了大量的工作来提高 Linux 的可扩展性，但是依然有些问题无法解决：首先，宏内核中存在大量复杂的数据结构，锁究竟能够细化到何种程度是个问题。锁越是细化，复杂度就越高，而分析如何加锁是很耗时而且容易出错的，在庞大的宏内核中，必须非常小心地避免死锁。其次，即使可以细化，从图 1 中可以看出，虽然在多核情况下使用更细加锁的 2.6 内核性能比 2.4 内核好，但是在核心数目较少时性能反而不如 2.4 内核，这是因为 2.6 内核中锁更多，而加锁操作本身也是相当耗时的。锁越多，加锁耗时就越多，但是为了可扩展性又需要细粒度的锁，因此加锁是一个需要仔细权衡的问题。

2　VTOS 的设计与实现

VTOS 是一个基于 Minix 3 的操作系统。Minix 是荷兰 Vrije 大学的 Andrew S. Tanenbaum 领导开发的操作系统，该系统采用微内核结构，其内核非常精简，并遵循 POSIX 标准，大部分系统服务运行在用户态并被划分成相互独立的小模块。传统的观念一般认为微内核的性能比较差，但是 Härtig 通过实验对比二代微内核 L4Linux 和宏内核 Linux 得出结论：相比于 Linux，应用程序在 L4Linux 上平均性能的下降大约在 5%～10%之间，由此可见，在对系统结构以及 IPC 采取一定优化措施的微内核系统的性能是可以接收的。而微内核架构带来的可靠性、灵活性在多核操作系统的设计中会带来很大优势。因此，决定采用 Minix 作为系统原型。

2.1　系统结构

如图 2 所示，VTOS 系统主要由微内核、系统任务进程、驱动进程、服务进程和应用程序组成。从图 2 中明显可以发现，该系统并没有采用主从式结构，而是在每个处理器上都可以运行微内核。微内核完成消息传递、最基本的中断处理、虚拟内存这些功能。服务进程、驱动进程等分别完成操作系统功能中的一部分功能，例如文件系统完成文件相关操作。与应用程序一样，这些服务进程和驱动运行在用户态，有自己独立的地址空间，彼此之间通过微内核提供的消息传递进行通信。操作系统的大部分功能都是在用户态服务进程中实现的，但是，有些操作必须要修改内核数据结构，例如用户程序执行 exit 调用，那么就必须将该进程从微内核的可执行队列中移除，并修改相应的进程控制块内容，这些工作是由系统任务进程完成的。系统任务进程与微内核共享地址空间，可以直接访问微内核中的数据结构，当完成操作系统功能（例如 fork 调用）时需要对微内核中数据结构（进程表）进行修改时，就需要给系统任务进程发送消息请求系统任务进程完成相应操作。

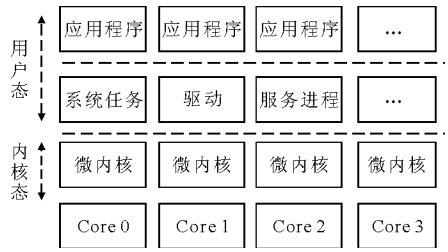

图 2　VTOS 的结构

VTOS 的系统结构与 Linux 等宏内核操作系统有根本的不同。在 Linux 等系统中，几乎所有的操作系统功能，包括驱动、进程管理、文件系统等，全部在内核中实现，因此内核很庞大，如前所述，给同步设计带来很大困难。而在微内核的设计中，各个模块彼此分离，在实现同步时，只需要考虑各个模块内部的同步，各个模块的依赖性大大降低。而且在实现服务进程的多线程化时，只需要根据处理器的数量，动态地决定每个服务进程的线程数量，并根据处理器拓扑结构绑定到不同的处理器上去。这样只要各个服务进程内部同步设计良好（在一个相比宏内核小很多的服务进程中进行同步也简单很多），那么解决系统的可扩展性问题就比宏内核系统要简单很多。

VTOS 的系统结构与其它的微内核系统也是有差别的。Tanenbaum 领导的 Minix 开发依然在进行，并且也添加了对 SMP 的支持。在 Minix 的设计中，对内核数据结构的操作都是在微内核中完成的，而在我们的设计中则是在系统任务中完成的。这样设计的原因是：首先，微内核操作系统的设计原则之一就是保持微内核尽可能的小，从而可以更容易地优化其性能，并减少错误。另外，这样的设计使得对微内核中数据结构的访问来源清晰地分成了两部分—微内核自身和系统任务，这有利于接下来要进行的同步设计。

2.2　内核同步设计

如前所述，按照我们的系统结构，对微内核中数据结构的访问有两个来源——微内核自身和系统任务。因此，不但需要考虑各个处理器上微内核之间的同步，也要考虑微内核与系统任务进程之间的同步。通过分析内核数据结构我们发现，这样的结构设计使得内核同步变得更加简

单。

大内核锁的设计很容易,只需要在内核入口和出口处进行加解锁操作,以及在系统任务进程需要访问内核数据结构时进行相应操作即可。如前所述,QNX 认为服务进程的并发比微内核更重要,因此采用了大内核锁设计。但是当核心数目很多时,大内核锁的设计必然会成为系统瓶颈。因此,这不是所需要的方案。而无锁的解决方案实现起来很复杂。因此,我们决定采用加细粒度的锁来保持同步。

首先总结出将微内核中需要访问的数据结构的集合,称为 KernelDataSet。在考虑微内核之间的同步时,KernelDataSet 中的数据结构都是需要考虑的。在考虑微内核与系统任务进程的同步时,则只有在系统任务进程需要访问 KernelDataSet 中的数据结构时才需要进行同步。按照我们的分析,KernelDataSet 中的数据结构主要有这些:

(1)进程控制块。一个进程可以在任何一个处理器上运行,在消息处理、时钟中断等情况下,我们都可能需要修改一个进程的状态。

(2)可运行进程队列。内核当然会访问以及修改自己的队列。

(3)物理内存管理系统的 zone_table 数据结构。

(4)记录哪些驱动使用了哪些中断的 irq_handlers 链表。

(5)闹钟队列。

(6)记录内核 log 的 kmess 数据结构。

(7)记录随机数的 krandom 数据结构。

内核同步是使用自旋锁实现的。下面分别讨论对这些数据结构的保护。

对于可运行进程队列、krandom 数据结构来说,只需要分别为其设置一个锁,当访问时对其加锁即可。每个核有自己的可运行进程队列,并且每个核极少访问其它核的队列,而闹钟队列、krandom 的访问也很少,因此在自旋锁上的冲突很少。

对于 kmess 数据结构,内核的 kprintf 函数会将内容打印到该结构中。由于我们并不希望两个核调用 kprintf 打印的内容交错出现,所以应该保护的不应该是 kmess 结构,而是 kprintf 函数。因此只需要在函数起始处和结束处分别加解锁即可。

对于 irq_handler 和闹钟队列这两个单向链表,可以采用类似于 RCU 的设计思想。在内核中对该链表只有读操作,因此任何事情都不需要做,只是直接读就可以了。而在系统任务修改该链表时,则通过使用原子操作来保证内核读到的数据都是正确可用的。为此,在加入一个节点时,系统任务需要先设置好节点内容,然后原子的加入链表,由于加入链表只需要使最后一个节点的相应指针指向新节点,因此是一个原子操作。删除节点时的情况则相反,先原子的从链表中删除,然后修改节点内容,删除时使上一个节点相应指针指向下一个节点的操作也是原子操作。由于节点是从一个静态数组中分配到的,而且不像 RCU 机制是动态分配的,并且节点被删除时并不是其内容马上被清空,而是延迟到进程退出时才清空,所以也不必像 RCU 一样进行繁杂的销毁操作。由于对这两个链表来说读操作无需加锁,而写操作非常少,因此可扩展性和效率都很好。

对于物理内存管理系统,现在可用页面信息都是放在一个共享数据结构中的,因此需要对该系统实现一个锁。但是考虑扩展性要求,以后可以像 Linux 一样实现 CPU 本地的空闲页面集合,每次从本地空闲页面中分配,只有本地页面不足时才从全局数据结构中分配。

在每个进程控制块中添加一个锁,这个锁不但保护进程控制块本身,还保护附属该进程控制块的数据结构,例如页表、等待该进程的队列等。进程控制块的加锁稍微复杂一些。

2.2.1 加锁的若干原则

为了保证系统的安全性,我们还需要若干原则,以保证系统不会发生死锁。

首先,在内核进行消息处理时,有的时候是要操作两个进程的,因此要对两个进程加锁。但是对两个进程加锁的顺序如果不当,就会导致死锁。比如 A 希望给 B 发消息,A 所在处理器的内核先对 A 加锁然后对 B 加锁,而 B 可能希望从 A 那里接受消息,于是 B 所在处理器内核先对 B 加锁然后对 A 加锁,于是形成死循环。为了解决这个问题,我们约定:遇到需要对两个进程加锁情况时,调用一个函数 dlock_proc 实现,该函数比较两个锁的地址,先对低地址加锁,然后对高地址加锁即可。由于系统中不存在需要同时持有两个以上进程锁的情况,并且要求持有两个进程锁时只能通过 dlock_proc 进行加锁,因此对进程控制块的加锁肯定不会相互等待形成死锁。

其次,由于我们使用的自旋锁是不能迭代的,也就是说如果一个核上两次获取同一个锁时就会导致死锁,因此要求必须在关中断的情况下才能使用锁,以避免获取锁之后发生中断,另一个内核路径产生对该锁的争用导致死锁。事实上,微内核运行时始终是关中断的,只需要在系统任务中需要加锁时关中断即可。在系统任务需要使用锁时关中断还有一个好处:由于自旋锁的忙式等待特性,因此加锁的一个重要原则就是在任何地方不能长久地持有一个锁。即使系统任务与其所在核上的微内核不发生死锁,如果系统任务没有关中断,导致在持有锁的情况下被切换出去,那么其它核上的微内核请求该锁时,可能要等待很久,直到系统任务被重新调度运行并且释放锁为止。通过要求在系统任务需要使用锁时关中断,可以完全避免这种情况的发生。

此外,我们约定,在任何地方,获取了除进程控制块锁之外的锁之后,不能再对其它任何数据结构加锁。这也就意味着,对除了进程控制之外的数据结构的使用完全不会导致死锁。上述工作表明,这是完全可以实现的。事实上,唯一有违该原则的地方就在于 irq_handlers 和闹钟队列链表,由于中断系统需要对该数据链表中记录的进程发

送消息，如果使用自旋锁的话，就需要先对该链表加锁，然后对进程控制块加锁（给进程发送消息可能会修改进程状态，因此需要加锁），从而违背该约定，可能导致死锁。但是通过使用类似于 RCU 机制，在读该链表的时候无需加锁，从而避免了在这里发生死锁的可能性。

第三个原则保证了对任何除了进程控制外之外的数据结构的使用完全不会导致死锁，而第一个原则保证了对进程控制块的加锁肯定不会相互等待形成死锁。结合第二个原则，可以得出结论，在系统中不会因为对数据结构的加锁导致系统死锁。

通过比较，可以发现，在微内核中进行同步比宏内核的系统要简单很多很多。在微内核只需要自旋锁和开关中断机制即可完整地实现内核同步，还可以保证系统中不会发生死锁，并拥有很好的可扩展性。而在宏内核中，这是非常困难的，例如在 Linux 中采用了开关中断、自旋锁、读写锁、顺序锁、RCU、信号量、打开和禁止抢占等很多复杂的机制来保证同步的正确性，但是依然不能保证死锁的发生。而在可扩展性方面，宏内核需要做很多的工作才能达到较好的可扩展性，而微内核由于其小巧的设计，为了实现较好的可扩展性则要容易很多。

3　结语

本文首先介绍了为了支持多核处理器，操作系统中所使用的主要的解决方案。然后分析了多数操作系统所采用的锁机制在当前硬件快速发展情况下所面临的问题。最后，提出了微内核的解决方案，并将其与其它操作系统的实现做了对比，指出相对于宏内核而言，由于微内核天然良好的结构，实现微内核的多核支持要容易很多，并且实现良好的可扩展性也比宏内核简单很多。完成微内核的多核设计之后，我们的工作还有待继续，分析系统的性能瓶颈，实现服务进程的多线程化是需要进一步研究的工作。

参考文献：

[1] GEER D. Chip makers turn to multicore processors [J]. Computer, 2005(5).

[2] W KNIGHT. Two heads are better than one [J]. IEEE Review, 2005(9).

[3] Tilera Corporation[EB/OL]. http://www.tilera.com.

[4] Intel Corporation[EB/OL]. http://techresearch.intel.com/ProjectDetails.aspx? Id=151.

[5] ANDI KLEEN. Linux Multicore Scalability[C]. Proceeding-Linux Kongress 2009 and Open Solaris Developer Conference 2009 in Dresden, 2009.

[6] ANDI KLEEN, TIM CHEN. Linux kernel scaling[R]. Santa Rosa, Linux Plumbers, 2011(15).

[7] S KAGSTROM, H GRAHN, L LUNDBERG. et al. Development Effort for Multiprocessor Operating System Kernels: Performance and Implementation Complexity in Multiprocessor Operating System Kernel[C]. Blekinge Institute of Technology, 2005.

[8] CURT SCHIMMEL. 现代体系结构上的 Unix 系统[M]. 张辉，译. 北京：人民邮电出版社，2003.

[9] ARWED STARKE. Locking in OS Kernels for SMP Systems[C]. The seminar Hot Topics in Operating Systems, 2006.

[10] PAUL E, MCKENNEY, JONATHAN APPAVOO, et al. Read-Copy Update[C]. Proceeding of the Ottawa Linux Symposium, 2001.

[11] HÄRTIG H, HOHMUTH M, LIEDTKE J, et al. The performance of μ-Kernel-Based Systems[C]. Proceedings of the 16th ACM symposium on Operating systems principles, 1997.

（责任编辑：杜能钢）

VTOS: Design and Implementation of a Micro-Kernel OS with Multicore Support

Abstract: As the emerging of processors with hundreds of cores, we are entering a new era. While providing much better performance, it also brings great challenge to the design of software, especially OS. OS developers need to spend much effort in synchronization design to improve system scalability. Because of it's complexity, the synchronization design for mono-kernel is very hard. The authors present a synchronization design for a micro-kernel system in this article. We found that it's much easy compared with mono-kernel systems.

Key Words: Multicore; Micro-kernel; Operating System; Scalability; Synchronization