计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(5):21−29 [doi: 10.15888/j.cnki.csa.008426]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

多内核操作系统综述①

林玉哲[1], 蒋金虎[2], 张为华[2]

[1](复旦大学 软件学院, 上海 200438)
[2](复旦大学 上海市数据科学重点实验室, 上海 200438)
通信作者: 张为华, E-mail: zhangweihua@fudan.edu.cn

摘　要: 操作系统在现代生活中具有举足轻重的地位. 为了服务于不同的硬件环境和多样的应用场景, 操作系统需要在保持性能的同时具备良好的扩展性和灵活性. 多内核操作系统, 作为一种分布式的操作系统, 是该问题的解决方案之一. 本文分析了多内核操作系统的设计原理, 调研了现有的多内核操作系统技术, 并对这些技术和一些相关技术进行了比较. 最后, 本文对多内核操作系统研究的现状与未来方向进行了总结.

关键词: 操作系统; 多内核; 多核

引用格式: 林玉哲,蒋金虎,张为华.多内核操作系统综述.计算机系统应用,2022,31(5):21–29. http://www.c-s-a.org.cn/1003-3254/8426.html

Survey on Multikernel Operating Systems

LIN Yu-Zhe[1], JIANG Jin-Hu[2], ZHANG Wei-Hua[2]

[1](Software School of Fudan University, Shanghai 200438, China)
[2](Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 200438, China)

Abstract: Operating systems play an important role in modern life. To serve different hardware environments and diverse application scenarios, they need to be scalable and flexible while maintaining good performance. A multicore operating system, as a kind of distributed operating system, is one of the solutions to this problem. This study analyzes the design principles of multicore operating systems, investigates existing multicore operating system technologies, and compares these technologies with other relevant ones. Finally, the current situation and trend of the research on multicore operating systems are summarized.

Key words: operating system; multikernel; multicore

1　引言

操作系统在现代生活中的应用越来越广泛, 其发展主要面对着来自两方面的压力: 一是多样化的硬件, 二是复杂多样的上层应用. 操作系统, 对上, 需要能为不同的应用提供相对应的系统服务, 对下, 需要能够支持各种硬件, 发挥出它们的性能, 这些都给操作系统的扩展性, 灵活性, 性能提出了不同程度的挑战. 从扩展性的角度来说, 操作系统需要能够支持不同种类的物理核心, 从灵活性的角度来说, 操作系统需要能够针对不同的应用提供合适的系统服务, 从性能角度说, 操作系统的性能直接影响到用户的使用, 因此操作系统必须在保证一定程度性能的前提下完成各种功能.

针对这些问题, 多内核 (multikernel) 操作系统是一种可行的设计理念. 多内核操作系统是一种在底层硬件上运行多个内核, 每个内核 (kernel) 各自维护一组内核数据的操作系统设计方案. 在诸如 Linux 等单内核操作系统中, 尽管每个核心 (core) 都运行自己的执行流, 但是它们操作的是同一组内核数据, 因此称之为单内核. 相较而言, 多内核操作系统则是分布式地运行了多个内核, 机器中的一部分核心运行其中一个内核, 维

① 收稿时间: 2021-06-30; 修改时间: 2021-07-30; 采用时间: 2021-08-12; csa 在线出版时间: 2022-02-21

© 中国科学院软件研究所 http://www.c-s-a.org.cn

护该内核的内核数据. 内核之间不直接共享内核数据, 而是通过其他的方式进行必要的同步.

多内核操作系统相比一般的单内核操作系统有着诸多的优势. 首先, 多内核操作系统可以为不同的核心选择不同的内核, 所以可以更容易地针对不同体系结构的核心进行扩展. 其次, 在特定的场景下, 分布式的设计也可以作为虚拟机的代替方案, 将整组机器中某一个核心, 分配给一个新的内核, 作为一个单独的操作系统或客户机进行使用. 最后, 多内核操作系统中内核之间尽量少地保留共享数据, 因此避免了在多核心之间进行频繁的数据同步, 避免了复杂的锁的设计, 在特定的情况下可以获得更高的性能.

与此同时, 多内核操作系统的设计也存在诸多的策略与考虑. 其中包括, 设计多内核操作系统的整体架构, 界定和规范一个内核中的数据, 配置每个内核所负责的物理资源, 设计内核与内核之间必要的通信与同步规则, 以及对新的硬件设备进行热启动和支持.

本文接下来首先对多内核操作系统进行介绍, 说明其出现的背景与应用场景. 然后介绍多内核操作系统设计上的考虑和难点. 接着, 本文将对现有的多内核操作系统进行分析. 而后对多内核操作系统与相关技术进行比对分析其优劣所在. 最后对多内核操作系统的研究和未来做总结.

2 研究背景与介绍

现代计算机硬件技术发展飞速, 多核以及异构架构的硬件日新月异. 相对地, 其上运行的软件却很难完美地跟上硬件更新的速度. 因此软件设计中适配性和可扩展性是十分重要的. 操作系统作为软件极为重要的一环也是如此. 操作系统的更新与扩展主要面临来自于两方面的压力: 越来越复杂的硬件与越来越复杂的应用场景.

从硬件的角度来说, 多核/众核体系结构逐渐流行起来, 随着核心数目的增加, 操作系统中核心间数据同步的问题也越来越严重. 由于不同核心共同维护同一组内核数据, 操作系统的设计者往往需要利用锁等方法对数据进行同步与保护, 这往往会带来很高的性能开销. 除了多核/众核架构下数据同步的问题, 硬件的多样性也会增加操作系统设计上的难度. 在不同的硬件上, 不同配置或设计策略的内核会产生不同的性能. 例如, 在一些 ARM 架构[1] 的硬件上, 同时存在多种不同性能的核心. 针对这些核心的不同, 在调度算法上面, 设计者会倾向于让计算能力更强的核心来负载需求高性能的任务, 而让计算能力稍弱的核心负责一些轻量型的任务. 对于一般的操作系统来说, 为了达到这样的目的, 需要在自身的代码里对各种硬件进行判断, 这样会增加内核的复杂度, 使内核的扩展越来越复杂.

从应用场景的角度来说, 不同的应用场景需要操作系统提供不同程度的系统服务. 如, 一些实时性任务会要求对任务时间进行精确的控制, 这要求操作系统不但能够支持普通的应用, 同时还需要为这些实时性任务提供更加高效且实时性的服务. 再比如, 数据中心厂商需要将众多硬件以不同的方式分配给不同的用户, 这就需要服务器的操作系统或管理系统能够利用诸如虚拟化等技术为不同的用户提供不同操作系统的客户虚拟机.

多内核操作系统的研究从某种角度上来说就是为了更好地解决以上这些问题. 多内核操作系统一般应用于多核/众核的硬件系统. 一般的单内核操作系统, 所有核心上的执行流维护的都是同一组内核的数据. 为了正确地维护这些数据, 以及并行执行某些系统服务, 会利用锁等机制来对数据进行同步, 这种同步会带来额外的性能开销. 另外, 面对不同物理硬件, 单内核操作系统虽然可以针对不同的核心进行专门的优化, 但随着硬件种类的增多, 系统会变得越来越臃肿. 相较而言, 多内核操作系统允许在一组机器中运行多个内核, 选择其中的某一个或某几个核心作为一组来运行一个特定的内核, 维护一组特定的内核数据, 而其他的核心以类似的方式负责另外的几个内核, 维护另外几组内核的内核数据. 在这样一种分布式的设计中, 由于每个内核是各自独立的, 因此可以大大减少不同内核间数据的同步, 并且可以为不同硬件上的内核专门准备独有的策略, 在进行扩展的时候也不会影响到其他内核的内部逻辑. 因此多内核操作系统拥有相比单内核操作系统更好的扩展性.

除扩展性以外, 多内核操作系统在应用上也具有很高的灵活性. 多内核操作系统可以作为实时操作系统的一种解决方案, 诸如 RT Linux[2–5], Xenomai[6,7] 等操作系统选择一种近似于多内核操作系统的双内核模式, 一个 Linux 内核用于一般的任务, 一个实时性内核用于实时性任务. 除了用于实时性操作系统, 多内核操作系统也可以作为虚拟化技术的一种替代方案. 虚拟

© 中国科学院软件研究所 http://www.c-s-a.org.cn

化技术[8–13] 是一类利用硬件或软件模拟计算机 硬件系统的技术, 用户可以将其模拟的硬件系统作为一个独立的机器进行使用. 相对的, 多内核操作系统中的一个内核由于具有较高的独立性, 因此也可以被客户视为一个单独的机器. 相比于虚拟化技术, 多内核操作系统中用户的内核直接运行在真实存在的硬件上, 而不是模拟出的硬件环境上, 并且多内核操作系统也避免了虚拟化技术中必要的中间层的设计, 减少了整体性能的损耗. 多内核操作系统在启动, 迁移和更新上也拥有很大的灵活性. 经过设计的多内核操作系统允许内核进行动态启动, 迁移和更新. 系统可以选择在不影响其他核心的前提下将一个核心上的内核迁移到新的核心上, 可以选择将空余核心分配给一个新的内核, 也可以对其上运行的内核进行更新. 在某个内核出现错误的时候, 可以避免影响到其他的内核而只对出现错误的内核进行重启或替换的操作.

除了灵活性和扩展性, 多内核操作系统根据设计的不同也可以增加整个系统运行的性能. 在单内核系统中, 因为要并行处理同一组内核数据, 往往会涉及到数据的同步问题. 一般来说, 由于存储层次的不同, 核心之间, 共享缓存层次越高, 数据同步代价就越小, 而共享缓存层次较低甚至只能直接通过内存进行同步数据的两个核心, 数据同步开销的代价就很大. 多内核操作系统可以选择让拥有更高层次共享缓存的核心负责处理一个内核, 拥有较低甚至没有共享缓存的核心负责不同的内核, 通过这样的方式, 就可以尽量减少受缓存一致性影响较大的核心之间的交互, 从而规避开销过大的数据同步.

3 多内核操作系统的设计

多内核操作系统的设计涉及到诸多方面, 接下来本文会从几个方面讨论多内核操作系统设计上的考虑和策略. 多内核操作系统的设计包括几个方面, 首先, 设计与规划每一个内核的数据, 其次, 确定物理硬件的资源分配与管理的方式, 接着, 考虑不同内核上的核心之间, 同一个内核上的核心之间, 以及运行在这些核心之上的用户线程应当以什么样的方式进行通信, 最后应当讨论内核的热启动, 迁移, 和更新.

3.1 内核数据的设计

单内核操作系统和多内核操作系统, 一个很重要的区别在于运行在核心上的各个执行流, 处理和维护的是否是同一组内核数据. 内核的数据一般包括核心的数据流自身运行需要的数据, 如每个核心拥有的栈, 用于维护整个内核运行或为用户提供服务的数据, 如页表, 文件表等. 单内核操作系统的内核数据只有一组, 而多内核操作系统则是其中的每一个内核都单独维护一组数据. 尽管如此, 多内核操作系统中一个内核维护的内核数据与单内核操作系统维护的数据没有什么本质上的不同. 图 1 和图 2 分别展示了单内核操作系统和多内核系统在物理硬件, 内核数据以及其上运行的应用的结构. 在单内核操作系统中, 所有物理核心维护同一组内核数据, 这些核心的执行流为同一个内核服务, 这个内核则利用这唯一的一组内核数据为其上运行的不同线程服务. 在多内核系统中, 核心被分成多个组, 每个组的核心维护一组内核数据, 形成一个内核, 这个内核为其上运行的线程提供服务. 不同组之间, 内核之间或其上运行的线程之间, 使用额外的设计来保证互相之间能够进行一定程度的通信.

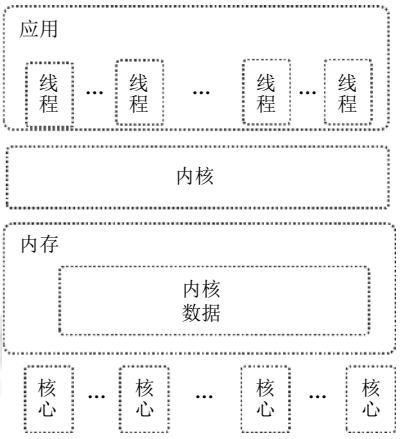

图 1 单内核操作系统架构

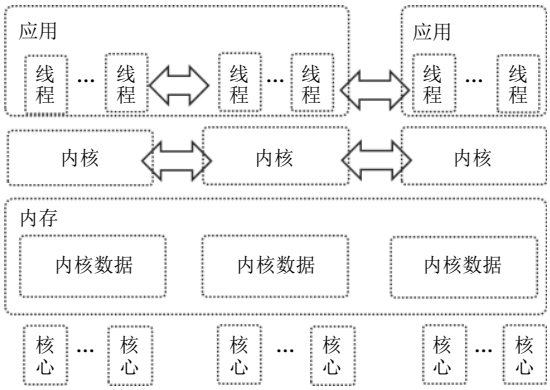

图 2 多内核操作系统架构

© 中国科学院软件研究所 http://www.c-s-a.org.cn

一般来说, 不论是单内核操作系统亦或是多内核操作系统, 在其中的一个内核中, 各个核心上执行流使用同一个内核页表, 拥有同一个地址空间, 因此也维护同一组内核数据. 在面对自然而然会出现的数据同步问题时, 内核会使用锁来进行保护, 对这方面的优化有很多, 包括但不局限于对于各种锁的使用以及将数据进行一定程度的分布式处理[14]. 但是总体上来说, 各个核心维护的依然是同一组数据. 这样做的结果, 就是在一个内核上运行的用户态的程序, 他们所能够看到的内核状态应当是一致的.

尽管多内核操作系统上不同内核在使用时可以被用户视为完全独立, 互不影响. 但是多内核操作系统依然要保障不同内核之间通信的功能. 首先, 几个内核很可能要共同使用同一组硬件, 诸如内存, 中断等, 那么它们之间就必须有某种程度上的同步, 使得他们能够共同使用这些硬件或是对这些硬件的使用权利进行协商. 一般的处理方案是, 维护一组专门的数据, 然后设计专门的通信方案来进行同步和更新. 例如, 可以选择每个内核保存一个副本, 然后通过设计一个共享的消息管道来进行同步, 或是选择将数据保存在一个共享的空间, 然后通过锁来访问和更新. 尽管这样的一段数据的维护是必要的, 但是多内核操作系统中, 这一部分被设计得尽可能的少, 以保证内核之间的独立性.

3.2 内核配置与资源管理

多内核操作系统的内核虽然是独立的, 但它们依然共同使用同一台机器上的硬件. 尽管理论上使用者可以按照自身意愿对内核进行配置和资源分配, 但是为了整个系统的性能与稳定, 依然需要一套专门的指导性方案.

首先需要考虑的是内核及其中数据的配置方式. 在一个内核的内部, 数据往往是通过锁来进行维护并在不同的核心之间同步的. 同步所需要的时间往往会影响到系统的性能, 常见的锁的方案包括大锁和细粒度锁, 前者使用一个全局的锁来维护所有共享的数据, 后者则会使用更细粒度的锁来进行维护. 从时间效率上说, 细粒度锁能达到更好的并行性, 但是设计相对复杂. 大锁设计上简单, 但是其相当于让不同的核心串行执行, 并行性较差. 多内核, 作为第 3 种方案, 选择每一个内核维护一组内核数据, 不同内核数据之间只在少数时间的时候通过诸如消息队列等方式进行同步, 既拥有良好的并行性, 设计上也不会过于复杂, 因此是一种相对合理的方案. 所以, 一个多内核操作系统应该是这样的: 其上的核心被划分到几个内核之中, 内核内部的核心通过大锁或细粒度维护数据, 内核之间的数据则只在需要的时候通过消息进行同步.

Peters 等人的研究[15] 指出, 对于一些特定的内核运行时间较少的内核, 如微内核[16,17] 等, 大锁相比于细粒度锁会拥有更好的性能. 其原因在于锁本身带来的代价甚至超出了并行带来的收益. 对于多内核操作系统来说, 其中一个内核对于大锁或者细粒度锁的选择可以是随意的, 微内核可以选择大锁, 而对并行性有更高要求的则可以选择细粒度锁.

确定了内核数据的配置方式后, 需要确定的是, 内核到物理核心的分配问题. 现代的多核硬件中, 往往会有不同层次的缓存, 对这些缓存的合理使用可以直接加快系统的运行速度. 在一个内核中, 当其中一个核心向内存中更新数据, 会通过缓存一层层写回到内存, 此时, 如果另一个核心和这个核心没有共享缓存的话, 想要获得更新后的数据就需要再从内存中将其读到自己的缓存中, 这会产生很大的开销. 而如果两个核心之间存在共享缓存, 那么就可以在数据真正写回到内存之前就通过共享缓存将其获取. 因此, 拥有共享缓存的核心能够在相对少的执行时间中完成数据的同步. 因此, 一个内核中核心数目的上限应当尽量由拥有共享缓存的核心数目来决定, 即, 有共享缓存的核心负责同一个内核, 没有共享缓存的核心负责不同内核.

至于一个内核核心数目的下限, 存在这样的方案: 每一个核心都为一个单独的内核服务. 在这种方案中, 每一个核心对应一个内核, 那么核心维护数据的时候就不再需要锁. 从某种角度来说, 这种方案并行度很高而且避免了锁和数据同步的代价. 但是, 在实际使用中, 有大量的用户应用是有多核运行的需求的, 单核心内核的模式会导致这类的应用频繁地使用多内核提供的跨内核通信的功能, 从而增加用户使用系统服务的复杂程度并增加内核间进行数据同步的次数.

对于其他的硬件, 诸如内存, 硬盘, 中断控制器等, 只要各个内核之间可以事前商定这些资源的使用, 多内核操作系统一般来说可以给予内核较大的自由.

3.3 多内核系统中的通信

多内核操作系统中的通信涉及到多个方面. 根据使用者可以分为内核层的通信和用户层的通信.

内核层的通信是指内核在维护内核数据时所做的

© 中国科学院软件研究所 http://www.c-s-a.org.cn

通信. 具体来说, 内核层的通信包括内核内的通信与内核间的通信. 内核内的通信指的是同一个内核中不同核心之间的通信, 这种通信理论上不能称之为通信, 仅是在需要时用锁等方式实现的一种数据的共享. 内核间的通信是指在不同内核之中的两个核心的通信, 这种通信一般用于处理一些内核之间需要共享的数据. 一种简单的方案是通过为两个内核设置相同的页表项, 从而设置一段共享的地址空间, 然后把这些共享的数据直接放到这个区间中, 通过锁的方式进行读取和更新, 或是在这段共享的区间中设计一个消息通道, 通过这种方式来进行消息通信. 总之, 内核层的通信只是为了满足在内核层进行一定程度的数据共享, 这种通信本质上只是一种代码上实现, 而不会体现成一组面向用户的规范的接口.

多内核操作系统不仅需要支持在其中某一个内核上运行的用户程序, 也需要支持同时运行在几个内核上运行的用户程序, 这就需要为用户提供一整套的用户层的通信方式. 用户层的通信一般是基于内核提供给用户的接口来完成. 以不共享地址空间的两个线程为例, 这样的两个线程由于不共享地址空间, 因此无法直接访问对方的数据. 因此需要调用一定的系统调用来进行通信. 此时, 分为两种情况, 两个线程在同一个内核上或在不同内核上.

如果这两个线程在同一个内核上, 那么其本质等于单内核操作系统中两个用户线程的通信. 这里有两个常见的方案: (1) 为两个线程申请一段共享的用户地址空间, 如 Linux 提供的共享内存的接口, 通过创建和修改页表项来实现. (2) 使用内核提供的进程间通信 (inner-process communication, IPC) 的接口. 进程间通信的实现方式有很多相关研究, 如 seL4[16] 的进程间通信, 为每个线程准备了一段独有的 IPC 缓存, 通信时将数据从一边的缓存移动到另一边. 除此之外 seL4 还专门对同核心上的两个线程之间的进程间通信进行了诸多优化, 如利用寄存器代替内存来进行信息传递.

不同内核上的两个线程的通信也是类似的. 对于不同内核的两个线程之间的通信大体上也是这两种方案: 一是通过创建和修改页表项来为两个线程准备一段共享的地址空间, 二是为用户线程准备一套完整的进程间通信的接口. 与同内核上的两个线程通信不同的是, 内核提供的这些功能需要能够做到跨内核. 如果要为两个不同内核的线程提供相同的地址空间, 首先要求两个内核都拥有分配该实际物理空间的权限, 然后两个内核各自将对应的物理地址写入到自己负责的用户页表的页表项中. 如果是要为两个不同内核上的线程提供进程间通信的接口, 可行的方案是由这两个内核约定一段共享区间, 然后将其实现为一个消息队列, 从而提供通信的功能.

3.4 热启动, 迁移与更新

对于多内核操作系统来说, 其内部的一个内核可以是现有的各种内核设计之一. 只要该内核能够遵循整个系统约定好的规范, 按照约定好的范围使用各种硬件资源, 那么它理论上就能够作为多内核操作系统的一部分, 多内核操作系统就能够依照这些规范对它们进行启动, 迁移和更新.

在一般的单内核操作系统的启动中, 以运行在 Intel 架构上的为例, 在机器通电之后, 会遵循诸如多重处理器规范等[18] 选择一个核心作为 BSP (bootstrap processor) 进行启动. 之后, BSP 会执行 BIOS 等引导程序 (bootloader), 遵循诸如 multiboot 的规范, 将执行流跳转到支持这些规范的内核代码处. 接下来, 由 BSP 运行内核代码进行更进一步的内核启动, 包括但不限于设置页表, 寻址模式等. 在时机成熟时, 它们会向其他尚未启动的核心, 这些核心被称之为 AP (application processor), 发送处理器间中端 (interprocessor interrupt), 使其他的核心的执行流跳转到它们需要跳转到的位置, 使这些 AP 能够参与到内核的启动之中.

多内核操作系统启动的第一步和一般的多核机器上单内核操作系统的启动是类似的. 具体来说, 多内核操作系统启动的第一个内核与单内核操作系统启动整个系统是类似的. 两者都是先启动一个 BSP, 然后启动其他的 AP. 区别在于, 多内核操作系统启动的 AP 只包括属于这个内核的核心. 在这样一个初始的内核启动后, 它将负责启动其他的内核. 为了方便表述, 本文称第一个启动的内核为主内核, 其他的内核为子内核. 在这里, 多内核操作系统往往会自己定义好一组规范, 定义诸多子内核运行所必须的内容, 如子内核可以使用的核心, 可以使用的内存, 可以使用的其他硬件资源以及必要的用于交互的共享空间. 主内核启动子内核的过程与 BSP 启动 AP 是类似的, 依然以 Intel 架构为例, 主内核中的一个核心, 向子内核中的 BSP 发送一个处理器间中端, 使得子内核的 BSP 跳转到自己内核的入口, 而规范定义好的数据的地址则可以作为一个参数

© 中国科学院软件研究所 http://www.c-s-a.org.cn

传递给子内核的 BSP. 主内核和子内核在启动之后本质上并没有什么不同, 理论上, 只要事先约定好权限, 启动新的子内核的可以是任何一个内核. 因此, 启动一个新的子内核, 这样一个行为完全可以设计成为一个系统调用, 交由用户程序来决定使用.

多内核操作系统中内核的迁移可能分为两种情况: (1) 将一个内核关闭, 然后将其上的任务迁移到其他的正在运行的内核上. (2) 将一个内核连带其上的任务迁移到其他的物理核心上. 这两种情况, 都可以划归为, 关闭一个内核, 将其上的任务迁移到一个新的内核上, 只不过对于前者来说, 这个新的内核是一个已经存在了的内核, 对于后者来说, 这个新的内核是一个与原来内核完全一致的内核. 对于一个内核来说, 关键的几项, 一是内核维护的内核数据, 二是内核所使用的硬件资源. 当系统对内核进行迁移时, 本质上是将内核维护的内核数据和使用的硬件资源, 通过数据拷贝或是传递指针的方式, 交给一个新的内核来维护, 至于这个新的内核, 是否使用原来的核心, 是否使用原来内核的代码, 都可以交由用户或操作系统自行决定.

内核的更新本质上就是对内核迁移的一种扩展, 即首先将一个内核上的数据迁移到其他内核, 或是单纯将其存储起来, 然后对原来的内核在内存中的代码进行更新, 然后将内核数据迁移回来.

4 现有的多内核操作系统

在本节中, 本文将对现有的多内核操作系统以及一些相关技术进行介绍和讨论. 现有的多内核操作系统的相关研究主要集中在, 多内核针对异构架构的设计, 多内核的迁移管理配置等方面, 除此之外, 还有一些将多内核系统的思想用于更广泛场景的研究.

4.1 Barrelfish

Barrelfish[19–22] 设计了一套相对完整的多内核操作系统. 该设计初衷主要为面向异构的硬件, 诸如一个同时包含多个不同种类核心的机器. 在 Barrelish 的设计中, 每一个物理的核心负责一个内核. 如图 3 所示, Barrelfish 将操作系统相关的职责分为两个部分, 位于内核空间的特权模式 (privileged-mode) 的 CPU 驱动 (CPU driver), 位于用户空间的用户模式 (user-mode) 的管程 (monitor), 前者相当于一般语境下的内核, 后者则类似于根线程.

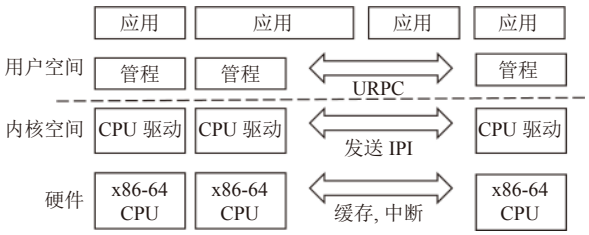

图 3 Barrelfish 架构图[19]

在数据上, Barrelfish 使用了 seL4 等微内核的设计策略, 把系统资源分为一个个的内核对象, 并把这些对象的引用 (在微内核里称作 capability) 以树的方式进行维护. 这样的一组内存数据, 被称作 OS node. 内核间的内核数据各自独立而不共享. 这些内核对象包含进程控制块, 页表等各种和用户息息相关的数据. 另外, 由于使用的是微内核的设计策略, 很多原本是操作系统负责的任务都变成了用户进程完成的任务. 例如, 用户需要自己调用系统去申请一个内存对象作为一个页表, 而内核会将这个页表的 capability 返回给用户让用户去使用. 这种内核的设计简化了内核本身, 使得内核在维护自身的数据时更加容易, 也为之后内核迁移的设计做下铺垫.

在通信方面, Barrelfish 中, 通信的双方是用户态的两个线程. 具体来说, 如果是同一个内核上的两个用户线程的通信, 则他们会通过一般的微内核的 IPC 进行, 如果这两个线程在不同的内核上, 它们会建立一段共享内存以作为跨内核通信的通道. Barrelfish 将这种跨内核的通信称为 URPC. 为了能够提供完整的 URPC 功能, 需要底层 CPU 驱动提供一些必要的功能, 如发送 IPI 的功能, 即由一个核心向另一个核心发送处理器间中断.

Barrelfish 允许动态地启动一个新的内核, 他们设计了一组 boot driver, 专门用于启动新的内核. 首先 Barrelfish 要求有一个已经启动的主内核, boot driver 运行在这个主内核上. 启动一个新的目标内核的顺序如下: 首先, 利用机器自身的机制 (如 ACPI) 检测新的核心, 然后, boot driver 为这个新的核心选择一个合适的内核, 按照核心启动的协议对核心进行启动, 将内核的代码加载到相应的位置, 并将目标核心的执行流跳转到内核入口, 然后目标核心就可以继续自己内核的初始化等诸多事项.

Barrelfish 同时也设计了一整套的内核更新与迁移的算法, 其主要方式为, 将待更新内核的 OS node 转交

© 中国科学院软件研究所 http://www.c-s-a.org.cn

给其他的内核, 然后待目标内核更新完毕后, 将 OS node 转移回来.

总体而言, Barrelfish 是多内核操作系统的一次很好的尝试. 在各个方面都提出了很多值得借鉴的方法.

4.2 LegoOS

LegoOS[23] 是 Purdue University 开发的一种名为 Splitkernel 的操作系统, 这种设计与一般多内核设计类似, 但是 LegoOS 能够适应更广泛的应用场景. LegoOS 的研究认为, 多内核操作系统针对的是单内存的多核心机器, 虽然其中的每个内核可以被配置于不同的物理核心上, 但是这些内核依然使用的是同一组内存. 而 LegoOS 扩展了使用的场景, 使它能够适应于更大型的服务器.

一般来说, 一个服务器系统包含的机器组包括多种核心, 多个不同种类的内存, 以及多个不同种类的硬件存储, 这些硬件往往通过网络的方式连接起来. LegoOS 将这些硬件资源分为 3 种组件: 核心, 内存, 硬盘. 在这样的情况下, 每一个内核的结点可以选择运行在任意的几个组件之上, 而任意一个组件也可以同时为几个内核进行服务. 相比一般的多内核有着更多的扩展性. 为了实现这种分布式服务器上的多内核系统, LegoOS 还有专门负责管理所有资源组件的全局的管理器. 相比一般的多内核, LegoOS 可以适用于不共享内存的机器组, 前者内核间的通信方式可以通过共同使用的内存, 但是后者就只能通过网络传输来进行通信. 因此 LegoOS 使用了诸如 RDMA 等相关技术来对网络通信进行优化.

4.3 基于双内核的实时操作系统

实时操作系统是一种能够快速接受和处理任务并能在规定时间内产生处理结果的一类操作系统, 其主要特点为拥有高实时性与可靠性. RT Linux 与 Xenomai 就是两个典型的面向实时的操作系统, 且两者都使用了“双内核”的设计思路.

RT Linux[2–5] 是一种基于 Linux 的实时操作系统, 如图 4 所示, 在 RT Linux 中, 存在一个直接运行在硬件上的 RT Linux 内核. 该内核为面向实时应用的内核, 直接负责管理硬件以及为实时应用提供服务. 在 RT Linux 内核中, 存在有调度器, 其负责调度实时应用和一般的 Linux 内核, 而一般的 Linux 内核则负责管理运行非实时性的任务. 在运行时, 实时应用可以直接使用硬件的中断并与硬件进行 IO 交互, 一般的应用则需要首先通过 Linux 内核, Linux 内核再通过 RT Linux 内核来与硬件进行 IO 交互. 在 RT Linux 系统中, 尽管存在这两个内核, 但是一般性的 Linux 内核更类似于 RT Linux 上运行的一个额外的应用, 与其他的实时性任务共同参与调度. 这样的设计保证了 RT Linux 能够通过调度提高实时性应用的优先级, 从而使系统能够为实时任务提供精确的时间控制以及服务.

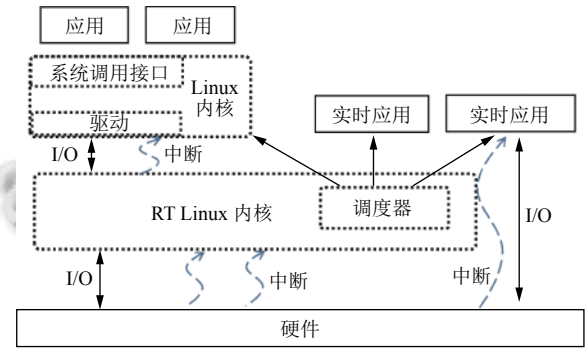

图 4 RT Linux 架构

Xenomai[6,7] 是一种实现了双内核机制的实时操作系统, 与 RT Linux 不同的是, 其中的用于一般应用的 Linux 内核与用于实时任务的内核都处在同一个级别, 参与调度. 如图 5 所示, Xenomai 使用了 i-pipe 的技术, i-pipe 负责中断的分发与传递. 通过设计中断的分发与传递规则来保证不同内核的优先级的不同. 举例来说, 当一个优先级高的实时内核注册了一个中断时, i-pipe 会立刻进行处理, 而遇到一个优先级低的 Linux 内核的中断时, 它会等到实时内核使用完物理硬件之后才触发中断运行 Linux 的内核.

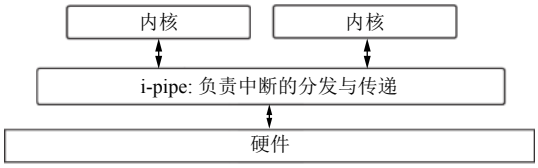

图 5 Xenomai 架构图

RT Linux 和 Xenomai 等实时操作系统与 Barrelfish 等系统有比较大的区别, 首先, RT Linux 和 Xenomai 都仅支持双内核, 并不等同于一般意义上的多内核操作系统. 其次由于实时操作系统主要面向于实时应用, 因此实时内核的部分一般占有主要的地位. 当实时应用需要的时候, 物理核心就会被实时内核所抢占, Linux 内核就会被调度出去, 换成实时内核或应用来运

© 中国科学院软件研究所 http://www.c-s-a.org.cn

行. 相较而言, Barrelfish 尽管也支持内核在物理核心间的转移, 但是其中各个内核依然是相对平等的.

4.4 相关技术

除去上文提到的多内核操作系统和与多内核操作系统相近的双内核操作系统, 还有一些操作系统相关研究, 虽然这些研究本身并不是多内核操作系统, 但是这些研究有的为多内核操作系统的发展提供了思路, 有的为多内核操作系统提供了可行的技术.

Corey[14] 是一项针对 Linux 系统的研究, 该研究分析了操作系统数据同步存在的瓶颈, 并针对该问题进行了解决. Corey 提出了分布式的设计, 将内核数据分为共享部分和非共享部分, 以此来减少同步带来的开销. 该理念也是多内核操作系统采用的方法的核心, 区别在于多内核操作系统是一种更极端的方式, 它几乎完全摒弃了共享数据的部分.

Theseus[24] 是基于 Rust 语言实现的操作系统, 其利用 Rust 语言的特性, 将内核数据分成一个个极小的组件, 每一个组件被对应到 Rust 的 crate, 一种独立的编程单元. Rust 在编译时就可以对内核数据的每个组件对应的内存进行追踪和保护. 将内核数据组件化是一种非常有效的方式, 该思路往往也会被应用到多内核操作系统中, 组件化的内核数据将更容易进行动态地加载与迁移.

PTask[25] 是一种针对 GPU 异构架构的研究. PTask 对操作系统进行了扩展. 使其能够将 GPU 也纳入操作系统的控制范畴中. 在此之前, GPU 仅仅是一个外部设备, 用户通过 IO 与 GPU 通信. PTask 将 GPU 变成了操作系统的一部分, 相当于一个特殊的核心, 加入到了调度之中. 多内核操作系统也有类似的设计理念, 即将各种物理核心都作为一种可分配的资源提供给各个内核, 而 GPU 与 CPU 同样, 也是可分配的核心资源之一.

5 多内核操作系统与虚拟化技术的优劣势分析

多内核操作系统与虚拟化技术都是能够在某种层面上解决多核/众核, 异构架构上扩展性, 性能稳定性, 以及资源分配的设计. 但是两者还是有着明显的不同.

如表 1 所示, (1) 多内核操作系统中的每个内核真实运行在单独的一部分的硬件上, 相对而言, 虚拟化技术中一个客户操作系统可能运行在一组被模拟出来的硬件上, 而模拟的硬件一般都会带来一定程度的性能损耗. (2) 多内核操作系统中的内核由于直接运行在硬件之上, 因此不支持跨平台执行. 相较而言, 虚拟化技术允许跨平台执行, 如一些虚拟化技术允许 ARM 架构的程序运行在 x86 宿主机上, 而多内核操作系统中, ARM 的程序只能在 ARM 的核心上执行. (3) 多内核操作系统尽管需要一定程度的内核管理, 但是这种管理的功能一般交由其中一个内核来进行, 相比而言, 虚拟化技术一般需要一层中间的管理层来负责不同虚拟机之间的切换和使用. (4) 在内存隔离方面, 一些虚拟化技术使用了辅助页表的方式为客户提供一个完整的虚拟地址空间, 但是这个空间映射到真实物理地址时可能并非连续, 这样的设计会导致缓存的性能降低. 而多内核操作系统会将真实的连续的物理内存直接分配给某个内核, 因此不会增加缓存的开销.

表 1 多内核操作系统与虚拟化技术的比较

项目	多内核操作系统	虚拟化技术
硬件与内核对应关系	内核构建在真实存在的硬件上	内核构建在模拟的虚拟硬件上
跨平台	不同平台的程序不能运行	不同平台的程序可以运行在模拟的硬件环境上
中间层	不需要中间层	可能需要Hypervisor管理层
内存资源的隔离	独占的连续物理内存	借助辅助页表的离散分布的内存

从这些比较可以看出, 多内核操作系统对于虚拟化技术互有优劣, 在技术选择时, 多内核操作系统可以作为替代虚拟化技术的技术选择.

6 总结与展望

多内核操作系统是面对多核异构系统的一种实现方式, 它从某种程度上解决了异构系统不同硬件的适配问题. 相比单内核操作系统, 多内核操作系统拥有更高的灵活性, 适应性, 容错性. 如果设计的足够合理, 能够正确地使用各项资源, 相比单内核操作系统, 也会有相近甚至较好的性能.

从未来发展的角度考虑, 为了让多内核操作系统能够适用于更多硬件以及更多现有软件, 需要为现有的单内核进行多内核化的改进. 然而现有的单内核设

© 中国科学院软件研究所 http://www.c-s-a.org.cn

计各有不同的宗旨, 提供的 API 也各有差异. 为了让不同硬件, 不同的内核, 使用不同 API 的软件都集合在一套多内核系统里, 一套专门的权威的多内核系统的规范是必要的, 而这样的过程很可能是漫长而充满挑战的.

总而言之, 多内核操作系统还有很多路要走, 虽然研究不乏少数, 但是依旧还有发展的空间.

参考文献

1 Greenhalgh P. Big. LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. White Paper, ARM Ltd., 2011.

2 Yodaiken V, Barabanov M. A real-time Linux. Linux Journal, 1997, 34(1): 31–33

3 Yodaiken V. The RTLinux manifesto. Proceedings of the 5th Linux Expo. Raleigh, 1999.

4 Yodaiken V. Cheap operating systems research. Proceedings of the 1st Conference on Freely Redistributable Systems. Cambridge, 1996.

5 Barabanov M. A Linux based real time operating system [Master's Thesis]. New Mexico: New Mexico Institute of Mining and Technology, 1997.

6 Gerum P. Xenomai-implementing a RTOS emulation framework on GNU/Linux. White Paper, Xenomai, 2004: 81.

7 Ugal AS. Hard real time Linux using Xenomai on Intel® multi-core processors. White Paper, Intel, 2009.

8 《虚拟化与云计算》小组. 虚拟化与云计算. 北京: 电子工业出版社, 2009.

9 林芊. 基于混合虚拟化技术的虚拟机性能优化研究及应用 [硕士学位论文]. 上海: 上海交通大学, 2011.

10 英特尔开源软件技术中心, 复旦大学并行处理研究所. 系统虚拟化: 原理与实现. 北京: 清华大学出版社, 2009.

11 Pratt I, Fraser K, Hand S, *et al*. Xen 3.0 and the art of virtualization. Proceedings of the 2005 Ottawa Linux Symposium. Ottawa, 2005.

12 广小明, 胡杰, 陈龙, 等. 虚拟化技术原理与实现. 北京:电子工业出版社, 2012: 96–100.

13 金海, 廖小飞. 面向计算系统的虚拟化技术. 中国基础科学, 2008, 10(6): 12–18. [doi: 10.3969/j.issn.1009-2412.2008.06.002]

14 Boyd-Wickizer S, Chen HB, Chen R, *et al*. Corey: An operating system for many cores. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 43–57.

15 Peters S, Danis A, Elphinstone K, *et al*. For a microkernel, a big lock is fine. Proceedings of the 6th Asia-Pacific Workshop on Systems. Tokyo: Association for Computing Machinery, 2015. 3.

16 Klein G, Elphinstone K, Heiser G, *et al*. SeL4: Formal verification of an OS kernel. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky: Association for Computing Machinery, 2009. 207–220.

17 Elphinstone K, Heiser G. From L3 to SeL4 what have we learnt in 20 years of L4 microkernels? Proceedings of the 24th ACM Symposium on Operating Systems Principles. Pennsylvania: Association for Computing Machinery, 2013. 133–150.

18 Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual: Volume 3 (3A, 3B, 3C & 3D): System programming guide. Intel Corporation, 2016.

19 Baumann A, Barham P, Dagand PE, *et al*. The multikernel: A new OS architecture for scalable multicore systems. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky: Association for Computing Machinery, 2009. 29–44.

20 ETH Zürich. The Barrelfish operating system. http://www.barrelfish.org/. (2020-03-23).

21 Zellweger G, Gerber S, Kourtis K, *et al*. Decoupling cores, kernels, and operating systems. Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2014. 17–31.

22 Menzi D. Support for heterogeneous cores for Barrelfish [Master's Thesis]. Swiss: ETH Zürich, 2011.

23 Shan YZ, Huang YT, Chen YL, *et al*. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 69–87.

24 Boos K, Liyanage N, Ijaz R, *et al*. Theseus: An experiment in operating system structure and state management. Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. Banff: USENIX Association, 2020. 1.

25 Rossbach CJ, Currey J, Silberstein M, *et al*. PTask: Operating system abstractions to manage GPUs as compute devices. Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais: Association for Computing Machinery, 2011. 233–248.

© 中国科学院软件研究所 http://www.c-s-a.org.cn