用户态调度相关工作调研

**可配置内核调度**

可配置内核调度是指在内核中提供灵活的机制，使用户能够根据具体需求自定义和扩展调度策略。[1]提出了一个针对软实时的保证系统，该系统能针对分层调度提供了确定性的保障，其中每个层次的分层调度算法支持任意调度算法。[2]通过可插拔策略的API来定制内核的的调度策略。尽管这些设计体现了调度策略定制的高效性，但是这些调度策略都是在内核空间中实现的，没有考虑到与应用程序进行紧密的协同设计。Enoki[3]是一个用于快速开发Linux内核调度器的框架, 支持将新的调度策略实时升级到内核中，支持用户空间调试，并且与应用程序之间具有双向通信功能,实现了跨应用程序和调度器的细粒度核心共享。Plugsched[4]以模块化的思想，将调度器与Linux内核解耦为一个独立的模块，并通过使用数据重建技术将状态从旧调度器迁移到新调度器。该方案可以直接应用于生产环境中的Linux内核调度器，而无需修改内核代码。[5]提出了一个Linux内核调度扩展的概念验证，该扩展设计为用户监控线程（UMT），允许用户空间进程在所选线程被阻塞或解除阻塞时收到通知，从而使运行时可以在空闲核心上安排额外的工作。[6]提出了一种内存感知的公平份额调度算法，该调度方法精心将运行任务的CPU周期与真正的内存相关停滞分离开来，并对任务进行补偿，使其在为时已晚之前获得所需的CPU份额。所提出的方法是自适应、有效且高效的，不依赖于任何静态分配或内存硬件资源的分区，并且仅具有可以忽略不计的运行时开销就能提高QoS应用程序的性能。此外，该调度设计是一种仅基于软件的解决方案，仅需对内核进行最小的修改，就可轻松地集成到内核调度器中。

**用户级调度**

用户级调度是指调度和管理线程的机制在用户空间（应用程序级别）中实现，而不是在内核空间中实现。例如，[7]在进程内，使用并发管理来管理保护域的线程。用户级调度支持高效的线程调度，但是需要考虑阻塞系统调用并与阻塞系统调用集成。[8]中定义了一个基于阻塞系统调用使用动态绑定内核到用户级线程的协议，然而，该协议需要在内核级和用户级线程库之间进行复杂的交互，增加了其复杂性。用户级的调度只能在保护域内起作用，这些方法不能调度其他保护域中的线程。同时，操作系统内核并不直接感知或控制这些用户级线程，而是将它们视为单一的进程来管理。特别是遇到阻塞操作时，无法利用内核的调度机制进行线程切换，可能导致资源浪费。传统的线程调度时，会为应用的每个请求创建一个内核线程，将应用进程在CPU上进行调度，这样导致系统资源利用率低下，而且会造成应用程序之间的资源的竞争。[9]实现了一个用户级线程调度方案（Arachne），它能依据负载对应用的线程进行CPU核心的调度, 让应用程序感知到所需的CPU核心数，并独占核心。同时，为每个核心创建的一个内核进程，消除与其他应用的资源竞争。通过对应用程序的线程依据负载在CPU核心上进行调度，以优化应用的性能。ghOSt[10]基于Linux内核之上构建了高性能的框架，该框架提供了丰富的API，可以从用户空间接收进程的调度决策，并将其作为事务执行。从而实现了将线程调度策略委派给用户空间进程，并进行了相关的策略优化、无中断升级和故障隔离。LibPreemptible[11]是一个灵活、轻量级且可扩展的抢占式用户级线程库。通过提供一种用于传递定时中断的快速轻量级硬件机制、通用用户级调度接口和为可供用户开发使用的API，该用户级线程库实现了根据应用程序需求进行自适应调度的调度策略。

**中间件调度**

中间件调度是一种调度机制，它通过利用内核API和优化调度策略，在内核和用户级别进行线程的优先级管理和资源调度，旨在在不同的保护域之间有效分配计算资源，处理调度阻塞，并简化用户级调度器的设计。[12]和[13] 通过显式分配优先级，以便只有单个目标线程具有最高优先级。通过巧妙地利用现有的内核策略和抽象，这使得这种方法即使在调度程序的保护域之外也能对线程进行调度。然而，这种方式存在以下缺陷：（1）不仅需要设计到内核的调度器，还存在双别内核上下文切换（用户态切换到内核态线程，内核态切换新的用户级的线程）和用于优先级管理的API的开销。(2)当线程进行系统调用时发生阻塞，唤醒该线程的时间不确定，这给调度器准确跟踪线程的状态带来挑战。[14]提出了一种两级调度机制，其中内核负责为应用程序分配处理器，而应用程序则负责自己的线程调度。这种机制允许应用程序在内核层面和用户空间层面之间进行交互，以更有效地管理线程的执行。

**并行系统调度**

并行系统调度是指在多核系统中有效管理和调度并行任务的过程，以最大化地利用多核处理器的性能。并行系统调度的主要目标是提升系统的并行性和同步能力，同时保证高效利用计算资源，避免任务间的阻塞和死锁。实时系统中，实时系统中主要适用OpenMP来提高程序的运行时的并行性能。在这种运行时中，[15], [16], [17] 为实时处理调度并行任务提出了一系列独特的挑战。这是由于，为了最大限度地利用大量内核，并行运行时将计算分解为细粒度的任务，这些任务非抢先执行，默认情况下在较少数量的系统线程上运行至完成。它们通常在任务执行延迟其进度的操作（例如，等待另一个任务完成）时定义调度点，此时运行库在同一线程上运行挂起的任务。OpenMP存在以下的特征，使系统在分析性地满足截止日期的同时保持高利用率的能力变得复杂且困难：（1）调度任务有效阻塞的点，（2）线程上下文被重用以执行挂起的任务的点，（3）对任务内共享数据结构的访问使用临界段原语，以及（4）任务非抢先执行。如果拥有关键部分的任务到达调度点，并且同一个线程执行试图进入相同关键部分的任务，则很容易发生死锁。OpenMP使用运行时提供的“avoidance”来解决这个死锁。为了避免这种死锁，默认情况下，所有任务都是绑定任务[18], [19]，这些任务限制运行时执行可能试图占用如此关键部分的任务,然而，这种情形下容易触发任务执行的最坏情况。在[20]中，提出了一种适用于在分层架构上执行具有不规则和大规模嵌套并行性的OpenMP程序的线程调度策略。该线程调度策略强制执行线程分配，最大化属于同一并行区域的线程，并在需要负载平衡时使用了一个NUMA感知的工作策略。

**用户级系统调度**

用户级系统调度允许用户级别对跨多个保护域的系统级别线程进行控制的调度机制。[21] 开发了一种用于 L4 微内核的层次化用户级调度架构，该架构将调度的功能转移到用户级，以提高调度的灵活性，当内核遇到调度不明确的情况时，引入用户决策。[22] 介绍了Composite组件系统中的用户级调度层次设计，其动机是为了创建一个既可靠又可预测的系统，并能根据特定应用需求进行配置。其设计目标是允许不受信任的开发者在保护域内安全地开发服务和策略，同时确保用户空间服务的时间控制，及时处理异步事件。其关键特性包括用户定义的调度策略、高效的异步事件处理和上行调用机制，以及对共享数据结构访问的同步管理。[23] 介绍了一种名为CPU继承调度的新型处理器调度框架，在该框架中任意线程可以作为其他线程的调度器，支持在一个系统中实现多种不同的调度策略，从而提供更大的调度灵活性。该框架允许模块化和分层控制处理器在不同管理域（如进程、作业、用户和组）中的使用，并准确地分配和记录CPU资源。应用程序和操作系统可以实现定制的本地调度策略，并通过框架确保所有不同策略的逻辑性和可预测性。作为附带效果，框架还通过提供一般形式的优先级继承机制来解决优先级反转问题。CPU继承调度自然地扩展到多处理器，支持处理器管理技术，如处理器亲和性和调度器激活，并且可以在典型环境中以可接受的开销提供这种灵活性。这些调度系统[21], [22], [23]是为系统级线程的用户级控制而设计的（即跨多个保护域）。它们支持线程阻塞和唤醒事件向调度器的矢量化[21], [22]。然而，这种方法需要线程调度的内核的介入，这也会带来额外的开销。

**分层调度**

分层调度通过在用户级实现结构化的调度层次，允许在子系统中对中断和传统线程调度进行管理。该机制支持在每个子系统内构建独立的调度策略，同时保证子系统与内核之间的隔离，从而提供低成本、可靠且可预测（可扩展）的系统。[21], [23]都提出了实现分层结构的用户级调度器的机制。此外，[13]认为用户级调度对实时系统是有用的，并提供了在中间件设置中适应它的方法。这些工作都没有试图从内核中删除所有阻塞和调度的概念。此外，这些方法没有提供在不求助于昂贵的调度器调用的情况下调度和记帐异步事件（例如，中断）的机制。[24] 探讨了如何在调度器的层次结构中提供组合硬实时保证的问题。该论文绍了一种组合方法，该方法通过子调度器的时间需求推导出父调度器的时间需求，确保只有在子调度器的时间需求得到满足时，父调度器的时间需求也能得到满足。[25] 提出了一种运行时算法，使得即使在对手完全控制时间的情况下，各分区也无法感知到其他分区的行为变化。这种方法支持使用动态时间分区机制，既提高了响应能力，又保证了算法级别的非干扰性，与静态方法的效果相当。时间可预测性是成功通过隐蔽时间信道进行通信的前提条件。实时系统特别容易受到时间信道的影响，因为实时应用程序由于时间分区的不确定性受到限制，往往会具有时间局部性。在[26]中，展示了即使在时间分区之间严格执行时间隔离的情况下，实时应用程序也可以创建隐藏的信息流。并且引入了一种在线算法，该算法对时间分区的时间表进行随机化，以减少时间局部性，同时保证时间分区的可调度性和时间隔离。

**用户态调度相关应用场景**

[27] 介绍了一个基于多核Linux用户态实时多任务调度框架ULight。ULight包括三个核心模块：多任务调度模块、定时器模块和用户态中断处理模块。这其中，多任务调度模块为Linux用户态提供基本的实时任务调度功能，定时器模块提供高精度的定时服务，用户态中断处理模块通过在内核态和用户态之间构建中断处理的快速通道，实现用户态任务直接处理硬件中断。[28] 针对数据中心负载中混杂请求对延迟敏感型请求响应尾延迟产生干扰的问题，提出了一种应用定义优先级调度的用户态协议栈。该设计利用用户态协议栈在数据中心请求处理中的关键位置，允许上层应用根据负载特征灵活定制优先级识别与调度策略，并通过协议栈提供的数据包和TCP流等状态信息，实现无需修改网络协议栈即可对不同负载场景进行灵活的优先级识别与调度，从而避免延迟敏感型请求受到其他混杂负载请求的干扰。针对多对多的线程模型失去了内核提供的隐式操作系统抢占功能及其线程必须显式地让出控制权的缺点，[29]提出了两种解决的方案：信号-让出和内核线程切换，以实现隐式抢占，提升处理器的核心利用率。

在云计算场景下，现代云应用程序容易出现高尾部延迟，因为它们的请求通常遵循高度分散的分布。为了减轻尾部延迟，现有的工作都是基于用户空间的调度和系统级调度相互结合的方法来进行优化。[30] 提出了一种名为LibPreemptible的抢占式用户级线程库，通过三种关键技术（快速轻量的定时中断硬件机制、通用的用户级调度接口、用户定制的自适应调度策略API）来减少微秒级工作负载的尾延迟并提高吞吐量。这种技术无需修改内核，灵活轻量且可扩展，能够适应不同负载水平和特性的实际应用。[31] 提出了一种名为SFS（智能功能调度器，Smart Function Scheduler）的新方法，用于优化无服务器计算环境中云函数的执行。SFS完全在用户空间工作，通过结合新的FILTER策略与Linux的CFS调度器，采用两级调度机制，实现接近最短剩余时间优先（SRTF）的调度。[32] 提出了Syrup，一个用户定义调度的框架，解决操作系统、网络栈和应用运行时缺乏对应用和请求的深入了解以及各层之间缺乏协调导致的次优调度决策问题。Syrup允许不受信任的应用开发者在不需要修改代码的情况下，跨系统层表达特定于应用的调度策略。开发者使用Syrup编写调度策略，将输入（线程、网络数据包、网络连接）与执行器（核心、网络套接字、NIC队列）匹配，并安全地部署在多个共存的应用上。

针对syscall密集型的应用程序，他们在执行的过程中会出现频繁的上下文切换开销。为了加速这类应用程序的执行，[33] 提出了一种用户空间绕过（Userspace Bypass，UB）的方法，通过透明地将用户空间指令移动到内核中来加速系统调用密集型应用。为了避免频繁系统调用带来的开销，内核识别出连续系统调用之间的短用户空间执行路径，并将这些路径中的指令转换为具有软件故障隔离（SFI）保障的代码块。

[1] J. Regehr and J. A. Stankovic, “HLS: a framework for composing soft real-time schedulers,” in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, London, UK: IEEE Comput. Soc, 2001, pp. 3–14. doi: 10.1109/REAL.2001.990591.

[2] B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems”.

[3] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High Velocity Linux Kernel Scheduler Development. In Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24). Association for Computing Machinery, New York, NY, USA, 962–980. https://doi.org/10.1145/3627703.3629569

[4] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. 2023. Efficient Scheduler Live Update for Linux Kernel with Modularization. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 194–207. <https://doi.org/10.1145/3582016.3582054>

[5] A. Roca, S. Rodríguez, A. Segura, K. Marquet and V. Beltran, "A Linux Kernel Scheduler Extension for Multi-core Systems," 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, 2019, pp. 353-362, doi: 10.1109/HiPC.2019.00050.

[6] Kim J, Shin P, Kim M, et al. Memory-aware fair-share scheduling for improved performance isolation in the Linux kernel[J]. IEEE Access, 2020, 8: 98874-98886.

[7] M. S. Mollison and J. H. Anderson, “Bringing theory into practice: A userspace library for multicore real-time scheduling,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA: IEEE, Apr. 2013, pp. 283–292. doi: 10.1109/RTAS.2013.6531100.

[8] Anderson T E, Bershad B N, Lazowska E D, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 53-79.

[9] Qin H, Li Q, Speiser J, et al. Arachne:{Core-Aware} thread management[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 145-160.

[10] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[11] Li Y, Lazarev N, Koufaty D, et al. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling[C]//2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2024: 922-936.

[12] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, “Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time,” in Proceedings of the Thirteenth EuroSys Conference, Porto Portugal: ACM, Apr. 2018, pp. 1–16. doi: 10.1145/3190508.3190539.

[13] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill, “Design and Performance of Configurable Endsystem Scheduling Mechanisms,” in 11th IEEE Real Time and Embedded Technology and Applications Symposium, San Francisco, CA, USA: IEEE, 2005, pp. 32–43. doi: 10.1109/RTAS.2005.17.

[14] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new OS architecture for scalable multicore systems[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 29-44.

[15] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu, “Randomized Work Stealing for Large Scale Soft Real-Time Systems,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Porto, Portugal: IEEE, Nov. 2016, pp. 203–214. doi: 10.1109/RTSS.2016.028.

[16] D. Ferry, Jing Li, M. Mahadevan, K. Agrawal, C. Gill, and Chenyang Lu, “A real-time scheduling service for parallel tasks,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA: IEEE, Apr. 2013, pp. 261–272. doi: 10.1109/RTAS.2013.6531098.

[17] Q. Wang and G. Parmer, “FJOS: Practical, predictable, and efficient system support for fork/join parallelism,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany: IEEE, Apr. 2014, pp. 25–36. doi: 10.1109/RTAS.2014.6925988.

[18] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, “Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Paris: IEEE, Dec. 2017, pp. 92–103. doi: 10.1109/RTSS.2017.00016.

[19] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, “Timing characterization of OpenMP4 tasking model,” in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Amsterdam, Netherlands: IEEE, Oct. 2015, pp. 157–166. doi: 10.1109/CASES.2015.7324556

[20] Broquedis F, Diakhaté F, Thibault S, et al. Scheduling dynamic OpenMP applications over multicore architectures[C]//OpenMP in a New Era of Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings 4. Springer Berlin Heidelberg, 2008: 170-180.

[21] J. Stoess, “Towards effective user-controlled scheduling for microkernel-based systems,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 59–68, Jul. 2007, doi: 10.1145/1278901.1278910.

[22] G. Parmer and R. West, “Predictable Interrupt Management and Scheduling in the Composite Component-Based System,” in *2008 Real-Time Systems Symposium*, Barcelona, Spain: IEEE, Nov. 2008, pp. 232–243. doi: 10.1109/RTSS.2008.13.

[23] B. Ford and S. Susarla, “CPU Inheritance Scheduling”.

[24] Insik Shin and Insup Lee, “Periodic resource model for compositional real-time guarantees,” in *Proceedings. 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748)*, Cancun, Mexico: IEEE Comput. Soc, 2003, pp. 2–13. doi: 10.1109/REAL.2003.1253249.

[25] M.-K. Yoon, M. Liu, H. Chen, J.-E. Kim, and Z. Shao, “Blinder: Partition-Oblivious Hierarchical Scheduling”.

[26] Yoon M K, Kim J E, Bradford R, et al. Timedice: Schedulability-preserving priority inversion for mitigating covert timing channels between real-time partitions[C]//2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2022: 453-465.

[27] 张旭;顾乃杰;苏俊杰;, “一种Linux用户态实时多任务调度框架,” 中国科学技术大学学报, no. 08 vo 47, pp. 635–643, 2017.

[28] 沈逸凡;张文力;刘珂;陈明宇;, “面向应用定义优先级调度的用户态协议栈研究,” 高技术通讯, no. 08 vo 33, pp. 781–790, 2023.

[29] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji, “Lightweight preemptive user-level threads,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Virtual Event Republic of Korea: ACM, Feb. 2021, pp. 374–388. doi: 10.1145/3437801.3441610.

[30] Y. Li *et al.*, “LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling”.

[31] Y. Fu, L. Liu, H. Wang, Y. Cheng, and S. Chen, “SFS: Smart OS Scheduling for Serverless Functions,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA: IEEE, Nov. 2022, pp. 1–16. doi: 10.1109/SC41404.2022.00047.

[32] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, “Syrup: User-Defined Scheduling Across the Stack,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, Virtual Event Germany: ACM, Oct. 2021, pp. 605–620. doi: 10.1145/3477132.3483548.

[33] Z. Zhou, Y. Bi, J. Wan, Y. Zhou, and Z. Li, “Userspace Bypass: Accelerating Syscall-intensive Applications,” 2023.