IoT-EDF: 基于 Unikernel 的物联网任务调度方法

董博南[1,2], 杨秋松[1], 李明树[1]

[1] 中国科学院软件研究所 基础软件国家工程研究中心 北京 中国 100190
[2] 中国科学院大学 北京 中国 100049

摘要 Unikernel 作为虚拟化领域的前沿技术, 在物联网环境中因其启动快速和低资源消耗以及高安全性的特点而被广泛应用。然而目前 Unikernel 缺乏根据不同任务特点所设计的动态调度机制, 用以保障物联网大规模任务调度场景下的工作效率。为解决这一问题, 首先总结了物联网环境下 Unikernel 的任务特点, 在此基础上, 提出一种新的基于 Unikernel 的物联网环境下的调度机制 IoT-EDF 及其数学模型, 该机制通过划分任务的重要性和截止时间, 不仅有效的提高了 Unikernel 在物联网环境中的任务调度效率, 避免了重要数据丢失的问题, 还能通过数学模型预测任务执行的总体情况, 降低任务执行失败的风险。同时, 基于网络时间协议, 提出一种适用于 Unikernel 物联网场景下的时钟同步方法, 通过调整从不同地点发送的数据包中的时间戳来生成时差表, 以解决全局时钟同步的问题, 从而确保 IoT-EDF 调度机制中对截止时间计算的准确。最后, 对 IoT-EDF 调度机制在 OSv Unikernel 上进行了实现和验证, 实验结果表明, IoT-EDF 在降低内存开销的情况下, 可以有效提升 OSv Unikernel 任务的执行成功率, 相比于未经改动的 OSv Unikernel, 对重要任务的完成率提升达 21%。此外, 还模拟实际应用场景, 对系统的吞吐量进行测试, 与原始的 OSv Unikernel 相比, 吞吐量提升了 30%, 进一步证明了 IoT-EDF 调度机制的有效性。

关键词 物联网; Unikernel; 调度机制; 时钟同步; 任务完成率
中图法分类号 TP302 DOI 号 10.19363/J.cnki.cn10-1380/tn.2026.01.11

IoT-EDF: Unikernel-based Task Scheduling Method for the Internet of Things

DONG Bonan[1,2], YANG Qiusong[1], LI Mingshu[1]

[1] National Engineering Research Center for Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China

Abstract Unikernel, recognized as a leading-edge technology in the field of virtualization, is widely utilized in the Internet of Things (IoT) environment due to its rapid startup, low resource consumption, and high security attributes. However, it is currently observed that Unikernel lacks a dynamic scheduling mechanism designed according to the unique characteristics of different tasks, which is crucial for ensuring efficiency in large-scale IoT task scheduling scenarios. To address this issue, the task characteristics of Unikernel in the IoT environment are first summarized. Subsequently, without altering the structural features of Unikernel, a new scheduling mechanism for the Unikernel-based IoT setting, named IoT-EDF, along with its mathematical model, is proposed. This mechanism, by categorizing tasks based on their importance and deadlines, not only significantly enhances the scheduling efficiency within the IoT environment but also prevents the loss of critical data. Moreover, it enables the overall situation of task execution to be predicted through the mathematical model, thereby reducing the risks associated with task execution failures. Simultaneously, based on the Network Time Protocol (NTP), a clock synchronization method suitable for the Unikernel IoT scenario is proposed. This method, without compromising the lightweight characteristics of Unikernel, adjusts the timestamps in data packets sent from various locations and generates a time difference table. By doing so, the issue of global clock synchronization is addressed, thereby ensuring the accuracy of deadline calculations within the IoT-EDF scheduling mechanism. Finally, the IoT-EDF scheduling mechanism is implemented and validated on the OSv Unikernel. Experimental results indicate that IoT-EDF can effectively enhance the success rate of task execution on OSv Unikernel while reducing memory overhead. Compared to the unmodified OSv Unikernel, the completion rate for critical tasks is increased by 21%. Moreover, the throughput of the system is tested in simulated real-world scenarios, demonstrating a 30% improvement compared to the original OSv Unikernel, further proving the effectiveness of the IoT-EDF scheduling mechanism in Unikernel under IoT scenarios.

Key words internet of things; Unikernel; scheduling mechanism; clock synchronization; task completion rate

通讯作者: 董博南, 博士在读, Email: bonan@iscas.ac.cn。
本课题得到中国科学院战略性先导科技专项基金资助项目资助(No. XDA-Y01-01, No. XDC02010600)
收稿日期: 2024-02-20; 修改日期: 2024-06-24; 定稿日期: 2025-12-11

1 介绍

近年来, 物联网(Internet of things, IoT)在智能城市、健康、家居以及供应链管理等领域中被大量使用[1], “万物互联”已经成为当今时代的主流。物联网虽然带来了智能化和互联的便利, 但也面临着一系列问题[2],首先是安全问题, 由于物联网设备的种类繁多且更新频率不一, 它们常常成为网络攻击的薄弱环节, 且这些设备往往收集和传输敏感数据, 一旦发生安全漏洞, 可能导致重大的隐私泄露和安全事故; 其次, 物联网设备通常电源受限, 需要运行的系统和应用程序要求小巧、高效以及低耗能; 最后, 物联网设备产生的海量数据需要进行有效的收集、存储和分析, 对数据处理能力提出了更高的要求。此外, 随着设备数量的剧增, 如何实现高效的能源管理和最小化运营成本也是一个挑战。

面对上述物联网存在的安全、能效以及数据处理等问题, Unikernel 提供了一种解决方案[3-4], Unikernel 是一种轻量级的、只包含必要功能的操作系统, 它将应用程序和操作系统内核紧密集成在一起, 形成一个可以直接在物理或虚拟硬件上运行的单一可执行文件。Unikernel 非常适用于物联网领域, 因为它兼具多个对物联网环境至关重要的特性。首先, Unikernel 将应用程序与操作系统内核紧密集成, 这种一体化设计不仅减少了系统的攻击面, 提升了安全性, 而且还能加速系统的启动时间, 使得设备能够迅速响应。其次, Unikernel 极其轻量级, 这意味着它占用的资源非常少, 适合部署在资源受限的物联网设备上。再次, Unikernel 简化了系统的复杂性, 使得设备的维护和升级更为便捷, 尤其是在需要同时管理大量分布式设备的物联网场景中, 这种易管理性显得尤为重要。此外, Unikernel 的高效性也意味着它能提供更快的处理性能, 对于需要实时或近实时数据处理的物联网应用来说, 这一点至关重要。最后, Unikernel 单一地址空间的结构特点, 减少了传统操作系统中用户空间和内核空间之间的上下文切换, 从而提供了更高的执行效率。

然而, Unikernel 在物联网环境下提供诸多优势的同时, 也为 Unikernel 在物联网环境下的调度机制带来了挑战。与其他场景不同, 物联网环境下的数据具有以下 3 个特点[5-7]: 首先, 延时敏感: 许多物联网应用场景, 诸如智能交通系统和实时健康监测等, 对数据的处理和响应有着实时性或近实时性的要求。在处理此类数据时, 面临的主要挑战在于 Unikernel 缺乏传统操作系统中用于管理实时任务的复杂调度机制。为了满足延时敏感性数据的实时处理需求, 需要对 Unikernel 本身调度策略进行重新定制, 同时, 在处理延时敏感性数据的过程中, 高精度的时钟同步机制是必不可少的。然而, 当前 Unikernel 尚未提供充分准确的时钟同步功能。这一缺陷表明, 首先, 为了适应物联网应用的高时效性特点, Unikernel 需要根据其结构特点, 设计其内在的时钟同步机制, 以确保数据调度和处理的时效性。其次, 数量大且价值密度低: 其中有价值的数据通常只占一小部分, 在考虑数据的价值密度较低的特征时, 当前 Unikernel 的调度机制设计似乎并未做出相应的适配。鉴于 Unikernel 的设计理念倾向于轻量化, 它们通常配备有限的计算和存储资源, 这种情况下, 在缺乏对于低价值密度数据特性考量的调度过程中, 可能导致对资源的非效率使用。特别是在资源受限且对延迟具有敏感性的物联网应用场景中, 这一问题尤为显著。因此, 这一挑战要求在设计 Unikernel 调度机制时, 必须慎重考虑在大量数据下如何处理和优化这些价值密度低的数据, 这可能涉及对这些数据进行有效的预处理和优化策略的选择, 以确保资源的最优分配和利用。最后, 安全性要求高: 在物联网的数据管理中, 数据往往涉及敏感性信息, 这要求在处理和传输过程中必须确保数据的安全性与隐私性。由于 Unikernel 提供的是一种轻量级虚拟化解决方案, 它相较于传统的虚拟化技术, 拥有更高的安全性优势, 这归因于其简化的代码和较小的攻击面。因而, Unikernel 的安全属性可以满足物联网数据处理所要求的标准。通过以上分析, 可以看出物联网数据的两个关键特征——对延迟的敏感性以及庞大的数据量伴随着低价值密度, 对 Unikernel 的调度机制提出了重要挑战。

现有 Unikernel 缺乏针对物联网场景及其数据上述两个关键特征所设计的调度策略[8]。文献 Mirage Unikernel[9]和 KylinX Unikernel[10]依赖于 MiniOS 为其提供的包括调度机制在内的内核功能, 采用了较为简单先来先服务(First Come First Service, FCFS)调度策略, 这种调度策略过于简单, 无法应对物联网下数据实时性要求较高的调度场景。OSv Unikernel[11]采用了多队列多处理器的调度机制(Multi-Queue Multiprocessor Scheduling, MQMS), 并在多个队列中应用了不同的策略, 但是由于任务不能在队列中移动, 导致该机制存在负载不均的问题, 此外, OSv Unikernel 在多个队列中仅采用了不同策略的相互组合, 这对于延时敏感型数据仍缺乏有效的调度策略。文献 X-Container Unikernel[12]和 UKL Unikernel[13]基

于 Linux 内核进行构建, 因此其调度机制为 Linux 内核中默认的公平调度策略(Completely Fair Scheduler, CFS), CFS 调度策略对 Unikernel 来说又过于冗余和复杂。因此, 这些调度机制并没有根据 Unikernel 特性和应用场景而深入设计, 进而降低了 Unikernel 的运行效率。在上述背景下, 本文提出一种新的调度机制 IoT-EDF 及其数学模型, 能够针对物联网环境下 Unikernel 的数据及任务特点进行有效调度, 进而提升 Unikernel 的运行效率。

本文的主要贡献如下: (1) 提出一种新的基于 Unikernel 的物联网环境下的调度机制 IoT-EDF 及其数学模型, 通过对任务重要性和截止时间进行划分, 能够更为有效地对物联网环境下 Unikernel 的任务进行调度, 同时, 利用 IoT-EDF 数学模型可以对任务总体执行情况进行较为准确的预测, 进而可以通过调整模型中的参数, 提升任务的执行成功率。(2) 在 IoT-EDF 调度机制中, 基于网络时间协议(Network Time Protocol, NTP), 提出一种适用于 Unikernel 物联网场景下的时钟同步方法, 以满足 IoT-EDF 对截止日期计算的准确性要求。(3)在 OSv Unikernel 上实现了 IoT-EDF, 验证了 IoT-EDF 调度机制的有效性。

本文第 1 节介绍研究背景和主要贡献; 第 2 节介绍 Unikernel 任务调度的相关工作及存在的问题; 第 3 节详细描述 IoT-EDF 和同步时钟的设计以及实现; 第 4 节给出相关实验及结果讨论; 第 5 节是结论及展望。

2 相关工作

2.1 Unikernel 调度机制存在问题

Unikernel 作为一种新型体系架构, 与传统虚拟机相比, Unikernel 可以为某个应用程序进行定制或修改组件, 同时只包含应用程序所必需的库, 因此具有更快的启动速度、更小的内存占用比和较小的攻击面; 与容器相比, 每个 Unikernel 有自己的内核, 比起共享内核的容器有更强的隔离性。由于 Unikernel 具有上述优势, 越来越多的研发团队加入到了 Unikernel 的研发中, 微软、IBM 和 NEC 等知名研究机构和 IT 企业都已经为 Unikernel 技术作出了显著贡献, 截至目前, 已有数十种 Unikernel 实例处于研发和优化中, 这些研究项目和研究成果广泛应用于云计算、人工智能、物联网等领域。表 1 从运行环境、主要特点以及调度策略等方面对目前主流 Unikernel 实例的相关工作进行了总结[2,4,7,9-10]。

调度机制作为 Unikernel 系统中的核心组成, 其作用是决定任务的执行次序以确保整个 Unikernel 系统能够高效地运行, 由此实现资源利用率的最大化。调度机制可以将任务集合分类为工作队列、就绪队列和等待队列。其中工作队列包含系统中所有任务; 就绪队列包括所有运行条件已具备的任务; 等待队列则是包含因等待或者其他情况而无法继续运行的任务[14]。然后调度机制根据调度算法从队列中选出待执行的任务, 结合表 1 可以看出, 常见的 Unikernel 调度算法有 FCFS、RR、CFS 以及 MQMS 等[15], 大部分现有 Unikernel 的调度机制均使用以上调度算法。

表 1 Unikernel 相关研发工作

Table 1 Unikernel related research

Unikernel	运行环境	特点	调度策略
MirageOS	Xen KVM	性能优异, 安全性高	FCFS(First Come First Service)
KylinX	Xen	基于微型操作系统重构, 强隔离	FCFS(First Come First Service)
IncludeOS	VirtualBox KVM	内存占用低, I/O 高效	FCFS(First Come First Service)
FLEXOS	KVM	采用多种安全配置策略	SJF(Shortest Job First)
HermitCore	KVM	重构 Linux 系统, 支持 Linux 程序	RR(Round-Robin)
X-Container	Xen	重构 Linux 系统, 支持 Linux 程序	CFS(Completely Fair Scheduler)
UKL	KVM Xen	重构 Linux 系统, 支持多种配置选项	CFS(Completely Fair Scheduler)
OSv	Xen KVM VirtualBox	兼容性较强, 与 Linux 程序和多个 VMM 兼容	MQMS(Multi-Queue Multiprocessor Scheduling)

FCFS 调度策略每次从就绪队列选择最先进入队列的任务, 然后一直运行, 直到任务执行完成或被阻塞, 才会继续从队列中选择下一个任务接着运行。文献 Mirage Unikernel[9]和 KylinX Unikernel[10]基于 MiniOS 进行构建并且依赖于 MiniOS 为其提供的包括 FCFS 调度机制在内的内核功能。文献 IncludeOS Unikernel[16]同样采用了 FCFS 的调度策略, 并且 IncludeOS 采用了一种非抢占式的任务管理方法, 通过使主处理器将任务分配给每个应用处理器来实现同时执行多个任务, 即每来一个任务就分配一个虚拟处理器。这种方法虽然提升了任务的执行效率, 但本质上依然是沿用了先来先服务的调度策略。FCFS 这种调度策略实现简单, 但并不适宜应用于物联网场景中, 最主要的原因是该策略对于延时敏感型数据, 难以满足其时效要求。例如: 在某些情况下, 延时敏感型数据需要等待前面的数据都执行完成才能

执行, 这种处理方式无疑对延时敏感型数据非常不利, 因此 FCFS 调度策略并不适用于物联网环境下 Unikernel 数据的特点。

相比于 FCFS 侧重于长作业的调度策略, SJF 调度策略会优先选择运行时间最短的任务来运行。文献 FlexOS Unikernel[17]采用了 SJF 调度策略, 该策略对短作业较为有利, 可以提升系统的吞吐率, 主要缺点是如果这个就绪队列有非常多的短作业, 那么就会使得长作业不断地往后推, 周转时间变长, 致使长作业长期不会被运行, 进而产生饥饿问题。与 FCFS 策略类似, SJF 策略仅仅通过作业长短对优先级进行划分, 并不适合物联网场景下延时敏感型数据的调度。

与 FCFS 和 SJF 侧重长作业和短作业不同, RR 调度策略为待执行的任务预先分配了时间片, 当前任务使用完时间片后才切换下一个任务运行。文献 HermitCore Unikernel[18]采用了基于优先级的 RR 策略, 该策略相比于 FCFS 和 SJF 来说, 对长作业和短作业相对公平, 但问题是时间片的长度设置, 如果时间片设置太短会导致过多的上下文切换, 降低了运行效率; 如果设置得太长有可能会引起对短作业任务的响应时间变长[19]。因此, RR 策略通过划分时间片的方式, 对物联网下延时敏感性数据处理能力有限。此外, 这种基于优先级的调度策略使得某些任务被赋予了较高的执行优先级, 但在确定这些任务的优先级顺序时, HermitCore 并未考虑这些任务的其他属性, 这可能导致最终的调度效率低于理想状态。

相比于上述较为简单的调度策略, CFS 是传统操作系统所使用的公平调度策略。文献 X-Container Unikernel[12]和 UKL Unikernel[13]基于 Linux 内核进行构建, 因此所使用的调度机制均为 Linux 内核中默认的 CFS 策略, 虽然 CFS 策略更能够保证任务之间的公平性以及应对物联网场景下 Unikernel 任务的特点, 但不符合 Unikernel 精简和轻量的设计要求[20]。类似的, Linux 调度设计中常见的进程优先级划分, 如限期进程、实时进程以及普通进程, 虽然可以确保任务调度的稳定性和合理性, 但这种以进程为单位的划分不适用于 Unikernel 单地址空间的架构特性, 进而导致这种进程划分设计无法在 Unikernel 中实现。

与以上较为简单或是过于复杂的策略不同, MQMS 是一种多队列调度机制, MQMS 包含了多个队列, 每种队列可以使用不同的调度策略或将多种策略进行结合[21]。OSv Unikernel 提出一种多队列调度机制, 其中包含有多个调度队列, 并在不同的队列中应用了 FCFS 和 RR 等调度策略, 并且每个调度队列相互独立, 有效避免了单队列方式的数据共享及同步问题。然而, 该策略面临着负载分配不均匀的挑战。这主要是因为各个队列采用了不同的调度策略, 导致任务处理时间各不相同, 从而引发了负载不均衡的问题。OSv Unikernel 尝试通过引入负载平衡器来减缓这一问题, 但在 Unikernel 应用广泛的物联网环境中, 特别是在任务量巨大时, 负载不均衡的问题仍然无法被完全解决。另外, 虽然 OSv Unikernel 在多个队列中使用了多种策略的组合, 但是仍难以满足延时敏感型数据的时效要求。

2.2 实时调度机制存在问题

除了上述常见的 Unikernel 调度机制外, 还有一类实时调度策略也适用于在物联网环境下进行任务调度[21], 但是, 实时调度机制并未在 Unikernel 中应用, 主要原因一方面是部分实时调度机制不符合 Unikernel 的精简性原则且开销较大; 另一方面是实时调度机制以任务截止日期先后作为任务执行顺序, 存在执行失败的风险。此外, 当前大多实时调度策略并未针对物联网任务的延时敏感特性和价值密度低特性进行设计。

在实时调度策略中[22-23], 其中最为常见的是最低松弛优先算法(Least Laxity First, LLF)和最早截止时间优先算法(Earliest Deadline First, EDF)。最低松弛优先算法 LLF 是一种动态优先级调度, 它考虑了任务的松弛时间(截止时间与剩余执行时间之差), 松弛时间越小的任务, 优先级越高。这种算法的优点是对系统资源的利用率较高并且可以有效地降低截止时间违约的风险, 但这种算法并未在物联网场景下的 Unikernel 中应用, 主要原因[24]一方面是 LLF 算法实现复杂, 不符合 Unikernel 的精简原则, 同时 LLF 算法会导致过多的上下文切换, 由于 Unikernel 采用单地址空间的结构设计, 这会降低 Unikernel 的工作效率; 另一方面是 LLF 算法对资源的开销较大, 不利于物联网场景下的节能要求。相比于 LLF 算法, 最早截止时间优先算法 EDF 的任务优先级基于它们的截止时间, 截止时间越早的任务, 优先级越高。该算法的优点是实现简单[25], 开销较小, 因此有利于在物联网场景下的 Unikernel 中所采用, 以对任务进行实时调度。但是, EDF 同样未在 Unikernel 中应用, 一方面, EDF 存在系统过载的问题, 如果系统中的任务过多, 使得总计算需求超过了处理器的处理能力, EDF 可能无法保证所有任务都能在截止时间之前完成, 需要改进和优化; 另一方面, EDF 对于物联网场景下的重要数据存在执行失败的问题, 例如某些重要数据的截止日期相对较晚, 但是仅按照截止日期

来划分优先级, 就会导致该重要数据执行不断延后, 进而存在执行失败的风险。

2.3 时钟同步机制存在问题

除了调度策略外, 在物联网实际场景下, 还需要考虑边缘设备上接收的数据来自不同的地点的不同终端, 并且这些数据中携带的绝对截止日期是从发送终端的本地时钟导出的, 因此调度机制中的时间同步也是非常重要的环节[26], 关系到调度机制中对任务截止日期计算的准确性。

目前, 在物联网场景下, 主流的时间同步方法有两种, 分别是精确时间协议算法(Precision Time Protocol, PTP)和网络时间协议算法 NTP[27]。其中, PTP 精确时间算法用于通过网络通信的形式实现设备之间的高精度时间同步, 文献[28]在分析 PTP 时钟同步协议基本原理的基础上, 提出了一种基于 STM32 单片机和 DP83640 芯片的物理层时钟同步系统, 使得系统同步时间误差小于 1 μs。文献[29]基于 PTP 时钟同步协议, 并使用 Texas Instrument 的 LaunchPad 开发板作为硬件环境, 通过抽象 PTP 包时间戳、PPS (Pulse Per Second)信号生成和外部事件时间戳, 提供了良好的同步性和可移植性。虽然基于 PTP 的协议算法提供了较好的时间精度, 但是其对硬件设施较为依赖, 额外开销较大, 同时实现也较为复杂, 并不适宜在物联网场景中的 Unikernel 环境使用。相比于 PTP 时间协议算法, NTP 网络时间协议算法[26]对硬件依赖较小, 算法的复杂性也低于 PTP, 并且也能提供较好的时间精度。文献[30]提出了一种提高 NTP 时间同步准确性的方法, 该方法考虑了由于前向和后向路径上的带宽或路由不同而导致的不对称传输延迟, 提升了同步的准确性。文献[31]通过时间过滤算法对时间戳进行计算, 导出时差表的方式来同步时间, 大大减少了硬件开销, 同时还提供了错误管理机制, 进一步保证了准确率。总体来看, 考察时间同步协议在 Unikernel 环境中的适用性时, 相比于 PTP, NTP 不依赖于硬件, 并且在算法复杂性上有所降低, 但 NTP 在错误管理和时间过滤系统方面的设计依然较为复杂。这种复杂性与 Unikernel 的设计原则相悖, 后者强调轻量级和简洁性。因此, NTP 在未进行适当简化和优化的情况下, 可能不完全适合植入 Unikernel 环境。此外, 目前大量研究采用基于哈希结构的时差表来处理时间差异数据[32]。尽管这种方法在某些应用场景中效率较高, 但哈希结构在遇到必须扩容时会引发系统停滞的问题。在 Unikernel 这种单地址空间的特殊体系结构以及追求最小化延迟的环境中, 系统停滞显得尤为不可接受。这进一步表明了传统的哈希结构在 Unikernel 中的局限性, 因此可能需要寻找更为适宜的数据结构。

综上来看, 首先, 物联网环境下 Unikernel 的数据具有以下 3 个特点: 延时敏感、数据量大且价值密度低、安全性要求高。因此在调度机制方面, 需要根据 Unikernel 的物联网应用场景和上述任务特点进行深入设计, 如果 Unikernel 缺少针对应用场景所设计的高效调度机制, 将会对 Unikernel 的运行效率造成影响, 导致其无法发挥性能优势。其次, 在时间同步方面, 需要结合物联网环境, 设计一种适用于 Unikernel 的时钟同步机制, 在避免硬件开销的前提下, 准确地对跨地域的全局时间进行同步, 以满足调度机制中对截止日期计算的准确性要求。

3 IoT-EDF

本文结合物联网应用场景, 从 Unikernel 系统层面提出了一种新的调度机制 IoT-EDF 及其数学模型, 其主要思路是在不影响 Unikernel 单地址空间结构特点的基础上, 通过对任务重要性和截止时间进行划分, 并基于 EDF 最早截止时间优先算法进行深度改进。另外, 通过对物联网场景下 Unikernel 任务特点进行分析, 并基于 NTP 协议进行优化, 解决全局时钟同步的问题, 以满足调度机制中对截止日期计算的准确性要求。

3.1 总体设计

IoT-EDF 的主要思想是基于 EDF 最早截止时间优先算法和 Unikernel 的架构特性以及物联网数据特点, 设计了新的调度机制。如图 1 所示, 物联网场景下总体工作流程示意图主要包括 3 个部分, 分别是用户终端、路由器和网关等边缘设备以及 Unikernel, 其中, 图 1 中的虚线框代表一个 Unikernel 及其内部的 IoT-EDF 机制。首先, 用户终端不定期地将需要处理的数据上传至边缘设备, 然后通过 Unikernel 进行数据处理, 最后再通过路由器等边缘设备将处理后的结果传回至用户终端。Unikernel 中的 IoT-EDF 机制主要由两部分组成, 分别是调度机制和时钟同步机制。调度机制主要包括预处理模块和多级动态队列, 其中, 预处理模块主要是对经过时钟同步后的数据进行分类以及根据优先级进行排序等操作, 执行队列则是根据任务的数量, 对队列进行动态扩展, 保证 Unikernel 在完成任务调度的前提下, 尽可能地节约系统资源。时钟同步机制主要是基于 NTP 算法, 并对算法优化。在新的机制中, 本文去除 NTP 中较为复杂的过滤算法以及错误管理机制, 采用了简洁的差值计算方法, 生成了时差表, 以符合 Unikernel 的精简

原则。这种方法的主要功能是调整不同地点发送的数据中所携带的时间戳, 使其与本地时钟一致。这样, 无论数据来自何处, 本地系统都可以将其视为在本地生成的一样处理, 从而简化了时间管理的复杂性。

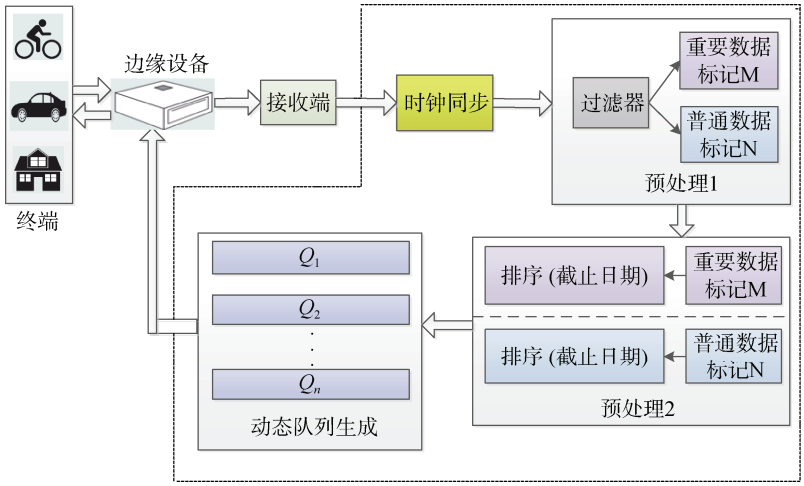

图 1　物联网场景下工作流程

Figure 1　Workflow in IoT scenarios

3.1.1　IoT-EDF 调度机制

IoT-EDF 调度机制是一种针对物联网下 Unikernel 负载特点所设计的调度策略。如图 1 所示, 包括两个预处理模块和多级动态队列, 其中预处理 1 的工作流程主要是对任务进行分类, 相比于其他 EDF 调度策略仅用截止时间来作为执行顺序的标准, 本文提出 IoT-EDF 调度机制针对物联网场景下数据的价值密度低以及延时敏感的特点, 添加新的执行标准, 即任务的重要性。因此, 预处理 1 的主要工作是通过图 1 中的过滤器, 对重要任务进行提取和标记。

然后, 经过预处理 1 分类后的数据, 进入预处理 2 模块, 预处理 2 模块实际是一个并行队列, 同时对重要任务和普通任务执行同等操作, 以提高预处理的执行效率。具体工作流程如图 2 所示, 在预处理 2 中, 以重要任务为例, 根据任务截止时间(*deadline*), 进行由小到大的排序。因为并行队列中的重要任务和普通任务数量不同, 此时, 若另一队列中的普通任务已优先完成预处理 2 的同等步骤, 则可以暂时进入下一级队列优先执行, 当重要任务预处理结束时, 普通任务立即停止执行, 将队列资源让给重要任务, 直至重要任务都执行完成, 普通任务再执行; 若另一队列中的普通任务未完成预处理 2 的步骤, 此时重要任务直接进入下一级队列优先执行, 直至执行完成, 然后普通任务再进入下一级队列执行。

最后, 考虑到 Unikernel 等边缘设备的能耗问题, 在任务执行时, 设计了动态队列的执行方式。在 Q_1、Q_2 以及 Q_n 的动态多级执行队列中, 会根据任务数量, 自动对执行队列进行扩展, 扩展方法如式(1)、式(2)所示:

$$T_{burst\ time}=\sum_{i=1}^{n}(t_{burst\ time\_1}+t_{burst\ time\_2}+\ldots+t_{burst\ time\_n}) \tag{1}$$

$$\lambda=\frac{T_{burst\ time}}{t_{\min(deadlines)\_k}-t_{arrive\ time\_k}},\ m<\lambda\le m+1 \tag{2}$$

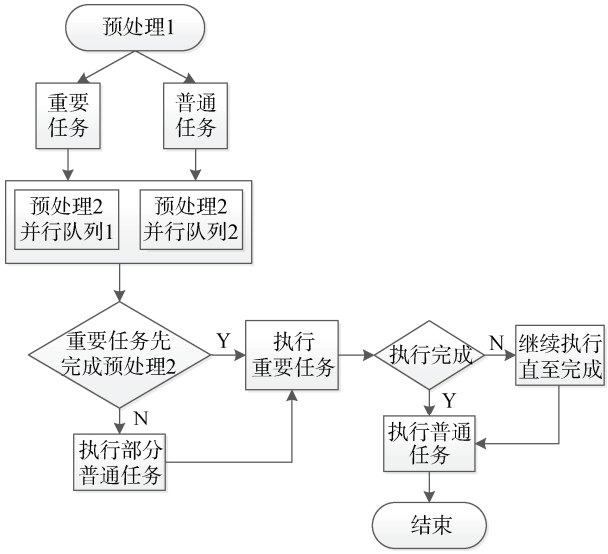

图 2　重要任务和普通任务工作流程

Figure 2　Important and regular task workflows

如式(1)所示, 将当前一组任务的运行时间, 即 *burst time* 进行累加, 可得当前一组任务所需的总运行时间 $T_{burst\ time}$。然后, 如式(2)所示, 在当前一组任务中, 选出最后截止时间 $t_{min(deadline)\_k}$ 和到达时间 $t_{(arrive\ time)\_k}$ 之差最小的任务 k, 此差值表示任务 k 的有

效可执行时间, 任务 k 也可理解为最紧急的任务, 式(2)中 m 代表为自然数, λ 为队列数。结合式(2), 将任务运行总时间与最为紧急任务 k 的有效可执行时间作比, 当比值小于 1 时, 则说明, 任务 k 的有效可执行时间大于所有任务的所需运行时间, 因此分配一个队列即可; 当比值小于 1 时, 则说明仅分配一个队列无法满足任务 k 的有效可执行时间要求, 因此需要将比值向上取整, 则为所需的队列数; 同理, 当比值等于 1 时, 则说明当前所有任务的所需运行时间和任务 k 的有效可执行时间相等, 因此分配一个队列刚好满足要求, 但为了避免此极端情况的发生, 依然采用向上取整的方式, 分配 2 个队列。举例来说: 当前有 n 个相等任务, 每个任务所需的执行时间 *burst time* 为 a, 并且每个任务的有效可执行时间同样为 a, 结合式(1)(2), 可知 $T_{burst\ time}$ 为 na, $t_{min(deadline)\_k}$ 和 $t_{(arrive\ time)\_k}$ 之差为 a, 则可得所需队列数为 n, 即每个任务独占一个队列, 刚好满足任务运行时间等于任务的有效可执行时间, 但为了避免极端情况的发生, 此处采用了向上取整的方式, 即分配队列数为 n+1。

3.1.2 IoT-EDF 时钟同步机制

在物联网实际场景下, 边缘设备上接收的数据来自不同地点的不同发送终端, 并且这些数据中携带的绝对截止日期是从发送终端的本地时钟导出的, 因此接收节点必须对时间差准确计算, 以维持数据执行的顺序, 进而使得调度器能够准确地按顺序处理这些数据。虽然 PTP 同步时钟算法可以解决上述问题, 但是成本和代价较高, 需要相应的硬件支持才可实现[28], 同时, 该算法较为复杂, 不适宜在物联网中的 Unikernel 环境下使用。相比于 PTP, NTP 同步时钟算法不依赖于硬件, 并且在算法复杂性上有所降低, 但 NTP 在错误管理和时间过滤系统方面的设计依然较为复杂。这种复杂性与 Unikernel 的设计原则相悖。

因此, 本文提出一种适用于 Unikernel 的基于 NTP 算法的时钟同步机制。在去除 NTP 较为复杂过滤算法和纠错机制的基础上, 引入了简单的差值计算和时差表生成方法。这种设计通过调整来自不同地点的数据中包含的时间戳, 以使其与本地时钟保持一致。这样, 无论数据从何处发出, 本地系统都可以像处理本地产生的数据一样处理这些数据, 从而大大降低了时间管理的复杂性。本文提出的方法依赖于 4 个关键的时间戳: 客户端发送数据包至服务器的时间戳(T_a)、服务器接收到该数据包的时间戳(T_b)、服务器回应数据包的时间戳(T_c)以及客户端接收到服务器响应的时间戳(T_d)。通过这 4 个时间戳, 本方法能够计算出往返延迟以及客户端和服务器之间的时钟偏差。通过 T_a 和 T_d 的差值, 可以得到数据包的往返时间, 而 T_b 和 T_c 提供了服务器处理请求的时间。具体校正如图 3 所示:

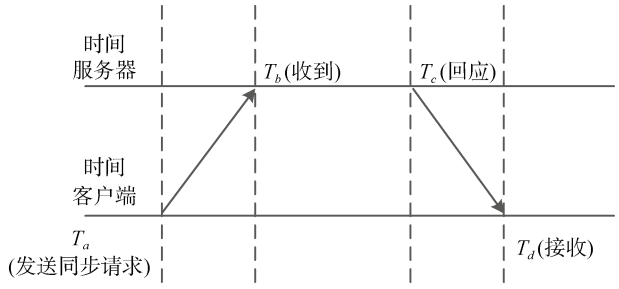

图 3 时钟校正

Figure 3 Clock correction

如图 3 所示, 在考虑网络通信延迟对时间同步精度的影响时, 一个常见假设是数据包在网络中的传输延时是对称的。这意味着客户端向服务器发送请求的网络延时 δ_1 等同于服务器回复客户端的网络延时 δ_2。基于该前提, 可以建立数学模型来估算客户端时钟与标准服务器时钟之间的差异 ζ 以及数据包在网络中的单向传播时间 η。

依据时间戳 T_a、T_b、T_c 和 T_d, 可以分别定义请求的延时和回复的延时, 即 $\delta_1 = (T_b - T_a)$ 和 $\delta_2 = (T_d - T_c)$。在假设这两个延时相等的条件下, 即 $\delta_1 = \delta_2$, 因此可以进一步推导出客户端时间与标准时间的偏差 ζ 以及网络传播时间 η。其中, 偏差 ζ 反映了客户端时钟相对于标准服务器时钟的时间误差, 而传播时间 η 提供了网络中数据包传输所需时间的估计。数学表达式如式(3)、式(4)所示:

$$\zeta = \left[(T_b - T_a) + (T_c - T_d)\right]/2 \tag{3}$$

$$\eta = (T_b - T_a) + (T_d - T_c) \tag{4}$$

式(3)计算了客户端与服务器之间的时钟偏差, 而式(4)则是基于往返时延(Round rip time, RTT)来估算单向传播延迟。其中, 在时钟同步框架中, 时钟偏差参数 ζ 和 η 的计算仅取决于时间戳 T_b 与 T_a 以及 T_d 与 T_c 之间的时间差, 而与 T_b 和 T_c 之间的时间差即服务器处理请求所花费的时间无关。因此, 可以利用这些时间戳之间的差值, 计算出时间偏移 ω 来调整本地时钟。

本文在处理时间差异时, 选用树形数据结构而非哈希表, 主要原因是树形结构无需经历哈希表那样的动态扩容过程, 避免了因扩容可能导致的 Unikernel 运行停滞, 这对于 Unikernel 来说是不可接

受的。相似的, 在传统操作系统内核中, 内核需要处理很多关键任务, 包括进程调度、内存管理、I/O 操作等, 因此内核使用了红黑树等更为稳定的数据结构, 这些数据结构虽然在某些操作上可能不如哈希表那么高效, 但它们在性能表现上更加稳定, 不会因为扩容等操作导致性能的突然下降和运行停滞。

虽然相比于哈希方式, 树查询引入了一定的计算开销, 但在本文场景下, 开销较小。一方面原因是在物联网环境中, 数据往往呈现出分层的特性, 其最突出的表现是每个数据包都包含 IP 地址信息。IP 地址本身是一种分层的标识符, 其中不同的地址段代表不同的网络层次, 从而反映出设备的物理位置。因此, IP 地址的这种固有分层特征与树形数据结构的层级组织天然对应。通过采用树形数据结构, 并通过逐层匹配 IP 地址的方法, 也可以实现对物联网数据的高效检索。这种方法利用了 IP 地址的结构特性, 允许快速定位和检索, 从而达到了较为高效的数据处理效率。另一方面, 树形结构更适合处理可能具有一定相似性的数据集, 这种相似性通常出现在数据源自同一或相似的终端设备。在哈希表中, 找到一个能够有效分散这类相似数据的哈希函数可能是具有挑战性的。如果不能合理分散, 数据就可能在哈希表中聚集, 导致冲突增多, 进而影响性能。同时, 不恰当的哈希函数还会导致空间浪费。相反, 树形结构可以更有效地组织这些数据, 减少冲突, 并提高空间利用率。

综上两节, 相比于传统 EDF 算法以及其他实时调度算法, IoT-EDF 调度机制根据物联网场景下 Unikernel 的任务特点进行了深度改进。在调度机制的设计方面, IoT-EDF 相比于传统 EDF 算法仅依靠截止时间来进行任务调度, IoT-EDF 结合物联网下数据价值密度低以及延时敏感的特性, 引入了任务重要性的概念, 并且在机制设计时, 将重要性的任务按照截止日期优先执行, 确保重要数据的完整, 然后再处理价值密度较低的普通数据。这种设计方式有效地避免了单纯按照截止日期进行执行时, 重要数据丢失的问题, 例如当重要数据的截止日期相对较晚, 但是仅按照截止日期来划分优先级, 就会导致该重要数据执行不断延后, 进而存在执行失败的风险。最后, 在任务执行方面, 考虑到物联网的应用场景, 部署在边缘设备中的 Unikernel 需要尽可能地降低能耗, 以节约成本。因此, 相比于传统操作系统中的多队列设计方案, 本文结合物联网环境下数据量大的特点, 采用动态队列的分配算法, 在保证重要任务完成执行的情况下, 尽可能地节约系统资源, 从而降低能耗。在时钟同步方面, 结合物联网应用场景, 提出一种时钟同步机制, 该机制基于 NTP 网络时间协议算法, 但经过了深度简化, 以适应 Unikernel 的设计原则。Unikernel 强调系统的简洁和轻量级, 因此对于 NTP 中的复杂过滤算法和错误管理机制, 进行了去除和简化。本文提出的这种时钟同步机制, 充分考虑了 Unikernel 的特性, 诸如其精简性原则等, 并在此基础上, 对传统的 NTP 算法进行了适当的简化和优化。这种机制的实施, 可以在 Unikernel 环境中实现高效和准确的时钟同步。 此外, 综合考虑 Unikernel 的特点, 本文所采用的树状结构时差表能够在不影响 Unikernel 稳定性的前提下, 为时间同步提供了一个既可靠又高效的解决方案。

3.1.3 IoT-EDF 数学模型

排队论[33]是研究系统随机聚散现象和随机服务系统工作过程的数学理论和方法, 又称随机服务系统理论, 经典排队理论通常假设系统的任务(客户)的等待时间没有限制。然而, 上述理论没有考虑任务的时间要求。在排队理论的背景下, 等待时间有限的任务通常被称为不耐烦作业(客户), 在实际的应用中被称为实时作业。实时作业有一个截止日期, 在此之前可以提供服务, 之后则必须离开系统。本节结合物联网场景下的数据特点, 基于实时作业的排队模型, 提出一种用于计算 IoT-EDF 调度机制中任务错过最后截止期限概率的数学模型。通过此模型, 可以对待执行任务错过截止期限起到一定预防和抑制作用。

首先, 在基于实时作业的排队论模型中[34], 假设作业到达的时间间隔服从参数为 λ 的泊松分布, 队列执行时间服从参数为 μ 的指数分布。此外, 相对截止日期与每个作业相关联, 该相对截止日期是作业到达与其截止日期之间的时间间隔。本文提出的方法是基于找到一个状态相关的损失率函数 γ_n, 该函数定义如下, 给定系统中在时间 t 有 n 个作业, 设 N 是自然数的集合, $R+$是正实数的集合。对于 $t,\varepsilon\in R+$ 和 $n\in N$, 设 $\psi_n(t,\varepsilon)$为作业在$[t,t+\varepsilon]$这个区间下错过最后期限的概率, 如式(5)所示:

$$\gamma_n(t)=\lim_{\varepsilon\to 0}\frac{\psi_n(t,\varepsilon)}{\varepsilon} \tag{5}$$

然后, 当系统达到稳定状态, 即 $t\to\infty$时, 可将式(5)化简为式(6):

$$\gamma_n=\lim_{t\to\infty}\gamma_n(t) \tag{6}$$

γ_n是系统中有 n 个作业时错过截止日期的比率。然后, 由此产生的系统的马尔可夫链模型 M 如图 4 所示。当系统的总体为 n 时, $\mathrm{Min}(n,m)\mu$ 表示完成了作业服务要求的比率, γ_n表示错过了作业的最后期限的比率。

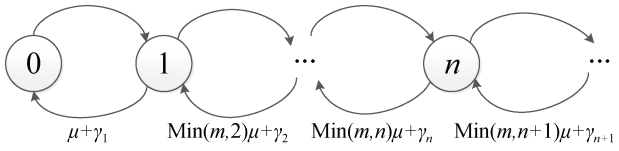

图 4 马尔可夫链 *M* 的状态率转移示意图

Figure 4 Schematic diagram of state rate transition in Markov chain *M*

接下来, 将给出求解系统模型 M 所需的方程, 并计算性能度量, 即系统的损失概率。假设 P_n 为稳定状态下, 系统此时存在 n 个作业的概率。因此, 当前系统在平衡状态下的平衡方程可以写成式(7):

$$\begin{cases} 0=-\lambda p_0+(\mu+\gamma_1)p_1, \text{ if } n=0 \\ 0=\lambda p_{n-1}-(\lambda+\text{Min}(m,n)\mu+\gamma_n)p_n \text{ if } n>0 \\ \quad +(\text{Min}(m,n+1)\mu+\gamma_{n+1})p_{n+1}. \end{cases} \tag{7}$$

根据式(7), 可得出 P_n, 如式(8)所示:

$$p_n=\frac{\lambda^n}{\prod_{i=1}^{n}(\gamma_i+\text{Min}(m,i)\mu)}p_0 \tag{8}$$

又因为 $\sum_{n=0}^{\infty}p_n=1$, 所以, 结合式(8), 可得 P_0, 如式(9)所示:

$$p_0=\left(1+\sum_{n=1}^{\infty}\frac{\lambda^n}{\prod_{i=1}^{n}(\gamma_i+\text{Min}(m,i)\mu)}\right)^{-1} \tag{9}$$

最后, 通过错过最后期限的平均比率除以任务到达的平均比率, 可以得到系统错过最后截止日期的概率, 如式(10)所示:

$$\beta=\frac{\sum_{n=1}^{\infty}p_n\gamma_n}{\sum_{n=0}^{\infty}p_n\lambda}=\frac{\sum_{n=1}^{\infty}p_n\gamma_n}{\lambda} \tag{10}$$

综上, 式(10)为 IoT-EDF 调度机制中任务错过最后截止日期的概率, 该数学模型可以对任务总体执行情况进行较为准确的预测。具体来说, 结合式(8)~(10), 首先, λ 表示为单位时间内随机事件发生的平均次数, 此处则表示单位时间内任务的到达数。因此, 通过调节参数 λ, 即当减小 λ 时, 把式(8)(9)代入式(10)可知, β 会相应降低, 则系统中任务错过最后截止时间日期的概率会降低; 其次, 式(8)中 Min(m,i)中的参数 m 可理解为系统当前的总体容量, 通过调整参数 m, 即 m 增大时, 会提高 Min(m,i)的取值上界, 进而降低 β 的值, 则同样可以有效减少系统中任务错过最后截止时间日期的概率。

3.1.4 IoT-EDF 算法复杂性分析

算法复杂性[35]的度量与计算时间和存储空间有关, 主要包括时间复杂度和空间复杂度这两类度量。由于空间复杂度计算相对简单, 并且无法反映算法执行速度或响应时间, 因此本节主要从更能表征算法运行时间与输入数据规模的时间复杂度来分析 IoT-EDF 算法。在程序实际运行中, 算法的执行包括加减乘除和排序等操作。其中, 排序主要包括交换和比较的运算, 而运算次数主要与使用何种排序算法以及数列有关。为便于评估算法, 可通过运行时间 $t(s)$ 来衡量算法的时间复杂度, 如式(11):

$$\begin{aligned} t(s)=&c_aA(s)+c_sS(s)+c_mM(s)+c_dD(s)\\ &+c_cC(s)+c_eE(s) \end{aligned} \tag{11}$$

如式(11), C_a …C_e 分别代表一次加法、减法、乘法、除法以及比较和交换运算所需的时间; 函数 A...E 分别为算法中加法、减法、乘法、除法以及比较和交换操作所对应的次数。

根据式(11), 对 IoT-EDF 的时间复杂度进行计算。首先, 对于时钟同步机制, 包括时钟延迟偏移计算以及生成树状时差表等操作, 因此, 其时间复杂度如式(12)所示:

$$t(syn)=O(n+n\log n) \tag{12}$$

接下来, 通过进入预处理 1 模块和预处理 2 模块, 其中预处理 1 模块仅将数据分为重要数据和普通数据, 因此, 其时间复杂度为 $O(n)$, 然后, 数据进入预处理模块 2, 主要包括对任务进行排序操作, 因此, 预处理模块 2 的时间复杂度为 $O(\phi n\log\phi n)$ + $O(\psi n\log\psi n)$。进而可以得出整个预处理的时间复杂度如式(13)所示, 其中ϕ和 ψ 分别为重要数据和普通数据所占百分比:

$$t(pre-deal)=O(n+\phi n\log\phi n+\psi n\log\psi n) \tag{13}$$

最后, 对于动态队列, 主要包括队列生成和任务执行两个环节, 因此, 此处时间复杂度较为简单, 如式(14)所示:

$$t(Q_{dynamic})=O(2\phi n+2\psi n) \tag{14}$$

因此根据上述公式, 可以得到 IoT-EDF 的时间复杂度如式(15)所示:

$$\begin{aligned} t(\text{IoT}-\text{EDF})=&O(2n+n\log n+\phi n\log\phi n\\ &+\psi n\log\psi n+2\phi n+2\psi n) \end{aligned} \tag{15}$$

如式(15), $O(n)$和 $O(+\phi n)$以及 $O(\psi n)$相对于 $O(n\log n)$影响较小, 也可隐去。进而可以将式(15)进一步化简为式(16):

$$\begin{aligned} t(\text{IoT}-\text{EDF})=&O(n\log n+\phi n\log\phi n\\ &+\psi n\log\psi n) \end{aligned} \tag{16}$$

在此基础上, 对 OSv 原始的调度策略进行时间

复杂度分析, 由于 OSv 是基于较为复杂的 freeBSD 内核进行重构[36]。因此, 在每级队列中采用了较为复杂的抢占式优先级以及公平调度策略, 时间复杂度为 $O(n\log n)$和 $O(n^2)$。因此, 参照 IoT-EDF 的时间复杂度方法, 可以得出 OSv 的时间复杂度如式(17)所示:

$$t(\mathrm{OSv}) = O\left\{\psi(n\log\psi n + n^2) + (1-\psi)\left[n\log(1-\psi)n + n^2\right]\right\} \tag{17}$$

其中, ψ 代表各级队列中任务数的百分比, 通过对比式(16)和式(17), IoT-EDF 的时间复杂度明显小于 OSv 中的原始调度策略的时间复杂度, 这也进一步证明了 IoT-EDF 机制的有效性。

3.2 实现

OSv Unikernel[11]采用单地址空间设计, 可以直接在虚拟机管理程序上运行, 相比于其他 Unikernel, OSv 最显著的特点是能够支持现有的 Linux 程序。另外, OSv 运行在 CPU 最高特权级下, 大部分设计为系统调用的接口, 在 OSv 中也转化为函数调用的形式供应用程序使用, 这进一步减少了上下文切换, 相比于传统操作系统具有显著的性能优势。另外, OSv 也通过提供可编译链接的语言运行时支持的方式, 支持 Java 等语言编写的程序。OSv 架构如图 5 所示:

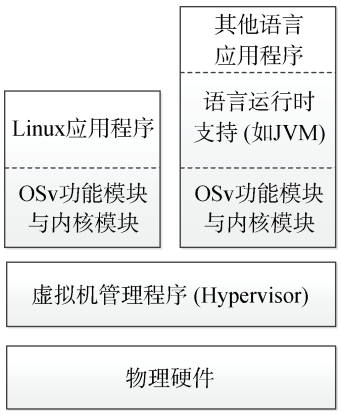

图 5 OSv 架构示意图
Figure 5 OSv architecture

以 OSv Unikernel 为基础, 对 IoT-EDF 进行设计和实现的好处主要在于:

(1) 使用 C++语言开发, 支持 JVM 等多种语言运行时, 能够满足物联网场景下运行不同编程语言应用的需求。

(2) OSv Unikernel 可以直接在标准的虚拟机管理程序上运行, 方便在不修改外部依赖平台的情况下提供新的系统特性, 通用性比较强。

(3) OSv 系统在 Unikernel 诸多系统设计中, 生态比较完善, 研究者比较活跃, 具有代表性。

(4) OSv Unikernel 采用了多队列的调度机制, 相比于其他 Unikernel, 其调度效果更好。因此, 基于 OSv Unikernel 进行修改, 可以更好的检验 IoT-EDF 机制的有效性。

在 OSv 上实现 IoT-EDF 机制时, 主要工作是将原 OSv 的调度策略修改为按照 IoT-EDF 调度机制来选择下一个待执行的任务。经分析, 其中的关键函数是由 OSv 中 CPU 类的 reschedule 函数触发, 因此在实现时主要工作是对类成员函数 reschedule 进行修改, 并将其实现为本文提出的 IoT-EDF 多级反馈队列机制, 其中, OSv 的 reschedule 函数的工作流程如图 6 所示:

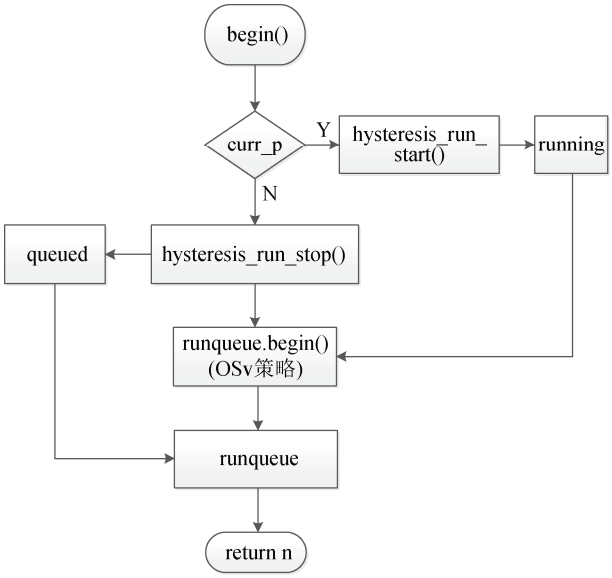

图 6 OSv 关键函数 reschedule 调度流程
Figure 6 OSv reschedule scheduling process

如图 6 所示, OSv 实际包含一个 CPU 类, 在该 CPU 类的 reschedule 函数中, 首先判断当前线程是否满足运行条件。如果满足, 则调用 hystereis_run_start()方法, 更新该线程的活跃度, 并继续保持 runing 状态, 直至运行完成, 最后调用 runqueue.begin()方法, 采用该函中的 OSv 调度策略, 从 runqueue 即就绪队列中选择下个运行的线程 n, 最后返回; 如果线程不满足当前的运行条件, 则调用 hystereis_run_stop()方法, 转入 queued 状态, 并进入就绪队列 runqueue 等待再次被调度。然后, 当前线程 p 不再运行时, reschedule 函数会调用 runqueue.begin()方法, 从 runqueue 即就绪队列中选择下个运行的线程 n, 最后返回。

根据图 6 对 OSv 中 reshedule 工作流程的分析, 本文主要从 CPU 类的 reshedule 函数中的 runqueue.begin()方法入手, 将其中的 OSv 调度策略修改为本文提出

的 IoT-EDF 策略。因此修改后的工作流程图如上图 7 所示, 对 runqueue.begin()进行修改主要包括以下操作: 首先利用通过两个 pre_deal()函数对任务进行两次预处理并计算所需队列的数目, 然后在执行队列中使用 FCFS 调度策略, 该调度策略由函数 sched_FCFS()实现, 最终, 完成所有任务的调度和执行。

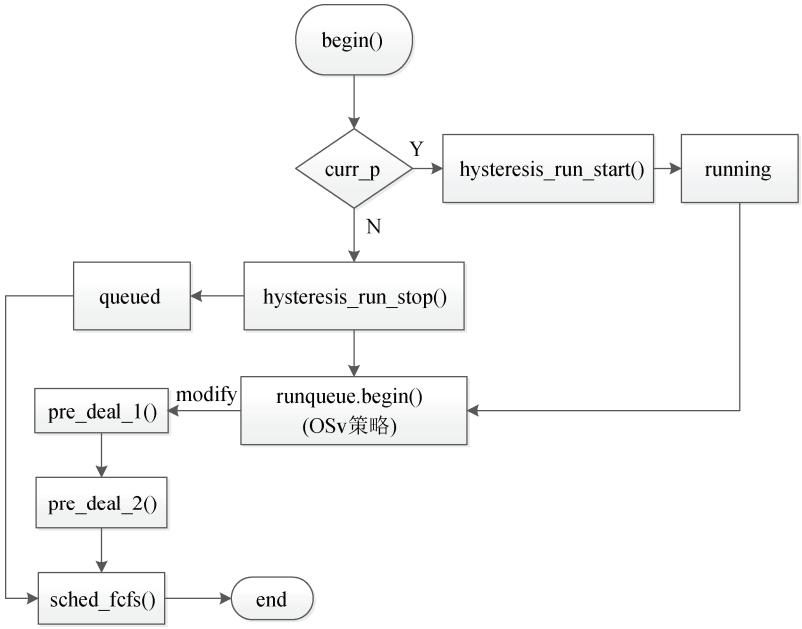

图 7 IoT-EDF 调度流程

Figure 7 IoT EDF scheduling process

4 实验

本研究在 Intel 平台上运行 Linux 内核版本为 5.4.0 的 Ubuntu20.04.6 操作系统上进行了实验, 具体配置信息如表 2 所示。

表 2 配置信息

Table 2 Configuration information

CPU	Intel Core™ i7-10700
操作系统	Ubuntu20.04.6
内核版本	5.4.0
内存	64G
硬盘	1T

4.1 实验设计

针对本文提出的 IoT-EDF 调度机制进行实验设计时, 为了更好地模拟实际物联网场景, 采用非周期型任务作为输入, 在这种输入情况下, 一旦同一任务的前一个作业到达后经过了最小的到达间隔时间, 任务的每个作业就可以在任何时间到达, 因此, 模拟方法如下:

1) 设置的输入为随机生成 5 组任务, 每组任务 500 个作业, 重要作业和普通作业随机分布。2) 其中每个作业的最后截止时间为作业的到达时间与每组任务间隔时间的和, 其中, 作业到达时间取值范围为当前时刻加上[0~20 ms]内的随机取值, 每组任务之间间隔为[1~3 s]内的随机取值, 每个作业运行时间为[1~100 ms]内的随机取值。3) 共计实验 5 次, 取均值。

4.1.1 局域网和广域网下时钟同步机制

针对本文提出的时钟同步机制进行实验设计时, 为了更为有效地验证时钟同步的一致性, 分别在广域网和局域网下进行实验, 具体实验方法如下: 在广域网(WAN)和局域网(LAN)环境下, 分别利用 5 台终端进行时钟同步实验。实验中, 每台终端在 10 min 内以随机时间戳向服务器上运行的 Unikernel 发送总计 100 个数据包。服务器端的 Unikernel 应用了本文提出的轻量级时钟同步算法, 对接收到的所有数据包进行时钟校正, 并按照时间顺序进行排序。排序完成后, 与所有终端汇总的数据包发送时的全局时间进行排序比对和校验。该实验持续 48 h, 以 10 min 为一个实验周期, 共重复进行了 288 个实验周期。

4.1.2 IoT-EDF 与其他调度机制

首先, 使用本文提出的 IoT-EDF 机制与其他常见的调度机制从任务执行完成率的角度进行对比, 以验证 IoT-EDF 的执行效率。然后, 在验证算法执行效率的基础上, 因为在物联网环境中, 随着任务数量的增多, 最有可能出现的情况就是内存资源紧张。

因此采用内存开销为指标, 对比 IoT-EDF 算法和其他常见算法的内存开销, 以验证算法对资源的使用情况, 具体实施方式如下:

(1) IoT-EDF 和其他调度机制的任务完成率: 本实验主要是对比 IoT-EDF 和其他常见调度机制对任务的调度能力, 以验证 IoT-EDF 机制及其相关算法本身的有效性。分别对比 IoT-EDF 和 MQMS、FCFS、SJF、RR 以及 CFS 的任务完成率。

(2) IoT-EDF 和其他算法的内存开销: 在本实验中, 分别对比 IoT-EDF 和 FCFS、SJF、RR、MQMS 以及 CFS 的内存开销。在具体实施时, 借助 Linux top 工具查看算法对应进程下的内存占用率作为计量方式, 对比 IoT-EDF 与其他常见算法的内存开销, 以验证算法对资源的使用情况。

4.1.3 IoT-EDF Unikernel 与 OSv 调度机制

在对算法本身进行完成率和内存开销实验的基础上, 结合 Unikernel 对未经修改的 MQMS OSv 机制和使用 IoT-EDF 机制后的 OSv 进行比对实验, 以验证 IoT-EDF 在 Unikernel 上的有效性。其中, MQMS 为多队列调度机制, 是 OSv 目前采用的调度机制。具体实施方式如下:

MQMS OSv 和 IoT-EDF OSv: 在本实验中, 主要是验证 IoT-EDF 在 Unikernel 上的有效性。分别使用 OSv 当前的调度机制 MQMS OSv 和本文提出的 IoT-EDF OSv 进行任务完成率的对比。

4.1.4 模拟实际场景

结合以上实验, 首先对本文提出的 IoT-EDF 算法的任务完成率和内存开销与其他算法进行了比较, 然后在此基础上, 结合 OSv Unikernel, 进行了任务完成率的对比。接下来, 将进一步结合实际场景, 对无改动的 OSv 和部署了 IoT-EDF 机制的 OSv, 以及 MirageOS Unikernel 的调度机制进行对比。这样的对比旨在评估在实际场景下, IoT-EDF OSv 与未经改动的 OSv 和其他 Unikernel (MirageOS)的调度效果, 以下是详细实施方案:

MQMS OSv 和 IoT-EDF OSv 以及 MirageOS Unikernel: 在本实验中, 通过在不同的 Unikernel 上实现 UDP 通信来测试吞吐量, 选取 UDP 主要是因为 UDP 提供了一种简单快速的数据传输方式, 进而可以通过建立 UDP 连接, 进行大量的消息收发, 便于对不同 Unikernel 的调度策略进行评估。此外, UDP 非常适宜在 Unikernel 中建立服务通信, 使得 UDP 请求可以被快速地接收和处理并返回。实验主要步骤以及设置如下: 1) 在 MQMS OSv、IoT-EDF OSv 以及 MirageOS Unikernel 上建立 UDP 连接。选取 MirageOS 的主要原因一方面是 MirageOS 大量应用于物联网场景, 适合与 IoT-EDF OSv 进行对比; 另一方面, MirageOS 开源且代码完善, 并且和 OSv 一样, 均提供了实现 UDP 通信的依赖库, 方便在 Unikernel 上建立 UDP 连接。2) 给 Unikernel 添加网络支持, 配置网络接口等, 并使其监听指定的端口, 接收并回复客户端的消息。3) 在另一台处于同一网络的机器上启动 UDP 客户端, 它将消息发送到 Unikernel 的 UDP 服务器, 接收并显示来自服务器的回复。4)最后利用 Wireshark 工具获取吞吐率。

4.2 实验结果与分析

4.2.1 局域网和广域网下时钟同步机制

局域网和广域网下时钟同步机制, 时间排序准确率对比, 实验结果如图 8 所示。

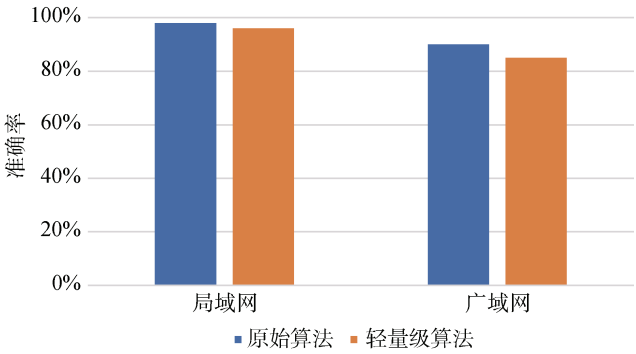

图 8 时钟同步机制对比

Figure 8 Comparison of clock synchronization Mechanisms

如图 8 所示, 在局域网下, 原始算法的时钟校正准确率为 98%, 而本文提出的轻量级算法的准确率为 96%。因此, 在网络条件较好时, 两种算法对于时钟校正的准确率基本一致, 本文提出的轻量级算法并未因去除过滤算法和纠错机制而影响校正准确率。在广域网下, 原始算法的校正准确率为 90%, 本文提出的轻量级算法为 86%, 相较于局域网环境, 准确率有所下降。然而, 两者相差并不大。因此, 在较为复杂的网络条件下, 本文提出的轻量级时钟同步算法也可以保持较高的准确率, 同时由于去除了复杂的过滤算法和纠错机制, 为 Unikernel 获得了较大的内存和空间优势。总体来看, 本文提出的时钟同步方法提供一种在计算复杂性、成本效益和维护便捷性之间的平衡, 并且能满足 Unikernel 在物联网场景下的时间同步需求, 尽管本方法相比于复杂滤波算法的同步策略在精度上有所妥协, 但设计理念与 Unikernel 精简性的目标相契合, 并且无需对 Unikernel 的结构进行调整, 同时还可以降低计算负

荷和实现的复杂度, 这对资源受限的 Unikernel 环境尤为重要。此外, 本方法还可以消除对硬件资源的依赖, 降低实施成本。

4.2.2 IoT-EDF 与其他调度机制

(1) IoT-EDF 和其他调度机制的任务完成率对比, 实验结果如图 9 所示。

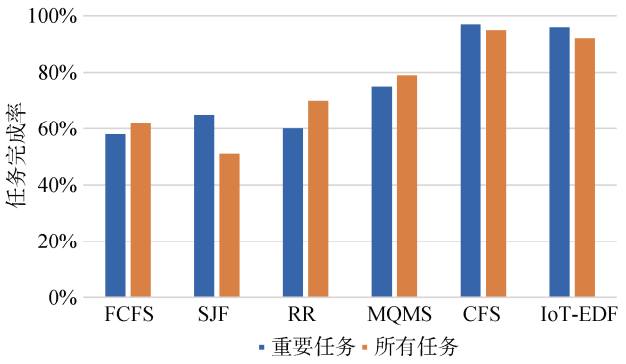

图 9 IoT-EDF 与其他调度机制任务完成率对比
Figure 9 Comparison of task completion rate between IoT-EDF and other scheduling mechanisms

如图 9 所示, 可以看出重要任务完成率和所有任务完成率最高的是 IoT-EDF 机制和 CFS 机制, 也就是说, 调度效果最好的是上述两种机制。其中, 从数据上来看, CFS 略微好于 IoT-EDF, 但差距极其细微, 并不明显。这是因为 CFS 是传统操作系统所使用的公平调度策略, 因此调度能力较强。但是 CFS 对于 Unikernel 来说过于复杂, 不符合 Unikernel 的精简设计原则。同时, 从图 9 还可以看出, 除了复杂的 CFS 机制外, IoT-EDF 在重要任务完成率和所有任务完成率上均优于其他机制, 并且 IoT-EDF 保证了极高的重要任务完成率, 达到 96%, 与 CFS 机制几乎持平。

(2) IoT-EDF 和其他算法的内存开销对比, 实验结果如图 10 所示:

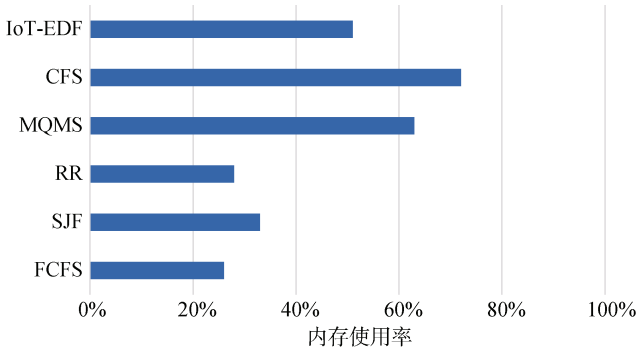

图 10 IoT-EDF 与其他调度机制内存开销对比
Figure 10 Comparison of memory overhead between IoT- EDF and other scheduling mechanisms

如上图 10 所示, 内存开销最高的是 CFS 算法, 因为该算法包含多次排序和抢占等复杂运算, 在同等条件下, 内存开销达 72%。接下来是 MQMS 算法, 由于该算法存在负载不均的问题, 这会导致内存增加, 同等条件下, MQMS 的内存开销约为 63%。然后是本文提出的 IoT-EDF 算法, 在同等条件下, 内存开销为 51%, 相比于 CFS 算法和 MQMS 算法, 内存开销分别减少约 21%和 12%。虽然 IoT-EDF 算法的内存开销高于较为简单的 FCFS 和 SJF 算法, 但是参考实验(1)的结果可知, IoT-EDF 获得了执行效率的优势, 并且内存消耗也小于 MQMS 和 CFS 等复杂算法。

4.2.3 IoT-EDF Unikernel 与 OSv 调度机制

MQMS OSv 和 IoT-EDF OSv 的任务完成率对比, 实验结果如下图 11 所示: IoT-EDF OSv 在重要任务完成率和所有任务的完成率上均高于 MQMS OSv。其中, IoT-EDF OSv 重要任务完成率达到 97%, 所有任务完成率达到 93%, 而 MQMS OSv 的重要任务完成率为 76%, 所有任务完成率为 80%。由此可见, IoT-EDF 在这两项指标上对原始未经改动的 OSv 调度机制提升了 21%和 13%。

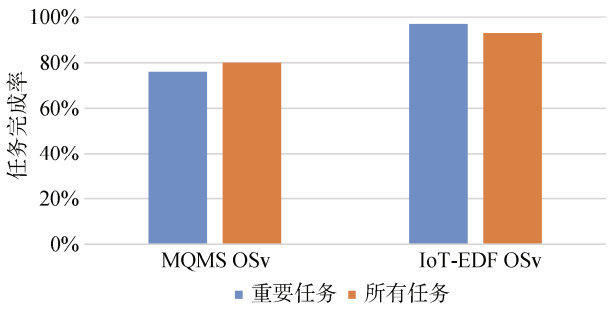

图 11 MQMS OSv&IoT-EDF OSv 任务完成率对比
Figure 11 Comparison of MQMS OSv and IoT-EDF OSv task completion rate

此外, IoT-EDF 的重要任务完成率达到 97%, 又因为所有任务完成率为 93%, 可知重要任务的完成效果优于普通任务, 这也和算法的设计思想相契合。反观 MQMS OSv, 并未对重要任务进行区分, 使得其重要任务完成率仅为 76%, 而且还低于其普通任务的完成率。

4.2.4 模拟实际场景

MQMS OSv 和 IoT-EDF OSv 以及 MirageOS Unikernel 的吞吐量对比, 实验结果如图 12 所示(服务器共计运行 40 个并发线程, 每个线程在测试机器上发出 50000 个请求, 其中重要请求和普通请求随机分布)。

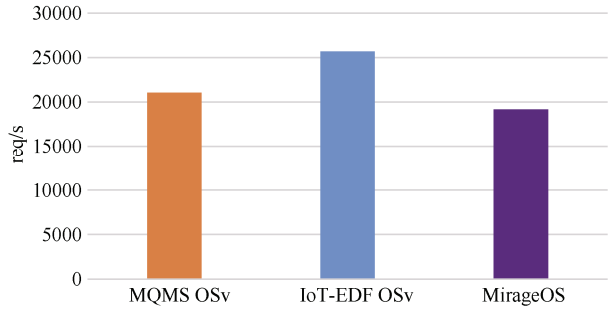

图 12　不同 Unikernel 的吞吐量对比

Figure 12　Comparison of throughput for different Unikernel

本实验结合实际场景, 利用吞吐量来对不同 Unikernel 的调度机制进行评估, 即吞吐量越高, 调度机制越为有效。如图 12 所示, IoT-EDF 的吞吐量最高, 相比 MQMS OSv 和 MirageOS 分别提高约 30.2%和 42.3%, 主要原因是 IoT-EDF 对物联网环境下 Unikernel 的任务特点更具有针对性, 因此表现出了吞吐量的优势, 这也验证了 IoT-EDF 在实际场景中的有效性。

5　结论

本文针对 Unikernel 在物联网环境下存在的任务调度问题, 提出一种新的基于 Unikernel 的物联网环境下的调度机制 IoT-EDF 及其数学模型, 在不影响 Unikernel 结构特点的前提下, 通过对任务重要性和截止时间进行划分, 能够更为有效的对物联网环境下的 Unikernel 任务进行调度, 并且根据 IoT-EDF 提出了一种可用于计算任务错过最后截止日期概率的数学模型, 该数学模型可以对任务总体执行情况进行较为准确的预测, 进而可以通过调整模型中的参数, 提升任务的执行成功率。同时, 为了更好地满足物联网数据对实时性要求较高的特点, 基于 NTP 协议, 提出一种适用于 Unikernel 的物联网场景下更为简洁的时钟同步方法, 通过调整不同地点发送的数据中所携带的时间戳等措施, 生成树状时差表, 不但避免了哈希结构时差表对 Unikernel 带来的停滞问题, 而且满足了 IoT-EDF 中对截止日期计算的准确性要求。

本文对 IoT-EDF 在 OSv Unikernel 上进行了实现和验证, 实验结果表明, 使用本文提出的 IoT-EDF 机制的 OSv 在重要任务完成率和总任务完成率上都优于原始 OSv 的调度机制, 使用 IoT-EDF 机制后, IoT-EDF OSv 比原始的 MQMS OSv 在重要任务完成率上提升 21%, 总任务完成率上提升 13%。另外, 在算法开销方面, 本文采用内存开销作为评价指标, 对 IoT-EDF 及其他主要调度算法进行比较, 验证了 IoT-EDF 在获得执行效率优势的同时, 内存开销也小于 CFS 等复杂算法。最后, 本文还结合实际场景, 通过计算吞吐量, 并与其他 Unikernel 进行比较, 进一步验证了 IoT-EDF 在实际物联网环境中的有效性。

目前, 本文提出的 IoT-EDF 机制主要是针对物联网环境下 Unikernel 的任务特点和应用场景进行设计实现和验证。随着 Unikernel 的应用场景[37]不断扩展至人工智能、高性能计算(High performance computing, HPC)和科学计算等新领域, Unikernel 的调度策略也要重新设计和进行针对性的改进。例如在机器学习场景下, 数据对资源的需求远远大于其他场景, 因为机器学习任务通常需要大量的计算资源、内存和存储空间, 这对 Unikernel 的资源管理和调度提出了新的挑战, 同时这也是本文研究扩展延伸的方向。

参考文献

[1] Chen S C, Xu R J, Sun W Q. An Edge Computing Architecture Based on Unikernel[C]. *2022 Australian & New Zealand Control Conference*, 2022: 188-191.

[2] Cozzolino V, Schwellnus N, Ott J, et al. UIDS: Unikernel-Based Intrusion Detection System for the Internet of Things[C]. *Proceedings 2020 Workshop on Decentralized IoT Systems and Security*, 2020: 15-21.

[3] Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: Library operating systems for the cloud[J]. *ACM SIGARCH Computer Architecture News*, 2013, 41(1): 461-472.

[4] Goethals T, Sebrechts M, Atrey A, et al. Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications[C]. *2018 IEEE 8th International Symposium on Cloud and Service Computing*, 2018: 1-8.

[5] Kuenzer S, Bădoiu V A, Lefeuvre H, et al. Unikraft: Fast, Specialized Unikernels the Easy Way[C]. *The Sixteenth European Conference on Computer Systems*, 2021: 376-394.

[6] Duncan B, Happe A, Bratterud A. Enterprise IoT Security and Scalability: How Unikernels Can Improve the Status Quo[C]. *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing*, 2017: 292-297.

[7] Cozzolino V, Ding A Y, Ott J, et al. Enabling Fine-Grained Edge Offloading for IoT[C]. *The SIGCOMM Posters and Demos*, 2017: 124-126.

[8] Li Z, Cheng J, Chen Q. {RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing[C]. *USENIX Annual Technical Conf-erence*, 2022: 53-68.

[9] Imada T. MirageOS Unikernel with Network Acceleration for IoT Cloud Environments[C]. *The 2018 2nd International Conference on Cloud and Big Data Computing*, 2018: 1-5.

[10] Zhang Y, Crowcroft J. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization[C]. *USENIX Annual Technical Conference,* 2018: 173-186.

[11] Kivity A, Laor D, Costa G. OSv-Optimizing the Operating System for Virtual Machines[C]. *USENIX Annual Technical Conference*,

2014: 61-72.

[12] Shen Z M, Sun Z, Sela G E, et al. X-Containers: Breaking down Barriers to Improve Performance and Isolation of Cloud-Native Containers[C]. *The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019: 121-135.

[13] Raza A, Unger T, Boyd M, et al. Unikernel Linux (UKL)[C]. *The Eighteenth European Conference on Computer Systems*, 2023: 590-605.

[14] Li T, Baumberger D, Koufaty D A, et al. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures[C]. *The 2007 ACM/IEEE conference on Supercomputing*, 2007: 1-11.

[15] Ali S, Alshahrani R, Hadadi A. A review on the cpu scheduling algorithms: Comparative study[J]. *Secur*, 2021, 21(1): 19-26.

[16] Bratterud A, Walla A A, Haugerud H, et al. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services[C]. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science*, 2016: 250-257.

[17] Lefeuvre H, Bădoiu V A, Jung A, et al. FlexOS: Towards Flexible OS Isolation[C]. *The 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022: 467-482.

[18] Lankes S, Pickartz S, Breitbart J. HermitCore: A Unikernel for Extreme Scale Computing[C]. *The 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016: 1-8.

[19] Alhaidari F, Balharith T Z. Enhanced round-robin algorithm in the cloud computing environment for optimal task scheduling[J]. *Computers*, 2021, 10(5): 63.

[20] Raza A, Sohal P, Cadden J, et al. Unikernels: The Next Stage of Linux'S Dominance[C]. *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019: 7-13.

[21] Zhao Y, Suo K, Wu X F, et al. Preemptive Multi-Queue Fair Queuing[C]. *The 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019: 147-158.

[22] Cozzolino V, Ding A Y, Ott J. FADES: Fine-Grained Edge Offloading with Unikernels[C]. *The Workshop on Hot Topics in Container Networking and Networked Systems*, 2017: 36-41.

[23] Chen K H, Günzel M, Jablkowski B, et al. Unikernel-Based Real-Time Virtualization under Deferrable Servers: Analysis and Realization[C]. *Euromicro Conference on Real-Time Systems*, 2022

[24] Omar H K, Jihad K H, Hussein S F. Comparative analysis of the essential CPU scheduling algorithms[J]. *Bulletin of Electrical Engineering and Informatics*, 2021, 10(5): 2742-2750.

[25] Mas L, Vilaplana J, Mateo J, et al. A queuing theory model for fog computing[J]. *The Journal of Supercomputing*, 2022, 78(8): 11138-11155.

[26] Cheng T, Liu S, Xu Q M, et al. Digital-Twin-Based Clock Synchronization in Industrial IoT Systems[C]. *2023 IEEE 13th International Conference on CYBER Technology in Automation*, *Control*, *and Intelligent Systems*, 2023: 193-198.

[27] Gore R N, Lisova E, Åkerberg J, et al. CoSiNeT: A Lightweight Clock Synchronization Algorithm for Industrial IoT[C]. *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems*, 2021: 92-97.

[28] Yuan K X, Guo X B, Tian J Y. Research and implementation of clock synchronization technology based on PTP[J]. *Journal of Physics*: *Conference Series*, 2021, 1757(1): 012139.

[29] Wiesner A, Kovácsházy T. Portable, PTP-Based Clock Synchronization Implementation for Microcontroller-Based Systems and Its Performance Evaluation[C]. *2021 IEEE International Symposium on Precision Clock Synchronization for Measurement*, *Control*, *and Communication*, 2021: 1-6.

[30] Mkacher F, Duda A. Calibrating NTP[C]. *2019 IEEE International Symposium on Precision Clock Synchronization for Measurement*, *Control*, *and Communication*, 2019: 1-6.

[31] Wei B Y, Liang K, Yu T. Simulation and Evaluation of Time Synchronization Performance Based on NTP and PTP[C]. *2023 Joint Conference of the European Frequency and Time Forum and IEEE International Frequency Control Symposium*, 2023: 1-4.

[32] Lan Y K, Chen Y S, Hou T C, et al. Development board implementation and chip design of IEEE 1588 clock synchronization system applied to computer networking[J]. *Electronics*, 2023, 12(10): 2166.

[33] Adhikari S, Hutaihit MA, Obaid AJ. Analysis of Average Waiting Time And Server Utilization Factor Using Queueing Theory In Cloud Computing Environment[C]. *International Journal of Nonlinear Analysis and Applications*, 2021, 12: 1259-1267.

[34] Ke J C, Wu C H, Zhang Z G. Recent developments in vacation queueing models: A short survey[J]. *International Journal of Oper-ations Research*, 2010, 7(4): 3-8.

[35] Zenil H. A review of methods for estimating algorithmic complexity: Options, challenges, and new directions[J]. *Entropy*, 2020, 22(6): 612.

[36] Izurieta C, Bieman J. The Evolution of FreeBSD and Linux[C]. *The 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 2006: 204-211.

[37] Ahuja A, Jain V. Challenges and Opportunities for Unikernels in Machine Learning Inference[C]. *2021 9th International Conference on Reliability*, *Infocom Technologies and Optimization* (*Trends and Future Directions*), 2021: 1-5.

董博南 于 2015 年在北京化工大学获得硕士学位。现在中国科学院大学软件工程专业攻读博士学位。研究领域为新型体系架构、系统安全。研究兴趣包括操作系统、系统安全。Email: bonan@iscas.ac.cn

杨秋松 于 2008 年在中国科学院大学获得博士学位, 现任中国科学院软件研究所研究员, 博士生导师。主要研究领域为操作系统、系统安全、软件工程。研究兴趣包括操作系统、软件工程、系统安全。Email: qiusong@iscas.ac.cn

李明树 于 1993 年在哈尔滨工业大学获得博士学位, 现任中国科学院软件研究所研究员、博士生导师、CCF 会士。主要研究领域为新型体系架构、操作系统安全。研究兴趣包括操作系统、软硬件融合。
Email: mingshu@iscas.ac.cn