软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006965]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于分层软总线的新型通用操作系统结构模型*

杨 攀[1], 董 攀[1], 江 哲[2], 丁 滟[1]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](University of Cambridge, Cambridge CB2 1TN, UK)

通信作者: 董攀, E-mail: pandong@nudt.edu.cn

摘　要: 传统的操作系统设计所面临的主要挑战是需要管理的资源数量, 多样性, 分布范围不断增加以及系统状态频繁变化. 然而, 现有操作系统结构已经成为应对上述挑战的最大障碍, 原因如下: (1) 紧耦合和中心化的结构不仅损害了系统的灵活性和扩展性, 还导致了操作系统生态分离; (2) 系统中单一的隔离机制, 如内核态-用户态隔离, 造成了各种能力之间的矛盾, 如安全性和性能等. 为此, 结合简捷的分层软总线设计思想和多样化隔离机制来组织操作系统组件, 提出一种新型操作系统模型——Yggdrasil. Yggdrasil 将操作系统功能分解为以软总线相连接的组件节点, 其通信被标准化为经软总线的消息传递. 为支持特权态等隔离状态的划分和不同的软件层次, Yggdrasil 还引入桥节点实现多层软总线的级联和受控通信, 通过自相似的拓扑特性使操作系统的逻辑表述能力和扩展性都得到了极大的强化. 此外, 软总线的简单性和层次性也有助于实现去中心化. 构建操作系统的分层软总线模型实例 HiBuOS, 通过 3 方面的具体设计验证并展示基于 Yggdrasil 思想来开发新型操作系统的可行性: (1) 根据目标操作系统的规模和要求设计规划多层总线结构; (2) 选择具体的隔离和通信机制实例化桥节点和软总线; (3) 实现基于分层软总线风格的操作系统服务. 对 HiBuOS 的评估表明, 它没有引入明显的性能损耗, 而且还在提高系统可扩展性, 安全性和生态发展方面具有显著的优势和潜力.

关键词: 分层软总线; 操作系统结构模型; 松耦合; 多样化隔离机制; 自相似性

中图法分类号: TP316

中文引用格式: 杨攀, 董攀, 江哲, 丁滟. 基于分层软总线的新型通用操作系统结构模型. 软件学报. http://www.jos.org.cn/1000-9825/6965.htm

英文引用格式: Yang P, Dong P, Jiang Z, Ding Y. Novel and Universal OS Structure Model Based on Hierarchical Software Bus. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6965.htm

Novel and Universal OS Structure Model Based on Hierarchical Software Bus

YANG Pan[1], DONG Pan[1], JIANG Zhe[2], DING Yan[1]

[1](School of Computer, National University of Defense Technology, Changsha 410073, China)

[2](University of Cambridge, Cambridge CB2 1TN, UK)

Abstract: The major challenges traditional operating system (OS) design faces are the increasing number, diversity, and distribution scope of resources to be managed and the frequent changes in system state. However, the structures of existing OSs have become the biggest obstacle to solving the above problems as (1) tight coupling and centralization of the structure lead to poor flexibility and scalability and separate OS ecology; (2) contradiction between various capabilities, e.g., security and performance, due to the unitary isolation mechanism such as kernel-user isolation. Therefore, this study combines the hierarchical software bus (softbus) principles with isolation mechanisms to organize the OS and proposes a new OS model termed Yggdrasil. Yggdrasil decomposes an OS into component nodes connected by softbuses, whose communications are standardized to message passing via the softbus. To support the division of isolated states such as supervisor mode and different software hierarchies, Yggdrasil introduces bridge nodes for cascading and controlled communication between

* 基金项目: 国家自然科学基金 (U19A2060)
收稿时间: 2022-10-26; 修改时间: 2023-03-12; 采用时间: 2023-05-08; jos 在线出版时间: 2023-10-11

softbuses, and enhances the logical representation capability and scalability of OS through self-similar topology. Additionally, the simplicity and hierarchy of the softbus help to achieve decentralization. To verify the feasibility of Yggdrasil, the study builds hierarchical softbus model for OS (HiBuOS) and demonstrates the feasibility of developing a new OS based on Yggdrasil's ideas through three specific designs: (1) designing and planning a hierarchical softbus structure according to the scale and requirements of the target operating system; (2) selecting specific isolation and communication mechanisms to instantiate bridge nodes and softbuses; (3) realizing OS services based on the hierarchical softbus style. Finally, the evaluation shows that HiBuOS has notable potential and advantages to enhance system scalability, security, performance, and ecological development without significant performance loss.

Key words: hierarchical software bus; operating system structure model; loose coupling; diverse isolation mechanisms; self-similarity

操作系统是管理硬件资源, 控制任务运行, 改善人机界面和为应用软件提供支持的一种系统软件[1]. 随着互联网技术的快速发展和普及, 操作系统的设计面临着以下挑战: (1) 操作系统面向的计算环境在从单机, 局域网平台向互联网平台延伸, 其概念不再局限于单机操作系统, 还包括处理机群协作的网络操作系统[2], 进而如何高效管理分布式资源以及利用分布式计算能力成为一个新的发展主线[3]; (2) 一方面具有海量, 异构, 自主等特性的新场景正在形成[2], 例如面向工业的操作系统; 另一方面操作系统管理硬件多样性和数量显著增加, 多样化的硬件组合出现在个人计算到数据中心的各种应用环境中[4,5], 因此, 操作系统支持按需定制和使用将更符合未来的发展趋势; (3) 操作系统需要对资源安全性进行区分, 通过充分利用软硬件技术进一步实施有目标性的防护. 然而, 现有操作系统, 例如 Linux, L4, Windows 等, 难以有效地应对这些挑战. 首先现有操作系统依赖其上的中间件系统来解决与网络相关的部分挑战. 虽然这种层次化技术有利于降低控制功能的复杂性, 但是通常以性能为代价. 且对应用软件的开放性和动态性支持能力不足[3]. 其次大多数操作系统具有强耦合和中心化的特点, 组件紧密结合在一起, 特别是核心功能总是集中在一个组件中, 例如内核. 这不仅限制了系统的可扩展性, 安全性, 灵活性和可维护性[6−8], 而且操作系统在完成后难以适配新的软硬件机制. 受架构影响, 每个操作系统都构建与其他操作系统不兼容的生态, 造成了资源的浪费和新操作系统成长的困难[9]. 最后, 大多数操作系统认为系统中仅有特权态 (内核态) 和非特权态 (用户态) 这两种状态, 并通过将功能组件分配到不同的特权态来调整系统的安全性和可扩展性[10−13]. 这种单一的隔离方式通常会导致安全性, 可扩展性, 性能等各种能力之间的矛盾. 例如, 微内核思想强调除了一个极其精简的内核外, 其他的所有服务都设计为独立的核外进程模块, 这虽然强化了系统服务的安全性, 但也带来了系统服务的性能缺陷.

计算机硬件总线[14]的设计为我们解决上述操作系统设计面临的挑战提供了很好的启发: (1) 在统一的体系结构框架下 (即总线和接口规范), 任何计算机厂商生产的配件, 只要遵循标准的接口规范, 都可以集成到系统中[14]; (2) 桥实现了通信能力不一样的分级互联, 例如 PCI-to-Cardbus 桥的主要作用是通过 PCI 总线扩展 Cardbus 总线[15]. 我们拟借鉴硬件总线灵活互联, 动态路由, 分级通信等特性设计一种多样化的隔离/互连建模方法来描述操作系统结构, 将操作系统每一级别都看作隔离与通信的组合. 具体地, 单机操作系统是由具有对性能, 安全及可靠性等能力不同需求的软件模块分级互联组成, 其中, 模块划分由不同通信级别 (类似于硬件系统中不同类别的总线, 如 PCI, Cardbus 等) 实现, 内核态和用户态的划分仅作为层级之间的一种隔离方式; 分布式系统则是基于特定的拓扑由单机操作系统互联而成. 我们将模块之间的通信称作软总线, 其与传统软总线机制[16−20]的区别在于它不一定需要功能库, 协议栈, 驱动栈等共同组成的基础软件服务的支持, 能够直接用于操作系统各级的设计中.

本文借鉴硬件总线的树型结构[15], 结合简捷的层次化的软总线设计思想和多样化隔离机制来组织操作系统组件, 提出了一种新的操作系统结构模型——Yggdrasil, 如图 1 所示. Yggdrasil 将操作系统功能分解为松散耦合的组件节点, 将组件节点之间的通信标准化为经软总线的消息传递. Yggdrasil 还通过组合多条软总线的方式来描述复杂的系统结构和充分分类具有不同性能和安全要求的节点. 在操作系统隔离的视角上, Yggdrasil 引入了桥节点来实现多层软总线之间的级联和受控通信, 规定每条软总线内部使用统一的隔离方式, 软总线间通过桥节点结合多样化的隔离机制来平衡系统安全性和性能. 同时, 桥节点也实现了结构自相似, 进而使操作系统的逻辑表述能力和扩展性都得到了极大强化. 总线域定义于软总线上, 挂接在此软总线上的所有组件节点都属于同一个总线域. 总线域还拥有状态属性, 用于确定或控制隶属于该总线域的资源集合的共同状态. 此外, 软总线的简单性和层次性也有助于实现去中心化.

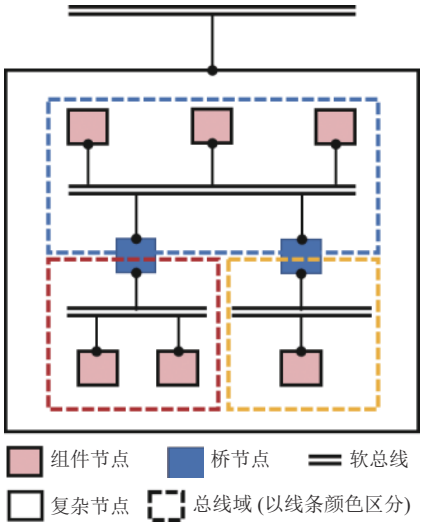

图 1　Yggdrasil 结构模型

从系统论的角度出发, 基于一个好的操作系统结构所实现的创新操作系统设计是提升系统核心能力的关键. 简而言之, Yggdrasil 同时具备了以下特点, 使操作系统在扩展性, 安全性, 性能, 适用性等方面都能满足高要求.

(1) 松耦合和去中心化. 任意组件在不了解内部通信机制的情况下就可以容易与操作系统连接, 并且不受其他组件漏洞的影响. 操作系统的结构是去中心化的, 故能避免系统能力瓶颈 (即中心节点) 的出现. 以松耦合和去中心化为基础, 操作系统不仅对故障具有一定的鲁棒性, 还能够更好地支持以组件为单元的按需定制, 加载和使用.

(2) 结构自相似性. 自相似性意味着操作系统结构从不同的“分辨率”上看都是相似的. 例如基于 Yggdrasil 模型的分布式系统中的一个组件节点可放大为一个基于 Yggdrasil 模型的单机操作系统. 自相似性有助于统一整个系统和组成系统的单个节点的刻画方式, 并支持操作系统结构可定制.

(3) 有效利用多样化 (硬件或软件) 的安全机制. 可以操作系统组件的边界上应用多样化的隔离机制, 以支持具有不同安全和性能要求的资源管理.

本文第 2 节介绍操作系统面临的挑战, 现有解决方案的不足与优化思路. 第 3 节描述一种基于分层软总线的通用操作系统结构模型及其松耦合, 去中心化, 自相似性和有效利用多种安全机制的特性. 第 4 节说明如何构建操作系统的分层软总线模型实例. 第 5 节从核心能力和性能测试两个部分评估了 HiBuOS. 最后是总结和未来工作展望.

HiBuOS 的源代码可在以下链接中公开获得: https://github.com/Yggdrasil-Model/HierarBusOS.

1　相关工作

操作系统面临的挑战源自计算机系统规模, 样式及复杂性的快速提升. 操作系统所承载的不再局限于单机节点中资源管理和任务调度等职能, 还包括处理物联网中各种各样的设备管理以及分布式环境中机群的协作. 其巨大的代码量, 功能模块数目, 相互作用关系和运行方式已具备了复杂系统的所有基本特点. 为了对操作系统结构进行剖析, 本文遵循软件工程的基本思想, 将操作系统看成是模块和通信的组合. 模块的边界定义会对通信的可选形态产生深刻影响, 因为边界往往是以某种隔离机制作为基础的, 而隔离机制又会导致相应通信方式的变化. 以熟知的宏内核或微内核架构为例, 它们从形态上来说都是由内核模块和核外模块组成, 是功能模块的边界划分以及消息通道的变化导致了两种结构在各种能力上显著差异. 近年来所出现的新型软硬件机制[21−25]引入了更为丰富的隔离边界和通信机制, 成为操作系统新的模块构建基础. 现有操作系统模型多以内核态-用户态作为模块划分标准, 这种非此即彼的划分方式难以满足模块对性能、安全、扩展等能力的不同需求, 可见基于两态划分的系统架构描

述模型过于简单, 无法详尽的反映操作系统的结构本质. 为此, 本文拟采用一种多样化的隔离/互连建模方法来描述操作系统结构模型. 这一解决思路中的关键问题有: 操作系统功能分解原则; 功能组件的组织连接方式和安全隔离机制. 本文将在第 1.1–1.3 节进行详细讨论.

1.1 操作系统功能分解

学术界和工业界已经提出了一些有关操作系统功能分解的建议.

基于微内核的操作系统 (图 2), 如 Mach[26]和 L4[27]以特权态和非特权态划分作为主要的安全基础, 将所有可能的功能组件如设备驱动, 文件系统等服务从内核中移除, 并转换为用户进程在用户态执行, 内核与用户进程之间的通过进程间通信交互. 这种方式虽然提高了可扩展性, 但是带来了显著的通信开销. 为此, Theseus[21]在同一特权级和同一地址空间, 利用语言层面的机制将整个操作系统的功能分散到多个拥有明确的定义以及运行时持久的边界的组件中, 并跟踪组件之间的相互依赖关系. 然而其关注的系统安全还属于可靠性的范畴, 即功能单元的缺陷不会对系统产生致命的影响, 难以应对私密性、完整性、可信性等范畴的安全威胁.

此外, 为了提高数据中心的资源利用率, 硬件弹性, 异质性支持能力和故障处理能力, 硬件分解也成为一个重要的趋势. IBM[28]和 dReDBox[29]将硬件资源打包在一个大箱子中, 并用 PCIe 等总线连接. 惠普[30]的规模是一个机架, 它用特有的相干网络连接系统级芯片 SoC (system on chip) 与非易失性存储器 NVM (non-volatile memory). 硬件打散后会带来许多优势[31]. 然而, 目前还没有支持这种硬件架构的操作系统. 为此, LegoOS[31]认为既然硬件已经被拆分, 那么操作系统也应该被拆分, 如图 3 所示. 它将传统操作系统的功能分解成低耦合的监视器, 每个监视器独立运行, 管理属于自己的硬件组件, 组件之间通过网络通信. 然而, 在当前环境下, 网络通信可能成为制约性能的瓶颈.

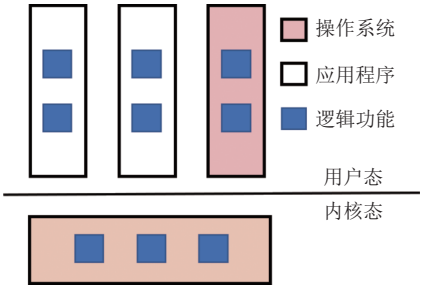

图 2 微内核结构

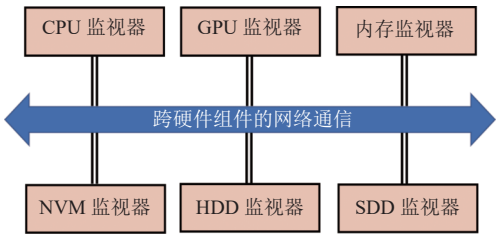

图 3 LegoOS 结构模型[31]

总而言之, 无论从软件方面还是硬件方面的发展角度来看, 将操作系统进行模块化分解已成为共识, 且都是在安全可靠, 灵活度, 以及性能等几个方面寻找平衡点. 现有的研究方案往往陷于多方面此消彼长的矛盾中, 怎样解决这些矛盾才是功能分解的关键. 通过对已有方案的集中分析, 我们注意到一个共同的矛盾点, 即这些方案都着眼于单一的模块划分标准或策略, 而这个单一标准会使模块间的安全性, 灵活性或性能等趋于僵化. 例如, 微内核思想强调除了一个极其精简的内核外, 其他的所有服务都设计为独立的核外进程模块, 这虽然强化了系统服务的安全性, 但也剥夺了系统服务的性能优化能力.

1.2 操作系统功能组件连接方式

消息传递是模块化操作系统通信的关键机制, 例如微内核系统对进程间通信 IPC (inter-process communication) 性能的极致追求一直是几十年来该领域的核心问题. 首先从模块化的视角来看, 若将操作系统中的模块作为节点, 节点之间的有限接口抽象为边或弧, 则操作系统结构可转化为一个复杂网络, 消息传递可允许操作系统部署众所周知的网络优化, 以更有效的利用互连[4]. Baumann 等人认为随着机器越来越像一个网络, 操作系统将不可避免地表现为一个分布式系统[5], 消息传递有助于推理系统的互连, 例如关于共享状态的哪些部分在什么时候被访问以及由谁访问等信息, 并且能更好地支持分布式的, 异构的资源管理[4]. 其次, 就系统性能而言, Chaves 等人[32]

研究了消息传递和共享数据结构, 发现在多内核操作系统中, 消息传递有利于性能权衡. 在分布式共享对象中[33], 对象的远程方法调用被编码为消息, 以提高通信效率. 在基于硬件分解的 LegoOS[31]中, 消息传递在网络带宽消耗方面要比维护跨组件一致性的方案有效得多. 并且消息传递系统的结构是模块化的, 它可以更容易地扩展和重新配置, 对故障也具有鲁棒性. 最后, 就消息传递的透明性和标准性而言, 软总线机制[16−20]作为传递消息的公共传输通道, 在消息传递的基础上增加了总线注册和路由机制等基础设施的支持, 发送端只需要向软总线发出消息而不用管消息被如何转发. 理论上, 任何符合总线接口规范的组件都可以集成到系统中与其他组件进行协作. Harmony[19]提出了分布式软总线, 并以此作为手机、平板、智能穿戴、智慧屏等分布式设备的通信基座, 为设备之间的互联互通提供了统一的分布式通信能力, 隐藏了网络方法和协议的实现细节.

综合这些研究, 消息传递的可靠性, 可控性和信道带宽构成了操作系统的能力根基, 并且良好定义的消息传递接口会使操作系统的模块设计更为简单, 通用. 现有操作系统的设计大都是以功能为核心的, 消息传递的设计作为其附属设计产物, 这无疑是造成其模块耦合性强, 中心化, 以及生态独立的主要原因. 此外, 现有的消息传递机制趋于复杂化, 不利于高效的操作系统实现.

1.3 安全隔离机制

隔离机制同消息传递事实上是一对伴生问题, 消息机制必然是基于某种隔离机制设计的. 现有基于消息传递的操作系统有 LegoOS[31], Barrelfish[4], Singularity[34]等. 其中, LegoOS 将硬件及管理硬件的软件分解, 并使用物理隔离的方式降低故障的传播范围. Barrelfish 采用了特权级机制, 具体来说就是把每个核上的操作系统实例分成一个特权态下的 CPU 驱动和一个用户态下的监控器, 监控器及 CPU 驱动封装了类似微内核中调度, 通信和低级别的资源分配等功能, 而设备驱动和系统服务 (如网络堆栈, 内存分配器) 等部分作为用户级进程运行. 两种特权态之间存在保护鸿沟, 分别处于两态的软件单元在相互通信协作的过程中需要付出较大代价. Singularity 在单一地址空间和单一权限级别依靠编程语言和内存安全实现隔离, 并结合静态验证和运行时检查来验证用户代码访问的内存区域.

当我们将功能组件视为点而消息传递视为连接功能组件的边时, 上述消息传递操作系统都没有充分发挥边-点结构的优势, 即利用多样化的软硬件隔离机制作为点接入的边界, 而是设计了操作系统指标难以均衡的单一的隔离机制. 为了满足不同组件节点对性能, 安全及可靠性等能力的不同需求, 需要在结构上对多种隔离机制进行灵活的支持, 例如不同层次的模块采用对应其安全和性能等需求的隔离机制.

1.4 思考与启发

综合现有操作系统设计在模块划分, 消息传递和隔离机制 3 方面的经验和不足, 本文认为可将操作系统功能组件之间的消息传递通道抽象为软总线, 并将其作为组件的组装架构. 各组件通过标准的软总线接口挂接在总线上, 并通过各自的连接件 (或称适配器) 向软总线发出请求. 软总线采用极简的协议对其进行解释并确定接收方的位置, 完成通信并实现互操作. 由于协议可以设计得非常简单, 可以将总线机制直接进行自包容的代码实现, 因而能够直接用于操作系统底层功能模块的高效和高可靠互连. 然而, 单一软总线结构 (即只有一条总线) 和星型网络结构有等同性, 即将软总线作为中心节点, 接入组件作为边沿节点. 这种结构给软总线留下了沉重的管理和协调负担, 所以它的应用场景有限. 例如, 它不适用于分布式计算环境. 因此, 我们应通过多条软总线组合的方式, 描述复杂的系统结构.

此外, 本文还提出了一种全新的思路, 即不针对具体的隔离机制的设计, 而是提供一种能够利用现有隔离机制的手段. 由于边-点结构没有提供区分边与边之间安全等级的途径, 我们引入了桥节点来实现分离软总线 (边) 之间受控的通信, 如图 1 所示, 其设计对应特定隔离方法提供的通信机制, 使得系统的安全性和性能设计更加灵活.

最后, 一条软总线可以通过桥节点扩展一系列子总线, 其子总线亦然, 进而形成了树状结构, 其自相似的拓扑特性使操作系统的逻辑表述能力得到极大强化. 具体地, 操作系统的任何组成部分都可以视为一系列采用软总线互连的子模块的协作组织, 当我们改变系统结构的“分辨率”时, 又可将子模块视为子子模块的协作, 此时软总线上的子模块可被放大为一个新的子总线结构, 这个过程在需要时可以不断迭代下去. 因此, 我们在操作系统设计

之初可以以较粗粒度借鉴或直接使用现有的模块, 只关注总线结构实现. 随着系统的成熟, 操作系统的设计实现可被不断地标准化. 这不但有利于系统的快速开发, 也有利于整合利用已有的或开源的代码成果, 消除生态的隔阂. 此外, 这种操作系统结构可以通过增加或裁减总线层适应到许多应用场景, 小到嵌入式操作系统, 大到分布式计算环境.

简而言之, 从海量异构资源的灵活的组织和隔离方式, 以及多样化的操作系统实现机制与形态方面, 基于分层软总线的操作系统结构模型能很好应对上述挑战.

2 Yggdrasil 的设计思想

Yggdrasil 将操作系统视为一个由众多功能和服务采用一种或多种通信机制所连接起来的用于管理各种计算资源和软件任务的软件系统. 因此, 操作系统中的要素可以划分为 4 个类别: 功能服务, 连接, 资源, 状态属性. Yggdrasil 旨在灵活、安全、高效地组织和管理上述 4 种要素. 其中, 功能服务和资源被视为 Yggdrasil 中的组件节点 (定义 1). 连接被视为 Yggdrasil 中的软总线 (定义 2) 和桥节点 (定义 3), 这二者的区别是前者负责组件节点间的消息传递, 后者负责实现软总线之间受控的通信. 状态属性用于确定或控制隶属于该总线域 (定义 4) 的资源集合的共同状态.

定义 1 (组件节点). 组件节点是对传统操作系统中各种系统服务, 功能, 以及应用等概念的具体设计. 每个组件节点一般包含两个逻辑单元, 即功能单元和总线适配器. 其中功能单元就是该节点在操作系统中所承担的功能的实现, 如果是应用程序, 则就是应用的功能. 总线适配器用于解析接收到的消息后调用功能单元提供的接口进行处理.

定义 2 (软总线). 软总线是组件节点的通信信道. 它包括控制器, 路由机制和总线接口. 其中总线接口是提供注册或注销组件节点, 传输数据的标准接口. 组件节点以及桥节点都必须通过这些标准总线接口接入和通信. 路由机制根据总线上所注册的节点信息对节点要求的消息传输进行路由, 当目标节点不属于此软总线时, 路由机制负责确认通往所属软总线的桥节点, 并将消息发送给桥节点, 从而转发到其他软总线. 控制机制包含了组件节点注册机制和注销机制, 负责将组件节点注册到软总线上, 包括登记节点的各项信息, 能力, 回调接口指针等, 还实现了对软总线状态的判断, 消息合理性的验证以及对总线服务能力和服务质量的综合控制.

定义 3 (桥节点). 桥节点包括上层桥接接口和下层桥接接口, 其中上层桥接接口与上层软总线相连接, 下层桥接接口与下层软总线相连接. 桥节点负责在分离的软总线之间实现受控的通信, 其设计对应特定隔离方法提供的通信机制.

定义 4 (总线域). 总线域定义于软总线上, 挂接在此软总线上的所有组件节点都属于同一个总线域. 总线域还拥有状态属性, 用于确定或控制隶属于该总线域的资源集合的共同状态. 例如: 对于分布式操作系统中某些基于网络互连的组件形成的总线域, 其状态可能表示共同的工作组; 对应于一个单机操作系统中的某个总线域, 其状态可以表示硬件上下文所属的状态和特定的地址空间.

2.1 松耦合且透明的通信方式

Yggdrasil 并不对操作系统作内核, 服务及应用等概念进行区分, 而是根据它们的特性要求, 将它们作为某种组件节点或者组件节点的集合挂接在不同层级的软总线上. 组件节点之间的通信被标准化为经软总线的消息传递. 软总线又将消息传递的接口与实现逻辑分离. 当需要通信时, 发送端只需要向其连接的软总线发出消息而无需关注消息被如何转发. 软总线拥有一套完备的路由机制, 负责查询注册表, 定位提供该服务的组件节点, 并传输服务请求. 当目标节点不属于此总线域时, 则需要通过桥节点转发到其他总线域中.

在基于 Yggdrasil 的系统中, 组件节点之间通信链接数是线性的, 并且由于接口的标准化和一致性, 通信的复杂度以及组件节点之间的松耦合程度会明显降低. 软总线具有功能简单, 结构层次化的特点, 因此能够摆脱成为瓶颈节点的嫌疑, 达到系统的去中心化. 这种分层和透明的通信机制有助于提高系统的可扩展性和异质性. 海量异构的组件节点 (如库, 二进制模块, 硬件设备等) 只要符合软总线的标准规范, 都可以集成到系统中与其他组件节点进行协作. 以松耦合和去中心化为基础, 操作系统不仅有助于避免故障传播, 提高系统可靠性, 还可以更好地解决

大规模和分布式计算环境带来的挑战.

2.2 多样化隔离机制的结合

分层的软总线结构模型 (边-桥-点) 提供了引入多种隔离机制的途径. 本文从两个级别来解释. 在软总线内部级别下, 组件节点之间采用同一种隔离机制进行隔离, 在软总线间级别下, 系统可以有效利用多种软硬件隔离机制, 并让桥节点承载这种机制, 如图 4 所示. 本文通过类比 Linux 来说明隔离机制被映射到桥节点的原因. Linux 利用地址空间将用户进程之间, 内核与用户进程之间相互隔离. 当一个用户进程请求一个内核服务时, 异常处理程序会切换上下文并转发服务请求. 类似地, 在 Yggdrasil 中, 当一个组件节点向另一软总线上的组件节点发送消息时, 桥负责切换软总线并传输该请求. 由此可见, 桥的实现对应基于某种隔离机制的通信.

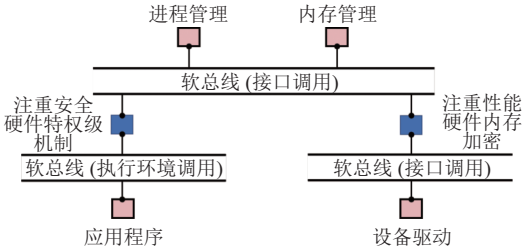

图 4 隔离机制利用实例图

这两种级别的隔离具有不同的侧重点: 同一软总线上的组件节点一般具有较一致的安全和可靠性等级, 注重频繁高效的交互, 跨软总线的组件节点具有不同的安全和可靠等级, 因此注重对资源和节点的访问控制能力的差异化, 桥节点的通信机制会更强调对于安全可靠的保护水平基础上的效率折中. 同时, 由于有多种隔离机制可供选择, 系统的安全和性能设计可以更加灵活.

2.3 结构自相似性

在 Yggdrasil 模型中, 复杂的组件节点可以分解为多个子节点, 这些子节点通过局部软总线进行连接, 如果子节点仍然比较复杂, 可以进一步分解. 如此分解下去, 整个系统形成了树状的拓扑结构. 众所周知, 树的分支也是树, 就像整个系统也可以作为一个组件节点, 通过更高层的软总线, 集成到更大的系统中. 如图 5 所示, 当我们改变系统“分辨率”的时候, 可以将分布式系统中一个软总线上的节点放大为一个新的子总线结构 (单机操作系统的总线结构). 因此, Yggdrasil 是自相似的. 基于此, Yggdrasil 有助于统一整个系统和组成系统的单个节点的刻画方式, 并可以伸缩变换到适应多种应用场景的形态, 具有广泛适应性.

QNX Neutrino[35]是一种健壮的, 可裁剪的基于 POSIX API 的开放系统, 这种系统广泛适用于从小型, 资源受限的嵌入式系统到高端的分布式计算环境. QNX Neutrino 是基于微内核和模块化结构设计的, 其中微内核充当一种软件总线的角色, 负责协调组件之间的交互和按需动态加载系统组件, 其结构如图 6 所示. 显而易见, QNX Neutrino 的系统结构是 Yggdrasil 众多形态中的一种 (单一软总线结构), 在一定程度上验证了 Yggdrasil 的合理性.

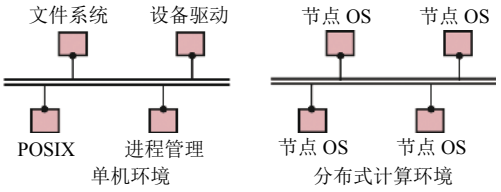

图 5 基于分层软总线模型的操作系统实例设计

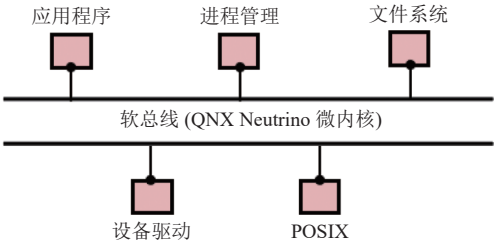

图 6 QNX Neutrino 的结构

3 操作系统的分层软总线模型实例构建

将软总线思想贯彻实现操作系统设计的每个层次是本文的一个思想创新, 也是难点所在. 已有的软总线设计大都构建在一些高层次功能库之上, 而无法直接用于操作系统底层功能组件的互连. 本文利用 Rust 语言[36]在 RISC-V 指令集架构[37]上构建了分布式系统的分层软总线模型实例——HiBuOS. 此实例粗略地实现了分布式系统中网络通信, 内存管理, 任务管理, 时钟管理, 串口设备管理 (键盘) 等部分功能, 目的在于: (1) 从分布式系统和构成分布式系统的单机操作系统的软总线层次结构设计验证 Yggdrasil 模型的广泛适用性; (2) 从软总线和桥节点的实例化验证 Yggdrasil 模型可通过利用多样化隔离机制来平衡系统的各种能力; (3) 从如何将传统操作系统服务作为组件节点连接在软总线上验证 Yggdrasil 模型的扩展性. 需要强调的是 Yggdrasil 并不对任何特定的互联拓扑、隔离方案、通信技术、路由机制作出假设, HiBuOS 只是其实现中的一种情况.

3.1 HiBuOS 的层次结构设计

系统软总线结构的整体设计可采用自顶向下的方式进行 (如图 7 所示).

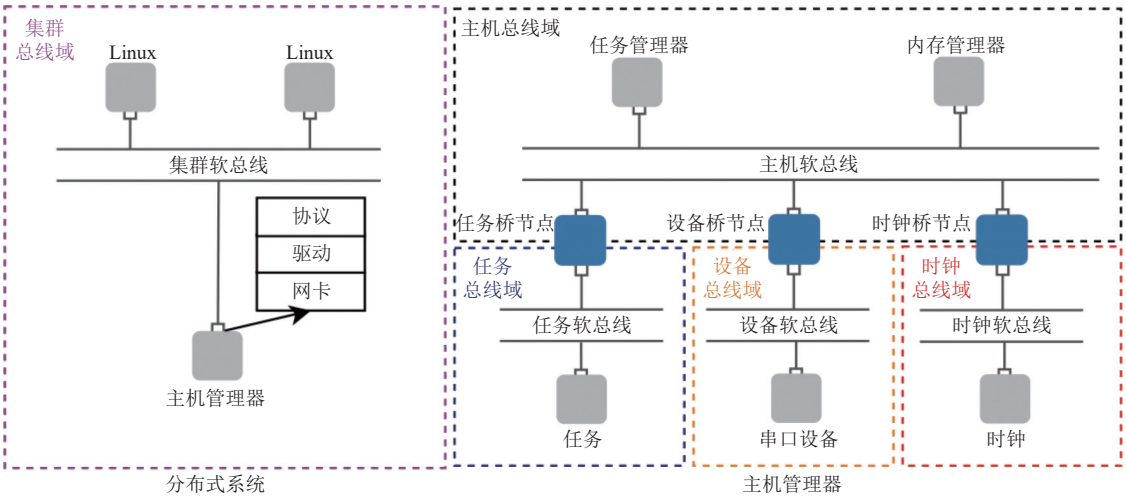

图 7 HiBuOS 的层次结构设计

针对目标操作系统所管理的资源类型、规模、范围和可用的隔离与通信方式, 规划目标操作系统应实现的软总线层级数量与各总线域, 这个过程中的设计原则如下.

原则 1: 根据各种隔离/通信机制进行自然的子系统划分, 再按照一个子系统的功能, 性能, 安全和可靠性需求规划软总线层级和总线域, 最后将多个组件节点采用对应级别软总线的接口进行逻辑连接.

原则 2: 每个软总线层级中的组件节点宜采用相同的隔离方法和通信方式, 并通过统一的接口和同一软总线进行连接 (采用标准化的统一定义), 组件节点中负责将内部逻辑同软总线接口进行对接的机制称为总线适配器. 通过对软总线和适配器的运用, 可以开发出能适配到该软总线的功能组件, 或者将已有软件逻辑改造为挂接在该软总线上的功能组件.

原则 3: 多层软总线中, 桥节点负责跨总线域的通信, 其实现应对应不同隔离方法提供的通信机制.

这套原则允许我们把整个复杂而庞大的系统设计分解, 后将这些子系统作为过渡, 逐步细化, 不仅有利于复杂系统的快速开发, 也有利于协同设计和已有的或开源的代码成果整合利用.

按照原则 1 所述, HiBuOS 针对一个分布式系统首先规划了一个软总线层次, 如图 7 左侧虚线方框中所示, 集群软总线用于连接管理不同主机的组件节点 (即单机操作系统, 本文后续将其称作主机管理器). HiBuOS 简单地将集群软总线视为网络传输通道, 主机管理器的总线适配器看作为网卡, 网卡驱动程序和协议栈的组合. 然后, 当改变 HiBuOS 结构的“分辨率”时, 又可以将集群软总线上的节点放大为一个新的子总线结构, 如图 7 右侧所示. 具体地, 我们将主机管理器再次分解为松耦合的组件节点, 包括: 任务管理器, 内存管理器, 任务, 时钟和串口设备等. 通

过衡量组件节点对性能和安全的不同要求及其自身特性, HiBuOS 将内存管理器和任务管理器连接到主机软总线上 (第 2 层). 第 3 层软总线中包含 3 类软总线. 一是任务软总线, 每条任务软总线可以连接一个或多个任务 (类似于 Linux 中一个进程所下属的多个线程). 二是设备软总线, 它用来连接串口设备. 三是时钟软总线, 它用来连接时钟. 从上述设计可以看出, 由于 Yggdrasil 模型具有自相似性, 软总线的层数是没有限制的. 因此基于 Yggdrasil 模型的系统不仅可以通过桥节点扩展软总线以支持大规模分布式系统, 还可以轻松地简化软总线和组件节点, 以适应小型嵌入式系统的设计.

3.2 桥节点和软总线的实例化

HiBuOS 将组件节点之间的通信简化为经软总线的消息传递, 其中, 组件节点的划分通过不同层级的软总线执行. 为了连接具有不同功能和特性要求的组件节点, 按照原则 2 和原则 3 所述, HiBuOS 选择了多种通信和隔离机制实例化软总线和桥节点.

(1) 集群软总线上的节点采用物理隔离, 与这种隔离相对应, 组件节点通过标准网络消息进行通信. 其实现方法可参考传统 CORBA (common object request broker architecture)[38,39]技术. HiBuOS 简单地将集群软总线视为网络传输通道, 主机管理器的总线适配器实现为网卡, 网卡驱动程序和协议栈的组合. 从硬件视角来看, 网卡负责接受计算机网络上发来的比特流存入内存并通知 CPU, 以及将组件节点的消息转换成比特流发送到计算机网络上. 从软件的视角来看, 消息的接受过程分为以下几步: 1) 当网卡收到数据包后, 它会根据消息队列提供的缓存区信息触发直接内存访问 DMA (direct memory access) 的数据复制操作, 随后产生硬件中断这个异步消息; 2) 网卡驱动程序负责处理此异步消息, 并将消息上抛到协议栈中处理; 3) 协议栈完成解封装处理并通过主机软总线将结果传递到对应的组件节点中. 类似地, 发送消息也分为 3 个阶段: 1) 组件节点向集群软总线发送消息; 2) 总线适配器中协议栈对此消息进行封装处理, 网络驱动程序将此消息复制到消息队列中缓存区信息对应的内存空间中; 3) 网卡通过 DMA 把内存空间复制到硬件发送队列中.

(2) 在主机软总线中, 节点注册表包含了组件节点标识符, 转发的下一级软总线标识符和回调接口指针. 原则上, 为了适应分布式计算环境中的合理资源分配, 节点注册表还可以包含服务能力, 服务距离等信息, 但是在 HiBuOS 中我们暂不考虑. 任何组件节点在加入主机管理器时必须调用注册函数, 随之主机管理器为每个组件节点分配一个唯一标识符. 主机软总线通过组件节点标识符来寻址一个组件. 具体地, 1) 主机软总线解析消息, 拿到组件节点标识符; 2) 查询节点注册表找到目标节点; 3) 判断目标节点所在的软总线标识符是否与本软总线匹配; 若匹配, 则通过其注册表中的回调接口指针转入目标节点的总线适配器进行处理, 此处出于性能考虑, 挂接在主机软总线的组件节点之间采用 (编程语言) 对象之间的接口调用来传输消息, 如图 8 所示; 若不匹配, 则通过桥节点转发到相应的软总线.

(3) 任务软总线和桥节点更注重行为不端的任务给系统带来的安全问题. 为确保主机管理器的安全, 任务不能访问任意的地址空间且不能执行某些可能破坏计算机系统的指令. 因此, 我们利用处理器设置了两个不同安全等级的执行环境: 任务总线域和主机总线域. 如图 9 所示. 当任务通过消息传递请求主机软总线上节点的服务时, 任务软总线利用执行环境调用将消息发送给任务桥节点, 核心代码如代码 1 所示. 桥节点负责跨总线域的通信, 自然地, 任务桥的实现应包含保存上下文, 切换硬件状态和地址空间, 消息传递等步骤. 可见在 HiBuOS 中, 我们利用不同的软总线对组件节点进行了划分 (任务软总线, 主机软总线), 传统的两态 (内核态, 用户态) 划分只是 HiBuOS 中桥节点实现的一种方式.

代码 1. 任务软总线消息传递核心代码.

```
llvm_asm!("ecall"
      : "={x10}" (ret)
      : "{x10}" (args[0]), "{x11}" (args[1]), "{x12}" (args[2]), "{x13}"(service_id), "{x17}" (node_id)
      : "memory"
      : "volatile"
);
```

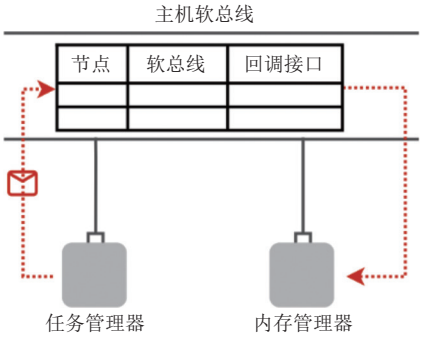

图 8 主机软总线实现示意图

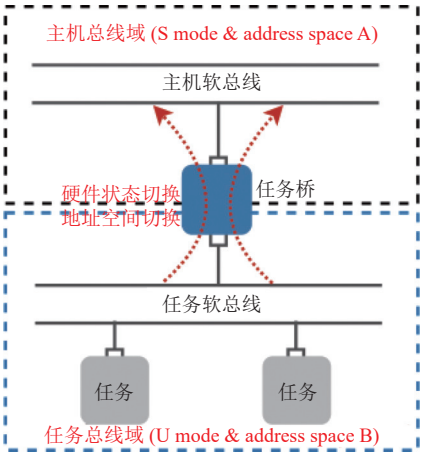

图 9 任务软总线和桥节点实现示意图

(4) 设备软总线负责连接串口设备, 如图 10 所示, 一般它需要完成如下一些功能: 1) 设备初始化, 包括对设备的初始配置, 设置好中断处理例程; 2) 处理设备发生的中断; 3) 根据其他组件的要求, 给串口设备发出消息 (命令). 具体地, HiBuOS 管理的计算机硬件系统是一台虚拟 RISC-V 64 计算机 (QEMU RISC-V-64 virt machine). 其中, 每个设备连接到父设备, 最后构成了一个设备树. bootloader, 即 OpenSBI 或者 RustSBI 固件会完成包括物理内存在内的各外设的探测, 并将探测结果以设备树二进制对象 DTB (device tree blob) 的格式保存在物理内存中的某个地方. 当 bootloader 启动操作系统时, 会把放置 DTB 的物理地址将放在 a1 寄存器中, 然后跳转到操作系统的入口地址处继续执行. HiBuOS 在遍历 DTB 数据时, 会根据设备的类型, 给相应的设备软总线发送进行初始化工作的请求信息. 对于处理外设中断来说, 由于其异步于当前执行的指令, 因此可将硬件中断视为一种异步消息. 在 RISC-V 中, 与外设连接的输入输出控制器的一个重要组成是平台级中断控制器 PLIC (platform-level interrupt controller). 当 HiBuOS 接受到一个异步消息的时候, 它可以通过读 PLIC 的寄存器来了解是哪个设备发出的异步消息, 再交给相应的设备软总线处理此消息. 当任务请求串口设备输入输出操作时, 经过层层软总线和桥节点将消息传输到设备软总线上. 接着, 设备软总线将消息放入消息队列中并通过某种通知机制 (如写某个寄存器) 通知设备, 设备适配器接收到通知之后, 会从消息队列的位置中取出请求消息并处理. 最后, 设备适配器完成处理或者出错后, 会将结果发送给设备软总线, 设备软总线再将次结果返回给任务.

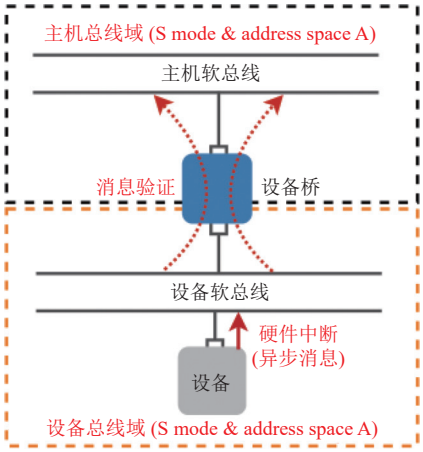

图 10 设备软总线和桥实现示意图

(5) 在 HiBuOS 中, 时钟也是一个组件节点. 作为一种硬件设备, 时钟发出的也是一个异步消息. 如图 11 所示, 此消息的处理时机与中断屏蔽有关. 具体地, 1) 在任务运行时, 任务软总线接收到了一个需要时钟软总线处理的异步消息; 2) 在合适的时机, 此消息会被路由到时钟软总线进行处理; 3) 时钟软总线通过时钟桥节点向主机软总线下的任务管理器发送任务调度请求.

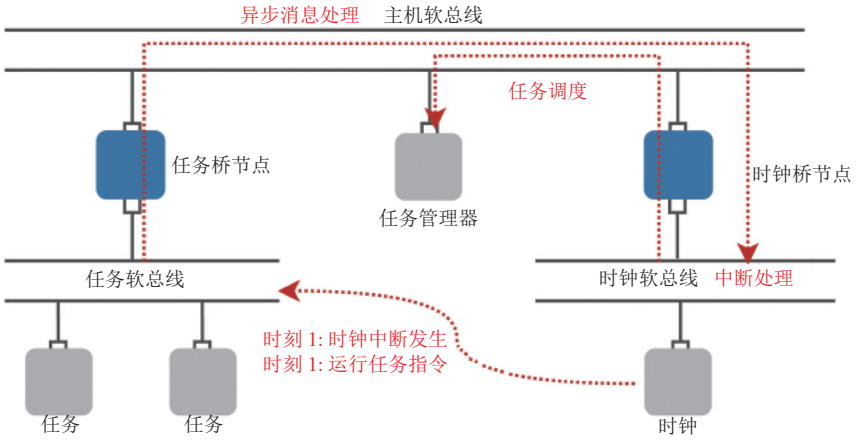

图 11 时间中断管理实现示意图

3.3 基于分层软总线风格的操作系统服务设计

HiBuOS 将传统操作系统的基本管理服务 (内存管理, 任务管理, 中断管理等) 转换为软总线上的组件节点, 具体分布在哪些总线域中将依据安全, 可靠, 性能等指标在第 3.1 节确定. 在实现组件节点时, 其功能单元可以单独设计也可以通过对成熟的已有操作系统模块源码进行改造获得. 单独设计时还可以继续遵从本文的方法, 对该组件节点进行更细致的多层总线域设计. 接下来, 本文将通过 3 个典型案例, 即内存管理, 异常/中断管理和任务管理, 来展示基于分层软总线风格的操作系统服务设计.

3.3.1 内存管理

HiBuOS 中的主机管理器将内存管理服务封装到主机总线上的组件节点中 (内存管理器), 如图 12 所示. 内存管理器的实现分为以下两个方面: (1) 内存管理器的功能单元, 包括: 动态内存分配, 虚拟内存管理, 这部分可通过对已有操作系统模块源码改造获得; (2) 内存管理器的软总线适配器, 负责组件节点与软总线的连接, 是实现组件节点的核心部分.

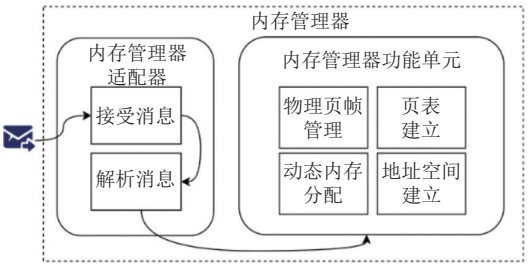

图 12 内存管理器示意图

就内存管理器的功能单元而言, 它利用栈式管理策略将空闲内存以单个物理页帧为单位管理起来. 具体地, 它保存了空闲内存的起始物理页号, 结束物理页号, 以后入先出的方式保存了被回收的物理页号的栈. 在分配的时候, 首先会检查栈内有没有之前回收的物理页号, 如果有的话直接弹出栈顶并返回; 否则的话 HiBuOS 从未分配过的物理页号区间上进行分配, 再将维护的起始物理页号加一. 内存管理器还将一块大内存空间作为初始的堆, 动态

地维护一系列空闲和已分配的内存块, 并提供在堆上分配和释放内存的函数接口. 其次, 主机管理器在启动之初就通过修改特定的控制与状态寄存器 CSR (control and status register) 启用了分页模式, 随后组件节点的访存地址会被视为一个当前地址空间中的一个虚拟地址, 需要内存管理单元 MMU (memory management unit) 查询多级页表完成地址转换才能访问相应的数据. 因此, 在我们的设计和构建中, 内存管理器为每个总线域分配了适当数量的物理内存帧以及建立了多级页表来存储虚拟内存和物理内存的映射关系, 对应的状态属性中保存了多级页表根节点所在的物理页号. 此后组件节点能够直接看到并访问的内存就只有其总线域中对应的地址空间, 且它的任何一次访存使用的地址都是虚拟地址. 鉴于组件节点只能通过虚拟地址读写自身的地址空间, 完全无法窃取或者破坏其他软总线上的数据, 进一步保障了系统的安全性和稳定性.

软总线适配器负责解析接收到的消息, 并调用对应函数接口进行处理. HiBuOS 利用 Rust 语言特性, 规定能注册到软总线上的结构体, 即内存管理器, 必须实现 Busadapter trait. Busadapter 包含一个必须实现的抽象接口, 即实现对消息的处理. 具体地, 消息作为结构体, 包含节点标识符, 服务标识符, 服务请求参数. 当消息被转发到内存管理器中时, 总线适配器根据服务标识符调用功能单元中相应的内存管理服务处理请求, 核心代码如代码 2 所示.

代码 2. 软总线适配器核心代码

```
pub trait Busadapter {
  fn handle(&self, service_id: usize, body: [usize;3])->isize;}
pub struct MemoryManager;
  impl Busadapter for MemoryManager {
    fn handle(&self, service_id: usize, body: [usize;3])->isize{
      match service_id {
        MEMORY_WRITE => {}
        MEMORY_READ => {}
        _=>panic!("Unsupported memory service id"), }}
}
```

3.3.2 异常/中断管理

在 RISC-V 架构中, 异常 (exception) 和中断 (interrupt) 都是一种陷入 (trap), 其区别是对于某个处理器核而言, 异常与当前 CPU 的指令执行是同步的, 异常被触发的原因一定能够追溯到某条指令的执行; 而中断则异步于当前正在进行的指令, 也就是说中断来自哪个外设以及中断如何触发完全与处理器正在执行的当前指令无关. HiBuOS 中异常的处理可通过任务总线域与主机总线域的通信过程说明: (1) 任务软总线利用执行环境调用向主机软总线发送消息; (2) 任务桥节点保存并切换上下文信息, 再向主机软总线传递消息. 在 HiBuOS 中, 中断异步于当前指令, 其管理过程可通过时钟中断的处理过程说明: (1) 在任务运行的同一时刻, 任务软总线收到了一个异步消息; (2) 在中断没有屏蔽的条件下, 时钟软总线会接收到此消息并进行处理.

3.3.3 任务管理

HiBuOS 将任务软总线作为库文件与应用程序一起编译得到二进制文件, 并将任务软总线的消息处理函数作为入口地址. 任务管理服务被封装到主机软总线上的组件节点中 (任务管理器). 任务管理器的实现分为以下两个方面: (1) 任务管理器的功能单元, 包括任务调度和任务切换; (2) 任务管理器的软总线适配器, 其实现思想与内存管理器的相同.

首先, 任务管理器采用简单的时间片轮转算法来对任务进行调度. 具体地, 任务管理器维护一个任务队列, 每次从队头取出一个任务执行一个时间片, 然后将其加入队尾, 以此类推到所有任务执行完毕. 最后, 任务切换实际上是来自两个任务在主机总线域中的陷入控制流之间的切换, 此过程包含两个主要的部分, 即特权级切换和陷入控制流的切换. 任务桥节点的实现对应着特权级切换. 陷入控制流的切换由任务管理器中一个特殊的函数

(switch) 负责. 在调用 switch 函数之前, 主机总线域的栈从栈底到栈顶分别保存了任务陷入上下文以及主机总线域下组件节点在陷入处理的过程中留下的调用栈信息. 任务切换的流程主要如图 13 所示: (1) 当陷入控制流 A 在调用 switch 之前, 其主机总线域栈上只有陷入上下文和陷入处理函数的调用栈信息, 而陷入控制流 B 是之前切换出去的; (2) 任务 A 在其任务上下文中保存当前寄存器的信息; (3) 读取指向的 B 任务上下文, 并根据 B 任务上下文保存的内容来恢复部分寄存器; (4) 保存栈顶指针的寄存器换到了任务 B 的主机总线域栈, 进而实现了控制流的切换. 此时, 任务管理器向任务 B 发送消息, 任务桥节点保存和切换上下文信息, 跨总线域转发消息, 最后当任务软总线接收到此消息后, 任务 B 开始运行.

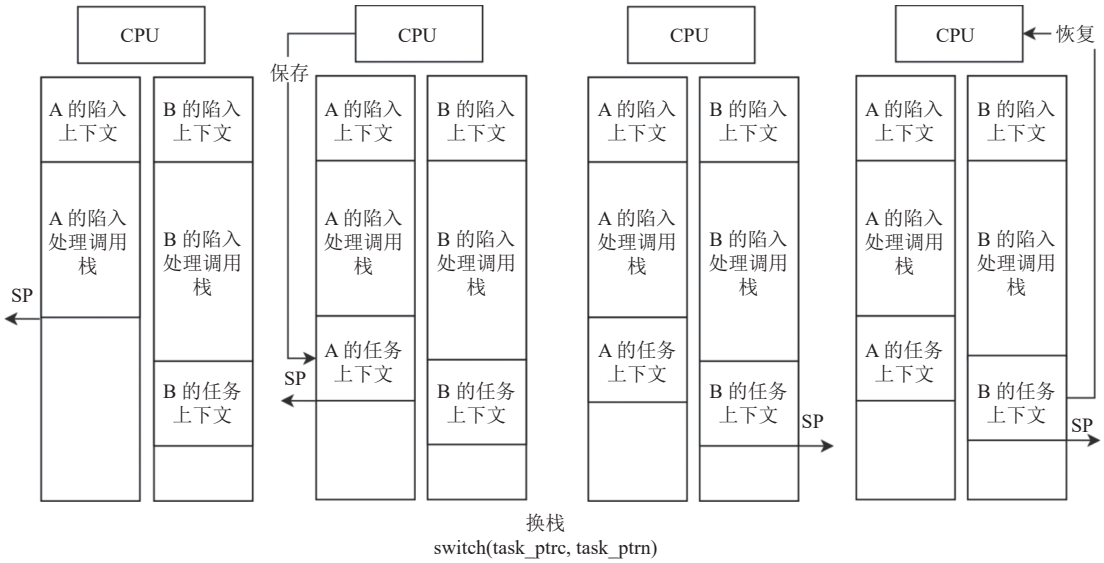

图 13 任务切换流程示意图

4 评 估

4.1 核心能力分析

● 扩展性: 本文从两个角度来评估 HiBuOS 的可扩展性, 即在系统中增加新服务的修改复杂度和系统应对负载 (组件节点) 增加的能力. 首先, 我们在构建 HiBuOS 过程中增加任务管理服务的修改可分为以下 3 步: 1) 利用现有任务管理的源码作为组件节点的功能单元, 将其对外的服务请求转换为基于总线接口的消息通信; 2) 增加软总线适配器以解析和处理其他组件节点发来的服务请求消息, 并调用功能单元中相应的服务; 3) 更新其挂接软总线中节点注册表的信息. 从上述构建过程我们可以看出在 HiBuOS 中增加新服务的核心在于软总线适配器的实现以及节点注册表的更新, 因为功能单元可直接对现有源码改造获得. 且由于组件节点之间松耦合的连接关系, 其他组件节点很少会受到影响, 故 HiBuOS 的修改复杂度远小于紧耦合的系统. 其次, 基于 Yggdrasil 模型的系统中组件节点连接的一个核心问题是如何寻址目标组件节点. HiBuOS 根据软总线中节点注册表的信息对消息进行路由, 这带来了一个问题, 即节点注册表的大小是否会成为扩展组件节点的瓶颈. 我们通过一个具体的案例来讲述 HiBuOS 如何应对负载 (组件节点) 增加. 假设系统最初只有一条软总线, 它需要扩展 M 个组件节点以满足任务的要求 (例如 MapReduce 处理海量数据的并行运算时需要多个工作机来分配任务). 首先我们规定一条软总线上最多连接 10 个组件节点和 9 条软总线, 这样规定的原因是为了方便组件节点寻址. 具体地, 我们将组件节点的编号格式设置为 $0x_1x_i\ldots x_ny$, 其中 0 代表最上层软总线 (即系统初始软总线), n 表示第 n 层软总线, x_i 表示第 i 层软总线的标识符, 相应的, 软总线标识符格式为 $0x_1x_i\ldots x_n$, 如图 14 所示. 由公式 (1) 可知扩展 M 个节点需要 $\log_9\frac{4M+5}{5}$ 条软总线. 软总线仅保存自身扩展的软总线信息与其每一级祖先软总线的信息 (例如标识符为 $0x_1\ldots x_i$ 的软总线要保存

标识符为 0, $0x_1,\ldots,0x_1\ldots x_{i-1}$等软总线的信息), 那么寻址的过程可概括为: 1) 找到源节点与目标节点的最长公共前缀$0x_1\ldots x_i$; 2) 源节点将消息转发到标识符为$0x_1\ldots x_i$的软总线上, 再由此软总线向下转发, 假设目标节点标识符为$0x_1x_i\ldots x_ny$, 寻址则需经过$n+1-i$条软总线, 节点注册表大小的增长在$O(\log M)$级别. 可见随着组件节点的增加, 软总线内部的节点注册表增长趋缓, 即系统具备扩展性. 例如, Gnutella 为了解决文件查询可扩展性的问题, 提出了分布式哈希表, 其设计原则是网络中每个节点只需要与另外的$\log N$个节点互动来生成分布式哈希表中的键值, 因此系统可扩展到容纳数以百万计节点.

$$10\times\left(9^0+9^1+\ldots+9^{n-1}\right)=M \tag{1}$$

● 安全性/性能: 基于 Yggdrasil 的 HiBuOS 非关注的是更好的隔离水平, 而是提供了一种性能和安全灵活可调的保护手段. HiBuOS 中主机管理器的分层软总线模型如图 15 所示, 当我们将系统服务桥节点实现为基于硬件特权级切换的通信时, 主机管理器等同于基于微内核的系统设计; 当我们将系统服务桥节点实现为函数调用时, 主机管理器类似于基于宏内核的系统设计. 为了协调系统服务的隔离性和性能, 我们被新型隔离机制——UnderBridge[40]所吸引. 其核心思想就是在特权模式下构建隔离的域, 提供高效的跨域交互, 使得用户空间的系统服务可以在这些域中运行. 由于系统服务运行在内核, 因此可以避免昂贵的与内核通信的代价. 当我们利用 UnderBridge 设计实现系统服务桥节点时, 主机管理器等同于基于 UnderBridge 的系统设计. 有研究测试了基于 UnderBridge 系统 (ChCore[40]) 的性能, 结果表明与微内核中的原生 IPC 相比, UnderBridge 设计将速度提高了 12.3 倍以上, 如图 16 所示. 可见, 桥节点有助于新型隔离和安全机制的融合和替换, 能利用未来新技术[41,42]提升操作系统的各种能力.

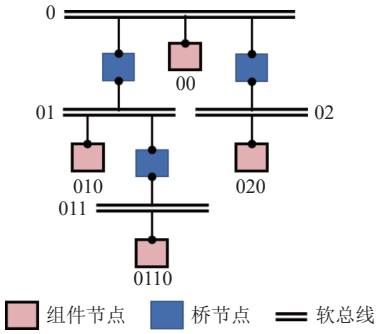

图 14 编码示意图

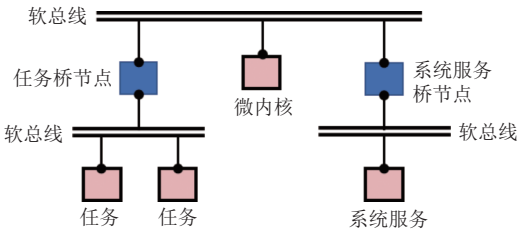

图 15 主机管理器的分层软总线模型

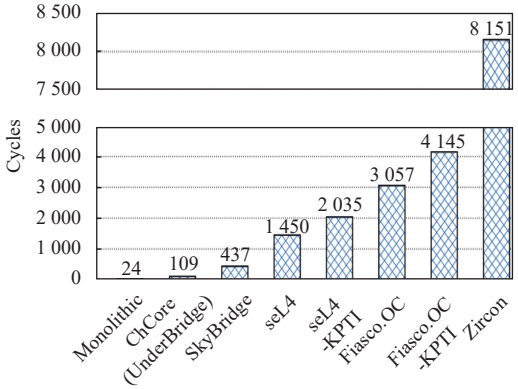

图 16 系统服务 IPC 的往返延时[40]

● 生态发展: 生态是超越计算机软硬件层次之上的存在. 新的操作系统要想获得成功, 必须能够利用开源软件或者已有成功操作系统的成果, 以避免陷入缺乏应用和开发者的尴尬境地. Yggdrasil 模型以松散的耦合和层次关系组织操作系统组件节点, 所以异构节点, 如开源软件或主流操作系统内核, 只要遵循软总线的接口标准, 就可以

集成到 HiBuOS 中. 并且从结构上来说, Yggdrasil 具有自相似性且对隔离和通信机制不做假设, 因此 HiBuOS 可以通过增加或裁减总线层级数量与各总线域的方式适应到许多应用场景. 总之, 任何现有操作系统生态中的组件都可以很容易地加入系统, 并且由于系统可配置的结构, 它能够适应多种应用场景, 小到嵌入式操作系统, 大到分布式计算环境. HiBuOS 在实现良好的生态发展, 促进学术界和产业界的有效整合方面拥有巨大的潜力.

4.2 HiBuOS 性能测试

本节旨在验证消息传递并未剥夺系统性能优化能力 (微内核设计的缺陷) 来说明将 Yggdrasil 应用于操作系统各级设计的可行性.

(1) 消息传递开销

我们测试了两种消息传递情况的开销, 即跨总线域的消息转发和同一软总线上的消息转发. 第一, 任务向任务管理器发送消息, 任务管理器的适配器处理完消息后立即返回, 这个过程包含了下述步骤的时间总和: (1.1) 任务软总线上的控制器处理和消息路由; (1.2) 任务桥节点跨总线的消息转发 (即基于硬件特权级切换的通信); (1.3) 主机软总线上的控制器处理和消息路由; (1.4) 任务管理器处理消息; 接着重复步骤 (1.3)、步骤 (1.2)、步骤 (1.1). 第二, 任务管理器向内存管理器发送消息, 内存管理器的适配器处理完消息后立即返回, 这个过程包含了下述步骤的时间总和: (2.1) 主机软总线上的控制器处理和消息路由; (2.2) 内存管理器处理消息; 重复步骤 (2.1) 返回. 结果如表 1 所示, 消息传递开销在可接受的合理范围内.

表 1 消息传递开销

行为	实时计数器周期数
任务向任务管理器发送消息 & 任务管理器的适配器处理消息后直接返回	124
任务管理器向内存管理器发送消息 & 内存管理器的适配器处理消息后直接返回	7
运行新任务——“hello world”	17383

(2) LMBench 基准测试

我们在 Linux 和机器管理器上运行选定的 LMBench 基准测试[43], 即部分系统调用, 任务创建, 内存映射, 写内存. 其中系统调用涉及往返于 Linux 内核地址空间, 或者往返于机器管理器中的主机软总线的过程, 是任务获取关键服务 (Linux 中的内核服务) 的一种途径. 任务创建测量的是创建一个新任务并让其运行的时间总和, 这个过程是命令行窗口程序 (shell) 内循环中的实体. 写内存测量的是映射, 取消映射大小为 4 KB 的内存的时间, 这个过程是创建一个新任务的基础. 结果如图 17 所示, 我们并不是说 HiBuOS 的性能普遍优于 Linux 等现有操作系统, 因为这两个系统具有不同规模和优化策略, 但我们的结果可以表明消息传递没有为系统带来明显的性能损耗.

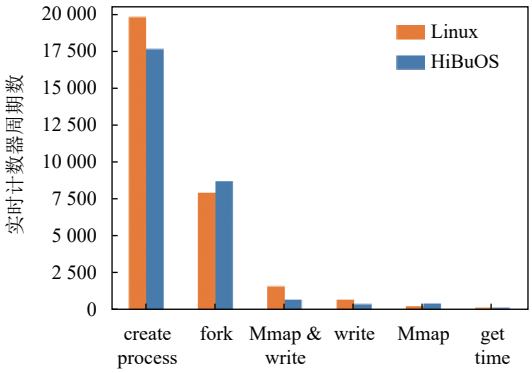

图 17 LMBench 基准测试结果

5 总 结

本文结合软总线设计原则和多样化隔离机制来组织操作系统, 提出了一种新型的操作系统结构模型 Yggdrasil.

Yggdrasil 通过软总线上的消息传递来实现系统组件之间的交互, 这种透明的通信方式降低了系统组件之间的耦合程度, 且软总线的简单性和层次性有助于实现去中心化, 进而系统的可扩展性和可靠性得到显著提升. Yggdrasil 引入桥节点来实现分层软总线间的受控通信. 桥节点作为隔离的边界, 有助于利用多样化的隔离机制来平衡系统能力. 此外, Yggdrasil 的结构自相似性使其具备广泛的适用性, 它不仅支持大规模的系统, 也被用于嵌入式小系统的设计. 为了评估 Yggdrasil 模型的能力, 本文设计并构建了分布式系统的分层软总线模型实例 HiBuOS. 本文的工作包括: (1) 层次结构设计; (2) 各级软总线和桥节点的实例化; (3) 操作系统服务设计与实现; (4) 测试与评估. 结果表明, HiBuOS 没有明显的性能损耗, 同时在提高系统的可扩展性, 安全性, 性能, 软硬协同和生态发展方面具有很大的潜力. 总而言之, 本文初步探索了 Yggdrasil 的可行性, 但未能将它与同新兴计算需求进行结合创新, 从而没有充分说明此模型的优势和用武之地. 后续工作将从模型自身的深度和其在特定应用场景下的探索两个方面继续创新, 更全面地评估 Yggdrasil 的核心能力.

References:

[1] Zhang XX. Encyclopedia of Computer Science and Technology. 2nd ed., Beijing: Tsinghua University Press, 2005 (in Chinese).

[2] Research report on the practice and prospects of ubiquitous operating systems (in Chinese). 2022. https://max.book118.com/html/2022/0823/6051115212004225.shtm

[3] Mei H, Guo Y. Network-oriented operating systems: Status and challenges. Scientia Sinica: Informationis, 2013, 43(3): 303–321 (in Chinese with English abstract). [doi: 10.1360/112012-413]

[4] Baumann A, Barham P, Dagand PE, Harris T, Isaacs R, Peter S, Roscoe T, Schüpbach A, Singhania A. The multikernel: A new OS architecture for scalable multicore systems. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: ACM, 2009. 29–44. [doi: 10.1145/1629575.1629579]

[5] Baumann A, Peter S, Schüpbach A, Singhania A, Roscoe T, Barham P, Isaacs R. Your computer is already a distributed system. Why isn't your OS? In: Proc. of the 12th Conf. on Hot Topics in Operating Systems. Monte: USENIX Association, 2009.

[6] Nikolaev R, Back G. VirtuOS: An operating system with kernel virtualization. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. New York: ACM, 2013. 116–132. [doi: 10.1145/2517349.2522719]

[7] Ganapathi A, Ganapathi V, Patterson D. Windows XP kernel crash analysis. In: Proc. of the 20th Conf. on Large Installation System Administration. Washington: USENIX Association, 2006. 149–159.

[8] Murphy B. Automating software failure reporting: We can only fix those bugs we know about. Queue, 2004, 2(8): 42–48. [doi: 10.1145/1036474.1036498]

[9] Herder JH, Bos H, Gras B, Homburg P, Tanenbaum AS. MINIX 3: A highly reliable, self-repairing operating system. ACM SIGOPS Operating Systems Review, 2006, 40(3): 80–89. [doi: 10.1145/1151374.1151391]

[10] Härtig H, Hohmuth M, Liedtke J, Schönberg S, Wolter J. The performance of μ-kernel-based systems. In: Proc. of the 16th ACM Symp. on Operating Systems Principles. Saint Malo: ACM, 1997. 66–77. [doi: 10.1145/268998.266660]

[11] Elphinstone K, Heiser G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In: Proc. of the 24th ACM SIGOPS Symp. on Operating Systems Principles. Pennsylvania: ACM, 2013. 133–150. [doi: 10.1145/2517349.2522720]

[12] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe E, Engelhardt K, Kolanski R, Norrish M, Sewell M, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: ACM, 2009. 207–220. [doi: 10.1145/1629575.1629596]

[13] Liedtke J. Improving IPC by kernel design. In: Proc. of the 14th ACM Symp. on Operating Systems Principles. Asheville: ACM, 1993. 175–188. [doi: 10.1145/168619.168633]

[14] Markussen J, Kristiansen LB, Borgli RJ, Stensland HK, Seifert F, Riegler M, Griwodz C, Halvorsen P. Flexible device compositions and dynamic resource sharing in PCIe interconnected clusters using device lending. Cluster Computing, 2020, 23(2): 1211–1234. [doi: 10.1007/s10586-019-02988-0]

[15] Wang Q. Introduction to PCI Express Architecture. Beijing: China Machine Press, 2010 (in Chinese).

[16] Zhang SK, Wang LF, Yang FQ. Hierarchical message bus-based software architectural style. Science in China Series: Information Sciences, 2002, 45(2): 111–120 (in Chinese with English abstract). [doi: 10.3969/j.issn.1674-7259.2002.03.015]

[17] Yan JP, Zhang Y, Cheng LL. Development and application of software bus technology. Radio Engineering of China, 2008, 38(11): 61–64 (in Chinese with English abstract). [doi: 10.3969/j.issn.1003-3106.2008.11.020]

[18] Zhang QY, Yuan ZT, Zhang DD, Ren L. Investigation of development technique of soft components based on distributional software bus.

Journal of Lanzhou University of Technology, 2005, 31(1): 93–96 (in Chinese with English abstract). [doi: 10.3969/j.issn.1673-5196.2005.01.025]

[19] Harmony. 2022. https://gitee.com/openharmony/docs

[20] Cheng JD. Connecting components with soft system buses: A new methodology for design, development, and maintenance of reconfigurable, ubiquitous, and persistent reactive systems. In: Proc. of the 19th Int'l Conf. on Advanced Information Networking and Applications (AINA 2005) Vol. 1 (AINA papers). Taipei: IEEE, 2005. 667–672. [doi: 10.1109/AINA.2005.139]

[21] Boos K, Liyanage N, Ijaz R, Zhong L. Theseus: An experiment in operating system structure and state management. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2020. 1–19

[22] Hua ZC, Yu Y, Gu JY, Xia YB, Chen HB, Zang BY. TZ-Container: Protecting container from untrusted OS with ARM TrustZone. Science China Information Sciences, 2021, 64(9): 192101. [doi: 10.1007/s11432-019-2707-6]

[23] Santos N, Raj H, Saroiu S, Wolman A. Using ARM TrustZone to build a trusted language runtime for mobile applications. In: Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Salt Lake City: ACM, 2014. 67–80. [doi: 10.1145/2541940.2541949]

[24] Sartakov VA, Vilanova L, Pietzuch P. CubicleOS: A library OS with software componentisation for practical isolation. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2021. 546–558. [doi: 10.1145/3445814.3446731]

[25] Ding Y, Dong P, Li ZP, Tan YS, Huang CL, Wei LF, Zuo YD. SLR-SELinux: Enhancing the security footstone of SEAndroid with security label randomization. Wireless Communications and Mobile Computing, 2020, 2020: 8866996. [doi: 10.1155/2020/8866996]

[26] Shi R, Zeng XP. The design of microkernels and a comparison of three implementations. Computer & Digital Engineering, 1998, 26(4): 61–68 (in Chinese with English abstract).

[27] Liedtke J. On micro-kernel Construction. In: Proc. of the 15th ACM Symp. on Operating Systems Principles. Copper Mountain: ACM, 1995. 237–250. [doi: 10.1145/224056.224075]

[28] Chung IH, Abali B, Crumley P. Towards a composable computer system. In: Proc. of the 2018 Int'l Conf. on High Performance Computing in Asia-Pacific Region. Chiyoda: ACM, 2018. 137–147. [doi: 10.1145/3149457.3149466]

[29] Katrinis K, Syrivelis D, Pnevmatikatos D, *et al*. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In: Proc. of the 2016 Design, Automation & Test in Europe Conf. & Exhibition. Dresden: IEEE, 2016. 690–695.

[30] The machine: A new kind of computer. 2022. https://www.hpl.hp.com/research/systems-research/themachine/

[31] Shan YZ, Huang YT, Chen YL, Zhang YY. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In: Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad: USENIX Association, 2018. 69–87.

[32] Chaves EM Jr, Das P, LeBlanc TJ, Marsh BD, Scott ML. Kernel-kernel communication in a shared-memory multiprocessor. Concurrency: Practice and Experience, 1993, 5(3): 171–191. [doi: 10.1002/cpe.4330050302]

[33] Bal HE, Bhoedjang R, Hofman R, Jacobs C, Langendoen K, Rühl T, Kaashoek MF. Performance evaluation of the orca shared-object system. ACM Trans. on Computer Systems, 1998, 16(1): 1–40. [doi: 10.1145/273011.273014]

[34] Hunt GC, Larus JR. Singularity: Rethinking the software stack. ACM SIGOPS Operating Systems Review, 2007, 41(2): 37–49. [doi: 10.1145/1243418.1243424]

[35] Li C. QNX Neutrino performance analysis. Microcomputer Applications, 2014, 30(3): 35–37 (in Chinese with English abstract). [doi: 10.3969/j.issn.1007-757X.2014.03.012]

[36] Klabnik S, Nichols C. The Rust programming language. 2023. https://doc.rust-lang.org/book/

[37] Waterman A, Lee Y, Patterson D, Asanović K. The RISC-V instruction set manual. Vol. 1: User-level ISA, Version 2.0. Berkeley: EECS Department, University of California, 2014.

[38] Bastide R, Palanque P, Sy O, Navarre D. Formal specification of CORBA services: Experience and lessons learned. In: Proc. of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. Minneapolis: ACM, 2000. 105–117. [doi: 10.1145/353171.353179]

[39] Emmerich W, Kaveh N. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In: Proc. of the 8th European Software Engineering Conf. Held Jointly with the 9th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Vienna: ACM, 2001. 311–312. [doi: 10.1145/503209.503259]

[40] Gu JY, Wu XY, Li WT, Liu N, Mi ZY, Xia YB, Chen HB. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In: Proc. of the 2020 USENIX Conf. on USENIX Annual Technical Conf. Berkeley: USENIX Association, 2020. 27.

[41] Dong P, Jiang Z, Burns A, Yan D, Ma J. Build real-time communication for hybrid dual-OS system. Journal of Systems Architecture,

2020, 107: 101774. [doi: 10.1016/j.sysarc.2020.101774]

[42] Jiang Z, Audsley NC, Dong P. BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems. In: Proc. of the 2018 IEEE Real-time and Embedded Technology and Applications Symp. (RTAS). Porto: IEEE, 2018. 75–84. [doi: 10.1109/RTAS.2018.00013]

[43] McVoy LW, Staelin C. Lmbench: Portable tools for performance analysis. In: Proc. of the 1996 USENIX Annual Technical Conf. San Diego: USENIX Association, 1996. 279–294.

附中文参考文献:

[1] 张效祥. 计算机科学技术百科全书. 第2版, 北京: 清华大学出版社, 2005.

[2] 泛在操作系统实践与展望研究报告. 2022. https://max.book118.com/html/2022/0823/6051115212004225.shtm

[3] 梅宏, 郭耀. 面向网络的操作系统——现状和挑战. 中国科学: 信息科学, 2013, 43(3): 303–321. [doi: 10.1360/112012-413]

[15] 王齐. PCI Express体系结构导读. 北京: 机械工业出版社, 2010.

[16] 张世琨, 王立福, 杨芙清. 基于层次消息总线的软件体系结构风格. 中国科学(E辑), 2002, 45(2): 111–120. [doi: 10.3969/j.issn.1674-7259.2002.03.015]

[17] 颜建平, 张焱, 陈路路. 软总线技术发展与应用研究. 无线电工程, 2008, 38(11): 61–64. [doi: 10.3969/j.issn.1003-3106.2008.11.020]

[18] 张秋余, 袁占亭, 张冬冬, 任磊. 基于分布式软件总线的软构件开发技术的研究. 兰州理工大学学报, 2005, 31(1): 93–96. [doi: 10.3969/j.issn.1673-5196.2005.01.025]

[26] 施嵘, 曾小平. 微内核操作系统设计方法及其设计实例的比较. 计算机与数字工程, 1998, 26(4): 61–68.

[35] 李存. QNX Neutrino实时操作系统性能分析. 微型电脑应用, 2014, 30(3): 35–37. [doi: 10.3969/j.issn.1007-757X.2014.03.012]

杨攀(1997－), 女, 硕士, 主要研究领域为操作系统, 系统结构.

江哲(1991－), 男, 博士, 主要研究领域为实时系统, 混合关键度系统, 片上网络, 虚拟化技术.

董攀(1978－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为系统软件, 系统安全, 实时操作系统.

丁滟(1977－), 女, 博士, 副研究员, CCF 杰出会员, 主要研究领域为操作系统, 系统安全, 可信云计算.