中国科学: 信息科学 2024 年 第 54 卷 第 3 期: 491–513
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
泛在操作系统理论、技术与开源生态构建专题 · 论文

BrickOS: 面向异构硬件资源的积木式内核

古金宇[1], 李浩[1], 夏虞斌[1], 管海兵[2], 丁佐华[3], 赵永望[4], 陈海波[1*]

1. 上海交通大学电子信息与电气工程学院软件学院, 上海 200240
2. 上海交通大学上海市可扩展计算与系统重点实验室, 上海 200240
3. 浙江理工大学信息学院, 杭州 310018
4. 浙江大学网络空间安全学院, 杭州 310007

* 通信作者. E-mail: haibochen@sjtu.edu.cn

收稿日期: 2022–10–24; 修回日期: 2023–09–30; 接受日期: 2024–01–16; 网络出版日期: 2024–03–11

国家自然科学基金重点项目 (批准号: 62132014)、国家杰出青年科学基金项目 (批准号: 61925206) 和国家自然科学基金青年科学基金项目 (批准号: 62202292) 资助

摘要 人机物融合的新兴领域需要新型操作系统内核以支持泛在计算, 对下管控海量异构硬件, 对上服务动态多变应用场景. 本文提出一种积木式内核架构 BrickOS, 可以根据使用场景灵活选择要加入内核的系统组件, 同时可以选择将系统组件运行在用户态以提供较好的安全性, 或者运行在共享地址空间的内核态中以提升性能. 为了保障运行在相同地址空间中的系统组件的安全性, BrickOS 为底层硬件的内存保护机制提供了统一的抽象, 并将其用于单地址空间的内存隔离. 测试结果表明 BrickOS 可以根据不同场景生成定制化内核, 并拥有较低的进程间通信 (inter-process call, IPC) 开销, 整体性能良好.

关键词 操作系统内核, 组件编排, 进程间通信, 内存隔离

1 引言

操作系统对下管理硬件资源, 对上构筑应用生态, 是信息产业的发动机和产业生态的基础支撑. 操作系统成功的关键在于生态, 泛在计算场景为重新定义操作系统生态提供可能. 美国计算机科学家 Weiser [1] 在 1991 年提出泛在计算 (ubiquitous computing) 的前瞻性设想, 所谓 “泛在计算” 可以理解为 “计算无处不在”. 近年来, 随着互联网技术的高速发展和延伸, 人类社会、信息系统、物理世界互相有机融合, 促进了工业互联网、智能物联网、网联汽车 V2X 等诸多新兴场景的涌现和发展, 这也验证了 Weiser 提出的 “泛在计算” 设想. 泛在计算带来的新蓝海给我国长期处于 “隔代追赶” 的操作系统研究提供了 “换道超车” 的机遇. 中国计算机科学家梅宏院士在 2018 年提出了 “泛在操作系统” 的概念 [2], 并指出新一代泛在计算模式和人机物融合场景需要新的、多样性的操作系统, 面向不同的计算设备、不同的计算系统, 或不同的应用模式和场景, 需要构建不同领域的泛在操作系统. 在高层软件架

引用格式: 古金宇, 李浩, 夏虞斌, 等. BrickOS: 面向异构硬件资源的积木式内核. 中国科学: 信息科学, 2024, 54: 491–513, doi: 10.1360/SSI-2022-0413
Gu J Y, Li H, Xia Y B, et al. BrickOS: specialized kernels for heterogeneous hardware resources (in Chinese). Sci Sin Inform, 2024, 54: 491–513, doi: 10.1360/SSI-2022-0413

构上, 一个具象化的泛在操作系统主要由操作系统内核和面向具体领域的操作系统框架构成. 本工作聚焦在泛在操作系统内核的构建技术.

硬件资源异构是泛在计算场景的重要特征之一. 传统计算场景中的计算平台往往趋于同构, 以传统云计算场景为例, 云厂商部署的计算平台基本都是 x86-64 体系结构的服务器; 云服务器在除体系结构外的计算资源方面也是以同构为主, 比如物理内存容量充足, 通常为几十甚至上百 GB, CPU 核心数量较多, 以多 CPU 和多硬件线程为主. 相比之下, 泛在计算场景中的硬件平台差异性显著, 以工业互联网场景为具体案例, 计算平台类型包括: 传统服务器用于大数据存储与处理等需求、工业 PC 机用于上位机或数字孪生等需求、自动引导车 (automated guided vehicles, AGV) 等场地内无人自主平台上的端处理器、工业产线上的嵌入式计算平台 (如用于控制的单片机) 等. 尤其是端计算平台和嵌入式计算平台的处理器体系结构种类很多, 例如 ARM Cortex-M 各系列芯片和 Cortex-A 各系列芯片; 这些平台在 CPU 特权级划分、是否配备内存管理单元以支持虚拟内存、物理内存容量大小等方面差异性很大. 目前, 这类工业互联网端平台中有部分尚未部署操作系统, 即仅部署单一专用程序 (缺乏可编程能力); 而已部署的操作系统多以 VxWorks[1)], QNX [3], NuttX[2)], FreeRTOS[3)], Windows 嵌入式版等不同国外操作系统为主 (缺乏自主可控, 不同操作系统造成碎片化问题, 缺乏统一抽象). 此外, 在异构硬件上运行的领域应用也可能具有异构需求. 例如, 在工业互联网场景下, 有运行在嵌入式处理器上的具有强实时要求的控制型应用, 有运行在富算力处理器上具有高吞吐需求的智能应用.

异构性对泛在操作系统研发提出 3 个方面的挑战. 挑战 1: 单一内核实现无法支撑资源差异极大的异构平台. 例如, 即便是应用生态极尽繁荣的 Linux 操作系统, 也无法向小内存、无内存管理单元 (memory management unit, MMU) 计算平台扩展. 挑战 2: 单一内核架构难以满足领域应用异构需求. 例如, 宏内核架构在性能方面具有优势, 而微内核架构在安全性可靠性方面具有优势, 适用于具有不同特征的领域应用. 挑战 3: 小批量难复制导致研发成本极高. 面向小规模领域研发定制化的操作系统内核, 由于无法大批量复制部署而具有极高的研发成本.

为此, 本工作提出操作系统内核的积木式设计方法 BrickOS, 以增强面向异构平台的可扩展能力. 具体来说, BrickOS 具有以下特点: (1) 组件积木化. 以操作系统内核的基础功能为粒度, 提出解耦化的内核组件积木式抽象, 可以自由定制调度模块、内存管理模块、文件系统模块等内核的组件, 根据目标应用场景进行选择. (2) 内核架构可配置. 在组件积木化的基础上, 提出内核架构可配置方法, 即根据应用场景需求选择内核架构 (包括宏内核、微内核、简要内核), 避免传统单一架构的局限. (3) 积木灵活组合. 内核组件既可以加载到内核态运行, 又可以加载到用户态运行. 在组件间隔离方面, 前者可通过单地址空间隔离机制实现, 后者通过独立进程地址空间实现. 支持根据用户提供的配置文件自动生成符合场景需求的内核.

本文的主要贡献包括: (1) 面向泛在计算领域异构硬件提出可敏捷定制领域操作系统内核的积木式设计方法; (2) 实现并测试 BrickOS 的系统原型, 证明其可行性.

2 背景与动机

2.1 主流操作系统内核架构

经典而主流的操作系统内核架构主要可以分为: 宏内核架构、微内核架构、简要内核架构、外核

1) Wind River. VxWorks. https://www.windriver.com/products/vxworks.
2) NuttX. https://www.osrtos.com/rtos/nuttx.
3) Richard B. FreeRTOS. https://www.freertos.org/zh-cn-cmn-s/index.html.

架构.

常见的操作系统如 Linux, FreeBSD 等都是采用了宏内核架构, 整个操作系统都运行在内核态且属于同一个地址空间. 操作系统中所有的模块组件, 如文件系统、网络栈、硬件驱动、进程管理等, 都运行在具有特权的内核态. Linux 作为应用范围最为广泛的操作系统之一, 经过多年的演化与发展, 在 2020 年已经拥有超过 2700 万行代码 [4], 其最为显著的优势在于拥有良好的南向与北向生态, 被广泛部署于云平台服务器、移动平台终端、个人计算机等. 不过, 宏内核操作系统架构缺乏系统隔离, 在安全和可靠方面存在不足, 例如每年 Linux 都会被发现不少安全漏洞. 微内核操作系统把文件系统、网络协议栈、硬件驱动等大部分操作系统服务模块运行在用户态的独立进程中, 仅在内核态保留必要机制用于支撑上层系统服务的运行, 提供不同系统服务之间的通信能力.

微内核的架构优势在于拥有良好的容错性和安全性. 因为它保证不同系统服务之间的隔离, 即使某个系统服务出现故障或受到安全攻击, 也不会直接导致整个操作系统崩溃或被攻破. 黑莓公司的 QNX 操作系统 [3] 是成功的微内核操作系统代表, 其广泛部署于汽车等高可靠领域. 学术界成功的 L4 微内核操作系统 [5] 也衍生出世界上第 1 个经过形式化验证的微内核 seL4 [6], 应用于德国政府安全手机的操作系统等. 不过, 微内核架构由于较强的隔离性而在时延与性能方面存在不足之处. 近年来, 国内外工业界对微内核操作系统的研发愈发积极, 例如谷歌 (Google) 公司发布的 Fuchsia 操作系统[4)]、华为公司发布的鸿蒙操作系统[5)] (面向车载领域鸿蒙 VOS 是华为自研的微内核操作系统) 等.

简要内核架构将应用程序与操作系统运行在相同特权级别和地址空间中, 主要面向不支持地址空间隔离和特权隔离等功能的嵌入式计算平台. 该架构操作系统的典型代表是 MS-DOS (microsoft disk operating system) [7]. 这种架构的一个优势在于应用程序和操作系统之间的交互简单且快速, 因为没有权限级别和地址空间的隔离; 但也正因如此, 应用程序中的错误可能直接导致整个系统崩溃. 除了 MS-DOS 外, 采用该架构的操作系统还包括 FreeRTOS 等, 它们主要运行在微控制单元等相对比较简单的硬件设备上.

外核 (exokernel) [8] 又被称为极限内核, 其设计理念是操作系统内核不应该对硬件资源进行过度地抽象, 因为过于通用的抽象难以有效契合于各种应用程序需求. 该架构旨在使应用程序拥有选择硬件资源抽象的能力, 因为应用程序知道什么样的硬件抽象对自身的运行是最佳选择. 该架构的实现方式是通过将硬件抽象封装到与应用程序直接链接的库操作系统 (LibOS) [9, 10] 中, 而库操作系统是应用程序开发人员自主配置或者是自行开发的. 在运行时, 库操作系统和应用程序运行在同一个地址空间中且运行在用户态, 外核不提供具体抽象而仅负责保证多个应用程序 (多个库操作系统) 能够安全且高效地共享硬件资源. 外核架构的应用范围局限性较多, 目前主要是在嵌入式场景和轻量级虚拟机中有一些应用. 面向较复杂的场景, 库操作系统将会变得非常复杂, 甚至相当于一个完整的宏内核, 从而丧失外核架构本身的优势. 目前, 在工业互联网和智能物联网中, 不同内核架构的操作系统均有部署, 比如基于宏内核架构的 Windows, Linux, VxWorks, MindSphere[6)], Android Things [11], SylixOS[7)]; 基于微内核架构的 QNX [3], Zephyr[8)], RT-Thread[9)]; 基于简要内核架构的 FreeRTOS、华为 LiteOS-M[10)], 不过它们尚缺乏对于异构计算平台的灵活适应能力.

面向异构硬件平台, 操作系统学术界研究者还提出过多内核架构 (包括 fos [12], Akaros [13],

4) Google. Fuchsia. https://fuchsia.dev.
5) Huawei. HarmonyOS. https://www.harmonyos.com/cn/home.
6) Siemens. MindSphere. https://new.siemens.com/cn/zh/products/software/mindsphere.html.
7) ACOINFO. SylixOS. https://www.sylixos.com.
8) The Linux Foundation. The Zephyr Project. https://zephyrproject.org.
9) RT-Thread. RT-Thread. https://www.rt-thread.org.
10) Huawei. Introduction to Huawei LiteOS. https://gitee.com/LiteOS/LiteOS/blob/master/README_EN.md.

Tessel [14], Hive [15], Barrelfish [16]). 基本思想是把 CPU 处理器划分成不同的单元, 在每个单元上部署一个操作系统内核, 以此获得降低内核中维护一致性性能开销的优势. Solros [17] 面向异构处理器提出了以数据为中心的操作系统设计, 从而取代传统以 CPU 为中心的操作系统设计. L4 微内核研究者面向新兴硬件提出 M3 [18] 微内核操作系统, 致力于从操作系统设计出发指导异构计算平台的演进与设计, 从而使得操作系统能够获得时延和隔离等方面的优势. Popcorn-Linux [19] 基于 Linux 操作系统进一步研发, 致力于向上层应用程序屏蔽体系结构之间的异构性, 从而支持灵活迁移等特性. Helios [20] 操作系统引入了 "卫星内核" 架构, 它在具有不同架构和性能特征的处理器之间提供一组统一的操作系统抽象. 通过远程消息传递, 使文件系统等 I/O 服务的访问透明化, 将标准微内核消息传递抽象扩展到卫星内核. 普渡大学 (Purdue University) 提出的 splitkernel [21] 内核架构针对未来云平台中的 "分布式计算机", 计算、内存、存储等硬件都将通过高速网络互联, splitkernel 的设计是在每个智能硬件上运行内核模块和通信组件, 进行分布式协同与管理. UnderBridge [22] 提出了基于新型体系结构特性的微内核架构和宏内核架构之间的低时延动态切换技术. 这些相关工作都缺乏面向工业互联网等泛在计算场景中异构硬件资源 (体系结构特征差异大、计算资源尺度差异大) 的设计.

2.2 异构硬件设备

在当下工业互联网的新兴场景中, 硬件设备也呈现出异构化的特点. 不同类型的智能设备对计算资源的需求不同, 因此硬件厂商也开发了不同种类的处理器供其使用. 以下列举了一些主流的处理器类型.

Intel x86 系列处理器泛指一系列英特尔公司推出的复杂指令集架构 (complex instruction set computing, CISC) 的处理器. 该系列处理器支持多种工作模式, 例如在实模式下通过段寄存器与偏移寻址; 在保护模式下通过段描述符实现内存权限的保护, 并可以通过段机制或者页表机制寻址. x86 硬件平台是一个高度标准化的计算机平台, 应用十分广泛. 得益于强大的性能与良好的兼容性, 它被大量用于个人计算机与服务器中. 但是对于大部分嵌入式设备来说, 它们对处理器功能的需求较少, 反而更关注能耗比等参数, 这也是 x86 系列处理器在嵌入式场景中较为少见的原因.

ARM Cortex-A 系列处理器是 ARM 公司开发的精简指令集架构 (reduced instruction set computing, RISC) 的处理器. 它们性能强劲, 主要针对消费娱乐场景, 被广泛应用于智能手机、平板电脑等产品. 例如 Cortex-A78 处理器是一款高性能处理器, 采用了 Austin 微架构, 提供完整的 MMU 支持. 它由于出色的性能表现被作为高端移动端芯片的主要计算组件. 与 Intel x86 系列处理器类似, ARM Cortex-A 系列处理器也较少被应用于嵌入式场景中.

ARM Cortex-M 系列处理器偏重于工业控制, 主要针对成本和功耗敏感的场景. 例如 Cortex-M3 处理器不支持 MMU, 而且使用固定内存映射的方式来访问内存. 它还支持一个可选的内存保护单元 (memory protection unit, MPU) 组件, 可以将地址空间划分为至多 8 个域来控制其读写权限. 由于 M3 处理器仅包含嵌入式场景关注的硬件 (例如嵌套向量中断控制器、定时器、调试访问端口等), 与 Intel x86, ARM Cortex-A 等复杂的全功能的处理器相比, 更适用于对实时性、功耗等要求较高的汽车车身系统、工业控制系统等场景. 最新发布的 Cortex-M55 处理器支持了更多新特性, 例如 4 阶段流水线、Trustzone 可信执行环境等. 但是其仍然不包含 MMU 硬件, 依旧采用固定映射的方式访问内存. M55 同样支持使用 MPU 来保护内存, 域的数目也扩展到最多 16 个.

硬件设备的异构性为上层操作系统内核的开发带来了挑战. 例如当前主流的全功能操作系统 (例如 Linux, seL4) 无法在 Cortex-M 系列上运行, 因为其缺少 MMU 等必要的硬件. 以微内核操作系统为例, 所有的非核心系统组件都被运行在用户态独立的进程中, 各自通过进程提供的隔离机制来避免

互相影响. 但是一些不包含 MMU 硬件的处理器往往使用固定映射的方式管理内存, 缺乏创建出独立地址空间的能力, 此类处理器上往往需要运行特殊的嵌入式操作系统. 这些嵌入式操作系统一方面拥有不同的设计模式, 且仅适配有限种类的硬件; 另一方面定制化程度过高, 开发成本较高, 且难以根据具体需求进一步扩展或裁剪功能.

2.3 泛在操作系统内核的设计需求

传统计算场景中硬件平台通常趋于同构或异构性不突出, 例如传统云计算场景中通常由相同或相似的云平台服务器组成集群, 因此相应操作系统内核的设计与开发往往面向同构计算资源. 然而, 泛在计算场景中计算平台的体系结构特征、硬件资源数量差异性极大, 即便是诸如 Linux 内核此类的通用操作系统内核也难以全部支撑. 针对各类硬件平台进行操作系统内核人工定制需要花费大量的人力和时间成本, 且众多独立内核难以进行后续演进和维护. 因此, 泛在操作系统内核应当具备适应其所面向领域中多种类异构平台的能力. 具体来说, 泛在操作系统内核有以下需求:

一是内核组件可自由定制. 不同场景对内核组件的需求不同, 泛在操作系统内核应当将系统组件解耦, 并可以选择需要的组件将其自由拼接组合以适应当前场景. 特别需要考虑计算平台的异构性, 其设计应当与硬件解耦, 不能过度依赖某些硬件特性而限制其通用性.

二是可以在多架构中自适应切换. 操作系统内核应当可以在启动时根据运行平台上的计算资源情况和用户定义的应用场景需求, 自主选择内核架构, 并启动和部署相应的内核组件.

三是运行的效率较高. 由于在不同组件间引入了明确定义的边界, 其通信时间延长成为了新的挑战. 操作系统内核应当采用低延时通信方法, 从而使得积木式内核能够表现出良好的性能.

四是开发难度较低. 当前缺乏一个统一的内核开发框架, 针对特殊场景开发新的内核十分困难. 泛在操作系统内核应该可以根据用户提供的配置文件快速生成高度定制化的内核.

2.4 相关工作

学术界和工业界对于组件化的操作系统内核设计存在不少研究. Software Dock [23] 支持操作系统组件化, 并使用了一种基于代理的软件模型来协商生产者与消费者. THINK [24] 采用了一种基于组件的软件编程模型, 定义了模块交互的接口, 并且支持自由选择组件装配操作系统. OpenCom [25] 也采用了一种基于组件的方式创建系统软件, 它定义了一个最小的运行时内核, 可以自由定制所需的组件, 其优势在于考虑了兼容不同的硬件部署环境, 由硬件环境提供者提供模块的加载器与链接器, 系统应用的开发者再基于此开发应用程序. Unikraft [26] 是开源的库操作系统 (Library OS), 支持选择内核组件定制化单地址空间的操作系统内核, 不过主要面向云端虚拟机应用, 依赖于运行在主机操作系统 Linux 之上. FlexOS [27] 提出一种操作系统隔离性可配置方法, 支持多种软硬协同的隔离机制, 重点关注内核的安全性与可靠性. 本文提出的 BrickOS 继承了现有工作组件化的思想, 更进一步地设计与实现内核架构可配置方法, 从而能够根据处理器特征、性能与隔离需求等灵活定制不同的操作系统内核, 具有面向软硬件异构性的良好扩展能力.

Pebble [28] 是一种为特定应用程序设计的专用操作系统, 它采用服务器/客户端设计模式, 服务器中包含一个精心裁剪的内核, 包括中断处理、调度、内存管理、虚拟内存等组件, 可以根据域用户态程序的需求进一步精简. Pebble 中不同的组件运行在不同的保护域 (protection domain) 中, 不同的保护域通过页表隔离. eCos [29] 操作系统也提供了相似的功能, 它旨在提供一个标准化的框架供开发者自定义嵌入式内核. eCos 注重提升易用性, 包含了大量可配置选项和 GUI (graphical user interface) 界面自动生成相关头文件. JBEOS [30] 将传统意义上的内核服务抽象为一系列系统构件, 系统构件接口被

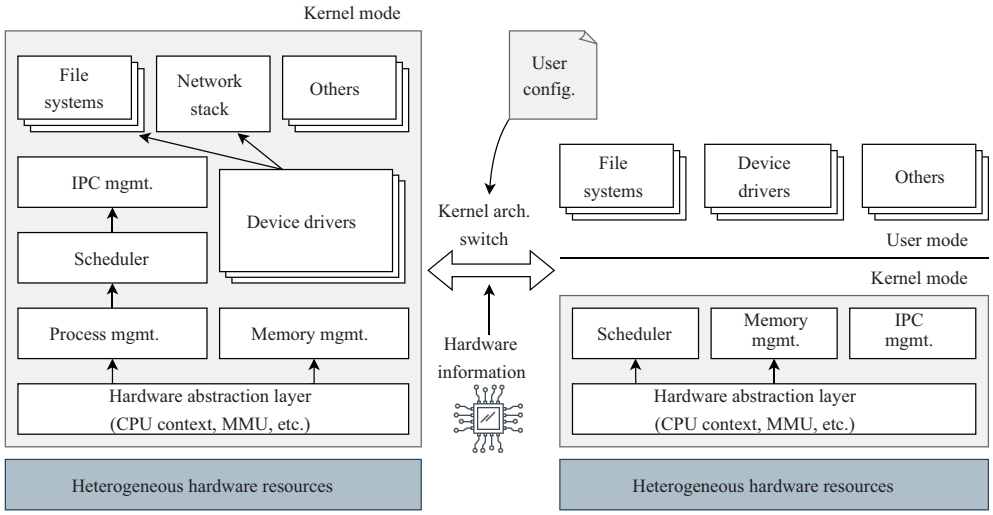

图 1　(网络版彩图) BrickOS 系统架构概览

Figure 1　(Color online) BrickOS architecture overview

良好定义, 可以支持以弹性系统为目标的系统构造, 能够很好地适应嵌入式场景. 这些操作系统虽然能够支持内核的组件化, 但是其使用场景有限, 仅能用于针对特定应用的嵌入式场景中. 本文提出的 BrickOS 则面向泛在计算场景, 支持更加复杂的语义, 进而可以支持更加复杂的程序.

针对泛在计算场景的系统设计也是重要的研究方向. 一些工作研究了在异构硬件场景下高性能计算系统的设计或生成方法 [31~33]. 这些工作主要关注应用程序设计与算法性能优化, 而不是基于异构硬件的操作系统构建及系统安全隔离. 文献 [34] 提出了一种泛在场景下的安全关键软件的建模与生成方式, 其主要关注于软件的异构性而非硬件的异构性. 文献 [35] 总结了泛在计算场景下的安全需求, 而操作系统提供的灵活的安全隔离机制是实现这些安全需求的基础. XiUOS [36] 是一款面向工业物联场景的泛在操作系统, 包括微型实时操作系统内核和其上的智能工业物联框架, 目前尚不具有内核组件积木式灵活配置能力.

3　积木式内核架构 BrickOS 设计

本文提出了一个能够适应多种类异构平台的积木式内核 BrickOS, 其架构图如图 1 所示. BrickOS 中所有的内核组件相互解耦, 其依赖关系也被明确定义, 不同的组件间边界清晰, 不存在跨组件的隐式状态共享, 具有依赖关系的多个组件通过接口进行协同. BrickOS 内核组件可以选择性地加载, 内核框架也可以根据硬件信息与用户配置信息自由切换. 如图 1 所示, BrickOS 可以将所有内核组件运行在特权级 (左图, 类似宏内核架构), 也可以将部分内核组件运行在非特权级 (右图, 类似微内核架构). 该设计通过硬件抽象层屏蔽底层计算能力、资源等异构的平台, 为上层内核组件的设计提供统一的抽象.

本节将从 4 个方面介绍 BrickOS 的设计: 3.1 小节研究内核组件积木化抽象, 其中每个积木组件提供基础的内核机制, 积木组件之间互相解耦且能够拼接; 3.2 小节介绍如何根据硬件平台特征与用户定义相结合的方式选择积木组件和具体内核架构; 3.3 小节介绍如何优化积木组件间通信的时延以提升整个内核系统的性能; 3.4 小节研究如何为异构性硬件提供的内存保护机制提供统一的抽象, 以及如何将其用于提升整个系统的安全性.

表 1 内核提供的基础抽象 (HW-dep: 硬件相关)

Table 1 Fundamental abstractions of kernel (HW-dep: hardware-dependent)

Abstraction	HW-dep.	Description
Thread (process)	No	States of programs running on processors
Context	Yes	Hardware states such as register values
Physical memory	No	Physical memory accessible to applications
Memory isolation	Yes	MMU/MPU devices that isolate memory access rights
File	No	Operating system resources like storage, network, and devices
Communication	No	Interfaces for inter-process/inter-thread communication
Interrupt	Yes	Hardware response to external events

表 2 内核提供的系统服务

Table 2 System services of kernel

System service	Description
Scheduler	Implement scheduling policies based on thread abstraction
Memory mgmt.	Implement physical memory allocator based on physical memory abstraction
Process mgmt.	Implement thread creation/update/deletion based on thread abstraction
File system	Implement file abstraction based on physical memory abstraction
Driver mgmt.	Implement external device interaction based on file abstraction
Network stack	Implement network communication based on file abstraction
IPC mgmt.	Implement inter-process call (IPC) interfaces based on process and communication abstraction
Interrupt handler	Implement interrupt handling based on interrupt abstraction

3.1 内核组件积木化

操作系统内核的关键职能是提供基础抽象和系统服务. 基础抽象指的是内核从计算平台硬件中提取出来的具有共性的特征, 它们屏蔽了底层硬件的差异性, 为上层的系统服务提供统一的接口. 表 1 总结了内核提供的 7 种基础抽象. 抽象间可能存在依赖关系, 例如线程抽象依赖上下文抽象, 由于上下文抽象屏蔽了硬件状态, 线程抽象可以被设计为硬件无关的; 同理, 内存抽象依赖地址翻译抽象, 因此得以向进程提供硬件无关的内存视图. 系统服务指的是内核中实现某一特定功能的模块, 它们一般依赖于内核提供的特定抽象. 表 2 总结了内核提供的部分重要系统服务. 其中一些服务在依赖抽象的同时, 也为内核提供更高层次的抽象, 例如文件系统服务提供文件抽象. 系统服务都是硬件无关的.

内核的 7 种基础抽象基于现有计算机硬件的共性结构设计, 能够满足操作系统对处理器、内存、外设 3 类设备的管理需求. 基于上下文抽象、线程 (进程) 抽象、通信抽象, 操作系统可以实现处理器的多任务分时复用与多任务协同. 基于物理内存抽象和内存隔离抽象, 操作系统可以实现内存资源的分配与内存访问权限隔离. 基于物理内存抽象 (设备内存)、中断抽象、文件抽象, 操作系统可以实现对网卡、磁盘、加速器等外设的控制, 并将外设暴露给用户使用. 内核提供的系统服务可以根据应用的需求进行增加或删减. 例如在单进程场景下可以删减通信管理服务, 在多用户场景下可以增加用户管理服务.

BrickOS 基于抽象与服务的定义和关系, 以积木化内核组件形式对它们进行具体设计与实现. 具体来说, 每个内核组件负责实现某个具体抽象或某个系统服务; 每个组件对外提供明确的调用接口, 不

同组件之间具有清晰的边界, 且不存在跨组件的隐式状态共享. 具有依赖关系的多个组件通过接口进行协同.

BrickOS 通过实现少数几个硬件相关的基础抽象实现了内核设计与硬件平台解耦. 这部分抽象被设计为图 1 中所示的硬件相关抽象层, 大部分内核组件能够通过该层实现硬件无关. 硬件相关抽象根据体系结构特征分类实现, 硬件无关抽象和服务能够被面向具体领域的不同泛在操作系统内核所共用.

3.2 多架构内核自适应切换

3.2.1 积木式内核组件的插拔与组合方法

内核组件被分为核心组件与非核心组件. 核心组件包括调度服务及其依赖的抽象、内存抽象和通信管理服务及其依赖的抽象, 它们必须被运行在特权级中的系统核心服务上下文 (非进程环境) 中. 其他组件为非核心组件, 它们以进程的形式独立运行. 核心组件为非核心组件的运行提供了运行环境: 调度服务决定了非核心组件进程运行的时机; 内存抽象为每个非核心组件提供了互不影响的内存地址空间; 通信管理服务则是非核心组件间通信的桥梁.

不同的内核组件间边界清晰, 并且通过明确的接口进行协同工作, 这为内核组件的自由化插拔与组合提供了可能. 用户可以根据需求在配置文件中选择系统组件, 由 BrickOS 负责将其加载到内核中. 为了解决组件依赖的问题, BrickOS 提出通过依赖图的方式追踪组件间的依赖关系. 每一个组件都可以向外暴露可供调用的接口, 例如当组件 A 调用了组件 B 提供的接口时, 则在依赖图中加入一条由 B 指向 A 的边, 每一个内核组件能够加入内核的前提是其入边对应的组件都已经被加入到内核中. 具体来说, 每一个可动态加载的内核组件都维护了一个配置文件, 其中定义了其依赖的其他组件类型, 例如网络栈组件依赖文件系统服务, 如果用户指定的内核组件中没有实现文件抽象的系统服务, 则无法生成对应的定制化内核. 当用户指定的所有内核组件的依赖图间构成一个有向无环图时, BrickOS 会遍历该图, 依次加载所有的内核组件.

为支持相关组件之间的自动化组合, 本工作研制了内核组件胶水代码生成机制, 可以在不需要人工参与的情况自动根据组件间接口和参数标记生成必要的组件间胶水代码, 负责参数和结果传输、控制流传递等流程. 默认情况下, 非核心组件以进程的形式执行, 胶水代码会将组件间的调用转换为进程间通信机制相关的代码. 同时 BrickOS 提供组件间同步与异步通信方式供用户灵活选择, 已取得时延和 CPU 利用率之间的平衡. 为了追求高性能, BrickOS 也支持将非核心组件运行在系统核心服务上下文中, 它们与核心组件共享同一个地址空间, 此时它们之间的调用胶水代码可以被优化为直接函数调用, 本文将在 3.4 小节讨论该优化方法.

为了支持适配不同的硬件, 在开发硬件相关内核组件时需显式标注针对的硬件平台特性, 通过该信息和组件依赖图, BrickOS 能够自动筛选出构建操作系统内核的备选组件.

3.2.2 内核架构的动态选择

操作系统内核的硬件平台自适应性还需要考虑到物理计算资源的限制. 具体来说, 泛在计算场景中的硬件平台一方面具有不同的体系结构特征, 比如是否支持权限级划分、是否支持虚拟内存等, 另一方面具有不同资源尺度, 比如物理内存容量、CPU 核心数量等.

支持特权级划分的处理器一般将所有的硬件指令划分为特权指令和普通指令, 只有在处理器切换到内核模式时才能执行特权指令. 整个系统由此被划分为特权级 (一般运行内核) 与非特权级 (一般运行用户态应用程序和用户态系统组件) 两部分. 运行在非特权级的程序无法执行影响系统核心功能的指令, 进而保护了系统安全. 通用处理器一般包含内存管理单元 (MMU) 来管理内存. MMU 通过

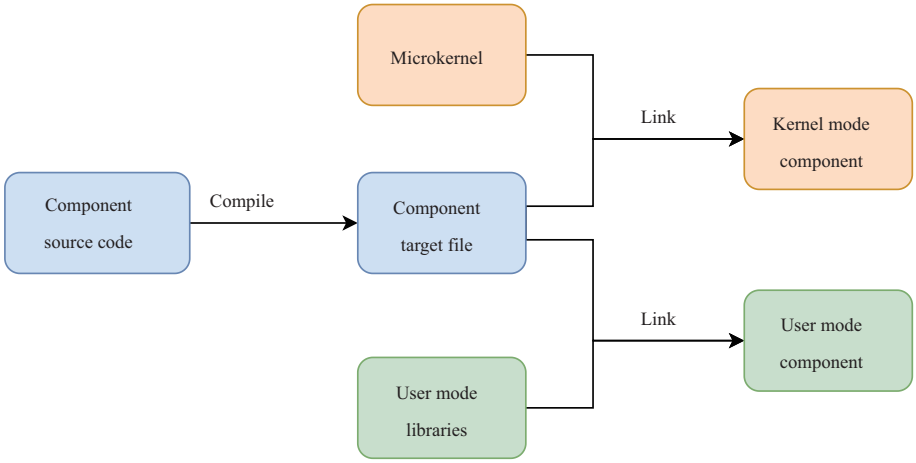

图 2 (网络版彩图) 选择性地加载非核心组件位置

Figure 2 (Color online) Selectively load non-core components to different locations

段或者页表管理的方式为每一个进程维护一个独立的虚拟地址空间, 由 MMU 硬件负责处理虚拟地址空间到真实物理地址空间的映射. 虚拟地址空间一方面为应用程序提供了连续统一的内存抽象, 为程序开发提供了便利; 另一方面实现了地址空间的隔离, 处于不同地址空间的进程无法互相访问内存, 提高了系统的安全性. 此外在支持特权级的处理器中, 内核的代码和数据结构被置于内核地址空间中, 处于用户态的应用程序无法直接访问, 进一步保证了内核的安全. 但是在硬件异构化的趋势下, 有的嵌入式处理器不提供特权指令划分或 MMU 硬件带来的虚拟内存抽象, 而是将所有组件运行在同一个权限级且采用固定映射的方式访问物理内存. 在此类嵌入式设备中, 无法运行通用的依赖虚拟内存抽象的操作系统内核 (例如 Linux, seL4), 而只能运行简要内核架构, 即将所有的内核组件运行在相同的地址空间中, 并固定分配每个组件使用的内存范围.

此外, 对于不同资源尺度的硬件, 结合不同的场景需求, 也存在不同的内核架构需求. BrickOS 可以灵活选择宏内核架构、微内核架构、简要内核架构等, 并且把积木式内核组件自动按照所选架构进行融合搭建. 内核架构的动态选择主要是通过切换非核心组件的加载方式实现的.

非核心组件的加载方式分为内核态和用户态两种. 如果被加载到内核态, 就可以直接与内核中的其他组件共享地址空间, 进而支持高效的组件间通信. 如果被加载到用户态, 该组件则运行在独立的地址空间中, 可以保证较强的安全性与扩展性. 为了减轻组件开发者的负担, BrickOS 采用了图 2 所示的设计, 非核心组件只需维护一套源代码, BrickOS 可以自动完成内核态与用户态两种运行方式的切换. 如果以内核态组件方式运行, 在链接时该组件对内核的所有函数调用都会被重定位为内核暴露的相应的函数地址; 同时该组件也会暴露一些功能接口, 供内核中的其他模块以及用户态进程进行符号重定位. 如果以用户态组件方式运行, 对微内核函数的调用会在链接时被重定位到 IPC 模块, 再由 IPC 模块转发到对应的组件处理. 在具体的设计实现中, BrickOS 提供了用户态库, 它与内核态 IPC 模块交互, 向用户态组件提供相同的接口.

当用户通过配置文件指定生成宏内核架构时, BrickOS 会将所有的组件都加载到内核态中, 并启用虚拟地址抽象, 此后执行的应用程序则被运行在用户态地址空间中; 当生成简要内核架构时, BrickOS 同时将所有的组件和用户态进程都运行在内核态中, 且关闭虚拟地址抽象, 此时整个内核为一个独立的程序; 当生成微内核架构时, BrickOS 仅将核心组件加载到内核态, 将非核心组件和用户态进程都加

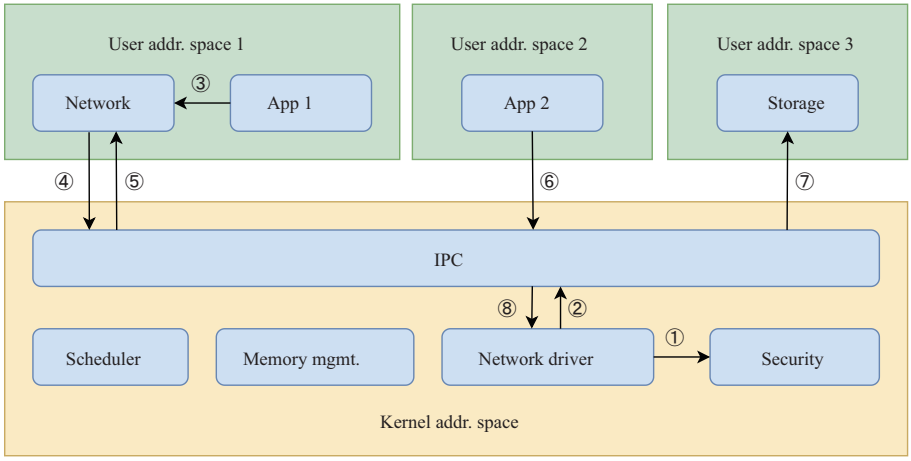

图 3　(网络版彩图) BrickOS 进程间通信示例

Figure 3　(Color online) BrickOS IPC examples

载到用户态中, 且每个非核心组件都处于独立的地址空间中; 此外, BrickOS 灵活的设计允许开发者自定义部分组件运行在内核态以提升性能, 而将一些不可信的组件运行在用户态以保证安全, 这种设计可以最大限度地适应应用场景的需求, 达到性能与安全性的平衡.

为了便于开发, 本文基于以上设计开发了内核自动构建工具, 可以根据开发者提供的配置文件自动编排非核心组件的加载方式, 并对用户透明地支持不同组件间的通信 (在 3.3 小节中讨论); 同时开发者也可以参与或辅助内核构建, 从而实现更加精细化的内核组件选择与融合.

3.3　组件间高效消息传递

传统的微内核架构将每个系统组件和用户态应用程序都运行在一个单独进程中, 每个进程拥有独立的地址空间, 这样可以提供很好的隔离性, 一个组件出错不会将错误扩展到系统的其他部分. 在这种架构中, 所有的进程依赖内核中的 IPC 组件进行消息传递, 且自身也实现了一个 IPC 服务器以监听其他组件的请求. 频繁进出内核导致 IPC 开销过大是造成微内核架构性能不佳的主要原因, 这也吸引了许多研究者提出优化方案 [6,37~41]. 学术界较为流行的微内核 seL4 [6] 提出了一系列优化措施, 例如使用寄存器传参、避免调度, 可以将一次 IPC 的开销降低到 1500 个时钟周期; SkyBridge [41] 利用虚拟化技术优化 IPC, 可以进一步将一次 IPC 的开销降低到 400 个时钟周期. 但是其开销仍与宏内核架构中的函数调用开销 (约为 20 个时钟周期) 有较大差距.

为了解决这一问题, BrickOS 允许在一个地址空间中运行多个系统组件, 在相同地址空间中的组件的通信无需经过内核中的 IPC 组件, 而是直接通过函数调用的方式完成. BrickOS 支持灵活的组件编排, 可以将非核心系统模块运行在内核态或者用户态. 运行在内核态的系统组件与内核中的其他组件共享地址空间, 由于内核中的其他模块可以直接通过函数调用来访问该组件, 因此该组件无需监听来自其他进程的 IPC 请求. 用户态的系统组件运行在不同进程的地址空间中, 它们可以被分为两类: 第 1 种是私有系统组件, 如图 3 中的网络模块, 它与用户态的应用程序运行在相同的地址空间中, 只负责为相同地址空间中的应用程序提供服务. 每一个用户态地址空间可以有一个或多个私有系统组件, 私有系统组件无需监听 IPC 请求, 同一地址空间的进程可以直接通过函数调用访问私有组件. 第 2 种是公有系统组件, 如图 3 中的储存模块, 它是独立存在于一个用户态地址空间的组件, 它负责为所有其他组件提供服务. 此类组件与内核中的 IPC 模块通信, 且需要监听来自其他组件的访问请求.

图 3 展示了一个应用场景, 网络驱动模块和安全模块被加载到内核中, 其他模块被作为私有模块或者公有模块加载到用户态地址空间中. 如果出现跨地址空间的通信, 则需要内核地址空间中的 IPC 模块的参与. IPC 模块时刻监听着来自不同进程的 IPC 请求, 并将其转发给目标模块执行. 具体来说, 会出现以下 4 种跨进程通信形式.

• **单地址空间内通信.** ① 和 ③ 分别展示了内核地址空间和用户态地址空间中通信的例子. 由于调用者和被调用者处于相同的地址空间中, 调用者可以直接通过函数调用的方式访问被调用者提供的接口. 为了支持这一特性, BrickOS 在链接系统组件的时候会重定位所有跨模块调用函数的地址, 如果待调用模块与调用者处于同一个地址空间中, 对应的函数地址会被重写为正确的地址.

• **用户态模块调用内核态模块.** 当用户态模块需要调用内核态模块时 (例如图中的网络模块调用内核地址空间中的网络驱动模块), 首先会向内核中的 IPC 模块发送请求 (④), IPC 模块中的监听函数收到请求后直接通过函数调用跳转到内核驱动模块中执行 (⑧), 执行完毕后返回到 IPC 模块 (②), 最后将结果返回给用户态模块 (⑤).

• **内核态模块调用用户态公有模块.** 内核态模块可以直接调用 IPC 模块的接口向目标公有模块发起 IPC 请求, 待调用模块监听到调用请求后执行相关函数, 再将执行结果返回给 IPC 模块.

• **用户态模块调用用户态公有模块.** 当用户态模块需要调用用户态公有模块时 (例如图中的应用程序 2 调用储存模块), 首先会向内核中的 IPC 模块发送请求 (⑥), IPC 模块将其转发给对应模块 (⑦), 执行完毕后再将结果传回.

此外, 针对一些嵌入式场景, 为了追求最优性能, BrickOS 支持将应用程序也运行在内核地址空间. 在这种情况下, 所有组件 (包括应用程序) 都运行在一个地址空间中, 所有的模块间通信均为函数调用, 整个系统没有 IPC 模块的参与.

3.4 异构内存保护模块抽象

BrickOS 支持将多个系统组件和应用程序运行在一个地址空间中, 与微内核中每个组件运行在独立的进程中相比, 这种做法缺乏隔离性, 一个系统组件可以随意修改其他组件的内存. 攻击者可以利用系统组件的漏洞窃取或篡改当前地址空间中其他组件的内存. 为了解决内存隔离的问题, 当前主流的硬件实现了不同的内存域机制, 它们的区别总结如下.

Intel memory protection keys (MPK) [42] 利用页表项中的 4 个未被使用的位来存储该项指向的页所属的内存域的标识符 (protection key). 它最多为每个地址空间支持 16 个内存域. 每个内存域对应的读写权限由用户态可以访问的寄存器 PKRU (user page key register) 控制. 由于 MPK 机制本身不提供安全机制, 即用户可以自由修改 PKRU 寄存器以达到放弃其他内存域的目的, 因此需要结合 CFI (control flow integrity) 和二进制扫描等技术来保证安全性. MPK 目前的实现不支持保护内核页, 但是现有的工作表明, 无论运行在用户态还是内核态, MPK 都会对用户可以访问的内存页做权限检查. 因此可以将内核页对应的四级页表中的 User/Kernel 位均设置为 1 (即 User 位), 以允许 PKU 在内核态保护内核页.

ARM memory domains[11] 是在 ARMV8 和 AArch32 中引入的内存域管理机制, 但是在 AArch64 中被废弃了. 该机制在第 1 级页表的页表项中使用 4 位来储存内存域标识符, 因此也是为每个地址空间支持 16 个内存域, 由于标识符仅存在于第 1 级页表中, 内存域的边界被限制为 1 MB 个块对齐. 每个内存域对应的读写权限由一个内核态的寄存器 DACR (domain access control register) 控制. DACR

11) ARM. ARM developer suite developer guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html.

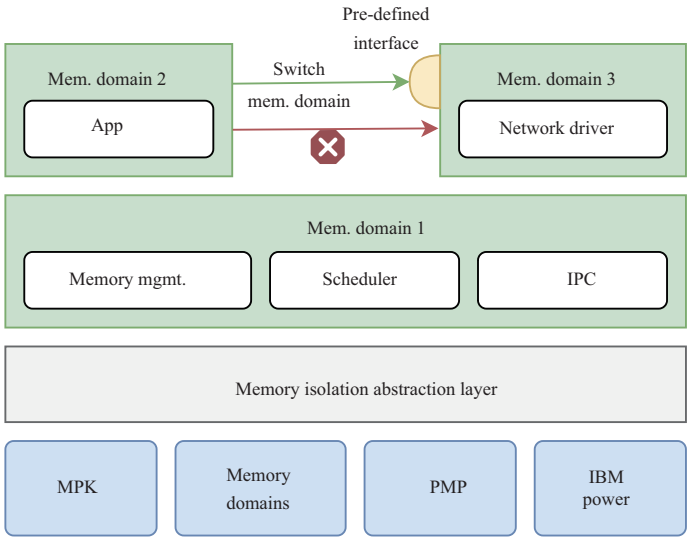

图 4　(网络版彩图) BrickOS 支持异构内存保护示例

Figure 4　(Color online) Schematic of BrickOS support for heterogeneous memory protection

为每个内存域配置了两个位, 共支持阻止访问、采用 PTE 设置的权限与允许访问 3 种内存域访问模式.

RISC-V physical memory protection (PMP)[12] 利用一系列 CSR (control status registers) 寄存器来控制特定物理内存区域的访问权限. 每个处理器核心支持 16 对 PMP 寄存器, 每一对包括一个配置寄存器 (pmpcfg) 和一个地址寄存器 (pmpaddr), 它们仅能在硬件特权级被访问. 配置寄存器和地址寄存器共同定义了每个内存域的范围和权限. PMP 采用白名单机制, 即不在 PMP 内存域中的地址均被视为不可访问. PMP 支持最小 4 字节的内存域.

IBM power[13] 架构采用 5 位的内存域标识符, 支持 32 个内存域. 该机制使用特权级寄存器 (AMR 和 IAMR) 控制每个内存域的读、写和可执行权限.

这些硬件机制提供的内存保护功能都可以被抽象为统一的接口: 配置内存域和切换内存域. 当新加载一个模块时, 需要将其内存空间置于一个内存域中, 这一般是通过修改页表 (如 MPK, memory domains, IBM power) 或者寄存器 (如 PMP) 来完成. 一个进程能够访问的内存域权限由特殊的寄存器决定 (例如 MPK 的 PKRU 寄存器、memory domains 的 DACR 寄存器), 因此可以在系统运行过程中通过修改寄存器的值在不同的内存域之间切换.

图 4 展示了 BrickOS 支持异构硬件保护单地址空间内存的一个例子. BrickOS 为上层系统服务和应用提供了一个硬件无关层, 为各种硬件隔离机制提供统一的接口. 在这个例子中, 网络驱动模块和应用程序都运行在内核态, 但是它们是两个进程, 分别运行在一个内存域中. 内核核心模块则处于一个单独的内存域中. 每个内存域都是一个最小内存访问权限的可执行环境, 其中加载了目标模块的代码段与数据段, 并且拥有独立的堆空间、栈空间. 每个内存域中的进程仅拥有访问当前内存域中内存的权限, 当其试图直接访问其他内存域中的内存时 (例如图 4 中的红线), 该访问会被系统阻止. BrickOS 为内存域中的每个对外提供功能的函数生成一个跳板函数, 其他内存域的进程只能通过这些预定义的调用接口进行跨内存域函数调用. 在跳板函数中, 首先切换到目标栈上, 接着切换内存域的权限到目

12) RISC-V. Risc-v isa specification. https://riscv.org/specifications.
13) IBM Corporation. Power ISA version 3.0b. 2017.

标内存域, 再进入目标内存域执行, 执行完毕后再换回内存域的权限和栈. 为了支持处于不同内存域的模块共享数据, BrickOS 引入了共享内存域的概念, 一个共享内存域可以被两个或多个模块或应用程序同时访问. 例如一个模块需要将一个数据对象作为参数传递给另一个模块, 为了使得目标模块拥有访问该数据对象的权限, 可以将这个数据对象置于共享内存域中.

需要注意的是, 这套机制仅限于单地址空间内的进程间隔离, 处于不同地址空间中的进程相互通信仍然需要 IPC 模块的参与. 供 IPC 机制使用的内存被置于一个特殊的共享隔离域中, 所有进程都具有访问这个域的权限. 此外, 该设计受到硬件的限制, 会导致内存域的数目有限 (例如 MPK 仅支持将一个地址空间划分为 16 个内存域), 此时可以通过划分更多的地址空间来缓解该问题.

4 分析与测试

本节的分析与测试主要分为 3 部分, 第 1 部分通过案例分析 BrickOS 如何为不同场景生成符合需求的定制化内核 (4.1 小节), 第 2 部分通过实验说明 BrickOS 内核的性能 (4.2 小节), 第 3 部分分析 BrickOS 的有效性与局限性 (4.3 小节).

4.1 应用场景分析

为了说明本文提出的积木式内核框架可以适应不同的场景, 本小节分析了两个典型场景案例: 首先以生活场景中的智慧家庭为例, 介绍如何利用 BrickOS 为智慧家庭的控制中枢和边缘智能设备生成符合各自需求的内核. 接着以工业化场景中的无人搬运车 (AGV) 为例, 介绍如何利用 BrickOS 设计实现其无人驾驶操作系统内核.

4.1.1 场景一: 智慧家庭

场景需求. 在智慧家庭场景中, 家庭中的智能家居可以通过互联网接收用户指令, 并可以完成预定的自动化操作. 智慧家庭中往往存在一个中心节点 (控制中枢) 和多个边缘节点 (智能设备). 控制中枢负责根据用户指令和来自各个智能设备采集的信息执行自动化操作. 常见的智能设备有智能灯具、智能门锁、智能窗帘、扫地机器人以及各类传感器, 它们往往相互独立, 只与智慧家庭的控制中枢传递信息; 而且它们使用场景明确, 例如智能灯具中仅运行的嵌入式操作系统, 负责将来自蓝牙或者无线网络的信息转化为控制灯具运行状态的指令.

内核组件分析. 智慧家庭的中心节点作为整个智慧家庭场景的核心, 需要较强的计算能力, 而且为了避免设备被攻击, 需要保证较强的安全性. 因此一般选用微内核架构, 将非核心组件运行在用户态中的独立进程中来保障强隔离性. 而智慧家庭边缘节点的使用场景往往非常明确, 依赖的内核模块较少, 因此整个内核可以被设计得较为精简. 具体来说, 第一, 所有的内核组件和应用程序可以被运行在同一个地址空间中, 以提高程序运行的效率. 第二, 内核无需支持多应用程序, 也无需实现复杂的调度算法. 第三, 硬件环境可能比较精简, 可能不支持 MMU 等硬件, 因此内核的设计需要考虑硬件兼容性. 第四, 由于整个操作系统内核和程序一般以固件的形式被烧录入硬件中, 用户无法通过加载不可信的应用程序以攻击内核, 因此内核中无需包含安全相关的模块.

在该场景下, 若使用传统操作系统, 需要分别为中心节点和边缘节点维护操作系统代码, 在中心节点选用微内核操作系统, 在边缘节点选用嵌入式操作系统. 虽然中心节点与边缘节点的操作系统架构不同, 但部分的系统组件 (如网络栈、驱动、内存管理) 可以共用. 两套操作系统会导致代码分化, 同一特性需要适配两次, 增加系统的维护成本. 使用 BrickOS, 开发者可以便捷地基于一套代码分别生成

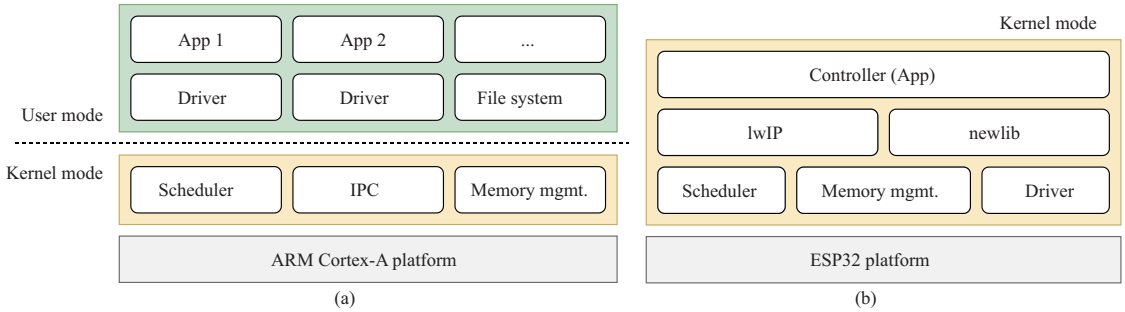

图 5　(网络版彩图) 智慧家庭场景下 BrickOS 内核架构. (a) 中心节点; (b) 边缘节点

Figure 5　(Color online) BrickOS kernel architecture for smart home. (a) Central node; (b) edge node

适配中心节点与边缘节点的操作系统, 降低了开发负担.

内核配置. 在该场景下 BrickOS 内核架构图如图 5 所示. 如图 5(a) 所示, 中心节点运行在 ARM Cortex-A 硬件平台中, 拥有较强的算力. 核心组件运行在特权级中, 非核心组件则以进程的形式运行在非特权级中, 它们彼此隔离, 保证了整个系统的安全性. 如图 5(b) 所示, 边缘节点运行在 ESP32 硬件平台上, 该平台拥有 Wi-Fi、蓝牙模块以及双核的 Xtensa 32 位的 LX6 处理器. 该处理器不包含 MMU 硬件, 仅支持固定内存映射. 通过 BrickOS 组装得到的定制化内核中包含内存管理模块、时钟模块、蓝牙驱动模块、Wi-Fi 驱动模块、lwIP 模块以及主控模块. 内核启动后即开始执行主控模块中定义的任务, 例如监听来自蓝牙或者网络的请求、定时执行相关任务等. 其内核架构类似于简要内核设计, 但是配置更加灵活.

4.1.2　场景二: AGV 小车无人驾驶操作系统

场景需求. 随着工业信息化的逐渐普及, AGV 小车的使用也越来越广泛, 例如在工厂的各个部门间运送托盘, 在集装箱码头运送货品. AGV 小车的核心功能为无人驾驶, 传统的基于单片机的简单控制系统已经无法满足其需求, 需要一个操作系统负责感知、规划、决策等相关的功能. 与传统的操作系统相比, 无人驾驶操作系统有两个核心需求: 一是实时性强, 当接收到外界数据时, 需要能够快速响应处理, 并且能在确定时间内处理完毕. 如果不能快速响应外界事件, 可能会导致严重的安全问题. 二是安全性高, 内核应当具有良好的模块隔离性, 针对每个模块的故障应当有相应的应对方案, 且不应导致整个内核故障.

内核组件分析. 无人驾驶操作系统内核组件的设计应当遵循以下几个原则: 一是驱动程序、协议栈、文件系统等组件应当运行在用户态, 以避免组件间的相互影响. 二是部分重要组件可以被运行在内核中, 以降低进程间通信的开销, 但是这些组件间应当相互隔离, 以避免影响内核的核心功能.

在该场景下, 若使用传统操作系统, 无法灵活地选择内核组件部署在用户态还是内核态, 也缺少内核态组件间隔离能力. 开发者只能选择安全性较弱的宏内核或性能较弱的微内核, 难以做到性能与安全性的平衡. BrickOS 支持开发者根据实际的安全性与性能评估结果灵活地配置内核组件的部署位置, 此外还可以提供内核态组件间内存隔离.

内核配置. 图 6 展示了 BrickOS 定制的无人驾驶操作系统内核的架构图. 整个系统属于混合内核架构, 一方面借鉴了微内核的设计思想, 将大部分内核组件运行在用户态, 以提升整个系统的安全性; 另一方面借鉴了宏内核/简要内核的设计思想, 允许将部分内核组件运行在内核中, 且和整个内核共享一个地址空间, 显著提升了系统性能. 允许在内核中的系统组件通过内存管理模块提供的隔离机

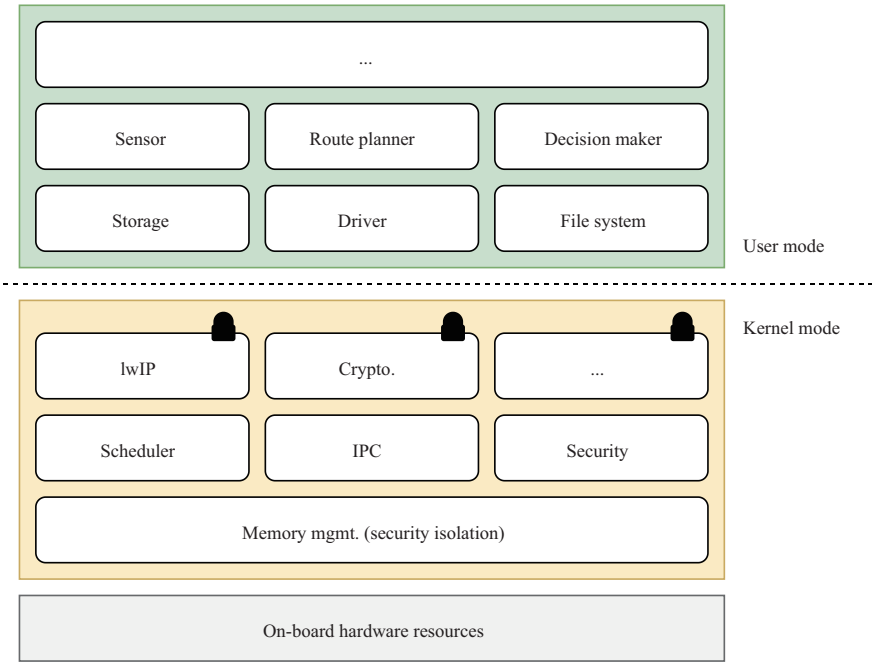

图 6 (网络版彩图) AGV 无人驾驶场景下 BrickOS 内核架构

Figure 6 (Color online) BrickOS kernel architecture for AGV smart driving

制相互隔离, 进一步保证整个系统的安全性. 作为一个可以高度定制的操作系统内核, 图 6 展示的架构也不是唯一可行的方案, 例如用户可以选择将经过严格测试的可信系统组件从用户态移动到内核态执行, 进而提升整个系统的性能.

此外, 在具有特定内存隔离特性的平台上, 还可以完全采用宏内核架构, 提升性能的同时获得与微内核接近的安全性. 具体来说, BrickOS 可以采用 UnderBridge [22] 和 EPK [43] 技术实现内核态组件隔离, 安全地将所有内核组件运行在内核态. UnderBridge 技术基于 Intel MPK 硬件构建内核态隔离域, 通过隔离域切换门禁、特权指令监控等技术对隔离域内的组件进行限制, 达到与用户态内核组件等同的隔离效果. 为了解决宏内核内核态组件数量多, 超过 Intel MPK 硬件隔离域数量的问题, UnderBridge 可以支持将内核组件动态地在用户态和内核态之间迁移. EPK 技术通过结合 Intel MPK 和 VMFunc 技术, 突破了上述隔离域数量限制, 可以在内核态创建 7680 个隔离域, 可以支持所有内核组件同时在内核态隔离运行.

图 7(a) 展示了 BrickOS 与 UnderBridge 结合后应用性能测试结果. 该测试在 BrickOS 上部署了 SQLite 3 数据库应用并运行了 YCSB-A 测试用例. SQLite 应用依赖于文件系统组件 xv6fs, xv6fs 组件依赖于磁盘驱动组件 RAMDisk. 测试分别在 xv6fs 开启与关闭页缓存两种情况下进行. 相比于当前性能最优的微内核操作系统 seL4 (图中的 microkernel), BrickOS-UnderBridge 在开启页缓存情况下可以带来 35% 性能提升, 在关闭页缓存情况下可以带来 105% 性能提升, 性能与宏内核 (图中的 monolithic kernel) 接近.

我们在开启页缓存的情况下对上述测试结果进行了细分分析, 如图 7(b) 所示. 现有的微内核操作系统 seL4, Zircon 和 Fiasco 至少在 IPC 上花费了 30% 的时间, 而 BrickOS-UnderBridge 在 IPC 上花费了 11% 的时间.

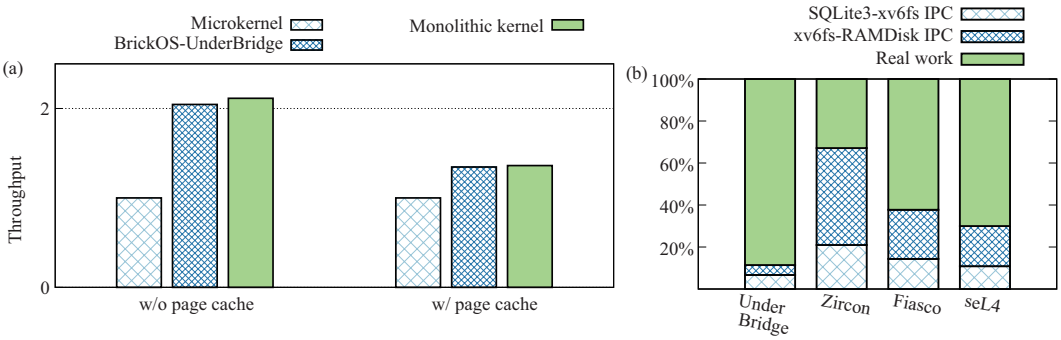

图 7 (网络版彩图) BrickOS-UnderBrige 的 SQLite 3 YCSB-A 测试. (a) 测试结果; (b) 细分分析

Figure 7 (Color online) SQLite 3 YCSB-A evaluation on BrickOS-UnderBrige. (a) Evaluation results; (b) breakdown analysis

表 3 不同运行场景下 IPC 时间开销

Table 3 IPC cost in various scenarios

Communication scenario	Time cost (cycles)
ChCore system call	107
ChCore IPC	1158
BrickOS umod → umod	1242
BrickOS umod → kmod	1208
BrickOS kmod → umod	1215
BrickOS kmod → kmod	14

4.2 性能分析

我们基于微内核 ChCore [22] 实现了 BrickOS 内核框架原型, 支持自由灵活的组件编排和隔离机制选择.

ChCore 微内核提供了 3.1 小节所述的内核抽象与系统服务的实现, 其中进程管理、文件系统、网络栈、驱动 4 类服务运行在用户态, 其余服务或抽象运行在内核态. BrickOS 在 ChCore 的基础上支持用户态组件在内核态隔离执行, 提供了新的构建系统帮助开发者编排组件部署位置, 根据编排结构自动生成组件间 IPC 胶水代码. 本实验以没有修改过的 ChCore 内核为测试基准, 通过 BrickOS 框架将系统服务运行在内核态, 利用新设计的 IPC 机制以提升整个内核系统的性能.

4.2.1 IPC 性能微基准测试

首先, 通过微基准测试展示 BrickOS 定制化内核在不同场景下的 IPC 性能. 将两个系统服务分别运行在不同场景中, 并测试其通信的时间开销. 系统服务暴露一个函数接口供外部调用, 该函数为空函数, 被调用后立即返回. 以下测试中, 我们均测试了 1000000 次, 并取调用开销的平均值. 测试平台为树莓派 3B+, 其中包含了一个主频率为 1.4 GHz 的 ARM Cortex A53 64 位处理器和一块容量为 1 GB 的 LPDDR2 内存.

表 3 展示了 BrickOS IPC 机制的性能. 首先在没有修改的 ChCore 上测试了系统调用和 IPC 的时间开销作为测试基准. 接着分别将两个系统服务分别加载到用户态和内核态, 表中的 umod (user mode) 表示加载到用户态, kmod (kernel mode) 表示加载到内核态. 当模块被加载到用户态时, 无论是

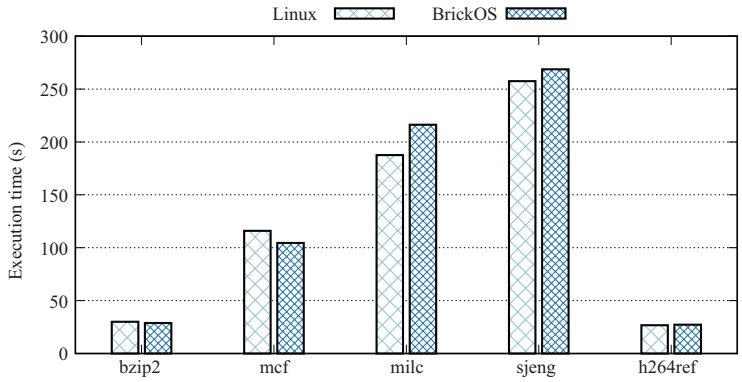

图 8 (网络版彩图) SPEC CPU 2006 测试集执行时间

Figure 8 (Color online) Execution time of SPEC CPU 2006 benchmarks

该模块调用其他模块, 或者说该模块被其他模块调用, 都需要内核中的 IPC 模块的参与, 因此其时间开销与 ChCore 中 IPC 类似. 而如果两个模块都加载到内核态中, 其时间开销可以被显著缩短 (仅需 14 个时钟周期), 这是因为模块间共享地址空间, 两个模块被链接为一个主程序, 其中的调用地址可以在链接时直接被重定位到真实地址, 因此调用时无需上下文切换, 其开销接近一般函数调用的开销.

4.2.2 BrickOS 微内核架构性能测试

由于微内核中不同系统服务间相互隔离, 其在时延与性能方面存在天然不足. 为了说明 BrickOS 架构在最差情况下的性能表现, 利用 BrickOS 搭建出微内核架构的内核, 将 tmpfs 运行在用户态进程中. 以下测试均在 Intel i7-12700 处理器上进行, 并使用了容量为 4 GB、频率为 3200 MHz 的 DDR4 内存.

SPEC CPU 2006 是一种行业标准化的性能测试套件, 我们选取了其中的部分程序来测试 BrickOS 微内核架构的性能表现. 图 8 展示了部分程序分别在 Linux 宏内核 (本文实验中使用的 Linux 版本均为 5.12) 和 BrickOS 微内核上的执行时间. 测试结果表明, BrickOS 最差情况下在 milc 和 sjeng 测试负载上与 Linux 相比分别带来了 15.2% 和 4.3% 的性能开销, 在其他负载上的性能表现与 Linux 持平甚至超过了 Linux. 这是由于 BrickOS 采用积木式框架, 省去了诸多不必要的系统服务, 使得整个系统内核更加精简, 因而可以获得更好的性能.

为了进一步说明 BrickOS 搭建的微内核的性能, 测试了操作系统内核中的部分基本操作, 并将其与 Linux 做对比, 表 4 展示了测试结果. 得益于更加精简的设计, BrickOS 在空的系统调用、上下文切换与进程创建过程中, BrickOS 的时间开销均小于 Linux. 在其他的基础测试中, BrickOS 搭建的微内核的性能要低于 Linux, 它们都涉及了更多的系统服务, BrickOS 需要依赖 IPC 来完成积木化组件间的通信, 其延迟要大于 Linux 中的直接函数调用.

此外, BrickOS 可以利用异构内存保护模块的方法将多个系统组件运行在一个地址空间中, 进一步提升系统的性能 (4.2.3 小节中分析讨论).

4.2.3 内存保护性能测试

我们使用 Intel MPK 保护内存域, 构造了一项微基准测试. 在测试中, 单独运行在的内存域中应用程序会调用一些系统服务, 我们使用 rdtsc 指令计算不同系统服务调用的来回延迟. 为了模拟真实场景中整个调用链中可能会包含多个系统服务的情况, 我们还分别模拟了系统服务间存在相互调用的

表 4 BrickOS 基础操作性能

Table 4 Execution latency of BrickOS's basic operations

Operation	Time cost (cycles)	
	Linux	BrickOS
System call (null)	136	107
Page fault	874	1555
Context switch	955	264
Create process	63268	6848
Read in-memory file (4 kB)	1748	3616
Read in-memory file (2 MB)	314563	754375
Write in-memory file (4 kB)	8402	7443
Write in-memory file (2 MB)	1067556	1299700

表 5 用户态程序调用系统服务的来回延迟

Table 5 Round-trip latency of one application-to-server IPC

Case	Time cost (cycles)	
	seL4	BrickOS
No inter-server IPC	1450	723
One inter-server IPC	2932	856
Two inter-server IPCs	4266	981

情况. 所有测试中的系统服务除了调用其他系统服务外均不进行任何其他操作.

我们将 BrickOS 与 seL4 [6] 进行对比. 在 seL4 中, 不同的系统服务运行在不同的用户态进程中, 分别拥有不同的地址空间; 而 BrickOS 将这些系统服务运行在相同的内核地址空间中, 并使用 Intel MPK 保证不同系统服务间的隔离. 测试结果如表 5 所示. 在无跨系统服务 IPC 的情况下, 亦即用户态进程调用系统服务后立即返回, BrickOS 的性能优于 seL4, 这是因为 BrickOS 中减少了一次特权级切换, 而且 IPC 的设计更加精简. 当出现一次或多次跨系统服务时, BrickOS 的性能表现更好, 这是因为 BrickOS 中的不同系统服务运行在相同的地址空间中, 可以仅通过修改相关的寄存器 (如 Intel MPK 的 PKRU 寄存器) 实现, 而 seL4 只能发送另一个 IPC 请求并切换页表.

上述测试结果表明, BrickOS 提供的内存保护框架可以有效地利用硬件特性完成高效的单地址空间内的内核组件间隔离.

4.2.4 真实应用场景测试

我们基于树莓派 3B+ 平台测试了 BrickOS 的磁盘文件读写时延和网络 I/O 时延. 文件读写测试涉及文件系统与磁盘驱动两类组件, 网络 I/O 测试涉及网络协议栈与网卡驱动两类组件. 文件方面, 分别测试了支持 Ext4[14] 和 FAT32[15] 两个文件系统组件, 与 SD 卡磁盘驱动对接. 网络方面, 选用 lwIP v2.1[16] 网络协议栈组件, 分别与两个内核组件以太网驱动和 Wi-Fi 驱动对接. 我们分别测试了将这些组件部署在用户态和内核态两种编排方式, 测试结果如表 6 和 7 所示. 测试结果表明, 通过 BrickOS

14) Kostka G. lwext4. https://github.com/gkostka/lwext4.

15) ChaN. fatfs. http://elm-chan.org/fsw/ff/archives.html.

16) Adam D. lwIP — a lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip.

表 6 BrickOS 磁盘文件系统读写 4 kB 数据时延

Table 6 Latency of 4 kB read/write in BrickOS disk file system

Operation	Time cost (μs)	
	umod	kmod
Ext4 read	59.7	6.8
Ext4 write	341.5	44.4
FAT32 read	160.9	39.4
FAT32 write	220.3	48.1

表 7 BrickOS 网络收发 1 kB 数据时延

Table 7 Latency of 1 kB send/receive in BrickOS network

Item	Time cost (ms)			
	eth/umod	eth/kmod	wifi/umod	wifi/kmod
TCP connect	21.5	4.9	54.8	6.2
TCP send	0.07	0.03	0.09	0.04
TCP receive	16.9	3.9	18.6	5.4

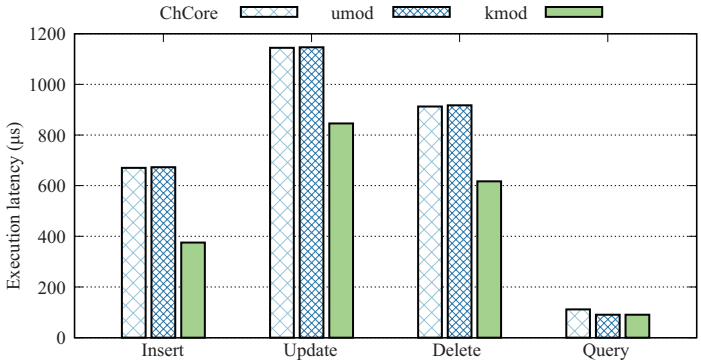

图 9 (网络版彩图) SQLite 3 访问内存文件系统性能开销

Figure 9 (Color online) Execution latency of accessing TmpFS using SQLite 3

将 ChCore 微内核内核组件部署在内核态运行可以有效提升系统性能.

SQLite 数据库[17]是一种嵌入式数据库, 与客户端/服务器架构的数据库 (例如 MySQL, PostgreSQL) 不同, 它自身非常精简, 一般被作为库文件直接链接到应用程序中, 因此被广泛适用于嵌入式设备中, 在移动电话、机顶盒、汽车、机床、遥控器、医疗器械还有机器人等使用场景中都发挥着重要作用.

我们构造了一个使用 SQLite 3 数据库访问内存文件系统的场景来测试 BrickOS 的性能表现. 我们将不做修改的 ChCore 作为基准, 将内存文件系统作为用户态进程运行在用户态, SQLite 3 作为另一个用户态进程访问该内存文件系统, 并执行插入、更新、删除以及查询操作. 我们利用 BrickOS 内核框架生成两个定制化内核, 第 1 个将内存文件系统运行在用户态 (类似 ChCore), 文件系统需要通过 IPC 机制与内核中其他模块通信; 第 2 个将其运行在内核态, 文件系统可以直接通过函数调用的方式访问内核中的其他模块. SQLite 3 应用程序始终运行在用户态.

17) SQLite. https://www.sqlite.org/index.html.

测试结果如图 9 所示, 其中 umod 表示文件系统运行在用户态, kmod 表示文件系统运行在内核态, 纵轴表示一次数据库操作的执行时间. 测试结果表明, 当文件系统运行在用户态时, 其性能与 ChCore 基准相近, 查询操作略快于 ChCore, 可能的原因是本机制的实现相较于 IPC 更精简, 针对特定的负载有一定的优化. 当文件系统运行在内核态时, 数据库的增删改查操作获得了 19%∼43% 的性能提升, 这主要是由于运行在内核态的文件系统无需通过 IPC 来访问内核中的其他模块 (例如储存设备驱动), 而是可以直接通过函数调用的方式来访问.

4.3 有效性与局限性分析

本小节分为两部分: 第 1 部分是有效性分析, 旨在说明 BrickOS 符合泛在操作系统内核的设计需求; 第 2 部分分析总结了 BrickOS 目前存在的问题与局限性, 并指出未来的研究方向.

4.3.1 有效性分析

泛在操作系统内核应当能够面向多领域适应多种异构平台, BrickOS 的设计实现符合其要求.

首先, BrickOS 的内核组件可以自由定制. 在面向特殊使用场景时, 用户可以选择需要的组件自由组合为新的内核. BrickOS 首次总结出内核提供的基础抽象与系统服务, 其中基础抽象在底层屏蔽了硬件的异构性, 使得系统服务可以摆脱对硬件特性的依赖. 过去的许多工作 [23∼29] 虽然也涉及内核组件化的研究, 但是一方面它们缺乏抽象与服务的划分, 这为组件开发带来困难; 另一方面它们的使用场景往往受限, 仅为微内核或者简要内核架构设计, 而且无法面向多领域适应多种异构平台.

其次, BrickOS 可以在不同的内核架构间自由切换. BrickOS 支持使用配置文件自动生成对应架构的操作系统内核, 这一方面得益于积木化的内核组件间精心设计的通信机制, 且它们不存在任何隐式的信息共享; 另一方面 BrickOS 可以根据需求自动地将积木化组件运行在内核态或者用户态, 进而决定了内核架构. 过去的许多工作 [23∼25] 虽然也可以定制内核架构, 但是往往需要大量的人工操作并分别适配内核态组件与用户态组件的代码. 本文则提出选择性地将内核组件与微内核库或者用户态库链接, 由一份源代码即可生成两种运行方式的组件, 且本工作研制了内核组件胶水代码生成机制, 可以自动生成组件间的信息传递、流程控制等代码.

再次, BrickOS 兼顾了运行效率与安全性. BrickOS 提出了组件间高效消息传递的方案: 允许多个组件运行在一个地址空间中, 这些组件间可以直接通过函数调用的方式完成组件间通信. 得益于 BrickOS 支持灵活的组件编排, 用户可以将具有相同安全隐患的组件运行在同一个地址空间中以减少跨地址空间通信带来的性能开销. 此外, 本工作还深入研究了异构内存保活模块的抽象, 支持使用主流的内存域隔离机制完成单地址空间内的内存隔离.

最后, BrickOS 的开发难度较低. BrickOS 提供的自动化编排工具和组件间通信的胶水代码生成工具显著减轻了开发者的开发负担, 且一份组件的源代码即可生成运行在内核态和用户态的组件. 此外, 单地址空间的隔离也对开发者透明, BrickOS 会在加载组件时按照开发者的配置文件创建相应的隔离域, 并在跨地址空间的组件通信间插入相关的胶水代码.

4.3.2 未来与展望

BrickOS 的设计与实现仍然存在一些局限性, 本文将它们列为未来的工作.

首先是兼容不同硬件工作量大, 缺乏自动化工具. BrickOS 的设计天生支持兼容异构硬件, 但是需要手动为不同的硬件编写不同的适配代码. BrickOS 目前支持主流的 x86 系列处理器与 Cortex-A 系列处理器, 并计划在未来支持更多的处理器架构和内存保护机制. 尽管如此, 硬件的适配和组件的开

发都可以被视为一次性的工作, 前期投入巨大, 但是后期可以直接进行工厂化组装以适应不同的使用场景.

其次是 BrickOS 暂不支持运行时插拔组件, 内核整体架构与组件组合方式都在生成内核时被决定. 本工作未来将探索如何在运行时将组件加载到内核态或者用户态地址空间, 以增强 BrickOS 的可扩展性; 还将探索如何在运行时将组件在内核态与用户态之间迁移, 因此可以在不重启的情况下将有缺陷的积木组件隔离运行到独立的地址空间以保证整个系统的安全性.

5 总结

本文针对泛在操作系统研发过程中的硬件资源异构性问题提出了一种积木式内核 BrickOS, 它的内核组件可以自由定制以适应不同场景, 它的架构可以在宏内核、微内核、简要内核、混合内核等多种架构间切换以适应不同硬件. BrickOS 还提出将组件运行在同一个地址空间以提升通信性能, 并通过引入领域隔离的抽象提升整个系统的安全性.

参考文献

1 Weiser M. The computer for the 21st century. Sci Am, 1991, 265: 94–105
2 Mei H, Guo Y. Toward ubiquitous operating systems: a software-defined perspective. Computer, 2018, 51: 50–56
3 Hildebrand D. An architectural overview of QNX. In: Proceedings of USENIX Workshop on Microkernels and Other Kernel Architectures, 1992
4 Swapnil B. Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd
5 Elphinstone K, Heiser G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013
6 Klein G, Elphinstone K, Heiser G, et al. seL4: formal verification of an OS kernel. In: Proceedings of the 22nd Symposium on Operating Systems Principles, 2009
7 Ray D. Advanced MS-DOS Programming. Redmond: Microsoft Press, 1988
8 Engler D R, Kaashoek M F, O'Toole Jr. J. Exokernel: an operating system architecture for application-level resource management. SIGOPS Oper Syst Rev, 1995, 29: 251–266
9 Hunt G C, Larus J R. Singularity: rethinking the software stack. SIGOPS Oper Syst Rev, 2007, 41: 37–49
10 Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: library operating systems for the cloud. SIGARCH Comput Archit News, 2013, 41: 461–472
11 Roy R, Dutta S, Biswas S, et al. Android Things: a comprehensive solution from things to smart display and speaker. In: Proceedings of International Conference on IoT Inclusive Life (ICIIL 2019), 2020
12 Wentzlaff D, Gruenwald C, Beckmann N, et al. An operating system for multicore and clouds: mechanisms and implementation. In: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010
13 Rhoden B, Klues K, Zhu D, et al. Improving per-node efficiency in the datacenter with new OS abstractions. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011
14 Liu R, Klues K, Bird S, et al. Tessellation: space-time partitioning in a manycore client OS. In: Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism, 2009
15 Chapin J, Rosenblum M, Devine S, et al. Hive: fault containment for shared-memory multiprocessors. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995
16 Baumann A, Barham P, Dagand P, et al. The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the 22nd Symposium on Operating Systems Principles, 2009
17 Min C, Kang W, Kumar M, et al. Solros: a data-centric operating system architecture for heterogeneous computing. In: Proceedings of the 13th EuroSys Conference, 2018
18 Asmussen N, Völp M, Nöthen B, et al. M3: a hardware/operating-system co-design to tame heterogeneous manycores.

In: Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, 2016
19 Barbalace A, Sadini M, Ansary S, et al. Popcorn: bridging the programmability gap in heterogeneous-isa platforms. In: Proceedings of the 10th European Conference on Computer Systems, 2015
20 Nightingale E B, Hodson O, McIlroy R, et al. Helios: heterogeneous multiprocessing with satellite kernels. In: Proceedings of the 22nd Symposium on Operating Systems Principles, 2009
21 Shan Y Z, Huang Y T, Chen Y L, et al. LegoOS: a disseminated, distributed OS for hardware resource disaggregation. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, 2018
22 Gu J Y, Wu X Y, Li W T, et al. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In: Proceedings of the USENIX Conference on Usenix Annual Technical Conference, 2020
23 Hall R S, Heimbigner D, Wolf A L. A cooperative approach to support software deployment using the Software Dock. In: Proceedings of the 21st International Conference on Software Engineering, 1999
24 Fassino J, Stefani J, Lawall J, et al. THINK: a software framework for component-based operating system kernels. In: Proceedings of USENIX Annual Technical Conference (USENIX ATC 02), 2002
25 Coulson G, Blair G, Grace P, et al. A component model for building systems software. In: Proceedings of the IASTED Conference on Software Engineering and Applications, 2008
26 Kuenzer S, Bădoiu V, Lefeuvre H, et al. Unikraft: fast, specialized unikernels the easy way. In: Proceedings of the 16th European Conference on Computer Systems, 2021
27 Lefeuvre H, Bădoiu V, Jung A, et al. FlexOS: towards flexible OS isolation. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022
28 Magoutis K, Brustoloni J C, Gabber E, et al. Building appliances out of components using Pebble. In: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, 2000
29 Thomas G. eCos: an operating system for embedded systems. Dr. Dobb's J Softw Tool Prof Programmer, 2000, 25: 66–72
30 Chen X Q, Xu D, Teng Q M. JBEOS: a component-based embedded operating system. Acta Electron Sin, 2005, 33: 2476–2480 [陈向群, 徐冬, 滕启明. JBEOS: 一种构件化的嵌入式操作系统. 电子学报, 2005, 33: 2476–2480]
31 Xu S, Wang W, Zhang J, et al. High performance computing algorithm and software for heterogeneous computing. J Softw, 2021, 32: 2365–2376 [徐顺, 王武, 张鉴, 等. 面向异构计算的高性能计算算法与软件. 软件学报, 2021, 32: 2365–2376]
32 Jiang X B, Xiong Y X, Zhang H, et al. ChattyGraph: highly scalable graph computing system for heterogeneous multi accelerators. J Softw, 2023, 34: 1977–1996 [蒋筱斌, 熊轶翔, 张珩, 等. ChattyGraph: 面向异构多协处理器的高可扩展图计算系统. 软件学报, 2023, 34: 1977–1996]
33 Tao X H, Zhu Y, Pang J M, et al. Parallel code generation for sunway heterogeneous architecture. J Softw, 2023, 34: 1570–1593 [陶小涵, 朱雨, 庞建民, 等. 面向申威异构架构的并行代码自动生成. 软件学报, 2023, 34: 1570–1593]
34 Zong Z, Yang Z B, Yuan S H, et al. Co-modeling and code generation for safety-critical heterogeneous software. J Softw, 2021, 32: 904–933 [宗喆, 杨志斌, 袁胜浩, 等. 安全关键异构软件混合建模及代码生成方法. 软件学报, 2021, 32: 904–933]
35 Li Y, Chen Y, Zhao J X, et al. Survey of ubiquitous computing security. J Comput Res Dev, 2022, 59: 1054–1081
36 Cao D G, Xue D L, Ma Z Y, et al. XiUOS: an open-source ubiquitous operating system for industrial Internet of Things. Sci China Inf Sci, 2022, 65: 117101
37 Bershad B N, Anderson T E, Lazowska E D, et al. Lightweight remote procedure call. ACM Trans Comput Syst, 1990, 8: 37–55
38 Du D, Hua Z C, Xia Y B, et al. XPC: architectural support for secure and efficient cross process call. In: Proceedings of the 46th International Symposium on Computer Architecture, 2019
39 Liedtke J. Improving IPC by kernel design. In: Proceedings of the 14th ACM Symposium on Operating Systems Principles, 1993
40 Watson R N M, Norton R M, Woodruff J, et al. Fast protection-domain crossing in the CHERI capability-system architecture. IEEE Micro, 2016, 36: 38–49

41 Mi Z Y, Li D J, Yang Z H, et al. SkyBridge: fast and secure inter-process communication for microkernels. In: Proceedings of the 14th EuroSys Conference, 2019
42 Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual. 2022
43 Gu J Y, Li H, Li W T, et al. EPK: scalable and efficient memory protection keys. In: Proceedings of USENIX Annual Technical Conference (USENIX ATC 22), 2022

BrickOS: specialized kernels for heterogeneous hardware resources

Jinyu GU[1], Hao LI[1], Yubin XIA[1], Haibing GUAN[2], Zuohua DING[3], Yongwang ZHAO[4] & Haibo CHEN[1*]

1. *School of Software, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai* 200240*, China*;
2. *Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai* 200240*, China*;
3. *School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou* 310018*, China*;
4. *School of Cyber Science and Technology, Zhejiang University, Hangzhou* 310007*, China*
* Corresponding author. E-mail: haibochen@sjtu.edu.cn

Abstract The emerging field of human-machine-material integration requires new operating system kernels to support the ubiquitous computing, so as to manage and control massive heterogeneous hardware and serve dynamic and changeable application scenarios. This paper proposes a configurable specialized kernel architecture, named BrickOS, which can flexibly select the system components of the kernel according to the usage scenario. Developers can choose to run the system components in user mode to provide better security, or run in a shared-address-space kernel mode to improve performance. In order to ensure the security of system components running in the same address space, BrickOS provides a unified abstraction for the memory protection mechanism of the underlying hardware used for memory isolation in a single address space. The test results show that BrickOS can generate customized kernels that meet the requirements for different scenarios, with low inter-process call (IPC) overhead and good overall performance.

Keywords operating system kernel, components arrangement, inter-process communication, memory isolation