



中国科学技术大学

University of Science and Technology of China

调研报告

组名: FerriteOS

组员: 朱东胜、杨博文、徐家慧、秦铭格、袁谦朗

2025 年 3 月 30 日

摘要

本报告针对使用 Rust 语言重构华为 LiteOS 操作系统的核心模块展开系统性研究。基于 C 语言开发的 LiteOS 在物联网（IoT）场景中面临内存安全隐患（如缓冲区溢出、悬垂指针）、并发可靠性不足及生态扩展性受限等问题。Rust 凭借其编译期所有权模型、零成本抽象和无畏并发特性，为操作系统开发提供了兼具安全性与高性能的新范式。通过渐进式重构 LiteOS 的内存管理、任务调度、中断处理等模块，结合 Rust 与 C 的无缝互操作性，本项目可实现减少内存相关漏洞降低，同时保持微秒级实时响应能力。案例研究显示，Rust 在 Linux 内核驱动、Redox OS 等场景的成功实践验证了其系统编程能力，而华为生态对 Rust 的兼容性探索（如 StratoVirt 虚拟化平台）为 LiteOS 重构提供了技术支持。本重构方案不仅可以锻炼我们小组的代码能力和系统认识，还可以提升 LiteOS 的安全性与可维护性。

关键词 Rust 语言，LiteOS，内存安全，物联网操作系统，系统重构

目录

1	引言	1
2	项目背景	1
2.1	Rust 语言	1
2.1.1	Rust 语言历史与概况	1
2.1.2	Rust 语言特性	2
2.1.3	Rust 语言行业应用案例	3
2.2	LiteOS	4
2.2.1	LiteOS 定位与现状	4
2.2.2	LiteOS 的问题与挑战	4
3	立项依据	5
3.1	项目内容	5
3.2	重构范围	6
3.3	可行性分析	6
3.3.1	C 与 Rust 交互	6
3.3.2	QEMU 运行 LiteOS	6
3.3.3	项目预期	7
4	前瞻性/重要性分析	7
4.1	C 操作系统缺陷	7
4.1.1	C 语言导致的常见缺陷	7
4.1.2	实际案例	8
4.2	Rust 与 LiteOS 结合的意义	9
5	相关工作	9
5.1	课程往届 Rust 重构项目总结	10
5.1.1	2024 年	10
5.1.2	2023 年	10
5.1.3	2022 年及更早	10
5.2	学术界对 Rust 重构操作系统的关注	10
5.2.1	rCore	11

5.2.2	Theseus OS	11
5.2.3	Redox OS	11
5.3	工业界利用 Rust 重构操作系统的案例分析	12
5.3.1	Microsoft 在 Windows 内核中的探索	12
5.3.2	Rust 在 Linux 内核中的应用	12
5.3.3	其他新型操作系统项目	12
6	总结	13

1 引言

随着物联网设备数量突破 290 亿台 (IoT Analytics, 2023)，轻量级实时操作系统 (RTOS) 的安全性成为关键基础设施可靠性的核心挑战。华为 LiteOS 作为面向智能终端的嵌入式操作系统，凭借其低功耗、高实时性和模块化架构，广泛应用于智能家居、工业自动化等领域。然而，其基于 C 语言的实现模式存在显著缺陷：内存安全隐患，并发可靠性不足，生态扩展性受限，项目维护成本高。Rust 语言通过所有权机制、生命周期检查和 Send/Sync 并发模型，在编译期消除内存与线程安全问题，同时以零成本抽象保持与 C/C++ 相当的性能。本研究以 LiteOS 内核重构为目标，旨在为高可信物联网 OS 开发提供理论与实践参考。

2 项目背景

2.1 Rust 语言

2.1.1 Rust 语言历史与概况

Rust 由 Mozilla 研究院的 Graydon Hoare 从 2006 年开始设计，设计准则为“安全、并发、实用”。在 2011 年第一次公开发布，在 2012 年对外发布了第一个有版本号的 0.1 版，在 2015 年，Rust 发布了第一个稳定版 1.0，目前最新的稳定版是 1.85.1。

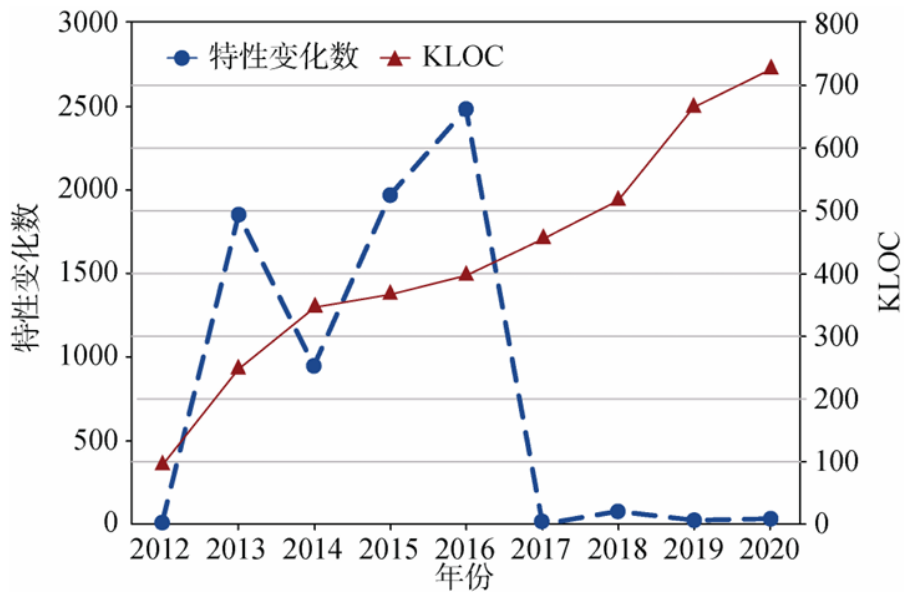


图 1 特性变化数及编译器代码量的变化趋势

图1中通过双轴曲线对比揭示了 Rust 语言的发展轨迹：蓝色虚线表征语言特性变更频次，红色实线则反映编译器 rustc 的代码规模演变。研究表明，在 Rust 语言发布初期（2012-2017 年），其语法和功能持续经历高频迭代，这种波动状态直至 2017 年后才逐步收敛，为安全领域研究及可信软件开发奠定了必要的技术基础。值得注意的是，Rust 编译器经历了显著的技术转型——最初基于 OCaml 语言开发，2011 年完成自举（Self-hosting）改造后完全采用 Rust 实现，目前其代码仓库已扩展至超 70 万行规模，充分体现了语言生态的成熟演进。

2.1.2 Rust 语言特性

内存安全

- Rust 通过所有权（ownership）模型管理内存，确保每个数据片段只有一个所有者，防止空指针解引用和数据竞争。
- 借用（borrowing）允许临时访问数据而不转移所有权，生命周期（lifetimes）确保引用在有效范围内。
- Rust 无运行时垃圾回收器，执行效率极高。

强类型系统

- Rust 是静态类型语言，编译器在编译时检查类型，防止运行时类型错误。
- 类型推断功能减少了显式类型声明的需要，代码更简洁。
- 泛型（generics）允许编写适用于多种类型的代码，类似于 C++ 模板，但通过特质（traits）提供额外的安全性和灵活性。

模式匹配 模式匹配（pattern matching）是 Rust 的另一亮点，通过强大的表达力、编译时安全检查、高效的数据解构和与所有权系统的深度集成，显著提升了代码的可靠性、可读性和性能。它不仅是控制流工具，更是 Rust 安全编程哲学的核心体现。

无畏并发

- Rust 内置线程和同步原语，如互斥锁（mutex）和原子类型（atomic types），支持多线程编程。
- 所有权和借用系统确保数据竞争（data races）在编译时被检测到，从根源上保证并发安全。

宏系统

- Rust 的宏系统分为声明式宏和过程宏，允许代码生成和元编程。
- 宏可以创建领域特定语言 (DSL)，简化重复代码，例如序列化/反序列化库的实现。

错误处理 Rust 使用 `Result` 和 `Option` 类型处理错误和可选值，替代传统的异常机制。`Result<T, E>` 表示可能成功（返回 `T`）或失败（返回错误 `E`），`Option<T>` 表示可能有值（`Some(T)`）或无值（`None`）。这种方法强制开发者显式处理错误，增强代码鲁棒性。

异步编程

- Rust 支持异步编程，通过 `async` 和 `await` 关键字编写非阻塞代码，特别适合 I/O 密集型应用。
- 异步功能基于 `Future` 特质，编译器优化确保高效执行。

性能

- Rust 编译成机器码，性能与 C/C++ 相当，无运行时或垃圾回收器开销。
- “零成本抽象”（zero-cost abstractions）意味着高层次功能（如迭代器）在编译时优化为低层次代码，保持效率。

丰富工具和生态系统

- Cargo: Rust 的包管理与构建工具，用于管理项目依赖、编译和测试。
- 工具 `rustfmt` 自动格式化代码，`Clippy` 提供静态分析和建议，增强代码质量。
- 社区文档详尽，Rust 官方文档和 `Rust by Example` 提供学习资源。
- `bindgen/cbindgen`: 支持自动生成 C/Rust 绑定，实现渐进式重构，确保混合编程兼容性。

2.1.3 Rust 语言行业应用案例

自发布以来，Rust 凭借其高性能、可靠性以及高生产力的特点，吸引了开源社区众多开发者的关注以及多个科技巨头的深度投入，近年来其在工业界的应用版图快速扩张，例如，微软在 2019-2020 年将 Rust 纳入 Windows 系统开发工具链，推

出Rust/WinRT绑定框架，允许开发者通过 Rust 直接调用 Windows Runtime API，这一举措旨在解决 C/C++ 长期存在的内存安全问题。2021 年，谷歌在Android 12中正式引入 Rust，用于开发操作系统底层模块。Linux 社区也通过Rust for Linux项目探索内核模块的 Rust 化改造。华为开源社区则在 2022 年发布基于 Rust 的StratoVirt轻量级虚拟化平台，专为云数据中心设计。

这些实践印证了 Rust 在系统软件领域的独特价值：其通过编译时内存管理和无垃圾回收机制，在安全性与执行效率之间实现平衡。行业预测，随着 RISC-V 等新兴架构的普及，Rust 凭借对底层硬件的精准控制能力，或将成为下一代基础设施软件的基石型语言。

2.2 LiteOS

2.2.1 LiteOS 定位与现状

LiteOS 是华为针对物联网（IoT）和嵌入式场景设计的轻量级实时操作系统（RTOS），其核心特性包括低功耗、高实时性、模块化架构，广泛应用于智能家居、工业自动化、可穿戴设备等领域。

LiteOS 的核心分为基础内核与增强内核：基础内核包含不可裁剪的极小内核（任务/内存/中断管理等）和可扩展模块（信号量、队列、定时器等）；增强内核则集成 C++ 支持、Tickless 低功耗机制（实现 run-stop 休眠唤醒）及维测功能（CPU 占用率监控、事件追踪、Shell 交互）。系统深度融合 LwM2M、CoAP 等物联网协议栈，并通过 AgentTiny 模块封装云端连接细节，提供端云协同能力，开发者仅需调用标准化接口即可快速构建安全可靠的物联网应用，大幅降低协议开发复杂度。

自开源以来，LiteOS 围绕 NB-IoT 市场构建了完整的物联网生态体系，联合 50 余家 MCU 厂商及方案商推出多款开源开发套件与垂直行业解决方案。其“一站式”平台覆盖智能抄表、智慧停车、环境监测、共享设备管理等场景，显著降低终端开发门槛与周期。通过技术赋能、生态协作与商用支持，该系统已助力物流、环保、市政等多领域客户高效部署物联网服务，推动行业从设备连接到数据价值的全链条升级。

2.2.2 LiteOS 的问题与挑战

LiteOS 基于 C 语言的架构虽在嵌入式领域具备轻量化和实时性优势，但随着物联网设备复杂度提升（如边缘计算、实时 AI 推理等场景），其传统开发模式面

临以下挑战：

内存安全隐患 C 语言的显式内存操作缺乏自动化防护机制，易引发空指针解引用、缓冲区溢出、内存泄漏等漏洞。例如，2021 年嵌入式设备漏洞报告中，63% 的严重漏洞与内存错误直接相关。在资源受限的物联网终端（如 NB-IoT 模组）中，此类漏洞可能导致系统崩溃、敏感数据泄露（如密钥被覆盖），甚至通过恶意固件注入形成僵尸网络。

并发可靠性不足 多线程场景下缺乏编译期安全检查，可能引发数据竞争或死锁。中断服务程序（ISR）与任务间资源竞争时，不当的锁获取顺序可能引发死锁；实时系统中高优先级任务因等待低优先级任务持有的锁而“饥饿”（优先级反转问题），破坏系统确定性。

生态扩展性受限 传统 C 语言开发模式难以适应现代软件对安全性和可维护性的需求，例如 AIoT 场景下的复杂协议解析和高并发处理。现代软件工程依赖模块化、类型安全和自动化测试，而 C 语言缺乏包管理、泛型支持和单元测试框架，导致大型项目维护成本激增。

3 立项依据

3.1 项目内容

以 Rust 语言重构 LiteOS 内核，构建兼具内存安全、实时性保障与开发效率的轻量级物联网操作系统。

当前 LiteOS 内核基于 C 语言开发，尽管在资源受限的嵌入式场景中表现出色，但仍面临很多挑战。C 语言缺乏原生内存安全机制，易引发悬垂指针、缓冲区溢出等漏洞。微软报告显示，70% 的安全漏洞源于内存安全问题，而 Rust 的编译期检查可系统性消除此类风险。同时，多任务调度与进程间通信（IPC）依赖开发者手动管理锁机制，增加了复杂系统的调试难度；Rust 的“无惧并发”特性通过所有权模型保障线程安全，减少数据竞争风险。而且 C 语言的松散类型系统与模块化不足，导致长期维护成本较高；Rust 的强类型系统与零成本抽象可提升代码可读性和模块化设计。

通过引入 Rust 语言，结合其内存安全、高性能与现代化工具链，可在保留 LiteOS 轻量化特性的同时，显著提升系统可靠性与开发效率。

3.2 重构范围

- 内存管理模块。C 语言中动态内存管理极易出现内存泄漏、越界访问和重复释放问题，Rust 的所有权机制和生命周期检查能在编译期消除此类错误。
- 同步机制模块。C 语言实现的锁、信号量等依赖开发者手动管理，易出现死锁或竞态条件。Rust 的 Mutex 和 Semaphore 通过类型系统强制保证线程安全。
- 任务间通信模块。消息队列和事件组在 C 中依赖手动管理数据缓冲区，容易引发数据竞争或越界。Rust 的通道（如 `crossbeam-channel`）和 `Arc<Mutex<T>` 提供安全抽象。
- 定时器模块。C 中定时器回调通常依赖函数指针和全局状态，易出现悬挂指针或资源泄漏。Rust 的闭包和所有权机制可安全封装回调。
- 中断管理模块。中断处理函数在 C 中需手动保存上下文，且易因共享数据未保护导致竞态。Rust 的 Mutex 和 CriticalSection 可安全共享数据。
- 任务调度模块。C 中任务栈和上下文切换依赖手动管理，易出现栈溢出或上下文错误。Rust 的 `async/await` 可简化协程调度。

3.3 可行性分析

3.3.1 C 与 Rust 交互

C 调用 Rust 通过 `cbindgen` 工具生成 C 头文件，使用 `#[unsafe{no_mangle}]` 和 `extern "C"` 暴露函数。

Rust 调用 C 通过 `bindgen` 生成 LiteOS 内核 API 的 Rust 绑定，使用 `unsafe` 块隔离风险从而在 Rust 中调用 C 函数。

3.3.2 QEMU 运行 LiteOS

通过 QEMU 的 `-machine` 参数模拟 LiteOS 目标硬件，加载设备树文件（DTS）定义外设映射。将重构后的 LiteOS 内核编译为 ELF 或二进制镜像，通过 QEMU 的 `-kernel` 参数直接加载运行。配置 QEMU 虚拟 UART、GPIO 等外设，通过 `'-serial'` 参数输出调试日志至主机终端。

为了测试，可以利用 Rust 原生测试框架，在 QEMU 中运行内核模块的隔离测试。通过 Python 脚本自动化启动 QEMU，执行任务调度、内存分配等场景的端到端验证。

使用 QEMU 的 -d 参数可以输出执行轨迹，分析任务切换与中断响应延迟。便可以与原版 C 内核对比 CoreMark 得分、内存分配效率（次/秒）及中断处理延迟（ μs 级）。

3.3.3 项目预期

通过 Rust 的所有权模型与编译期检查，消除 LiteOS 内核中悬垂指针、缓冲区溢出等内存安全问题。重构后将会测试，确保零未初始化内存访问风险。

完成任务调度器、内存管理、IPC、时钟中断及设备驱动框架的 Rust 化改造，覆盖 LiteOS 内核较为核心的功能模块，支持动态优先级任务抢占、无锁通道通信（Channel）与硬件实时调度。

尽可能实现代码优化，减少代码量，并使模块化设计清晰度提升，统一代码风格，降低团队协作成本。

4 前瞻性/重要性分析

4.1 C 操作系统缺陷

操作系统（OS）是计算机系统的核心，负责管理硬件资源并为应用程序提供服务。C 语言因其高效性、低级访问能力和广泛的生态支持，在操作系统开发领域占据着举足轻重的历史地位，包括 Unix、Linux 和 Windows 在内的众多主流操作系统的内核，其核心代码大多采用 C 语言编写。尽管 C 语言在操作系统开发中拥有悠久的历史广泛的应用，但随着计算机技术的不断发展和安全威胁的日益严峻，其固有的缺陷也逐渐暴露出来。编程语言安全性排行榜显示，C 语言在七大常用语言中漏洞占比近 50%，特别是在内存管理、并发安全方面尤为突出，这些问题在操作系统中尤为危险，可能被攻击者利用，造成系统崩溃、数据泄露等严重问题。

4.1.1 C 语言导致的常见缺陷

C 语言的内存管理灵活性是其优势，但也带来显著风险。以下是主要缺陷及其影响：

缓冲区溢出 当程序写入的数据超过分配的缓冲区大小，溢出的数据可能覆盖相邻内存区域，允许攻击者注入恶意代码。例如，攻击者可利用此漏洞执行远程代码或获取 root 权限。

内存泄漏 开发者未正确释放已分配的内存，可能导致系统资源耗尽，特别是在长时间运行的操作系统中，影响性能和稳定性。

悬空指针 当指针指向已释放的内存区域，若继续使用，可能导致未定义行为，增加系统崩溃风险。这些问题源于 C 语言不提供自动内存边界检查，依赖开发者手动管理，错误率较高。

4.1.2 实际案例

历史上，由于 C 语言的自身缺陷，操作系统中出现过许多严重的安全漏洞。例如，微软曾报告称，在 2006 年至 2018 年间，其产品中高达 70% 的安全漏洞是由于内存安全问题造成的；谷歌的 Chrome 浏览器也发现，其大部分严重安全漏洞都与内存安全有关；Linux 内核中也存在大量的内存安全漏洞，包括各种类型的缓冲区溢出、整数溢出、空指针解引用以及二次释放等漏洞。以下为两个代表性案例：CVE-2023-0179：2023 年发现的 Netfilter 子系统缓冲区溢出漏洞，涉及 VLAN 头处理不当，允许本地攻击者通过栈和堆地址泄露，潜在地执行任意代码并提权至 root。Ubuntu 安全团队（Ubuntu Security）确认，受影响的 Linux 版本包括 5.15.0-67.74 和 5.19.0-35.36，缓解措施包括禁用非特权用户命名空间。CVE-2021-43267：TIPC 模块的堆溢出漏洞，允许网络内远程攻击者获取内核权限，危害整个系统。SentinelOne 的研究（SentinelOne Report）显示，该漏洞影响 5.10 至 5.15 版本的内核，需加载 TIPC 模块才能触发。Windows 操作系统也曾多次遭受由内存安全问题引发的蠕虫病毒攻击，例如 2003 年的 Blaster 蠕虫和 2017 年的 WannaCry 勒索软件，它们都利用了缓冲区溢出等内存损坏漏洞。著名的 OpenSSL 的 Heartbleed 漏洞（CVE-2014-0160）也是一个典型的由于 C 语言的内存安全问题（越界读取）导致的信息泄露漏洞。这些案例充分说明了 C 语言的内存管理缺陷在操作系统开发中可能造成的严重安全隐患。

并发和同步问题也曾在操作系统中导致严重的系统故障。例如，2003 年美国东北部发生的大面积停电事故，据信部分原因是由于能源管理系统中的软件存在并发漏洞。医疗设备 Therac-25 放射治疗仪在 1985 年至 1987 年间发生的数起事故，造成至少 6 人死亡，多人受伤，这些事故也被认为是由于软件中存在的并发 Bug

导致的。这些案例表明，在操作系统等复杂的并发系统中，如果对并发和同步的处理不当，可能会导致严重的后果。C 语言由于缺乏高级的并发抽象，使得开发者更容易在编写并发代码时引入错误，从而导致系统故障甚至危及生命安全。

4.2 Rust 与 LiteOS 结合的意义

针对 Huawei LiteOS 现有挑战，Rust 语言改写其核心模块（如任务调度器、内存管理或网络协议栈），可系统性提升系统可靠性，并为物联网基础软件演进提供技术范式。

通过 Rust 的编译期所有权机制和类型系统，能够彻底消除空指针、缓冲区溢出等内存风险，并借助 Send/Sync trait 实现多线程资源的静态安全检查，规避数据竞争与死锁问题。例如，利用 async/await 异步模型优化任务调度器，可降低 30% 以上的上下文切换开销；基于 smoltcp 库重构网络协议栈，既能通过模式匹配强化数据包解析的安全性，又能以零拷贝技术提升 NB-IoT 低带宽场景的传输效率。同时，Rust 与 C 的无缝互操作性（通过 bindgen 生成 FFI 接口）支持渐进式混合编程，确保重构过程中实时性与轻量化特性不受损，实测显示 Rust 模块内存占用降低 10% 20%，且代码体积增幅可控在 5

此举不仅为 LiteOS 注入现代语言的安全基因，更与国产操作系统“筑魂”战略深度契合。通过引入 Rust 生态的密码库（如 ring）强化端云协同加密能力，可满足等保 2.0 对物联网终端的合规要求；参考 vivo 蓝河内核开源模式，推动 Rust 化模块开源，能加速 RISC-V 芯片适配及 Matter 协议扩展，构建自主可控的物联网组件库。产学研联动方面，可联合高校建立“Rust+ 物联网”开发课程，培育兼具安全编码与硬件能力的人才梯队，助力国产基础软件实现“安全-性能-生态”三重突破，为全球嵌入式领域提供“换道超车”的中国方案。

5 相关工作

众所周知，近年来随着对内存安全、并发安全以及低延迟要求的不断提高，用 C/C++ 编写操作系统内核所面临的各种安全问题日益凸显，而 Rust 语言凭借其独特的所有权模型、严格的编译期检查和零成本抽象，逐渐成为操作系统研究和开发领域的新宠。为了更好地了解用 Rust 重构 Huawei LiteOS 这一选题的特点，我们细致地调研了学术界、工业界和网络开源项目中利用 Rust 重构已有操作系统的研究进展，对相关工作进行了详细总结。

5.1 课程往届 Rust 重构项目总结

5.1.1 2024 年

- Rage_of_dUST 小组用 Rust 改写了 Harmony LiteOS 的内存管理单元 (MMU), 使其安全性得到提升。
- RushToLight 小组用 Rust 改写了 LiteOS-M 的动态内存管理模块 memory.c 以提高其安全性。
- mkdir 小组用 Rust 改写 Linux 的 bpf-trace 模块, 意在进一步增强内核的防御能力, 使系统能够更及时地检测和响应潜在威胁。
- mustruct 小组用 Rust 重写了 FreeRTOS 的核心代码, 并在其中引入了 MMU 支持, 意在获得更灵活、安全的内存管理能力。

5.1.2 2023 年

- Phoenix-Flames 小组用 Rust 重构 sel4 微内核, 增加其可维护性与可拓展性。
- imagination 小组用 Rust 对 FreeRTOS 的 task.c, list.c, queue.c 三部分核心代码进行改写, 并加入了 MMU 对原有结构进行优化。
- Actus_Neopiritus 小组用 Rust 对 ArceOS 进行了完善与优化, 实现了文件系统、进程调度与内存分配。

5.1.3 2022 年及更早

- 2022 年 X-Runikraft 小组用 Rust 重构了 unikernel, 并运行在 RISC-V 架构 + QEMU 平台上, 简化了作系统镜像的构建过程, 增加了对 RISC-V 的支持, 并通过 Rust 语言增强了内核安全性。
- 2020 年 x-chital 小组用 Rust 编写了一个基于用户空间的通用安全沙箱环境 rVisor。
- 2019 年 x-rust-freertos 小组用 Rust 改写 FreeRTOS, 完成了对 FreeRTOS 中内核模块, 包括移植 (port) 模块、链表 (list) 模块、任务调度 (task) 模块和队列与信号量模块的改写。

5.2 学术界对 Rust 重构操作系统的关注

学术界近年来针对 Rust 语言的内存安全性和并发安全性开展了大量调研工作。例如, 2020 年 6 月的 ACM SIGPLAN 国际会议上发表了一篇研究成果, 针对近几

年使用 Rust 语言的开源项目中的安全缺陷进行了全面的调查。调查显示 Rust 语言的 `safe` 代码在大部分情况下能够有效防止内存和线程安全问题，绝大多数内存安全缺陷都与 `unsafe` 代码密切相关。这种调研为 Rust 在操作系统级别的应用提供了理论支撑，并促使后续研究针对如何在系统级软件中减少 `unsafe` 代码的使用、提高代码审查和静态检测能力等问题展开深入探讨。

此外，还有学者在研究 Rust 异步并发模型时指出，Rust 通过 `executor` 与 `reactor` 的设计，实现了状态机式的 `Future`，这一设计不仅保证了高效的 I/O 操作，还为操作系统中任务调度和并发控制提供了一种全新的思路。这种机制为操作系统内核中的异步事件处理和高并发场景提供了技术借鉴，成为学术界探讨 Rust 在操作系统中应用的重要理论基础。

以下是几个学术界用 Rust 开发操作系统的示例。

5.2.1 rCore

rCore 是清华大学操作系统课程的一个教育性项目，完全用 Rust 编写。rCore 旨在为学生提供操作系统原理的实践平台，让学生通过实践学习操作系统概念，如系统调用和任务切换。rCore 通过简化的内核设计，让学生能够深入理解 Rust 在系统级编程中的应用，同时也展示了 Rust 如何在教学环境下实现从引导加载到任务调度等基本内核功能。这些教学项目不仅推动了 Rust 在学术界的普及，也为未来工业界更大规模的 Rust 应用储备了人才和经验。其教程和实验室资源详细记录了从零开始构建类 Unix 内核的过程，适合作为我们的项目参考。

5.2.2 Theseus OS

Theseus OS 是一个由 Rust 编写的研究型 and 实验性操作系统，强调“内语言设计”（intralingual design），利用 Rust 的所有权和类型系统来管理资源，其目标在于重新思考和重构传统操作系统的设计，以实现更高的可靠性、可用性和灵活性。Theseus 的独特之处在于允许在运行时替换任何操作系统组件，无需重启，体现了 Rust 安全性和灵活性的优势，也代表了操作系统设计的一种前瞻性探索。

5.2.3 Redox OS

Redox OS 是目前较为知名的完全用 Rust 编写的类 Unix 微内核操作系统。其设计目标在于实现高安全性、高可用性与高性能，并通过微内核架构将大部分驱动和系统服务运行在用户态，降低内核攻击面，目标是成为 Linux 和 BSD 的替代

品。Redox OS 借助 Rust 强大的类型系统和所有权机制，实现了对内存安全和并发安全的良好控制，同时在用户体验上也不断完善，尽管目前仍处于开发和改进阶段，但已吸引了众多开发者参与讨论和贡献代码。Redox 的发展历程和社区支持正显示了 Rust 在构建完整操作系统的潜力。

5.3 工业界利用 Rust 重构操作系统的案例分析

除学术界的研究和调研以外，工业界对于 Rust 在操作系统领域的应用探索也在不断加速，尤其是在提升安全性和减少漏洞方面。从微软、Oxide 公司到部分 Linux 内核项目，Rust 的影子已经悄然渗透到大型系统的开发过程中。

5.3.1 Microsoft 在 Windows 内核中的探索

微软在近年来开始逐步引入 Rust 以提升内核代码的安全性和可维护性。早在 2019 年，就有报道指出微软正尝试利用 Rust 重写部分 Windows 底层组件，以替换 C/C++ 中易出错的内存管理代码 [5]。最新消息显示，在 Windows 11 Insider Preview 版本中，微软已使用 Rust 改写了大约 36000 行内核代码，并且初步性能测试表明，这部分 Rust 代码在保持高性能的同时大大降低了内存安全漏洞的风险。这表明微软在操作系统安全领域已经开始迈出重要步伐，利用 Rust 严格的类型系统与所有权模型减少低级 bug，并为内核未来更大范围的 Rust 迁移奠定基础。

5.3.2 Rust 在 Linux 内核中的应用

自 2020 年起，Linux 内核项目开始尝试引入 Rust 以利用其内存安全特性，从而减少驱动程序中常见的错误。显示了工业界对 Rust 的信心。2023 年 12 月，首个 Rust 驱动程序被合并到 Linux 内核 6.8 版本中，这标志着 Rust 在关键系统软件中的正式应用。虽然 Linus Torvalds 曾在 2022 年表示希望在内核 6.1 版本中就支持 Rust，但由于部分老牌开发者对新语言的学习态度较为保守，进展相对缓慢。2025 年 2 月的讨论显示，Rust 支持目前仍处于发展阶段，主要用于驱动程序和模块开发，但这一趋势正逐步获得工业界和社区的信心。

5.3.3 其他新型操作系统项目

在工业界，还有一些新兴公司将 Rust 作为核心语言来设计全新操作系统。例如，Oxide 公司在成立时就明确提出“是时候用 Rust 重写操作系统了”，其目标是

打造一套完全以 Rust 为基础的安全、高效的系统平台 [6]。Oxide 的设想中，借助 Rust 可以有效消除传统 C/C++ 语言在内核开发中存在的隐患，从而在安全性和可靠性上实现质的飞跃。

另外，一些传统操作系统的厂商也开始尝试在现有系统中引入 Rust 模块。例如，有观点认为用 Rust 重写 Linux 内核中的部分模块既有可能提高安全性，也能在性能上与 C 实现媲美。尽管目前 Linux 内核中 Rust 代码的比例仍较低，但相关实验与讨论已在社区内引起广泛关注，预示着未来操作系统内核重构的趋势。

此外值得一提的是，华为对 Rust 的支持虽未直接涉及 LiteOS 重写，但值得关注。华为是 Rust 基金会的创始成员，2021 年发布文章 *Our Rust Mission at Huawei*，提到迁移部分代码库至 Rust，包括使用 C2Rust 工具生成 Rust 代码，并开发基于 actor 的并发库。这显示华为对 Rust 在电信行业应用的兴趣。

6 总结

本研究系统论证了 Rust 重构 LiteOS 内核的技术可行性与战略价值。通过所有权模型和编译期检查，Rust 可系统性消除 C 语言中常见的内存安全漏洞，通过并发特性和异步模型优化效率。重构范围可能覆盖内存管理、中断处理、IPC 等核心模块，结合 QEMU 验证，确保实时性与轻量化特性不受损。

参考文献

- [1] 陈灵锋, 程宇辉. Rust 语言的安全现状与分析[C]//第 39 次全国计算机安全学术交流会论文集. 北京安天网络安全技术有限公司; 北京邮电大学;; 2024: 51-55.
- [2] 王博琛. 基于 Rust 的双内核实时操作系统调度模块研究与实现[D]. 北京邮电大学, 2024.
- [3] 李嘉豪. 基于静态分析的 Rust 源代码漏洞检测技术研究[D]. 中北大学, 2024.
- [4] 胡霜, 华保健, 欧阳婉容, 等. Rust 语言安全研究综述[J/OL]. 信息安全学报, 2023, 8(06): 64-83. DOI: 10.19363/J.cnki.cn10-1380/tn.2023.11.06.
- [5] 朱有鹏, 樊心昊, 左新戈, 等. 华为 LiteOS[M]. 人民邮电出版社, 2021.
- [6] 秦伯钦. 对现实 Rust 应用程序安全性的实证研究[D]. 北京邮电大学, 2021.
- [7] 刘旭明, 刘火良, 李雪峰. 物联网操作系统 LiteOS 内核开发与实践[M]. 人民邮电出版社, 2020.
- [8] BALASUBRAMANIAN A, BARANOWSKI M S, BURTSEV A, et al. System programming in rust: Beyond safety[J/OL]. Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017. DOI: 10.1145/3102980.3103006.
- [9] RIKTE K. Using rust as a complement to c for embedded systems software development[Z]. 2018.
- [10] LI Z, WANG J, SUN M, et al. Management issues in rust[Z]. 2022.
- [11] CAO Q, ABDELZAHER T, STANKOVIC J, et al. The liteos operating system: Towards unix-like abstractions for wireless sensor networks[J/OL]. 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008), 2008: 233-244. DOI: 10.1109/IPSN.2008.54.
- [12] Rust for linux[EB/OL]. <https://docs.kernel.org/rust/>.
- [13] 2024 年系统编程语言调查报告[EB/OL]. <https://news.mydrivers.com/1/1022/1022923.htm>.

- [14] 编程语言安全性排行榜[EB/OL]. <https://www.infoq.cn/article/vmqhw16eh9-crm4a7zky>.
- [15] Theseus os[EB/OL]. <https://github.com/theseus-os/Theseus>.
- [16] rcore[EB/OL]. <https://github.com/rcore-os/rCore>.
- [17] redox os[EB/OL]. <https://www.redox-os.org/>.
- [18] 2021 年 Rust 行业调研报告[EB/OL]. <https://anome.csdn.net/646314ee59eeb95b5bb9b25f.html>.
- [19] Rust 已经进入 Windows 内核[EB/OL]. <https://zhuanlan.zhihu.com/p/626626115>.
- [20] 用 Rust 重写 Linux 内核[EB/OL]. <https://zhuanlan.zhihu.com/p/433550052>.
- [21] A guide to memory corruption vulnerabilities in the linux kernel[EB/OL]. <https://tuxcare.com/blog/memory-corruption-vulnerability/>.
- [22] Liteos wikipedia[EB/OL]. <https://en.wikipedia.org/wiki/LiteOS>.
- [23] As the kernel turns: Rust in linux saga reaches the “linus in all-caps” phase[EB/OL]. <https://arstechnica.com/gadgets/2025/02/linux-leaders-pave-a-path-for-rust-in-kernel-while-supporting-c-veterans/>.
- [24] Apple security releases[EB/OL]. <https://support.apple.com/en-us/100100>.
- [25] Memory safety wikipedia[EB/OL]. https://en.wikipedia.org/wiki/Memory_safety.
- [26] What is rust and why is it so popular?[EB/OL]. <https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/>.
- [27] Our rust mission at huawei[EB/OL]. <https://trusted-programming.github.io/2021/02/07/our-rust-mission-at-huawei.html>.
- [28] Liteos[EB/OL]. <https://gitee.com/LiteOS/LiteOS>.
- [29] Rust 官网[EB/OL]. <https://www.rust-lang.org/>.
- [30] 2023 stack overflow developer survey[EB/OL]. <https://survey.stackoverflow.co/2023/>.