题目

ReL4: 基于用户态中断的异步微内核设计与实现

摘要

以seL4为代表的现代微内核操作系统将同步IPC作为进程间通信的主要方式，同时借助需要内核转发的通知机制来实现用户态的中断处理和异步通信。然而随着微内核生态的发展，内核有必要支持需要进行大量和频繁的系统调用和IPC的应用，同步系统调用和IPC导致了大量的上下文切换和二级缓存失效，同时由于阻塞等待导致系统无法充分利用多核的性能；此外，seL4中的通知机制广泛用于用户态驱动、线程间异步通信等，但由于需要内核进行转发，其中的上下文切换和缓存失效在某些平台将造成不可忽视的开销。

本文聚焦微内核的异步通信机制，利用用户态中断技术改造seL4的通知机制，使得信号无需通过内核转发，减少上下切换的开销。同时利用无需内核转发的通知机制，设计和实现避免陷入内核的异步系统调用和异步IPC框架，在提升用户态并发度的同时，减少用户态和内核态的切换次数，提升系统的整体性能。

关键词：微内核；异步；中断。

1. 选题依据

1.1 选题背景及意义

微内核从被提出以来，最大的性能瓶颈就是IPC（进程间通信），30年前Liedtke[1]提出的L4通过对内核系统的重新设计，证明了微内核的IPC也可以很快，之后以seL4[2]为代表的微内核的IPC框架也基本延续了最初的L4，以同步IPC作为主要的通信方式，同时引入异步的通知机制来简化多线程程序设计，并提升多核的利用率[3]。然而随着软件性能要求不断提升，seL4中的IPC通信方式并不能很好的满足。

首先是软件对系统有了新的要求，随着软件复杂性的提升，系统级软件如数据库管理系统、网络服务器等，需要进行大量的系统调用和IPC，这要求系统能够以快速高效的形式处理大量系统调用和IPC[4]，而微内核将操作系统的大部分服务（如网络协议栈、文件系统等）移到用户态，从而使得IPC数量和频率激增，内核态与用户态之间的上下文切换成为性能瓶颈。此外，新出现的硬件漏洞如Meltdown[5] 和 Spectre[6] 漏洞促使 Linux 使用 KPTI 补丁[7]来分离用户程序和内核的页表，进一步增加了陷入内核的开销，seL4 中也有类似的机制。最后，外设速度越来越快，而现代微内核的外设驱动往往存在于用户态，外设中断被转化为通知信号，需要用户态驱动主动陷入内核来进行接收，这在很大程度上成为了外设驱动的性能瓶颈。

综上所述，以seL4为代表的现代微内核在IPC和系统调用架构的设计上仍然有很大的上下文切换开销，主要体现在以下两个方面：

- 通知机制需要内核转发。
- 系统调用和同步IPC需要频繁的出入内核。

本选题旨在调研现代微内核的IPC发展趋势及性能瓶颈，通过利用用户态中断绕过特权级切换，减少微内核的IPC开销，从而提升微内核系统IPC时延和吞吐率。同时设计和实现异步的IPC框架，提高微内核系统的资源利用率。

1.2 国内外研究现状

现代微内核的大部分IPC优化始于Liedtke[1]提出的L4，L4通过组合系统调用的方式减少不必要的内核陷入开销，通过消息寄存器避免内存拷贝，通过同步IPC减少等待时间和上下文切换的开销，通过快速路径的方式减

少IPC内核路径。

同步IPC虽然在抽象上是最小化的，并且在L4的设计与实现上都很简单，但后续的实践证明了同步IPC强制用户对原本简单的系统进行多线程设计，导致线程同步变得复杂[3]，例如：由于缺乏类似于Unix中select功能，每个中断源都需要一个单独的线程。此外，同步IPC强制非依赖关系的IPC调用以顺序的形式进行，无法充分利用硬件的并行性[3]。

L4-embedded[8]通过在内核中新增非阻塞的通知机制来解决这个问题，其在seL4中被完善为一组二进制信号量的数组，通知对象通过非阻塞操作来发送信号，而接收方可以通过轮询或阻塞等待信号来检查通知字。此外，内核通过该方式将硬件中断传递到用户态，从而方便了用户态驱动的框架的设计与实现。尽管严格来说通知机制不是最小化的（因为它可以通过同步IPC机制模拟实现），但它对于减少用户态开发复杂性、充分利用硬件并发性起着至关重要的作用。

此外，随着访存速度的加快，L4中通过物理的消息寄存器进行零拷贝优化的方案带来的收益逐渐削弱，相反使用物理寄存器导致的平台依赖和编译器优化失效问题极大地限制了系统的性能，因此纷纷被现代微内核以虚拟消息寄存器的方式所替代[3]。

原始L4微内核使用临时内存映射的方式避免长消息的内存拷贝，但由于在内核中引入了缺页异常的可能性，增加了内核行为的复杂性，现代微内核纷纷放弃了长消息的传递，一般通过通知机制和共享内存组合的方式来进行长消息的传递[3]。

上述的研究和优化方案大都聚焦于IPC的内核路径优化，通过减少IPC路径的解码和调度，减少内存拷贝等方式来减少IPC的性能开销，也已经取得了很大的成果，事实上，在其最简单的情况下，同步IPC仅仅是一个上下文的切换，甚至不会改变消息寄存器。然而，随着内核路径开销越来越少，上下文切换的开销变得无法忽略，特别是对于有频繁的短消息传递和频繁的系统调用的应用，用户态-内核态的上下文切换开销已经成为限制系统性能的最大瓶颈。

现代微内核在软件上通过消息寄存器的方式减少上下文切换的大小，从而在一定程度上减少了上下文切换的开销；通过组合Send & Recv系统调用（以及Recv & Reply）减少陷入内核的次数；在硬件上通过利用ASID减少块表的冲刷频率，从而减小页表的切换开销，这些方法虽然都在一定程度上减少了上下文切换的开销，但无法彻底消除。

在微内核外，还有一些工作致力于减少系统调用中的切换开销，主要分为两类。

第一类方法通过将用户态和内核态的功能扁平化来减少内核与用户态的切换开销，如unikerel将所有用户态代码都映射到内核态执行[9, 10, 11]，Userspace Bypass[12]通过动态二进制分析将两个系统调用之间的用户态代码移入内核态执行，从而减少陷入内核的次数，kernel bypass[13, 14]则通过将硬件驱动（传统内核的功能）移入用户态，从而减少上下文的切换。这些方法要么需要特殊的硬件支持，要么难以与微内核的设计理念兼容，因此都只能提供一定的参考价值。

第二类方法则是允许用户空间对多个系统调用请求排队，并仅通过一个系统调用来将他们注册给内核。如FlexSC[15]通过在用户态设计一个用户态线程的运行时，将用户态线程发起的系统调用自动收集，然后陷入内核态进行批量执行。该方法虽然可以有效的减少陷入内核的次数，却需要在内核态引入复杂的异步运行时，这与微内核的最小化原则相悖。

综上所述，上述方法要么无法彻底消除上下文切换的性能瓶颈，要么无法有效地实施到微内核中。随着硬件的不断发展，用户态中断——一种新兴的硬件技术方案逐渐被各个硬件平台（x86、RISC-V）采纳并很好地解决了上述提到的问题。用户态中断通过硬件的方式，在无需陷入内核的情况下，将信号发送给其他用户态程序。该机制仅需在通信注册过程中陷入内核，以分配用于通信的相关硬件资源，后续的通信过程无需内核接

入，很好地避免了用户态和内核态的上下文切换。目前已经在Sapphire Rapids x86[16]处理器上和RISCV的N扩展[17]中有支持。

优化方法	详细分类	实例	缺点
减少内核路径	惰性调度	[1, 8, 18, 19]	上下文切换开销已经成为性能瓶颈
	快速路径	[1, 2, 8, 18, 19]	
	消息寄存器		
减少上下文切换开销	消息寄存器	[1, 2, 8, 18, 19]	无法从根本上消除切换开销
	组合系统调用		
	ASID机制	[2]	
	统一地址空间	[9, 10, 11, 12, 13, 14]	与微内核设计理念相悖，无法有效地实施到微内核中
	批量系统调用	[15]	

2. 研究内容

本选题旨在调研现代微内核的IPC发展趋势及性能瓶颈，利用用户态中断技术，设计和实现无需内核转发的异步IPC方案，减少微内核的IPC开销，从而从提升微内核系统的资源利用率、IPC时延和吞吐率。主要研究内容分为以下两个部分：

- 微内核的通知机制改进：利用用户态中断改进现代微内核的通知机制，旨在减少甚至消除内核的转发，减少上下文的切换开销，从而降低通知响应时延。
- 微内核的异步IPC和异步系统调用框架设计与实现：利用改进后的通知机制以及共享内存，设计新的IPC和系统调用框架，由于线程无需陷入内核等待，因此在用户态设计异步运行时以充分利用空闲CPU资源。

3. 研究方案

Rust近年来在系统编程方面[20, 21]拥有较高的抽象能力，能够更容易写出高内聚低耦合的代码，对形式化验证[22]会比较有利，重要的是，Rust有着强大的异步编程的支持，能够更加容易地写出内存安全的异步运行时代码，因此本选题使用Rust语言从零构建一个完全兼容seL4内核基本功能的微内核ReL4。

3.1 支持用户态中断的通知机制

新的通知机制的设计分为注册和通信两个部分，为了兼容之前的接口，我们将在notification内核对象的基础上进行扩展。

注册

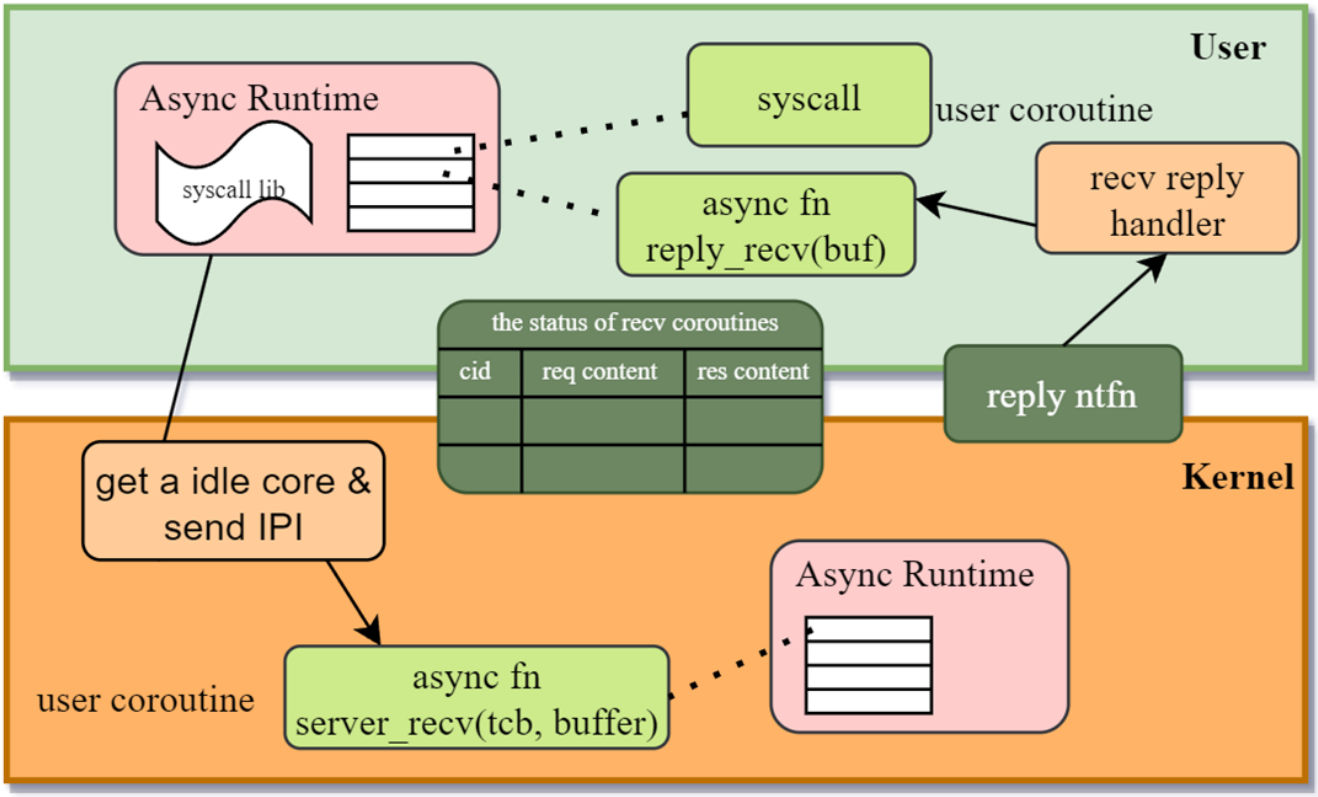
- 创建一个notification对象：保留notification对象的创建接口，在内核实现中扩展内核对象的字段，分配用户态中断的资源并保存在notification中。

- 绑定接收线程：依然使用seL4_TCB_BindNotification保持不变即可，同时在接收线程tcb中设置用户态中断处理程序的入口地址。
- 赋予发送端唯一标识：依然通过 seL4_CNode_Mint 接口，将发送端的中断号作为唯一标识保存到capability中。
- 将IPCbuffer中新增一个标志 uisender_flag, 用于保存当前线程的用户态中断发送端口的注册情况，在调用Signal时，先检查该标志，如果已经注册，则调用 send_uipi 发送信号，如果没有，则陷入内核态进行注册后在调用 send_uipi。

通信

- 发送端：通过notification调用Signal，将从IPCbuffer中读取对应标志位判断当前线程是否注册用户态中断，如果没有注册将陷入内核态进行注册，否则无需陷入内核即可发送用户态中断来实现通知。
- 接收端：对于开启用户态中断的系统，并不需要主动调用Recv，而需要开发对应的对应的用户态中断处理程序。为了兼容此前的接口，对于 seL4_Wait 和 seL4_Poll 有不同的修改：
  - Wait：被转化为异步实现，不会将线程阻塞在内核态，而是切换用户态线程去执行其他操作。
  - Poll：循环读取用户态中断寄存器，无需陷入内核。

3.2 异步系统调用设计方案

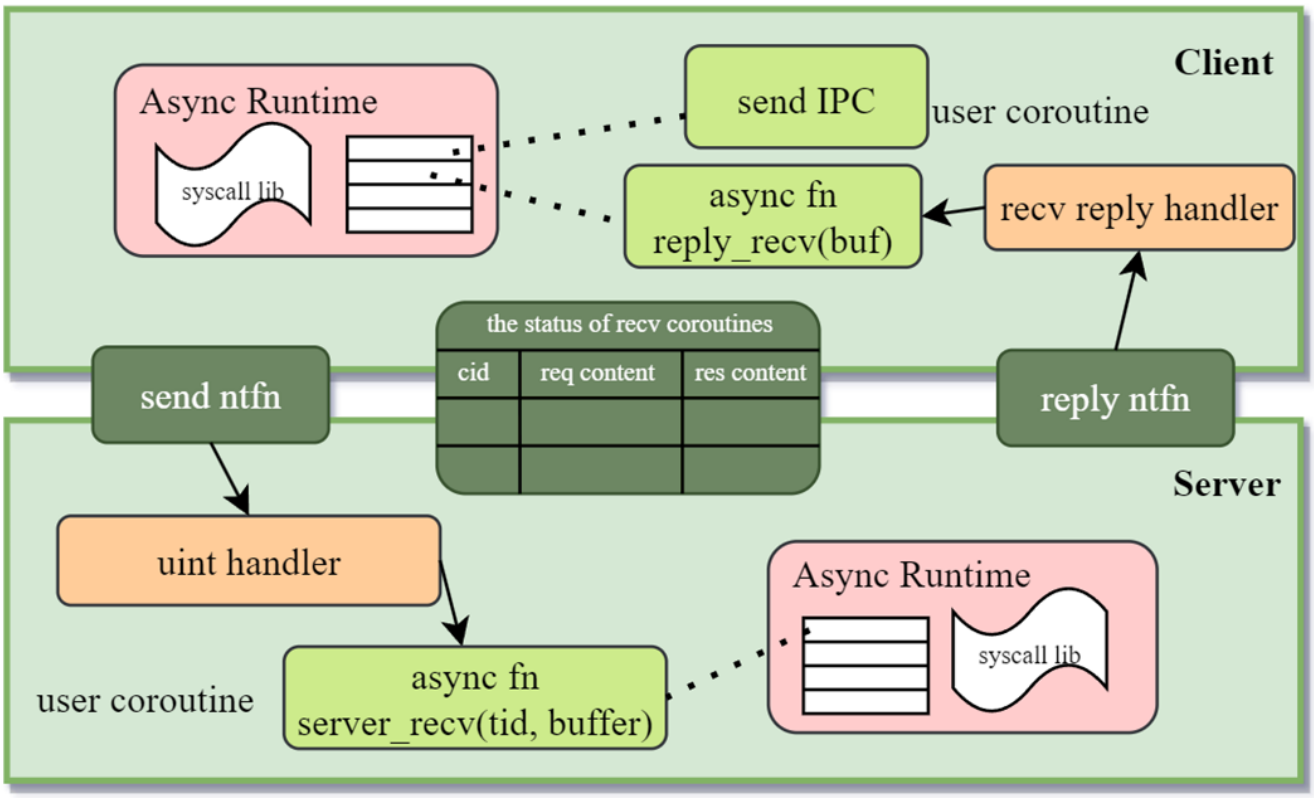


异步系统调用架构分为用户态和内核态两个部分。用户态提供的异步协程运行时允许用户态在等待系统调用结果时可以执行其他协程来减少CPU等待时间，实现分时并发；内核态提供的接受请求的协程则允许内核使用其他CPU核心来出处理系统调用请求，实现同时并行。

- 用户态运行时：在用户态运行时库中提供新的系统调用接口，待运行时初始化完成后，所有的用户态执行流都被封装到一个协程中，在协程中调用的系统调用会链接到新提供的运行时库中，将请求提交到syscall buffer，然后切换到其他协程进行执行。
- 内核运行时：内核启动后会维护一个内核运行时，在刚启动时为空，当有线程注册了异步系统调用的运行时之后，会新建一个Recv协程，添加到内核运行时，用于处理该线程的异步系统调用请求。

- 注册：用户态准备一块共享内存用于存放系统调用请求参数和请求结果等数据。然后调用 `register_async_syscall` 系统调用，在内核运行时中新建一个内核协程，该协程会遍历共享内存中的请求，当没有请求时会挂起该协程。（一个线程对应一个syscall buffer，对应一个内核协程）。
- 提交异步系统调用请求：用户态运行时在提交系统调用请求时会检查当前线程对应的内核协程是否已经被唤醒（syscall buffer 中维护一个用户态可读的原子变量来获取内核协程状态）。
    - 如果对应的内核协程是被唤醒的，则将请求写入buffer后直接切换用户协程。
    - 如果对应的内核协程没有被唤醒，则陷入内核态，将对应的内核协程唤醒，并检查所有的核心是否有空闲的核心在：
        - 如果有：则向空闲核心发送IPI中断，让其执行内核协程。
        - 如果没有（所有的核心都在忙于处理用户态程序）：则直接返回到用户态切换用户协程，等到某个核心的时钟中断到来时再处理内核协程。
- 提交异步系统调用结果：内核协程在执行时能够很轻松的知道当前请求发起者（tcb，cid），以及tcb对应的syscall buffer，因此在处理完请求之后将对应的结果写入syscall res条目中，并根据tcb绑定的notification对象通知用户态中断处理程序，唤醒对应的用户态协程。

3.3 异步IPC设计方案



异步IPC和异步系统调用类似，唯一的两点区别在于注册和提交IPC请求时：

- 注册：客户端除了需要将ipc buffer注册给服务端之外，还需要注册一个notification的发送端，用于客户端唤醒对应的服务端接收协程。
- 提交IPC请求：同样会在ipc buffer中维护接收协程的运行状态：
    - 如果是唤醒的：直接将请求写入ipc buffer后切换用户协程即可。
    - 如果不是唤醒的：通过notification发送用户态中断，由用户态中断处理程序来唤醒对应的服务端协程。

3.4 实验方案

baseline: seL4

裸机：qemu，FPGA。

1. 通知机制的改进后性能：
    1. micro benchmark：内核陷入、上下文切换的周期 vs 用户态中断的上下文切换周期数。
    2. macro benchmark：用户态的网络协议栈通过notification机制通知用户程序数据准备完成，或者各种事件驱动的现实应用，如redis，测量延时和吞吐量，或串口驱动等。
2. 异步IPC的性能：
    1. micro benchmark：每个IPC的切换周期数 vs 批量IPC下的平均IPC的切换周期数。
    2. macro benchmark：选择需要大量IPC调用的应用：如Redis数据库读写等，测量平均延时和吞吐量。

4. 研究工作进度安排

2023年10月～2024年1月：文献阅读。

2024年1月～2024年2月：硕士开题，方案设计。

2024年3月～2024年5月：原型系统设计与实现。

2024年6月～2024年8月：实验和测试。

2024年9月～2024年10月：中期报告。

2024年11月～2025年1月：毕业论文撰写。

2025年2月～2025年6月：毕业论文完善和毕业答辩。

5. 预期研究成果

预期CCF B的期刊或会议论文一篇。

6. 本课题创新之处

1. 在微内核中设计了完全异步的IPC和系统调用框架。
2. 利用用户态中断机制消除微内核IPC的上下文切换开销。

参考文献

[1] Liedtke J. Improving IPC by kernel design[C]//Proceedings of the fourteenth ACM symposium on Operating systems principles. 1993: 175-188.

[2] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 207-220.

[3] Heiser G, Elphinstone K. L4 microkernels: The lessons from 20 years of research and deployment[J]. ACM Transactions on Computer Systems (TOCS), 2016, 34(1): 1-29.

[4] Jeff Caruso. 1 million IOPS demonstrated. https://www.networkworld.com/article/2244085/1-million-iops-demonstrated.html. Accessed: 2021-12-01.

[5] Lipp M, Schwarz M, Gruss D, et al. Meltdown[J]. arXiv preprint arXiv:1801.01207, 2018.

[6] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution[J]. Communications of the ACM, 2020, 63(7): 93-101.

[7] The kernel development community. Page table isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html. Accessed: 2021-12-01

[8] van Schaik C, Leslie B, Dannowski U, et al. NICTA L4-embedded kernel reference manual, version NICTA N1[R]. Technical report, National ICT Australia, October 2005. Latest version available from: http://www. ertos. nicta. com. au/research/l4.

[9] Kuo H C, Williams D, Koller R, et al. A linux in unikernel clothing[C]//Proceedings of the Fifteenth European Conference on Computer Systems. 2020: 1-15.

[10] Olivier P, Chiba D, Lankes S, et al. A binary-compatible unikernel[C]//Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2019: 59-73.

[11] Yu K, Zhang C, Zhao Y. Web Service Appliance Based on Unikernel[C]//2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017: 280-282.

[12] Zhou Z, Bi Y, Wan J, et al. Userspace Bypass: Accelerating Syscall-intensive Applications[C]//17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023: 33-49.

[13] Jeong E Y, Wood S, Jamshed M, et al. {mTCP}: a Highly Scalable User-level {TCP} Stack for Multicore Systems[C]//11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). 2014: 489-502.

[14] Yang Z, Harris J R, Walker B, et al. SPDK: A development kit to build high performance storage applications[C]//2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017: 154-161.

[15] Soares L, Stumm M. {FlexSC}: Flexible system call scheduling with {Exception-Less} system calls[C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). 2010.

[16] Nassif N, Munch A O, Molnar C L, et al. Sapphire rapids: The next-generation Intel Xeon scalable processor[C]//2022 IEEE International Solid-State Circuits Conference (ISSCC). IEEE, 2022, 65: 44-46.

[17] Waterman A, Asanovic K. The RISC-V instruction set manual, volume II: Privileged architecture[J]. RISC-V Foundation, 2019: 1-4.

[18] Heiser G, Leslie B. The OKL4 Microvisor: Convergence point of microkernels and hypervisors[C]//Proceedings of the first ACM asia-pacific workshop on Workshop on systems. 2010: 19-24.

[19] Smejkal T, Lackorzynski A, Engel B, et al. Transactional ipc in fiasco. oc[J]. OSPERT 2015, 2015: 19.

[20] Levy A, Andersen M P, Campbell B, et al. Ownership is theft: Experiences building an embedded OS in Rust[C]//Proceedings of the 8th Workshop on Programming Languages and Operating Systems. 2015: 21-26.

[21] Balasubramanian A, Baranowski M S, Burtsev A, et al. System programming in rust: Beyond safety[C]//Proceedings of the 16th workshop on hot topics in operating systems. 2017: 156-161.

[22] Reed E. Patina: A formalization of the Rust programming language[J]. University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02, 2015, 264.