Abstract

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，同时辅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，因此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开销。

本文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。同时在内核中提供异步系统调用接口，在用户态运行时自动收集异步的系统调用，批量地注册到内核中，进一步减少系统的上下文切换开销，并提升用户态的并发粒度。

1. Introduction

微内核从被提出以来，最大的性能瓶颈就是IPC，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notification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方式和异步notification机制都暴露出一些问题。

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，这要求操作系统能够以快速高效的形式处理大量系统调用[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，从而面临比宏内核更严峻的挑战。此外，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。

减少系统调用开销的方法主要有两类，第一类方法通过将触发系统调用的线程部分指令段代理给内核态[8, 9, 10, 11]，或者将内核的部分功能代码映射到和用户程序同一个地址空间[12, 13]，来减少内核与用户态的切换开销；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一起发出[14]。而在微内核中，大部分宏内核的系统调用代码都通过IPC的形式代理了另一个用户态线程，因此第一类方法很难应用于微内核。

而对于异步信号处理的评判标准则主要是响应时延以及CPU资源利用率。当前业界主流的处理异步信号的方式有两种：轮询和中断，轮询以牺牲CPU利用率来减少响应时延，而中断则相反。虽然有些研究[15, 16]已经证明了在某些高速设备上使用轮询的延迟和开销会低于中断，但对于大部分低速设备而言，使用中断仍然是更好的选择。而seL4中的通知机制是一种以软件形式实现的、由内核进行转发的异步处理机制，即使对于高速设备而言，使用轮询的优势已经被陷入内核的开销所大幅削减，因此中断将会是更好的异步处理形式。

综上所述，现代微内核以同步IPC的方式简化内核路径，通过各种方法提升单个IPC的速度，却也因为同步的形式限制了微内核系统的整体性能，本文将从用户态的角度，通过异步的形式减少用户态发起IPC的次数，同时辅助硬件支持通知机制，减少用户态和内核态的切换开销，提升整个系统的性能。

2. Background and Motivation

当前seL4系统的异步机制（包括用户态的中断处理）全部依靠notification保证[3]，接收端需要在难以把握的时机主动询问内核，过早的时机会导致线程阻塞等待，过晚的时机则会造成信号处理不及时的问题。此外，发送方和接收方都需要主动陷入内核来发送/接收信号，增加了上下文切换的开销和响应延迟（interrupt delivery），这在某些平台上会占据相当一部分比重[17]。seL4用户态提供的 `seL4_Wait` 来阻塞当前线程，将CPU分配给其他线程执行，但这将会导致信号到达后需要等待CPU切换到目标线程之后再进行处理，增加了响

应时延，而提供的 `seL4_Poll` 系统调用则使用 loop 循环来忙碌等待信号的到来，在降低时延的同时也降低的CPU的利用率。

这些问题本质上都是由于接收线程需要主动询问内核（Poll）而导致的，因此本文采用抢占式的中断形式来彻底解决这个问题，利用CPU硬件的用户态中断[18]技术，使得信号不通过内核转发，抢占式地打断接收线程的执行，从而达到减少上下文切换的同时提升CPU利用率的目的。

当前的seL4内核中的系统调用都是同步的，这是由于seL4的系统调用都十分简洁且不会在内核中浪费过多时间[19]，设计是合理的。但从整个操作系统来看，大部分服务被转移到用户态，通过同步IPC来实现类似于宏内核中系统调用功能，而对于某些原本在Linux需要大量系统调用的应用，比如某些数据库系统，上下文切换导致的直接开销和间接开销都将在微内核中放大[20, 21]。更严重的是，客户端可能需要请求多个服务端的服务，这些服务没有依赖关系，同步的IPC将导致这些请求以线性的顺序被处理，无法利用多核的性能[3]。

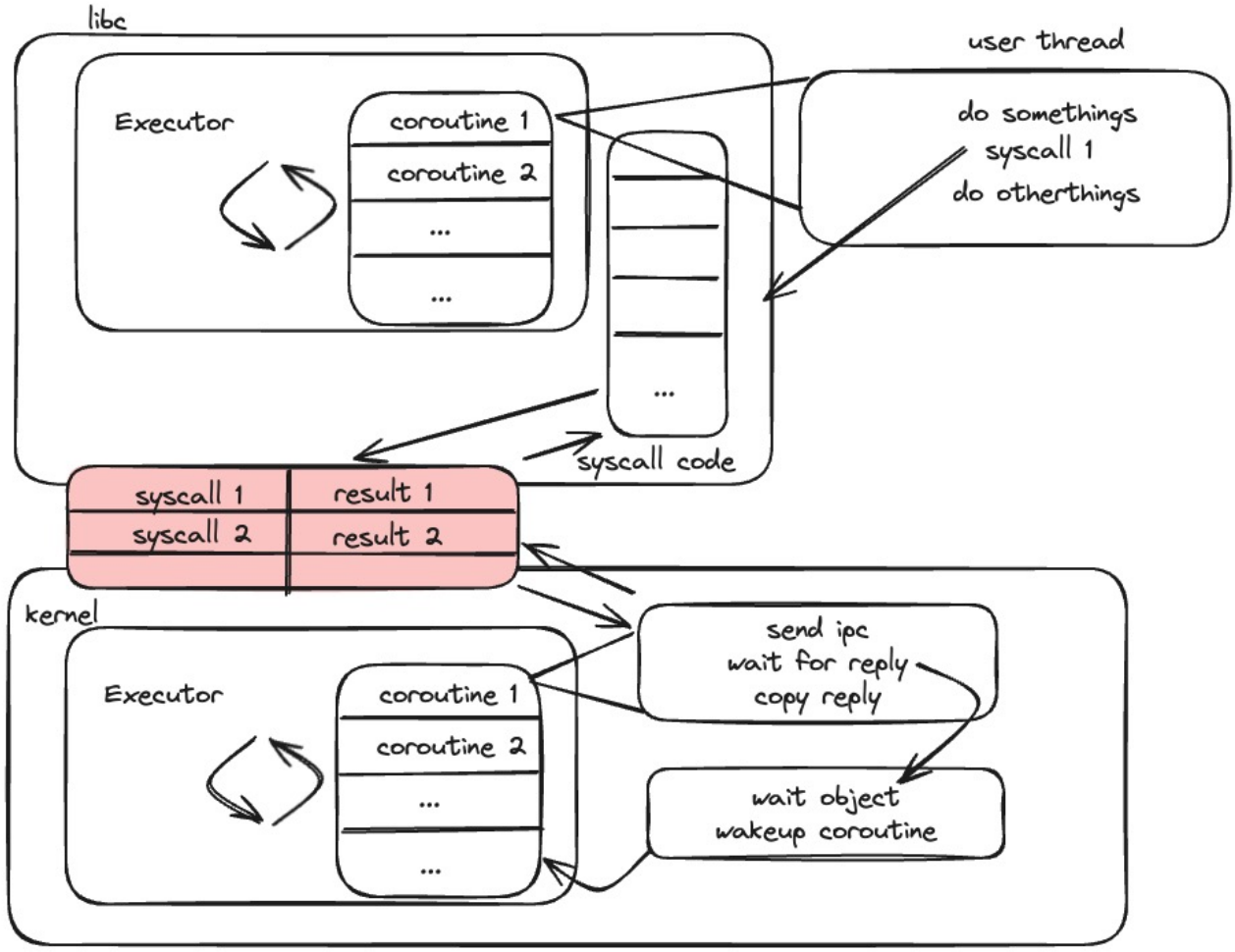
这些问题本质上都是由于IPC的同步性导致的，因此本文将在内核中提供异步运行时，允许用户批量提交系统调用和IPC请求并返回响应，以减少上下文切换并提高用户态线程的并发粒度，同时在用户态提供自动收集异步系统调用的运行时，允许原有的用户态程序以最小的更改来获得异步系统调用带来的好处。

3. Design and Implementation

当前的seL4内核采用c语言进行实现，虽然为了形式化验证的考虑只使用了c的子集[2]，但由于c语言的抽象能力较弱，模块之间划分不够清晰，导致模块化分析困难。更因为c语言的表达能力较弱，对异步编程的支持较弱，因此在原先的seL4中实现异步运行时比较困难，而Rust近年来在系统编程[22, 23]方面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[24]的异步运行时代码，因此本文使用Rust语言从零构建了一个完全兼容seL4内核基本功能的微内核ReL4。

3.1 异步IPC

异步IPC的目标是将无依赖的IPC在用户态收集起来，统一提交到内核态，减少内核陷入次数，充分利用多核性能。因此我们将从用户态-内核交互、内核态异步状态管理、兼容性性设计三个方面进行介绍。



用户态-内核交互

由于用户态提交的IPC请求并不会立刻提交到内核，因此需要内存缓存将请求缓存起来，等待收集完毕后一同提交到内核中，对IPC结果也是如此，因此我们使用共享内存来设计两个缓存队列（Req Queue， Res Queue）。用户态将请求写入Req Queue后返回执行其他代码，等到请求全部写入之后通过 `seL4_AsyncSyscall` 系统调用陷入内核，然后由内核读取Req Queue，（因此Req queue是用户态可写，内核态可读的权限），由于整个过程是同步的，因此不需要额外的同步措施来保证并发安全。相似的，内核将所有请求的执行结果写入 Res Queue后返回内核态，由用户态读取Res Queue 的执行结果（因此Req queue是用户态可读，内核态可写的权限）。

内核态异步状态管理

由于Rust的无栈协程需要将异步的状态保存到额外的内存中，而seL4的内存设计原则之一则是内核不持有额外的内存，所有的内存由用户态显示分配，因此我们在内核中新增 `Excutor` 的内核对象，在用户态通过capability机制进行访问控制，而在内核态则用于管理IPC的异步状态。用户态调用`seL4_AsyncSyscall`后陷入内核态便利task_queue, 根据每个请求的操作符新建不同的协程任务加入执行队列，然后开始执行协程，直到所有的协程全部执行完成，该线程才取消阻塞状态。

```
fn batch_ipc(tcb) {
    while task = tcb.ready_queue.fetch() {
        match task.execute() {
            Ready => {
                // 请求执行完成，写 RQ 后删除任务
```

```
                write_res_queue()
                delete_task()
            }
            _ => {}
        }
    }
    if tcb.task_queue.empty() {
        // 所有异步IPC都完成，唤醒当前线程
        add_sched_queue(tcb)
    }
}
```

对于需要等待事件的协程，会将该协程对应的 `Waker` 加入到对应的内核对象中（如Endpoint和Notification）。当等待事件到来时可以通过内核对象中的Waker来唤醒对应的协程。以 `seL4_Call` 为例，异步任务的伪代码如下：

```
async fn send_ipc(endpoint) {
    loop {
        match endpoint.state {
            Idle | Send => {
                add_waker_to_endpoint()
                await
            }
            Recv => {
                copy_regiser_to_recv()
                add_waker_to_reply_ep()
                await
                copy_reply_to_self()
                break
            }
        }
    }
}
```

兼容性设计

为了能够让用户态程序改动尽量少的代码就能获得异步IPC的性能提升，我们在用户态设计了一个异步的运行时静态库，同时携带了对应的异步系统调用的实现，用户程序只需要添加少量的注册流程，链接异步运行时库即可。静态库提供用户态线程的创建接口，在用户态线程中，调用的系统调用将被链接给重写的静态库。静态库会将请求写入请求队列中，然后切换用户态线程，等当前调度器所有的用户态线程都无法执行了之后，调度器会发起批量异步系统调用`seL4_AsyncSyscall`陷入内核。

3.2 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。

Reference

[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] 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.

[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] 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.

[13] 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.

[14] 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.

[15] Yang J, Minturn D B, Hady F. When poll is better than interrupt[C]//FAST. 2012, 12: 3-3.

[16] Harris B, Altiparmak N. When poll is more energy efficient than interrupt[C]//Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. 2022: 59-64.

[17] Elphinstone K, Zarrabi A, Mcleod K, et al. A performance evaluation of rump kernels as a multi-server os building block on sel4[C]//Proceedings of the 8th Asia-Pacific Workshop on Systems. 2017: 1-8.

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

[19] Heiser G. The seL4 Microkernel–An Introduction[J]. The seL4 Foundation, 2020, 1.

[20] 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.

[21] Klimiankou Y. Micro-CLK: returning to the asynchronicity with communication-less microkernel[C]//Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems. 2021: 106-114.

[22] 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.

[23] 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.

[24] 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.