清 华 大 学

综 合 论 文 训 练

题目：reL4 微内核的设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：李龙昊

指 导 教 师：王生原　副教授

联合指导教师：张福新　正研级高级工程师

2023 年 6 月 5 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

本文介绍了 reL4 微内核的开发过程，reL4 是参照 seL4 微内核的功能模块，使用 rust 语言重新实现的一款微内核。

seL4 内核注重安全性与高效性，它采用严格的内存保护机制，保证了组件之间的隔离性，同时经过了全面的形式化验证，证明了内核实现的正确性。当前的 seL4 内核由 c 语言编写。c 语言灵活高效，但容易产生内存安全与线程安全方面的问题，并不适合 seL4 这款注重安全性的内核。从这一角度出发，本文使用 rust 这门保证内存安全、线程安全的语言，实现与 seL4 有相同功能的 reL4，在语言角度和内核角度两方面确保系统的稳定性与可靠性。

在经历了几个多月的开发之后，reL4 已经实现了 seL4 的全部功能，并通过了 seL4 自带的测例 seL4Test, 证明了 reL4 对 seL4 的可替代性。

关键词：seL4；Rust 语言；微内核

ABSTRACT

This paper describes the development of the reL4 microkernel.reL4 is a microkernel based on the functional modules of the seL4 microkernel,

The security and efficiency are strongly emphasized in seL4 kernel. It adopted a strict memory proctection mechanism to ensure isolation between componets. Besides , it has passed comprehensive formal verification , which proved the correctness of the kernel implementation. Currently , seL4 kernel is written in c language,which is a flexible and efficient language. However, c language has flaws can influence memory and threads safety,making it unsuitable for the security oriented seL4 kernel. From this perspective ,this paper proposes a new idea that implements a kernel called reL4 which has same function as seL4,but using rust language , which guarantees memory and threads safety.

After few months of development, reL4 has implemented all the functions of seL4 and passed teh seL4Test whichi is a built-in testcased of seL4,proving the substitutability of reL4 for seL4.

Keywords: seL4; Rust; Microkernel

目　录

主要符号表

IPC	进程间通信 (Inter-Process Communication)
TCB	线程控制块 (Thread Control Block)
CSpace	能力空间 (Capability Space)
VSpace	虚拟地址空间 (Virtual Space)
POLP	最小权限的安全原则 (Principle of Least Privilage)
Capability	权限令牌

第 1 章　背景与相关工作

seL4 是一款高安全性、高性能的操作系统微内核。作为 L4 微内核家族的一员，它解决了传统微内核进程间通信效率低的致命弱点，同时由于采用了权限令牌 (capability) 的设计特点，它的安全性也有所保证。2011 年，seL4 内核被证明了能够保证数据的完整性，即在没有权限令牌授权的情况下不能修改数据，同年证明了 seL4 是一个混合临界系统（mixed-criticality-systems），即在同时运行可靠代码和不可靠代码的情况下，可以保证可靠关键代码不受非可信代码的影响，仍能保证可靠代码的硬实时性。2013 年，seL4 通过了全面的形式化验证，证明了 seL4 产生的二进制代码是 seL4 的 C 代码的正确翻译，从而不必信任 C 编译器。当前 seL4 已被应用于高安全性、高可靠性的系统，如：无人机、汽车嵌入式设备和安全通信设备等。

rust 语言是一门由 Mozilla 研究院开发的，面向系统编程的高性能编程语言。它的设计目标是为了提供编程语言的安全性与高效性，同时它也拥有着与 C 语言相同的底层硬件控制能力，这使用 rust 来开发操作系统成为可能。近年来，已有多个以 rust 语言为基础的内核问世，如 Redox OS 等，这些内核的成功也说明了 rust 语言在内核开发领域的可行性与优越性。而用 rust 重新实现 seL4 的初衷也是源于此：用 rust 语言的内存、类型安全性来作为 seL4 安全性的更有利支持。

本章将从 seL4 的内核简介出发，介绍 seL4 的高安全性保障，接着介绍 rust 语言，由此引入 reL4 内核的实现初衷：从语言与内核模块两个角度保障内核的安全性。

1.1　seL4 微内核简介

1.1.1　seL4 设计理念

作为一款典型的微内核，seL4 与主流的操作系统（如 Linux 等）有着很大的不同。

如图1.1所示，左侧 Linux 内核中黄色部分为特权态部分，支持多种多样的服务比如进程间通信、文件系统等，这导致 Linux 内核代码异常庞大，约两千万行源码，里面包含了成千上万的漏洞。可信计算基础（Trusted-computing-base，TCB）是指计算机能够正确运行所依赖的系统子集。Linux 代码量的庞大也意味着可信

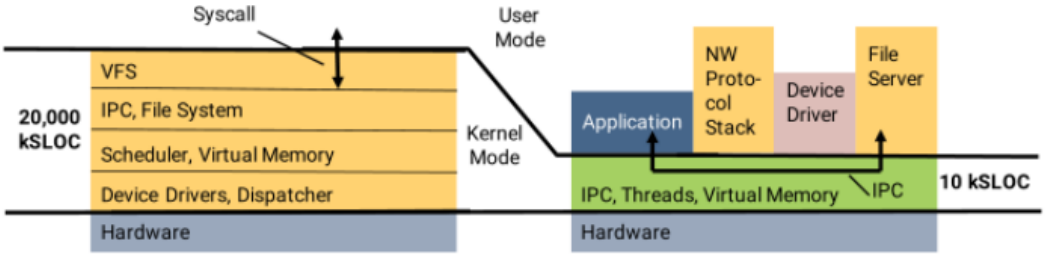

图 1.1　seL4 与 Linux 内核对比图[1]

计算基础的庞大，这说明了 Linux 无法保证自身的绝对安全性，一旦一个漏洞被攻击，就可能导致整个系统的崩溃。

考虑到这个因素，seL4 减少了 TCB 的规模，由此来减少被攻击的概率。图1.1右侧绿色部分为 seL4 的特权态代码，仅有一万行，相较于 linux 代码来说小得多。代码量的减小也意味着 seL4 不可能像 Linux 提供相同规模的服务。实际上，seL4 几乎不提供任何服务，它只是硬件的一个薄包装，但是足以安全地复用硬件资源，关于 seL4 提供的内核服务，我们将在接下来的一个小节中进行简单介绍。除此之外，用户程序所需的所有服务，如文件系统、设备驱动等都运行在用户态。这些服务与普通的用户程序无异，都需要通过进程间通信来向用户程序提供服务。

1.1.2　seL4 设计模式

在简单介绍了 seL4 的设计理念后，我们了解到了 seL4 对自身系统的安全性、通信高效性的重视，在本小节中，我们将围绕这两个原则来介绍 seL4 做了哪些具体的工作。

1.1.2.1　基于权限令牌的访问控制

seL4 提供了一个基于权限令牌（Capability）的访问控制模型。权限令牌是向特定对象传达特定权限的访问令牌。当用户程序想要执行某种特定操作时，就必须调用对所请求的服务具有足够权限的令牌。除了安全性以外，基于权限的访问控制还有两点好处：

1. 细粒度访问控制

   权限令牌可以提供细粒度的访问控制，符合最小权限的安全原则（Principle of Least Privilage, POLP）。

   在 Linux 系统中，用户对文件的操作权限由 ACL 访问控制权限决定，当用户想要运行某个不受信任的程序时，这个程序对当前文件夹下的操作权限

来自于用户权限。倘若用户想要让不受信任程序读取某个文件而不访问其他文件时，这一点在 Linux 中就很难做到。而基于权限的访问控制的机制就从设计上避免了这个问题，权限并不与用户绑定，不受信任的程序只能访问它被授予的权限所对应的文件，而不能访问其他文件。细粒度的访问控制进一步保证了 seL4 系统中数据的保密性。

2. 打桩机制

   权限令牌支持打桩机制。被授予权限令牌的对象并不知道令牌本身对应的到底是什么，他只是去调用对应对象上的接口。继续拿一个修改文件的不受信任程序举例，它被授予了修改文件的权限，通过这个权限绑定的对象的接口，它可以向文件中写入内容，在这个程序看来，自己与文件的读写入口直接相连。但实际上，与他相连的可能是某个安全性的监视器，读取程序对文件的操作是否合法，如果合法才会执行对应的操作。打桩机制提供了一种监视不受信任程序的手段，是对 seL4 数据完整性的补充。

1.1.2.2 seL4 内核服务

在 seL4 微内核中，仅提供了少量的内核服务原语，更复杂的功能用户程序可以通过提供的服务原语自身实现。由于时间以及任务量的关系，我只完成了不包含混合临界系统在内的 seL4 内核的重写工作。在这里就仅介绍不包含混合临界系统在内的 seL4 内核提供的服务。在**不支持混合临界系统（Mixed-Criticality Systems）**时，seL4 仅提供以下基本服务：

1. **Threads**: 线程是支持软件运行的 CPU 执行的抽象。
2. **Address Spaces**：每个用户程序单独一个地址空间，由此实现程序间的隔离。
3. **Inter-process communication(IPC)**: 由 Endpoint 支持进程间的通信，可以在用户程序之间传递消息与权限令牌。
4. **Notifications**:Notifications 提供了一组类似于二进制信号量的信号机制。
5. **Capability Spaces**：隶属于每个线程，记录当前线程所拥有的权限令牌。

1.1.2.3 隔离性设计

在 seL4 的设计中，服务与应用程序一样都运行在用户态的独立的沙箱之中，通过 IPC 来向外界提供调用接口。如果服务遭到攻击，那么攻击只会局限在沙箱之中，导致这个服务的不可用，并不会危害整个系统，与 Linux 内核形成了鲜明的对比。有研究表明，在已知的 Linux 漏洞中，29% 漏洞能被微内核的设计完全消除，还能额外减轻 55% 的安全隐患[2]。

1.2 Rust 简介

本章开头介绍到，rust 语言的目标是在性能与 C 语言不相上下的情况下，提供更高的安全性。为此，rust 语言提供了一整套的措施。它将安全性细分为了三方面：内存安全、类型安全与并发安全。Rust 引入的所有权机制可以有效的保证内存的安全性，而编译期间的检查能够保证同时保证了这三方面的安全性。由于编译期检查的约束，为 rust 语言的开发带来了很多不方便性，rust 语言引入了 unsafe 块来绕开编译期的检查的限制。unsafe 块的引入，为 rust 语言带来了灵活性，但编程者需要自己保证 unsafe 块中代码的安全性。在本节接下来的内容中，我们将介绍 rust 中的所有权机制与 unsafe 块。

1.2.1 所有权机制

所有权机制是 rust 语言管理内存的一种方式。所有程序运行时都会占用内存，内存在程序退出后也要被安全的释放。在其他语言中，这件事或许由垃圾回收机制（garbage collection）机制完成，或许由程序手动释放。而 rust 语言提供了一种新的方案来管理，它在编译期间提供了一些规则性检查，满足规则后编译器会自动插入释放指令释放内存。这种方案被称为所有权机制。所有权机制要求满足的规则为：

1. Rust 中每个值都有一个变量视为这个值的拥有者。
2. 每个值都只能有一个拥有者。
3. 当拥有者不在程序运行范围时，这个值所占内存空间将被释放。

只要满足以上规则，rust 就能够保证在不使用 unsafe 语句的情况下内存的安全性。

1.2.2 unsafe 块

由于底层软件的开发固有的不安全性，如果要求内核的所有代码都能够通过 rust 的编译期的检查，是根本完成不了。举一个最简单的例子，开启页表在 riscv 中需要内联汇编设置 satp 的值，而内联汇编作为 rust ffi（rust Foreign Function Interface）的一部分是不安全的，无法通过编译期的检查。这就需要我们使用“不安全的 rust”，通过引入 unsafe 划分代码块，告诉编译器这部分代码运行逻辑的正确性由开发者保证而无需 rust 保证。同时，值得注意的是，unsafe 给 rust 带来了与 c 相同的灵活性，开发者需要保证非必要情况下不引入 unsafe，以及 unsafe 块内代码的安全性。

1.2.3 包管理工具

抛开安全性与高效性不谈，rust 与 c 语言相比仍有一个巨大的优势，那就是 rust 高效的包管理工具 **cargo**，cargo 之于 rust，正如 maven 之于 java，让 rust 语言开发者能够免去造轮子的烦恼，大大提高开发效率。rust 中想要添加第三方库，只需在 **cargo.toml** 文件中写入即可。而 c 语言中并没有一个专用的包管理工具。要想引入第三方库，我们就只能用 makefile 让文件一起链接，这让 c 语言开发人员的工作效率受到影响。

1.3 Rust Based OS

在完成 sel4test 的 rust 重写工作之前，学术界已经有很多 OS 采用了 rust 开发，他们的项目在我的开发过程中给到了一定的借鉴意义，在本节中，将对其进行介绍。

1.3.1 Redox OS[3]

RedoxOS 是一款 rust 语言开发的类 Unix 的微内核操作系统。注重于高效性、安全性与稳定性。RedoxOS 与 POSIX 有适度的兼容性，这允许 RedoxOS 无需移植就可以运行许多应用程序, 天然的为 RedoxOS 提供了生态环境。在设计理念上，Redox 遵循了微内核的设计方式，在设计过程中参考了 SeL4、Plan 9 和 BSD 等众多微内核。但它又不像一个微内核一样，几乎不提供服务。它提供内存分配、文件管理、显示管理等服务。这些服务都遵循微内核的设计理念，均在用户态运行，这种做法让 RedoxOS 相较于 Linux 而言有着更高的安全性。

此外，RedoxOS 可以支持常见的 Unix 指令，并对 Rust 标准库有着完整的支持，他还有着可选的 GUI 程序——Orbital。当前 Redox 支持 x86 架构的 CPU，能够正确驱动网卡、鼠标、键盘和显卡等设备。

1.3.2 rFreeRTOS[4]

FreeRTOS 是一个迷你的实时操作系统内核，提供了包括：任务管理、信号量、消息队列、内存管理在内的诸多功能。与其他商业内核相比较，FreeRTOS 具有开源、调度灵活、移植性好等优点。

在清华大学操作系统课程中，三位同学完成了 FreeRTOS 的 rust 语言重写工作，实现了 FreeRTOS 中原有的任务调度、双向链表、队列、信号量、事件组等核心功能。并通过了 FreeRTOS 自带的大部分功能性测试。他们成功的将 rFreeRTOS

移植到了 D1-H 开发板上，成功证明了移植的正确性。他们的工作，与我所做的工作类似。于我而言，有着很高的参考借鉴价值。

1.3.3 rCore[5]

rCore 是一款由 rust 语言开发的用于教学的操作系统内核，在整体设计上采用宏内核，具体实现中引入了许多简单易用的库，如 **buddy_system_allocator** 作为堆分配器，大大简化了编写内核的工作量，在较短时间内实现了对 GCC,Nginx 等程序的支持。

在 rust 语言重写 seL4 工作的初期，我大量借鉴了 rCore 中的实现，学习了 rCore 作为操作系统本身是如何与底层硬件配合工作的。虽然最后不曾保留 rCore 的相关代码，但 rCore 仍给我的工作起到了很大的学习指导作用。

1.4 RISC-V 简介

RISC-V 指令集是加州伯克利开发的一款精简指令集（RISC），它全部开源，架构简单，不像 x86 与 arm 架构一样有着众多的历史包袱。RISC-V 指令集支持灵活的模块化设计，存在机器模式（Machine Mode）、特权模式（Supervisor Mode）、用户模式（User Mode）多个模式，支持不同情况下的模式排列组合使用。在生态方面，RISC-V 有着完整的工具链，gcc qemu 等工具已支持 RISC-V 架构的使用。

由于我在先前的计算机组成原理课程中系统学习过 RISC-V，所以为了减少学习成本，我将 qemu 模拟的 RISCV64 虚拟机作为自己的实验平台，来完成 seL4 的 rust 重写工作。

第 2 章　seL4 整体架构

实际上，seL4 内核往往不作为一个内核单独运行，它常常作为 seL4 其他工程项目的一部分参与编译链接。不将 seL4 内核单独运行的原因，我认为有两点：1. 我们需要 seL4 内核，是需要 seL4 的高效性、高安全性为上层用户程序提供服务，单独运行一个内核而不运行用户程序并没有任何意义。2.seL4 内核运行在高位全 1 的虚拟地址上，我们将在3.2节中进行详细介绍，这就要求在这之前有一个工具完成物理地址到虚拟地址的映射。在 seL4 项目中，这件事交给了 elfloader，单独的 seL4 内核无法完成这项工作，也不能正常运行。本章将介绍依托于 seL4 的项目 seL4Test 的项目结构，并对后续的毕设的实现方法做简单概述。

2.1　seL4Test 项目结构

seL4Test 采用 cmake 工具为项目生成 build.ninja 文件，再由 ninja 编译，整个 seL4Test 的项目结构如图 2.1 所示。在图2.1中，kernel 部分为 seL4 内核，而运行

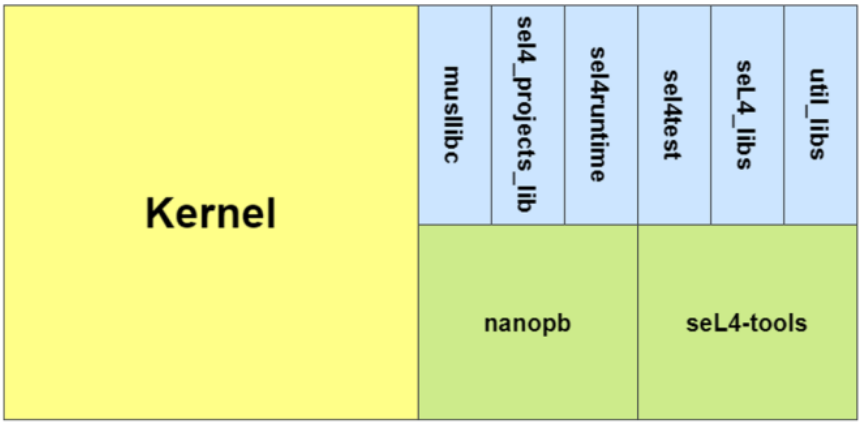

图 2.1　seL4Test 项目结构

seL4test 项目还需要其他的依赖库，部分依赖库的解释说明列在表 2.1 中。

表 2.1　seL4Test 依赖仓库介绍

仓库名	描述
kernel	seL4 微内核源码
sel4test	seL4 用户程序，elf 文件的入口地址
sel4_tools	构建 seL4Test 用的工具，如 cmake-tool elfloader 等
seL4_libs	在 seL4 微内核上编写用户程序的程序库
util_libs	seL4 微内核启动使用的程序库，如设备树库等
seL4_projects_lib	seL4 的库集合，与 seL4_libs 兼容

从图表来看，seL4Test 项目结构复杂, 其中仅用于描述库依赖关系的 cmake 文件就有一万多行，要想在毕设完成的时间内完成整个 seL4Test 项目的 rust 重写工作，不可能也没必要，在接下来的一节里，我将简单介绍自己的重写 seL4 的实现方案。

2.2　实现方案

在相关工作中，我介绍了用于教学的 rCore 操作系统，rCore 作为一个内核架构非常完整，与底层硬件的连接、中断异常、页表、线程上下文、loader 等模块一应俱全。于是在一开始，我决定先将 seL4 的功能性模块以模块化的方法添加进入 rCore 当中，如果有功能重复的模块，就用 seL4 的模块替代 rCore 的模块，完成一个有着 seL4 部分功能的内核。这样做的好处是，借助 rCore 的框架，可以使用 rCore 的测例与编写的简单测例，检测实现的功能的正确性。

2.2.1　seL4 模块化分析

图2.2所示即为 seL4 各个模块之间的依赖关系,A 指向 B 的箭头表示 A 的正确运行依赖 B, 其中 **boot** 与 **rootserver** 两部分涉及到了 seL4 的启动部分，由于 seL4 的启动方式与 rCore 相比有着很大的不同，而且更改了启动方式会导致 seL4 内核不能单独运行，所以这两部分我们不做移植。剩余大致可以分成几个部分：中断异常 (Trap&Syscall)、调度模块 (Scheduler)、权限令牌模块 (Capability Space,CSpace)、虚拟地址空间模块 (VSpace) 与内核对象模块 (Object)。从图中可以看到权限模块作为管理其他模块的管理模块，可以做到被其他模块依赖而不依赖其他模块。虚拟地址空间模块与内核对象模块仅依赖权限模块，而中断异常、调度模块又向后

依赖着内核对象模块。所以，我们可以将模块化 seL4 的工作分为三部分:

- 权限模块的实现
- 虚拟地址空间、内核对象的实现
- 中断异常、调度的实现

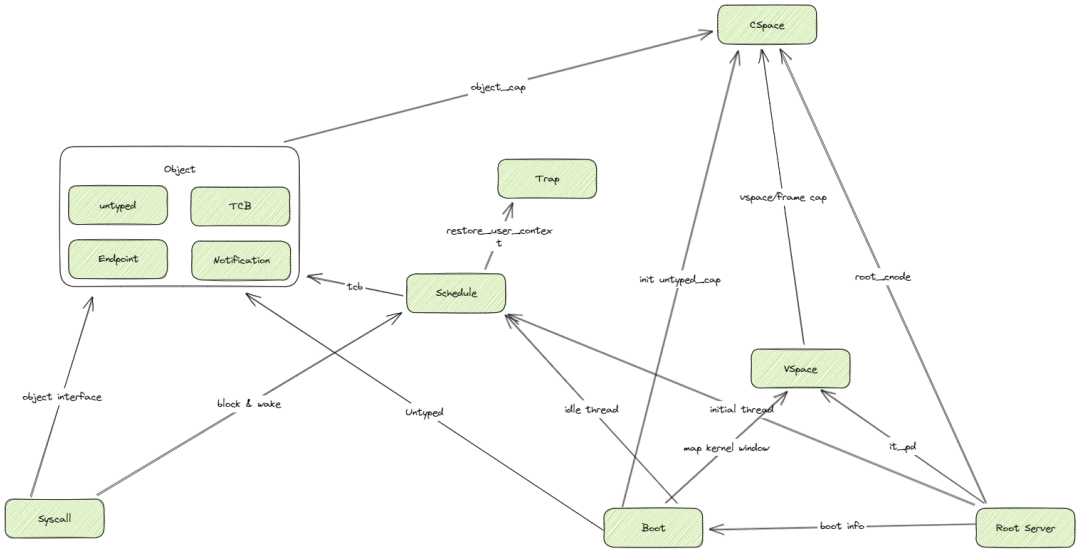

图 2.2　seL4 模块间依赖关系

2.2.2　完整 seL4 内核的实现

当完成 seL4 功能在 rCore 上的移植后，后续的实现方向有两种选择:

1. 以 rCore 为基础，继续用 rust 实现 seL4Test 部分测例的翻译，然后再运行来验证自身完成功能的正确性。

   看到此可能会有读者产生疑问：用户程序与内核的交互是通过系统调用，理论上来说只要将 seL4Test 当做用户程序加载进入 seL4 内核，再通过系统调用让 seL4 的内核提供功能不就可以正常工作了吗？为什么要用 rust 重写测例？原因在于，上文介绍到，在 seL4 中有两部分没有实现：**rootserver** 与 **boot**, seL4Test 的运行是非常依赖 boot 阶段的，因为 boot 阶段会计算自身可用的内存空间，将这部分信息以 bootinfo 结构体的方式传递给 seL4Test 程序供其使用。此外 seL4 的地址空间设计和 elf 文件加载方式都与 rCore 有着很大的不同，如果直接加载运行 seL4Test 不确定性因素太多，可行性不高。所以，作为一种稳妥的方法，翻译 seL4Test 中的测例虽然工程量大，但能够保证能够正确运行并验证功能的正确性。
2. 继续修改 rCore，不仅从功能上完成 seL4 的功能，还要在运行逻辑上同 seL4

相同，然后将 rust 代码编译成为静态链接库，再通过添加 cmake 第三方依赖库的方法，替代 seL4Test 项目中的 kernel 部分去进行 seL4Test 的测例测试工作，此时，我们完成的工作就是一个用 rust 重写的 seL4。

考虑了实现难度、意义两方面的因素，我最终选择了第二种方法作为下一步的计划。从实现难度来说，第二种方法的难度低于第一种，而从工作意义来说，第二种方法的可拓展性与实际意义也更优于第一种方法：reL4 在完成当前的工作后，可以考虑完成形式化验证的工作。让 rust 语言的安全性、高效性在 reL4 中充分发挥作用。同时，后续还可以尝试移植 seL4Bench 项目来比较 rust 语言与 c 语言的效率，评判 rust 语言的来编写内核的优与劣。

第 3 章　设计与实现

第二章主要从整体架构的角度讲述了 reL4 重新实现的方案。本章将从具体部分出发，讲述 reL4 实现过程中遇到的问题与解决方案。从目录来看，本章节的小节顺序似乎很乱，启动阶段放到了3.4，而部分模块的具体实现细节放到了启动阶段之前，很不符合内核的运行逻辑。事实上，笔者作这样的章节划分，是考虑了各个模块之间的依赖关系，从不被依赖的模块讲起，由浅入深介绍整个内核的基本运行逻辑，再介绍内核为上层应用提供的服务（如进程间通信），才能将整个内核讲明白。

3.1　权限令牌机制实现

权限令牌模块是 seL4 中的管理模块，其余模块几乎均依赖权限令牌提供的服务，所以在实现中，权限模块是第一个要被实现的模块。

3.1.1　capability 具体实现

在 seL4 中权限令牌负责权限审查工作。在实现上，令牌是一个十六字节的数组，由于管理的功能模块不同，capability 每一位对应的信息也不相同。表3.1为存放 capability 的内核对象 cnode 对应的 capability，其中存储的信息包括 cnode 结构的大小，保护位的位数、保护位的值等信息。

表 3.1　cnode capability 结构

untyped cap 部分：位数	**描述**
capFreeIndex:39	用于计算可用内存块大小
capIsDevice:1	是否为设备内存，设备内存只能当做页表页使用
capBlockSize:6	用于计算可用内存块大小, 计算方法为 2^BlockSize+FreeIndex
capPtr:39	指向可用内存起始地址的指针

cnode cap 只是一种 capability，在实现中，每一个内核对象都有一个对象的 capability，如表3.2所示。篇幅原因不再具体展示每个 capability 内部各位对应信息，如果接下来实现中用到具体 capability 再做具体介绍。

表 3.2　capability 类型

capability 类型	描述
cap_untyped_cap	管理内存地址空间
cap_endpoint_cap	管理内核进程间通信中的 endpoint 通信方式
cap_notification_cap	管理内核进程间通信中的 notification 方式
cap_cnode_cap	管理存放 capability 的空间
cap_thread_cap	管理线程控制块
cap_pagetable_cap	管理根页表
cap_frame_cap	管理次级页表
cap_asid_control_cap	管理地址空间标识符记录

每一个线程都有一个绑定的存放 capability 的结构，被称作 CSpace,CSpace 由一个或多个 cnode 内核对象组成，cnode 则是 Capability table entry（cte）的数组，实现中，cte 结构如代码1所示，其中存放着一个 capability 对象以及一个双向链表的节点，其中存放着指向前一个和后一个 cte 的指针。双向链表维护的，是 capability 的派生关系，Capability 不能凭空构建，只能由初始线程派生出新的 capability 插入其他线程的 CSpace 中，这时就会在初始线程和其他线程间建立双向链表，记录 capability 的派生关系，当初始线程的 capability 被删除时，派生出的 capability 也要一起被删除。

```
#[repr(C)]
#[derive(Debug, PartialEq, Clone, Copy)]
pub struct cte_t {
    pub cap: cap_t,
    pub cteMDBNode: mdb_node_t,
}
```

代码 1　cte 结构

在 reL4 中，由于线程被删除可能导致链表中的一个节点消失，所以，需要一组函数来维护这张双向链表，于是就有了一组用于管理双向链表的 invocation（关于 invocation 的信息可参考3.5.3 ），简介位于表3.3中。

表 3.3　capability 模块提供的 invocation

invocation	描述
CNodeRevoke CNodeDelete	撤销 删除当前的 capability
CNodeCancelBadgedSends	删除进程间通信所带的标记（badge）
CNodeMove	将当前的 capability 从一个 cte 槽移到另一个 cte 槽
CNodeRotate	同时修改两个 capability 的 cte 槽，放入新的 cte 槽中
CNodeMutate	更新当前的 capability，并将它放入一个新的 cte 槽
CNodeMint	由当前 capability 派生出一个新的 capability 并插入目标 cte 槽中

3.1.2　capability 的查询

用户想要执行某种操作就要提供对应的 capability，而如何寻找对应的 capability 将会成为一个问题，根据上节实现的双向链表寻找显然不合适，应用程序并不清楚不在自己 CSpace 空间中的 capability 所在位置，而且根据链表由前至后的搜寻复杂度高，会降低微内核的性能。于是，综合用户程序可见的权限以及微内核的性能两种因素，reL4 实现了基于用户 CSpace 空间的查询方式。应用程序只需给出对应的槽号，内核在 CSpace 中搜寻，找到对应槽号的 capability 去执行操作即可。

依照 CSpace 查询的方法支持固定寻址、依据索引寻址两种方式。部分 capability 被所有线程需要，如管理虚拟地址空间的 capability、管理线程控制块的 capability，所以在 CSpace 中，我们将这几个 capability 的 cte 槽号固定下来，需要某种操作的时候，直接去固定的 cte 槽中取出 capability 即可。

依据索引寻址主要针对不是必需的 capability，如 endpoint cap，不是每个线程都需要 endpoint 来进程间通信，这时候存放 endpoint capability 的槽号不固定，就需要依据索引寻址，具体操作如图3.1所示：

1. 内核从寄存器中读取索引 index 与搜索深度 depth。
2. 进入 cnode，比较 cnode 的保护位与 index 中对应位的值，不相等就报错，depth 减去索引位位数。
3. 读取 cnode 大小，从 index 中读取对应的位数，找到对应 cte 槽，depth 减去 cnode 大小位数。
4. 如果 depth 为 0，搜寻结束，返回对应 cte 槽的 capability，如果不为 0，则对应 cte 槽存储的是另一个 cnode 对象的 capability，转到另一个 cnode 中从步骤 2 开始执行。

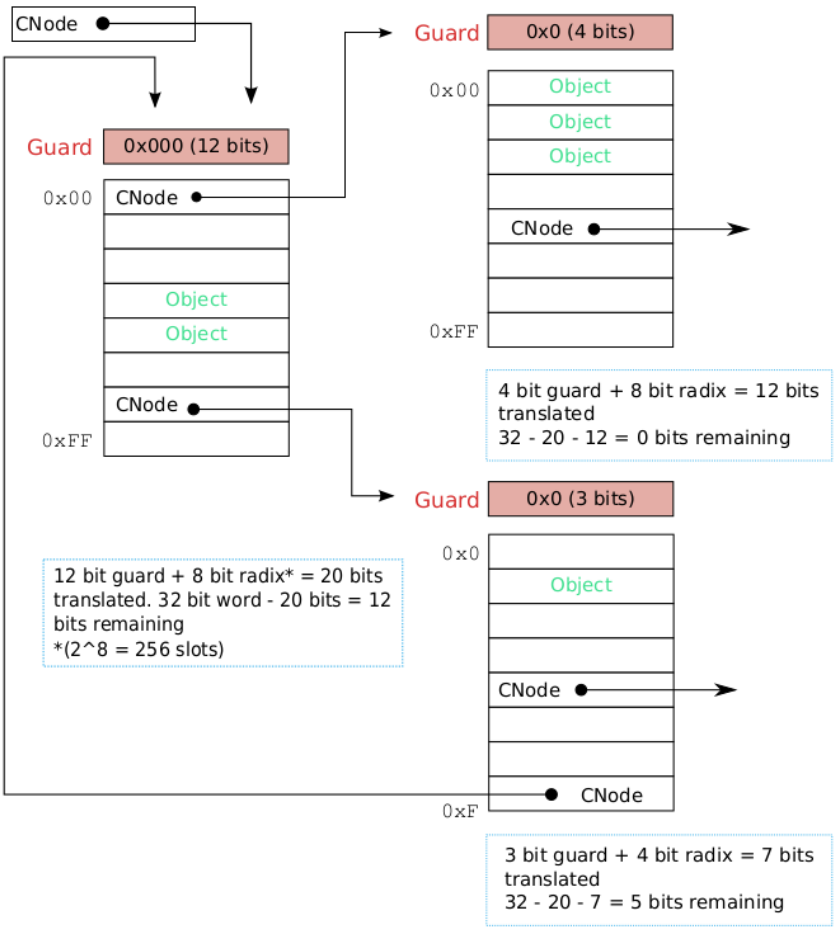

图 3.1　capability 按索引寻址

3.2　地址空间设计

3.2.1　seL4 的地址空间设计

rCore 的地址空间设计与 seL4 相比有很大的不同，rCore 采用了双页表的设计，即用户态与内核态各有一张页表，两张页表都占据了整个地址空间。我认为这样做有两点好处：1. 内核态的代码对于用户程序来说是不可见的，能够保证内核代码的安全性。2. 物理地址设置为虚拟地址的附近代码能够保证平滑，这个问题我们将在接下来的3.2.2节进行介绍。当然这样做也会引入特权级切换时栈指针与页表的同时切换的问题，需要引入跳板机制，由于我们并没有采用这种设计方法，就不展开介绍了。

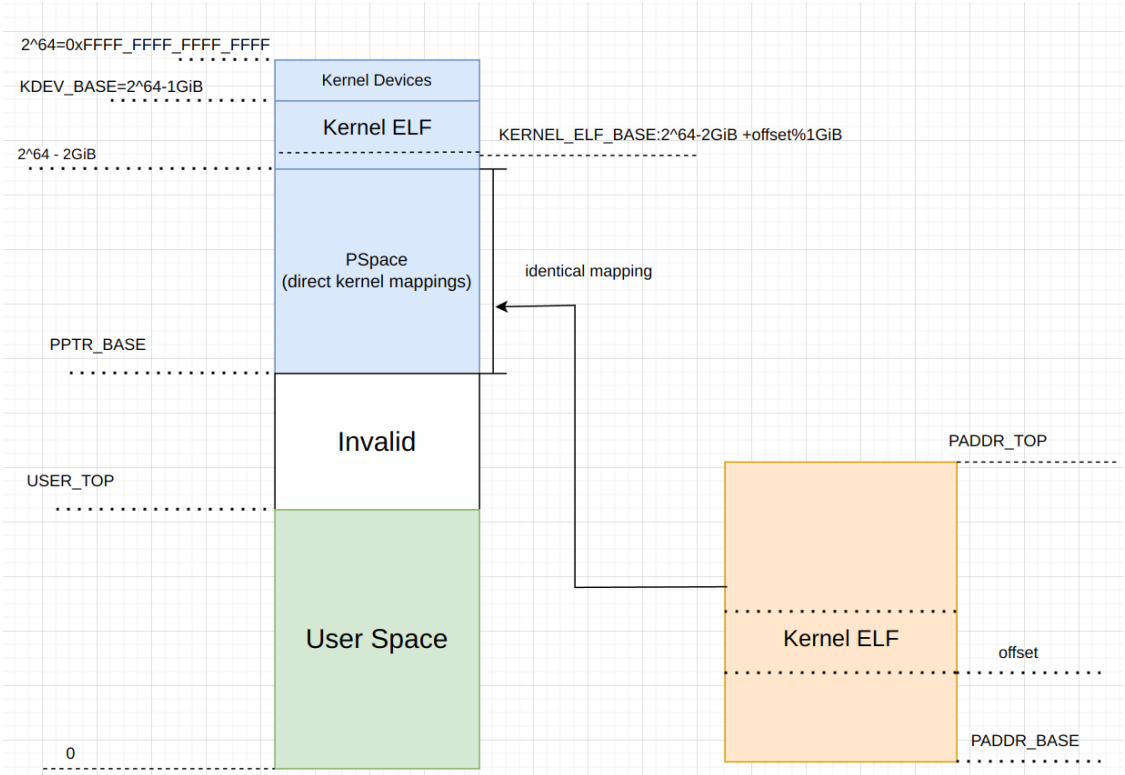

图 3.2　seL4 地址空间设计[6]

表 3.4　seL4 地址空间变量解释，图3.2的补充

变量名	描述
KDEV_BASE	内核驱动加载的起始虚拟地址
KERNEL_ELF_BASE	内核加载的起始虚拟地址
PPTR_BASE	物理地址空间直接映射的起始虚拟地址
USER_TOP	用户地址空间的顶部虚拟地址
PADDR_BASE	起始物理地址
PADDR_TOP	物理地址顶部
offset	内核开始地址相对于起始物理地址的偏移
PSpace	物理地址直接映射到虚拟地址的一段区域

图3.2展示了 seL4 的地址空间设计，左侧为虚拟地址空间，右侧橙色部分为真实物理地址空间。seL4 将内核空间放在了高位全 1 的地址上（图中蓝色部分），将用户空间放在了低位的虚拟地址空间（图中绿色部分）。seL4 采用了单页表的设计方法，即内核与应用程序使用同一张页表，不同的用户程序之间使用多张页

表，在为用户程序创建页表的过程中，seL4 会将内核代码拷贝到用户程序的高位地址空间，当发生中断异常时，直接跳到中断处理程序的虚拟地址就可以在内核中进行执行。

两种地址空间的设计方法都各有千秋，在选择使用方案的时候，我考虑到了进程间通信使用的 IPCBuffer 开在用户地址空间，内核要想访问就必须使用 seL4 的单页表设计方案。所以最终选择了这种方案。但完成以后，随着重写的过程中越来越了解 seL4，我发现 seL4 IPCBuffer 的访问原理并不是通过用户态的虚拟地址空间，在图3.2中可以看到，内核地址空间中存在着 PSpace 段，这一段是物理地址空间直接映射的结果。内核可以通过访问 PSpace 段中的虚拟地址来访问用户程序的 IPCBuffer 来完成进程间通信的任务。现在仔细想来，seL4 完全可以使用用户态页表、内核态页表分离的设计方式，这种设计方式还有这防止侧信道攻击的优点，但 seL4 最终没有使用这种方法，我认为可能与性能有关，因为系统调用的频繁调用，进入跳板页切换页表会浪费大量时间影响实际性能。最终，我还是选择了 seL4 这种内核地址空间设计模式。在实现前，我们还要解决两个相关的问题：单页表的安全性问题与物理地址虚拟地址平滑切换的问题。安全性问题 riscv 已经能够帮我们解决，在 riscv 设计中页表项中存在着 U 位，用来管理当前页表项对应的物理地址能否在用户态被访问。我们将内核代码的页表项置为 0 后，就能够保证内核代码只被特权态访问而不被用户程序访问。关于平滑性问题，我们在3.2.2小节解释。

3.2.2 地址转换平滑问题

```
satp::write(words); //由物理地址转换为虚拟地址的代码
sfence();//清空 cache
```

在执行完上方的第一句代码后，就开启了虚拟地址访问。而 cpu 并不会知道这一句指令是开启虚拟地址，于是它就仍然按照正常指令的方法去将 pc 寄存器加 4，而这时，会用虚拟地址的方法去寻找 sfence() 指令，因为 pc 只简单的做了加 4 处理，要想保证能够正确的寻找到 sfence() 指令，就必须保证 sfence() 指令的虚拟地址与物理地址位置相同，再进一步说，就必须保证开启页表前后的代码在虚拟地址中做的是直接映射。这就是保证开启虚拟地址前后代码的平滑性保证。在 seL4 中，这件事情由 elfloader 完成，elfloader 做了一个直接映射，并将 pc 放到了高位地址空间，完成这些后，再进入内核执行。所以，我们如果查看 kernel

elf 文件的入口地址，可以看到入口地址是一个高位全 1 的虚拟地址。

要想使用 seL4 的地址空间设计，我们就要实现 seL4 在 elfloader 中完成的开启虚拟地址的工作。在参考了几份其他内核的复现工作后，我发现了解决方法：在 boot 阶段，开启页表前，我们先开启一个简易的页表，这张页表只做了两个映射，将 0x8000_0000 与 0xffff_ffff_8000_0000 开始的虚拟地址，都映射到 0x8000_0000 的物理地址上。开启页表后，再将 pc 跳转到 0xffff_ffff_8000_0000 开始的虚拟地址上，就能够成功运行。具体实现方法在下方的汇编代码中给出。

```
    set_boot_pt:
        la   t0, boot_page_table_sv39
        srli t0, t0, 12
        li   t1, 8 << 60
        or   t0, t0, t1
        csrw satp, t0
        sfence.vma
        la   t0, rust_main
        li   t1, 0xffffffff00000000
        add  t0, t0, t1
        add  sp, sp, t1
        jr t0
    boot_page_table_sv39:
        .quad 0
        .quad 0
        # 0x00000000_80000000 -> 0x80000000
        .quad (0x80000 << 10) | 0xcf
        # removed
        #.quad 0
        .zero 8 * 507
        # 0xffffffff_80000000 -> 0x80000000
        .quad (0x80000 << 10) | 0xcf
        .quad 0
```

3.2.3 实现细节

riscv 中 satp 寄存器用来管理虚拟地址，satp 的结构如表3.5所示，值得注意的是，satp 的根页表地址只保存了高 44 位，机器读取根页表的地址的方法是将 ppn 左移 12 位。也就是说，我们需要将根页表放在一个 4096 字节对齐的位置上，为此，我修改了 reL4 的链接脚本。创建了一个 4K 字节对齐的段，将全局页表放在这个段中。

表 3.5　satp 结构

satp 段名：位数	描述
PPN : 44	用来存放根页表地址
ASID : 16	根页表对应的地址空间标识符
MODE: 4	管理是否开启虚拟地址，以及开启虚拟地址的格式

```
. = ALIGN(4K);
.page_table :
 {
      . = ALIGN(4K);
      *(.page_table.aligned)
 }
 . = ALIGN(4K);
```

代码 2　链接脚本添加内容

reL4 的地址空间基础就是一个简单的 sv39 页表，支持地址空间标识符 (Address Space ID ,asid)。而在所有的页表操作之上都增加了一个 capability 的管理操作，在这里，我们主要介绍 reL4 如何使用 capability 来管理内核地址空间。地址空间部分 reL4 主要设计了两种 capability，pagetable cap: 管理根页表与 frame cap: 管理其他级页表。两者差别不大，由于篇幅原因，在这里展示 pagetable cap 的管理方法。表3.6列出的是 pagetable cap 的具体结构，用于理解代码3中的内容。在其他的内核中，pagetable unmap 只是将某一个页表项从根页表中抹去，要做的操作应为从根页表开始，逐级查询虚拟地址，找到目标槽后将目标槽清空即可。在 reL4 的底层处理逻辑中，也是这样的逻辑执行，但 reL4 在这一底层逻辑上又封装了一层，这一层做的工作就是 capability 对 pagetable 的管理，这一层位于代码3中。这一层操作的对象为 pagetable cap, 刚才说的底层操作需要的信息，如根页表地址，虚拟地址等内容都存在于 capability 中，reL4 内核将这些信息从 capability 中取出，再去执行 unmap 操作，最后修改当前 capability 中存储的信息。将以上过程抽象为一个层，用户程序想要完成地址空间中的底层操作只需要提供对应 capability，由于 capability 存储的信息只能由内核设置、修改，可以保证用户程序操作的合法性，不会访问、修改非法的内存数据。

表 3.6　pagetable cap 结构

capability 结构：位数	描述
MappedASID:16	当前页表对应的 asid
BasePtr:39	当前页表的根页表的地址
IsMapped:1	当前页表是否被映射
PTMappedAddress:39	映射的虚拟地址的开始位置
padding:28	填充位

```
#[no_mangle]
fn performPageTableUnmap(cap: &cap_t, ctSlot: *mut cte_t){
    if capPTIsMapped(cap) != 0 {
        let pt = getCapPTBasePtr(cap) as *mut pte_t;
        unmapPageTable(
            getCapPTMappedASID(cap),
            getCapPTMappedAddress(cap),
            pt,
        );
        clearMemory(pt, seL4_PageTableBits);
    }
    unsafe {
        setCapPTIsMapped(&mut (*ctSlot).cap, 0);
    }
}
```

代码 3　Pagetable Unmap 操作

除去 capability 管理层的设计，reL4 的内核地址空间模块的实现与其他 riscv 内核不再有任何其他区别，所以不再详细介绍实现的过程，只是简单的罗列实现的功能，位于表3.7中。

表 3.7　地址空间模块支持的 invocation

invocation	描述
PageTableMap / PageTableUnmap	map 或 unmap 根页表
PageMap / PageUnmap	map 或 unmap 一个页，可以是次级页表，也可以是 pte 页
PageGetAddress	获得根页表地址
ASIDControlMakePool	创建一个 ASID 池，用于在应用程序之中创造子集
ASIDPoolAssign	为 ASID 池绑定一个根页表地址

3.3 线程调度

3.3.1 ThreadControlBlock 的结构设计

seL4 的线程控制块（ThreadControlBlock，TCB）与其他的各个模块都有着关联，几乎所有的模块都要用 TCB 中记录的数据。接下来，我们简单的介绍一下 seL4 的 TCB 结构，通过代码注释的方式解释其中各个数据的作用。

表 3.8 线程控制块结构

线程控制块元素	描述
tcbArch	记录整个 trap 上下文信息，包括 32 个通用寄存器、sepc、sstatus
tcbState	占用 24 个字节，记录当前线程状态、线程等待的 IPC 内核对象的信息
tcbBoundNotification	线程绑定的 notification 对象, 关于 notification 可参考3.6
tcbFault	记录当前线程发生的普通错误
tcbLookupFailure	记录当前线程发生的查询错误，如查找页表、capability 时发生的错误
tcbFaultHandler	错误处理函数，需要在用户态由用户线程实现
domain	当前线程运行的域
tcbMCP	当前线程可控优先级（能够设置自身或他人的最高优先级）
tcbPriority	当前线程的优先级
tcbTimeSlice	当前线程的时间片
tcbIPCBuffer	用于进程间通信的 buffer 所在位置（物理地址）
tcbSchedNext/tcbSchedPrev	维护了用于调度的双向链表，一张链表中线程的优先级相同
tcbEPPrev/tcbEPNext	维护了用于进程间通信的链表，同一链表中线程用一个 endpoint

除去这些在 tcb 中定义的数据，每个线程还绑定着一个 CSpace 、VSpace，也就是自身的 capability 空间和虚拟地址空间。CSpace 通过内存的设计跟 tcb 绑定在一起，如图3.3所示，一个 tcb 的指针指向的区域实际上就是 Cspace 空间中的一段区域，而要想真正读取 tcb 中的信息则还需要加一个 TCB_SIZE_BITS 大小的偏移。而 VSpace 的绑定则与 CSpace 有关，在 CSpace 中，有一个固定的槽位专门用来存放根页表的地址空间，通过根页表就可以找到整张页表的位置。

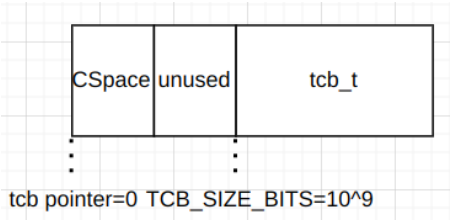

图 3.3　tcb 结构

tcb 作为存储各个模块所需信息的结构，其提供的 invocation 类型也是多种多样，我们可以简单的将其分为三类：

1. 对自身存储的任务上下文中的寄存器的操作，如：TCBReadRegisters、TCB-WriteRegisters、TCBCopyRegisters
2. 对自身存储信息的更新：TCBSetPriority、TCBSetIPCBuffer、TCBSetSpace
3. 对线程执行状态的操作：TCBSuspend、TCBResume

3.3.2　调度

reL4 调度方式采用的最简单的时间片轮转算法，时间片默认大小为 5，当发生时钟中断时，时间片减一，时间片耗尽后选择新线程继续运行。如果没有进程间通信，reL4 的调度方法就可以用一句话概括，但由于有了进程间通信，导致调度的具体情况要考虑阻塞线程变为非阻塞的特殊情况。为了保证进程间通信的高效性，reL4 希望从进程间通信完成后，进程间通信中被阻塞的一方如果满足优先级条件，能够率先执行。这样可以提高进程间通信的效率。

为了将提高进程间通信的目标与普通的时间片轮转算法结合，reL4 设置了 ksSchedulerAction、ksCurThread 变量，变量类型为线程控制块的指针。ksCurThread 存储的是下一个将要被运行的线程，而 ksSchedulerAction 则代表着调度时采取的行动，它有三种可能：

1. 重新运行先前运行的线程，ksSchedulerAction 的默认行为。当先前线程因为普通的中断、异常、系统调用等原因，进入内核态，返回时需要继续执行上述线程。
2. 选择新的线程继续运行，当时间片耗尽时，可将 ksSchedulerAction 设置为这种行为。
3. ksSchedulerAction 本身代表一个竞争线程，当进程间通信中阻塞线程变为非阻塞时，可以将阻塞线程的值写入 ksSchedulerAction。

为了根据 ksSchedulerAction 的值选择合适的线程写入 ksCurThread 进行执行，

reL4 实现了 schedule 调度方法，它分别处理了上述三种情况，针对前两种方法的处理相对比较简单，schedule 只需去做对应的操作即可。而针对第三种情况，schedule 并不会直接将竞争线程当做下一次执行的线程，它会判断竞争线程是否为最高优先级，如果是，才会去执行进程线程，否则，就会将竞争线程写入就绪队列，等待调度。

3.4 启动阶段

启动阶段的流程：

1. 做内核地址空间的映射，将内核映射到虚拟地址上，并将整个物理地址空间映射到内核的虚拟地址空间中，完成后开启页表。
2. 做一些简单的内核初始化，如开启中断异常、开启时钟。
3. 管理可用的物理内存空间，将所有的空间交给 untyped capability 管理, 原理和实现细节可参考3.4.1节。
4. 计算初始线程 (initial thread) 所需的内存大小，寻找合适的物理地址放置初始线程。
5. 创建初始线程所需的变量，为初始线程构建用户态页表，并创建初始线程，将部分初始化信息传递给初始线程。
6. 开始调度运行初始线程。

接下来将分小节讲述启动阶段涉及到的部分的具体实现。

3.4.1 物理内存空间管理

可用物理内存空间由设备树文件规定，通过 python 脚本读取，写为文件后再跟随内核一起编译链接，spike 对应的可用内存只有一块（代码4）, 由 0x80000000 至 0x17ff000000。

```
memory@80000000 {
    device_type = "memory";
    reg = <0x00000000 0x80000000 0x00000000 0xfff00000>;
};
```

代码 4　dts 可用地址空间

内核在获得可用的地址空间后，首先会保留加载了内核、设备树二进制文件、用户程序的地址空间，然后创建初始线程，线程创建完成后将全部的剩余地址空

间交由初始线程管理。具体实现为在所有可用内存上建立 untyped capability，然后，将 untyped capability 插入初始线程的 CSpace 中。

3.4.2 untyped cap 对地址空间的管理

untyped capability 结构如表3.9所示，用户想要创建内核对象，必须通过 untyped capability 使用 UntypedRetype, 一种特殊的 invocation，因为用户运行在虚拟地址上，地址空间中并没有剩余的可用的内存。UntypedRetype 方法会首先检查使用的 untyped cap 的大小能否创建一个内核对象，如果能，就更新 untyped 大小、起始位置信息，并创建内核对象的 capability，插入用户程序的 CSpace 中。使用 untyped capability 来管理内存空间，杜绝了用户程序访问非法地址空间的可能，同时全部内存均被 capability 覆盖管理，不会发生内存泄漏、重复释放等问题。

表 3.9 untyped capability 结构

untyped cap 结构：位数	**描述**
capFreeIndex:39	用于计算可用内存块大小
capIsDevice:1	是否为设备内存，设备内存只能当做页表页使用
capBlockSize:6	用于计算可用内存块大小, 计算方法为 2^BlockSize+FreeIndex
capPtr:39	指向可用内存起始地址的指针

3.4.3 初始线程 (initial thread)

初始线程并非我们所运行的用户线程，而是 seL4 的管理线程，我们所运行的用户线程是全部是初始线程的子线程，由用户线程加载进内核、创建页表、线程控制块等内核对象后再调度运行。seL4 内核为初始线程创建了一个最小启动环境，这个环境包括了初始线程所需要的一切环境：CSpace、页表、ipcbuffer、asid 池、线程控制块。除去这些内核对象以外，seL4 还为初始线程提供了部分信息，由 bootinfo 结构体传递给初始线程，bootinfo 结构体如表3.10所示，内核将 bootinfo 写入指定的内存区域，由初始线程读取内存，从而实现消息在初始线程和内核间的传递。

表 3.10 untyped bootinfo 结构

bootinfo 结构	**描述**
nodeID	当前 cpu 节点 id

bootinfo 结构	描述
numNodes	cpu 节点个数
untypedList	可用内存的区间范围
ipcBuffer	初始线程绑定的 ipcBuffer 位置
initThreadCNodeSizeBits	初始线程 CSpace 大小

3.5 异常处理

3.5.1 特权级切换

reL4 特权级的切换是通过 trap_handler 与 restore_user_context 两个函数完成了，两个函数做的事情相反，trap_handler 将寄存器的值保存进内存中的线程上下文，保存运行现场，而 restore_user_context 则将线程上下文的值写入规定的寄存器，恢复运行现场。

线程上下文保存的值为 32 个通用寄存器，sstatus、scause、spec 三个特权级寄存器以及下一次执行的地址。在线程运行时，线程上下文的内存地址写在 sscratch 寄存器中，当遇到问题后，将 sscartch 的值与 t0 寄存器交换，开始保存线程上下文。

由于 reL4 采用了用户态与内核态共用一张页表的设计模式，这让 reL4 在特权级切换的过程中处理简单了不少，只需要保存、恢复线程上下文的值并切换栈指针即可。

3.5.2 异常处理

异常处理的入口有五个，分别是 c_handle_syscall、c_handle_interrupt、c_handle_exception、c_handle_fastpath_reply_recv、c_handle_fastpath_call，后两者与进程间通信有关，我们将在3.6.3节进行介绍。前三个入口分别处理系统调用、中断与异常，根据 s0 寄存器的值来判断进入哪个异常处理函数。c_handle_exception 函数将所有的例外全部通过进程间通信交由 fault_handler 处理，fault_handler 程序放在用户态由用户线程实现提供服务。c_handle_interrupt 中仅支持发送信号（signal interrupt）、时钟中断（timer interrupt）两种中断，均可以由内核处理。这两种异常处理相对比较简单，而 c_handle_syscall, 则相对比较复杂，我们在接下来的一个小节中进行介绍。

3.5.3 系统调用的处理

seL4 支持的系统调用相对较少，列在表3.11中。

表 3.11 系统调用

系统调用	描述
SysSend	发起一个阻塞的 invocation。
SysNBSend	与 SysSend 相比是非阻塞的，但如果内核不能立即处理会发生错误
SysRecv	获得进程间通信传递来的消息
SysNBRecv	非阻塞的接收进程间通信传递的消息，不存在就报错
SysYield	将自身线程挂起，不在占用 cpu
SysCall	与 SysSend 类似，但用于处理进程间通信问题时，与 SysSend 逻辑不同在3.6节被提及
SysReply/SysReplyRecv	在进程间通信时，用于由接收者向发送者传递消息

reL4 中，SysSend、SysNBSend、SysCall 并非只是语义上的进程间通信的发送，它们还负责用户程序向用户请求的一种特殊的系统调用:invocation。它们三者在处理 invocation 上的不同之处在于：

- SysSend: 处理不需要向用户程序传递信息的 invocation，如 setPriority、Delete capability 等，这类服务内核完成后直接返回用户程序即可，不用告诉用户程序。
- SysNBSend: 与 SysSend 类似，但处理 invocation 非阻塞，如 Delete capability 如果当前不能删除，直接返回。
- SysCall: 处理需要向用户传递信息的 invocation，如 ReadRegister，当读取完成后会将寄存器的值写入用户线程的消息寄存器。

笔者认为这种 invocation 完全可以用多个系统调用代替，也想过将所有的 invocation 改为系统调用，但由于系统调用涉及到上层应用程序请求内核服务的方式，为了保证上层应用的正常运行，reL4 在实现中只能使用 seL4 invocation 的设计模式。

3.5.4 invocation 寄存器使用

invocation 的处理如同函数调用类似，使用了 a0-a5 作为参数值写入 invocation，其中 a0 寄存器固定存放请求的操作对应的 capability 的索引，内核根据索引在用户线程 CSpace 中读取对应的 capability。a1 寄存器用于传递消息，判断需

要进行哪种 invocation,a1 寄存器存储的信息结构在表3.12中。剩余寄存器与具体的 invocation 相关，如 ReadRegister 时，a2 a3 寄存器存储着读取寄存器的开始位置和个数信息。

表 3.12　消息处理的 a1 寄存器的结构

名称：位数	描述
label：52	应用程序请求的 invocation 的标签
capsUnwrapped：3	用于进程间通信，传递 capability
extraCaps：2	对应的 capability 个数，写在 IPCBuffer 中
length：7	消息的长度（使用寄存器的个数）

3.6　进程间通信

seL4 的进程间通信是指 seL4 的线程与线程之间的一种协作方式，在 seL4 中并没有进程这一概念。进程间通信模块主要由两种内核对象组成：endpoint 与 notification，前者主要用于在线程之间传递消息、capability，后者则在线程间传递信号。

3.6.1　endpoint 对象

endpoint 内核对象的结构展示在表3.13中，其中包含一个记录 endpoint 的链表，以及当前 endpoint 的状态，endpoint 的链表只记录了链表的头、尾两个节点，链表指向的是线程控制块 (tcb), 而链表不属于头尾的中间元素，由 tcb 中的 tcbEPNext 与 tcbEPPrev 连接, 这样就可以构成一整张双向链表。

表 3.13　endpoint 结构

endpoint 结构：位数	描述
epQueue_head:64	进程间通信队列头指针
epQueue_tail:37	进程间通信队列尾指针
state:2	当前 endpoint 的状态

endpoint 状态机中有三种状态，列在表3.14中，当发送方、接收方有一方就绪时，状态会进入 Send/Recv 中等待另一方就绪，双方均就绪后，就会进入后续的处理流程。

表 3.14　endpoint 状态

endpoint 状态	**描述**
EPState_Idle	发送线程与接收线程均未就绪
EPState_Send	发送线程就绪
EPState_Recv	接收线程就绪

在后续的处理流程中，seL4 支持 send 和 call 两种处理方式。send 的处理方法相对简单，它会将信息写入接收方的寄存器和 IPCBuffer，将发送方的状态由阻塞态改为就绪态，然后完成这次进程间通信。而 call 的处理则稍显复杂，它不仅会将信息传递给接收方，同时，会在接收方的 CSpace 中插入一个 reply capability，此时发送方将被设为阻塞态，然后再完成此次进程间通信。reply capability 用于接收方发送 SysReply、SysReplyRecv 系统调用，用于接收方告诉发送方自己接收到了消息或向发送方传递消息，只有发生 reply 的系统调用后，发送方才会由阻塞态改为就绪态。

3.6.2　notification

seL4 中的 Notification 是一种异步的信号传递机制，它由线程控制块 tcb 中的 tcbBoundNotification 直接与线程绑定。

Notification 对象中存储着标记位（badge），每一位代表某个一个信号，发送方发送信号其实就是将 badge 中的某一位置 1。它能够实现异步的原理也在于此：修改 badge 后，发送信号的线程就可以继续运行，无需阻塞；接收方线程接收信号时，如果信息已被激活，则可立刻返回，如果未被激活，也有着阻塞等待或退出运行两种选择。Notification 状态表列在表3.15中。

表 3.15　Notification 状态

Notification 状态	**描述**
NtfnState_Idle	发送线程与接收线程均未就绪
NtfnState_Waiting	接收线程就绪，发送线程未就绪
NtfnState_Active	发送线程就绪

3.6.3　进程间通信的高效性

seL4 的进程间通信高效性仅考虑 endpoint 通信时的效率，为了保证通信时的高效性，做了两点优化：

1. 消息寄存器设计：在 seL4 中，a2、a3、a4、a5 被设计为消息寄存器，当线程间传递的信息较短时，可以通过消息寄存器直接向对方发送，而当传递信息长时，则需要线程绑定的 IPCBuffer 做拷贝内存的方法发送，此时的通信效率就会变低。
2. fastpath 实现：fastpath 是一种 seL4 设计的快速处理消息的方法，它直接被中断处理函数调用，支持对 SysCall/SysReplyRecv 两种系统调用使用。fastpath 的优化原理是优化线程调度的时间，当由于发生上述两种系统调用导致线程切换运行时，如果满足一定的条件，就可以使用快速系统调用处理方式 fastpath。由于只有上述两个系统调用涉及到线程间的切换运行，所以只能优化上述两种系统调用。fastpath 代码可以分成两部分，前半部分做是否符合 fastpath 的条件判断（如是否有对应的 endpoint capability 等），当条件满足判断后，再由后半部分完成 call/replyrecv 操作。如果 fastpath 进行中条件不满足，或在后半部分执行过程中出现错误，则进入常规的系统调用处理函数继续处理。图3.4即为 fastpath 的原理，每一个节点代表一种内核的操作，左侧部分为常规的系统调用处理，内核的操作主要集中在后半段，处理起来很复杂，右侧部分为 fastpath 处理，操作主要集中在前半段，条件通过后处理简单。fastpath 需要满足的条件为:
    - 必须是 Call SysReplyRecv 两种系统调用
    - 通信的信息大小不超过消息寄存器大小
    - 线程必须有合法的地址空间
    - 进程间通信的过程中没有权限令牌的传递
    - 其他所有线程的优先级均小于或等于进程间通信线程的优先级

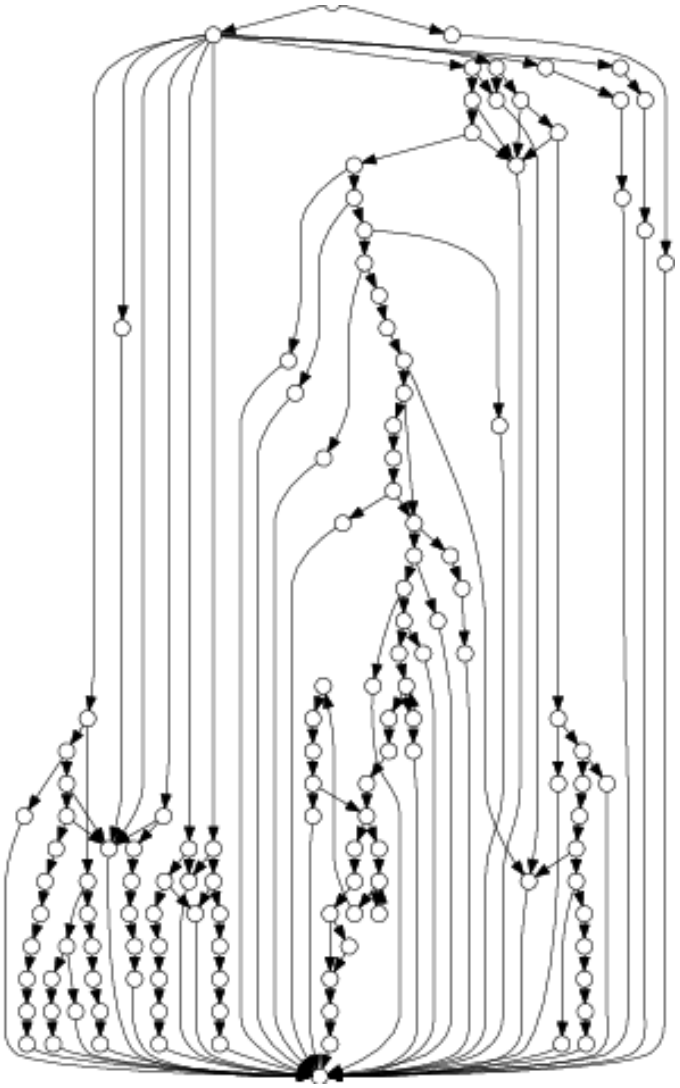

图 3.4　fastpath 原理

第 4 章　reL4 测试与分析

上一章完成了 reL4 功能模块的具体实现工作，但仍留有许多问题亟待解决：reL4 内核功能实现是否正确？invocation 能否被正确解析？rust 实现的 reL4 内核能否兼容 c 编写的用户程序？rust 引入的 unsafe 是否会对 reL4 带来新的问题？reL4 性能相比 seL4 是否会有下降？本章将聚焦这些问题，通过运行 seL4Test 测试的，分析 unsafe 引入原因的方法来解决大部分问题。需要特别说明的是，seL4 有着专门的测例 seL4Bench 来测试 seL4 的性能，但该测例当前仅支持真实硬件上的运行, 而笔者当前的工作大都基于 qemu 模拟的 riscv 虚拟机，由于时间原因暂时没有上板运行，因此暂时无法测试 reL4 的性能，而在后续工作中，笔者会完成 reL4 的上板工作并进行该项测试。

4.1　seL4Test 测例

seL4Test 测例程序是 seL4 中规定的初始线程，他会通过创建众多子线程的方法来测试 seL4 中各个功能模块的实现正确性，由于 reL4 的功能模块设计与 seL4 相同，且在实现过程中 reL4 的启动方式也更改为了 seL4 的启动方式，所以 seL4Test 从理论上讲也可以对 reL4 进行测试，进而证明 reL4 的功能正确性。

4.1.1　运行 seL4Test 测例方法

在2.1节，我们简单介绍了 seL4Test 的测例结构，说明了 seL4Test 的依赖库众多，cmake 文件就有一万行代码。要想从头构建整个 seL4Test 项目来供 reL4 使用是不可能的。而如果简单分析 seL4Test 项目，简单可以将其分为三个模块，如图4.1所示，大概可以分成三个模块：elfloader、kernel 与 seL4Test，我们只需要关系两个模块与进入 kernel 的接口是什么，对 elfloader 而言是 _start 内核起点函数，对 seL4Test 来说是 trap_handler 中断处理函数，所以只需要将 reL4 的这两个接口定义为 #[no_mangle]，即编译不改变函数名，让链接时 seL4Test 能找到即可。为了确保 seL4 的内核不会被调用，在 seL4Test 中还需要将 seL4 的内核代码全部注释掉。

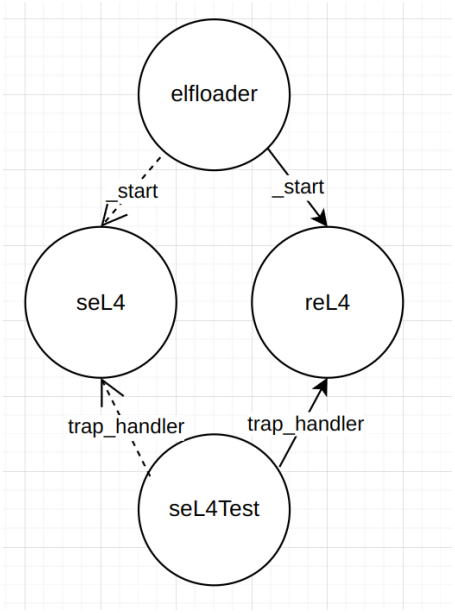

图 4.1　seL4Test 模块

完成上述工作，只是简单的将 seL4 模块从 seL4Test 项目中整个去除，为了添加 reL4 内核，还需要将 reL4 编译成为静态链接库，然后通过下方的代码，在 seL4Test 项目中将 reL4 内核加载进来。至此，整个针对 reL4 内核的 test 测试程序，我们称为 reL4Test 测例就已经搭建好了，可以进行后续的实验工作。

```
link_directories("path to reL4 library file")
```

代码 5　CMakeList.txt 添加内容

4.1.2　测例运行情况

在 reL4 中系统调用号、invocation 标签号均与 seL4 中设置相同，理应能够正常的支持上层应用的运行，实验测试结果与理论相符，表4.1为测例的通过情况，图4.2所示为最终通过测试的截图，可以看到当前能够运行 seL4 中的 112 个测例，还有 51 个测例与混合临界系统、io 设备、真实物理硬件有关，当前暂不支持。

表 4.1　reL4Test 测例通过情况

测例名称	通过情况	测例名称	通过情况
SysCall	15/15	Bind	4/4
Cancel Badged Sends	2/2	Cnode Operation	9/9

测例名称	通过情况	测例名称	通过情况
CSpace	1/1	Domain	3/3
Fpu	2/2	Frame	3/3
IPC	9/9	IPCRights	5/5
PageFault	9/9	Untyped Retype	3/3
Schedule	6/6	Serial	19/19
Sync	4/4	Thread	2/2
VSpace	6/6		

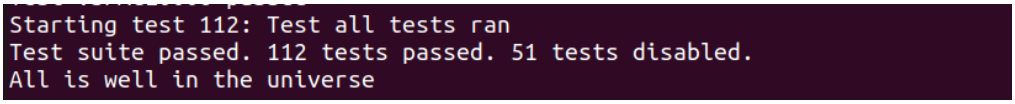

图 4.2　reL4Test 测试结果

4.2　reL4 unsafe 块引入探究

unsafe 块是 rust 特有的机制，它使 rust 的使用更加灵活，但却破坏了 rust 提供的安全性保障。unsafe 块是底层软件实现过程中必然使用的一环，在 reL4 的实现过程中也多次使用了 unsafe 块。reL4 将更高的安全性作为用 rust 重写 seL4 的初衷，其安全性是否会被 unsafe 块的引入影响？为了解决这个问题，我们要分析 reL4 中 unsafe 块的引入原因，分析使用是否会带来问题。

4.2.1　静态可变变量的访问修改

在 rust 中，访问和修改静态变量是不安全的，因为 rust 担心存在数据竞争问题，多个线程同时访问修改静态变量的值，造成代码的执行逻辑出错。参考其他的内核的实现，无论是 c 还是 rust 实现的内核，当定义了一个静态可变变量时，往往在变量的外面加一个锁，每次访问时都在临界区中进行访问。而 reL4 的实现没有这种考虑，seL4 设计的目标之一为通过形式化的验证，而并发的系统会为形式化验证引入新的问题，所以在 seL4 设计之初就否认了并发的设计[7]。作为 seL4 的 rust 版实现，reL4 自然也没有引入并发，全局变量能保证每次只被一个线程访问修改，因此不会造成问题。

4.2.2 解引用裸指针

rust 中提供了两种裸指针类型：*const T 与 *mut T, 而解引用裸指针，特别是将一个整数变量类型转换为裸指针后直接对其进行解引用，这种操作在 rust 中是十分危险的。我们常常能在 C 语言中见到这种操作，但显然这种操作是不安全的，它可以访问任意的内存区域并修改数据，严重威胁了数据的保密性、完整性与可用性。在日常的代码编写中，应尽可能的少使用这种操作，但对于底层软件而言，这类操作是难以避免的，reL4 由于引入了 capability 机制、进程间通信等功能模块，这类操作使用更加频繁，为了保证安全性，我们需要逐一分析解引用的裸指针的地址的合法性。笔者简单统计了 reL4 中解引用裸指针的情况：

1. 向 CSpace 中插入 capability、读取 CSpace 中的 capability：这类操作需要向固定的内存写入数据或访问固定内存的数据。内存的地址为 CSpace 地址加给定的 offset，寻址时，内核保证 offset 不会越界，否则会报错，而 CSpace 是内核在可用物理内存上创建的内核对象，它的地址也是可以保证合法的。两者均能保证合法，就可以保证这类操作的安全性。
2. boot 阶段内核向初始线程通过 bootinfo 结构体传递初始化信息：bootinfo 结构体写入的位置由内核直接确定，可以保证安全性。
3. 进程间通信访问修改 ipcBuffer 写入数据：ipcBuffer 的地址也由内核确定，可以保证安全性。
4. 各类内核对象的访问：在 capability 中存放着指向特定内核对象的指针，由此来管理全部的内核对象，内核对象的创建由内核在可用内存上构造，并存入 capability 中，因此访问内核对象的地址都能够保证是合法的。

reL4 解引用裸指针的情况基本就分为以上 4 种，分别说明各种操作解引用指针的地址都是合法的，进而证明了在 reL4 中解引用裸指针的操作是安全的。

4.2.3 asm

rust 中的 asm 函数可以直接向代码段中增加汇编内容，可以直接整个系统的运行，因此它是 unsafe 的，使用时，我们要保证 asm 功能的正确性。本项目中 asm 使用场景有：切换地址空间，开启、关闭中断，由内核态返回特权态。在每个场景中，asm 函数使用都参考了 riscv 特权级手册，能够保证使用的正确性。

4.2.4 from_raw_parts_mut 函数

from_raw_parts_mut 函数是 rust core slice 库中的第三方函数，它的作用是以一个裸指针开始，将指定区域的大小变成这个裸指针类型的切片,unsafe 的本质原因与解引用裸指针相同，都是担心访问不合法的物理地址。而 reL4 中该第三方函数被使用了两次：1.clearMemory 函数，顾名思义，使用该函数来清除一片内存区域，而 clearMemory 只会在构建内核对象前清理对应的合法内存区域，内核能够保证使用上的安全性。2. 加载设备生成树的二进制文件到指定内存，内存由内核指定，也能保证访存的安全性。

4.3 reL4 性能测试

本章开始提及官方的 seL4 的性能测试程序，seL4Bench 需要真实物理硬件才能运行。但仍可以通过修改 seL4Test 测例的方法进行简单的性能测试。于是，笔者选择了 seL4 中性能方面最重要的进程间通信来进行简单性能测试。

4.3.1 运行环境

硬件环境使用 Ubuntu 上安装的 QEMU 6.2.0 模拟 riscv 架构 64 位 CPU。

4.3.2 准备工作

seL4Bench 不能直接运行的原因之一是 seL4 内核本身并不提供时钟计时功能，时钟需要用用户态的程序实现，并通过进程间通信告诉其他线程。在修改 seL4Test 的初期，也面临着这样的问题，而 risc-v 中的 time 寄存器恰好可以提供时钟的功能。只需在 reL4 中添加新的系统调用 getTime，并将 time 寄存器的值写入线程上下文的 a0 寄存器中即可，用户程序只需要通过内联汇编进行调用，就可以在用户程序中获取时间信息。

```
pub fn handleGetTime() {
    let tmp: usize;
    unsafe {
        asm!("rdtime {}",out(reg)tmp);
    }
    unsafe {
        setRegister(ksCurThread, a0, tmp);
    }
}
```

代码 6　reL4 内核添加 getTime 系统调用

```
static int get_time(){
    int ret= SBI_ECALL(getTime,0,0,0);
    return ret;
}
```

代码 7　reL4 内核添加 getTime 系统调用

完成修改后，就可以在进程间通信中测量时间。

4.3.3　测例介绍

本次实验使用的测例是 seL4Test 中关于进程间通信的测例, 具体测例编号为 **IPC0002**, 测例会在三种不同的发送方、接收方优先级关系下，进行进程间通信，记录进程间通信耗费的总用时。重复十次实验，平均用时的结果记录在4.2中。

表 4.2　性能结果

内核情况	用时（单位为 time 寄存器变化数）
seL4 内核（使用 slowpath）	593578
seL4 内核（使用 fastpath）	624197
reL4 内核（使用 fastpath）	830531

表4.2结果表明，reL4 内核在进程间通信方面效果仍不如 seL4，笔者推测的原因如下：

1. reL4 当前只是进行了 seL4 功能的实现，性能方面不曾进行任何优化，导致存在部分性能的损失。

2. 笔者仍处于 rust 语言的学习阶段，部分 rust 语言的特性仍使用不熟练，导致部分优化的功能不能很好的实现，如 seL4 在实现内存拷贝的时候，会使用 register 修饰符，先将源数据写入寄存器再写入目标内存地址，通过使用寄存器来优化拷贝内存的性能，而 rust 中笔者则不清楚相关的机制，导致类似的拷贝速度较慢。当通过链接外部函数的方法将 c 的拷贝内存函数接口接入 reL4 项目后，进程间通信时间由 830531 降至 807826。这些影响性能的问题也将成为今后优化 reL4 的方向之一。
3. seL4 中从硬件角度充分考虑了编译期方面的优化，内核中的每一个分支跳转语句都 **likely unlikely** 关键字帮助编译期优化，功能性的简单函数都设置为 **inline** 类型，由于笔者项目开发经验较少，开发 reL4 时并未想过此类优化方面。

表 4.2中还进行了 seL4 使用 fastpath 和不使用 fastpath（seL4 中系统调用的最普遍处理方式，我们称为 slowpath）来处理系统调用的情况对比，结果表明，使用 fastpath 效果甚至不如不使用 fastpath，这与 seL4Bench 中的测量结果是不相符的，针对这个问题我们将在下一小节进行讨论。

4.3.4 fastpath 问题

在3.6.3节中，我们只是介绍了 fastpath 理论上会快的原因：可以不经过调度直接进入进程间通信的等待线程进行执行。如果从这一角度来看，对当前测例来说效果确实会不理想：当前线程有仅有发送、接收两个线程，就算使用 slowpath 来处理，选择线程调度运行也是非常方便的，而 fastpath 前期又会判断很多的条件是否满足，因此性能方面 fastpath 略差于 slowpath。解释了上述原因，笔者又想通过构造测例的方式证明 fastpath 在某些情况下有着更优秀的进程间通信性能，在此笔者进行了多方面的尝试，情况都不理想。网上的资料对于 fastpath 的解释有点模糊，只说明了 fastpath 可以节约系统调用处理完成后调度的时间。笔者认为，节约的时间有两种可能：1.slowpath 会在进程间通信过程中，进入无关线程执行而没有进入进程间通信的线程，导致进程间通信时间变长，而 fastpath 则不会有这样的问题。2. 调度本身会花费大量的时间，fastpath 会省去这部分时间。

第一种情况

针对这种情况，笔者创建了一个等待线程，这个线程进入后什么都不会做，等到达特定时间后退出，该线程有着和进程间通信中接收方相同的优先级，保证 fastpath 满足条件。

```
static int wait_thread()
{
    int start = get_time();
    while (get_time() - start < 1e10)
    {
        continue;
    }
    return SUCCESS;
}
```

代码 8　等待线程代码

实验结果表明，fastpath 与 slowpath 通信耗费时间仍然基本相同，也就是说 slowpath 在执行完发送方指令后也能立即调度到接收方执行，仔细分析逻辑后，发现 slowpath 通过内部的 possibleSwitchTo 函数，也能够保证发送方发送完成后直接调度到接收方执行。因此，fastpath 并没有进行这方面的优化。

第二种情况

为了检验第二种情况 fastpath 是否有效，我们创建了一组等待线程加入就绪队列等待调度，看调度本身的耗时是否增加，但实验结果表明 fastpath slowpath 的耗时关系与第一次实验相同，也排除了 fastpath 进行了这方面的优化。

网上资料有很多内容都说明了 fastpath 对 seL4 进程间通信有着很好的效果，甚至 seL4Bench 官方也有着 fastpath 比 slowpath 性能好的数据，因此笔者断定自己的实验结果是存在问题的，几个可能存在问题的点为：

1. 计时方法有问题，笔者现在采用的计时方法为通过系统调用的方式从内核中读取时间，而在上板实验时，seL4Bench 采用了真实物理时钟作为应用程序的方法向 seL4Bench 项目提供计时功能，计时方法的不同导致计算性能出错。
2. qemu 只是模拟器不是仿真器，它只能模拟运行的行为，或许不能模拟精确的用时。
3. fastpath 主要针对混合临界系统或多核进行优化，而我们并未实现这一部分功能。

由于笔者毕设时间有限，尚未对此进行进一步的研究，但在今后，这将会是笔者研究的方向之一。

4.4 rust 开发内核难点分析

先前对 rust 的介绍，主要说明了 rust 开发内核的优点，而 rust 写内核的实际感受又会如何呢，笔者将从一个对 rust 了解不多的编程人员的角度入手，分析自己在 rust 开发内核过程中遇到的问题与难点。

硬件控制能力

rust 有着和 c 相同的硬件控制能力，使用 unsafe 块可以做到直接访问、修改内存数据，而 rust 自带的内联汇编又支持对寄存器的操作。这两个特点让 rust 有着和 c 相同的硬件控制能力，但 rust 对裸指针的使用并不像 c 那样简洁。在修改结构体指针的成员变量时，不能像 c 一样通过箭头直接修改成员变量，只能通过 unsafe 块和解引用裸指针的方法得到结构体再访问成员变量，而 seL4 中又在特定内存放置了大量的结构体，这导致 reL4 开发的代码复杂且不美观。

入门难度高

由于笔者自身为 rust 初学者，对于 rust 特别的用法了解不多，这给笔者对 rust 开发内核带来了很多的问题，浪费了大量的时间。举个最简单的例子，riscv 中基页表要放在 4096 字节对齐的位置上，因为 satp 寄存器中默认基页表后 12 位为 0，对于 c 语言来说，只要在全局变量上加入 aligned 即可，而在 rust 中并没有对应操作，笔者只能在链接脚本中开辟了特定的一个段，将页表直接放入这个段中。

rust 入门难度高的另一点体现在所有权机制上，由于 rust 所有的赋值均为移动赋值，而 c 语言中默认为拷贝赋值，这种不同让笔者在将 c 翻译为 rust 的过程中特别容易出现所有权转移的问题，甚至在某些复杂情况下用直接用拷贝赋值的方法代替移动赋值，这反而让 rust 的优势没有发挥出来。

未定义行为

在 reL4 的开发过程中，遇到了很多未定义行为 (undefined behavior,ub)，特别是在内联汇编中。

```
asm!("mv a0,a2",in("a2") badge);
asm!("mv a1,a2",in("a2") msgInfo);
asm!("mv t0,a2",in("a2") cur_thread_regs);
```

代码 9　rust 内联汇编代码

```
ld a2,16(sp)
mv a0,a2
mv a2,a0
mv a1,a2
mv a2,s6
mv t0,a2
```

上面展示即为一种 ub，这段代码只是简单的将三个变量的值借助 a2 寄存器写入对应的寄存器，但在写入 msgInfo 的值时，直接将修改后的 a0 寄存器的值写入 a1，导致寄存器写入多的错误。这类 ub 也浪费了笔者大量的 debug 时间，不知是否因为笔者使用有误。

rust 在内核开发中存在的诸多问题，大都与笔者不熟悉 rust 语言有关，而 rust 提供的安全性是 c 语言无法比拟的，相信对于精通 rust 语言的内核开发者来说，rust 比 c 语言有着更大的吸引力！

第 5 章　总结与展望

到目前为止，reL4 已经能够完全支持 seL4 中的系统调用、invocation，支持 seL4Test 测例的正常运行并通过测试。但当前 reL4 还存在着很多问题：reL4 仅支持在 qemu 模拟的 riscv 架构 cpu 上进行运行，离上板实验还有一定的距离；reL4 作为内核为应用程序提供服务的流程复杂；没有在性能上同 seL4 相比较；没有进行形式化验证等。这些都是下一步工作可以展开的方向，我今后也会选择合适的工作继续完善项目。

本项目的开发过程, 也是用 rust 语言开发内核的一次新的尝试，在开发的过程中，我们看到了 rust 代替 c 作为底层软件开发中的可行性与优越性。相信 Rust OS 今后会在底层软件的开发过程中占据一席之地！

插图索引

表格索引

参考文献

[1] G H. The sel4 microkernel–an introduction[j][M]. 2020.

[2] Biggs S, Lee D, Heiser G. The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security[M/OL]. New York, NY, USA: Association for Computing Machinery, 2018. https://doi.org/10.1145/3265723.3265733.

[3] Community R. Redox os book[EB/OL]. 2015. https://doc.redox-os.org/book/.

[4] LiDongYang Z, ChenYunYi. 总结报告[EB/OL]. 2022. https://github.com/LDYang694/RFREERTOS/blob/master/docs/%E6%80%BB%E7%BB%93%E6%8A%A5%E5%91%8A.pdf.

[5] Chen Y. rcore-tutorial-book-v3[EB/OL]. 2022. https://rcore-os.cn/rCore-Tutorial-Book-v3/.

[6] Team T S. sel4 reference manual[M]. 2021.

[7] Gernot Heiser. sel4 design principles[EB/OL]. [2020-03-11]. https://microkerneldude.org/2020/03/11/sel4-design-principles/.

致　　谢

感谢陈渝老师和张福新老师，在毕设完成过程中对我整体方向上的指导，让我从对毕设工作无从下手变成现在最终成功完成毕设。二位老师在我学习操作系统的过程中也给予了我很多的帮助和指导，让我对操作系统有了初步的理解。

感谢陈洋和刘庆涛两位学长对我毕设的大力支持，解决我关于 seL4 内核的问题，详细为我讲解 seL4 的模块功能。他们对 reL4 的开发、调试、测试都起到了关键性的作用。

感谢向勇老师和廖东海对我中期后方向的指导，他们悉心的为我确认了中期后继续进行实验的方向，并对 seL4Test 测例的正常运行给出了详细的方案。

感谢龙芯公司和龙芯实验室的全体成员，他们为我提供了高效的学习环境，让我在毕设的开发过程中心无旁骛，潜心学习，没有他们的支持，在寝室中我真的完成不了毕设。

感谢本科期间的各位老师的认真教导，感谢本科同学的陪伴与帮助，感谢家人朋友的关怀与体贴，让我度过了安逸又快乐的四年本科时光。

附录 A　外文资料的书面翻译

从 L3 到 seL4,L4 微内核 20 年来我们学到了什么？

目录

A.1 摘要

L4 微内核已经使用了 20 年, 它有一个活跃的用户和开发人员社区，并且有大规模部署在安全关键系统中的商业版本。在这篇论文中，我们考察了在这 20 年中吸取的多年的微内核设计和实现中的教训。我们重新审视 L4 设计论文，并审视了其演变从最初的 L4 到最新一代 L4 内核，尤其是 seL4, 它将 L4 模型推向了更远的位置，seL4 是第一个经过其完整的形式验证的操作系统微内核，我们证明尽管改变了很多，但极简主义和高 IPC 性能仍然是设计的主要驱动因素。

A.2 介绍

20 年前，Liedtke[1]认为他设计的 L4 微内核 IPC 可以很快，可以比其他当代微内核快 10-20 倍。

微内核将内核提供的功能进行了极简化处理：内核提供了一组通用的机制，而用户态提供宏内核的服务[2]。用户态程序通过进程间通信与服务器提供服务，通常是传递消息。因此，进程间通信处在系统调用的关键路径上，IPC 成本至关重要。

在 90 年代初期，IPC 性能已经成为了微内核的致命弱点：单次消息传递的成本大约 100 微秒，这对于构建高效系统来说是不能接受的，导致微内核发展趋势中将核心服务移回内核[3]。也有人认为高 IPC 成本是微内核的系统架构天生带来的，不能克服[4]。

在上述发展情况下，Liedtke 展示的进程间通信优化非常可观，在之后的工作中，他证明了 L4 内核的原理和机制[5] [6]，L4 在准虚拟化 Linux 的演示上只有百分之几的开销。L4 内核已在数十亿移动设备上部署。最后，L4 内核已经成为了第一个证明功能正确性的微内核[7]。L4 微内核也对其他研究产生了深远影响[8]。

在这篇论文中，我们考察了 L4 过去 20 年的发展，具体来说，我们关注了现代 L4 内核与 Liedtke 的最初设计有什么关联，以及他设计的微内核中的哪一个必需点保留了下来。我们具体研究了过去的教训对最新一代 L4 微内核的设计的影响，我们以 seL4 为例[7]，但也指出其他当前 L4 版本在哪些方面做出了不同的设计决策。

A.3 背景

A.3.1 L4 微内核家族

L4 是由 Liedtke[9]于 20 世纪 80 年代初在 i386 上开发的早期系统 L3 开发而来。L3 是一个内置持久性的完整操作系统，它已经具备了用户模式驱动程序，这仍然是 L4 微内核的特点。L3 是商业性的部署在几千个设施中（主要是学校以及法律实践工作）。像当时所有的微内核一样，L3 的 IPC 成本约为 100µs。

Liedtke 最初使用 L3 来尝试新的想法，他在早期出版物中称之为“L3”[4]，但实际上是彻底的重新设计。1995 年，他第一次使用的名称“L4”伴随着“V2”ABI 从 1995 年开始在社区传播。在下文中，我们将此版本称为“原始 L4”。Liedtke 在基于 i486 的 PC 上用汇编程序完全实现了它，并很快将其移植到了 Pentium 上。

这项工作演变了 20 年，伴随着多次 ABI 的修订以及推翻从头再来，正如图A.1所示。这项工作始于德累斯顿大学和新南威尔士大学重新实施 ABI（做出必要的调整），后者在 C 中实现了所有更长的运行操作。这两个内核都实现了亚微秒级别的 IPC[10]，并作为开源项目被发布。UNSW Alpha 内核是 L4 的第一个多处理器版本。

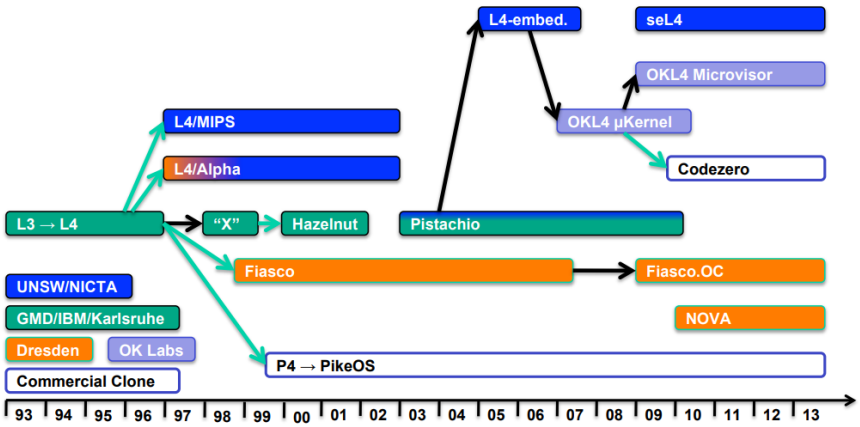

图 A.1 L4 家谱（简化）。黑色箭头表示代码，绿色箭头表示 ABI 继承。

后来，Liedtke 从 GMD 转到了 IBM Watson 工作，他继续开发在后来被称为 X 版的 ABI。事实证明，对其他研究人员来说过于严格的知识产权制度促使 Dresden 实施了一种新的从零开始的 x86 版本，称为 Fiasco。开源 Fiasco 是第一个几乎完全用高级语言（C++）编写的 L4 版本，当前代码仍被维护。它是第一个具有重要商业用途的 L4 内核（预计出货量高达 100000）。

Liedtke 搬到卡尔斯鲁厄后，他和他的学生们自己从头开始用 c 语言研发内

核，Hazelnut，这是第一个移植（而不是重新实现）到另一个体系结构（从 Pentium 到 ARM）的 L4 内核。

Liedtke 在卡尔斯鲁厄开发 X 版本和 Hazelnut 的经历让他开发了一个主要的 ABI 修订版：V4, 旨在提高内核和引用程序的可移植性、多处理器系统的支持和解决各种问题。2001 年 Liedtke 不幸去世后，他的学生在一个新的开源内核中实现了这一设计，L4Ka::Pistachio（简称为 Pistachio），它是用 C++ 编写的，最初是在 x86 和 PowerPC 上编写的，我们很快将其移植到了 MIPS,Alpha，ARM.1 等架构上。在这些移植中，少于 10% 的代码发生了变化。

表 A.1　L4 内核 IPC 成本

名称	年份	处理器	主频	周期	时间（微秒）
Original	1993	i486	50	250	5
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.1
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

在 NICTA，我们很快就将 Pistachio 用在了嵌入式设备中，叫做 NICTA::Pistachio-embedded。当高通公司将其作为保护态的实时操作系统后，它被大规模的生产部署。Open Kernel Labs 继续做了内核的开发工作，将其命名为了 OKL4 微内核。另一个部署版本是 PikeOS，这是 Sysgo 的商业 V2 克隆，经认证可用于安全关键航空电子设备，并部署于飞机和火车。

EROS 的影响[11]和日益增加的安全关注导致了 L4 发展向访问控制令牌[12]的转变。首先是 OKL4 的 2.1 版本（2008 年），随后是 Fiasco（更名为 Fiasco.OC）。针对形式验证，这对于并非为此目的而设计的代码库来说似乎是不可行的，因此相较于优化内核，我们选择了从头开发内核。

在过去的几年里，还出现了两种专门旨在支持虚拟化作为主要关注点的新设计，即来自德累斯顿的 NOVA[Steinberg 和 Kauer，2010] 和来自 OK 实验室的 OKL4 微内核[13]。

贯穿这二十年的一个共同主线是第A.4.1节中介绍的最小化原则，以及对关键 IPC 操作性能的高度关注；内核实现者通常旨在保持接近微架构设置的限制，如表A.1所示。因此，L4 社区倾向于以周期而不是微秒为单位来衡量 IPC 延迟，因为这更好地与硬件限制相关。事实上，第A.4.1节提供了硬件上下文切换友好性的有趣视图（比较 Pentium 4 和 Itanium 的周期数，它们都来自高度优化的 IPC 实现！）

A.3.2 当前代表工作

我们基于对 L4 设计演变的考察以及对 seL4 的评估（我们对此非常了解在许多方面都与最初的设计相去甚远）。我们注意到最近的其他版本最终采用了不同的设计，并试图了解其背后的原因，这些内容告诉了我们 L4 社区中的微内核设计的一致认同度。

与其他任何系统不同，seL4 的设计从一开始就支持安全性的保障，同时保持了 L4 传统的最小性、高性能和支持几乎任意系统架构的能力。

为此我们实现了一种全新的资源管理模型，其中所有空间分配都是显式的[14]，并由用户态代码（包括内核内存）管理。它也是第一个理论上有完整的最坏执行情况分析的操作系统内核[15]。

第二个相关的内核是 Fiasco.OC 它的独特之处在于它是一个经历过 L4 的大部分历史的内核，在某些情况下，他可以支持许多不同的 L4 ABI 版本。当前，它已经发展成了最新一代具有高性能、基于权限令牌访问控制的微内核。Fiasco 作为实验平台为许多设计探究提供支持，尤其是关于实时性操作系统的研究。

最后是两个从头开始的设计：NOVA，用于 x86 平台上硬件支持的虚拟化；OKL4 用于在 ARM 处理器上实现高效的准虚拟化。

A.4 微内核设计

Liedtke 指出了原始 L4 内核的设计原则与机制，我们探究了经过几十年的发展，原始设计与当前设计有什么不同。

A.4.1 最小化原则

Liedtke 的主要驱动力是最小化原则和高效的 IPC, 他坚信前者有助于后者。具体而言，他提出了内核的最小化原则：

> 只有当将一个概念移除 μ-内核，会影响系统所需的功能的实现的时候，才会在 μ-内核内部使用这个概念。例如，允许竞争的实现[5]。

这一原则一直是 L4 设计的基础。一下各界的讨论中将不断展示社区为了将某个特点换为更简单功能做出的努力。

对这一原则的坚持可以从代码规模中看出，如表A.2所示，虽然系统大小随着时间推移而增长是正常的，seL4 扔与早期版本的代码规模相当。

表 A.2 L4 内核代码规模

名称	架构	代码规模
Original	486	6.4
L4/Alpha	Alpha	14.2
L4/MIPS	MIPS64	10.5
Hazelnut	x86	10.8
Pistachio	x86	23.0
L4-embedded	ARMv5	9.0
OKL4	3.0 ARMv6	15.0
Fiasco.OC	x86	37.6
seL4	ARMv6	10.2

然而，直到今日，也没有一个 L4 内核开发者站出来说他们已经开发出了一个纯粹的、严格遵守最小化原则的微内核。举个最简单的例子，L4 内核中存在一个调度程序，它实现了一个特定的调度策略（通常是优先级循环），到目前为止，没有一个人提出一个真正通用的内核调度器，或是一个可行的低开销的调度策略机制。

A.4.2 IPC

先前，我们介绍了 IPC 的重要性，L4 内核持续的想要提高 IPC 性能，其中的细节已经发生了很大的改变。

A.4.2.1 同步 IPC

最初的 L4 中，仅支持同步 IPC 作为唯一通信方式。同步 IPC 避免了内核中缓冲的管理和拷贝所带来的开销。这也将是我们之后介绍的一些实现技巧的先决条件，如延迟调度、进程直接切换和临时页表映射。

尽管同步进程间通信符合最小化原则，概念实现上也很简单，但经验告诉我们同步 IPC 存在缺陷：强制在简单系统中加入多线程设计，造成同步的复杂性。例如，由于缺少 Unix 中的 select 操作需要每个中断源有单独的线程，而一个单线程的服务器没有办法同时监听和响应中断。

我们在 L4 中通过添加异步通知来解决这一问题，异步通知是异步 IPC 的一种非常简单的形式，我们后来在 seL4 中将此模型细化为异步端点通信：发送是非阻塞的且与接收方异步，后者可以选择阻塞或轮询来等待消息。从逻辑上讲，异步端点类似于单个变量中分配的多个二进制信号量，每个端点都有一个通知字段，发送方指定一个掩码位，该位与通知字段做“或”操作。

在我们目前的设计中，异步端点可以被绑定到特定线程。如果当线程在同步端点上等待时收到通知，通知像同步消息一样传递（并表明这实际上是一个通知）。

总结一下，与大多数其他 L4 内核一样，seL4 保留了同步 IPC 的模型，但也使用异步通知机制。OKL4 完全放弃了同步 IPC，取而代之的是虚拟中断（本质上是异步通知）。NOVA 使用计数信号量增强同步 IPC，而 Fiasco.OC 也有具有虚拟中断的同步 IPC 机制。

有两种进程间通信方式的设计违背了最小性原则，因此 OKL4 的方法在这一点上做的更好（尽管它的信道抽象是另一种违规行为）。此外，同步的 IPC 在多核的上下文切换中表现更差：类似 RPC 的服务器调用顺序化客户端和服务器，如果它们在单独的内核上运行，则应避免这种情况。因此，我们期望基于异步的通信机制变得更加普遍，而只有异步的 IPC 在未来是可能做到的。

A.4.2.2 IPC 消息结构

最初的 L4 上，IPC 具有丰富的语义。除寄存器支持的短消息外，它还支持几乎任意大小的消息（有着四字节对齐的“缓冲区”以及多个未对齐的“字符串”）。再加上全同步设计，这种方法避免重复复制问题。

消息寄存器不需要拷贝: 内核总是从发送方的进程上下文切换到接收方的进程上下文而不改变消息寄存器的值。

这种设计的缺陷是体系结构依赖性，以及可以零成本拷贝的消息的大小很小。事实上，可用寄存器的数量随着 ABI 的改变而频繁变化，因为系统调用参数的更改会改变寄存器的使用情况。

Pistachio 引入了中等规模大小的虚拟消息寄存器的概念。实现中将它们部分映射到了物理寄存器上，剩余的随线程绑定在固定的地址空间上。绑定的地址空间保证不会存在缺页的错误。内联函数的使用隐藏了虚拟寄存器和物理寄存器的区别。后来 seL4 和 Fiasco.OC 都沿用了这种设计方法。

使用虚拟寄存器的动机有两方面：虚拟寄存器调高了体系结构间的移植性，更重要的是，他们减少了传递过大消息的惩罚（因为虚拟寄存器多于物理寄存器）。

利用寄存器来传递消息的优势随时间而减少，因为体系结构的上下文切换代价成为 IPC 的关键。例如，ARM11 上的寄存器间消息传递相较于（通过内核栈传递）将 IPC 的性能提高了百分之 10，而在 Cortex A9 上，性能提高降至百分之四，在 x86-32 上，保留任何的寄存器来传递消息会破坏编译器对代码的优化。

长消息可以在一个 IPC 中调用多个缓冲区，来分摊上下文切换的成本。长消息可以使用一种简单的拷贝方法：在发送方上下文中执行，内核在发送方建立一个临时的映射窗口，直接复制到接收方缓冲区内。

这种方法可能会在源或目的地址空间上触发缺页错误，这需要嵌套处理异常。此外，处理这样的异常需要调用一个用户态的页表处理函数。这个处理函数必须在内核处理系统调用时启动，处理函数还要认为这个错误发生于用户态执行期间。返回时，原始的系统调用上下文必须重新建立。这样的结果必然给内核带来复杂性，伴随着许多新的错误的产生。

尽管长消息的通信在模拟中可以以少量代价执行，实践中还是很少被使用。共享缓冲区可以避免任何地址空间间的显式赋值，因此它常备用来处理大量消息的传递。除此之外，异步的接口也可以用来传递大量的消息而无需内核中显式的批处理支持。

长 IPC 通信的主要用途在于 POSIX 读写服务器接口，这需要将任意缓冲区的内容发送给不一定有权访问客户端内存的服务器。然而，Linux 作为 POSIX 的中间态，Linux 可以与应用程序共享内存，取代了这一用途。而长 IPC 的其他用途均可被共享存储器取代，因此长 IPC 本身也违背了最小性原则（只谈功能，不谈性能）。

由于这种内核的复杂性和用户态替代方案的存在，我们将长 IPC 从 NVOA、

Fiasco.OC 和 L4embedded 中移除了。

对于 seL4 来说，去除长 IPC 有着更深远的意义：形式化验证要求避免任何并发的操作，而嵌套中断显然不符合这一点。此外，引入长 IPC 会引入新的 c 语言的控制流程，这会增加形式化验证的工作量。当然，内核的缺页问题可以通过检查避免，但这也会引入复杂性（不符合最小化原则），进一步也会导致形式化验证的更加复杂。

OKL4 提供了一种称为新到的异步通信机制。然而，这实际上是为了与早期的 OKL4 微内核兼容而保留的折衷方案，该内核旨在支持内存的嵌入式系统。它被用来将保护边界改造成高度多线程（>50 个线程）的实时应用程序，其中每对通信线程单独的通信页面成本太高。

A.4.2.3 IPC 目的地

尽管 Liedtke 指出端口可以以 12% 的开销实现（主要是 2 个额外的 TLB 未命中）, 最初的 L4 以线程作为 IPC 操作的目标。动机是避免与间接级别相关的高速缓存和 TLB 污染，。该模型要求线程 ID 是唯一标识符

这种模型的缺点是信息隐藏性差。多线程服务必须向客户端进程公开其内部结构，比如包含几个进程、每个线程的 ID 是多少，以分散客户端负载，或者使用网关线程，这可能会成为 IPC 性能瓶颈，并会带来额外的通信和同步开销。有许多建议可以缓解这种情况，但都有缺点。另外大页等机制已经使得间接通信带来的 TLB 污染的问题缓和了很多。

受 EROS 的影响，seL4 和 Fiasco.OC 采用了端点作为 IPC 的目的地，seL4 端点本质上就是端口: 挂起的发送方或接收方队列的头是一个单独的内核对象，而不是 TCB 的一部分。IPC 端点与 mach 端口的不同点事 IPC 不提供任何缓冲区。

A.4.2.4 IPC 超时处理

阻塞的 IPC 通信机制为拒绝服务攻击提供了可能，例如，一个恶意客户端向服务端发送请求和从不尝试收集回复，由于会合式 IPC, 发送方将会永久阻塞，直到监控程序重启服务。L4 的长 IPC 存在一种复杂的攻击方法：恶意客户端向服务器发送长消息，确保出现缺页问题，并阻止异常处理函数处理这个问题。

为了防止此类攻击，最初的 L4 中设计了超时机制，具体来说，IPC 系统调用设置了 4 个超时：一个用于限制发送阶段开始前的阻塞，一个用于在接收阶段限制阻塞，另外两个用于限制在（长 IPC 的）发送和接收阶段期间，页面缺失时的阻塞。

超时时长以浮点方式存储，支持从一毫秒到无穷的等待时间，这种机制增加了唤醒列表的复杂性。

然而，实际上，超时对拒绝服务攻击的防范是没有多大作用的。在一个简单的系统中，超时值的选择没有一种很好的启发式方法，只能使用零或者无穷大：客户端发送和接收的超时时间设为无穷，服务器等待请求的超时时间无穷而回复时间为零。（客户端使用 RPC 风格的调用操作，包括发送，然后是到接收阶段的原子切换，以确保客户端准备好接收服务器的回复。）传统的看门狗计时器代表了检测无响应 IPC 交互（例如，由死锁导致的交互）的更好方法。

放弃长 IPC 后，我们在 L4 嵌入式中使用了一个支持轮询（零超时）或阻塞（无限超时）的标志位取代了超时机制。只需要两个标志位，一个用于发送方，一个用于接收方。seL4 遵循这样的模型，而 OKL4 本身就与超时机制不兼容，且没有拒绝服务攻击的问题。

超时也可以通过等待来自不存在线程的消息的方法来用于定时睡眠，这一功能在实时系统中很有用。德累斯顿试验了包括绝对超时在内的扩展，它在特定的时钟时间到期，而不是相对于系统调用的开始。我们的方法允许用户访问（物理或虚拟的）计时器。

A.4.2.5 通信控制

在最初的 L4 内核中，内核将发送方的 ID 发送给接收方。这种实现方法允许服务器能够通过忽略不重要的消息的方法实现形式自由的访问控制。然而，服务器可能会可能会被恶意程序用消息轰炸。接收此类消息花费的时间会阻止服务器执行其他工作，而检查辨认此类消息又会花费大量时间。因此，这种消息有可能会造成拒绝服务攻击。为了解决这个问题，内核需要支持控制消息的发送。强制的访问控制还需要一种调节和授权的通信机制。

最初的 L4 中，实现了一种部族与酋长的机制：多个进程分成不同的部族，每一个部族中有一个指定的酋长。在部族内部，所有的消息均可以自由的传递，由内核保证消息的完整性。但部族间的通信，都会重定向到酋长处，从而控制信息的流动。这种机制还支持对不可信系统的控制。

Liedtke[5] 认为，部族与酋长机制虽然只在 IPC 中添加了两个周期，但是部族的 ID 编码在线程中可以做到更快。然而，只有直接通信的情况下能够做到低开销。一旦消息被重定向，每次重定向都会在 IPC 通信中多传递两个消息，这是一个很大的开销。除此之外，严格的层次结构在实现上是不灵活的。为了实现强制

访问控制，这种模型方法很快就会给酋长带来很大的负担。这是一种内核的强制策略限制地址空间设计的主要例子。

由于这些缺点，许多 L4 内核并没有实现部族与酋长机制，但这并不是意味着没有办法控制 IPC。有一些更通用的 IPC 重定向模型[16]，但这些实验尚未被证明。这个问题最终通过灵活的带有权限令牌的端点访问控制解决。

A.4.3 用户态设备驱动

最小性原则的关键，或许也是 L4 内核的最激进的创新是所有的设备驱动都变成了用户态的线程。当前，这仍是 L4 内核的标志，形式化验证也推动了这种设计方案的执行：所有任何没有验证的代码，比如驱动代码，添加到内核中都会消除所有的保证，并且在现实系统中验证大量的驱动程序也是不现实的。

少量的驱动仍保存在内核之中。在现代的 L4 内核中，这些驱动通常包括：用于支持抢占调度的时钟，将中断分发给用户态程序的中断控制驱动。

用户态的驱动模型通常将中断与 IPC 消息绑定，内核通过 IPC 将中断发送给驱动。多年来，许多细节发生了变化，但通用的方法仍然适用。

最显著的变化是为了中断的传递，将同步的 IPC 改为了异步的 IPC。这是为了简化实现，因为同步的传递中断需要内核中的虚拟线程作为 IPC 中断源。

用户级应用程序受益于虚拟化驱动的硬件改进。输入输出的内存管理单元（IOMMUs）支持安全地直接被驱动访问。用户态的驱动还收益于减少中断的开销，特别是对当前网络接口的中断合并。在 x86 中，用户态驱动还受益于 TLB 带来的低开销上线文切换。

当然用户级的驱动程序在现在已经成为了主流。在 Linux、Windows、MacOS 上都已经受到了支持。这些系统的 IPC 开销通常高于 L4 内核，但是我们在过去已经证明，即使在 Linux 上，也可以在上下文切换友好的硬件上实现低开销[17]。然而，在实践中，只有一小部分设备对性能至关重要。

A.4.4 资源管理

在最初的 L4 资源管理中，很大程度上基于进程的层次结构。这适用于管理进程和虚拟内存。层次化结构是一种管理和回收资源的一种有效的方法，这种结构还能提供一种约束的子模型（子线程的权限等于或低于父线程），但是这样的开销是很大的。层次结构是一种设计方法，但与微内核并不匹配。

权限令牌可以提供一种摆脱层次结构约束的方法，这也是现在所有 L4 内核采用的访问控制模型。在这里，我们将展示了 L4 内核中最重要的资源管理问题，

以及如何解决这些问题。

A.4.4.1 线程层次结构

进程（本质上是一个页表和许多相关的线程）消耗了很多内核资源。不经过检查就分配线程控制块和页表很容易造成拒绝服务攻击。最初的 L4 内核通过进程层次结构解决了这个问题：任务的 ID 本质上就是权限令牌，允许进程创建、销毁内核对象。

任务 ID 数量有限（大约有几千个），内核最初有着控制权。他们可以被委派，但只能向层次结构上、下传递。在常用的设置中，初始的用户进程获得全部任务 ID, 然后再进一步创建进程。

这种设计方法是不灵活的，最终它被全面的权限令牌替代。

A.4.4.2 递归的页表映射

原始的 L4 内核中将物理内存帧上的全向绑定到了现有的页面映射上。一个有效的映射权限可以允许这个页面映射到另一个地址空间。程序可以通过授权的方法将页面交给其他程序而无需通过映射。地址空间创建时是空的，并使用映射原语填充。

递归的地址空间映射模型基于一个初始的地址空间，原始地址空间接收内核启动后所有的剩余空闲页面的映射。初始地址空间是所有进程的缺页处理控制器的地址空间，每当进程需要它时，就将自己的页面映射到进程地址空间上。

需要注意的是，尽管 L4 的内存模型创建了一个基于每一帧的映射层次结构，它并不强制要求地址空间的层次化：映射是通过 IPC 建立的（类似于通过 IPC 消息传输权限），进程可以将页面映射到 IPC 允许的其他进程（只要对方同意接收）。与 Mach 相比，L4 没有内核对象的语义，只有低级的地址空间管理机制，与用户可见的内存对象抽象相比，这些机制更接近 Mach 的 inkernel pmap 接口[18]。内存对象、写时复制都是用户级创建的抽象或实现方法。

递归映射模型在概念上简洁又优雅，Liedtke 显然对此感到自豪，这种机制在许多论文中都有重要的地位。然而经验表明，这种设计存在着很大的缺陷。

为了支持页面粒度的撤销, 递归的地址空间模型需要大量的日志放在映射数据库中。此外，L4 内存模型的通用性允许两个串行的线程通过递归的将同一帧映射到彼此地址空间中不同页面的方法来迫使内核消耗大量内存，这是一种潜在的拒绝服务攻击，只能通过 IPC 机制（部族与酋长机制）来防止。

在 L4 嵌入式中，我们删除了递归映射模型，因为我们观察到，对于真实世

界的用例，即使没有恶意进程，映射数据库也消耗了 25-50% 的内核内存使用。我们用一个更紧密地反映硬件的模型取代了它：映射总是源自物理内存帧。

这种方法是以丢失细粒度的控制和内存撤销为代价的（而不是通过页面的暴力扫描）, 因此，我们认为这种方法只是一种缓解，OKL4 在一定程度上扩展了这个最小模型，但没有实现原始模型的通用性和细粒度控制。

映射的控制在基于权限令牌的系统中是很容易实现的，只需要使用标准的委派模型[19]。这就是 seL4 所做的事：映射到物理页帧的权利通过权限令牌控制，而不是直接访问虚拟的页表来传递的。因此，seL4 模型不是递归的，即使使用了页面的权限两排，页面也由内核内存模型严格控制，描述如下。

Xen 提供了一个有趣的比较点。授权表允许创建（基于拥有合法的映射）有效的帧的权限，这种权限可以传递到另外一个域来建立共享映射。最近的一项提案中扩展了授权表，允许撤销帧的权限。内存映射的语义与 seL4 语义大致相似，减少了缺页的可能。在 Xen 的情况下，支持细粒度的委托和撤销带来的开销只存在于共享时。

NOVA 和 Fiasco.OC 都保留了递归的地址空间模型，映射的权限由拥有的合法映射确定。Fiasco.OC 中的每个任务内核内存池解决了由此导致的无法限制映射和日志分配的问题。

现有的 L4 地址空间模型代表了通用性和最小性之间的不同全河南，以及潜在的更具空间效率的特定方法。

A.4.4.3 内核内存

虽然权限令牌机制提供了一个干净优雅的委派模型，但它本身不能解决资源管理问题。一个单独具有委派权限的恶意线程仍然可以用令牌创建大量的映射，迫使内核消耗大量内存，由此产生拒绝服务攻击。

L4 内核传统上有一个固定大小的堆，内核从堆中分配内存。最初的 L4 中有一个页面分配器，用户态程序向它发送请求来申请更多的内存。这种方法并不能解决用户态恶意代码的问题，它只是转移了问题，因此这种方法并被大多数 L4 内核采纳。

大多数操作系统都面临的根本问题是，通过共享内核堆的方法无法实现用户程序隔离。一个令人满意的方法必须要满足隔离要求。根本问题是，即使在权限控制系统中，如果权限控制之外还存在着资源，就不可能证明这个系统的安全性。

将内存作为用户态的内容能够部分解决这个问题。虽然基于缓存的方法消除

了拒绝服务攻击的问题。但它不能做到严格的内核内存隔离，这是实时系统中高性能的先决条件，并可能引入隐蔽通道。

Liedtke 研究了这个问题，并提出了每个进程均有一个内核堆的方案。NOVA Fiasco 和 OKL4 都采用了这种方法的变体。每个进程单独一个内核堆的设计模式简化了用户态，但这样的代价是在不破坏进程的情况下撤销了分配的权限。社区仍在探讨这种设计。

seL4 采用了不同的方法，管理内核内存的模型是 seL4 对操作系统内核设计的主要贡献。处于对资源的使用和内存隔离的想法，我们将所有的内核内存都放置在权限的管理之下。具体来说，完全删除了内核栈，并为用户态提供了一种机制，无论何时内核分配数据结构，都可以识别为内核的内存。这降低了内核的大小和复杂性，对验证也有好处。

seL4 的关键是所有的内核对象都是显式的，并且都基于权限访问控制。这种方法受到了基于硬件的权限系统，特别是 CAP[20], 其中硬件解释的权限直接指向内存。HiStar 也明确了所有的内核对象，尽管采用了缓存的方法来管理内存。

当然，用户可见的内核对象并不是意味着内核对象对有权限的人可以直接读、写内存。这种权限提供了应用特定内核对象的方法，包含对象的销毁。至关重要的是，内核对象的类型包括未使用的内存，在 seL4 称为 Untyped，可用于创建其他对象。

具体来说，Untyped 上唯一可能的操作是将其中一部分变成具体的内核对象。新对象与原始 Untyped 的关系记录在能力派生树种，该树记录着其他能力的派生关系，比如创建能力的副本。一旦 untyped 变为具体内核对象，对原始 untyped 可行的操作就只剩下了撤销派生对象。

通过 untyped 是唯一能够生成内核对象的方法。因此，可以通过对 untyped 内存访问的限制来控制系统可控制的资源分配。创建新对象可以生成更小的 untyped，在对小的 untyped 进行独立管理——这是委派资源管理的关键。Untyped 方法保证了内核的完整性，没有两个 untyped 会存在重叠关系。

表A.3 描述了 seL4 对象的集合以及具体用途。用户态只能访问与其地址空间映射的帧相对应的存储器。

表 A.3　seL4 内核对象

内核对象	描述
TCB	线程控制块
CNode	存放各种权限
同步端点	用于同步的进程间通信
异步端点	类似端口的异步 IPC
页表	根页表
页表项	页表页节点
帧	虚拟地址空间
Untyped	可分配的物理地址空间

seL4 模型具有以下属性：

1. 所有权限都有明确的授权
2. 数据的访问与授权均可以被控制
3. 内核本身遵守分配给应用程序的权限，包括物理内存的使用
4. 每个内核对象均可独立于其他内核对象进行回收
5. 所有的操作都在短时间内完成或可抢占

一到三点属性确保了系统资源可安全性推理。特别是对于属性 3，对于正式的证明内核的完整性、权限限制和机密性至关重要[21]. 属性 5 确保了所有的内核延迟都是有界的，支持硬实时系统[15]。

属性 4 保证了内核的的完整性。任何应有权限的程序都可以在任何时候回收一个对象。例如，可以回收页表内存，而不必破坏相应的地址空间。这要求内核能够检测正在回收内核对象的无效引用。

在能力推导树的帮助下，该需求得到了满足。通过调用树上更远的 Untyped 对象上的 revoke 方法来撤销对象；这将删除从 Untyped 派生的所有对象的所有功能。当对象的最后一个功能被删除时，对象本身也被删除。这消除了它可能与其他对象之间的任何内核内依赖关系，从而使它可以重复使用。删除最后一个功能很容易检测到，因为它会清除功能树中引用特定内存位置的最后一个叶节点。

撤销的方法需要用户态的日志来将 untyped 权限与内核对象连接器起来，同常是在用户态上定义更高级别的抽闲。Untyped 的精确语义与日志的关系仍在探索中。

A.4.4.4 时钟

除去内存以外，系统中另一个关键的资源是 cpu，与内存不同，内存可以在多个进程之间同时有效共享，而 cpu 一次只能单独一个线程使用，因此必须时间复用。

所有的 L4 均采用了固定策略调度器的方法来实现复用。原始 L4 内核的调度模型是硬优先级循环，现在仍被使用，尽管它严重违反了微内核政策自由的设计原则。过去所有的将调度模块写入用户态的方法均失败了，原因在于开销过大。

德累斯顿集团对实时性问题做了广泛的研究，包括绝对的超时。他们还探究了几种调度方法以及适用的实时系统结构，并分析了基于 L4 的实时系统[22]。

虽然能够解决一些具体的问题，德累斯顿并没有定制一个通用的标准。Fiasco.OC 基本恢复了传统的 L4 模式。最近关于调度上下文的提议允许将基于分层优先级的调度映射到单个优先级调度器[23]。虽然很有希望，但这还不能处理实时社区中使用的广泛的调度方法，尤其是最早最后期限优先（EDF）调度。

有人可能会争辩说，适用于所有目的的单一通用内核的概念可能不再像以前那样重要了——如今我们已经习惯了特定于环境的插件。然而，seL4 的正式验证对更改内核产生了强烈的抑制作用，它强烈增强了为所有使用场景提供单一平台的愿望。因此，处理时间问题的政策自由方法也很有必要。

A.4.4.5 多核

早在 L4 社区中就已经探讨了多处理器问题。L4/Alpha 和 L4/MIPS 的大部分工作都是在 x86 平台上完成的，因为 x86 平台是最早的最便宜的多处理器。早期的 x86 多处理器和多核具有较高的核间通信成本，并且没有共享缓存。因此，标准的方法是每个处理器一个调度队列，线程的迁移只在用户明确请求时发生。Uhlig 探讨了多核平台上的锁定、同步和一致性问题，并开发了基于 RCU 的内核数据结构[24]。NOVA 和 Fiasco.OC 广泛使用了 RCU。

随着重点从高端服务器转移到嵌入式和实时平台，多处理器问题退居幕后，直到最近嵌入式处理器的多核版本出现才重新出现。它们的特点是低的核心间通信成本和通常共享的 L2 缓存，这意味着与 x86 上的权衡不同。由此产生的低迁移成本通常不能证明迁移线程的系统调用的开销是合理的，全局调度程序更有意义。

形式化验证引入了新的约束条件。并发性给验证带来了巨大的挑战，我们尽可能地将其排除在内核之外。对于多核，这意味着采用大内核锁或多内核方法[8]。

对于系统调用很短的微内核来说，前者并不像一开始看起来那么愚蠢，因为锁争用会很低，至少对于共享 L2 的少数内核来说是这样。

我们目前正在探索的方法是集群多内核，大锁内核（跨共享二级缓存的内核）和多内核的受限变体（内核之间不允许内存迁移）的混合[25]。集群多内核避免了大多数内核代码中的并发性，这使得一些形式保证能够在某些假设下继续保持。最重要的假设是（1）锁本身及其外部的任何代码都是正确的且无竞争的，以及（2）内核对内核和用户级别之间共享的内存的任何并发更改都是鲁棒的（对于 seL4，这只是一个虚拟 IPC 消息寄存器块）。

这种方法的吸引力在于它保留了现有的单处理器证明，只做了很小的修改。我们已经正式取消了单处理器自动机的并行组合，并证明了精化仍然有效。缺点是形式保证不再覆盖整个内核，提升框架使用的大步语义阻止了形式框架的进一步扩展，以涵盖关于锁的正确性、用户内核并发性和资源迁移限制的任何放松的推理。

集群多内核的变体最终可能是获得多处理器内核的完全形式验证的最佳方法，尽管我们在这里没有进行明确表示。在形式验证方面需要做更多的工作来解释微内核规模的细粒度性。

A.5 微内核实现

Liedtke 列出了一组设计决策和实现技巧，这些决策和技巧有助于在最早的 i486 版本中实现快速 IPC，尽管其中许多都有优化过早的味道。

有些已经提到了，比如长 IPC 中使用的临时映射窗口。其他的则没有争议，比如单个系统调用中的发送和接收组合（类似 RPC 调用的客户端调用服务器应答风格）。我们将详细的讨论剩下的方法，包含一些传统的 L4 的实现，这些方法很少被公开，但在 L4 社区内一直被当做理所当然。

A.5.1 严格的进程定位和虚拟线程控制块列

最初的 L4 中每个线程都有一个单独的内核堆栈，分配在同一页的 TCB 上。因此 TCB 的基址与堆栈的基址偏移量是固定的。可以通过低几位置零的方法来获得堆栈起始地址。只需要一个 TLB 表项就可以涵盖线程控制块和堆栈。

此外，所有的 TCB 都分配在一个稀疏的、虚拟寻址的数组中，由线程 ID 索引。在 IPC 期间，这可以非常快速地查找目标 TCB，而无需首先检查 ID 的有效性：如果调用方提供了无效的 ID，则查找可能会访问未映射的 TCB，从而触发页

面故障，内核通过中止 IPC 来处理该故障。如果没有发生故障，则可以通过将调用方提供的值与 TCB 中找到的值进行比较来确定线程 ID 的有效性。（原始 L4 的线程 ID 有版本号，当线程被销毁和重新创建时，版本号会发生变化。这样做是为了使线程 ID 在时间上唯一。在 TCB 中记录当前 ID 可以检测过时的线程 ID）。

这两个特性都是有代价的：许多内核堆栈占据了每线程内存开销的主导地位，同时也增加了内核的缓存占用。虚拟 TCB 阵列增加了内核的虚拟内存使用，从而增加了 TLB 占用空间，但避免了查找表所需的额外缓存占用空间。具有单一页面大小和未标记 TLB 的处理器除了对数据结构进行分组以最大限度地减少所接触的页面数量之外，几乎没有机会进行优化。然而，RISC 处理器具有较大的页面大小（或物理内存寻址）和标记的 TLB，这改变了权衡

当 L4 在嵌入式空间获得吸引力时，内核的内存使用成为了一个重要问题，因此需要重新审视设计。

初步试验使用了单内核栈的奔腾处理器，实验表明内核的内存开销减少了，IPC 的性能在微基准下得到了改善。在 ARmv5 处理器上，Pistachio 对内核的进程和基本事件进行了全面的性能评估。他在微基准上得到了类似的性能，但在多任务的负载上，性能优势增加了 20%。实验还发现，每线程的内存使用量是内核进程的四分之一。

同时，Nourai 分析了 TCB 在虚拟寻址与物理寻址上的权衡。他在 Pistachio 上实现了物理寻址，尽管是在 MIPS64 处理器上。他发现微基准测试中 IPC 的性能几乎没有差异，但是物理寻址在强调 TLB 工作负载上的性能表现更好。MIPS 使用了 MMU 支持物理寻址，但在大多数其他架构上，物理寻址是通过大页面映射的方法模拟的。尽管如此，Nourai 仍然证明了虚拟寻址的 TCB 对性能并没有显著的好处。

基于事件的内核避免了内核的缺页错误，从而保留的 C 语言的语义，保证 C 语言的语义范围降低了验证的复杂性。

总之，这些结果使我们选择了一种基于事件的设计，其中包含 L4 嵌入式的物理寻址内核数据，seL4 也紧随其后。虽然这一决定最初是由资源匮乏的嵌入式系统的现实和后来的验证需求驱动的，但该方法的好处并不局限于这些情况，我们认为它通常是现代硬件上最好的方法。

A.5.2 懒调度

在 IPC 的会合模型中，线程的状态频繁在运行与阻塞间切换。这意味着频繁地将线程插入、删除队列。

Liedtke 的懒调度技巧减少了队列操作的次数。当线程阻塞在 IPC 中是，线程会在 TCB 中更新状态，但并不会将其移出队列，因为阻塞可能很快结束。但调度程序的时间片耗尽时，他会遍历就绪队列，找到真正可以运行的线程，并删除不可运行的。

懒调度将高频操作改为了不太频繁的调用。我们在实时系统中观察到了懒调度的缺点：调度器的执行时间受线程数量的限制。

为了解决这个问题，我们采用了一种替代优化，称为 Benno 调度，它不受病态时序行为的影响: 就绪队列包含除当前执行的线程之外的所有可运行线程。因此，当线程在 IPC 期间阻塞或取消阻塞时，就绪队列通常不会被修改。在抢占时，内核将被抢占的线程（仍然可以运行，但不再执行）插入到就绪队列中。取消超时意味着不再需要操作唤醒队列。端点的等待队列必须严格的维护，在通常情况下，他们在缓存中常被使用，所以这些队列操作的成本也很低。这种方法与懒调度的性能相当，但是有着有限的最坏执行时间。

A.5.3 直接进程切换

传统的 L4 设计中，尽量避免在进程间通信时发生调度。如果线程在 IPC 调用期间被阻塞，内核切会切换到一个易于识别的可运行线程，这个线程在原来的时间片上继续运行。这种方法叫做直接进程切换。

当服务器与客户端有相同优先级时，一方面，如果客户端线程执行调用操作，客户端就会被阻塞，直到服务器回复。为了能够执行系统调用，线程必须时优先级最高的可运行线程，观察优先级的最好方法时确保被调用者的尽快完成。另一方面，如果服务器对等待的客户端进行，并且服务器有来自另一个客户端的请求在等待，则通过执行其 IPC 的接收阶段来继续服务器利用已准备好的缓存是有意义的。

现代的 L4 内核关注争取的实时性行为，在符合优先级的地方保留了直接进程切换的方法，否则则使用调度器，事实上，直接进程切换是时间片捐赠的一种形式，它可以用来实现优先级继承、优先级翻转写一。

A.5.4 抢占

传统上，L4 实现在内核内执行时会禁用中断，尽管有些（如 L4/MIPS）在长期运行操作中包含抢占点，在这些操作中会短暂启用中断。这种方法大大简化了内核的实现，因为大多数内核不需要并发控制，并且通常可以获得更好的平均情况性能

然而，最初的 L4 ABI 有许多长期运行的系统调用，早期的 Fiasco 工作使内核完全可抢占，以提高实时性能。后来的 ABI 版本删除了大部分长时间运行的操作，Fiasco.OC 恢复到最初的、基本上不可抢占的方法。

在 seL4 中，不可抢占的内核设计还有一个额外的愿意：避免出现并发，导致形式化验证变得难以处理。考虑到 seL4 对安全关键系统的关注，其中许多系统具有严格的实时性，我们需要对中断传递的延迟进行严格限制。因此，必须避免长时间运行的内核操作，并在不可能的情况下使用抢占机制。我们在抢占点的放置位置上做出了巨大的努力。

需要注意的是，基于延续的事件内核为抢占点提供了结构支持。

A.5.5 不可移植性

Liedtke 认为，微内核实现不应追求可移植性，因为硬件抽象引入了开销并隐藏了特定于硬件的优化机会。他引用了“兼容”的 i486 和奔腾处理器之间细微的体系结构变化，这导致了权衡的变化，并暗示了优化方案会发生重大变化。Liedtke 本人用 Pistachio 论证了一点：仔细的设计和实现使得开发一个性能达到 80%-90% 与体系结构无关内核是可能的。

在 seL4 中, 架构无关的代码仅占 50% 左右，一半的架构相关代码涉及虚拟内存管理，这在不同体系结构中必然不同。可移植代码的比例低是 seL4 体量小造成的，大量可移植的代码已经迁移到用户态了。

A.5.6 非标准调用约定

最初的 L4 内核完全是在汇编程序中实现的，因此函数的调用约定在内核内部是不相关的。在 ABI 中，所有不需要作为系统调用参数的寄存器都被指定为消息寄存器。库接口提供了内联汇编程序存根，以将编译器的调用约定转换为内核 ABI。

在下一代 L4 内核中，部分用 C 代码编写，使用 C 后，内核必须重新使用 C 编写器的调用约定，并在返回时恢复内核的约定。这使得调用 c 函数的成本很高，

因此除了必须操作外，不鼓励使用 c 语言。

在后来的内核中基本完全使用 c 语言，调用约定不匹配的代价意味着 c 代码没有和汇编一样的高性能。因此，内核实现着开始着手部分汇编程序工作来完成快速路径。这使得 IPC 性能与最初的 L4 相当。

传统的方法不适合 seL4，因为验证框架只能处理 c 代码，我们希望尽可能完整的验证内核的功能。这需要将汇编代码限制在最低限度，并排除调用约定转换这一操作，迫使我们采用标准的调用约定。

A.5.7 实现所用的语言

seL4 依赖快速路径来获得强有力的 IPC 性能，但快速路径必须在 c 中实现。由于汇编代码的高维护成本，汇编程序实现的快速路径已经被商业实现中抛弃了。对于 seL4 来说，与同一架构上的最快内核相比，我们愿意容忍 IPC 性能下降不超过 10%。

幸运的是，事实证明，精心设计的快速路径，可以实现极具竞争力的 IPC 延迟。这意味着手动重新排序语句，使用编译器无法通过静态分析确定的（已验证的）不变量。

在 ARM11 处理器上的单向 IPC 比最快的 IPC 高出 10%，这是因为简化了 seL4 ABI 和 IPC 的语义，以及基于事件的内核不再需要保存和恢复 C 的调用约定。我们还做了编译期相关的分支优化，这有利于代码的局部性。

在任何情况下，结果都表明了汇编器不再成为性能的约束条件。事实上，德累斯顿发现他们在没有任何快速路径的情况下也可以做到高效的 IPC 通信。

第一个完全用高级语言编写的 L4 内核是 Fiasco，它选择了 C++ 而不是 C（几年前已经用于 MIPS 内核的部分）。考虑到 C++ 编译器当时的状态，这似乎是一个勇敢的决定，但至少部分原因是 Fiasco 最初的设计并没有考虑到性能。这一变化以及最近 Fiasco 的经验表明，如果使用得当，C++ 代码不会对性能造成影响。

卡尔斯鲁厄团队也使用了 c++ 语言开发 Pistachio，主要是为了支持可移植性。尽管德累斯顿和卡尔斯鲁厄对 c++ 有着高度的热情，但我们从未看到 c++ 为微内核实现提供了任何令人信服的优势。此外，OK 实验室发现，良好的 c 编译期的可用性在嵌入式领域是一个真正的问题，他们将自己的版本换为了 c 语言。

对于 seL4，形式化验证的要求迫使选择 C。虽然德累斯顿的 VFiasco 项目试图验证 C++ 内核，但它从未完成 Fiasco 使用的 C++ 子集的语义形式化。相反，通过使用 C 来实现 seL4，我们可以在现有的 C 形式化的基础上进行构建，这是验

证的关键促成因素。

A.6 结论

很少有一个操作系统有着开发者社区又有很好的商业部署。L4 就是这样一个系统，它经历了 20 年的 API 演变。我们认为这是一个很好的机会，可以反思经得起时间考研的原则和专业知识，以及那些在演变中幸存下来的东西。

设计选择中，各种方案都经历了多次尝试，然而 sel4 中最普遍的原则最小化，在开发者眼中仍具有重要意义。具体而言，我们发现关键的微内核性能指标 IPC 延迟基本上没有变化（就时钟周期而言），只要不同的 ISAs 和微架构之间的比较具有任何有效性，这与 Ousterhout 在 L4 创建前几年确定的趋势形成了鲜明对比。此外，也许最令人惊讶的是，代码大小基本上保持不变，这在软件系统中是一个相当不寻常的发展。

形式化的验证增加了最小化的重要性，也增加了简化实现的成本。一些设计决策中，如简化的消息结构、内核内存的用户级控制和多核方法，都受到验证的强烈影响。它还影响了许多实现方法，例如使用面向事件的内核、采用标准调用约定以及选择 C 作为实现语言。然而，我们不认为这导致了我们在忽视核查时会认为较差的权衡。

经过正式验证，L4 令人信服地实现了微内核多年前做出的核心承诺之一：鲁棒性。我们认为，这是对 Liedtke 最初 L4 设计辉煌的有力证明，这是在保持或可能是由于忠于最初 L4 理念的情况下实现的。这可能花了很长时间，但时间终于证明了 Brinch Hansen 曾经激进的想法是正确的。

到目前为止，有一个概念一直不能让人满意：时间。L4 内核仍然实现特定的调度策略，在大多数情况下是基于优先级的循环，这是内核中的一个最主要的策略。德累斯顿和 NICTA 正在进行的工作表明，单个参数化内核调度器实际上可能能够支持所有标准的调度策略，我们预计这将不需要 20 年的时间。

A.7 致谢

NICTA 由澳大利亚政府通过 ICT 卓越中心计划资助，由宽带、通信和数字经济部以及澳大利亚研究委员会代表。

L4 如果没有它的发明者 Jochen Liedtke 就不会存在，我们向他的才华致敬。我们也非常感谢 20 多年来为 L4 做出贡献的许多人，感谢 IBM Watson、TU Dresden、

卡尔斯鲁厄大学、新南威尔士大学和 NICTA 的几代员工和学生；太多了，无法一一列举

我们特别感谢对本文草稿提供反馈的 L4 社区成员：Andrew Baumann、Ben Leslie、Chuck Gray 和 Hermann Hartig。我们感谢 Adam Lackorzynski 挖掘出原始 L4 来源并提取 SLOCounts，以及 Adrian Danis 在最后一刻对 seL4 进行的一些优化和测量。我们感谢匿名 SOSP 评审员的真知灼见，感谢我们的指导者 John Ousterhout 的反馈。

参考文献

[1] J. Liedtke. Improving ipc by kernel design. In *ACMPUB27New York, NY, USA*, 1993.

[2] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Acm Symposium on Operating Systems Principles*, pages 132–140, 1975.

[3] M. Condict, D. Bolinger, E. Mcmanus, D. Mitchell, and S. Lewontin. Microkerael modularity with integrated kernel performance. 1994.

[4] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Symposium on Operating Systems Principles*, 1993.

[5] J. Liedtke. On μ-kernel construction. In *Proc Acm Symposium on Operating System Principles*, 1995.

[6] J. Liedtke. Toward real microkernels the inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems. 1996.

[7] G. Klein, Kevin John Elphinstone, G. Heiser, J. Andronick, David A Cock, P. Derrin, D. Elkaduwe, E. Kai, R. Kolanski, and M. Norrish. sel4: formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009.

[8] A. Baumann, P. Barham, Pierre Variste Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009.

[9] J. Liedtke. A persistent system in real use - experiences of the first 13 years. In *International Workshop on Object Orientation in Operating Systems*, 2002.

[10] G. Heisery. Achieved ipc performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, 1997.

[11] J. M. Smith and D. J. Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.

[12] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Massachusetts Institute of Technology*, 1965.

[13] G. Heiser and B. Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Acm Sigcomm Asia-pacific Workshop on Systems*, 2010.

[14] D. Elkaduwe, P. Derrin, and K. Elphinstone. *Kernel design for isolation and assurance of physical memory*. ACM, 2008.

[15] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Real-time Systems Symposium*, 2012.

[16] T. Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko, and Y. Park. Flexible access control using ipc redirection. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, 1999.

[17] BenLeslie, PeterChubb, NicholasFitzroy-Dale, StefanG(o)tz, CharlesGray, LukeMacpherson, DanielPotts, Yue-TingShen, KevinElphinstone, and GernotHeiser. User-level device drivers: Achieved performance. 计算机科学技术学报：英文版, 20(5):11, 2005.

[18] R. Rashid and A. Tevanian. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, 1988.

[19] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM (JACM)*, 1977.

[20] R. M. Needham and Rdh Walker. The cambridge cap computer and its protection system. *Acm Sigops Operating Systems Review*, 11(5):1–10, 1977.

[21] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[22] Hermann Hrtig and M. Roitzsch. Ten years of research on l4-based real-time systems. *Proceedings of Theghth Real Time Linux Workshop*, 2008.

[23] A Lackorzyński, A. Warg, M. V?Lp, and H. H?Rtig. Flattening hierarchical scheduling. In *Tenth Acm International Conference on Embedded Software*, page 93, 2012.

[24] P. E. Mckenney, A. Kleen, O. Krieger, R. Russell, and M. Soni. Read-copy update. In *Ottawa Linux Symposium, 2001*, 2001.

[25] M. V. Tessin. The clustered multikernel: An approach to formal verification of multiprocessor os kernels. *ws on systems for future multi*, 2012.

书面翻译对应的原文索引

[1] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.