ReL4：高性能异步微内核设计与实现

廖东海

**** 年 * 月

ReL4：高性能异步微内核设计与实现

廖东海

北京理工大学

中图分类号： TQ028.1

UDC分类号： 540

★ 特别类型
□ 交叉研究方向
□ 政府项目留学生

ReL4：高性能异步微内核设计与实现

作 者 姓 名 廖东海

学 院 名 称 计算机学院

指 导 教 师 陆慧梅

答辩委员会主席 ***

申 请 学 位 工学硕士

学 科 / 类 别 计算机科学与技术

学 位 授 予 单 位 北京理工大学

论 文 答 辩 日 期 **** 年 * 月

ReL4: Design and Implementation of High-performance Asynchronous Microkernel

Candidate Name:	Liao Donghai
School or Department:	****
Faculty Mentor:	Lu Huimei
Chair, Thesis Committee:	***
Degree Applied:	****
Major:	****
Degree by:	Beijing Institute of Technology
The Date of Defence:	*, ****

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下独立完成的研究成果。文中所撰写内容符合以下学术规范（请勾选）：

□ 论文综述遵循“适当引用”的规范，全部引用的内容不超过50%。

□ 论文中的研究数据及结果不存在篡改、剽窃、抄袭、伪造等学术不端行为，并愿意承担因学术不端行为所带来的一切后果和法律责任。

□ 文中依法引用他人的成果，均已做出明确标注或得到许可。

□ 论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

□ 与本人一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此声明。

签　　名：　　日期：

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：

① 学校有权保管、并向有关部门送交学位论文的原件与复印件；

② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；

③ 学校可允许学位论文被查阅或借阅；

④ 学校可以学术交流为目的，复制赠送和交换学位论文；

⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

签　　名：　　日期：

导师签名：　　日期：

摘要

微内核将大部分系统服务运行在用户空间，相比于宏内核有更好的稳定性、扩展性和内核安全性，在微内核系统中，应用程序通过进程间通信 (IPC) 而非系统调用来请求服务，频繁的 IPC 造成的特权级切换将产生巨大开销，成为了系统的性能瓶颈。以 seL4 为代表的现代微内核将同步 IPC 作为主要通信手段，并以异步通知机制作为辅助手段来提升系统并发度，这些机制在一定程度上减少了 IPC 的开销，提升了系统性能，然而它们在设计上仍有三点不足：1）在支持同步 IPC 的情况下冗余地支持了异步通知机制，这违反了内核最小化原则；2）通知机制依赖内核的转发，会造成大量的特权级切换；3）系统调用和同步 IPC 会导致无关请求顺序执行，无法充分利用硬件资源。

针对以上三点缺陷，本文设计并实现了 ReL4——一套异步化的高性能微内核架构，其主要设计理念是将异步通知机制作为内核支持的唯一通信手段，在用户态通过共享缓冲区实现 IPC 的数据传递，并通过通知机制进行同步，保证了内核最小化原则；同时 ReL4 基于用户态中断，在兼容 seL4 接口的基础上，设计了无需内核转发的 U-notification 机制，避免了通知机造成的大量特权级切换；最后，ReL4 通过异步运行时来实现异步 IPC 和异步系统调用，避免了无关 IPC 和系统调用请求的顺序执行，在充分利用硬件资源，提升系统并发度的同时，提升系统易用性。

本文在 FPGA 上实现了 ReL4 的原型系统，将 U-notification、异步 IPC 和异步系统调用分别与 seL4 进行了对比测试，并评估了真实的 TCP Server 在 ReL4 的性能表现。测试结果表明，相比于 seL4，ReL4 能够大幅减少系统中特权级切换的次数，在低并发场景下有着接近的性能，在高并发场景下拥有更卓越的表现，从而证明了 ReL4 架构的可行性和优越性。

关键词：微内核；异步；进程间通信；用户态中断

Abstract

The microkernel runs most system services in the user space. Compared with the monolithic kernel, it has better stability, extensibility, and kernel security. In a microkernel system, applications request services through inter-process communication (IPC) instead of system calls. The frequent privilege level switches caused by IPC generate huge overheads, which become a performance bottleneck of the system. Modern microkernels represented by seL4 use synchronous IPC as the main communication means and adopt an asynchronous notification mechanism as an auxiliary means to improve the system concurrency. These mechanisms reduce the IPC overhead to a certain extent and enhance the system performance. However, there are still three design deficiencies: (1) The asynchronous notification mechanism is redundantly supported while synchronous IPC is supported, which violates the principle of kernel minimization; (2) The notification mechanism relies on the forwarding of the kernel, resulting in a large number of privilege level switches; (3) System calls and synchronous IPC will cause irrelevant requests to be executed sequentially, failing to fully utilize hardware resources.

Aiming at the above three defects, this paper designs and implements ReL4, a set of asynchronous high-performance microkernel architecture. Its main design concept is to use the asynchronous notification mechanism as the only communication means supported by the kernel. In the user space, the data transfer of IPC is realized through a shared buffer, and synchronization is carried out through the notification mechanism, ensuring the principle of kernel minimization. At the same time, based on user-space interrupts and on the basis of being compatible with the seL4 interface, ReL4 designs the U-notification mechanism that does not require kernel forwarding, avoiding a large number of privilege level switches caused by the notification mechanism. Finally, ReL4 implements asynchronous IPC and asynchronous system calls through an asynchronous runtime, avoiding the sequential execution of irrelevant IPC and system call requests. While fully utilizing hardware resources and improving system concurrency, it also enhances the usability of the system.

This paper implements a prototype system of ReL4 on an FPGA. The U-notification, asynchronous IPC, and asynchronous system calls are respectively compared and tested with

seL4, and the performance of a real TCP Server on ReL4 is evaluated. The test results show that compared with seL4, ReL4 can significantly reduce the number of privilege level switches in the system, has a similar performance in low-concurrency scenarios, and has more excellent performance in high-concurrency scenarios, thus proving the feasibility and superiority of the ReL4 architecture.

Key Words: microkernel; asynchronous; inter-process communication; user-mode interrupt

目录

插图

表格

主要符号对照表

BIT	北京理工大学的英文缩写
LaTeX	一个很棒的排版系统
LaTeX 2$_\varepsilon$	一个很棒的排版系统的最新稳定版
XeTeX	LaTeX 的好兄弟，事实上他有很多个兄弟，但是这个兄弟对各种语言的支持能力都很强
ctex	成套的中文 LaTeX 解决方案，由一帮天才们开发
H_2SO_4	硫酸
$e^{\pi i} + 1 = 0$	一个集自然界五大常数一体的炫酷方程
$2\,H_2 + O_2 \longrightarrow 2\,H_2O$	一个昂贵的生成生命之源的方程式

第 1 章　绪论

1.1　课题研究的背景和意义

随着科技的发展，微内核广泛应用于工控系统、嵌入式系统等领域。相比于宏内核，微内核将内存管理、设备驱动、文件系统等与核心功能分离，运行在用户空间，这种隔离机制使得单个服务的故障不会直接影响到内核和其他服务，从而提升了系统的整体稳定性[1]；微内核通过精简核心功能，减少攻击面，从而提升内核安全性；此外，模块化的设计使得系统更易于维护和升级。在微内核系统中，应用程序通过进程间通信 (IPC) 而非系统调用来请求服务，这或许能够满足早期性能不敏感的软件需求，然而在对软件性能有着更高要求的今天，频繁 IPC 造成的特权级切换将产生巨大开销，成为系统的性能瓶颈[2]。

30 年前 Liedtke 提出的 L4[3] 重新设计了微内核系统，通过组合系统调用、快速路径、消息寄存器等优化手段，从硬件层到软件层对 IPC 进行了系统性优化，证明了微内核的 IPC 也可以很快，之后以 seL4[4] 为代表的现代微内核的 IPC 框架也基本延续了 L4 的设计理念，以同步 IPC 作为主要的通信方式。然而同步 IPC 迫使单线程中上下文无关的请求以顺序的形式执行，系统只能通过多线程实现并发，而为了更好地利用硬件资源，现代微内核大多引入异步的通知机制来简化并发程序设计，提升多核的利用率，这违反了微内核的最小化设计原则，增加了内核的复杂性。

而随着软件复杂性的提升，用户希望系统级软件如数据库管理系统、网络服务器等能够快速处理大量系统调用和 IPC[5]，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，特权级切换成为性能瓶颈。此外，新出现的硬件漏洞如 Meltdown[6] 和 Spectre[7] 漏洞促使内核使用 KPTI 补丁[8] 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。最后，现代微内核的外设驱动往往存在于用户态，外设中断被转化为异步通知，需要用户态驱动主动陷入内核来进行接收，这在一定程度上成为了外设驱动的性能瓶颈[9]。综上所述，由 IPC 和通知机制引起的特权级切换已经成为制约系统性能的主要因素。

本文提出 ReL4，一个用 Rust 编写的高性能异步微内核，它将同步 IPC 从内核中移除，基于用户态中断技术设计了无需陷入内核的 U-notification 机制，在兼容 capability 机制的基础上改造微内核的通知机制，并利用改造后的 U-notification 和异步化编程

设计和实现了一套绕过内核的异步 IPC 和异步系统调用框架。ReL4 在设计理念上将内核最小化原则贯彻得更加彻底，并通过软硬件协同的方式进一步提升微内核的 IPC 性能，为下一代微内核的发展指出一个可能的方向。

1.2 国内外研究现状及发展趋势

1.2.1 微内核 IPC 的发展现状

现代微内核的大部分 IPC 优化始于 Liedtke 提出的 L4，由于之前的微内核 IPC 存在性能瓶颈，L4 从硬件优化、系统架构、软件接口的各个方面对 IPC 进行了重新设计。其中的优化角度可以简单划分为内核路径优化和上下文切换优化。

对于内核路径优化，L4 通过物理消息寄存器来传递短消息，从而避免了内存拷贝，然而随着访存速度的加快，消息寄存器的零拷贝优化带来的收益逐渐减弱，使用物理寄存器导致的平台依赖和编译器优化失效反而限制了系统的性能[10]，因此物理的消息寄存器逐渐被现代微内核以虚拟消息寄存器代替；此外，L4 使用临时映射的形式来进行长消息的传递，避免多余的内存拷贝，但却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[10]，现代微内核一般放弃了这个优化；针对常用且普遍的 IPC 场景，L4 设计了专门的快速路径，避免了复杂繁琐的参数解析和任务调度，然而该优化手段对消息长度、任务优先级有着严格的限制，也无法对多 CPU 核心进行支持。

对于特权级的切换优化，L4 使用的物理消息寄存器在一定程度上减少了上下文切换的开销，但其副作用超过了优化收益导致其被虚拟消息寄存器代替[10]；同时 L4 敏锐地观察到大部分 IPC 通信遵循 C/S 模型，因此通过组合系统调用的形式，将 Send + Reply 组合为 Call，将 Reply + Recv 组合成 ReplyRecv，从而减少了特权级切换的频率，该优化至今作为现代微内核的重要优化手段，但仍然无法避免特权级的切换；此外，L4 通过通过 ASID 机制，在快表项中维护地址空间标识符，减少快表冲刷的频率，缓解了快表污染的问题，然而依然无法避免特权级切换带来的快表污染和缓存失效。

除此之外，L4 仅支持同步 IPC，对于多核架构，同步 IPC 会导致服务调用被顺序执行，导致资源浪费。其次，同步 IPC 强制用户态以多线程的形式处理并发请求，导致了线程同步的复杂性。现代微内核在内核中引入异步通知机制，简化并发编程模型，却使得内核功能冗余，违反了内核最小化原则。

操作	占比
保存和恢复上下文	14.5%
地址空间切换	59.4%
fast - path 检查	20.1%
消息拷贝	1.5%
其他	4.5%

表 1.1　seL4 一次 IPC 中各操作的开销占比

总而言之，现代微内核在单核环境下的 IPC 内核路径上的优化已经较为完善，在最理想的情况下仅需要两次特权级切换，然而对多核环境下，由于需要核间中断，IPC 无法进入快速路径，导致多核下的 IPC 内核路径依旧冗长。而现代微内核在特权级切换的优化方面仍然停留缓解的层面上，导致特权级切换会成为 IPC 的性能瓶颈。

1.2.2　特权级切换

特权级的开销主要分为两部分，一部分是直接开销，包括了保存上下文带来的额外指令开销，另一部分是间接开销，地址空间切换所引起的缓存污染会导致 CPU 执行效率降低。表1.1展示了在 FPGA 上部署的 seL4 一次 Call IPC 操作所带来的开销占比，大部分的开销都花费在地址空间切换上，其次是上下文的切换和 fast-path 检查，如果 fast-path 检查失败，将会进入 slow-path 进行更加冗长的解码和调度流程，进一步降低 IPC 性能。

本文聚焦现代微内核架构设计中的特权级切换开销，旨在设计一种新型的 IPC 架构，减少甚至消除 IPC 和系统调用中的特权级切换，先前已经有大量的工作从软硬件的角度致力于减少特权级切换开销。

从硬件出发的角度出发，大多数工作通过设计特殊的硬件或者特殊的指令来绕过内核实现 IPC。如 SkyBridge[11] 允许进程在 IPC 中直接切换到目标进程的虚拟地址空间并调用目标函数，它通过精心设计一个 Root Kernel 提供虚拟化的功能，通过 VMFUNC 地址空间的直接切换，并通过其他一系列软件手段来保证安全性，但这种方案仅适用于虚拟化环境中。XPC[12] 则直接使用硬件来提供一个无需经过内核的同步功能调用，并提供一种新的空间映射机制用于调用者与被调用者之间的零拷贝消息传递，然而该方案没有相应的硬件标准，也没有一款通用的处理器对其进行支持。这些方法都基于特殊的环境或者没有标准化的硬件来实现，适用范围有限。

从软件出发的角度出发，相关工作主要分为两类：第一类方法通过将用户态和内

优化方法	详细分类	实例	缺点
减少内核路径	临时地址映射	[3]	上下文切换开销已经成为性能瓶颈
	快速路径	[3, 4, 24, 25, 26]	
	消息寄存器		
减少上下文切换开销	消息寄存器	[3, 4, 24, 25, 26]	
	组合系统调用		无法从根本上消除切换开销
	ASID 机制	[4]	
	统一地址空间	[13, 14, 15, 16, 17, 18]	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	[19]	
硬件优化	虚拟化指令	[11]	仅适用于虚拟化环境
	直接硬件辅助	[12]	没有硬件标准，没有通用硬件的支持
		用户态中断	—

表 1.2　国内外研究现状汇总

核态的功能扁平化来减少内核与用户态的切换开销，如 unikernel[13–15] 将所有用户态代码都映射到内核态执行，Userspace Bypass[16] 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[17,18] 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。第二类方法则是允许用户空间对多个系统调用请求排队，并通过一次提交将他们注册给内核，如 FlexSC[19] 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，但如何设置提交的时机难以把握，过短的提交间隔将导致切换次数增加，过长的提交间隔则会导致 CPU 空转。

虽然现有工作难以广泛且有效地应用到微内核中，但他们的思路值得借鉴，他们的缺陷驱使研究者去寻求更好的方案。在硬件方面，一种新型的硬件技术方案——用户态中断[20,21] 逐渐被各个硬件平台（x86，RISC-V）采纳，它通过在 CPU 中新增中断代理机制和用户态中断的状态寄存器，当中断代理机制检测到状态寄存器发生变化时，会将中断以硬件转发的形式传递给用户态程序，从而绕过内核。该硬件方案已经在 Sapphire Rapids x86 处理器上和 RISCV 的 N 扩展中有了一定的支持，适用范围更加广泛。而在软件方面，异步被广泛用于请求合并和开销均摊，传统类 Unix 系统提供的类似 select IO 多路复用接口相对简陋，迫使用户态代码采用事件分发的编程范式来处理异步事件，代码相对复杂，可读性较弱。而新兴的 Rust[22,23] 语言对异步有着良好的支持，其零成本抽象的设计也让它作为系统编程语言有着强大的竞争力。使用 Rust 进行内核和用户态基础库的开发，可以更好地对异步接口进行抽象，改善接口的易用性和代码的可读性。

1.3 主要研究内容

本文以免费开源的 seL4 的 IPC 系统主要研究对象，介绍了 seL4 中 IPC 的设计细节以及相关设计缺陷，针对 seL4 的相关缺陷，基于用户态中断，给出了高性能异步微内核的整体设计方案，包含了无需内核转发的通知机制 U-notification、异步 IPC 与异步系统调用，并基于 Rust 语言在 RISC-V 平台实现了 ReL4 原型系统。在 FPGA 上将 U-notification、异步 IPC 和异步系统调用分别与 seL4 进行了对比测试，并评估了 ReL4 在真实的 TCP Server Benchmark 上的性能表现。论文内容结构安排如下：

第 1 章，绪论部分，阐述了本课题的研究背景及意义，国内外研究现状和发展趋势，最后概述了本课题的研究内容和对内容的组织结构。

第 2 章，介绍了 seL4 系统的发展历史和主要特征，并对 seL4 中的基本概念以及 IPC 设计进行了系统性阐述。

第 3 章，系统设计部分。首先介绍了 ReL4 与 seL4 的关系，ReL4 的特点以及设计目标，然后介绍了 ReL4 如何对系统中各个部分的通知机制进行支持和优化，最后介绍了用于保证兼容性和提升易用性的异步运行时的组成部分和设计思路。

第 4 章，系统实现部分。主要介绍了新增的系统调用，以及基于 U-notification 的异步运行时如何实现异步 IPC 和异步系统调用。

第 5 章，兼容性讨论。主要讨论了 ReL4 在通知机制与 IPC 机制中对 seL4 的兼容程度。

第 6 章，系统评估部分。首先介绍了 ReL4 的测试方案，然后针对 ReL4 的各个机制进行了性能评估和分析，最后测试了 ReL4 在真实的 TCP Server Benchmark 上的性能表现。

第 7 章，总结与展望，对本次设计和论文工作进行了总结，并提出了进一步的研究方向与对未来工作的展望。

第 2 章　seL4 介绍

2.1　seL4 的发展历史及主要特征

seL4 是一款具有创新性和里程碑意义的微内核。seL4 项目始于 2006 年的澳大利亚悉尼大学，其目标是创建一个经过形式化验证的微内核，从而确保内核的安全性和可靠性。在 2009 年，seL4 正式发布了针对 arm 11 处理器的功能正确性的形式化证明，是全球首个经过完整形式化验证的微内核。在 2014 年，seL4 正式开源，得到了来自开源社区的广泛关注，seL4 的生态蓬勃发展。在接下来的几年里，seL4 陆续完成了对不同 CPU 和不同指令集架构的验证和支持、虚拟化支持等，并在应用生态领域有了丰富的支持。

seL4 的设计遵循一下几个原则[24]：

- Verification：截止目前（2025 年 3 月），seL4 依然是第一个经过形式化验证的内核，形式化验证对 seL4 是个坚持不懈的努力目标，为了验证方便，禁止在内核里并发处理，不允许在内核态的大部分场景里再次发生中断。
- Minimality：一方面最小化原则是 L4 家族的根本设计理念，另一方面，最小化也是方便 seL4 做形式化验证的重要条件，seL4 内核除了中断控制器、定时器、MMU 相关的一点硬件驱动代码，其它驱动都在用户空间运行。
- Policy freedom：seL4 对于大部分资源分配策略都移到了用户态进行定制，通过 Capabiltiy 进行管理。
- Performance：虽然极度关注安全、可形式化验证，seL4 着重对热点路径的优化，因此依然有着突出的性能优势。
- Security：seL4 在安全性设计上遵循最小权限原则（Least privilege），通过 capability 机制来保证任何组件只拥有完成其工作所需的权限。

2.2　seL4 的基本组成

如图2.1所示，seL4 基本采用分层的设计理念，在内核态保留了虚拟地址管理（VMM），进程间通信（IPC）、通知机制（Notification）、任务调度（Scheduler）和中断管理（Interrupt Manager），同时在内核态提供能力访问控制（Capability）来进行权限

管理。驱动程序和大部分系统服务（如网络协议栈和文件系统等）运行在用户态，应用程序通过 IPC 请求系统服务，同时，硬件中断通过 Notification 传递给用户态驱动。

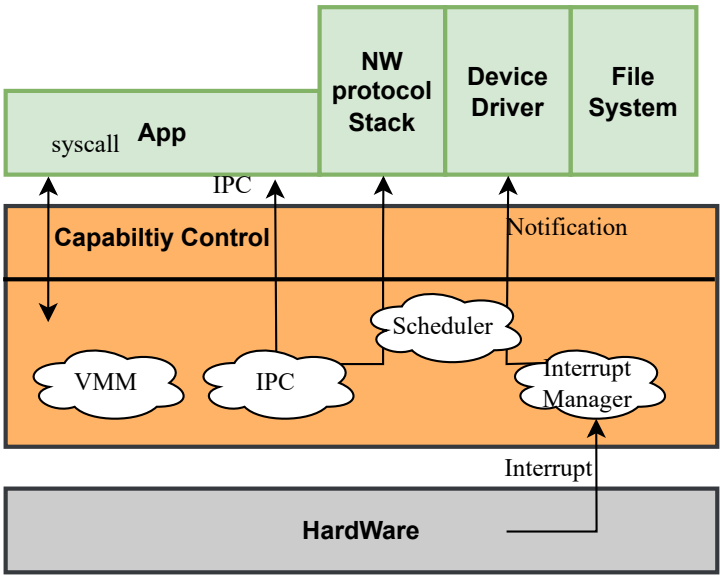

图 2.1　seL4 的系统结构图

2.2.1　内核对象与 Capability 机制

在 seL4 中，内核对象是操作系统内核所管理和操作的基本实体，是系统资源的软件抽象。内核通过维护内核对象的状态来维护系统状态。主要的内核对象如表2.1所示。

表 2.1　seL4 中的主要内核对象

内核对象	作用
线程控制块（TCB）	内核调度的基本单位，保存了用户任务运行所需的上下文。
能力空间（CSpace）	访问能力的集合，维护了各个内核对象的访问能力和对应权限。
地址空间（VSpace）	地址空间，维护了一段虚拟地址和物理地址的映射关系。
物理页框（Frame）	对应一个物理页，维护了物理页号和访问权限.
端点（Endpoint）	同步 IPC 的桥梁，维护了一个消息收发的状态机。
通知对象（Notification）	通知机制的桥梁，维护了一个通知状态位。
无类型对象（Untyped）	物理内存管理的承载者，通过系统调用转化为其他内核对象。

内核对象无法直接被用户态访问，只能通过 Capability 机制将能力句柄（Capability

handler）暴露给用户态，用户态在 handler 上调用系统调用来对内核对象进行访问。能力句柄是能力（Capability）的索引，在 Capability 中，内核维护了对应的内核对象地址以及对应的访问权限，当用户态在 handler 上发起系统调用时，内核会通过查找到相应的 Capabiltiy 并检查对应的权限，权限检查通过之后会根据地址对内核对象进行相应的操作。

Capabiltiy 可以被转移和派生，转移之后原始的用户态进程就不再拥有对相应内核对象操作性的权限，而派生则保留了原始进程的权限，派生的 Capabiltiy 的权限是原始权限的子集，被派生的 Capabiltiy 可以进一步派生，由此在内核形成一个能力派生树（Capability Tree），为了保证可控性，父节点可以通过 revoke 操作随时回收子节点的能力。由此，seL4 通过能力派生的形式递归地构建了整个系统的权限控制体系。

2.2.2 内存管理

seL4 中将物理内存管理和虚拟内存管理分开。物理内存由用户态管理，而虚拟内存由内核态管理。

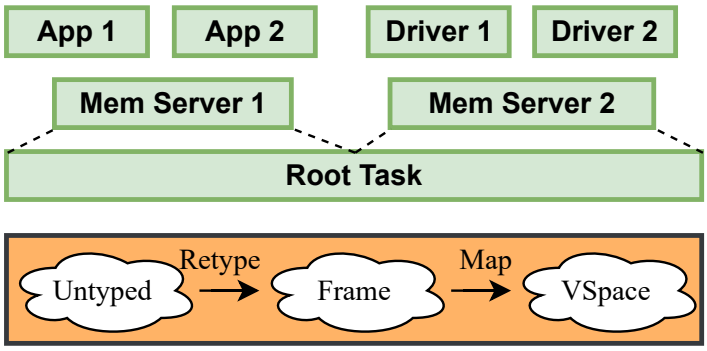

图 2.2 seL4 的内存管理

如图2.2所示，在 seL4 中，有一类特殊的无类型内核对象 Untyped，保存了未被分配的物理内存信息，用户态通过 Retype 系统调用将其转化为小的 Untyped 或者有类型的内核对象（如 Frame）。内核在启动完成之后将所有的 Untyped 对象的 Capabiltiy 交给 Root Task 用户进程，Root Task 将通过 Retype 或能力派生的形式对空闲的物理内存进行进一步分配，由此可见，seL4 系统中的物理内存管理也是分布式递归管理的。

在 seL4 中，每个进程都包含了一个 VSpace 内核对象，对应一个根页表，维护了

进程地址空间的映射关系，而用户态通过 map/unmap 系统调用将 Frame 内核对象和虚拟地址的映射关系修改到页表中。

2.2.3 任务调度

在 seL4 中，线程是任务调度的基本单位和执行单元，每个线程包含了一组能力空间和一套执行上下文。进程的概念被弱化，通常情况下，我们将含有相同能力空间和地址空间的称为同一个进程，将进程抽象为资源分配的基本单位，因此在 seL4 中，进程只作为一个逻辑概念存在，内核的设计和实现只涉及到线程。

在 seL4 中，线程调度主要分为两类：优先级调度和抢占调度。优先级调度时机一般是某个任务时间片用完、或由于阻塞事件无法继续执行时，内核从优先级队列中取出最高优先级的队头任务进行调度执行。而抢占调度一般发生在同步 IPC 中，当某个线程的操作唤醒了另外一个线程，在不违反优先级调度原则的前提下，会优先插队调度。具体的同步 IPC 流程描述参考2.2.4。线程的状态转换图如2.3所示。

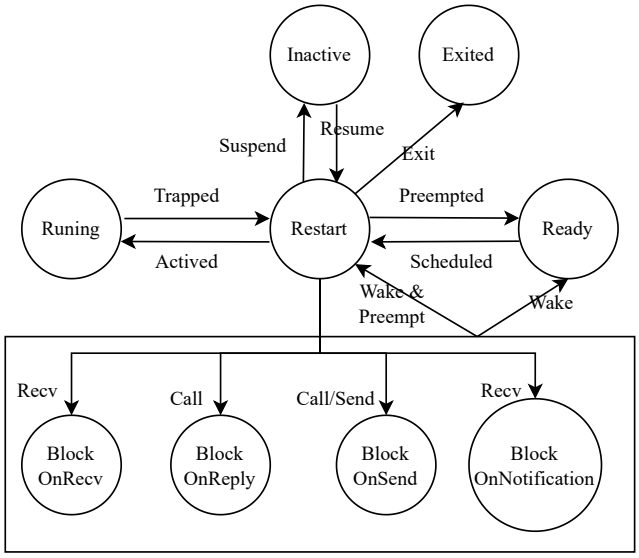

图 2.3　seL4 的线程状态转换

此外，seL4 的 MCS 扩展还支持混合关键性系统（MCS）来进行实时性调度，内核通过时间片预留、动态调整和模式切换来保证任务的实时性。

2.2.4 同步 IPC 和通知机制

seL4 设计者认为系统中的大部分 IPC 都遵循客户端/服务器（C/S）架构，客户端线程发起请求后等待服务端线程接收请求并返回响应，在此之前，客户端进程会阻塞在一个名为 Endpoint 的内核对象上，相似的，服务端线程也会阻塞监听 Endpoint 对象，直到有请求到来。如2.4内核通过维护 Endpoint 状态和阻塞队列来维持通信的秩序。值得注意的是，在 seL4 中，同步 IPC 更像是任务调度的一部分，同步 IPC 可能会导致任务的切换，以客户端在 OnRecv 状态的 Endpoint 发起请求为例，在不违反优先级调度原则的前提下，内核会将阻塞队列中的服务端线程唤醒并切换执行（即2.2.3中的抢占调度），而将客户端线程阻塞并等待响应。

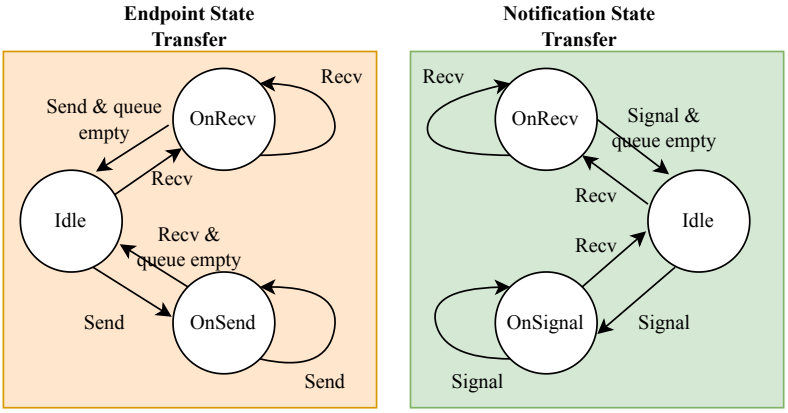

图 2.4 seL4 的 IPC 相关内核对象状态转换

由于同步 IPC 强制用户态以多线程的形式处理并发，因此 seL4 还支持非阻塞的通知机制，客户端线程非阻塞地通知接受线程，不会阻塞地等待服务端发起接收流程，由此解放了客户端线程，而服务端线程则依然要阻塞地接收客户端的信号，因此对于服务端来说依然是同步的流程。如2.4所示，与同步 IPC 类似，seL4 通过在 Notification 内核对象中维护对象状态和一组信号字来实现通知机制，在相似的情况下也会导致任务的切换。

2.2.5 中断管理与 SMP 支持

seL4 在中断管理与对称多处理机（SMP）支持方面也与主流内核有所区别，seL4 为了保证内核行为的可预测性，在内核中屏蔽了所有外部中断，禁止内核中的中断抢占，这是由于 seL4 中的大部分内核任务都极其简短，因此停留在内核中（屏蔽中断）

的时间非常少，不会过多地影响系统的实时性，而对于少量可能造成内核执行时间较长的系统调用，seL4 在这些系统调用中插入了可抢占点，将内核的行为约束在可控范围内。

出于同样的目的，对于多 CPU 核心系统，seL4 通过内核锁保证只有一个核心运行在内核态，避免了复杂的内核资源竞争，同时保证了内核行为的可预测性，大部分的系统调用都只会短暂停留在内核中，一般不会出现饥饿等待的情况，而对于少量可能造成内核执行时间较长的系统调用，seL4 在流程中添加重启点，保存少量用于指示当前执行状态的信息，然后释放内核锁，等待下一次获取内核锁之后重启流程，以此避免饥饿等待的发生。

第 3 章　ReL4 系统设计

ReL4 是一款用 Rust 编写的，兼容 seL4 基本系统调用的高性能异步微内核，它借鉴了 seL4 的权限管理、内存管理和 SMP 设计等基本框架，但在 IPC 机制和任务调度方面进行了优化和改进，如3.1所示，ReL4 将 IPC 从内核中移除，内核仅支持基于硬件的通知机制，应用程序仅需在注册时陷入内核分配相应的硬件资源，在通知过程中，通过硬件直接传递信号，减少特权级切换；同时，ReL4 在用户态设计了一套异步运行时，由用户态实现异步 IPC 和异步系统调用，所有的数据传递都通过共享缓冲区实现，由通知机制进行同步，并提供协程的概念作为任务调度的基本单位，提升系统并发性；最后，ReL4 使用 TAIC——一种基于用户态中断的调度器硬件加速单元，减少低并发场景下的运行时开销。

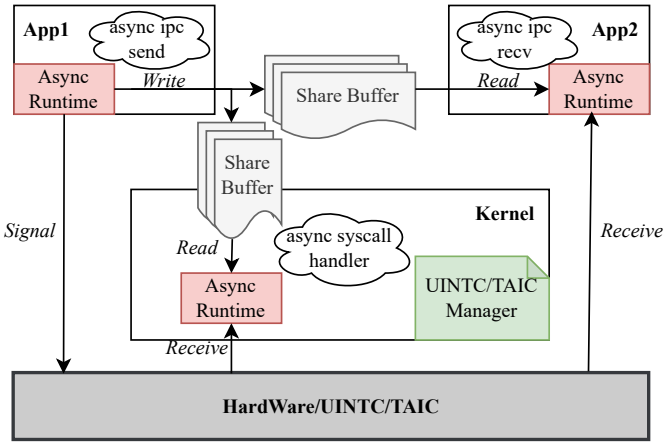

图 3.1　ReL4 的系统架构图

ReL4 的设计原则如下：

- 内核最小化原则：精简内核，在内核中移除同步 IPC，由用户态实现。
- 避免特权级切换：通过软硬结合等手段避免系统在 IPC、通知和系统调用过程中的频繁地进行特权级切换。
- 易用性原则：通过编程语言支持和接口封装等手段，避免用户层接口的改动，同时提供更易用的异步化接口简化编程模型。

本章的剩余内容将详细介绍系统设计的两个核心内容：通知机制和异步运行时。

3.1 通知机制

ReL4 将整个系统中的通知机制按照收发双方的特权级进行分类：1）用户态通知用户态；2）内核态通知用户态；3）用户态通知内核态；4）内核态通知内核态。本文希望借助用户态中断并辅助软件设计，尽量避免通知过程中的特权级切换。对于 1）和 2）而言，ReL4 借助用户态中断，重新设计了 seL4 的通知机制，避免了特权级切换，对于 3），ReL4 通过系统调用的形式通知内核，并通过自适应轮询机制减少通知的次数，对于 4），不存在特权级的切换，仅通过核间中断就可以实现内核态之间的通知。因此本文将着重介绍基于用户态中断的通知机制（U-notification），以及用于减少通知次数的自适应混合轮询机制。

3.1.1 U-notification

如3.2所示，用户态中断使得控制流和数据流相互分离。ReL4 在 notification 内核对象中维护了对应的硬件资源索引，控制流主要由用户态向内核进行注册，申请硬件资源，数据流则通过特殊的用户态指令访问用户态中断控制器，从而在通信过程中避免了特权级的切换。

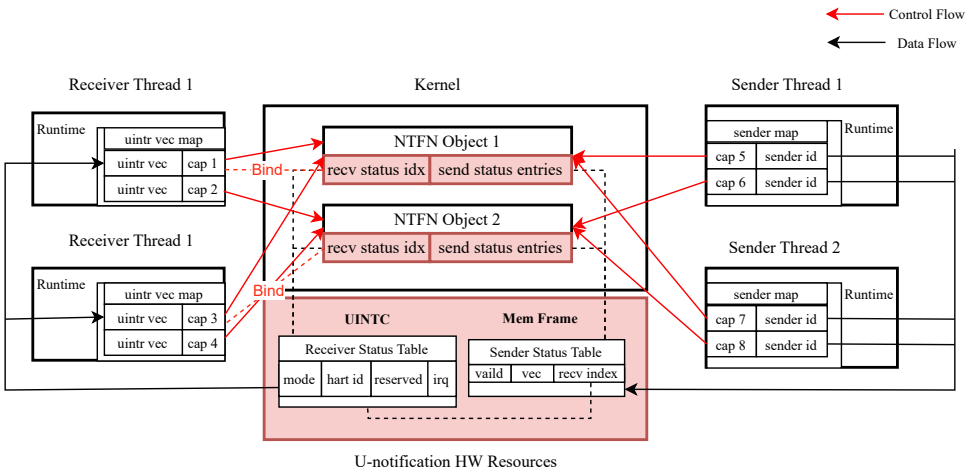

图 3.2 ReL4 的系统架构图

控制流主要分为发送方的注册和接收方的注册。接收方在用户态通过 $Untyped\_Retype$。申请一个 Notification 对象之后，调用 TCB_Bind 接口进行硬件资源绑定，运行时进一步调用 $UintrRegisterReceiver$ 系统调用，将运行时中定义的用户态中断向量表注册到 TCB 中，申请 UINTC 的接收状态表项，并绑定到 Notification 对象及其对应的

线程上。发送方通过 Capability 派生的形式（直接构造发送方的 Capability 空间，或通过内核转发的形式获取 Capability）获取指向 Notification 对象的 Capability，第一次调用 Send 操作时，运行时会判断 Cap 是否有对应的 Sender ID，如果没有，则调用 $UintrRegisterSender$ 系统调用进行发送端注册，并填充对应的 SenderID。相关资源的回收则通过已有的 $revoke$ 或 $delete$ 系统调用注销内核对象。

数据流由硬件直接传递，无需通过内核。发送端在注册完成之后，可以直接调用 $uipi\_send$ 指令，指令根据 Sender Status Table Entry 中的索引设置中断控制器中的寄存器。如果接收端本身在 CPU 核心上运行，会立刻被中断并跳转到注册的中断向量表，否则会等到被内核重新调度时再处理通知。

与传统的 notification 相比，U-notification 只需要在注册阶段陷入到内核，而通信过程由硬件完成，

3.1.2 自适应的混合轮询

虽然用户态中断的开销小于特权级切换，但仍然对程序局部性和内存局部性不够友好，且用户态通知内核态仍然要进行特权级的切换，而轮询虽然可以避免中断式通知，但却会导致 CPU 资源的浪费，因此 ReL4 在共享缓冲区中维护通知处理程序的就绪状态标识。当通知频率足够高或处理程序负载足够大，以至于上一个通知还未被处理完成，下一个通知就即将发起，处理程序会始终处于运行状态，此时发送端无需发送额外的通知，其工作方式等价于轮询模式。当通知频率较低时，处理程序在大部分时间处于阻塞状态，节省 CPU 资源，工作方式等价于中断模式。

3.2 异步运行时

由于内核不再支持同步 IPC，为了提升用户态的易用性，ReL4 在用户态设计了异步运行时，它提供了如下功能，使得用户态程序设计变得更加简单和高效：

- 共享缓冲区：用于跨进程的零拷贝数据传递。
- 协程与调度器：提升用户态的并发度，减少用户态中断和特权级切换次数，并为不同负载的任务提供可定制性调度策略。
- API 兼容层：提供与 seL4 相同的通知机制、异步系统调用和异步 IPC 的用户态接口，提升系统易用性。

3.2.1 共享缓冲区

由于 U-notification 只能传递通知信号，因此 ReL4 依然需要共享缓冲区来作为 IPC 数据传递的主要形式。以 IPC 中最常见的 Call 为例，客户端需要将请求数据准备好并写入共享缓冲区中，而服务端将在某个时刻从共享缓冲区中读取请求，处理后将响应写回共享缓冲区，而客户端也将在之后的某一时刻从共享缓冲区中读取响应并进行相应处理。这个流程中有几个挑战需要明确：

1. 请求和响应的格式和长度如何设计才能使得缓冲区访问效率更高。
2. 在共享缓冲区中如何组织请求和响应的存取形式，才能在数据安全读写的前提下保证性能。
3. 客户端和服务端如何选择合适的时机来接收数据。

如3.3所示，一个 IPC 消息（请求或响应）被定义为 IPCItem，它是 IPC 传递消息的基本单元，为了减少消息读写以及编解码的成本，ReL4 采用定长的消息字段。每个 IPCItem 的长度被定义为缓存行的整数倍并对齐，消息中的前四个字节存储发送端的 sender id，方便后续发送 U-notificaiton 进行唤醒，后四个字节记录写入的协程 id，便于后续进一步唤醒，msg info 用于存储消息的元数据，包含了消息类型、长度等。extend msg 将被具体的应用程序根据不同的用户进行定义。

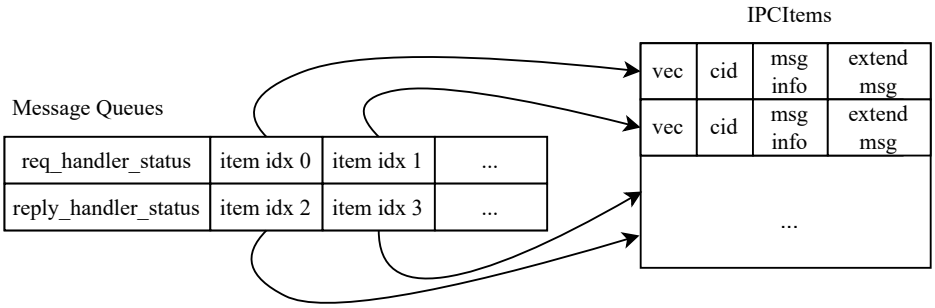

图 3.3　共享缓冲区的结构图

客户端在发起请求之前需要先从缓冲区中申请一个 IPCItem 并将对应的索引写入请求队列，服务端会根据请求队列中的索引读取请求消息，并将响应写回到对应的 IPCItem，并将索引写入响应队列。由于请求队列和相应队列会被一个以上的线程同时访问，因此需要设计同步互斥操作来保证数据的读写安全。同时队列的访问极为频繁，需要尽可能避免数据竞争来保证读写性能。ReL4 将请求和响应的索引放到不同

的环形队列中，同时不同的发送方和接收方使用不同的环形队列以保证单生产者单消费者的约束，消除过多的数据竞争，最后，ReL4 使用无锁的方式 [27] 进一步提升环形队列的读写性能。

最后，为了支持 2.1.2 中所提到的自适应轮询机制，ReL4 还在队列中维护了对端处理程序的就绪状态标识 handler_status，客户端和服务端将根据该标志位来决定是否发送 U-notificaiton。

3.2.2 协程与调度器

传统微内核中的同步 IPC 会导致发送端线程阻塞，从而造成一些没有依赖的 IPC 被迫以顺序的形式执行，或者强制要求多线程来实现并发。而 ReL4 中的异步运行时将协程作为任务的执行单元，以提升用户态并发度。同时，为了减少调度的开销，主要是跨进程唤醒协程的开销，ReL4 基于 TAIC 来进行硬件加速。

如3.4所示，TAIC 是一个基于用户态中断的调度器加速单元，它将用户态中断的中断号与协程号进行一一绑定，并使用硬件来自动唤醒协程。在理想情况下，软件无需手动唤醒协程，然而由于硬件资源有限，用户态中断号仅支持 0～31，而协程的数量远远大于这个量级，因此每个运行时中常驻了一个 dispatcher 协程，并绑定 0 号中断向量，当中断号不够用时，硬件唤醒对应的 dispatcher 协程，之后由 dispatcher 协程手动唤醒其他协程。

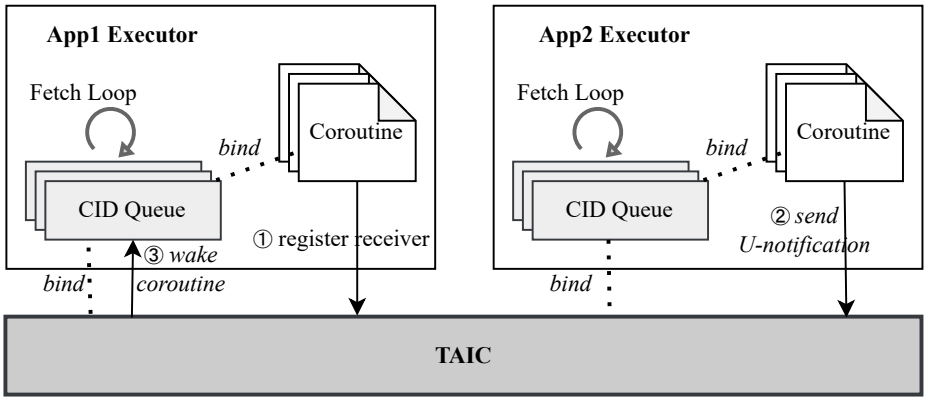

图 3.4 调度器的结构图

ReL4 将协程分为 worker 协程和 dispatcher 协程，用户态的 IPC 任务都将被封装到 worker 协程，由运行时内的调度器进行调度，而 dispatcher 协程则在硬件资源有限

的情况下进行二级唤醒。从调度器的角度来看，不同进程的调度器相互独立，每个进程中的 worker 协程和 dispatcher 协程存在着一定的依赖关系。以 IPC 场景下的客户端为例，worker 协程用于发起 IPC 请求，dispatcher 协程则是处理响应。从高吞吐率的角度来讲，自然是希望更快的处理 worker 协程，而从低延迟的角度来讲则是希望优先调度 dispatcher 协程，高吞吐和低延迟的特性由上层业务决定，框架层只根据业务配置进行支持。此外，不同的 worker 协程也需要有轻重缓急之分，以便更有效率地利用 CPU 资源。

基于上述原因，ReL4 在调度器中维护了一个优先级队列，每个协程都被设有相应的优先级，调度器在内部维护了一个优先级位图和若干任务队列，调度器将根据优先级位图选出最高的优先级，找到对应的任务队列并以先进先出的形式选出任务来运行。用户态程序根据业务特点设置相关的优先级，以达到性能调优的目的。

3.2.3 API 兼容层

为了使异步系统调用和异步 IPC 能够与同步接口保持一致，异步运行时提供了异步系统调用和 IPC 的的 hook 库，用户态的请求将会被 hook 库接管，hook 库根据不同的调用类型，来自动选择是否转化为异步系统调用或 IPC。需要注意的是，无法转化为异步系统调用的主要有以下两类：

1. 由于异步系统调用依赖于异步运行时，因此与异步运行时初始化相关的系统调用无法被异步化。
2. 对于实时性要求较高的系统调用无法进行异步化，如 $get\_clock()$。

而为了尽可能兼容 seL4 中的 capability 机制，运行时库中还维护了 notification cap 与用户态中断相关资源的映射：

1. sender map：由于 U-notification 以及异步 IPC 都无需通过内核，因此运行时需要维护 capability 与 sender id 以及共享缓冲区的映射关系。
2. uintr vec map：用户态中断通过中断向量区分发送端，而 seL4 通过 capability 区分发送端，为了兼容多发送端，运行时需要维护相关的映射关系。

第 4 章　ReL4 系统实现

为了简洁高效地实现异步微内核的原型系统，本项目使用 Rust 语言在 RISC-V 平台上实现了一个兼容 seL4 的微内核 ReL4，目前已经支持 SMP 架构和 fast-path 优化。在兼容 seL4 原始功能的基础（包括 SMP 和 fast-path 优化）上，ReL4 实现了 U-notification 以及异步 IPC 和异步系统调用。在实现过程中对内核接口更改和使用的一些重要细节将在本章描述。

4.1　新增系统调用

如4.1所示，为了支持内核对 U-notification 的硬件资源管理，ReL4 新增了系统调用：UintrRegisterSender 和 UintrRegisterReceiver 用于申请相关的硬件资源，其参数是 notification 内核对象对应的 Capbability。内核会从 UINTC 中分配对应的硬件寄存器索引，并将对应的索引绑定到 TCB 中，资源的释放不需要额外的系统调用，当 TCB 或 notification 对象销毁时会自动释放掉硬件资源。

此外，为了支持异步系统调用，共享缓冲区也需要通过系统调用 (UintrRegisterAsyncSyscall) 注册给内核，内核会为将共享缓冲区与 TCB 绑定，并为该线程注册处理该缓冲区请求的内核协程。最后，由于用户态中断不支持用户态直接通知内核态，内核提供一个用于唤醒系统调用处理协程的系统调用 UintrWakeSyscallHandler，内核会将对应的处理协程唤醒，并找到一个空闲的 CPU 核心发送核间中断去抢占执行。

为了对 seL4 进行兼容，这些系统调用均由异步运行时在初始化时进行调用，用户程序无需感知。

syscall	参数	描述
UintrRegisterSender	ntfn_cap	注册通知发送端
UintrRegisterReceiver	ntfn_cap	注册通知接收端
UintrRegisterAsyncSyscall	ntfn_cap, buffer_cap	注册异步系统调用处理协程
UintrWakeSyscallHandler	-	唤醒系统调用处理协程

表 4.1　ReL4 中的新增系统调用

4.2 异步 IPC

异步 IPC 作为 ReL4 中的主要的 IPC 方式，其实现依赖于异步运行时和 U-notification。以 IPC 中最常见的 Call 为例，如4.1所示，客户端进程和服务端进程在双方建立连接时，首先会想内核申请两条通道所占用的硬件资源，然后每个进程中的 runtime 会注册一个 dispatcher 协程并将绑定 0 号中断向量，用于在 TAIC 资源不够的情况下，用软件方式唤醒 worker 协程。服务端和客户端的 worker 协程则负责发起和处理请求，由 runtime 将数据写入共享缓冲区中。由于中断向量有限，runtime 会尝试为每个 worker 协程分配中断向量，分配到中断向量的协程，其唤醒过程由 TAIC 完成，没有分配到中断向量的协程，其唤醒过程由 dispatcher 协程完成。

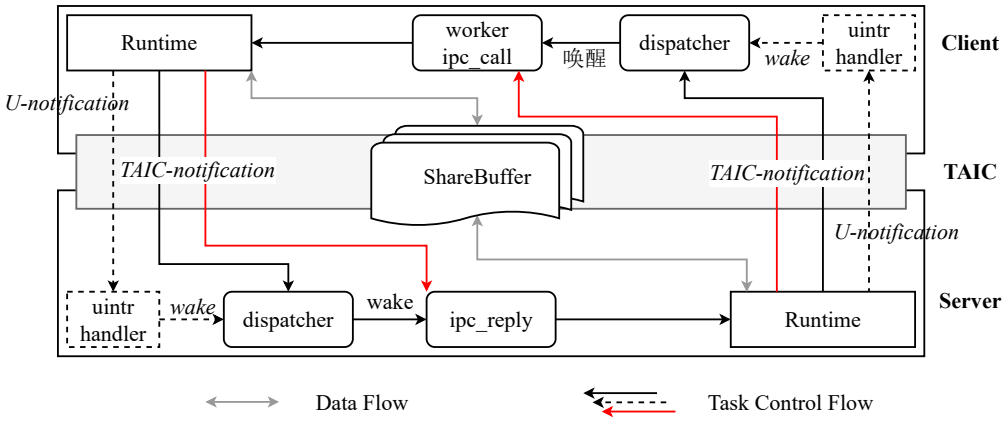

图 4.1　调度器的结构图

Call 的主要流程分为以下几个阶段：

1. 客户端发起请求：用户态程序将以 worker 协程的形式发起 IPC 请求，异步运行时首先会尝试分配一个中断向量给 worker 协程，如果如果没有分配到，则复用 dispatcher 协程的 0 号中断，然后根据请求的数据和协程的协程号、中断号生成 IPCItem 并写入请求的环形缓冲区中并将当前协程阻塞，然后检查缓冲区的 $req\_handler\_status$ 标志位，如果对方的 dispatcher 协程已经就绪，那客户端无需通知对方进程，对方进程的异步运行时会在某个时刻调度到 dispatcher 协程并处理请求。如果对方的 dispatcher 协程处于阻塞状态，则异步运行时会将 $req\_handler\_status$ 标志位置位，并发送 U-notification 通知对方进程唤醒 dispatcher 协程并重启调度。
2. 服务端处理请求并写回响应：服务端的 dispatcher 协程会在合适的时机读取

出请求并进行解码和处理，然后根据处理结果构造响应的 IPCItem 并写入响应的环形缓冲区中，如果中断号是 0 号协程，则 runtime 会检查缓冲区中的 $req\_handler\_status$ 标志位后尝试唤醒客户端的 dispatcher 协程，否则直接通过 TAIC 唤醒客户端的 worker 协程。如果缓冲区内容为空，dispatcher 协程会将 $req\_handler\_status$ 标志位置空，并将自己阻塞。

3. 客户端处理响应：客户端的 dispatcher 协程会在合适的时机重新被调度并唤醒没有分配到 TAIC 资源的 worker 协程，唤醒后的 worker 协程会从缓冲区中读取响应并释放缓冲区资源。

其伪代码如1所示。

Algorithm 1: 异步 IPC 流程的伪代码

```
1  fn(async ipc_call(cap, msg_info) → Result<IPCItem>)
2      vec = get_vec_from_pool();
3      item = IPCItem::new(vec, current_cid(), msg_info);
4      buffer = get_buffer_from_cap(cap);
5      buffer.req_ring_buffer.write(item);
6      if buffer.req_co_status == false then
7          buffer.req_co_status = true;
8          u_notification_signal(0);
9      end
10     if let Some (reply) = yield_now ().await then
11         return Some (reply);
12     end
13     return Err (());
14 fn(async ipc_recv_reply(cap))
15     buffer = get_buffer_from_cap(cap);
16     while true do
17         if let Some (item) = buffer.req_ring_buffer.get() then
18             reply = handle_item(item);
19             buffer.resp_ring_buffer.write(reply);
20             if buffer.reply_co_status == false then
21                 buffer.reply_co_status = true;
22                 u_notification_signal(item.vec);
23             end
24         end
25         else
26             buffer.req_co_status = false;
27             yield_now ().await;
28         end
29     end
```

4.3 异步系统调用

从广义的角度来看，异步系统调用是一类特殊的异步 IPC，其接收方为内核。因此 ReL4 在内核中提供了一套相似的异步运行时以支持异步系统调用。异步系统调用与异步 IPC 的主要不同之处有两点：

1. 由于接收端是内核，发送端无法使用 U-notification 去通知内核。
2. 异步 IPC 中进程的异步调度器就是进程的执行主体，无需考虑异步任务的执行时机，而内核除了异步系统调用请求需要调度器执行，本身就有如中断、异常、任务调度等其他任务需要被执行。

对于第一点，ReL4 新增一个系统调用去用于唤醒相关的内核协程即可。而对于第二点，一个很简单的思路是每次时钟中断到来时去执行异步系统调用，然而这可能会导致空闲的 CPU 核心无法及时触发时钟中断而空转，因此，在不破坏原本的线程优先级调度前提下，ReL4 使用核间中断来抢占空闲 CPU 核心或正在运行低优先级线程的 CPU 核心，更好地利用空闲 CPU 资源，减少响应时延。

为了避免破坏微内核中原本的优先级调度机制，ReL4 在内核中对每个 CPU 核心维护了相应的执行优先级 ($exec\_prio$)，执行优先级区别于上文提到的运行时协程优先级，是由内核调度器维护的线程优先级。内核中的任务主要分为三类：

1. idle thread: 空闲 CPU 核心执行 idle 线程，此时 CPU 核心的执行优先级为 256，属于最低的执行优先级。
2. 内核态任务：正在处理中断、异常、系统调用等，此时 CPU 核心的执行优先级为 0，最高优先级，不可被抢占。
3. 用户态任务：正在执行用户态的任务，此时 CPU 核心的执行优先级为当前线程的优先级，可以被更高优先级线程提交的异步系统调用请求打断。

当发送端通过系统调用陷入内核去唤醒相应协程后，会检查当前线程的优先级是否可以抢占其他 CPU 核心，如果可以，则发送核间中断抢占该 CPU 核心去执行异步系统调用，当前 CPU 核心则返回用户态继续执行其他协程。如果没有可以被抢占的 CPU 核心，则在下一次时钟中断到来时执行异步系统调用请求，其伪代码2所示：

Algorithm 2: 唤醒内核中异步处理协程的伪代码

```
1  fn(wake_syscall_handler())
2      current = get_current_thread();
3      if let Some(cid) = current.async_sys_handler_cid then
4          coroutine_wake(cid);
5          current_exec_prio = current.tcb_prio;
6          (cpu_id, exec_prio) = get_max_exec_prio();
7          if current_exec_prio < exec_prio then
8              // 抢占低执行优先级的核心
9              mask = 1 ≪ cpu_id;
10             ipi_send_mask(mask, ASYNC_SYSCALL_HANDLE);
11         end
12     end
```

4.4 兼容性讨论

为了提升用户态程序的易用性，ReL4 对 seL4 的程序提供一定的兼容性。ReL4 中已经实现了 seL4 的基本系统调用并支持对称多处理机（SMP），但采用不同的通知机制和 IPC 设计和系统调用处理机制，因此有必要讨论这两部分的兼容性。

4.4.1 Notification 与 U-notification

相比于原始的通知机制，U-notification 在通信权限控制方面同主要存在以下两点不同：

1. 原始的通知机制允许多个接收线程竞争接收一个内核对象上的通知，这种设计的目的是为了支持多接收端的场景，事实上，多接收端已经通过多个内核对象来进行支持，因此这种机制相对冗余，而由于 U-notification 中接收端对接收线程的独占性，这个能力将不再被支持。
2. 原始的通知机制允许单个接收线程接收多个内核对象上的通知，这种设计的目的是更灵活地支持多发送端的场景，在 U-notification 中，同一个内核对象可以被设置为相同的 recv status idx，不同的发送端则通过使用中断号 (uintr vec) 来进

行区分。

除了权限控制有所不同之外，改造前后的通信方式也有所区别。原始的通知机制需要用户态接收方通过系统调用主动询问内核是否有通知需要处理。根据是否要将线程阻塞，一般被设计为 Wait 和 Poll 两个接口。而 U-notification 无需接收线程主动陷入并询问内核，接收方被硬件发起的用户态中断打断，并处理到来的通知，这在很大程度上解放了接收方，程序设计者无需关心通知到来的时机，减少了 CPU 忙等的几率，提升了用户态的并发度。而为了提升 U-notification 的易用性，ReL4 对原始的通信接口进行了兼容：

1. Poll: 无需陷入内核态，在用户态读取中断状态寄存器，判断是否有效并返回。
2. Wait: 对该接口的兼容需要用户态的异步运行时的调度器提供相关支持，在没有有效中断时，该操作将阻塞当前协程并切换到其他协程执行，等待用户态中断唤醒。

综上所述，对于多接收方的场景，U-notification 可以通过多个内核对象进行实现，除此之外，U-notification 可以实现 API 级别的兼容。

4.4.2 同步 IPC 与异步 IPC

异步 IPC 通过异步运行时可以在基本通信场景下上实现 API 级别的兼容，然而 seL4 中的同步 IPC 还有额外的能力：

1. 错误处理：同步 IPC 可以用于缺页异常等处理，seL4 通过在 TCB 中维护一个 Endpoint 对象来发送错误信息给用户态程序进行处理，而在 ReL4 中，TCB 中将维护对应的 U-notification 对象，以及对应的共享缓冲区指针，当异常和错误发生时，将错误信息写入共享缓冲区，并发送 U-notification 通知用户态程序。此场景下依然可以实现 API 级别的兼容。
2. 能力派生：seL4 中的同步 IPC 拥有能力派生与传递的功能，虽然内核已经支持了 Capability Space 相关的系统调用，同步 IPC 使得能力传递更加灵活。而由于异步 IPC 不经过内核，因此 ReL4 中不再支持通过 IPC 来进行能力派生，仅通过系统调用进行能力派生，损失了一部分灵活性，保留了功能的完整性。

此外，异步运行时导致用户态任务模型存在语义上的区别，异步 IPC 任务将以协程作为任务的基本单位，因此相比于同步 IPC 任务之外，异步 IPC 提供了主动让权的

API，同时，相比于同步 IPC，相同运行时内的不同异步 IPC 任务不存在并行性，无需同步互斥等操作，提升了用户态易用性。

综上所述，异步 IPC 在大部分情况下依然能实现 API 级别的兼容。

4.4.3 同步系统调用于异步系统调用

与异步 IPC 类似，异步系统调用的发起依然以协程为单位进行，但与同步系统调用的处理不同，为了充分利用 CPU 硬件，异步系统调用的处理采用多核心、抢占式处理，如果系统中存在空闲核心或执行低优先级任务的核心，异步系统调用会抢占该核心并处理系统调用请求，因此系统调用的发起和处理有可能是多核心且并行的。

此外，有两类系统调用无法异步化：

1. 由于异步系统调用依赖于异步运行时，因此与异步运行时初始化相关的系统调用无法被异步化。
2. 对于实时性要求较高的系统调用无法进行异步化，如 $get\_clock()$。

对于上述两类，我们通过异步运行时中的 API 兼容层进行自动判断，对于无法异步化的系统调用，运行时将自动转化为同步系统调用进行处理。综上所述，异步系统调用可以实现 API 级别的兼容。

第 5 章　性能评估

为了评估 ReL4 的兼容性，本文在 ReL4 上成功运行了 seL4test 并通过了相关的测试用例。而为了评估 ReL4 的性能表现，5.1 节设计了消融实验，来评估 U-notification 与自适应混合轮询对系统性能的影响；5.2 节通过内存分配器来评估异步系统调用的效率，5.3 节构建了 Ping-Pong 实验来评估异步 IPC 的性能；最后，5.4 节构建了一个真实的高并发的 TCP Server benchmark 用于评估 ReL4 在真实应用中的表现。

参考文献

[1] Abhilash T. Microkernel vs monolithic kernel design trade - offs[J]. International Journal For Multidisciplinary Research, 2024, 6(2): 1 – 5.

[2] Liedtke J. Toward real microkernels[J]. Communications of the ACM, 1996, 39(9): 70–77.

[3] Liedtke J. Improving ipc by kernel design[C]. Proceedings of the fourteenth ACM symposium on Operating systems principles. [S.l.: s.n.], 1993: 175–188.

[4] 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. [S.l.: s.n.], 2009: 207–220.

[5] Caruso J. 1 million iops demonstrated[M]. [S.l.: s.n.], 2021.

[6] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space[J]. Communications of the ACM, 2020, 63(6): 46–56.

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

[8] kernel development community T. Page table isolation(pti)[M]. [S.l.: s.n.], 2021.

[9] Blackham B, Shi Y, Heiser G. Improving interrupt response time in a verifiable protected microkernel [C]. Proceedings of the 7th ACM european conference on Computer Systems. [S.l.: s.n.], 2012: 323–336.

[10] 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.

[11] Mi Z, Li D, Yang Z, et al. Skybridge: Fast and secure inter-process communication for microkernels [C]. Proceedings of the Fourteenth EuroSys Conference 2019. [S.l.: s.n.], 2019: 1–15.

[12] Du D, Hua Z, Xia Y, et al. Xpc: architectural support for secure and efficient cross process call [C]. Proceedings of the 46th International Symposium on Computer Architecture. [S.l.: s.n.], 2019: 671–684.

[13] Kuo H C, Williams D, Koller R, et al. A linux in unikernel clothing[C]. Proceedings of the Fifteenth European Conference on Computer Systems. [S.l.: s.n.], 2020: 1–15.

[14] 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. [S.l.: s.n.], 2019: 59–73.

[15] Yu K, Zhang C, Zhao Y. Web service appliance based on unikernel[C]. 2017 IEEE 37th International

Conference on Distributed Computing Systems Workshops (ICDCSW). [S.l.]: IEEE, 2017: 280–282.

[16] 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). [S.l.: s.n.], 2023: 33–49.

[17] Jeong E, 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). [S.l.: s.n.], 2014: 489–502.

[18] 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). [S.l.]: IEEE, 2017: 154–161.

[19] 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). [S.l.: s.n.], 2010.

[20] 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): volume 65. [S.l.]: IEEE, 2022: 44–46.

[21] The risc-v instruction set manual. Volume ii: Privileged architecture, document version 1.12[M]. [S.l.: s.n.], 2020.

[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. [S.l.: s.n.], 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. [S.l.: s.n.], 2017: 156–161.

[24] Dude M. sel4 design principles[EB/OL]. 2020. https://microkerneldude.org/2020/03/11/sel4-design-principles/.

攻读学位期间发表论文与研究成果清单

（二）发表的学术论文

[1] XXX，XXX. Static Oxidation Model of Al-Mg/C Dissipation Thermal Protection Materials[J]. Rare Metal Materials and Engineering, 2010, 39(Suppl. 1): 520-524.（SCI 收录，IDS 号为 669JS，IF=0.16）

[2] XXX，XXX. 精密超声振动切削单晶铜的计算机仿真研究 [J]. 系统仿真学报，2007，19（4）：738-741，753.（EI 收录号：20071310514841）

[3] XXX，XXX. 局部多孔质气体静压轴向轴承静态特性的数值求解 [J]. 摩擦学学报，2007（1）：68-72.（EI 收录号：20071510544816）

[4] XXX，XXX. 硬脆光学晶体材料超精密切削理论研究综述 [J]. 机械工程学报，2003，39（8）：15-22.（EI 收录号：2004088028875）

[5] XXX，XXX. 基于遗传算法的超精密切削加工表面粗糙度预测模型的参数辨识以及切削参数优化 [J]. 机械工程学报，2005，41（11）：158-162.（EI 收录号：2006039650087）

[6] XXX，XXX. Discrete Sliding Mode Cintrok with Fuzzy Adaptive Reaching Law on 6-PEES Parallel Robot[C]. Intelligent System Design and Applications, Jinan, 2006: 649-652.（EI 收录号：20073210746529）

（二）申请及已获得的专利（无专利时此项不必列出）

[1] XXX，XXX. 一种温热外敷药制备方案：中国，88105607.3[P]. 1989-07-26.

（三）参与的科研项目及获奖情况

[1] XXX，XXX. XX 气体静压轴承技术研究, XX 省自然科学基金项目. 课题编号：XXXX.

[2] XXX，XXX. XX 静载下预应力混凝土房屋结构设计统一理论. 黑江省科学技术二等奖, 2007.

致谢

本论文的工作是在导师……。

作者简介

本人…。