南 开 大 学

本科生毕业论文（设计）

中文题目：**面向 Rel4 操作系统的用户态中断机制研究**

外文题目：Research on User-Mode Interrupt Mechanisms for the Rel4 Operating System

学　　号：2112495

姓　　名：魏靖轩

年　　级：2021 级

专　　业：计算机科学与技术

系　　别：计算机科学与技术

学　　院：计算机学院

指导教师：宫晓利　教授

完成日期：2025 年 5 月

关于南开大学本科生毕业论文（设计）的声明

本人郑重声明：所呈交的学位论文，是本人在指导教师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或没有公开发表的作品内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：魏靖轩

2025 年 5 月 18 日

本人声明：该学位论文是本人指导学生完成的研究成果，已经审阅过论文的全部内容，并能够保证题目、关键词、摘要部分中英文内容的一致性和准确性。

学位论文指导教师签名：

2025 年 5 月 18 日

摘 要

微内核以 IPC 完成内核对用户的服务，作为热点路径，在系统运行中大约 20% - 40% 的时间与 CPU 资源被用于了 IPC。一个微内核系统的整体性能很大程度上取决于 IPC 的效率，如何提高 IPC 的性能也一直是微内核领域研究的重点。此外，现有的 IPC 大多为主动的显式调用来轮询获得消息，这一机制在很多场景下也会出现轮询被阻塞延后导致时延上升的问题。

本文使用微内核 reL4 作为研究平台，将 x86 架构下的新硬件特性 - 用户态中断（Uintr）引入到 reL4 中，作为一种 IPC 机制使用。为 reL4 实现了用户态中断相关的系统调用，作为内核与用户程序的接口层，来完成使用前需要的用户注册与 CPU 寄存器状态设置，并根据接收线程是否被阻塞（即是否处于用户态）实现了两条内核通路。最终为微内核的用户程序提供了一个用户态中断的使用流程标准，让用户程序可以在用户态通过中断的方式来传递消息或做通知使用。

同时本文将用户态中断的使用范围进一步扩大，将其引入到 I/O 过程中，用于 I/O 完成时的通知。本文为 reL4 实现了一个异步 I/O 框架，它借鉴了 Linux 的最新高性能异步 I/O 框架 - io_uring，但根据微内核的架构设计和微内核与宏内核的差别进行了改进，让其更符合微内核的执行逻辑和规范。且为此框架实现了两种 I/O 完成时的通知方式：用户程序主动轮询查看 I/O 是否完成；I/O 完成后发送用户态中断，用户线程被中断打断进入中断处理函数，在中断处理函数中完成 I/O 任务。用户态中断的通知方式可以通过中断实时打断来结算 I/O 任务，具有比轮询更低的 I/O 时延。

最后，本文对用户态中断进行了实验评估。使用 Ping-Pong 实验来测试用户态中断作为 IPC 的功能性以及性能。实验表明，在有时钟中断和内核调度等因素的干扰下，用户态中断的 Ping-Pong 时延最快可以达到现有最快 IPC 方式的 25%，且最劣化情况基本与其相等。同时本文也测试了异步 I/O 框架的不同通知方式的 I/O 时延，实验结果表明，用户态中断会额外带来轮询的约 5% 的总时间的开销，但证明了可以通过中断的方式及时完成 I/O 任务的结算，让 I/O 任务的时延降低为轮询的 15% 左右。

关键词：用户态中断；微内核；操作系统

Abstract

The microkernel uses IPC to complete the kernel's service to users. As a hot path, about 20% - 40% of the time and CPU resources are used for IPC during system operation. The overall performance of a microkernel system depends largely on the efficiency of IPC. How to improve the performance of IPC has always been the focus of research in the field of microkernels. In addition, most of the existing IPCs are active explicit calls to poll for messages. This mechanism will also cause polling to be blocked and delayed in many scenarios, resulting in increased latency.

This paper uses the microkernel reL4 as a research platform and introduces the new hardware feature user-mode interrupt (Uintr) under the x86 architecture into reL4 as an IPC mechanism. The system calls related to user-mode interrupts are implemented for reL4. As the interface layer between the kernel and the user program, the user registration and CPU register status setting required before use are completed, and two kernel paths are implemented according to whether the receiving thread is blocked (that is, whether it is in user mode). Finally, a user-mode interrupt usage paradigm is provided for the user program of the microkernel, allowing the user program to pass messages or make notifications in user mode through interrupts.

At the same time, this paper further expands the scope of use of user-mode interrupts and introduces them into the I/O process for notification when I/O is completed. This paper implements an asynchronous I/O framework for reL4, which draws on Linux's latest high-performance asynchronous I/O framework io_uring, but improves it based on the microkernel's architectural design and the differences between microkernels and macrokernels to make it more in line with the execution logic and specifications of microkernels. Two notification methods for I/O completion are implemented for this framework: the user program actively polls to see if the I/O is completed; after the I/O is completed, a user-mode interrupt is sent, and the user thread is interrupted by the interrupt and enters the interrupt handling function, completing the I/O task in the interrupt handling function. The notification method of user-mode interrupts can settle I/O tasks through real-time interrupts, and has a lower I/O latency than polling.

Finally, this paper conducts an experimental evaluation of user-mode interrupts.

The Ping-Pong experiment is used to test the functionality and performance of user-mode interrupts as IPC. Experiments show that under the interference of clock interrupts and kernel scheduling, the Ping-Pong latency of user-mode interrupts can reach 25% of the fastest existing IPC method, and the worst case is basically equal to it. At the same time, this article also tests the I/O latency of different notification methods of the asynchronous I/O framework. The experimental results show that user-mode interrupts will bring about 5% higher total time overhead than polling, but the settlement of I/O tasks can be completed in time through interrupts, reducing the latency of I/O tasks to about 15% of polling.

Key Words: User Interrupts; MicroKernel；Operating System

目 录

第一章　绪论

第一节　研究背景与挑战

近年来，嵌入式设备的数量快速增长。从工业领域的汽车、无人机等设备，到手机、mp4 等电子消费产品，再到冰箱、洗衣机等传统电器设备，到处都是嵌入式设备的身影。而嵌入式系统作为一种完全嵌入器件内部，只为特定应用设计的专用电脑系统，通常它的用户程序是一段带有特定要求的预定义任务。

通常情况下，嵌入式平台上运行的操作系统会根据应用的需要，对内核的组件部分进行定制化裁剪，去除应用功能外的部分。同时由于其任务和应用场景的要求，嵌入式平台对操作系统的可靠性、实时性和安全性等特性有非常高的需求 [1]。

可靠性是指嵌入式设备中的嵌入式系统需要长时间运行而不出现差错。如飞机中的嵌入式系统通常需要冗余性设计，保证其在飞行时不出现诸如崩溃重启等问题危害乘客生命安全。这一特点就要求嵌入式系统上的操作系统需要尽量避免运行时错误的产生，或者在发生错误时能够快速自动复位，并尽量避免在内核中使用没有经过验证的、不稳定的内核模块。

实时性是指操作系统能够在尽量短的时间内对外部的异步事件做出响应并完成处理操作。一次正确的响应操作不仅会要求系统在逻辑功能上完全正确的，也会要求系统在规定的时间内完成这些操作。通常来说，实时又分为硬实时和软实时，硬实时要求任务在规定时间内必须完成，这由操作系统来保证；而软实时要求事件响应是实时的，并按照任务的优先级，尽可能在短时间内完成任务 [1]。在对实时性要求高的平台上，操作系统应该首先调度一切可利用的资源完成有实时性要求的任务，其次才会去考虑提升操作系统的整体执行效率。

安全性具体是指要求嵌入式设备在与外部连接的过程中，其内部的数据不会偶然或被恶意地破坏、更改或者泄露，维持嵌入式系统中信息的保密性和完整性。

而操作系统作为底层系统软件，一直是计算机系统的重要组成部分。自操作系统诞生以来，关于操作系统的分类主要呈现为三类：宏内核（Monolithic Kernel）、微内核（Microkernel）和混合内核（Hybrid Kernel）。

宏内核通常以 Linux 为代表，其典型特征为所有内核模块都位于内核中，处于内核态的保护下，用户态程序需要经过执行系统调用 Syscall 指令陷入内核态

后，完成相应的功能相应和处理。在内核中，各个模块间不涉及特权级的转换，平等的以函数调用的方式互相调用完成自身功能，各个模块间通常有复杂的耦合关系。

微内核则以 L4 系列内核为主要代表 [2]，今年随着 Open Harmony 的逐步商用化，微内核也进入了商用内核阶段 [3]。与宏内核不同，微内核的主要设计模式是将不必要的内核模块移出内核，放于用户态中，内核只保留最基本的操作原语和 IPC 调用。各个内核模块间通过 IPC 通信完成用户程序服务。相对来说，微内核的内核较为简单，其主要的作用是处理有特权级要求的操作、全局（跨进程地址空间）资源调度、资源空间隔离等。

混合内核则以 Windows 为主要代表，其介于宏内核和微内核之间，对有性能和资源调度要求的模块置于内核中，其余模块仿照微内核的设计原则置于用户态。

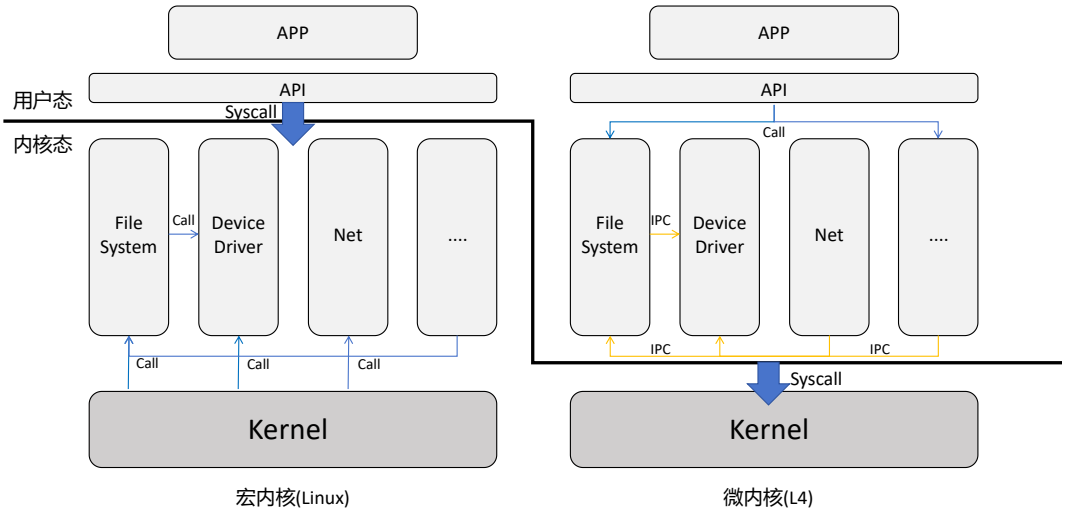


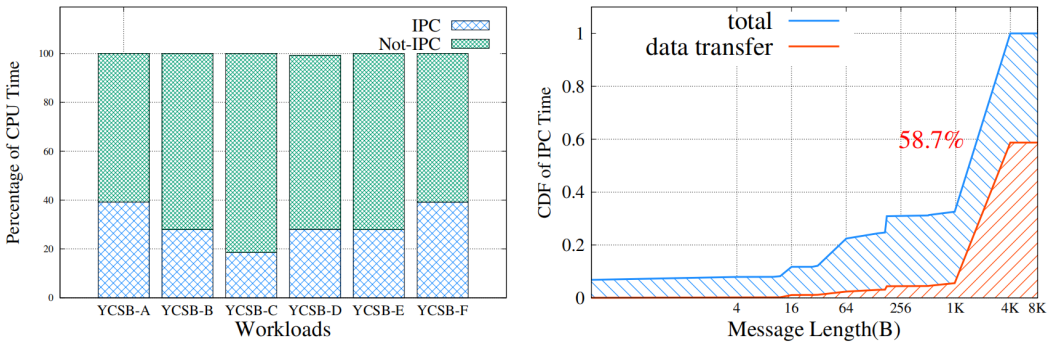
图 1.1　宏内核与微内核的内核架构区别示意

如图第一节所示，宏内核的功能模块全部位于内核中，彼此之间互相调用耦合，使得内核十分复杂。进而在嵌入式系统领域难以轻量化定制，同时过多的模块和耦合依赖也导致了其在可靠性、实时性和安全性方面相比微内核有所不足。随着微内核（microkernel）操作系统的理论和实现越来越成熟，微内核操作系统开始成为嵌入式平台一个很好的选择。

虽然历经了数十年的研究 [4][5][6][7]，微内核也形成了一些优秀的内核（如 seL4[7]，苹果 iOS 安全隔区上的 L4[2] 以及谷歌名为 Fuchsia 的下一代操作系统），但微内核仍然面临着安全性和性能之间的权衡，更细粒度的隔离通常会带

来更好的安全性和容错性，但也会导致更多的进程间通信（IPC），进而引起整体系统的性能降低。这也是微内核迟迟未能大规模商用的主要原因 [8][9][10]。

例如在英特尔 Skylake 处理器上，seL4 在快速路径下需要大约 468 个周期完成一次单向 IPC[11]，谷歌的 Fuchsia 操作系统的内核（称为 Zircon）完成一次往返 IPC 需要数万个周期。同时，IPC 作为不同功能模块之间完成系统服务的桥梁，其在系统运行过程中所占的比重也是非常大的。在 SiFive U500 RISC-V 开发板上，以 seL4 系统为底座运行 YCSB 系列的 Benchmark，如图 1.2 所示，可以看到有 18% 39% 的 CPU 时间用于了 IPC，同时对于每一次 IPC 而言，其主要的开销主要是两部分：域切换和消息传递 [11]。



(a) CPU time spent. **(b) IPC time on YCSB-E.**

图 1.2　YCSB Benchmark 结果 [11]

以往对微内核的研究，大多都在探索对 IPC 的优化方法，这些方法可以分为软件层面 [12][13][14][15] 和硬件层面 [11][16][17][18][19][20]。对于大多数软件解决方案而言，陷入内核的开销是不可避免的，而且消息传递要么会导致多次数据拷贝，要么会引发转换后备缓冲器（TLB）失效操作。而部分硬件解决方案，例如 CODOMs [21]，采用带标签的内存而非页表来实现隔离，它们运用单一地址空间以减少域切换和消息传递的开销。不过，这些新的硬件解决方案通常需要对现有的、为多地址空间设计的内核进行比较大的修改。

同时，以往的 IPC 大多为同步 IPC ，也即应用程序需要主动的显式调用相关接口进行检查并接收消息。即使是应用程序不感知的信号（Signal）机制，其本质不过是将显式轮询转移到了内核中，并没有改变其运行的本质逻辑。这一运行逻辑可以让编码的过程更为清晰和易懂，因为 IPC 的发生时机是程序员可定义的，它被固定在了一个确定的时机–即调用函数处。但在一些高性能异步 I/O 场景下，频繁的主动轮询会带来不必要的开销，同时强顺序的代码流也会让轮询等操作被其余部分的代码阻塞，引起延时升高，降低系统的实时性，这对嵌入式系统和高实时性要求的应用来说是致命的。

第二节　主要工作及贡献

reL4 作为微内核系统，依赖 IPC 为应用提供服务。但传统 IPC 需用户程序主动轮询，在 CPU 密集型任务占用时会导致消息延迟，影响实时性。尽管现有异步方案可以通过任务拆分缓解问题，但仍有部分局限性。本文创新性地引入用户态中断机制，利用中断的打断特性提升 IPC 实时性。

尽管当前的硬件支持软件中断，但实现复杂：需陷入内核态且特权级要求高，用户程序无法自定义处理流程。本文为 reL4 设计了一套易用的用户态中断范式，支持用户态直接转发中断并区分进程，通过回调函数实现高度自定义，这对泛用性是有很大帮助的。

Intel 于 2021 年推出 x86 用户态中断支持 [22][23]，但四年来主流内核鲜有适配。本文首次为微内核适配用户态中断的研究，比对宏内核的 Linux 实现后，结合微内核的内核设计特性与原则，为微内核架构的内核适配用户态中断提供了一个可行的内核实现方案。将宏内核上实现用户态中断所借助的一些微内核无法实现的特性（如 Linux 使用共享 Fd 建立链接），使用微内核所支持的方式实现，并验证其在功能上的等效性与有效性。

用户态中断作为一个新型 IPC 机制，它不仅仅可以用于简单的传递消息。结合微内核的特性，微内核的功能模块均位于用户态中，以用户程序的方式存在。如某个设备的驱动程序就以一个用户程序存在，其通知更高层的功能模块（如网卡驱动与网络协议栈）就可以采用用户态中断的方式。且由于中断的实时打断特性，其已最大化利用当前系统的所有资源来完成中断处理任务，这一实现可以最大化的提升系统的实时性。基于此背景，本文为 reL4 实现了一个支持用户态中断的异步 I/O 框架，以探索用户态中断在高性能 I/O 下的应用场景，它借鉴了 Linux 最新的高性能异步 I/O 框架 - io_uring。同时结合宏内核与微内核的架构差异，本文将 io_uring 进行了本地化微内核适配，将所有部件迁移入用户态，通过共享内存与用户代理 SQ 线程来全程在用户态完成服务，最后使用用户态中断作为 I/O 完成的通知机制。

最后，本文对用户态中断进行了详细的实验评估。为了测试用户态中断作为 IPC 的功能连通性和时延，本文设计了 Ping-Pong 实验测试其性能，并与 reL4 中已有的 IPC 方式（Slowpath 与 Fastpath）进行性能比较，最终 reL4 在其他中断（时钟中断）和调度影响下最劣化性能依然与 Fastpath 持平，最快可以达到 Fastpath 3-4 倍的性能。在实际 I/O 场景下，本文对 I/O 任务与 CPU 密集型任务场景下同步 I/O，异步 I/O，使用用户态中断的异步 I/O 三种方式进行测量，使

用用户态中断的异步 I/O 方式可以兼顾异步下的总时间减短和低 I/O 时延，证明其使用中断作为通信方式的优越性。

综上所述，本文的主要工作和贡献归结为以下五点：

1. 为微内核 reL4 引入新型 IPC 方式 - 用户态中断。
2. 为微内核 reL4 提供用户程序使用用户态中断的使用范式。
3. 为微内核 reL4 实现不同于 Linux 的用户态中断内核适配方案，为微内核适配用户态中断提供一个可能方案。
4. 仿照 Linux 的异步 I/O 框架 io_uring，为微内核 reL4 实现一个使用用户态中断的异步 I/O 框架。
5. 设计 Ping-Pong 实验测量用户态中断的性能，以实际网卡 I/O 测试用户态中断的异步 I/O 时延性能表现。

第三节　论文的组织结构

本文共分为六章，各个章节的组织结构如下：

第一章是绪论部分，主要介绍了目前商业界和工业界对于嵌入式系统-操作系统的选择和应用。指出了目前微内核的各种局限性和应用的挑战，进而引出了微内核的最重要瓶颈-IPC，并探讨了 IPC 的两种形式。介绍了 Intel 为 x86 架构引入的新型 IPC 通路-用户态中断的，并以此介绍了本文的主要工作与贡献。

第二章是背景知识介绍，主要介绍使用的微内核平台 reL4 的相应内核特性和实现与用户态中断的硬件规范与内核实现规范以及 IO 框架。此章节提供后续系统实现的规范支撑，是主要工作的实现主要遵守规范，也是本文的主要研究特性的重要说明书。

第三章是系统设计与实现，主要介绍对微内核 reL4 的改造，为其适配用户态中断的具体工作。分为用户侧的使用和内核侧的实现。

第四章是对用户态中断在 IO 中的应用，实现的 IO 框架的介绍。第一节介绍 IO 框架的具体实现和各个部分的结构。第二节介绍用户态中断作为通知机制在 IO 框架中的具体应用。

第五章是实验评估部分。通过 Ping-Pong 实验测试用户态中断的 IPC 性能，并使用数据对 IO 过程进行测试，测试用户态中断在 IO 中的 IO 时延。

第六章是总结与展望。总结了本文的主要工作，并基于学界现有的工作和成果，对本文工作作未来的展望。

第二章 背景知识

第一节 微内核 reL4

reL4 是 seL4 的 Rust 语言重写版本，但很遗憾，这项工作目前还在进行中，对于 x86 架构的适配还远没有完成，本文所使用的更接近于其原本的内核实现 - seL4（它是 C 语言和 Rust 语言并存混合编译的现状）。reL4 的原 C 语言内核是 L4 微内核家族的成员，这个家族可追溯到 90 年代中期.

操作系统微内核是操作系统的最小化内核，它将高权限代码的数量降到最少。与用户态程序不同，OS 可以使用 CPU 的特权模式（kernel mode），这意味着 OS 可以直接访问硬件。而用户态程序只能使用用户模式（user mode），仅可以访问 OS 允许它访问的的硬件。

事实上，它几乎不提供任何服务：它只是对硬件的一个薄薄的包装，仅仅足够安全地复用硬件资源。微内核主要提供的是隔离，即一个程序可以在不受其他程序干扰的情况下执行的沙箱。同时它提供了一种受保护的过程调用机制（出于历史原因，称为 IPC）。这允许程序安全地调用另一个程序中的函数。微内核在程序之间传输函数的输入输出并控制接口使用权：函数只能在一个导出的入口点被调用，并且只能被授权的客户端调用。

因此，它提供了一个正式的、数学的、机器检查的形式化验证，这意味着就其规范而言，内核很大程度上上是“没有 bug 的”。事实上，它也是世界上第一个经过形式化验证的系统 [7]。除了实现上的正确性之外，它还进一步提供了安全性验证 [24]。在一个正确配置的 reL4 系统中，内核保证了机密性、完整性和可用性等经典的安全属性。

除此之外，reL4 继承了通过能力（Capability）进行安全高效的访问控制的机制。能力是访问令牌，它可以对哪个实体可以访问系统中的特定资源进行非常细粒度的控制。他们根据最低特权原则（也称为最低权威原则，即 POLA）设计，提供强大的防盗能力。这是高安全系统的核心设计原则，在主流系统（如 Linux 或 Windows）中是不可能实现访问控制的。

最后，reL4 也继承了安全和性能兼顾的表现。在性能上，它与两个开源的微内核系统（Fiasco.OC 和 Zircon）相比，性能胜过任何其他微内核。reL4 的 C 语言内核的 IPC 成本比内核进入、地址空间切换和内核退出的硬件限制大约高出 10% 20%。Fiasco.OC 比 reL4 的 C 语言实现慢 2 倍以上（接近硬件极限的 3

倍），Zircon 的速度几乎是 reL4 的 C 语言实现的 9 倍 [17]。同时也有学者比较了 CertiKOS 和 reL4 的 C 语言实现的性能 [25]，在 CertiKOS 中实现双向 IPC 花了 3820 个运行周期，而在 reL4 中花了 1830 个周期。然而，事实证明，sel4bench（seL4 基准测试套件）在处理 x86 上的计时器时存在一个 bug，导致了夸大的延迟。正确的 reL4 的 C 语言内核性能值大约是 720 个循环，比 CertiKOS 快 5 倍以上。另外 CertiKOS 提供特性非常有限，也没有基于能力的安全性 [26]。

第二节 reL4 的 IPC

2.2.1 Slowpath IPC

进程间通信（IPC）是用于在进程之间同步传输少量数据和能力的微内核机制。在 seL4 中，称为端点（endpoint）的小型内核对象促成了 IPC，端点充当的是通讯的端口。端点对象的调用通常被用于收发 IPC 数据。端点们由一系列的等待收发信息的线程组成。例如：n 个线程正在一个端点上等待一个信息。如果 n 个端点在一个端点上发送信息，那么 n 个在该端点上等待信息的线程都将会收到这些信息且被唤醒。如果此时有第 n+1 个发送者发送一个信息，那么这个发送者将会排队等待。

一个线程可以通过系统调用（如 Send）在端点上发送信息，它将被一直阻塞到信息被其他线程接收。非阻塞系统调用 NBSend 也可以用，它将会执行轮询发送：该发送仅当已经有一个线程阻塞等待一个信息的时候才会发送成功，否则就失败。为了避免出现反向通道，NBSend 不返回指示消息是否已发送的结果，这将可以有效的避免安全问题。

同样的，系统调用 Recv 可用于接收消息，NBRecv 可用于轮询接收消息。他们与 Send 系列调用互为一组，完成了微内核中最基本的 IPC 实现。

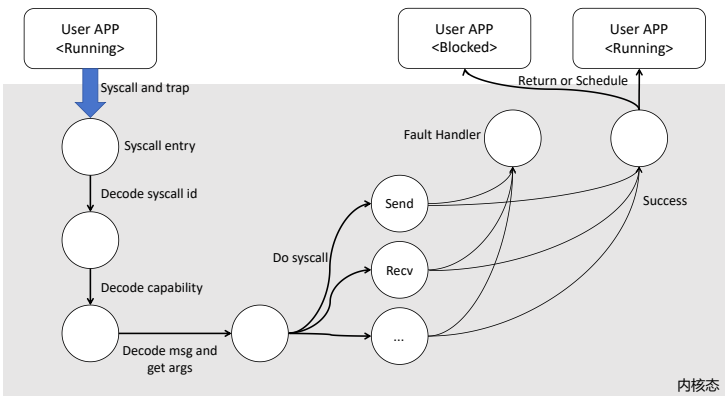


图 2.1 Slowpath 的基本 IPC 系统调用执行流程图

正如图 2.1 所示，系统调用 Send、Recv 等归属于“系统调用”大类中，与其他系统调用在同一路径上处理。由于 slowpath 与其他内核对象的 invocation 共用（这里的 invocation 即为系统调用路径的入口点），因此会进行层层解码，根据传入的系统调用参数进行不同的处理，最后将解码出的参数按对应系统调用号进行分发，此时才会进入到对应的处理函数。

此路径被称为“Slowpath”，它是与下面要介绍的“Fastpath”相对的。而它的“slow”之处正是在于其过于规范化的执行路径，这对于提升性能是非常不利的。

2.2.2 Fastpath IPC

对于 IPC 而言，因为其执行的频繁性，因此我们可以考虑将其创造一条单独的执行通路，不在与其他内核对象的 invocation 共用，单独产生一条新的、经过优化的路径，这样可以针对 IPC 调用过程做单独的、针对性的优化。这无疑是对提升系统整体性能的最佳选择。

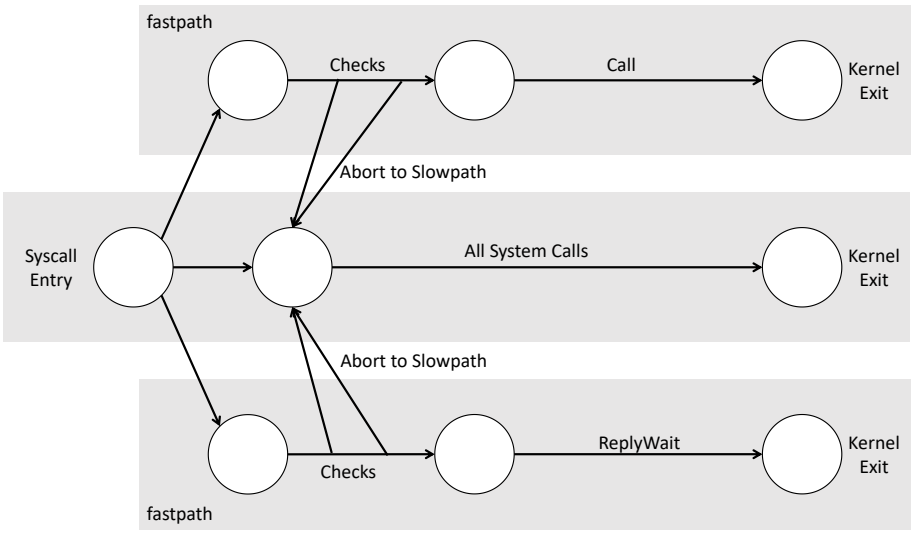


图 2.2 Fastpath 执行流程图

图 2.2 展示了 Fastpath 的处理流程，他是独立于其他系统调用的一条单独处理路径，它仅仅服务于 Call、ReplyRecv 两个系统调用，上文的 Send、Recv 系统调用默认全部走 Slowpath 路径，在应用程序通过”syscall” 或”ecall” 等指令进入内核态后，会检查系统调用号，仅有 Call、ReplyRecv 两个系统调用才能进入。

系统调用 Call，本质上结合了 Send 和 Recv，但是有一个主要区别：在接收阶段，使用此函数的线程被阻塞在一个一次性能力上，称为回复能力（reply

capability），而不是在端点本身上。在 client-server 场景下，客户端使用 Call 发出请求，服务器可以显式的回复正确的客户端。

回复能力（reply capability）内部存储在接收者的线程控制块（TCB）中。系统调用 Reply 会调用这个能力，向客户端发送一个进程间通信（IPC）并将其唤醒。ReplyRecv 也执行相同操作，但它在一个组合的系统调用中发送回复并阻塞在提供的端点上。由于线程控制块（TCB）只有一个空间来存储回复能力（reply capability），如果服务器需要处理多个请求（例如在硬件操作完成后再进行回复），可以使用 CNode_SaveCaller 将回复能力保存到接收者的 CSpace 中的一个空槽中。

当识别完系统调用号之后，还需要经过一系列的检查以确保在快速切换下内核不会出现错误，具体来说，检查项如下：

- 消息长度不大于某个值（与体系架构相关）。
- 当前线程没有未处理的错误。
- 是在 endpoint 能力上调用的 IPC 且该能力可以发送消息。
- endpoint 对象处于 Recv 状态。
- 目标线程的 vspace（虚拟地址空间对象）合法。
- 目标线程是当前调度域中最高优先级或不低于当前线程优先级。

通过这些检查的 IPC 可以以最少的代码切换到目标线程，对于此过程，基本可以将内核看作了一个 CPU 驱动。如果检查不通过，那么会退化回 Slowpath 进行处理。

值得一提的是，Fastpath 是一个可选择的内核模块，是否启用 Fastpath 只会对内核的性能产生影响，对于内核的行为不会产生任何的影响。

2.2.3 Notification

reL4 脱胎于 L4 kernel，原始的 L4 仅使用同步的 IPC 机制作为通信、同步、和信号传递的唯一机制。同步的 IPC 避免了内核里的消息缓存和拷贝成本。在最简单的情况下，IPC 仅仅是进行上下文的切换，而没有任何的消息拷贝和传输。

虽然这种模型十分简洁且迅速，但缺点也显而易见：由于同步的限制，对于 C/S 结构的系统，我们只能强制使用多线程的形式来处理多个客户端的请求，这将导致线程同步的复杂性。

此外，对于多核架构，同步的 IPC 显然也不会适用，因为同步的 IPC 会导

致类似 RPC 的服务调用被顺序排放在一个核心上，导致资源浪费。

因此 reL4 引入 notification 机制，提供一个类似 Unix 的 select 的机制。notification 不是通过后门引入异步机制的，而是与同步 IPC 部分解耦的，虽然不是严格最小化的（没有提供其他机制不能模拟的功能），但它们对利用硬件的并发至关重要。

实现 Notification 需要引入 notification 内核对象，它与 endpoint 内核对象略有区别，它的内部会多一个数据字，它的每一位表示一个二值信号量。于此对应的，引入了 send_signal 、Wait 和 poll 系统调用，他们需要使用 Notification 的能力进行调用。

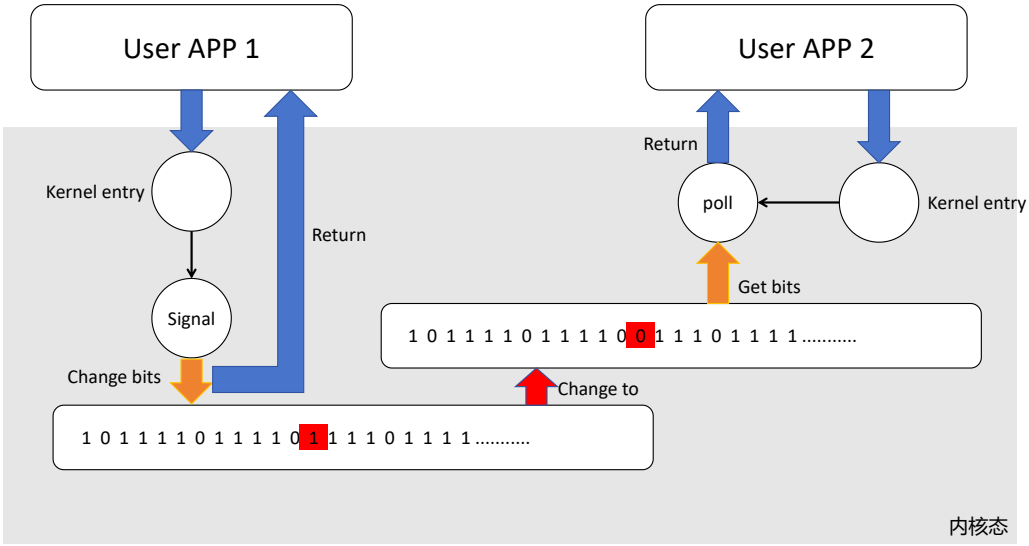


图 2.3　Notification 执行流程图

图 2.3 给出了实际执行流程。Notification 对象有三种状态：Waiting （有 TCBs 队列等待在该通知上等待被唤醒），Active（TCBs 已就该通知发出了数据），Idle （没有 TCB 排队，并且自上次设置为空闲以来没有 TCB 向该对象发出信号）。相应的系统调用则会根据信号量的状态改变当前线程的状态。

实际上，send_signal 、Wait 和 poll 系统调用只是对系统调用 Send、Recv 的封装，线程的行为状态因此不再赘述。其实现异步 IPC 的原理更类似使用 Notification 对象作为消息的中转暂存站，仅此而已，用户程序仍需显式的调用并有可能被阻塞。

第三节 用户态中断 Uintr

2.3.1 背景与介绍

硬件中断（Interrupt）是现在 OS 与设备交互的一种主要方式。设备有事情需要通知系统时（通知有请求到来，或有操作完成），会触发特定的中断引脚，CPU 则根据中断引脚调用指定的中断处理函数，实现 OS 和设备的交互（如硬盘与 CPU、网卡与 CPU）。

硬件实现的中断自然性能最好，延迟最小。但目前而言无论是触发中断，还是处理中断，都必须在内核态中处理。由于此，目前大多数涉及 I/O 操作的硬件设备，都需要在内核态实现驱动。如果需要在用户态实现驱动，则要借助 Linux 中的 uio 框架来转发硬件中断的信息，或使用 uio 框架把设备的地址空间暴露到用户态，以让用户态驱动直接读写。

而在高并发编程中，我们也常常遇到这样的场景：进程 1 需要等待进程 2 完成某事的标志，此时进程 1 可以选择轮询，也可以选择休眠，等待进程 2 完成后提醒进程 1（类似软件中断）。轮询需要消耗大量的 CPU 时间做无用的工作，延时最低，而软件实现的中断机制往往延时较大（依赖调度器的实现），不过由于等待通知的进程可以休眠，相比起轮询可以节省大量的 CPU 时间。

无论是怎样处理，都有各自的弊端，没有哪一种是最好的。因此引入一种全新的通知机制–用户态中断。让硬件实现的中断可以在用户态触发和处理。

用户态中断是 Intel 为降低相应时延而尝试添加的硬件拓展，它最早于 2021 年被提出并上市，搭载于第四代至强（Xeon）处理器。主要用于服务器端 CPU 的硬件，但在 2025 年，Intel 进一步扩大了用户态中断的搭载面积。总的来说，目前有下列架构的处理器全面搭载用户态中断 [22][23]：

- Sapphire Rapids 架构：至强系列 CPU，最早于 2021 年上市。
- Sierra Forest, Grand Ridge 架构：服务器端 CPU，消费端并不常见。
- Arrow Lake 架构：2024 年 10 月发布，2025 年所有消费端 CPU 的架构。
- Lunar Lake 架构：2024 年 9 月发布，2024 年及以后高端轻薄本 CPU。

用户态中断会被传递给运行在 64 位模式且当前特权级（CPL）等于 3 的软件，并且不会对分段状态造成任何改变。单个用户态中断由一个 6 位的用户中断向量来标识，该向量会在用户中断传递过程中被压入堆栈，作为用户中断传递的一部分。UIRET（用户中断返回）指令则用于撤销用户中断的传递。

简单来说，用户态中断特性非常容易理解，它不过是为存在了几十年的中

断机制添加了用户态的支持，让中断的应用面拓展到了应用侧。至于在细节方面，这项“新”特性也跟传统中断十分相似，但为了突破用户空间的隔离，它也有很大的不同。

2.3.2 用户态中断的数据结构

本小节会简单介绍用户态中断的整个流程会使用到的数据结构。有必要说明的是，这些数据结构需存在于内存中并且为它们创建 MMU 页面映射，因为他们是用户态下 CPU 跨进程访问资源，定位中断接收方的重要凭证。

在宏内核，如 Linux 中，因为 Linux 本身在内核态也提供虚拟内存管理和动态堆内存分配，因此这些数据结构的管理全部由用户通过系统调用委托给内核代为处理。但在微内核 reL4 中，内核中不提供虚拟内存管理，也不提供动态堆内存分配，因此需要将其转移到用户态，由用户程序预处理好后委托给内核，由内核进行需要特权级限制的操作，这一部分会在后续实现章节详细介绍。

对于用户态中断，其需要增加的部分主要分为三类：状态量、MSR 寄存器和数据结构 [23]。

首先介绍状态量：

- UIRR: user-interrupt request register，用户态中断请求寄存器。

此结构为一个 64 比特位值，其每一个 bit 都代表一个用户中断向量的启用与否。如果 UIRR[i] 为 1，那么说明向量为 i 的用户态中断正在请求服务。同时使用符号 UIRRV 来表示 UIRR 中已被置位的最高位是第几位。特别的，如果 UIRR 为 0，那么 UIRRV 也为 0。

- UIF: user-interrupt flag，用户态中断是否启用标志位。

此标志位用于标志用户态中断是否能被正常送达。如果 UIF 为 0，那么用户态中断会被阻塞。如果一个用户态中断被成功送达，那么 UIF 位会被清零，在执行 UIRET 指令返回后 UIF 会被重新置位为 1。

- UIHANDLER: user-interrupt handler，用户态中断处理函数地址。

此状态量为 64 位地址，代表了中断处理函数的地址。当用户态中断被触发后，用户程序会跳转到此地址执行中断处理。

- UISTACKADJUST: user-interrupt stack adjustment，用户态中断栈调整值。

该值控制在用户中断传递之前对栈指针（RSP）的调整。它可以被配置为使用备用的栈指针来加载栈指针（RSP），或者被配置为防止用户中断传递覆盖当前栈顶上方的数据。用户中断栈调整值（UISTACKADJUST）如果第 0

位为 1，在用户中断传递时会使用用户中断栈调整值（UISTACKADJUST）来加载为栈指针（RSP）；否则，会从栈指针（RSP）中减去用户中断栈调整值（UISTACKADJUST）。无论哪种方式，用户中断传递随后都会将栈指针（RSP）对齐到 16 字节边界。

- UINV: user-interrupt notification vector，用户态中断通知向量。

当用户中断通知到来时，以普通中断处理时的向量号。当 CPU 接收到用户态中断通知时，它会处理由 UPIDADDR 所引用的 UPID 中的用户中断。

- UPIDADDR: user posted-interrupt descriptor address，UPID 地址。
- UITTADDR: user-interrupt target table address，UITT 地址。
- UITTSZ: user-interrupt target table size，UITT 表的大小。

随后是 MSR 寄存器，它们可以通过 RDMSR 和 WRMSR 访问 [23]：

- IA32_UINTR_RR MSR (MSR address 985H)，UIRR 的存储 MSR 寄存器。
- IA32_UINTR_HANDLER MSR (MSR address 986H)，UINTR_HANDLER 的存储 MSR 寄存器。
- IA32_UINTR_STACKADJUST MSR (MSR address 987H)，UINTR_STACKADJUST 的存储 MSR 寄存器。
- IA32_UINTR_MISC MSR (MSR address 988H)，UITTSZ 和 UINV 的存储寄存器。Bit[31:0] 存储 UITTSZ，Bit[39:32] 存储 UINV，Bit[63:40] 是保留位，尝试写此保留区域会引起 GP 异常。
- IA32_UINTR_PD MSR (MSR address 989H)，UPIDADDR 的存储寄存器。其 Bit[5:0] 是保留区域，写此区域会引起 GP 异常。
- IA32_UINTR_TT MSR (MSR address 98AH)，UITTADDR 的存储寄存器。该寄存器的 Bit[63:4] 存储 UITTADDR，Bit[3:1] 为保留区域，Bit[0] 标志 SENDIPI 指令是否被启用。如要发送用户态中断，那么 Bit[0] 需被置位。

对于地址的存储，都会要求地址是一个线性地址，相对于处理器所支持的最大线性地址宽度，它必须是合法的。如果向该寄存器执行写操作（WRMSR）时，源操作数（相应地址）不满足此要求，将会引发通用保护故障（#GP）。

最后介绍一下需要使用到的两个主要数据结构 UPID 和 UITT[23]：

首先是 user posted-interrupt descriptor（UPID）。它被用于用户态中断的接收方。当用户态中断即将被产生时，我们需要一个描述符来唯一标记目标线程，因此引入此数据结构。在 C 中，它是以一个严格顺序的结构体来定义，并且内部

含有需要的字段值。同时它将在用户态中断的发送过程中被 CPU 从内存中读取，按照预先约定好的格式解析，获取所需的字段。其构成如下：

- Bit[0]：Outstanding notification。如果这一位被设置，那么在（PIR）中就存在一个或多个未处理的用户态中断。
- Bit[1]：Suppress notification。如果这一位被设置，那么此 UPID 将不能接受用户态中断。
- Bit[15:2]：Reserved。保留位，必须为 0。
- Bit[23:16]：Notification vector。用于发送用户态中断时使用。
- Bit[31:24]：Reserved。保留位，必须为 0。
- Bit[63:32]：Notification destination。目标的物理 APIC ID。如果 CPU 使用 xAPIC 模式，那么 Bit[47:40] 存储 8 位的 APIC ID。如果 CPU 使用 x2APIC 模式，那么存储 32 位的 APIC ID。
- Bit[127:64]：Posted-interrupt requests (PIR)。对应 UIRR，如果某一位被置 1，那么此向量号存在一个尚未被处理的用户态中断。

其次是 user-interrupt target table（UITT）。它被用于用户态中断的发送方，位于线性地址 UITTADDR 处。它由 UITTSZ + 1 个 16 字节的条目组成，每个条目称为 user-interrupt target table entry（UITTE）。当执行 SENDIPI 指令发送用户态中断时，CPU 会将指令的源操作数作为索引值，获得对应的 UITTE，以 UITTE 中存储的目标信息进行发送。对于每一个 UITTE，它的构成如下：

- Bit[0]：Valid。如果这一位被设置，那么此 UITTE 是有效的。
- Bit[7:1]：Reserved。保留位，必须为 0。
- Bit[15:8]：User Vector。用户态中断向量值，它的值必须为 0 ~ 63（对应于接收方的 UIRR 的 64 位中的第几位），因此 Bit[15:14] 必须为 0。
- Bit[63:16]：Reserved。保留位，必须为 0。
- Bit[127:64]：UPIDADDR。存储发送目标的 UPID 的地址。

用户态中断需要用到的状态量、寄存器和数据结构大致如上，细节部分不做过多介绍，详情可以查看 Intel 官方手册 [22][23]。

2.3.3 用户态中断的处理流程

对于用户态中断的处理，首先需要发送进程完成一系列初始化和配置之后，执行用户态中断的发送指令：SENDUIPI <index>。这一步是整个流程的开始。

SENDUIPI 指令通过在由 UPIDADDR 所引用的用户发布中断描述符（UPID）

中发布一个向量为 V 的用户中断（这里即会将 UPID 中的 Bit[127:64]：PIR 某一位置位），然后会将该 UPID 中指定的任何通知中断作为普通的处理器间中断（IPI）发送出去（此处需要借助 APIC，因此 UPID 需要存储 APIC ID），以此来发送一个用户中断。

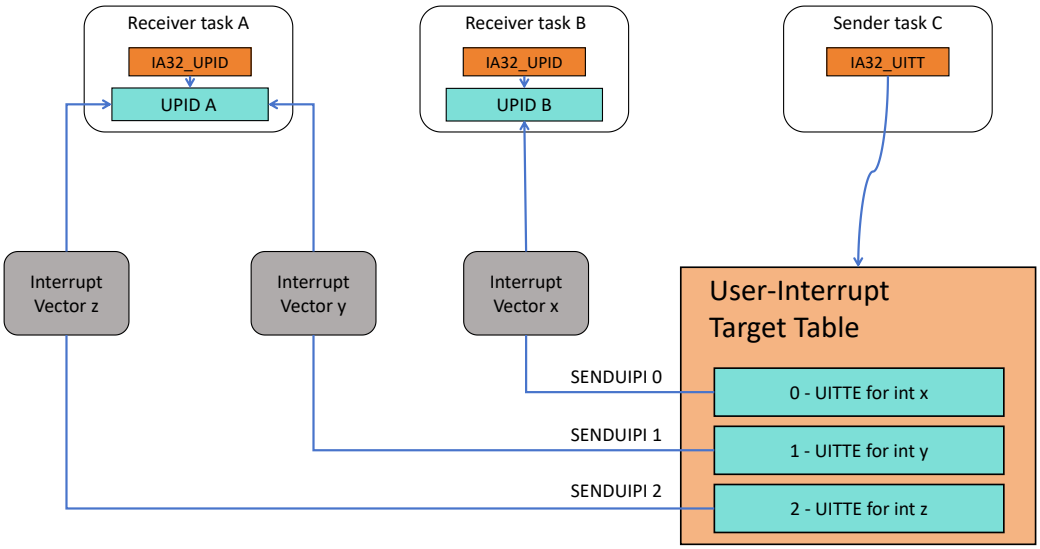


图 2.4　用户态中断发送指令 SENDUIPI 执行流程

如图 2.4 所示，SENDUIPI <index> 指令会通过置位目标的描述符中的一些字段来实现中断发送，且他发送的是一个普通中断（中断向量号为用户态中断向量号，但仍走普通中断处理通路）。

随后进入 User-interrupt posting （用户态中断发布）过程。如果 CR4 寄存器中的 UINTR 位等于 1 且 IA32_EFER 寄存器的 LMA 位也等于 1，那么当 CPU 接收到一个普通中断时，它会执行用户态中断通知识别操作（检查中断向量号）。其主要处理流程如下：

- 检查本地高级可编程中断控制器（APIC），这个操作可以获得当前 CPU 号（CPU ID）和中断向量号。
- 如果中断向量号等于用户态中断通知向量号（UINV），CPU 继续下一步操作。否则，此中断将通过中断描述符表（IDT）正常传递（随后进入传统意义上的内核的中断处理流程）；此算法的其余部分不适用，并且不会发生用户中断通知处理操作。
- 处理器向本地 APIC 中的中断结束（EOI）寄存器写入零；这会从本地 APIC 中消除向量 V = UINV 的中断。

必须说明的是，用户态中断通知识别操作涉及对本地 APIC 的检查和访问，

因此仅在普通中断未被屏蔽时才会发生。如果处理器完成了上面的所有步骤，那么会进入下一个步骤：user-interrupt notification processing（用户态中断通知处理）。

一旦 CPU 识别出一个用户态中断通知，它就会使用模型专用寄存器 IA32_UINTR_PD 中线性地址所指向的用户发布中断描述符（UPID）来执行用户中断通知处理操作，其具体流程如下：

- CPU 清除用户发布中断描述符（UPID）中的未处理通知位（第 0 位）。此操作是以原子方式进行的，以便不修改描述符的其余部分。
- CPU 将已发布中断请求（PIR，即 UPID 的第 127 位到第 64 位）读入一个临时寄存器，并将 PIR 的所有位都写入 0。此操作是以原子方式进行的，以确保 PIR 中被清零的每一位都在临时寄存器中被设置。
- 如果临时寄存器中的任何一位被设置，逻辑处理器就会在用户中断请求寄存器（UIRR）中设置与临时寄存器中被设置位相对应的每一位（例如，通过逻辑或操作），并识别出一个挂起的用户中断（如果它尚未这样做的话）。

CPU 以不可被中断的方式执行上述步骤。步骤 1 和步骤 2 可以合并为一个单一的原子步骤。如果步骤 3 识别出一个用户中断，处理器会在接下来的指令边界处传递该用户中断，进入 User-Interrupt Delivery （用户态中断传递）阶段。

必须说明的一点是，用户中断通知识别和处理可能在任何特权级别下发生，同时上面的步骤 1 和步骤 2 中的所有内存访问操作都是以超级用户特权级别来执行的。

CPU 会依据用户中断请求寄存器（UIRR）来判断是否有用户中断需要传递。一旦 CPU 识别出存在挂起的用户中断，它会在后续的指令边界处，通过引发一个与软件执行异步的控制流改变来传递该中断，进入中断处理流程。

当 UIRR 不为 0 时，表示一定有一个用户态中断等待被处理。正常情况下，有一些操作会改变 UIRR 的值，当这些操作执行完成时，UIRR 的值变的不为 0 了，那么这将导致处理器识别到一个用户态中断，如果操作完成后 UIRR 为 0，那么则不会触发中断处理。这里列出这些可能的操作：

- 使用 WRMSR 指令修改 IA32_UINTR_RR MSR 寄存器。
- 使用 XRSTORS 指令修改 Uintr 的 XSAVE 状态。
- 一个用户态中断被处理器递送到此 UPID 。
- 一个其他的普通中断被内核委托为 Uintr，作为 Uintr Notification 来发送。
- VMX 转换操作中加载了 IA32_UINTR_RR MSR 寄存器。

当一个用户态中断被识别后，会进入相应的执行流。当以下所有条件都满足时，该用户中断将在指令边界处被传递：

- CR4 寄存器的 CR4.UINTR = 1。
- UIF = 1。
- 不存在由 MOV SS 指令或 POP SS 指令造成的阻塞情况。
- 当前特权级（CPL）等于 3，即必须处于用户态，内核态不满足条件（仅此阶段对特权级有要求）。
- CPU 运行在 64 位模式下，即不支持 32 位系统。
- 用户软件不在内核保护区（enclave）内执行。

值得一提的是，用户中断的传递优先级仅低于普通中断。它可以将 CPU 从通过执行 TPAUSE 和 UMWAIT 指令而进入的状态中唤醒，但它无法唤醒处于关机状态或等待系统初始化处理器间中断（SIPI）状态的 CPU。

同时，整个用户态中断的处理过程都不会改变当前处理器的特权级，即会一直在 CPL = 3 下运行。如算法 2.1 给出用户态中断发送的整个过程的行为流程：

算法 2.1 用户态中断在接收进程被传递-送达的过程 [23]

```
if 当前分页模式下 UIHANDLER 非规范 then
    #GP(0);
end if
holdRSP := RSP;
if UISTACKADJUST[0] = 1 then
    RSP := UISTACKADJUST;
else
    RSP := RSP - UISTACKADJUST;
end if
RSP := RSP & ∼FH;
压入 holdRSP;
压入 RFLAGS;
压入 RIP;
压入 UIRRV;
if 当前特权级 3 下影子栈启用 then
    影子栈压入 RIP;
end if
```

18: **if** 当前特权级 3 下终止分支跟踪启用 **then**
19: IA32_U_CET.TRACKER := 等待终止分支;
20: **end if**
21: UIRR[向量] := 0;
22: **if** UIRR = 0 **then**
23: 停止识别所有待处理用户中断;
24: **end if**
25: UIF := 0;
26: RFLAGS.TF := 0;
27: RFLAGS.RF := 0;
28: RIP := UIHANDLER;

在用户中断传递过程中对堆栈的访问可能会引发错误（页面错误，或者由于违反规范形式而导致的堆栈错误）。在传递此类错误之前，栈指针（RSP）会恢复到其原始值（堆栈顶部以上的内存位置可能已被写入数据）。同时如果在用户中断传递期间发生了错误，用户态中断请求寄存器（UIRR）不会被更新，用户态中断标志（UIF）也不会被清除。因此，CPU 会继续识别到存在挂起的用户中断，并且在错误得到处理后，用户中断传递通常会再次发生。

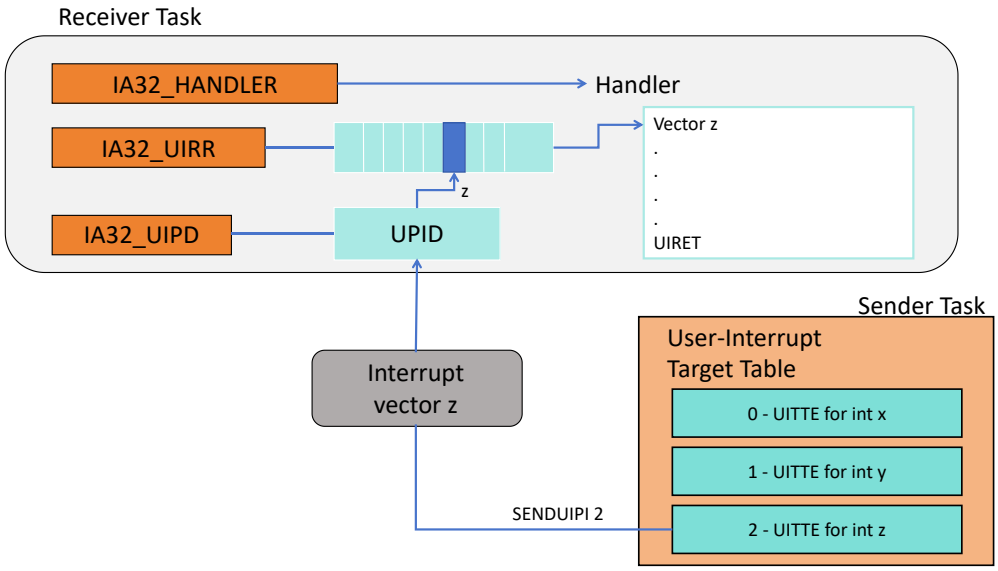


图 2.5 用户态中断的全部流程

图 2.5 概括了用户态中断的全部流程。从执行 SENDUIPI 指令开始，到引起中断设置 UPID 中的字段，最后根据 UPID 中的字段修改相应寄存器的值，最后进入中断处理函数，执行完后执行 UIRET 指令返回。

第四节 同步 I/O 与异步 I/O

2.4.1 传统的同步 I/O

在应用程序中，整个生命周期可以抽象为 CPU 密集型任务（如计算等指令）和 I/O 任务（与非 CPU 设备交互）的组合。CPU 密集型任务取决于处理器架构和制程设计，通常具有严格的同步顺序。而 I/O 任务相对于 CPU 密集型任务具有一定的可异步性，且不使用 CPU 资源，如采用同步 I/O 方式会大大浪费 CPU 资源，造成系统整体的效率降低。

在 Linux 中，目前绝大部分程序中的 I/O 操作都是同步的，最后通过 read/write 系列的系统调用实现对文件的读写。同步 I/O 的性能受限于文件类型和底层的设备性能，还有可能造成线程阻塞，在要求实时性的嵌入式系统场景下显然不能满足需求。为此，出现了多种异步 I/O 的实现机制，来解决实时 I/O 的问题。异步 I/O 是一种 I/O 接口类型，调用这种类型的接口进行 I/O 操作，可以在 I/O 操作实际完成之前就返回，而不会阻塞等待。

在 reL4 中，此问题同样存在。不同的用户模块之间需要使用 IPC 提交 I/O 请求，与 Linux 中使用 read/write 系列的系统调用本质上并无区别。即使使用非阻塞系列的 IPC 也仍会遇到轮询的开销问题，这在 Linux 的 POSIX 系列的 epoll 系列系统调用中也存在。

2.4.2 异步 I/O

因此高性能的系统往往要求使用异步 I/O 的方式来完成 I/O 类型任务，总的来说，目前常见的异步 I/O 实现方式有 3 种：

POSIX AIO。这种方案的实现可在 linux 手册中看到，且通过 glibc 实现，通过使用 pthread 库创建用户态多线程的方式实现异步 I/O 的接口，最后调用的 I/O syscall 仍然是内核的同步接口。在手册中写道，“这套机制局限性很大，最显著的一点就是使用多线程实现异步 I/O 的效率和可扩展性太差”。因此，手册中还提到了下一种异步 I/O 方案。

LINUX AIO。这套方案是 linux 内核提供的接口，名字也叫 AIO。这套方案是 io_uring 出现前 Linux 内核支持异步 I/O 的唯一接口，由于 glibc 中已经提供了同样名为 AIO 的机制，因此没有为其提供封装，需要通过专门的 libaio 库进行调用。且 Linux AIO 存在很多缺陷：AIO 最大的局限性是只支持 direct I/O，这不仅导致程序无法使用 cache ，更让程序无法使用普通的 malloc/new 等方式分配的内存用于 I/O，而必须使用 mmap 方式直接分配 4K 对齐的页。在绝大部分

情况下，这显然是无法实现的，因为即使开发者在自己的代码中管理内存分配，却无法控制引用的各类库和包的内存管理方式，因此在大部分情况下 AIO 没有实用价值。此外 AIO 在某些场景下仍然会退化为阻塞调用，其接口传递的参数过大因此效率也不高。因此 linux 社区一直认为“这套 AIO 机制不够成熟，还不能将 glibc 中的 POSIX AIO 接口改用这套机制实现”。

事实上，当前主流的异步 I/O 模型，大多是程序直接根据程序逻辑实现的。例如异步日志，就是在主要的工作线程中将需要写的日志内容写入一个专门的 buffer 队列中，再由另一个专门的日志写入线程将队列内容写入文件。也有一些库能够实现这样的功能，例如 log4j2。本质上这种实现与 POSIX AIO 的原理是一致的，但通过结合程序的实现功能需求和配置，就避免了通用的 AIO 机制带来的过多线程创建管理开销。

2.4.3 宏内核 Linux 的 io_uring

现有的内核 AIO 机制经过多年的开发，仍然存在大量的问题。Linux 内核维护者认为已有机制问题太多，使用场景太少，早期设计时没有考虑到异步 I/O 的实际需求，在其基础上开发新特性已经没有意义。因此 Linux 开发者决定针对 AIO 机制开发中发现的一系列问题，从头开发一套新的异步 I/O 机制，称作 io_uring。io_uring 的高效性就是建立在使用用户态 (user-space) 可访问的无锁环形队列 (ring) 的基础之上的。此机制作为 Linux 目前主干中最新、性能最高的异步 I/O 框架，是 Linux 探索高性能内核的重要组成。

io_uring 的原理是让用户态进程与内核通过一个共享内存的无锁环形队列进行高效交互。相关的技术原理其实与 DPDK/SPDK 中的 rte_ring 以及 virtio 的 vring 是差不多的，只是这些技术不涉及用户态和内核态的共享内存。高性能网络 I/O 框架 netmap 与 io_uring 技术原理更加接近，都是通过共享内存和无锁队列技术实现用户态和内核态高效交互。但上述的这些技术都是在特定场景或设备上应用的，io_uring 第一次将这类技术应用到了通用的系统调用上。

具体来说，Linux 通过引入 SQ（提交队列）和 CQ（完成队列）来进行通信，内核线程与用户线程只需要将自己生产的请求放入这块共享内存的队列中，另一方就会作为消费者消费掉这个队列项，以此循环，不断的完成 I/O 任务。

在 Linux 中，io_uring 提供了 io_uring_enter 这个系统调用接口，用于通知内核 I/O 请求的产生以及等待内核完成请求。但这种方式仍然需要反复调用系统调用，进行上下文切换，并唤醒异步处理逻辑去处理请求。显然这种方式会产

生额外的开销，而且受限于系统调用速率，无法发挥 I/O 设备的极限性能。为了在追求极致 I/O 性能的场景下获得最高性能，io_uring 还支持了轮询模式。轮询模式在 DPDK/SPDK 中有广泛应用，这种模式下会有一个线程（也即 SQ 线程）循环访问队列，一旦发现新的请求和事件就立即处理。

对于用户态程序来说，轮询只需要一个线程持续访问请求完成事件队列即可。但这个层次的轮询只是轮询了 io_uring 的队列，但内核从 I/O 设备获取完成情况仍然是基于设备通知的（或轮询设备状态寄存器）。Linux 通过在初始化时设置 IORING_SETUP_IOPOLL 标志，可以在获取完成事件时，内核会使用轮询方式不断检查 I/O 设备是否已经完成请求，而非等待设备通知。通过这种方式，能够尽可能快的获取设备 I/O 完成情况，开始后续的 I/O 操作。

同时，在内核中还支持了一个内核 I/O 模式，通过 IORING_SETUP_SQPOLL 标志设置。在这个模式下，io_uring 会启动一个内核线程，循环访问和处理请求队列。内核线程与用户态线程不同，不能在没有工作时无条件的无限循环等待，因此当内核线程持续运行一段时间没有发现 I/O 请求时，就会进入睡眠。如果内核线程进入睡眠，用户态程序需要在有新的 I/O 请求时通过带 IORING_ENTER_SQ_WAKEUP 标识的 io_uring_enter 调用来唤醒内核线程继续工作。

需要注意的是，如果 IORING_SETUP_IOPOLL 和 IORING_SETUP_SQPOLL 同时设置，内核线程会同时对 io_uring 的队列和设备驱动队列做轮询。在这种情况下，用户态程序又不需要调用 io_uring_enter 来触发内核的设备轮询了，只需要在用户态轮询完成事件队列即可，这样就可以做到对请求队列、完成事件队列、设备驱动队列全部使用轮询模式，达到最优的 I/O 性能。当然，这种模式会产生更多的 CPU 开销。

第三章　rel4 中用户态中断机制设计与实现

第一节　用户依赖库与使用流程

对于用户态中断，用户程序如果要使用需要按照一定的流程进行。

首先用户程序需要完成相关内存结构的映射和共享，以便能够让发送线程完成对接收线程的数据结果的访问与修改，这是完成用户态中断的必须条件，具体过程在上一章的执行流程中已详细介绍。

在进入中断处理函数时，是通过一个类函数调用的方式进入的，因此接收线程在中断处理时可以正常执行对全局变量等该地址空间下的全部操作。包括执行系统调用陷入内核态、yield 让出控制权、创建一个新的子线程等等。

在上述操作之后，双方用户程序需要使用内核提供好的系统调用进行注册，将相关数据结构委托给内核进行处理，由内核将这些数据结构的地址（如 UPID 和 UITT）写入相关寄存器，并进行其他的操作与置位。

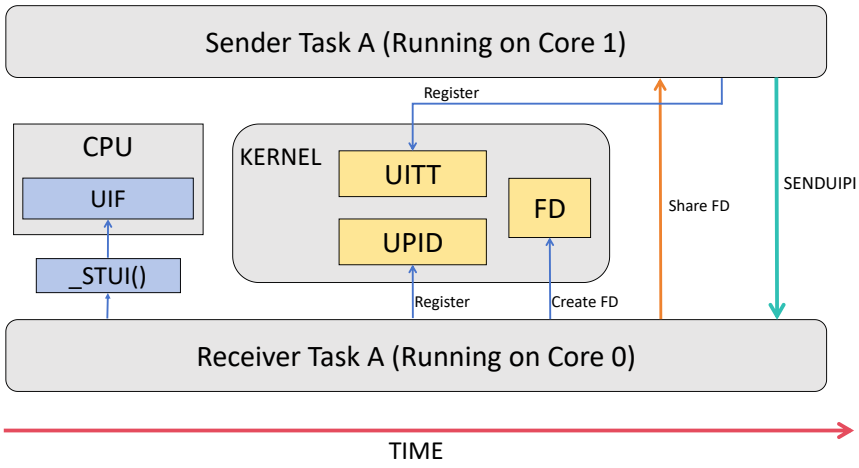


图 3.1　用户程序使用 Uintr 的全流程

如图 3.1 所示，给出了时间顺序下一对线程使用用户态中断的过程。对于接收线程，需要额外执行两个函数：_stui()，_clui()。这两个函数是 GCC 的编译器预留函数，他们分别设置 UIF 位和清空 UIF 位，如果不开启 UIF 位那么将不会收到用户态中断。对于发送线程，发送用户态中断无需通过内联汇编执行 SENDUIPI 指令，同样使用 GCC 的编译器预留函数：_senduipi(index)。用户程序的编译需要 GCC 11 以上的版本，同时需要包括用户态中断的头文件才能正常生成堆栈切换等操作。

下面对内核侧用户态中断的实现做介绍。

第二节 用户态中断机制的相关系统调用

应用程序使用用户态中断，需要初始化 UPID 和 UITT 结构，同时向部分 MSR 寄存器写入处理好的地址、向量号等内容。此操作需要用户程序通过系统调用委托给内核执行，因为相关结构需要以线程为颗粒度进行区分，只有内核才能跨用户程序地址空间访问所有的资源。因此定义如下的系统调用以供用户程序和内核调用：

- uintr_register_handler()。注册为用户态中断处理函数的接收方。
- uintr_unregister_handler()。注销用户态中断处理函数。
- uintr_vector_fd()。创建一个用户态中断的文件描述符。
- uintr_register_sender()。注册为进程间用户态中断的发送方。
- uintr_unregister_sender()。注销发送方。

上面的系统调用是一对用户程序完成一次用户态中断的发送和接收所必须的系统调用最小集。下面针对每个系统调用做详细的说明。

- uintr_register_handler(Uint64 handler_address, Uint32 flags, Uint64* addr)：

此系统调用由线程单位调用，且该线程将作为用户态中断的接收方运行。对于多线程进程，该用户中断处理程序仅为发起此系统调用的线程进行注册。其中 handler_address 为中断处理函数的地址；flags 为标志位，目前必须为 0，预留给未来可能的用处；addr 应为长度为 2 的数组，它保存需要被注册的 UPID 的虚拟地址和物理地址。

每个用户线程只能注册一个中断处理程序且每个想要接收中断的线程都必须进行一次注册。该注册不会在类似 fork() 的 TCB 复制操作或在同一进程内创建额外线程时被继承。一个线程一旦注册了中断处理程序，在通过 uintr_unregister_handler() 注销该处理程序之前，不能更改它。

在微内核平台，由于内核提供的功能非常少，且 reL4 内核中不提供虚拟内存管理（内核中的所有地址统一使用伪虚拟地址，即物理地址增加一个偏移量映射至 64 位空间的最高处），因此上述系统调用中注册相关的需要传入虚拟地址和物理地址。

在 Linux 中，内核可以调用 malloc 系列函数，其返回的地址为自动插入页表映射的虚拟地址，因此宏内核中相关系统调用的实现无需传入地址，任何结构（UPID 和 UITT）都由内核管理。但是在 reL4 中，内核中的所有内存是以

Untyped 类型管理，内核中没有动态分配内存的接口，相应能力被封装后移入用户态。

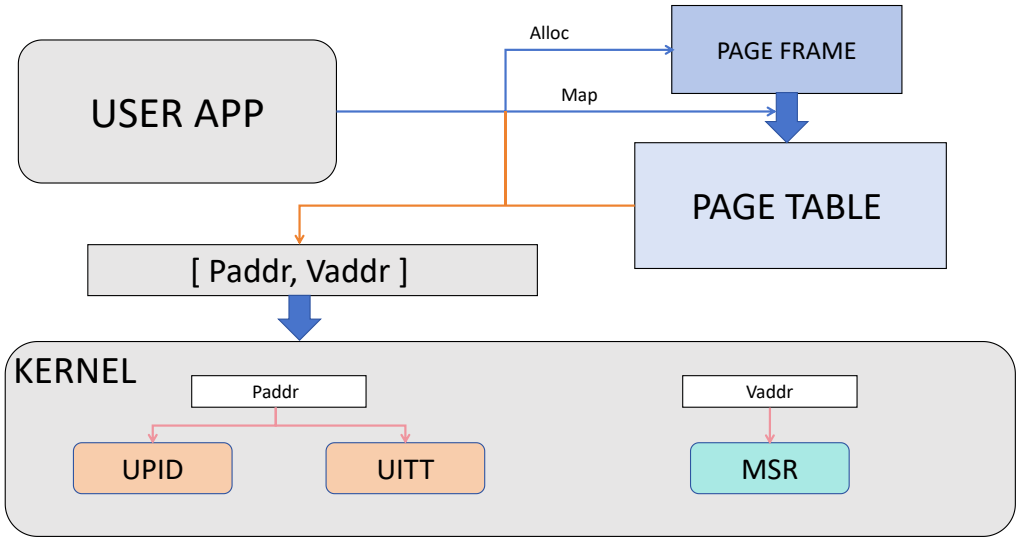


图 3.2　reL4 中系统调用的地址处理流程

而用户态中断的 CPL = 3 的特权级下处理会默认使用 MMU 页表映射访问 UPID 和 UITT，所以需要用户程序自行申请一块物理内存作为页帧，调用相关接口将此页帧插入页表，获得一个虚拟地址，最后将经过这些处理之后的虚拟地址和物理地址作为入参传入系统调用，内核直接使用这些地址进行用户态中断的相应注册操作。详细图示如图 3.2 所示。

- uintr_unregister_handler(Uint32 flags)：

此系统调用由线程单位调用，用于注销先前已注册的用户中断处理程序。其中 flags 为标志位，目前必须为 0，预留给未来可能的用处。如果线程此前没有注册过任何中断处理程序，uintr_unregister_handler() 将返回一个错误。同时，为该线程分配的诸如中断向量和 uintr_fd 之类的中断资源将被停用。其他向该线程发送中断的发送方所发送的中断将不会被传递。

- uintr_vector_fd(Uint64 vector, Uint32 flags)：

此系统调用由线程单位调用。入参中 vector 指示将要使用 PIR 的 64 位中的第几位；flags 为标志位，目前必须为 0，预留给未来可能的用处。它会基于调用进程所注册的向量来分配一个新的用户中断文件描述符（uintr_fd）。这个 uintr_fd 可以与其他进程和内核共享，以便它们能够使用关联的向量来生成中断。

调用者必须先通过 uintr_register_handler() 注册一个中断处理程序，之后才能尝试创建 uintr_fd。基于此 uintr_fd 生成的中断只会传递给创建该文件描述

符的线程。对于使用 uintr_create_fd() 注册的每个向量，都会生成一个唯一的 uintr_fd。每个线程都有一个私有的向量空间（即 PIR 字段），包含从 0 到 63 共 64 个向量。向量号 63 具有最高优先级，而向量号 0 具有最低优先级。如果有两个或更多中断等待传递，那么向量号较高的中断会先被传递，接着是向量号较低的中断。应用程序可以选择合适的向量号，来为某些中断赋予比其他中断更高的优先级。

当中断被传递时，中断处理程序会被调用，同时向量号会被压入栈中，以帮助识别中断的来源。由于向量空间是按线程划分的，每个接收者最多可以接收 64 种不同的中断事件。除此之外，接收者可以选择将同一个 uintr_fd 与多个发送者共享。由于使用相同向量号的中断都会被传递，接收者需要使用其他机制来准确识别中断的来源。

此处的 uvec_fd 命名是一个命名遗留问题，它最初出现于 Intel 官方给 Linux 内核适配用户态中断的相关规范和定义中。在 Linux 中，匿名 Inode 和匿名 Fd 是临时文件和共享内存传递的一个十分常见的手段，这里同样采用了这种操作，用于在接收方和发送方之间共享 fd。但是微内核 reL4 中内核不提供文件系统，相应的文件系统的实现和接口全部位于用户态，且文件系统并不一定存在，因此 reL4 中虽然沿用了此命名方式，但内核中的实现方式与 Linux 并不相同。

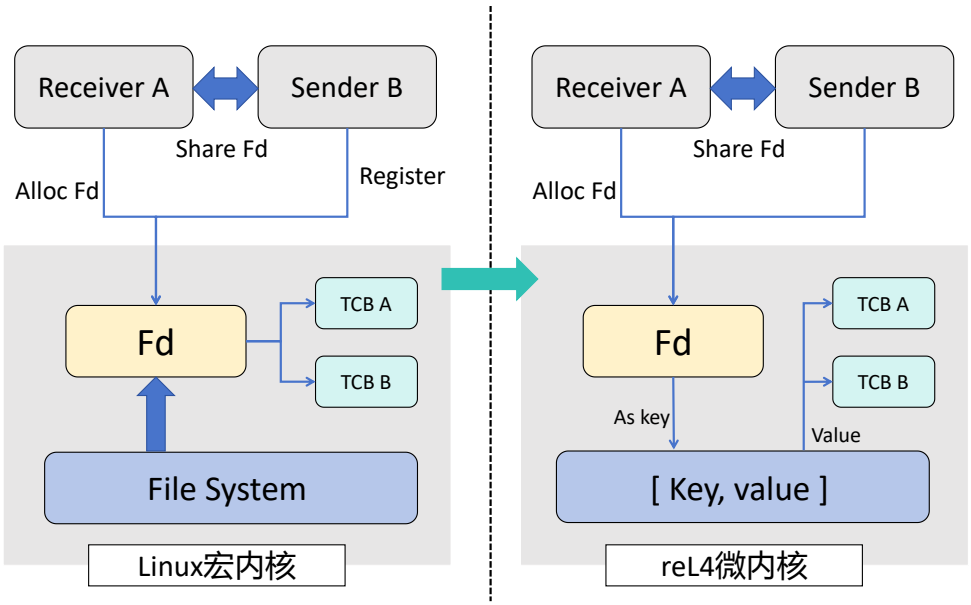


图 3.3　reL4 和 Linux 对 Fd 的不同实现

具体来说，Fd 的作用仅仅用于向发送方提供一个 token，让内核能够找到接收方的 TCB 而已，因此在 reL4 中使用 key-value 的键值对来维护。如图 3.3 所示。具体的是，将 fd 作为一个 key，接收方的 TCB ID 作为 value，每次申请 fd

时会创建一个新键值对然后返回此 fd，实现与 Linux 中类似的功能。

- uintr_register_sender(Int32 uvec_fd, Uint32 flags, Uint64* addr)：

此系统调用由线程单位调用。入参中 uvec_fd 为需要传入的 fd token，以此 fd 来定位和区分发送后哪一个线程会收到用户态中断；flags 为标志位，目前必须为 0，预留给未来可能的用处；addr 应为长度为 3 的数组，它保存三个地址：0 为接收方 UPID 的虚拟地址（需要将接收方的 UPID 映射至此线程，其物理地址必须一致，虚拟地址可以不一致），1 与 2 为发送方的 UITT 的虚拟地址与物理地址。

发送方线程基于 uintr_fd 与一个用户中断（Uintr）接收方建立连接。它会返回一个用户处理器间中断（IPI）索引（uipi_index），发送方进程可以将这个索引与 SENDUIPI 指令配合使用，以生成一个用户 IPI。

- uintr_unregister_sender(Int32 ipi_index, Uint32 flags)：

此系统调用由线程单位调用。入参中 ipi_index 为需要注销的索引；flags 为标志位，目前必须为 0，预留给未来可能的用处。

对于多线程进程，uintr_unregister_sender() 只会断开进行此调用的线程的连接。其他线程可以根据它们自己的 uipi_index 继续使用与 uintr_fd 的连接。

第三节　用户态中断的中断处理流程

3.3.1　接收线程非阻塞时

对于用户态中断，其主要执行流程中的发送与接收均在用户态下执行，此时接收线程与发送线程均处于用户态，且均拥有 CPU 资源。

此时我们假设相关结构均已正常初始化与注册，且 CPU 的相关标志位也已正常设置，那么会按照图 3.4 的方式来执行：

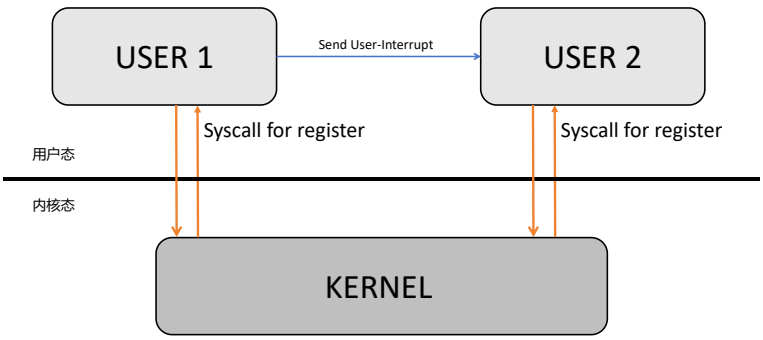


图 3.4　接收线程被阻塞时的情况

图 3.4 就是第二章所介绍的普遍与理想情况，具体流程已多次介绍，此处不再赘述。

3.3.2 接收线程阻塞时

由上一章所讲，用户态中断的全部执行流程必须在 CPL = 3 的特权级下运行，否则将不会引起用户态中断的送达。但是考虑下面的场景：接收线程因调度原因或其他阻塞原因被内核阻塞（毕竟接收线程和内核完全不知道何时会收到一个用户态中断，所以他们无法确保自己都能在用户态下接收中断），由用户态转换为内核态（或者称为阻塞态），那么此用户态中断是否被送达？该如何被送达？

发送方线程因为内核的隔离，导致其无法获取接收线程的状态，因此其执行 SENDUIPI <index> 指令时是无法确认接收方是否能够立刻被引起中断进入中断处理的（假设所有其他条件都满足）。但 SENDUIPI <index> 指令仍会执行，它会将目标 UPID 地址处的 PIR 的相应位置 1，并引起中断，也就是说对于发送方，接收线程是否处于用户态并不重要，发送方也无法感知到这个用户态中断是否发送成功，这一点有点类似于非阻塞系列的 IPC 逻辑。

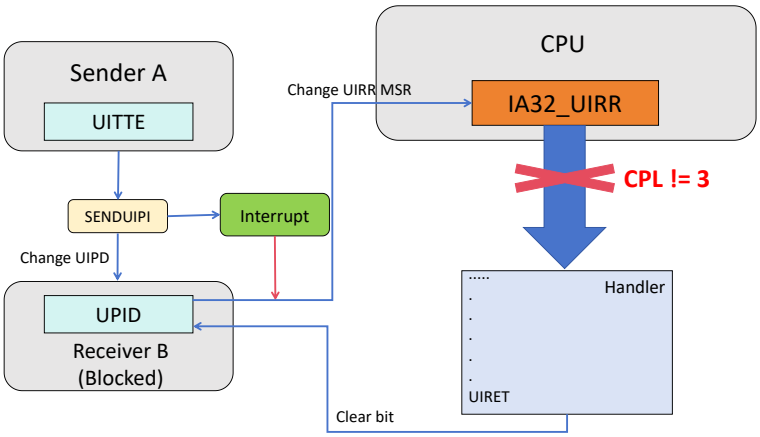


图 3.5 接收线程被阻塞时的情况

差异会在后面的环节发生，如图 3.5 中所示，因为当前 CPL 不满足条件，因此接收线程不会进入到中断处理部分，用户态中断无法被送达。且 PIR 中的对应位不会被清除，因此接收线程的 UPID 中的 PIR 字段是一定不为 0 的。它会一直存在直到内核手动清除此比特位或此用户态中断被成功送达。

对于用户程序而言，用户态中断应该是即时的、立刻被送达的，因此需要在此线程在第一次转为用户态时就触发此中断。为此，需要在内核的调度部分，

一个线程即将从 Ready 转换 Running 时，在内核中检查其是否为接收线程。如果为接收线程，会检查其 UPID 中的 PIR 字段是否为 0，如果不为 0，那么内核会手动向 APIC 发送一个中断号为 UintrV 的核间普通中断，此时会触发 17 页最后列出的触发用户态中断送达的条件中的第 4 条，继续之前没有被执行完的流程。具体如图 3.6 所示。

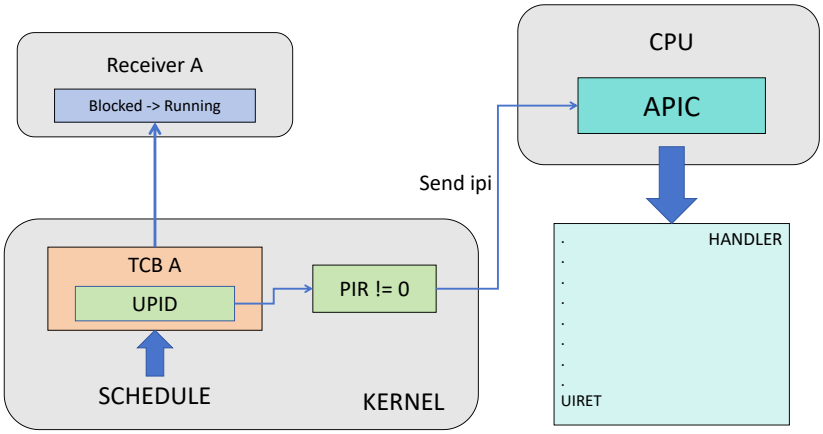


图 3.6　接收线程恢复用户态时的处理

虽然用户态中断的理想目标情况是接收线程在非阻塞时进行中断的接收，但事实往往会随着内核的复杂性提高而变得与预期不同。在很多较为完备的内核中，时钟中断与调度等方式会强制阻塞接收线程并进入内核态。如果不对用户程序做强制性干预，由内核代为发送用户态中断的情况会变为大多数。

第四章　支持用户态中断的异步 I/O 框架

第一节　微内核异步 I/O 框架 io_uring

仿照 Linux 中的 io_uring 设计，本文也在 reL4 中引入了一个基于微内核设计的 io_uring，但它与 Linux 中的在很多细节上并不相同，这是由宏内核和微内核的架构设计引起的，但在核心的部分尽可能的与 Linux 中的设计保持一致。同时由于一些原因，本文为 reL4 实现的仅为一个简单的主干，将 Linux 的 io_uring 的主要设计原则和特性完成了复现，但在内存屏障与保序等高级特性方面仍有很多的欠缺，并没有 Linux 中的那样完备。

reL4 中的 io_uring 由多个结构组成，它包含了许多独特的设计准则，下面对这些结构做简单的介绍，同时给出其与 Linxu 的宏内核设计的不同之处。

Linux 中一次系统调用会产生很多开销，因此为了最大程度的减少系统调用过程中的参数内存拷贝，io_uring 采用了将内核态地址空间映射到用户态的方式。通过在用户态对 io_uring fd 进行 mmap，可以获得 io_uring 相关的两个内核队列（I/O 请求和 I/O 完成事件）的用户态地址。用户态程序可以直接操作这两个队列来向内核发送 I/O 请求，接收内核完成 I/O 的事件通知。I/O 请求和完成事件不需要通过系统调用传递，也就完全避免了 copy_to_user 和 copy_from_user 的开销。

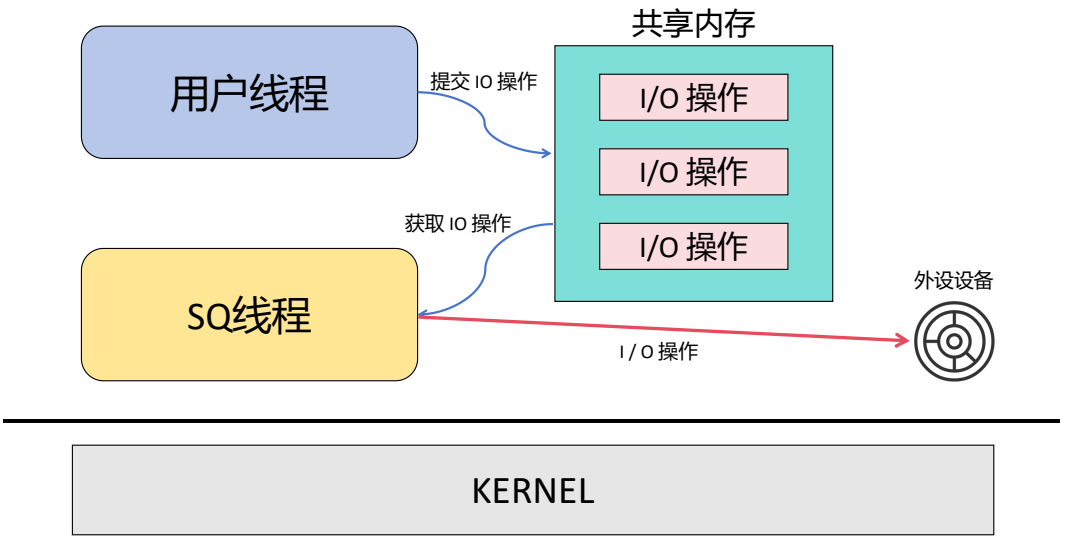


图 4.1　reL4 中的共享内存使用

但在 reL4 中，I/O 操作并不通过系统调用实现，而是用户程序通过 IPC 向

处于用户态的驱动程序提交一个请求，当驱动程序完成 I/O 操作后会调用回复系列 IPC 来完成一次操作。这一过程中，reL4 的开销主要集中在 IPC 上，跨进程的参数内存拷贝依然存在。但 reL4 与 Linux 的不同在于，其不会使用一个“内核线程”来完成 I/O 操作，而使用一个处于用户态的 SQ 线程，共享内存也不是内核与用户程序的共享，而是两个用户程序之间的共享内存。

io_uring 使用了单生产者单消费者的无锁队列来实现用户态程序与内核对共享内存的高效并发访问，生产者只修改队尾指针，消费者只修改队头指针，不会互相阻塞。需要注意的是由于队列是单生产者单消费者的，因此如果用户态程序需要并发访问队列，需要自己保证一致性（锁/CAS）。

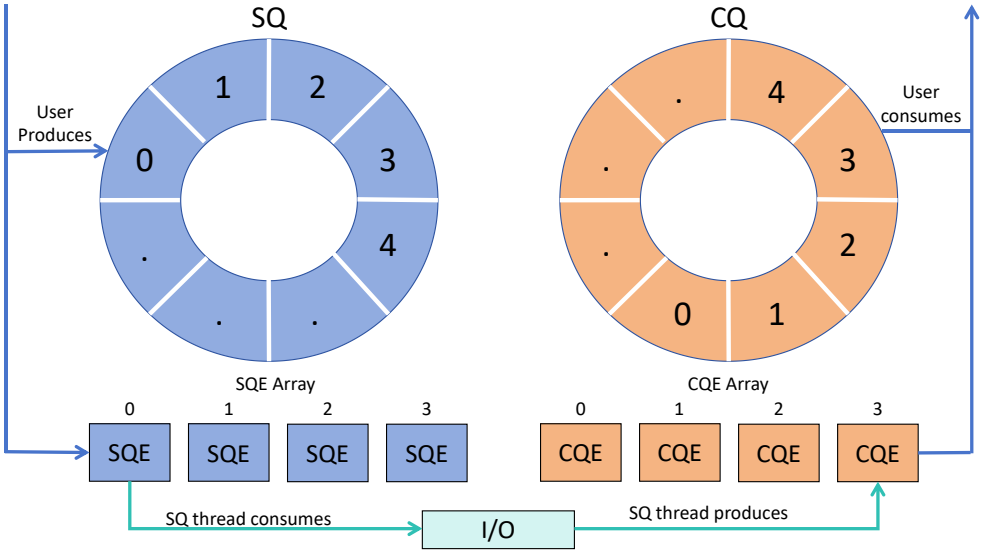


图 4.2　reL4 中的环形 SQ、CQ 队列（基于共享内存）

如图 4.2 所示，reL4 中含有两个队列：提交队列（Submission Queue, SQ）和完成队列（Completion Queue, CQ）。以及对应的每一项：提交队列项数组（Submission Queue Entry，SQE）和完成队列项数组（Completion Queue Entry，CQE）。

针对提交队列（SQ），应用是 I/O 提交的生产者（producer），SQ 线程是消费者（consumer）；反过来，针对完成队列（CQ），SQ 线程是完成事件的生产者，应用是消费者。

不同于 Linux 的队列，reL4 是两个用户程序的生产与消费，Linux 则是内核与用户程序的生产消费。Linux 采用此共享内存的目的是通过映射后直接访问来彻底消除系统调用的开销，reL4 则是以此来消除 IPC 的开销。

除此之外，这样做还可以支持 buffered I/O，充分利用缓存，减少数据碰盘

产生的系统延迟。

第二节 微内核中 io_uring 的通知机制

4.2.1 轮询

reL4 的 io_uring 支持用户线程通过轮询的方式来获取一个消息（这里指 I/O 完成）。

在 reL4 中，只实现了最优 I/O 性能下的轮询模式。SQ 线程会不断的轮询提交队列，查看是否有 I/O 需求需要完成，当获取到 I/O 请求之后会执行相应的 I/O 操作，当 I/O 操作完成之后会提交到完成队列。用户线程则在提交 I/O 请求之后，需要定时来轮询查看完成队列是否有 I/O 操作已完成。上述的操作完全通过共享内存进行通信，并没有额外的 IPC 通讯，实现了 I/O 性能的最大化，但会有比较大的 CPU 开销。详细的流程如图 4.3 所示。

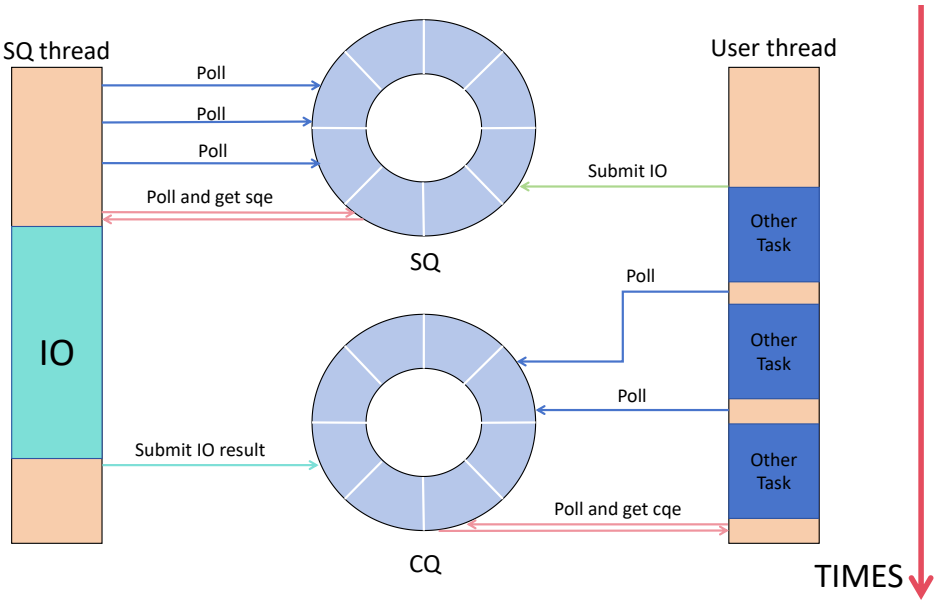


图 4.3 reL4 中的运行流程

注意到的是，在用户线程中，用户线程通常在 I/O 任务之外还有其他的任务需要执行，通常来说是 CPU 密集型任务（如计算任务等）。这些 CPU 密集型任务往往会在一段时间内占用 CPU 资源，且具有强顺序性，这也导致应用程序往往只能在这些 CPU 密集型任务结束之后才能轮询一次完成队列。这一段时间往往是具有不确定性的，如果一个任务执行时间过长，导致应用程序无法及时查看完成队列完成 I/O，那么会在应用程序方体现为 I/O 任务耗时过长。

这一局限性反映到真实应用中，如网络应用长时间集中于某个视频解码任务，导致一些网络数据包不能及时的处理，使网络性能呈现“假性下降”。类似

情况等等非常常见。

引起这一现象的原因是应用程序的主动轮询制度与协程（CoRoutine）十分类似，必须等待另一个协程（计算任务协程）执行完成后才能执行其他任务，而计算任务往往是不会主动让出控制权的，因此此现象在这种 I/O 模式下很难解决。

4.2.2 用户态中断

上一节所讲的基于轮询的异步框架下会引起异步任务的完成时延变长的问题，其本质是由于“完成异步任务”这一过程需要主动的 CPU 资源与不可让出 CPU 的其他任务之间的冲突导致。

而用户态中断可以提供不破坏上下文的打断机制，提供被动的 CPU 资源以检查完成队列。在使用轮询作为通知方式时，io_uring 虽然可以将 I/O 任务异步化，缩小总体时长，但是仍需要等待其他任务结束之后才能对 CQ 发起检查，且如果 CQ 轮询后没有完成任务，那么会引入多次轮询的开销。

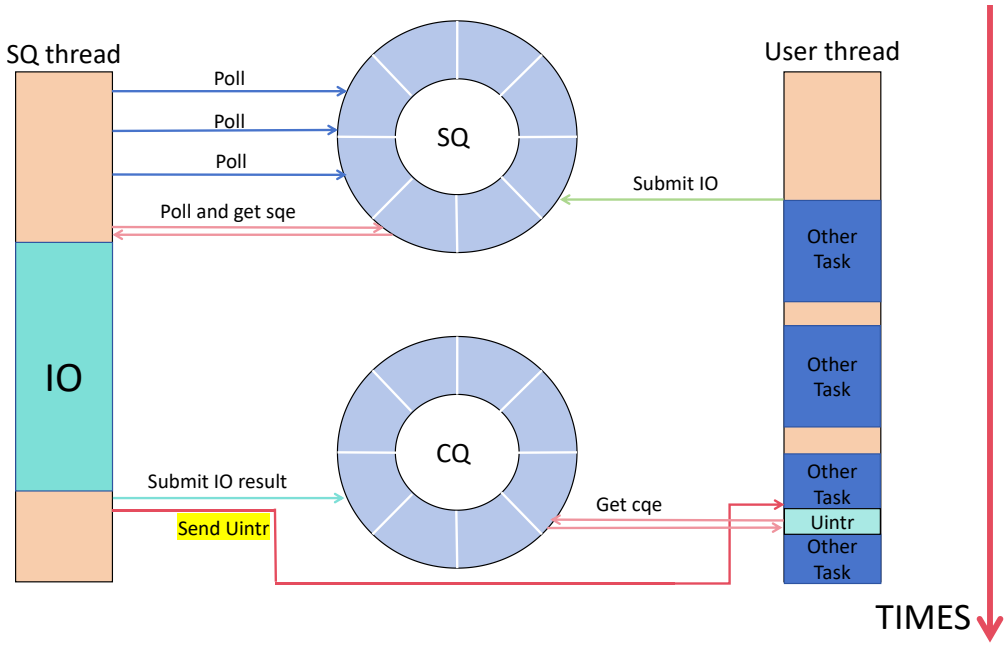


图 4.4 reL4 中的运行流程（加入用户态中断）

图 4.4 给出了加入用户态中断之后的 io_uring 的执行流程。用户程序不需要再去轮询，只需要在中断处理函数中定义好处理流程即可。且只要一次用户态中断的开销小于多次轮询的开销，那么这种方式就不会使整体的时长劣化过于严重。

通过引入用户态中断（Uintr），可以让用户程序放弃轮询，只需将检查 CQ 并完成 I/O 任务写入中断处理函数，即可在未来的某个时刻接收到用户态中断，进入用户态中断处理函数中完成 I/O 任务。因为中断的随时打断特性，使得用户程序可以在执行其他任务时将控制流转向，完成 I/O 任务后继续原本的任务，此过程由硬件确保上下文不被破坏和切换，效率是远高于软件的上下文切换的。

第五章　实验评估

第一节　实验设置

为了评估用户态中断在 reL4 中作为一种 IPC 方式的表现，以及基于用户态中断的异步 I/O 框架在实际的 I/O 任务与 CPU 密集型任务并存时的性能表现，本文设计 Ping - Pong 实验评估其作为 IPC 方式的表现，以实际的 I/O 任务与计算任务作为测试数据集衡量异步 I/O 框架的表现。

两个实验均使用相同的硬件平台，其详细参数如下：

- 宿主机：MacBook Pro 2018（M1 Pro）
- 宿主机环境：插电状态，仅开启 docker 与 vscode
- 虚拟机：Docker Engine（v27.4.0），Ubuntu 22.04
- 虚拟机平台：Qemu 6.2.94

第二节　用户态中断功能验证实验

为了评估 reL4 中用户态中断机制的设计与实现是否符合规范，能否正常完成预期的功能，设置功能性测试实验进行验证，检查用户程序能否正常使用用户态中断，能否在自定义的中断处理函数中完成用户程序的目的。

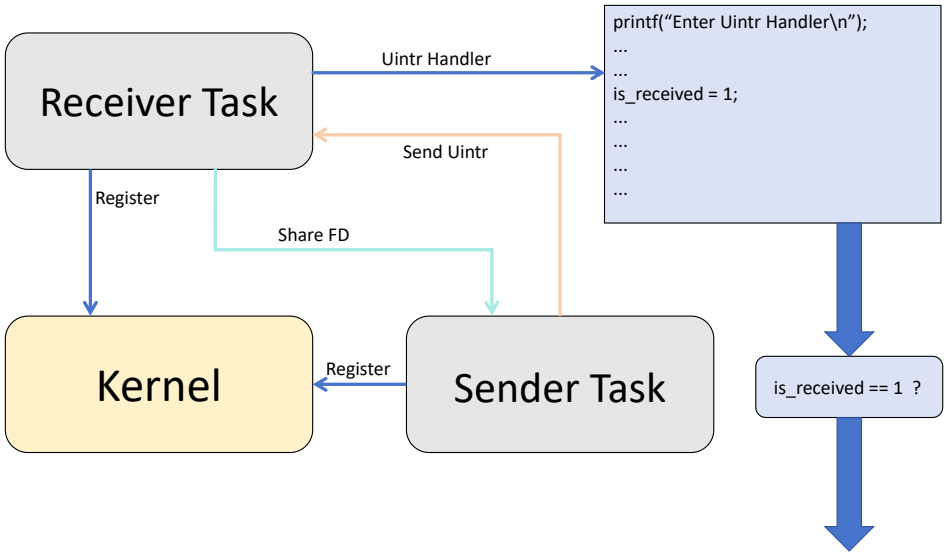


图 5.1　功能验证实验流程

如图 5.1 所示，展示了功能性验证的实验的设计过程，分别设置一个接收线程和一个发送线程，并使用相关系统调用向内核发起注册操作，完成用户态中

断的初始化工作。

随后接收线程自定义中断处理函数，函数中含有输出打印语句与一个标志位设置语句，用于标记接收线程成功接收到了用户态中断，以标记位检测是否正常进入了用户态中断处理函数中同时正常退出中断处理函数返回原位置。

同时，针对用户态中断的两个接收线程的不同状态，设置两组功能性验证测试：

- 测试一：单核情况，只能同时运行一个线程，接收线程和发送线程只能有一个正在运行，另一个线程休眠，对应于接收线程阻塞时的内核中断处理通路，测试由内核代为发送用户态中断是否成功。
- 测试二：多核情况，可以同时运行多个线程，接收线程和发送线程可以同时正在运行，对应于接收线程非阻塞时的内核中断处理通路，测试由 SENDUIPI 指令触发的中断全流程是否正常。

将上述测试接入 reL4 原有的测试框架中，随后根据不同的情况进行测试，测试结果输出如图 5.2 所示：

```
Running test UINTR0001 (Test uintr for basic send&recv on two core(Need SMP))
========== UINTR HANDLER START ==========
=========== UINTR HANDLER END ===========
Test UINTR0001 passed
Starting test 29: UINTR0002
Running test UINTR0002 (Test uintr for basic send&recv)
========== UINTR HANDLER START ==========
=========== UINTR HANDLER END ===========
Test UINTR0002 passed
```

图 5.2 功能验证实验结果

可以看到，测试程序成功运行，中断处理函数中的输出语句成功打印了语句，提示进入了用户态中断的中断处理流程，并成功修改标志位后返回原处。这证明 reL4 中用户态中断机制的设计与实现是正确的。

第三节 Ping-Pong 实验

在计算机领域，Ping - Pong 实验通常是用于测试网络连通性、测量网络延迟等方面的一种简单而有效的实验方法。在 IPC（进程间通信）领域，Ping - Pong 实验也有相关应用，主要用于测试和验证进程间通信的功能、性能和可靠性。

Ping-Pong 实验可用于验证不同进程之间是否能够成功建立通信通道。例如，在一个分布式系统中，通过让两个进程进行 Ping - Pong 式的消息传递，来确认它们之间的通信链路是否正常工作。如果发送进程能够成功发送消息，接收进

程能够正确接收并返回响应，就初步证明了 IPC 机制在功能上是可用的。

Ping-Pong 实验也可测量进程间通信的延迟。通过记录 Ping 消息发送和 Pong 消息接收之间的时间间隔，可以计算出数据在进程间传输的往返时间，从而评估 IPC 机制的延迟性能。这对于实时性要求较高的应用场景非常重要，如音频和视频处理系统，较低的延迟才能保证音视频的同步和流畅。

因此本文设计一次 Ping - Pong 实验，测量一次 IPC 下 Slowpath、Fastpath、Uintr 的功能表现和进程间通信的延迟。

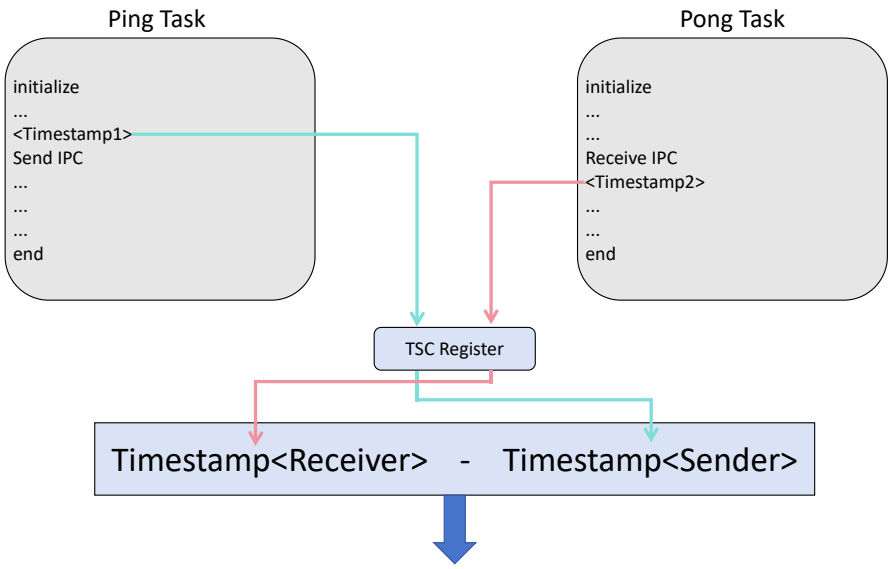


图 5.3 Ping-Pong 实验设计流程

如图 5.3 所示，展示了 Ping-Pong 实验的执行流程，它测试一次 IPC 所需要的时间，以此来评估 IPC 的性能是否优异。

同时实验中为 IPC 提供 Warmup，让 IPC 的链路延迟保持稳定，不出现震荡。计时则使用 CPU cycles 来计算，它通过使用 x86 架构下的 RDTSC 指令来实现，读取 CPU 自启动以来的时钟周期数，即处理器的时间戳。这是目前 x86 架构下常用的高精度的计时方法，在图 5.3 中也有做标注。

进行 10 次实验测量，以并计算平均值，实验结果如下表所示：

测试结果（Cpu cycles）

测试序号	Slowpath	Fastpath	Uintr
1	2344667	1071708	1164041
2	2430667	1158042	891333
3	2425292	1106042	995000
4	2286458	1142958	1064041

续表

测试序号	Slowpath	Fastpath	Uintr
5	2314125	1075875	1075666
6	2340708	1154125	1077791
7	2445542	1280833	368833
8	2440458	1146625	1119750
9	2420458	1149458	1113583
10	2339625	1161416	276916
Average	2378800	1144708	914695

如上表所示，reL4 中的 Slowpath 路径最慢，而 Fastpath 的时钟周期数减少到了 Slowpath 的一半，优化效果非常显著。这也说明了 reL4 中通用路径下对 Capability 和 IPC 参数的层层解码的开销是如此之大，开销基本等于完成一次 IPC 的总和。

而 Uintr 的实验结果产生了比较大的方差，最低可以做到 276916 个时钟周期即可送达，但大部分情况下是劣化到与 Fastpath 基本相当的性能水平。

分析其原因，主要由以下几点造成：

- CPU 资源限制导致内核产生调度，如此时单核情况下产生时间片调度，那么接收线程会被休眠，此时只有发送线程在运行，导致中断被延后，将在接收线程被重新调度入用户态运行时，由内核代为发送一个用户态中断。
- 其他行为导致当前核心陷入内核态，如此时虽然是接收线程正在运行，但此时产生了一个时钟中断，需要陷入内核处理时钟中断，而用户态中断必须在 CPL == 3 下才能继续执行，因此中断被延后，依然是在重新回到用户态时由内核代为发送一个用户态中断。
- 用户态中断优先级导致处理被延后。用户态中断的优先级是低于普通中断的，因此如果此时有其他中断一起到来，如外设中断、时钟中断、软件中断等等，那么 CPU 会优先触发优先级更高的中断，导致用户态中断的处理被延后，且此时有可能会进入第二种情况的处理。

结合实验结果，可以看到能够真正不受影响执行用户态中断的次数是比较少的，这在更为复杂的系统中可能更为严重，这也是用户态中断应用的一个问题。其如此设计的原因，推测更多是为了确保其他模块和功能的行为不被破坏或更改，让新特性对现有系统和模块的影响降到最低。

但总的来说，用户态中断在此情况下，最劣化的性能也与目前最快的 Fastpath 的性能相当，并有可能达到优于 Fastpath 几倍的性能，且独特的 IPC 方式也为应用程序提供了新的选择。

第四节 I/O 任务实验

本文设计此实验模拟 CPU 密集型任务和 I/O 任务共同存在时的用户程序行为。实验采用如下的设定, 将程序的任务抽象为 I/O 任务和计算任务, 这样的抽象符合绝大多数程序的设定。

对于 I/O 任务，本文使用 Intel e1000e 82574L 网卡，发送 ARP 数据包。对于 CPU 密集型任务，本文使用多次乘法运算。实验中的组别如图 4.4 所示，分为三组：

- 普通的同步 I/O 方式, 仅使用同步的方式完成 I/O。它并没有其他辅助线程以辅助其完成 I/O 任务，是单进程环境。
- io_uring + SQPOLL, 用户程序异步提交 I/O 请求到请求队列, I/O 完成后会由 SQ 线程放入完成队列, 用户程序需要显式的检查完成队列的完成情况。
- io_uring + SQPOLL + Uintr, 用户程序异步提交 I/O 到请求队列, I/O 完成后由 SQ 线程放入完成队列, 同时利用用户态中断的机制进行提醒, 用户程序无需显式调用相关的接口, 只需要在中断处理函数中处理完成的 I/O 即可。

同时，为了更明显的说明这三组 I/O 方式的差别，将每个任务给出一个假设的耗时，并在图中标出，下面即是每个 I/O 方式的具体图示：

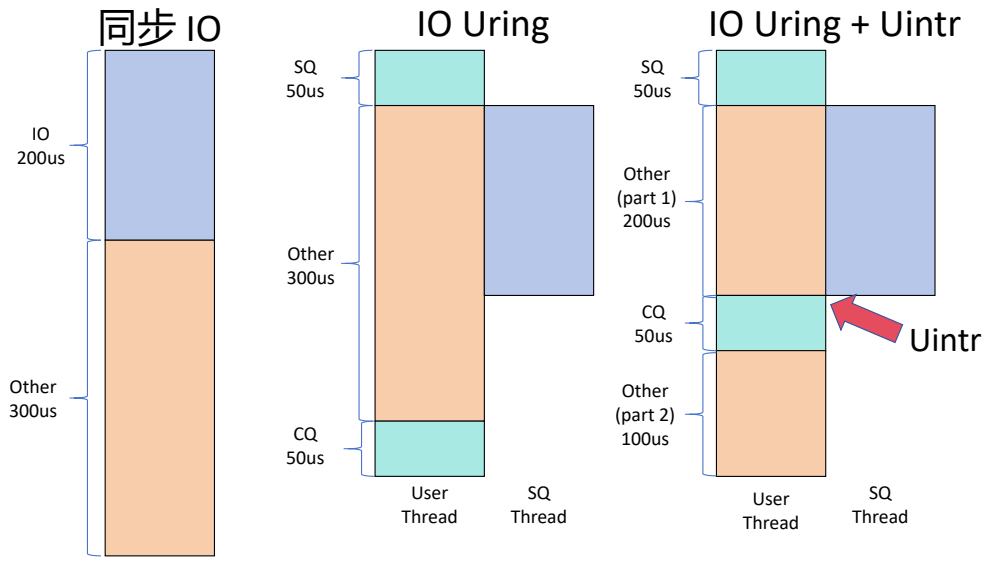


图 5.4 三种 I/O 模型

按照我们的设定, 我们预期使 io_uring 的方案会有更短的完成所有任务的时间（因为可以利用异步使得计算和 I/O 重叠）, 同时预期同步的方式会有最短的 I/O 延迟。而 io_uring + 用户态中断的实现方式既能利用 I/O 和计算的重叠提高总体的速度, 同时又能利用用户态中断的提醒机制达到短的延迟。

最终测量三组中的完成 I/O 任务、完成 CPU 密集型任务、完成所有任务的三个完成时间，计时依然使用 CPU cycles 来计算。采取测量 10 次后取平均值以保证数据准确性。最终测试结果如下表所示（表中数据均为平均值）：

测试结果（Cpu cycles）			
测试组别	I/O 任务	计算任务	总时长
Normal	10978424	29935812	40914237
io_uring	31931929	31205075	31931929
io_uring + Uintr	4711854	28867950	33579804

从实验结果可以看出实验符合预期，普通任务具有最长的总任务时长，且 I/O 任务因为使用同步 IPC 通路完成 I/O，使得 I/O 任务时间较长。引入 io_uring 之后，省去了 IPC 的开销，但是因为计算任务的阻塞导致延后一段时间后才能获取 I/O 结果，使得 I/O 时延上升。而引入 Uintr 之后，在保持异步 I/O 缩短总时长的情况下，通过中断的特殊通知特性，能够及时的完成 I/O 任务且没有同步任务的 IPC 开销，具有最短的 I/O 时延。虽然用户态中断的开销会大于轮询完成队列的开销，但整体开销的提升是可以接受的。

第六章　总结与展望

2025 年作为微内核（如 OpenHarmony）正式大规模商用和用户态中断硬件正式大规模铺开（Intel Lunar Lake & Intel Arrow Lake）的“双元年”，本文将用户态中断引入了微内核 reL4 中，探索基于用户态中断的新型 IPC 方式和应用。

用户态中断在用户态的两个用户程序之间传递中断，并支持用户程序自定义实现中断处理。微内核仅将必要的操作原语和进程隔离置于内核中，其余部分均为用户态的组件，这一特性天然契合用户态中断的使用环境。可以将 I/O 等驱动任务作为用户态中断中的接收方/发送方，实现用户态新型 I/O 范式。

本文依据此实现了一个使用用户态中断的异步 I/O 框架，它借鉴了 Linux 中目前最新、最高性能的的 io_uring 的实现思路，根据微内核做一定的适配改良，最终在实验中取得了符合预期的成果，证明了用户态中断这种新型 I/O 通知方法的有效性。在一些方面可以实现新的实现方式上的突破。

但用户态中断也存在些许不足之处，这些都是未来用户态中断真正应用需要考虑的地方。第一, 建立的链接数量有限, 硬件的中断向量位数只有 64 位, 接受方在当前的实现下只能区分 64 个不同的发送方, 这在一些情况下显然是不能满足需求的, 这个问题或许可以通过软件的方式进行解决; 第二, 用户态中断只是提醒机制, 且不方便使用, 用户态中断是指通过中断进行提醒, 中断函数中的内容需要由用户定义, 同时数据的传递也需要额外的方式进行处理, 如何让用户态中断更加易用, 编写更加方便和标准的接口也是值得探索的方向，且其对用户程序的破坏性重写也会引入很多的工作量。

参考文献

[1] 张荫芾. 基于多核处理器架构的嵌入式微内核操作系统的研究与设计 [mathesis], 2009.

[2] Gernot Heiser, Kevin Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. ACM Trans. Comput. Syst. Apr. 2016. https://doi.org/10.1145/2893177.

[3] Haibo Chen, Xie Miao, Ning Jia, *et al.* Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). Santa Clara, CA: USENIX Association, July 2024: 465–485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo.

[4] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, *et al.* Curios: Improving reliability through operating system structure. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. USENIX Association, 2019: 59–72.

[5] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, *et al.* Tappan Zee (north) bridge: Mining memory accesses for introspection. In: CCS 2013 - Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, 2013: 839–850.

[6] Bryan Ford, Mike Hibler, Jay Lepreau, *et al.* Microkernels meet recursive virtual machines. SIGOPS Oper. Syst. Rev. Oct. 1996: 137–151. https://doi.org/10.1145/248155.238769.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, *et al.* seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky, Montana, USA: Association for Computing Machinery, 2009: 207–220. https://doi.org/10.1145/1629575.1629596.

[8] D. R. Engler, M. F. Kaashoek, J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. Copper Mountain,

Colorado, USA: Association for Computing Machinery, 1995: 251–266. https://doi.org/10.1145/224056.224076.

[9] J. Liedtke. A persistent system in real use-experiences of the first 13 years. In: Proceedings Third International Workshop on Object Orientation in Operating Systems, 1993: 2–11.

[10] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In: Proceedings of the 2007 Workshop on Experimental Computer Science. San Diego, California: Association for Computing Machinery, 2007: 4–es. https://doi.org/10.1145/1281700.1281704.

[11] Dong Du, Zhichao Hua, Yubin Xia, *et al.* XPC: Architectural Support for Secure and Efficient Cross Process Call. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019: 671–684.

[12] B. Bershad, T. Anderson, E. Lazowska, *et al.* Lightweight remote procedure call. SIGOPS Oper. Syst. Rev. Nov. 1989: 102–113. https://doi.org/10.1145/74851.74861.

[13] Bryan Ford, Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference. San Francisco, California: USENIX Association, 1994: 9.

[14] Ben Gamsa, Orran Krieger, Jonathan Appavoo, *et al.* Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. New Orleans, Louisiana, USA: USENIX Association, 1999: 87–100.

[15] Jochen Liedtke. Improving IPC by kernel design. SIGOPS Oper. Syst. Rev. Dec. 1993: 175–188. https://doi.org/10.1145/173668.168633.

[16] Wenhao Li, Yubin Xia, Haibo Chen, *et al.* Reducing world switches in virtualized environment with flexible cross-world calls. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture. Portland, Oregon: Association for Computing Machinery, 2015: 375–387. https://doi.org/10.1145/2749469.2750406.

[17] Zeyu Mi, Dingji Li, Zihan Yang, *et al.* SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In: Proceedings of the Fourteenth EuroSys

Conference 2019. Dresden, Germany: Association for Computing Machinery, 2019. https://doi.org/10.1145/3302424.3303946.

[18] Jerome H. Saltzer. Protection and the control of information sharing in multics. Commun. ACM, July 1974: 388–402. https://doi.org/10.1145/361011.361067.

[19] Lluís Vilanova, Marc Jordà, Nacho Navarro, *et al.* Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC. In: Proceedings of the Twelfth European Conference on Computer Systems. Belgrade, Serbia: Association for Computing Machinery, 2017: 16–31. https://doi.org/10.1145/3064176.3064197.

[20] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, *et al.* Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. IEEE Micro, Sept. 2016: 38–49. https://doi.org/10.1109/MM.2016.84.

[21] Lluis Vilanova, Muli Ben-Yehuda, Nacho Navarro, *et al.* CODOMs: protecting software with code-centric memory domains. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. Minneapolis, Minnesota, USA: IEEE Press, 2014: 469–480.

[22] Intel. Architecture Instruction Set Extensions and Future Features, Programming Reference. In: 2024.12.

[23] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. In: 2024.12.

[24] Gerwin Klein, June Andronick, Kevin Elphinstone, *et al.* Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. Feb. 2014. https://doi.org/10.1145/2560537.

[25] Ronghui Gu, Zhong Shao, Hao Chen, *et al.* CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, Nov. 2016: 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[26] Gernot Heiser. The seL4 Microkernel, An Introduction. In: 2025.01.08.

致 谢

时光易逝，岁月如梭。行文至此，落笔为终。

最是人间留不住，朱颜辞镜花辞树。四年大学生活转眼走到了头，想来不免伤怀，思绪万千，目之所向，皆是回忆。

“饮水流者怀其源，学其成时念吾师”。我要特别感谢宫老师，从选题到开题报告再到初稿和定稿，每一步都离不开宫老师的淳淳教诲，给予的宝贵建议。得遇良师，何其有幸。

“幸得诸君同舟渡，共揽星辉照坦途”。感谢这四年学习路上遇到的志同道合的朋友们，特别是 rCore 社区的朋友与老师们，是他们在我完成此论文的过程中给予了我许多帮助。

“孤灯砺剑千重浪，素履独行万里霜”。最后我要感谢这四年来的自己，每一次整理心情后的前行，每一次重整旗鼓的奋进，都成就了今天的我，也激励我在未来继续前行。

韶光易逝，终有别离。感恩所有的相遇，相信这些经历都让我们的花样年华变得独一无二。