北京理工大學

本科生毕业设计（论文）

北京理工大学本科生毕业设计（论文）题目

The Subject of Undergraduate Graduation Project (Thesis) of Beijing Institute of Technology

学　　院：	计算机学院
专　　业：	计算机科学与技术
班　　级：	07112103
学生姓名：	王菁芃
学　　号：	1120211759
指导教师：	王菁芃

2025 年 5 月 14 日

原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名：　　日 期：　　年　月　日

关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（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本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名：　　日 期：　　年　月　日

指导老师签名：　　日 期：　　年　月　日

目 录

第 1 章 绪论

1.1 研究背景及意义

1.1.1 选题背景

当今计算领域对操作系统的性能、安全性和可靠性要求日益严苛，微内核技术因其固有的优势而备受关注。其中，seL4 微内核作为目前最先进的微内核之一，已被广泛应用于安全关键领域。然而，seL4 的同步系统调用和 IPC 会产生大量的特权级切换，且无法充分利用多核的性能。虽然微内核对异步通知有一定的支持，但仍需要内核进行转发，其中的特权级切换开销在某些平台和场景下将造成不可忽视的开销。

为了进一步提高系统性能和效率, ReL4 项目使用 rust 语言重写的 seL4 微内核，基于用户态中断技术改造 seL4 的通知机制，设计了无需陷入内核的异步系统调用和异步 IPC 框架，在提升用户态并发度的同时，减少特权级的切换次数。而 ReL4 项目中，异步 IPC 的引　造成了额外的运　时开销，导致异步 IPC 在低并发的场景下性能显著低于同步 IPC。

1.1.2 研究目的

本研究旨在针对 ReL4 项目中异步进程间通信的性能瓶颈问题，提出并实施了一种创新性的解决方案。该方案通过利用硬件资源，部分替代传统的异步调度器功能，对异步运行时进行硬件层面的加速，以期显著提升异步 IPC 的性能表现。

1.1.3 研究意义

本研究旨在提高了 ReL4 项目中异步进程间通信的性能，此研究成果可直接迁移至实际操作系统开发，以增强系统运行效率，降低资源消耗，进而提升用户交互体验。

在安全性及可靠性增强方面，本研究通过降低特权级切换频率，不仅优化了操作系统性能，亦提升了系统的安全性和可靠性。此优化对于安全关键领域，如航空航天、军事、医疗等，具有尤为重要的意义，有助于确保这些领域信息系统的安全稳定。

在技术应用与产业推动层面，本研究为微内核操作系统开发者提供了切实可行

的技术路线，推动了微内核技术在多核处理器环境下的广泛应用。同时，本研究对于硬件加速技术的应用具有典范作用，有助于促进相关产业的技术进步和创新。

此外，本研究促进了操作系统与硬件设计、并发编程等领域的深度融合，为跨学科研究提供了新的研究路径和方法论。这对于促进计算机科学与其他工程学科的技术融合，具有重要的学术价值和实践意义。

1.2 国内外研究现状

目前，国内在硬件调度器领域已经有了一些进展。如关沫张晓宇采用 VHDL 语言设计了适用于硬件化的实时操作系统调度器, 基于 FPGA 使用组合电路和时序电路完成了系统内核调度器的搭建。国外学者 Y.Klimiankou 提出了一种提出了 Micro-CLK，一种基于微内核的多服务器操作系统设计，其核心思想是将进程间通信从内核中移除，从而提高效率并简化内核设计。进程间通信的优化成为微内核性能优化的重点。目前的研究工作虽然在不同领域取得了进展，但没有将硬件调度器与异步的进程间通信与异步的系统调用结合的实例。

1.3 研究内容和研究方法

本研究将围绕以下几个核心环节展开：首先，本研究将对 Rel4 内核中的用户态异步运行时机制以及 TAIC 硬件调度器的设计理念与具体实现进行详尽剖析，将深入挖掘其工作原理、性能特点以及接口设计，为后续的优化与改进提供理论基础。其次，将着手进行硬件适配的异步运行时的体系结构设计与编码，主要包含以下工作：a. 将内核中异步运行时的队列改为 taic 实现，以适应异步系统调用。b. 将异步系统调用的库函数实现由系统调用改为读写 taic。c. 将异步系统调用回复时的用户态中断改为读写 taic，唤醒用户态对应的协程。接下来，将在 Qemu 模拟器和 FPGA 硬件环境两种不同的平台上进行仿真与测试，以确保我们的异步运行时能够在多种场景下稳定运行。主要测试内容包括：a. 对改进前后的异步 ipc 和异步系统调用在不同并发度下进行测试。b. 对改进前后的异步 ipc 和异步系统调用低并发下不同环节的时间消耗进行测试。最后，我将对仿真与测试的结果进行全面评估与分析，针对发现的问题进行调优。通过这一系列的优化措施，旨在提升异步系统调用的性能，使其更好地适应各种应用场景。

1.4 论文结构安排

1.5 本章小结

第 2 章 系统架构与设计

2.1 硬件化改造需求分析

在 Rel4 系统原有的实现中，异步系统调用的过程会经历特权级的切换和用户态中断的触发。具体过程大致如下：

异步系统调用的发起需要用户态的异步运行时以及异步系统调用库函数的支持。发起异步系统调用的协程是一个由异步运行时所调度的异步协程。在该协程提交异步系统调用后，库函数会将请求写入缓冲区，然后根据缓冲区中的原子变量判断内核中的处理协程是否正在执行。若内核中的处理协程已经空闲阻塞，则进行系统调用来唤醒内核中的处理协程。此时，CPU 会陷入内核对该内核任务进行调度，并尝试寻找一个空闲的内核发送核间中断以执行该处理任务。综上，异步系统调用的发起引入了异步任务调度所需的必要开销以及两次特权级切换所带来的上下文保存恢复开销。

在处理协程完成异步系统调用处理后，内核会通过通知机制对调用方进程发送用户态中断，以唤醒用户态进程的中断处理函数。中断处理函数再调用接受回复的异步协程, 对内核的回复进行分发。接受回复的异步协程会根据回复消息中的协程 id，唤醒对应的因等待回复而阻塞的异步系统调用发起协程。此过程包含了用户态中断所带来的上下文保存恢复、中断处理函数中的协程唤醒，以及接受回复的异步协程进行的分发处理。

在整个异步系统调用的实现过程中，缓冲区的写入，读取，系统调用的实际处理是必要的。而此外，由于异步的引入，额外带来了（可能的）两次上下文切换开销，异步任务的调度开销，中断处理函数执行开销，以及回复分发的执行开销。

根据原 Rel4 异步系统调用的性能测试结果，在低并发场景下，异步系统调用的性能较低，且远高于同步。根据上文的过程分析可知，低并发场景下，内核中的处理协程有较大的概率处于空闲状态。这就导致 CPU 需要频繁的陷入内核以唤醒内核中的处理协程。同理，用户态中断的触发也会相较高并发场景更加频繁。由此可见，频繁的系统调用和用户态中断是导致低并发场景下异步系统调用性能较低的主要原因。异步的引入旨在提高系统整体吞吐率，而低并发下高额的开销无法被较多的并发协程均摊。综上所述，Rel4 中原有的异步系统调用需要性能优化，以减小异步功能的

引入所带来的额外开销。

2.2 异步系统调用的新架构

为了减小上述过程所产生的不必要开销，我设计将 Taic 硬件调度器集成到 Rel4 操作系统中，用于代替异步运行时中原有依赖软件实现的调度逻辑。进而优化异步系统调用路径，改善整体的性能。

2.2.1 异步运行时改造

由 xx 结分析可以得知，Taic 内部提供了多个硬件级的任务队列，具备跨特权级别的任务调度能力，即允许内核直接操作用户态调度上下文，而无需进行中断和陷入。基于这一特性，我首先对异步运行时进行了结构上的调整，将原本基于软件实现的协程阻塞与唤醒机制，替换为基于 Taic 队列的硬件实现。

2.2.2 异步系统调用路径改造

由于硬件调度器的引入，异步系统调用的提交不再需要通过原有的系统调用，也就无需陷入内核态进行相关处理。因此，我将异步调用的提交路径完全保留在用户态。利用 taic 队列的信号收发机制，直接在用户态将任务发送至内核控制流，大幅简化了路径、提升了执行效率。

同样的，内核处理完成后的回复消息也不再需要用户态中断进行间接唤醒。理想情况下，每个因等待内核回复而阻塞的异步发送协程均可被硬件直接唤醒。因此可以舍弃原有的分发协程与中断处理函数。但由于硬件唤醒的通道数量存在限制 (xx 小结中详述)，本设计保留了分发协程用于在高并发场景下，二次唤醒未成功申请到硬件通道的协程。

2.2.3 系统调用完整流程

根据上文的架构设计，改进后的异步系统调用流程如下：

1. 在内核初始化时，会对异步运行时进行初始化。该过程中会调用 taic 相关接口分配资源，申请一个内核专用的队列。

2. 在创建用户态进程时，异步运行时会申请 Taic 队列，创建缓冲区，并完成的异步系统调用的注册。此时内核会创建一个单独的处理该用户态进程异步系统调用的协程，并将该协程加入阻塞队列。

3. 当用户态进程需要发起异步系统调用时，会创建一个异步协程。该协程会构造请求消息并将消息写入缓冲区。随后根据内核中的执行情况，调用硬件能力唤醒内核协程。一切准备就绪会，该协程会阻塞自身，等待回复

4. 内核执行处理异步系统调用的协程。该协程会从缓冲区中读取消息，进行处理并将结果写回缓冲区。随后，根据协程的 id 以及分发协程的执行情况，唤醒对应的协程。

5. 用户态阻塞等待回复的协程因唤醒而被调度执行。该协程会从缓冲区收到回复，至此一次完整的异步系统调用完成。

2.3 异步运行时的硬件适配

2.3.1 系统架构设计

为了将 Taic 引入到异步运行时中，我设计采用分层架构，如图 X 所示，系统结构从上至下依次为：内核/用户态程序、异步运行时、Taic 接口层、Taic 驱动层以及最底层的 Taic 硬件。

内核态与用户态中的异步程序位于最上层，可直接通过异步语义发起异步系统调用。异步协程的阻塞、唤醒与调度由下层的异步运行时负责管理，从而实现了异步过程的自动化控制。

为了更好的在异步运行时中调用硬件调度器的能力，我设计增加了一层 Taic 接口层作为运行时与底层驱动之间的抽象桥梁。该层提供统一接口和地址映射功能，在 Taic 原有驱动的基础上进行了进一步的封装，实现了支持跨越特权级的任务控制操作。

2.3.2 相关接口调整

在原有的初始化过程中，加入硬件能力的初始化。替换异步运行时的就绪队列为一个硬件结构抽象出的队列对象。

2.4 硬件资源分配与映射策略

2.4.1 中断向量映射策略

在 3.2 章节中所叙述的优化框架中，协程需要被一一对应的唤醒。由 2.2.3 中所阐述的 Taic 模型可知，协程的唤醒需要通过队列中提前分配的“中断向量”。Taic 硬件调度器仅提供固定数量的中断向量（共 32 个），因此，如何在有限的中断资源下高效地映射和调度大量用户态协程，成为本设计中的核心问题之一。为尽可能高效的利用已有的硬件资源并提高低并发下的异步性能。本设计采用动态映射策略分配中断向量。

该设计在异步运行时中保留一个 0 号协程（dispatcher 协程）。该协程常驻于异步运行时中，用于在硬件资源不足时间接唤醒协程，以实现硬件资源的复用。

在客户端执行异步系统调用请求的过程中，请求协程首先尝试向申请空闲向量，用于在内核完成操作后反向唤醒自身。

- 若当前中断向量表尚未满载，即存在可用中断向量，则会成功分配到一个向量号。协程会将该中断向量写入调用传递的消息中，连同其他上下文信息发送给内核。当内核处理完毕后，即可通过该中断向量向对原始协程的直接唤醒，无需额外中间调度，极大提升响应效率。

- 若中断向量表已满，即所有中断向量已被活跃协程占用，当前协程无法获得中断资源。此时，协程会将消息结构中的中断向量字段显式置 0，表明该请求不绑定硬件中断唤醒。内核在处理完该请求后，会将该请求的回复消息 id 加入缓冲区中的控制队列（3.5 章节中有叙述），并尝试唤醒 dispatcher 协程。dispatcher 协程在被唤醒后，解析消息中的相关字段并唤醒对应的用户协程，完成该异步系统调用的回复流程。

中断向量的管理由硬件接口层负责。在系统初始化阶段，硬件接口层在内存中构建并初始化中断向量表结构。该向量表采用位图方式存储中断向量的分配状态，其中 0 号中断向量被保留用于唤醒 0 号协程。同时，接口层向上层提供向量申请与释放接口，从而实现硬件资源的动态分配。

该映射机制有效地实现了对硬件接口的封装与抽象。整个过程由异步运行时与硬件接口层协同完成，上层应用代码无需感知底层实现细节。

2.5 异步通信缓冲区结构优化

原有的缓冲区采用环形队列实现消息的存储与管理。为了更好的适配新的处理流程并提高异步性能，我对缓冲区的结构设计进行了重新设计与优化。

由于在硬件资源充足的情况下取消了 dispatcher 协程的分发过程，原阻塞协程直接由内核唤醒。这一变更虽然减少了调度开销，但也带来了新的挑战：由于消息处理的顺序不再严格遵循写入顺序，原有的环形队列已不再适应这种处理模式。因此，我将缓冲区的结构由环形队列改为基于“消息槽”的形式，并设计了一套槽位分配机制，尽可能的降低写入和读取所造成的开销。

新的缓冲区由数据区、索引队列和状态变量三部分组成。

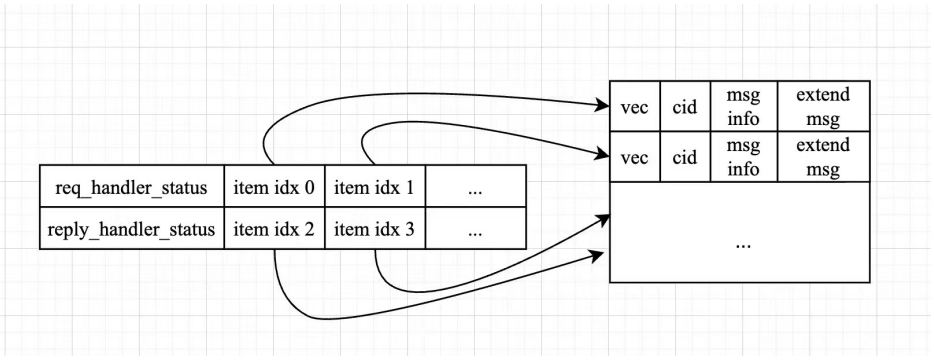

图 2-1 缓冲区结构示意图

数据区是一个预分配的内存块，用于存储异步系统调用过程中传递的消息。整个区域被划分为多个“消息槽”，每个槽用于存储一条完整的消息内容。

索引队列包括数据区的控制信息。索引队列中存储着需要内核协程处理的请求消息 id 和需要分发协程分发的消息的 id。协程可以从队列中依次取出需要处理的消息 id，然后根据该 ID 从数据区中读取相应的消息内容，进行处理或分发。

状态变量由两个原子标志组成，分别表示内核中的处理协程和用户态的回复分发协程是否阻塞。

当协程发起异步系统调用时，首先会向缓冲区申请一个槽位 id，然后将消息写入数据区对应的位置并将该消息的 id 入队。内核处理协程在执行过程中，会从队列中取出待处理的消息 id，读取对应槽中的数据进行处理，并将处理结果写回同一槽位。

当硬件资源充足时，回复消息不需要经过分发协程分发，因此不需要将回复的消息 id 入队。若并发数超过 31，部分协程不能直接通过硬件唤醒，则需要将回复的消息 id 写入队列。并尝试唤醒分发协程。分发协程同样会依次取出队列中的消息 id，然后根据消息的 id 读取数据区的消息，进而再唤醒对应的协程。

2.6 异步事件注册机制的应用

由 3.5 节叙述可知，协程的中断向量采用了动态分配的策略，中断向量的注册是在运行时进行的。而中断向量的注册机制会对异步系统的实时性造成影响。如第 2.2.x 节所述，Taic 提供了两种中断注册方式：“一次性注册”与“持续注册”。

默认的注册机制采用“一次性注册”策略：每当协程因中断事件被唤然而，在高并发或高频唤醒中，频繁注册与注销事件会带来非忽略的开销，影响整体调度性能。

为优化上述问题，Taic 提供了一种基于标志位控制的“持续注册”机制。在该模式下，开发者可通过设置注册参数中的特定标志位，实现中断向量的保留与复用，避免不必要的重复注册。此机制在高并发或重复事件场景下具有显著优势，能有效减少调度延迟与资源竞争。但在中断频度较低或并发度不高的系统中，可能会限制灵活性。

基于上述两种机制，本设计均进行了完整实现，并构建了相应的测试用例用于功能验证与性能评估。具体的实验配置、对比分析与测试结果将在第 5.3 节中详述。

2.7 本章小结

本章围绕 Rel4 异步系统调用的性能瓶颈，分析了其在低并发场景下性能下降的主要原因，指出频繁的特权级切换与用户态中断是系统开销的重要来源。针对这一问题，设计并引入了 Taic 硬件调度器，替代原有软件调度逻辑，从而重构了异步系统调用路径。本章详细描述了异步运行时与系统调用流程的改造方法，提出了支持硬件调度的异步运行时架构。随后，介绍了中断向量的动态分配策略及其在硬件资源受限情况下的调度优化机制，并对缓冲区结构进行了适配与重构，以支持新的唤醒模式。最后，分析了异步事件注册机制的两种实现方式，并为后续实验验证做了准备。

第 3 章 实现细节

3.1 硬件地址映射与接口层封装

3.1.1 中断向量分配器实现

为实现 3.3.4 章节中的中断向量复用，本设计在硬件接口层实现了一个中断向量分配器。

由于硬件中存储了协程 id 与中断向量号的对应关系，因此软件接口只管理每个中断向量号的使用情况。本设计中，分配器基于 32 位位图实现。该位图提供基本的方法，如设置位、清除位和查询首个零位。

其中为提高性能，查询首个零位使用 Rust 内建方法 `trailing_zeros`。该方法返回最低有效零位的索引，在 risc-v 中可以被映射成 CLZ 等硬件指令。具体算法如下：

算法 1 查找位图中首个零位

function FIND_FIRST_UNSET(bitmap: 32-bit integer)
  $inverted \leftarrow\sim bitmap$
  if $inverted = 0$ **then**
    return None
  else
    return trailing_zeros(*inverted*)
  end if
end function

分配器基于上述的位图结构进行封装，提供了初始化、分配、释放接口，结构示意图如下所示：

3.1.2 硬件队列封装

由于内核和用户态的地址空间存在差异，因此分别在用户态和内核态的接口层对 `taic_driver` 进行了不同的封装。

内核中接口层首先定义了 TAIC 的寄存器物理地址基址、本地队列数量、内核线程 id 等常量。然后，使用 `lazy_static` 宏声明了一个内核中全局唯一的本地队列 `LOCAL_QUEUE`。该队列使用常量参数调用 `taic_driver` 中 `alloc_lq` 的方法分配。

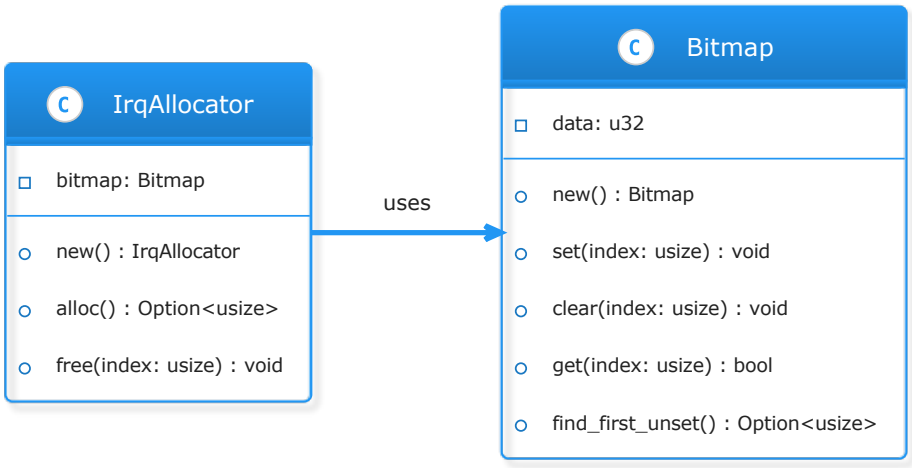

图 3-1 中断向量分配器类图

接口层提供 `get_lq` 方法返回 `LOCAL_QUEUE` 的 Arc 智能指针，用于异步运行时就绪队列的初始化。

接口层封装了发送者和接收者的注册逻辑。

其中 `register_sender` 用于将内核的队列的注册成为另一个全局队列的消息发送方。该方法直接调用 `LOCAL_QUEUE` 中的 `register_sender` 方法，并固定 `os_id` 参数为常量。

其中 `register_receiver` 用于将内核的队列的注册成为另一个全局队列的消息接受方。该方法将设置了两个布尔变量：preempt 与 reusable 分别表示该次注册是否抢占之前的注册以及是否可循环使用。接收方 ID、可抢占性与可复用性被按位编码为一个 `handler` 值，并传递给 `LOCAL_QUEUE` 的 `register_receiver` 方法完成注册。

接口层还提供 `send_signal` 方法，用于向其他队列发送信号，通过中断向量唤醒对方队列中的协程。

用户态的接口层中，使用 `#`

$$thread\_local$$

宏定义了线程局部变量 `LQ_MANAGER`。该变量作为硬件资源的全局管理实例，整合了中断向量分配器以及全局队列。用户态的接口中使用 `LQ_MANAGER` 中的队列代替全局队列，实现了硬件资源的线程隔离。

3.2 缓冲区优化

3.2.1 IPC 消息字段修改

为了在通信时附带协程所申请的中断向量信息，本设计在 `IPC_Item` 中新增了 vec 字段用于指示发送消息的协程的中断向量。若该字段为 0，则表明该协程无中断向量，需通过 `dispatcher` 协程唤醒。

vec	cid	msg_info	extend_msg
u32	u32	u32	[u16; MAX_IPC_MSG_LEN]

图 3-2 ipcitem 结构

3.2.2 环形队列结构优化

依据2.5中的设计，缓冲区实现如图3-3所示。

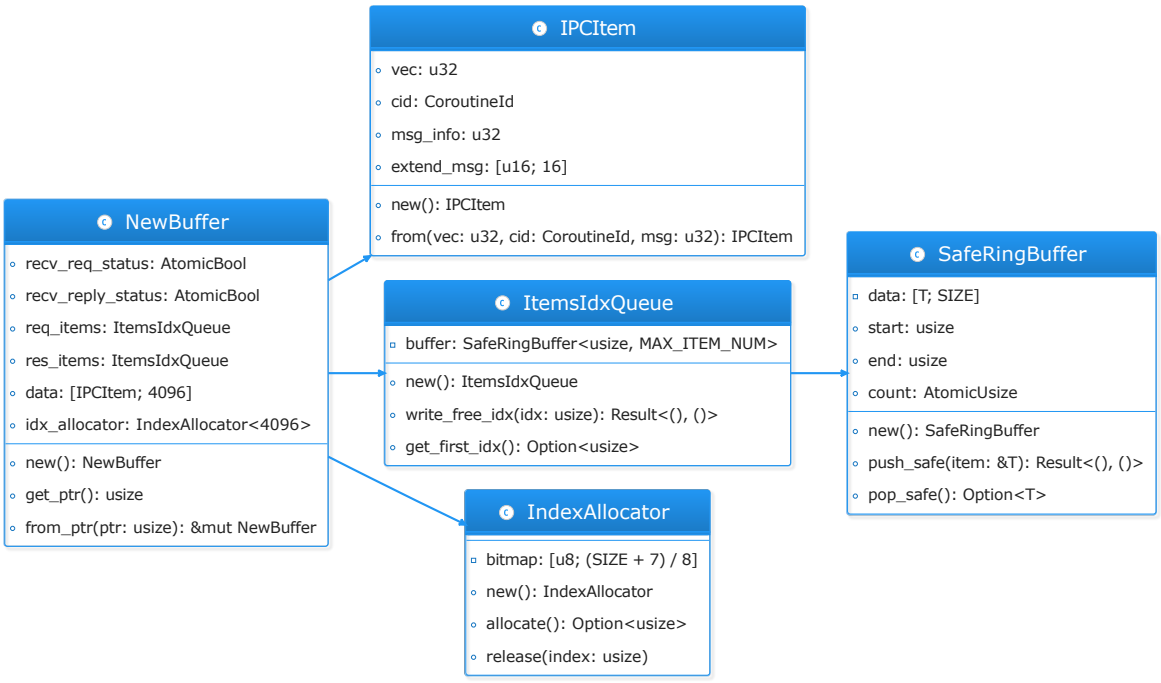

图 3-3 缓冲区类图

缓冲区中 `data` 为数据区，由长度为 4096 的 `ipc_item` 数组实现，用于存储调

用时的传递消息。

req_items 和 res_items 分别为请求消息与回复消息的索引队列，两者均采用 ItemsIdxQueue 数据结构实现。ItemsIdxQueue 基于 SafeRingBuffer 封装，实现了一个线程安全的环形队列，提供了入队 write_free_idx 和出队 get_first_idx 两个基础方法。SafeRingBuffer 是一个无锁环形缓冲区，其内部使用原子变量 count 记录缓冲区中有效元素的数量，以此保证安全的读写。

recv_req_status 和 recv_reply_status 为两个布尔类型的原子变量，分别表示接受请求的协程的当前状态与接受回复的 dispatcher 协程的当前状态。这两个状态标志减少了唤醒的次数，避免了重复调度。

idx_allocator 为数据区存储空间的索引分配器，负责管理数据区索引号的分配与回收，其内部同样使用位图实现。

缓冲区索引号的分配仅由用户态线程发起，而内核中的处理协程始终使用相同的索引号进行回复。因此，缓冲区的索引号总是由同一线程进行分配，具有天然的线程安全性。同时，两个环形队列的通过原子变量实现了并发访问控制，整个缓冲区在多线程环境下可实现无锁并发访问。该设计降低了因分配器引入导致的性能开销，使分配回收操作在极短的时间内即可完成，确保了高并发场景下的稳定性与效率。

3.3 执行器硬件适配

为了使异步运行时能够调用硬件能力，本设计对内核中异步运行时的执行器 Executor 中部分数据结构与方法进行了重写。

因为缓冲区的逻辑优化,已不再需要进行消息转发,因此本设计删除了 immediate_value 变量，并将原有的就绪队列改为 taic 中的本地队列的 Arc 引用 Arc<LocalQueue>。在异步运行时初始化时，会调用下层的接口层代码，申请一个本地队列，并将其引用传给执行器完成就绪队列的初始化。

spawn 为执行器的协程生成方法。在该方法中,Executor 使用 taic 提供的 task_enqueue 方法将生成的协程加入就绪队列。因为 taic 句柄的第一位和第二位用作重复注册和抢占的标志位，因此调用该方法时需要协程 id 左移 2 位。

fetch 为执行器从就绪队列中调度协程执行的方法。在该方法中使用 taic 提供的 task_dequeue 方法取出就绪队列中的就绪协程。

3.4 异步系统调用注册机制

为实现上述修改，本设计修改了原有的异步系统调用注册相关代码，删除了用户态中断的部分逻辑，并修改了协程生成的相关逻辑。

3.5 客户端发起

3.5.1 进程初始化

为使用户态进程支持异步系统调用能力，系统需要在创建进程时准备相关数据结构，完成异步系统调用的初始化。

系统首先需要对异步运行时进行初始化，然后对初始化 taic 的硬件能力。再此过程中，会将 taic 的读写寄存器映射到用户态地址空间。然后对该进程分配接受内核回复通知的能力。基于该通知，利用 taic 的 `alloc_reveiver` 函数为进程分配一个队列。接着会为进程初始化一个异步通信的缓冲区，将该缓冲区注册用于异步系统调用，并访问缓冲区的能力。然后，基于通知能力，进行系统调用，进行 3.2 所述的异步系统调用注册。在注册成功后，利用接口层的 `register_sender` 函数，将当前进程的 taic 队列注册成为系统内核全局队列信号的发送者。最后，调用异步运行时的接口，生产用于中断向量不足时间接唤醒的分发协程。

该部分流程图如下：

3.5.2 异步系统调用发送协程

使用该系统的用户可利用异步运行时，生成协程发起异步系统调用。目前给出可能的协程内进行异步系统调用的方法。

协程首先应调用结构层的 `alloc_vec` 函数尝试申请分配一个 vec。若分配成功，则利用该 vec，调用 `register_receiver` 注册函数将本协程注册成消息的接受者。然后即可通过异步系统调用接口进行异步系统调用。在调用完成后，应释放分配的 vec。

3.5.3 异步系统调用接口

为方便用户使用异步系统调用，异步库提供异步系统调用接口，异步系统调用进行封装。该类接口中会对参数进行编码转换，构造成异步系统调用消息。最后调用异步 call 函数进行异步调用。

3.5.4 异步 call 函数

`seL4_call_with_item` 为发起异步系统调用的关键函数。该函数接受三个参数：接收方 id, 中断向量以及进程间通信条目，可对目标发起异步的 `sel4_call` 调用。在接收方为 0 号的异步系统调用中，会调用通知对象的 send 能力，对系统中处理异步系统调用的端点发送消息，并等待 reply 回复。

该方法首先获取接受方的异步通信缓冲区，然后向缓冲区申请 id。成功申请到 id 后，依据传入的消息参数将消息写入缓冲区。此时，会检测冲区中的原子变量 `recv_req_status`。若原子变量为 `false`，则说明内核中的处理协程已经因空闲阻塞。此时会先调整原子变量的值，防止并发时重复唤醒。然后调用接口层中的 `send_signal` 方法像内核发送信号，唤醒对应协程。完成操作后，该方法会在此阻塞，等待内核的回复。当协程因收到回复而被再次调度执行时，会依据先前的缓冲区 id 获取回复消息并进行返回。

该方法的伪代码如下所示：

算法 2 `sel4_call_with_item(recv, vec, item)`

```
buffer ← try get mutable buffer from SENDER_MAP[*recv]
if buffer is None then
    return Err("Failed to get service buffer")                ▷ 获取失败
end if
idx ← buffer.idx_allocator.allocate()
if idx is None then
    return Err("Failed to allocate index in buffer")          ▷ 索引分配失败
end if
buffer.data[idx] ← *item
if buffer.req_items.write_free_idx(idx) is Err then
    return Err("Failed to write free index")                  ▷ 写入请求队列失败
end if
if buffer.recv_req_status.load() == false then
    buffer.recv_req_status.store(true)
    send_signal(*recv, vec)                                   ▷ 唤醒接收方
    TEST_TAIC_SEND_SIGNAL += 1
end if
yield_now().await                                             ▷ 阻塞等待响应
buffer.idx_allocator.release(idx)                             ▷ 释放索引
return Ok(buffer.data[idx])
```

3.5.5 分发协程

当中断向量不足，需要分发协程进行唤醒的软件转发。异步库提供该协程的实现：

该协程相当于接收回复消息的服务端。协程首先会从参数中解析出异步系统调用的缓冲区，然后将自身注册成为 0 号中断向量的接收者，然后会进入主循环。在循环内部，该协程不断从缓冲区的“接受回复索引队列”中取出需要唤醒的消息 id。如果成功取到, 则会根据消息中的协程 id，通过 `coroutine_wake` 接口唤醒对应的协程。如果队列为空，则会调整缓冲区中表示自身状态的原子变量 `recv_reply_status`，然后调用 `yield_now` 接口阻塞自身，让出执行权。

3.6 服务端处理

内核中异步系统调用的处理协程相当于内核中的服务端，与分发协程类似。该协程会在异步系统调用注册时生成，并通过参数传递得到缓冲区能力。协程解析出缓冲区后，即进入主循环处理消息。

在主循环中，协程尝试从缓冲区的 `req_items` 索引队列中获取消息索引。如果成功获取到消息，则解析消息中的标签，根据不同的系统调用进行不同的处理。处理完成红，会根据消息索引将处理的结果写回缓冲区。如果请求消息中的中断向量为 0，则直接通过硬件接口 `send_signal` 发送信号唤醒原协程。否则，会先将回复消息的索引写入队列 `res_items`，然后根据缓冲区中的原子变量 `recv_reply_status` 判断分发协程的状态。若分发协程已经阻塞，则通过 `send_signal` 接口唤醒分发协程。

当缓冲区中所有消息都被处理完毕后，协程会调用 `yield_now` 接口阻塞自身，等待唤醒。

具体处理流程图如下：

3.7 本章小结

本章详细介绍了支持异步系统调用的系统关键实现细节。首先，通过中断向量分配器的设计，实现了协程与硬件中断之间的高效映射，使用基于位图的分配结构以及硬件友好的算法，提升了分配性能。随后，在接口层对内核与用户态的硬件队列进行了合理封装，分别实现了局部队列的独立管理与中断驱动的异步唤醒机制。

在缓冲区部分，针对高并发场景进行了深度优化。通过双环形队列配合原子状态变量，确保了消息队列的无锁并发访问能力，同时减少了不必要的线程唤醒，有效提升了运行效率。缓冲区索引的线程安全性也通过线程唯一分配策略得以保证。

针对执行器模块，本设计完成了其与硬件异步能力的适配，替换了原有就绪队列逻辑，并引入基于 taic 队列的协程调度机制，形成软硬协同的调度执行路径。此外，为适配上述机制，对异步系统调用的注册、发起、接收、回复及中断转发等环节进行了系统性重构。

最后，完整实现了服务端处理协程与客户端分发逻辑，为异步系统调用提供了完整的数据路径和调度保障，确保异步系统调用在用户态与内核态间的高效传输与响应。通过本章的设计与实现，整个系统的异步通信能力得以构建，为后续章节的性能评估与应用部署奠定基础。

第 4 章 实验与评估

基于上述实现，本设计完成了对 rel4 系统中，基于 taic 的异步系统调用优化。为验证系统的功能正确性与性能优化情况，本设计编写了测试用例并 FPGA 上进行实验验证。本章将对实验的情况进行讲解，并对实验结果进行分析。

4.1 测试环境与平台

本实验采用的宿主机环境如下表所示：

CPU	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
操作系统	Windows 11 专业版，操作系统版本 26100.3915
系统架构	x86_64
内存大小	32 GB

表 4-1 宿主机环境配置

本实验采用的虚拟机环境如下表所示:

虚拟机软件	VMware® Workstation 17 Pro
虚拟机版本	17.5.0 build-22583795
虚拟机操作系统	Ubuntu 24.04.1 LTS
虚拟机内核版本	Linux 6.11.0-25-generic
虚拟机核心数与内存大小	4 核心，8GB

表 4-2 虚拟机环境配置

本实验在 FPGA 上对异步系统调用性能进行评估验证，开发板部分信息如下：

4.2 功能测试

4.2.1 taic 调用测试

本测试用于测试将 taic 适配进系统并允许用户线程访问后，是否能正常使用全局队列的各项功能。测试模拟用户线程进行任务的入队出队操作。测试中，线程使用硬件调度器分配一个全局队列，然后将 100 个任务入队，再将 100 个任务出队。

开发板型号	AXU15EG
FPGA 型号	Zynq UltraScale+ XCZU15EG-2FFVB1156 MPSoC
FPGA 开发板 RISC-V 子系统参数表	
核心数	2
频率	100 MHz
BTB 入口数量	40
L2 缓存大小	2 MB
内存大小	2 GB

表 4-3 FPGA 开发板基本信息

测试结果如下图所示

由结果可知，测试成功将 100 个队列入队并出队，测试结果与预期相符，说明用户线程使用硬件能力正常。

4.2.2 缓冲区读写测试

本测试用于测试共享缓冲区的读写功能是否正常。测试模拟用户协程申请缓冲区索引并写入数据，内核服务协程从缓冲区读取数据并进行回复。测试时，用户协程向缓冲区写入字符串”message from usermode”，内核协程拿到数据后回复”message from kernel”。

测试结果如下图所示

有结果可知，用户协程与内核协程成功利用缓冲区进行通信，索引分配回收正常。

4.2.3 异步系统调用测试

本测试分别测试进行硬件优化后，Rel4 系统中原有的各种异步系统调用功能是否正常。分别为输出类系统调用、通知类系统调用和内存映射类系统调用。

4.2.4 输出类 async_sycall 功能性测试

输出类系统调用测试用例如下：

1. 调用 `PutChar` 异步系统调用输出一个字符 ‘’X’‘。预期结果：控制台输出字符 ‘’X’‘，并在日志中记录系统调用被调度。

2. 调用 `PutString` 异步系统调用输出字符串 ‘”11111”‘。预期结果：控制台输出字符串 ‘”11111”‘，日志显示异步系统调用调度信息。

3. 调用 RISC-VGetPageAddress 异步系统调用获取目标虚拟地址 vaddr 所在页帧的物理地址，并通过日志输出。

   预期结果：日志记录异步系统调用触发，并显示页帧的物理地址，物理地址应与同步调用结果一致

测试结果如下图所示：

4.2.4.1 通知类系统调用

4.2.4.2 内存映射类系统调用

4.3 性能测试

4.3.1 测试对象选取

本实验中选取内存映射（RISC-VPageMap）与取消映射（RISC-VPageUnmap）两个系统调用作为测试对象并设计实验场景。在基于 RISC-V 架构的 rel4 微内核操作系统中，内存空间的管理权下放至用户态程序，用户负责页表的维护与内存区域的动态分配与释放。因此，用户态程序在运行过程中必须频繁地调用 RISC-VPageMap 和 RISC-VPageUnmap，以实现对虚拟地址空间的有效控制。因此，将二者作为本实验的测试对象，可以用于评估异步系统调用的性能。

4.3.2 性能测试流程

本实验中，测试会模拟线程多次映射物理页帧与解除物理页帧的情况。映射与解除映射时分别调用异步系统调用 RISC-VPageMap 与 RISC-VPageUnmap。测试程序首先会初始化一个线程并为其赋予内存分配能力。随后，线程生成一定数量的物理页框供测试时映射。

为表征系统在不同并发度下的性能表现，实验设置了 1、2、4、8、16、32、64、128 及 256 多组并发度。对于每个并发度，测试程序会生成对应数量的协程。每个协程对同一页框执行 10 轮调用操作，每轮包括一次页帧映射与解除映射。

测试分别基于现有实现和 Rel4 原有的实现进行，以此对比引入 taic 硬件调度器前后的性能差异。此外，为进行完整的性能评估，实验使用同步系统调用作为对照。同步测试中，使用单线程循环进行系统调用，对页框进行映射与解除映射。系统调用总次数与异步调用相同。

性能评估采用以下两个指标：

1. 平均耗时

在异步测试中，该指标为从所有协程启动运行时，到所有协程均完成系统调用后所有的总时间与系统调用次数的比值。其反应异步过程中，等效的单次用时。

在同步测试中，该指标为第一次系统调用开始，到最后一次系统调用结束所有的总时间与系统调用次数的比值。

2. 陷入频率

在测试 Rel4 原有实现的性能时，该指标为平均每次异步系统调用中，需要陷入内核唤醒内核中协程的次数。

在测试引入硬件调度能力后，该指标为使用硬件唤醒内核中协程的次数。

测试整体的流程如下图：

4.3.3 实验结果与评估

该性能测试结果如下

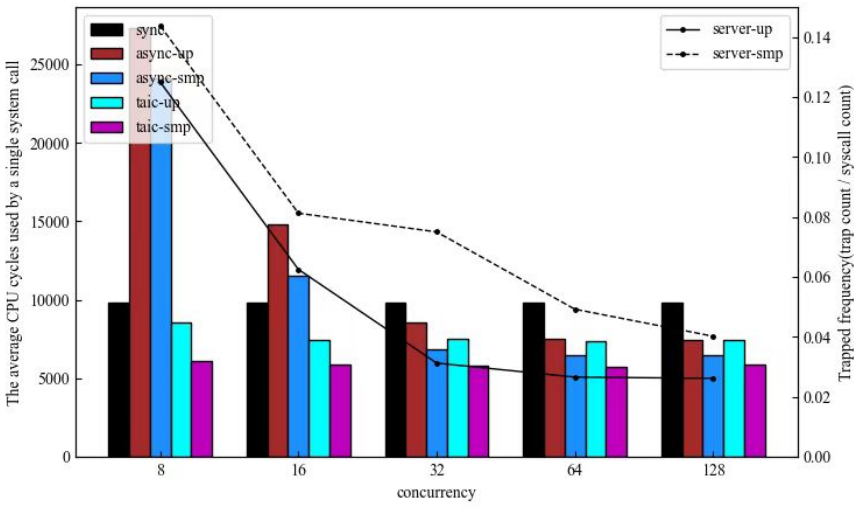

图 4-1 异步系统调用性能测试结果

4.3.3.1 平均用时分析

从结果中可以看出，在所有并发度下，使用 taic 优化后的异步系统调用平均用时均小于优化前原 Rel4 系统中的实现。这是由于 taic 降低了异步运行时引入所造成的开销。根据 3.x 章节中的分析，taic 的使用取代了用户态中断与内核态陷入，减少

了大部分原有实现中造成性能瓶颈的上下文切换。测试结果体现了优化后异步系统调用的性高效性与可用性。

由柱状图中可以看出，在并发度为 8 以下时，原异步实现平均用时显著高于同步，尤其在并发度为 1 时，平均用时大约为 75000 个时钟周期，约为同步调用的 3 倍。而优化后的异步系统调用耗时仅 15000 个时钟周期，有大幅度的降低。这是由于在低并发下，原有实现中内核中的异步系统调用处理协程与用户线程中的分发协程因为没有足够数量的处理请求，大多处于空闲阻塞状态，这导致平均每次调用所需的唤醒频次较高。而 taic 硬件的使用针对这一问题进行了优化，因此在低并发下的性能有较大的提高。

随之并发度的不断增加，所有方案的平均用时均有下降趋势，但使用 taic 优化后，平均用时仍具有最低的平均用时。这是由于并发数量的增加均分了内核处理时用户协程的等待时间。但并发度 32 以上，使用 taic 优化后的平均用时变化较小。这是由于并发的提高通过减少唤醒频次，减小了上下文切换的开销。引入 taic 后，取消了上下文切换的过程，因此优化后的平均用时随并发的增加变化较小。

在并发度为 8 以前，优化后异步系统调用的性能仍高于同步，这是由于异步功能的引入仍具有一定的开销，低并发下缺少足够数量的并发请求来均摊异步功能所造成的额外开销。

4.3.3.2 陷入频率分析

图中折线图部分展示了系统调用的陷入频率。

经过对实验结果的分析可知，不论在单核还是多核的测试环境下，采用优化后的异步系统调用模型的陷入频率均呈现出随着并发度提升而持续下降的趋势。如图中所示，在单核环境下，异步系统调用的陷入频率从初始并发度 8 时的约 0.06 开始，逐步下降至并发度 128 时的约 0.02，下降幅度超过 66%；而在多核环境下，该频率最初达到约 0.14，在并发度达到 64 后开始趋于稳定，最终在并发度 128 时下降至约 0.04，整体下降幅度达到 70% 以上。

造成这一现象的主要原因是，随着并发度的增加，用户态线程发起系统调用的整体速率提高，导致内核中协程的处理量增加。内核中的处理协程进入较长时间的持续运行状态，使得大多数系统调用请求可以被协程直接响应，无需阻塞或等待唤醒，从而显著减少了主动唤醒的次数。

4.4 两种中断注册机制的对比

为验证 taic 两种注册机制对异步性能的影响，本实验为两种实现方式进行了对比实验，分别测试在一次性注册与持续性注册下异步系统调用的平均用时。

实验结果显示，两种中断注册机制在系统调用性能方面存在显著差异，其中持续性注册机制在所有并发条件下均表现出更优的性能。特别是在高并发环境下，持续性注册机制的优势更加明显，其平均系统调用用时显著低于单次注册机制。

随着并发度的提升，单次注册机制的平均调用用时呈现持续上升趋势，而持续性注册机制则保持相对稳定，特别是在并发度为 64 以上时，单次注册机制的平均系统调用延迟迅速上升，远高于持续性注册方式。

造成此现象的原因是，两种机制对 taic 中接受方注册的方式不同。单次注册机制在每次阻塞前均注册一次信号接收。这种实现虽然较为通用，但为了避免内核与用户态线程同时读写硬件寄存器而造成冲突，硬件接口层需对相关函数添加互斥锁进行保护。该互斥锁在低并发下影响较低，但在高并发场景中有较显著的性能影响。随着并发线程数的增加，频繁的争用互斥锁会导致 CPU 大部分时间在无意义的等待中。

而持续性注册机制在异步系统调用注册时就完成了接收方的注册，并保持注册状态。该方法下整个线程运行期间, 内核无需再次进行注册操作，省去了互斥锁操作的时间成本。该机制避免了高并发情况下的冲突问题，使得优化后的异步运行时在高并发下仍有较低开销。

综上所述，在本设计中，持续性中断注册机制具有更好的性能表现，尤其在高并发场景下优势更为明显。因此，本设计最终采用持续性注册机制作为默认实现，以充分发挥硬件调度器在异步系统调用中的优势。

4.5 本章小结