基于软硬协同的中断响应和任务调度研究

(申请清华大学工学硕士学位论文)

培 养 单 位： 计算机科学与技术系

学　　科： 计算机科学与技术

研　究　生： 赵 方 亮

指 导 教 师： 向　　勇　副研究员

二〇二五年三月

Research on interrupt response and task scheduling based on software and hardware collaboration

Thesis submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Computer Science and Technology

by

Zhao Fangliang

Thesis Supervisor : Associate Professor Xiang Yong

March, 2025

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

刘 XX	教授	清华大学
陈 XX	副教授	XXXX 大学
杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所

答辩委员会名单

主席	赵 XX	教授	清华大学
委员	刘 XX	教授	清华大学
	杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所
	黄 XX	教授	XXXX 大学
	周 XX	副教授	XXXX 大学
秘书	吴 XX	助理研究员	清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

在请求完成后，I/O 设备会发出一个中断信号来通知 CPU，这会中断正在运行的任务并影响吞吐量，或者 CPU 主动轮询 I/O 设备的状态寄存器，消耗宝贵的 CPU 周期。在使用超高速 I/O 设备时，中断驱动和轮询机制的有害影响不容忽视。

为了解决这个问题，我们将任务调度器的功能卸载到中断控制器（TAIC）。TAIC 直接维护任务队列并唤醒被阻塞的任务，以实现快速唤醒机制。这种方法保护了 CPU 免受中断的影响，减少了 CPU 在中断响应以及维护调度队列方面的开销，从而降低了 I/O 响应延迟。我们将这一机制与网络卡驱动程序集成，并在 FPGA 平台上进行了验证。它减少了与中断相关的开销并降低了 CPU 利用率。在使用 YCSB 工作负载的 Redis 上的评估表明，TAIC 实现了低尾部延迟，并将吞吐量提升了 6%。

在通用操作系统中，随着并发量的增加，传统的多线程模型由于与内核多线程相关的高上下文切换成本而变得不足。在本文中，我们提出了一种新的并发模型，称为 COPS。COPS 采用基于优先级的协程模型作为基本任务单元，取代了高并发场景中的传统多线程模型，并为内核和用户空间协程提供了一个统一的基于优先级的调度框架。COPS 将协程作为操作系统中的第一类公民引入，以提供异步 I/O 机制，使用内核协程作为 I/O 操作和设备之间的桥梁，以及用户协程来连接应用程序和操作系统服务。

我们基于 COPS 设计了一个原型 Web 服务器，并在基于 FPGA 的系统上进行了广泛的实验，以评估 COPS。结果表明，所提出的模型在保持相对较低的开销的同时，相比于多线程模型，在大型并发应用中实现了高达四倍的吞吐量。

关键词：关键词 1；关键词 2；关键词 3；关键词 4；关键词 5

Abstract

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summary of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Keywords are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 keywords, with semi-colons used in between to separate one another.

Keywords: keyword 1; keyword 2; keyword 3; keyword 4; keyword 5

目 录

摘 要.....I
Abstract.....II
目 录.....III
插图清单.....V
附表清单.....VI
符号和缩略语说明.....VII
第 1 章 引言.....1
1.1 课题研究背景与意义.....1
1.2 国内外研究现状.....2
1.2.1 中断机制.....2
1.2.2 任务调度.....4
1.3 本文的主要研究内容与组织结构.....5
第 2 章 相关技术研究.....7
2.1 任务模型.....7
2.1.1 批处理作业模型.....7
2.1.2 进程模型.....7
2.1.3 线程模型.....8
2.1.4 协程模型.....9
2.2 任务调度.....9
2.3 I/O 模型.....10
2.4 用户态中断.....11
2.5 本章小结.....12
第 3 章 基于软硬协同的中断响应和任务调度设计.....13
3.1 基于执行流的任务模型设计.....14
3.2 基于软硬协同的中断响应和任务调度机制.....16
3.2.1 硬件任务调度器.....16
3.2.2 硬件中断处理机制.....18
3.2.3 硬件支持的任务通信机制.....19
3.3 软硬件交互接口.....20
3.4 本章小结.....21

第 4 章 基于软硬协同的中断响应和任务调度实现 22
4.1 系统整体结构 22
4.2 控制器实现 22
4.2.1 任务调度 23
4.2.2 中断处理 24
4.2.3 任务通信 25
4.2.4 资源分配与回收 26
4.3 任务抢占 27
4.4 软件适配 28
4.4.1 申请硬件资源 28
4.4.2 任务通信 29
4.5 本章小结 31
第 5 章 性能评估 32
5.1 测试环境 32
5.2 微基准测试 33
5.2.1 任务调度 33
5.2.2 任务通信 35
5.3 综合测试 37
5.3.1 特征分析 37
5.4 本章小结 37
第 6 章 总结与展望 38
参考文献 39
致 谢 44
声 明 45
个人简历、在学期间完成的相关学术成果 46
指导教师评语 47
答辩委员会决议书 48

插图清单

图 3.1 基于软硬协同的任务状态模型 ……17
图 3.2 硬件处理时钟中断 ……19
图 4.1 基于软硬协同的中断响应和任务调度系统整体架构 ……22
图 4.2 就绪队列 ……23
图 4.3 外部中断处理 ……25
图 4.4 任务通信 ……26
图 4.5 任务抢占的流程 ……28
图 5.1 任务调度对比，其中 taic 表示使用硬件任务队列，tokio 表示使用软件任务队列。 ……34
图 5.2 IPC 延迟对比，taic 表示使用硬件任务通信机制的测试。其中，bi 表示双向，双方互相发送消息；uni 表示单向；uint 表示任务优先级高，需要发送中断。eventfd、pipe、signal 均为双向； ……35

附表清单

表 3.1　软硬件交互接口 ……20
表 5.1　测试硬件环境 ……32
表 5.2　IPC 性能对比……36
表 5.3　中断延迟对比 ……36

符号和缩略语说明

PI	聚酰亚胺

第 1 章　引言

1.1　课题研究背景与意义

近年来，云应用程序（例如网络搜索、社交网络、电子商务、实时媒体等）在日常生活中发挥着越来越重要的作用。为了保证良好的人机交互体验，这些应用程序需要在几十毫秒的时间尺度上对用户的操作作出响应，而它们通常将单个请求分散到数据中心的数百台计算机上运行的数千个通信服务，其中耗时最长的服务成为了影响端到端响应时间的主要因素，这要求每个参与的服务的尾部延迟在几百微秒的范围内；随着这些亿级用户应用的爆发式增长，数据中心面临的性能挑战已远超人机交互的低延迟需求，数据中心需要每秒钟能够处理数百万请求，现代计算机系统正经历着前所未有的吞吐量压力考验。

嵌入式实时系统应用也呈现出相似的发展趋势，其应用场景逐渐泛化，从传统的工业控制领域（工业化流控、车辆制造等）向消费级场景（智能家居、健康监护、车辆智能座舱等）逐渐渗透，催生出与新型技术融合的需求，模糊了工业与消费电子的传统界限。在消费端应用中，嵌入式实时系统开始集成轻量化人工智能模块，既要求系统保留实时响应能力的同时，还需要实现微型机器学习等新型功能；在车载智能座舱领域，嵌入式实时系统需要支持多媒体数据处理、车载网络通信、车辆控制等多种功能，既需要达到工业级安全标准的硬实时性要求，又需要满足娱乐性、互动性等消费级体验需求。

这些应用场景的复杂化对计算机系统的性能提出了复杂的要求，而操作系统作为计算机系统的核心软件组件，通过分层抽象机制实现了对底层硬件资源的统一管理，其进（线）程管理、内存管理、设备管理和文件系统等功能模块对物理层面的中央处理器（CPU）、存储器、输入输出设备以及网络接口等资源进行了标准化抽象，向上层应用提供了访问系统服务的接口。

任务调度作为操作系统的核心机制，其作用不仅体现在对 CPU 时间的直接分配上，还渗透至操作系统的其他功能模块中。例如，在进（线）程管理中，任务调度决定了进（线）程执行的顺序，针对不同的任务类型采取差异化的调度策略，既保证了交互式任务的响应速度，又维持了计算密集型任务的吞吐率；在设备管理模块中，中断机制作为连接应用程序与外部设备的核心桥梁，通过事件驱动的方式实现异步通信。当外部设备产生中断信号时，系统首先保存当前任务执行状态，随后调用中断服务程序解析设备事件，更新相关任务的状态，并将其重新加入调度队列，在中断返回路径上，触发任务调度机制进行决策，基于任务时效性要求评

估是否需要立即进行任务切换，从而有效降低 I/O 任务的等待延迟，从而满足应用程序低延时的需求；

针对操作系统中的任务调度机制展开研究，是计算机系统研究的重要方向之一。本文基于软硬协作的方法，展开对中断响应与任务调度机制的研究，旨在降低系统的响应延时、提高系统的吞吐量，增强人机交互体验，以应对复杂多样化的应用场景需求。

1.2　国内外研究现状

1.2.1　中断机制

中断机制作为一种提高计算机工作效率的技术，最初是用于解决处理器忙等外部设备导致的效率低下问题，在硬件与软件之间构建起高效的协作桥梁。当外部设备完成 I/O 请求时，设备向 CPU 发送中断信号，CPU 被迫暂停当前执行的进（线）程，转而优先处理突发 I/O 事件。中断机制打破了传统顺序执行的局限性，使得计算机即能够快速响应键盘输入、网络数据传输等紧急任务，又能通过精密的时钟中断定期重新分配计算资源，从而营造出多进（线）程流畅运行的假象。

然而，中断机制为系统提供实时响应能力以及时分复用机制的同时，也引入了一些开销。每次中断触发时，CPU 必须暂停当前执行的指令流，保存通用寄存器、程序计数器等现场状态，当中断处理完成后恢复现场，造成直接开销；并且，中断处理程序可能覆盖原任务的指令/数据缓存内容，在原程序恢复执行时，需要重新加载，造成间接开销[1-5]。

在早期，I/O 设备每秒只能完成几百个请求，传统的中断机制足以支持软件层面上的并发处理，由中断引起的开销在系统稳定性方面可以忽略不计。然而，现代高速 I/O 设备每秒可以完成数百万个请求[6-8]。如果使用传统的中断机制，如此高的 I/O 请求完成频率可能会导致不可接受的上下文切换开销。目前，针对这个问题，已经展开了大量的研究工作，主要研究方向如下：

- 轮询机制：为了避免中断频繁干扰处理器，一些研究和技术选择采用轮询机制[9-12]。例如，IX[13] 和 DPDK[14]（以及 SPDK[15]）直接将设备暴露给应用程序，并在用户空间实现轮询，绕过了内核和中断的需求。Intel 的 DDIO[16] 和 ARM 的 ACP[17] 允许网络设备直接将传入数据写入处理器缓存，通过将内存映射的 I/O 查询转换为缓存命中，显著提高了轮询效率。陈旭辉等人提出了一种根据外部事件发生频率分配优先级的方法，并基于这些优先级进行轮询，有效地提高了对外部事件的响应速度[18]。管海兵等人提出的 sEBP（智能基于事件的轮询模型）通过收集各种系统事件优化了网络卡的轮询机

制[19]。为了减少轮询中的高 CPU 开销，Linux 内核建议使用混合轮询[20-21]。这种方法避免轮询整个 I/O 等待时间。相反，I/O 线程会被阻塞一段时间，然后无论 I/O 是否已完成都会被唤醒进行轮询。关键思想是，没有必要从 I/O 过程的开始就启动轮询，因为 I/O 操作需要一些时间来完成。但是，轮询机制要求 CPU 必须定期主动查询设备状态，即使这些查询没有结果。这会导致更高的 CPU 利用率，尤其是在设备不太活跃的情况下[18,22-24]。并且，轮询机制的响应时间受轮询间隔的限制，如果间隔很长，系统的响应可能会延迟，从而影响效率。

- 混合中断与轮询：除了单独使用中断或轮询机制外，一些研究[22,25] 已尝试将两者结合起来，以保留它们各自的优势。Langendoen 等人结合了轮询、中断和线程管理，利用线程调度器感知任务等待外部事件的能力，当 CPU 空闲时，它采用轮询方式接收网络卡数据，并在有可运行线程时切换到中断方式[25]。AlQahtani 等人采用 EDP（使能-禁用中断和轮询）机制，根据系统负载灵活地在轮询和中断策略之间切换[22]。然而，这些混合方法需要依赖启发式的策略，预设的启发式规则难以适应动态变化的负载。
- 硬件支持的快速上下文切换：大量的研究致力于优化中断上下文切换的开销[26-29]。例如，RMTP（响应式多线程处理器）芯片已经在硬件中实现了资源分配、上下文切换和中断唤醒机制。其专用的上下文切换指令能够在四个时钟周期内完成一次上下文切换。此外，RMTP 的硬件中断唤醒机制可以在单个时钟周期内切换到响应任务以处理中断[30]。多管道寄存器架构（Multi Pipeline Register Architecture，MPRA）处理器也已经展示了在几个时钟周期内完成上下文切换的能力，并且利用这一特性优化了中断处理[31-32]。然而，这些基于特定硬件平台的优化方法缺乏通用性。
- 中断合并：尽管前述研究已经优化了单个中断上下文切换的开销，但并未解决由于中断数量过多导致的中断过载问题。因此，许多研究[33-38] 开始探索中断合并技术。中断合并技术减少了 CPU 需要处理的中断数量，以避免中断过载。例如，Salah 等人评估了中断合并技术与传统中断处理的对比[33]。Ahmad 等人提出了一个虚拟中断合并方案，用于在虚拟机监控器中实现虚拟 SCSI 硬件控制器[34-35]。董耀祖等人进行了高效的中断合并和虚拟接收端扩展，充分利用了多核处理器的优势，用于网络 I/O 虚拟化[36-37]。市场上许多网络卡和存储设备都内置了中断合并功能，但这些功能通常基于静态配置，缺乏动态调整能力。Amy Tai 等人[38]在静态配置的基础上引入了适应性中断合并，使设备能够根据请求的延迟敏感性自适应地合并中断。然而，中断合

并与混合中断与轮询存在着相同的问题。

- 硬件智能：一些研究还在探索如何通过硬件增强来提升系统的智能水平。例如，G.R. Gao 等人提出的轮询 Watchdog 硬件扩展，只有在显式轮询无法及时处理到来的消息时才会触发中断，从而减少了不必要的中断[23]。在 Gomes 等人的研究工作中，中断控制器可以识别当前运行线程的优先级，并确保低优先级中断不会抢占高优先级线程的执行，有效地避免了由中断引起的优先级反转问题[39]。Erwin 等人使用的队列内容寻址存储器（CAM）技术提供了一种高效组织和管理中断的新方法[40]。Fabian Scheler 等人使用外围控制处理器（PCP）协处理器可以在中断发生时直接更新内存中的任务状态，只有当高优先级中断任务准备就绪时才通知 CPU 进行重新调度[41]，然而，这种方法也有一定的局限性，因为 CPU 和 PCP 同时访问内存可能会导致高同步和互斥开销，有时会增加中断处理的延迟。这些方法的共同之处在于它们都利用了内核的任务调度器，通过增加与任务控制相关的属性（如任务优先级、状态和时间尺度）来增强它。然而，由于内核和硬件都访问内存中的任务控制块，因此需要额外的同步和互斥机制。

1.2.2　任务调度

任务调度机制直接决定了系统的性能表现，针对不同的应用场景需要的性能指标，目前已经存在着大量的研究工作。

源于中断机制的固有特性，低时延与高吞吐这两项指标是一对矛盾体。一方面，中断提供的抢占机制有效的缓解了传统轮询模式下的队头阻塞问题，保证关键 I/O 请求的优先处理，从而降低响应延时；另一方面，中断机制带来的上下文切换开销、缓存失效开销以及频繁中断造成的中断风暴等问题，会导致系统的吞吐量降低。很多研究工作围绕着这两项指标展开。Wierman 等人在论文[42] 中证明，没有单一的调度策略可以最小化所有可能工作负载的尾部延迟，在任务的尾部延迟较小时，FCFS（first come first served）策略的尾部延迟是渐进最优的；在任务的尾部延迟较大时，则 PS（processor sharing）策略的尾部延迟是渐进最优的。并且这些策略体现出明显的二分性，即在轻尾工作负载下表现良好的策略在重尾负载下表现较差，反之亦然。在此基础上，Prekas 等人在开展 ZygOS[43] 的研究工作时，对单队列和多队列进行了分析，证明了无论是 FCFS 还是 PS 调度策略，单队列的尾部延迟均优于多队列。近年来，有很多研究工作围绕着构建低时延服务进行。在上述的理论前提下，他们结合了操作系统的一些其他优化手段，成功构建了一些低时延服务[1,44-47]。Shenango 保留额外的处理器核心用于满足高负载情况下的低延时要求[46]，但在低负载的情况下存在着 CPU 资源浪费的问题；Shinjuku[45]

使用专门的调度线程为每个请求创建上下文来支持抢占和重调度，当工作线程上的某个请求耗时过多时，调度线程向该工作线程发起中断，让工作线程能够及时响应需要快速处理的请求；Concord[44] 则在 Shinjuku 的基础上进行了优化，它通过编译器插桩达到了用协作式调度近似 Shinjuku 中的抢占式调度的效果，减小了 Shinjuku 中工作线程由于中断抢占带来的开销，但它与 Shinjuku 都需要一个专用的调度核分发任务；Skyloft[47] 使用了 x86 架构下的用户态中断技术[48]，提供了一个通用且高效的用户态调度框架，满足了应用程序多样化的需求，但它只能在特定的硬件平台上发挥优势。

随着吞吐量需求的增加，任务调度的对象也开始发生变化，开始由多线程模型向协程这种轻量级、且易于与异步机制结合的任务模型转化。近年来，以协程为任务单元的研究开始引起学术界和工业界的关注，DepFast[49] 在分布式仲裁系统中使用协程；Capriccio[50] 使用相互协作的用户态线程来实现可扩展的大规模 web server；Demikernel[51] 利用了 Rust 无栈协程上下文切换低成本与适合基于状态机的异步事件处理机制的特点，能够在十几个时钟周期内完成任务切换，向应用程序提供异步 I/O；基于 Rust 协程实现的为嵌入式设备上运行的异步驱动生成框架 Embassy[52] 在中断耗时和中断延迟方面远胜于用 C 语言实现的 FreeRTOS[53]。但这些基于协程模型的任务调度仍然需要中断机制来保证低延时的性能需求。

这些研究取得了相当不错的成果，但仍然存在一些问题。这些研究工作大多是基于软件层面的优化，甚至某些工作是针对特定的硬件设备展开，由软件来适配硬件资源（CPU 等）以及硬件特性（中断机制、I/O 设备特性），其中存在着硬件资源浪费等问题，而围绕着硬件展开的工作，限制了只能在特定硬件平台上才能发挥优势，缺乏通用性。

在应对复杂多样的应用场景时，中断机制与任务调度机制是相辅相成，单纯从某一机制展开研究无法满足复杂多样化的应用场景需求，关于任务调度中的调度策略、任务模型的研究已经足够全面，而中断机制的固有特性也不足以做出新的突破。因此，本文尝试从软硬件结合的角度出发，展开对中断响应与任务调度机制的研究，以期在降低系统的响应延时、提高系统的吞吐量方面取得新的突破。

1.3　本文的主要研究内容与组织结构

任务调度涉及到任务模型、调度算法、同步互斥、中断处理、通信等多个领域，其中每个领域都有众多的研究课题，针对任务调度机制展开研究，是一项复杂的系统性工程。

本文第一章为引言。在这一章中作者首先结合云应用和嵌入式实时应用的发

展趋势，阐述了论文的研究背景以及现实意义，然后立足于中断响应和任务调度的性能，对国内外相关研究进行了详细的分析并介绍了各自的优缺点。

本文第二章为相关技术研究，作者从任务模型出发，介绍了任务模型的变迁过程，并介绍了在任务变迁过程中涉及到的任务调度算法、I/O 模型的变化，并介绍了用户态中断技术的研究现状。

本文第三章基于前文的研究内容，展开了基于软硬协同的中断响应和任务调度设计。相较于传统的基于性能指标进行的任务建模，作者基于任务运行环境进行建模，提出了一种基于执行流的任务模型。在此任务模型的基础上，作者提出了一种软硬协同的任务状态模型，在硬件中实现任务调度、中断处理以及任务通信等功能，定义了一套软硬件交互的接口。

本文第四章将第三章所述的设计方案与具体的硬件平台以及操作系统内核结合，落实到具体的硬件实现，以及与软件之间的适配，搭建了一个基于软硬协同的中断响应和任务调度的系统原型。

本文第五章对前两章所提出的设计方案与系统原型进行了验证试验，通过微基准测试以及综合性能测试，验证了设计方案的有效性。

论文第六章是总结与展望，作者在总结全文的基础上，指出了后续可能的研究方向。

第 2 章　相关技术研究

2.1　任务模型

操作系统任务模型的演化历程深刻反映了计算范式的变革与软硬件协同设计的迭代进步，其发展轨迹始终与底层硬件架构革新及并发编程理论突破紧密耦合。

2.1.1　批处理作业模型

在早期的操作系统中，任务以物理作业卡为载体形成离散的批处理单元，其建模遵循冯·诺伊曼提出的串行执行假设[54]，在内存中仅存放一个作业，每个作业作为封闭的地址空间实体独占系统资源，按照顺序自动执行磁带上的作业，虽然减少了人工操作时间，但 CPU 在等待 I/O 操作时仍会闲置，导致整体效率受限。这一问题促使多道程序设计（Multiprogramming）概念的诞生，允许多个程序同时驻留内存并共享 CPU 资源，当某个程序因 I/O 请求暂停时，CPU 可立即切换执行其他程序，显著提升了资源利用率和系统吞吐量，但多个程序之间没有形成保护边界。并且，批处理作业模型也因为缺乏人机交互性，在程序调试和实时响应方面存在局限。

批处理作业的核心思想在于将任务或数据集合处理，通过减少任务切换开销和资源闲置时间来优化系统的整体效率。这种思想不仅应用于早期操作系统，也深刻影响了现代计算系统设计，例如在大数据场景中通过批量处理提升吞吐量，甚至在某些场景下通过聚合操作降低宏观层面的延迟。

2.1.2　进程模型

Saltzer 于 1966 年完善了进程作为虚拟处理器的形式化定义[55]，Dijkstra 在 THE 系统中首次提出“协同顺序进程”概念[56]。进程模型作为现代操作系统的核心抽象，借助了硬件内存管理单元（MMU，Memory Manage Unit）提供的地址空间隔离机制，保证了不同进程的独立性和安全性。

进程模型引入了状态机模型，其基础的三状态模型包括就绪态（已获资源但等待 CPU）、运行态（占用 CPU 执行指令）和阻塞态（因等待 I/O 等事件暂停执行），扩展的五状态模型增加了创建态（分配初始资源）和终止态（资源回收），而七状态模型进一步引入了挂起状态（如就绪挂起和阻塞挂起），用于处理内存不足时进程被换出到磁盘的情况。状态转换由事件触发，例如运行态因为时间片耗尽转为就绪态，或主动请求资源进入阻塞态，待事件完成后再恢复就绪态。

进程模型使用进程控制块（PCB，Process Control Block）来描述进程的状态以及对系统资源的占用情况（包括程序计数器、通用寄存器、内存映像、文件描述符、进程 ID 等信息），并将 PCB 作为任务的唯一标识以及任务调度的基本单位，操作系统通过管理 PCB 来实现进程的创建、调度、终止等操作。UNIX V7 采用多级反馈队列（MLFQ）算法，结合固定时间片轮转与优先级抢占机制，将缩短了任务平均周转时间[57]，但此时进程既是资源容器（拥有文件描述符、内存映像等），又是调度实体，但重量级上下文切换制约了细粒度并发。

2.1.3　线程模型

由于硬件提供的地址空间隔离机制，进程间通信（IPC，Inter-Process Communication）需要依赖一些复杂机制（例如管道、套接字或共享内存），其上下文切换涉及特权级切换、地址空间切换等操作，导致通信开销显著增大，例如，共享内存通信需要缓存一致性协议来保证数据同步，而消息传递模型则需要多次复制数据。并且，由于进程同时是资源分配的单位，创建和销毁的开销也较大，这些问题促使多线程模型的出现。

线程模型则将执行单元与资源所有权解耦，进程仍然为系统资源分配的单位，线程为执行和任务调度的单元，共享进程的资源（如内存和文件句柄等），降低通信的开销，每个线程在线程控制块（TCB，Thread Control Block）中维护了状态信息（包括程序计数器、通用寄存器等）。这些创新共同促成调度粒度从进程级的毫秒量级向线程级的微秒精度迈进。并且，硬件上的多核处理器技术（例如对称多处理器和非对称多处理器）①以及超线程技术②进一步推动了多线程模型的发展，进一步提高了计算机系统的性能。

多线程模型根据其实现方式可以分为内核级线程（1：1 模型，每个用户态线程对应一个内核态线程）、用户级线程（N：1，多个用户态线程对应一个内核态线程）以及两者的混合模型（M：N，将 M 个用户态线程映射到 N 个内核态线程）。内核级线程由操作系统内核直接管理，线程的创建、调度、销毁等操作都由内核完成，因此线程切换的开销较大，但能够充分利用多核处理器的并行性。用户级线程则由用户空间的线程库管理，线程的创建、调度、销毁等操作都由用户空间的线程库完成，因此线程切换的开销较小，但无法利用多核处理器的并行性，单个线程阻塞会导致其他线程无法执行。混合模型则结合两者优势，既降低开销又支持多核并行。

然而，线程模型需要额外的同步互斥机制（如互斥锁、信号量）来管理共享的

① 多个物理处理器之间共享内存子系统以及总线结构等。
② 单个物理核心模拟多个逻辑核心，实现指令级并行与资源复用。

进程资源，避免数据竞争，这带来了额外的开销。例如，互斥锁的获取与释放操作在竞争激烈场景下可能导致线程阻塞，而自旋锁虽减少上下文切换但会导致 CPU 空转。

2.1.4　协程模型

协程（coroutine）的概念早已经被提出，Marlin 在他的博士论文中总结了协程的核心特征[58]：

- 协程的本地数据在连续调用之间不变；
- 当协程失去控制权时，其暂停执行；在协程重新获取控制权后，协程会从上一次暂停的地方恢复执行；

但是这个通用的定义却没有解决协程结构的相关问题，影响了编程语言中对于协程的支持方式，其中一些实现对协程的表达能力存在误解，此外，一等延续（first-class continuations①）的引入和多线程作为并发编程的“事实标准”采纳，导致协程没有得到广泛使用。直到 Moura 等人证明了完全协程（full coroutines）具有和一次性续延（one-shot continuations）和一次性限定续延（one-shot delimited continuations）同等的表达能力，协程才逐渐复兴[59]。如今，现代编程语言（如 C++、Go、Rust、Python、Kotlin 等）都提供了不同程度的协程支持。

协程作为协作式任务单元，其无栈式设计（Stackless）相较于进（线）程模型，占用的内存资源更少。协程通过 CPS（continuation-passing style）②或者状态机保存执行现场，将上下文切换开销降至纳秒级，并且其与 IO 多路复用、事件驱动等机制非常契合，协程的这些特性展现出在高并发场景下的潜力，现代的运行时甚至能够每 GB 内存承载百万级轻量协程。

2.2　任务调度

Linux 内核中的 $O(1)$ 调度器设计突破了传统 $O(n)$ 调度器的性能瓶颈，通过创新性的双队列轮转机制实现了时间复杂度优化，$O(1)$ 调度器采用基于优先级的分层队列架构，将可运行任务划分为 *active*（活跃队列）和 *expired*（过期队列）两个独立集合。当任务的时间片耗尽时，调度器会即时执行优先级重计算，而非等待所有任务时间片耗尽后才统一处理，这一设计显著降低了调度决策的时间复杂度。而 Linux 内核使用的公平调度算器（CFS，Completely Fair Scheduler）为了保证公

① continuation 是计算机程序控制状态的抽象表示，它实现了程序控制状态，即 continuation 是一个表示程序执行中给定点的计算过程的数据结构；所创建的数据结构可以被编程语言访问，而不是隐藏在运行时环境中。

② 是一种通过显式传递“后续操作”来控制程序执行流程的编程范式。其核心思想是将函数原本隐式返回值的逻辑，替换为将结果传递给一个显式的回调函数（称为 Continuation）。

平性，赋予任务虚拟运行时间（*vruntime*）的概念，保证了在任意调度周期内，所有任务 *vruntime* 的累积增量相等，从而实现公平调度。

在任务调度算法的建模和优化领域，现有研究通常围绕系统核心性能指标展开针对性设计。以 Concord[44] 为例，它通过构建涵盖抢占式调度、线程同步与通信、任务分发等环节的开销模型，对系统尾部延时的生成路径进行解耦分析，最终在保障吞吐量的同时将尾部延时优化至微秒级。而葛文博等人提出的众核嵌入式实时调度策略，引入了任务关键度概念，建立任务最坏响应时间和优先级分配的数学模型，有效的降低了传统调度在众核环境下的调度开销[60]。

然而，现有研究多聚焦于单一或有限维度的性能指标建模，在动态可重构的系统中，这些确定性的模型与动态场景的适配不足，难以适应复杂应用场景的多样化需求。

2.3　I/O 模型

I/O 模型描述了应用程序与外部设备之间通信的过程，应用程序通过系统调用来执行 I/O 操作，内核来执行具体的 I/O 操作。I/O 模型设计面临的核心问题是 CPU 与外部设备速度不匹配，不同的模型通过不同的方式协调这一矛盾：（1）同步与异步：同步模型要求应用程序主动轮询或等待结果，而异步模型则由内核主动通知应用程序；（2）阻塞与非阻塞：阻塞模型会挂起线程直到操作完成，而非阻塞模型则允许线程立即返回并执行其它任务。常见的 I/O 模型包括以下几类：

- 同步阻塞 I/O：应用程序发起 I/O 操作后，线程会完全阻塞，直到内核完成数据准备和拷贝到用户空间两阶段操作。例如发起 `read()` 系统调用时，若内核缓冲区无数据，线程进入休眠状态，直到数据到达后被唤醒。这个模型逻辑简单直观，但线程资源利用率较低，适用于低并发的场景，在高并发时需要创建大量线程而导致内存与调度开销增大。
- 同步非阻塞 I/O：应用程序通过 `fcntl()` 系统调用将文件描述符设置为非阻塞模式，当发起 `read()` 系统调用时，若内核无数据，立即返回 `EWOULDBLOCK` 错误，不会将线程阻塞在内核中，线程回到用户态可以执行其他操作，但仍然需要定期轮询 I/O 状态。这种模式减少了线程空闲等待的时间，提升了单线程利用率，但定期轮询导致了 CPU 空转，存在资源浪费，并且受到轮询间隔的影响，在数据准备就绪到应用程序进行处理存在一定的响应延时。
- I/O 多路复用：单个线程通过 `select()`、`poll()`、`epoll()` 等系统调用，阻塞在内核中，内核会监控多个文件描述符的 I/O 状态，当其中某个文件描

述符就绪时，系统调用就会返回到用户态，线程可以执行 I/O 操作。这种模型适用于大量 I/O 事件的场景，避免了线程创建和销毁的开销，但这些机制需要内核维护额外的数据结构以及在内核与用户程序之间拷贝文件描述符集合，存在着一定开销。

- 异步 I/O：应用程序通过异步读/写接口发起 i/O 操作后，立即从内核态回到用户态执行其他操作，当 I/O 事件准备就绪，内核将数据拷贝至应用程序的缓冲区中，内核发送一个信号或者或者基于线程的回调函数来完成 I/O 处理。以信号机制作为异步通知机制为例，应用程序使用 `sigaction()` 系统调用注册 `SIGIO` 信号后，回到用户态执行其他的任务，当 I/O 准备就绪时，内核向应用程序发送 `SIGIO` 信号，当应用程序下一次进入内核并返回到用户态时，会进入预先注册的信号处理函数中，完成 I/O 事件的处理，在信号处理函数尾部，通过 `sigreturn()` 系统调用使应用程序从被打断处恢复执行。这种机制避免了轮询造成的 CPU 开销，但由于信号处理函数需要等到应用程序下一次进入内核并返回时才能执行，这与同步非阻塞 I/O 模型存在着类似的响应延时，并且增加了额外的系统调用开销。此外还存在其他的异步通知机制，例如用户态中断技术。异步 I/O 模型通过解耦 I/O 操作与程序执行流，突破了传统同步模型的性能瓶颈，在构建高并发的云服务场景下效果显著。

在选择 I/O 模型时，需综合考虑应用场景的负载类型、性能需求、并发规模及系统资源限制等因素。例如，高并发短连接场景（如 Web 服务器）通常采用 I/O 多路复用模型以高效管理大量连接，而实时系统或响应式界面更适合非阻塞 I/O 以避免阻塞延迟；对于简单低负载应用（如嵌入式设备），阻塞式 I/O 因其实现简单且资源消耗低可能是更优选择。此外，跨平台兼容性、开发复杂度（如回调机制）和操作系统特性（如 Linux 对异步 I/O 的模拟实现）等因素也需要考虑。

2.4　用户态中断

用户态中断（User-Level Interrupts）作为近年来计算机体系结构和操作系统领域的研究热点，旨在绕过传统内核态中断处理的高开销，通过硬件与软件的协同设计实现用户态程序的低延迟异步事件响应。其核心思想是通过硬件直接传递中断信号到用户空间，并允许用户态程序注册自定义的中断处理逻辑，从而消除传统的由内核处理中断机制导致的上下文切换开销（特权级切换开销，在开启内核页表隔离机制（KPTI）[①]后，还包括地址空间切换开销）。

① 内核页表隔离机制是 Linux 内核的一项功能，它将用户空间和内核空间页表完全分离来强化内核安全性。

目前已经存在多项工作围绕着用户态中断技术展开，Skyloft[47]、uProcess[61]使用了 Intel 提出的 UINTR（User Interrupts）扩展[48] 来实现用户态的抢占式调度。此外，RISC-V 指令集[62] 的 N 扩展也实现了对用户态中断技术的支持，Sandro 等人已经将其用于嵌入式系统，用于构建可信执行环境[63]。

用户态中断技术除了可以为实现用户态抢占式调度提供了硬件原生支持，还可通过内核旁路（Kernel Bypass）和零拷贝（Zero-Copy）机制显著加速 IPC。用户态中断充当通知机制，中断信号直接触发用户态处理逻辑，并且通过共享内存或其他机制绕过多层内核协议栈与内存拷贝操作，从而消除传统 IPC 中因系统调用、上下文切换和数据复制导致的开销。这对于高性能通信具有重要影响，但用户态中断技术也面临着安全隔离性（如恶意中断注入、中断嵌套）、跨平台兼容性（需依赖特定 CPU 架构）以及调试复杂性（缺乏内核调试接口）等挑战。

2.5　本章小结

本章围绕中断响应与任务调度的相关技术展开探讨。首先从硬件体系结构的迭代升级切入，揭示了计算机系统如何通过软硬件协同设计推动任务模型的持续演进——从早期的单任务串行执行逐步发展为支持多任务抢占式调度的复杂架构；然后从不同应用场景对延时、吞吐量等性能指标的需求展开对任务调度算法以及 I/O 模型的讨论，最后介绍了用户态中断技术的研究现状。这些技术的发展和应用为本文后续的研究工作提供了理论基础和技术支持，尤其是用户态中断技术，它打破了内核与用户态的壁垒，推动了软硬件深度融合的进程。

第 3 章　基于软硬协同的中断响应和任务调度设计

在第二章中已经对任务模型以及相关的任务调度、I/O 模型等进行了详细的介绍，传统的进程、线程、协程模型是针对内核或者用户程序的各个功能单元来进行建模的，系统通过对这些任务对应的任务控制块进行管理，来实现任务的调度、同步、通信等功能，因此这些模型包含了任务生命周期内的所有相关的执行流（不仅包括在用户态运行的由应用程序定义的执行流，还包括了运行在内核中用于提供系统服务、中断处理以及异常处理的执行流），但这带来了以下问题：

- 无法描述边界代码：在尚未完成任务管理模块初始化时，系统执行的初始化代码无法划分到其中的某个模型的实例中，在任务管理模块初始化之后，才划分到某个实例中。
- 无法描述中断处理例程：当 CPU 收到中断信号后，硬件保存部分现场，并暂停当前正在执行的任务，软件将剩下的寄存器现场保存在被暂停任务使用的栈上，紧接着复用被暂停任务的使用的栈跳转至中断处理例程，来完成中断信号的处理。在完成中断处理后，返回到被暂停的任务继续执行或者切换到新的任务。尽管在处理的过程中，当前正在运行的任务记录没有改变，且中断处理例程复用了当前被打断任务的栈，但它的功能不属于当前被打断的任务需要负责的功能单元，因此中断处理例程是无法归类到这些模型中的。由于中断处理例程的时间过长会严重影响系统的性能，Linux 内核将中断处理划分为上半段和下半段，上半段为一些必须紧急处理的事项（例如将设备的中断寄存器清空，完成必要的数据拷贝操作等）；下半段则转化为一个单独的任务（例如网络协议处理等）。但这种将中断处理划分上下半段的方式仍然无法用传统的任务模型来描述上半段。如果使用单独的任务控制块来描述中断处理例程，这会增加系统的复杂性与开销，因为中断处理例程使用单独的栈来保存自己的状态，这会增加额外的任务切换开销，让中断响应延时（从中断触发产生到中断函数执行的时间）增加，并且由于需要为每个中断都需要准备额外的中断任务栈，无法应对中断嵌套的情况，在多核环境下，每个 CPU 都需要为同一个中断准备一个单独的中断任务栈，导致内存占用增大。
- 执行流之间的通信繁琐：由于内核只能对任务在内核中的执行流进行感知，用户态的执行流的变化无法被感知，因此导致了用户态的执行流无法直接与内核态的执行流进行通信，增加了通信的开销。在上一章节中对 I/O 模型的介绍详细描述了这个问题。（1）当用户态的执行流通过系统调用切换到内核

执行流，若内核执行流阻塞，用户态的其他执行流将无法继续运行（同步阻塞 I/O 模型），内核执行流无法通知其他用户态执行流继续运行；（2）若内核执行流返回 EWOULDBLOCK 通知其他用户态执行流可以继续执行，则增加了额外的轮询开销（同步非阻塞 I/O 模型）；（3）在 I/O 多路复用模型中，内核执行流需要额外的机制（文件描述符集合的管理、事件循环等）来维护用户态执行流与 I/O 的状态信息；（4）基于信号或基于线程的回调函数构建的异步 I/O 模型增加了特权级切换的开销。

这些问题的核心在于任务模型的抽象粒度过大，需要将任务模型的粒度更加细化，并且需要打通执行流之间的通信壁垒。为此，本文提出了一种更细粒度的任务模型——基于执行流的任务模型。基于这种细粒度模型以及“**任务通信**”的视角，本文从以下方面展开对系统性能的优化：

- 任务本身的开销；
- 任务切换的开销；
- 任务之间的通信开销；

针对系统中实现一定功能的普通任务的优化收益甚小，因此本文将优化的目标定为“中断处理”这个特殊的任务以及任务切换和任务之间的通信开销，设计了一套软硬协同的中断处理和任务调度机制，将中断处理任务和任务调度卸载到硬件中，减小了任务切换过程中调度的开销以及中断处理的开销，同时在硬件中搭建了任务之间的通信桥梁，减小任务的通信开销，并且提供了软硬件的交互接口。

3.1　基于执行流的任务模型设计

软件的执行流是指在 CPU 上执行的一段特定的功能单元，它既包含程序代码在运行过程中指令执行顺序和逻辑路径的动态描述（如顺序执行、条件分支、循环跳转等），也包含维持运行所需的独立资源环境（每个执行流属于某个操作系统或操作系统中的某个进程，运行于特定的特权级和地址空间中）。不同的执行流既能独立运行，也能通过通信协作完成特定功能。

本文基于软件执行流所处的独立资源环境（地址空间、特权级、堆栈）来构建任务模型，其中地址空间用来区分不同的操作系统或者操作系统中的进程，特权级用来区分用户态和内核态的执行流，堆栈用来保存执行流的函数调用关系等执行逻辑（线程模型）或状态（协程模型）[①]。因此，本文在保留进程概念的基础上（进程用于区分地址空间）使用 $\boldsymbol{T}_{(P_i,L_j,S_k)}$ 来符号化标识某个执行流的具体实例，其

① 无栈协程使用有限状态机来表示内部的执行逻辑。

中：

- P_i：表示执行流所处的进程（地址空间），P_i 可以是内核，也可以是某个进程；
- L_j：表示执行流运行的特权级，L_j 可以是用户态、内核态或其他特权级；
- S_k：表示执行流使用的堆栈；

由于这种任务模型的设计，任务之间的边界更加清晰，应用程序的逻辑由 N 条在用户态运行的执行流与 M 条在内核中运行的执行流共同组成，每个执行流构成独立的任务单元，任务之间的切换可以使用三元组（$[prev] \longrightarrow [next] : \{condition\}$）进行描述，其中 $prev$、$next$ 分别表示切换前后的任务，$condition$ 表示切换的条件，可以是时间片耗尽、事件触发、系统调用等。典型的任务切换[①]如下：

- 相同地址空间内不跨越特权级的切换：

$$T_{(P_i,L_j,S_k)} \longrightarrow T_{(P_i,L_j,S_l)} : \{\text{调度} \mid \text{内核态收到中断}\}$$

  包括了分别在用户态/内核态由于调度引起的任务切换，这种切换只需要根据堆栈上保存的信息完成寄存器的上下文切换，因此开销较小。此外，无论是否使能 KPTI 机制，内核态任务所属的地址空间中均包含了内核地址空间的映射，因此在内核的任务收到中断信号后，不需要切换地址空间也不需要切换特权级，尽管复用了被暂停任务的栈，但栈上没有保存与中断处理任务相关的信息，因此可以认为切换了堆栈，所以这种切换属于第一类切换。

- 相同地址空间内跨越特权级的切换：

$$T_{(P_i,L_j,S_k)} \longrightarrow T_{(P_i,L_l,S_n)} : \{\text{系统调用} \mid \text{异常} \mid \text{中断}\}$$

  在没有使能 KPTI 机制的情况下，进程地址空间中包含了内核地址空间的地址映射，因此用户态任务获取内核提供的服务不需要切换地址空间，只需要切换到内核态下的任务；同理，异常与中断处理也不需要切换地址空间；

- 跨越地址空间但不跨越特权级的切换：前一个任务与被调度的下一个任务属于不同的进程，这种切换必须在高特权级中才能进行。

$$T_{(P_i,L_j,S_k)} \longrightarrow T_{(P_l,L_j,S_n)} : \{\text{内核态任务调度}\}$$

- 跨越地址空间且跨越特权级的切换：因为需要跨越地址空间，需要先从相同地址空间中的低特权级态切换到高特权级态，才可以切换地址空间，因此这种情况属于上述的第二类切换与第三类切换的组合。当使能 KPTI 机制时，用户态的任务必须要先切换到内核态下，再切换到内核的地址空间中，才可以使用系统提供的服务。

① 在任务切换过程中，切换代码本身属于一段特殊的代码，它与任务本身的功能逻辑无关，但隶属于每个任务，当任务运行到切换代码时，切换代码即属于那个任务。

通过这种细粒度的任务建模，传统的系统调用以及中断处理可以被视为“**任务通信**”。系统调用可以视为用户态的任务将系统调用相关的参数信息打包成消息，通过相关的指令传递给处于内核态的用于处理系统调用的任务，内核态的任务处理完成后将结果返回给用户态的任务。

中断处理从表象上看是中断处理任务与硬件设备之间的交互，但其本质仍然是中断处理任务与其他任务之间的通信。例如，当系统调用任务调用 `sleep()` 函数时，它告知中断处理任务，它需要在一段时间后被唤醒，中断处理任务在接收到时钟中断后，唤醒这个系统调用任务，这一套通信的流程构成了内核中提供的 `sleep()` 系统服务，同理其他的系统服务也可以通过这种通信机制来进行解释。

3.2　基于软硬协同的中断响应和任务调度机制

任务调度器需要根据任务的状态以及调度策略来决定任务所处的队列，并维持队列的偏序关系。通常，任务的状态被保存在任务控制块中的某个字段中，任务调度器对任务状态字段所在的位置进行读写操作实现对任务状态的管理，并根据任务的状态将任务插入到不同的任务队列中。这一系列行为要求任务调度器获取任务控制块的位置、状态字段在任务控制块的偏移、任务的优先级以及任务队列的位置等信息。因此，将任务调度从软件卸载到硬件中，需要让硬件能够感知任务的状态、优先级等信息，并且在硬件中维护若干个任务队列，从而实现硬件任务调度器。

中断处理任务的功能是根据收到的中断信号，找到对应的任务，修改任务状态，并将任务放入到正确的任务队列中，这一系列操作需要获取中断信号、任务的状态、优先级等信息，以及任务队列的位置。因此，将中断处理任务卸载到硬件中需要以实现硬件任务调度器为前提。因此，基于软硬协同的中断响应和任务调度机制包括以下几个方面：

1. 建立软硬协同的任务状态模型，实现硬件任务调度器；
2. 以硬件任务调度器为前提，实现硬件中断处理机制（中断响应）；
3. 基于硬件中断处理机制实现硬件支持的任务通信机制；

3.2.1　硬件任务调度器

实现硬件任务调度器需要对传统的任务状态模型进行扩展，使得硬件能够识别并修改任务的状态，本文设计了基于软硬协同的任务状态模型，如图 3.1 所示。

任务状态仍然为创建、就绪、运行、阻塞和退出这五种状态，其中灰色框线中的就绪态与阻塞态以及它们之间的状态转移由硬件任务调度器进行。硬件任务调

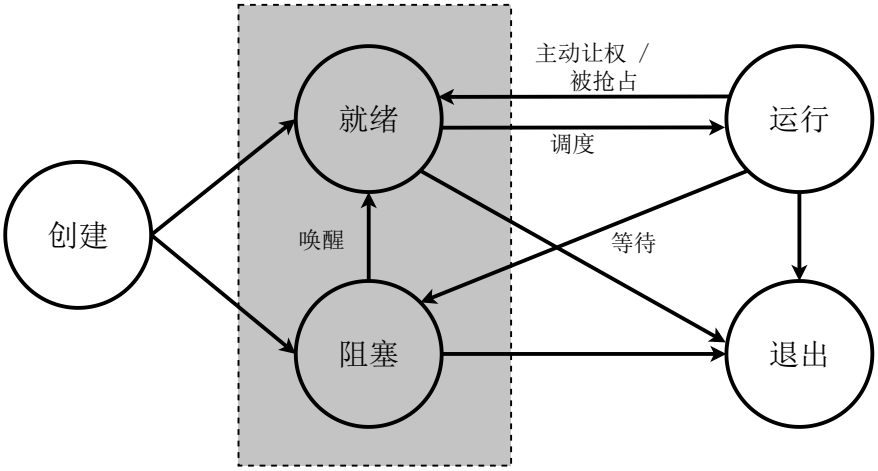


图 3.1　基于软硬协同的任务状态模型

度器在内部维护若干不同的就绪任务队列 RQ_* 和阻塞任务队列 BQ_*，每个队列中存放任务的标识 $T_{(P_i,L_j,W_k)}$（由任务的符号化标识 $T_{(P_i,L_j,S_k)}$ 和任务的优先级 W_k 等信息复合而成），并通过任务的优先级等信息维护任务队列的偏序关系。硬件调度器可以根据任务队列感知任务的状态以及优先级等信息，硬件通过将任务标识在这些队列中迁移实现对任务状态转移。硬件无法感知灰色框线外的其他任务状态，这些状态以及它们之间的转移由软件来维护，其他任务状态与就绪态、阻塞态之间的状态转移也由软件通过读写硬件调度器的响应端口来发起。任务的状态转移如下：

- 创建 ⟶ 就绪、创建 ⟶ 阻塞：软件直接修改任务状态后，写硬件调度器的端口将创建的任务的标识添加至硬件调度器的就绪队列或者阻塞队列中；
- 就绪 ⟶ 运行：软件读取硬件调度器的端口可以获取到就绪任务队列中最高优先级的任务标识，软件将任务的状态修改为运行态；
- 运行 ⟶ 就绪：软件修改任务状态后，写硬件调度器的端口将任务标识添加至硬件调度器的就绪队列中；
- 运行 ⟶ 阻塞：软件修改任务状态后，写硬件调度器的端口将任务标识添加至硬件调度器的阻塞队列中；
- 运行 ⟶ 退出：软件直接修改任务状态；
- 阻塞 ⟶ 就绪：硬件调度器根据特定的条件（例如收到中断信号）将阻塞队列中的任务标识添加至就绪队列中；这个过程硬件不直接修改任务状态，在软件从硬件调度器中取出就绪任务后，直接修改任务状态；
- 就绪 ⟶ 退出、阻塞 ⟶ 退出：软件向硬件调度器端口中写任务标识，硬件调度器将任务标识从就绪队列或阻塞队列中移除后，软件修改任务状态；

软件需要申请相应的就绪队列和阻塞队列，硬件任务调度器在成功分配了这

些队列的硬件资源 $RQ_{(P_i,Lj)} + BQ_{(P_i,Lj)}$ 后，将会把这些队列相应的句柄（软件可以访问的硬件端口）返回给软件，软件通过该句柄来使用硬件任务调度器的功能。由于软件运行于一定的地址空间和特权级下（进程），在硬件队列申请成功后，直到进程生命周期结束或者软件主动释放掉硬件队列之前的这段时间，硬件队列与进程的地址空间 P_i 和特权级 L_j 相绑定。

使用硬件任务调度器，可以利用硬件机制来实现对任务队列的互斥访问，从而避免由于并发访问软件调度器带来的同步互斥开销。

3.2.2　硬件中断处理机制

硬件中断处理机制的本质是用硬件电路来唤醒由于等待中断（I/O、定时器事件等）而进入阻塞状态的任务，这需要软硬件共同完成，其完整的流程如下：

1. 软件向硬件调度器的端口中写任务标识，将其注册为某个中断信号的响应任务（硬件任务调度器将任务标识放入到与中断相关的阻塞队列中），预先将任务标识与中断进行绑定；
2. 当硬件调度器收到中断信号后，根据中断信号的来源，找到对应的阻塞队列以及负责响应中断的任务标识，根据任务标识中的优先级信息，将任务标识从阻塞队列中移除，并放入到就绪队列中的合适位置；
3. 如果唤醒的任务的优先级比当前正在 CPU 上运行的任务的优先级高，那么硬件任务调度器应该向 CPU 发送中断，CPU 在收到硬件任务调度器的中断信号后，直接进行任务切换。

以硬件处理时钟中断为例（如图 3.2），硬件任务调度器在内部实现了与时钟中断相关的阻塞任务队列（BQ_{timer}）来维护由于等待定时器事件而进入阻塞状态的任务标识（$T_{(P_i,L_j,W_k)}(\tau_t)$，标识中增加了定时器的截止时间 τ_t），该队列根据任务所等待的定时器的截止时间 τ_t 的先后顺序进行排序。在 $\tau_{2.5}$ 时刻，硬件时钟产生时钟中断，将信号发送给硬件任务调度器，硬件任务调度器检查 BQ_{timer} 队列头部的任务所等待的定时器事件的截止时间，发现任务标识 $T_{(P_1,L_1,W_3)}(\tau_1)$ 和 $T_{(P_4,L_1,W_7)}(\tau_2)$ 的截止时间小于当前时间，因此，硬件任务调度器根据任务标识中的进程标识和特权级标识找到对应的就绪队列，并根据任务标识中的优先级信息将其放入到就绪队列的适当位置，完成硬件中断处理。

通过硬件快速响应中断，唤醒阻塞的任务，可以让 CPU 不被一些中断信号打断，减少由于中断导致的中断上下文切换开销。若被唤醒的任务优先级较高，需要进行任务切换，在这种情况下，仍然可以减少由于中断导致的调度开销。

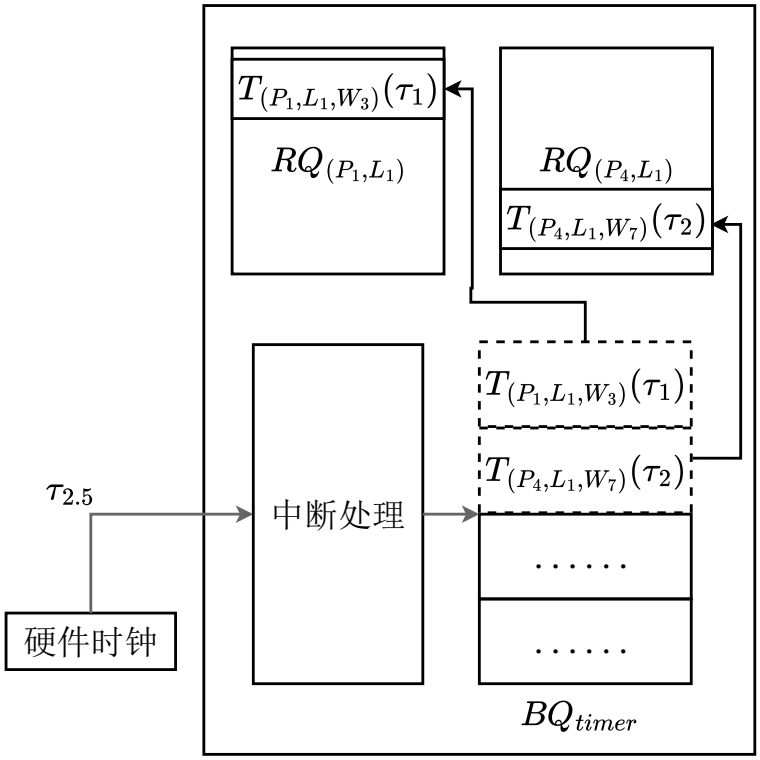


图 3.2　硬件处理时钟中断

3.2.3　硬件支持的任务通信机制

通信中的两个任务分别为发送方与接收方，它们属于不同的地址空间或不同特权级下的进程。使用硬件支持的任务通信机制的前提是通信的双方都使用了硬件提供的任务调度器功能（双方进程分别使用不同的任务队列），并且，在上述的硬件中断处理机制的基础上，将中断信号的发送方从硬件设备扩展为软件上的发送方任务。此外，硬件中还需要维护与双方通信能力相关的能力槽，并配置额外的能力检查机制来避免恶意通信。其完整的通信流程如下：

1. 发送方与接收方申请使用硬件任务调度器；
2. 发送方任务向硬件任务调度器的端口中写接收方任务的进程标识以及中断号，硬件任务调度器将这两项信息填写至发送方的发送能力槽中，从而注册可以向指定接收方进程发送的指定中断的能力；
3. 接收方任务向硬件任务调度器的端口中写发送方任务的进程标识、中断号以及自己的标识，硬件任务调度器将这些信息填写至接收方的接收能力槽中，从而注册可以接收指定发送方的指定中断的能力；
4. 发送方任务向硬件任务调度器的端口中写接收方的进程标识以及中断号，尝试进行通信。硬件任务调度器首先检查发送方的发送能力槽中是否存在对应

的接收方进程标识以及中断号。如果存在，则硬件任务调度器根据接收方的进程标识找到对应的接收方进程所使用的就绪队列以及接收能力槽，如果接收能力槽中存在对应的发送方的进程标识、中断号以及接收方的任务标识，则硬件任务调度器将接收方的任务标识放入到接收方的就绪队列中的适当位置，完成通信。

基于硬件支持的通信机制，跨域不同地址空间和特权级的执行流之间可以实现快速交互，减少了切换地址空间以及特权级的开销。

3.3　软硬件交互接口

I/O 端口作为软件访问硬件的入口，也是硬件响应软件操作并返回结果的媒介。软件通过向 I/O 端口发送读写命令，将数据与控制信号传递给硬件，最终转化为对硬件中的寄存器或状态的操作，从而驱动硬件完成特定的功能。在本文的基于软硬协同的中断响应和任务调度机制中，软硬件相互协作，共同完成任务调度、中断处理以及任务通信等功能，软硬件之间的交互接口如表 3.1 所示。

表 3.1　软硬件交互接口

功能	接口	说明
硬件资源申请与回收	alloc	软件写端口，申请的队列资源，并从端口中读出队列资源句柄
	free	软件写需要回收的句柄，硬件回收队列资源
任务调度	task_enqueue	软件写任务标识，硬件将标识添加至就绪队列中
	task_dequeue	软件从硬件中读出优先级最高的就绪任务标识
	remove_task	软件写任务标识，硬件将删除队列中的任务标识
外部中断处理	register_extint	软件写任务标识，将任务与中断绑定
	unregister_extint	软件取消任务与中断之间的绑定
任务通信	register_sender	软件写接收方进程标识和中断号，注册发送能力
	unregister_sender	软件写接收方进程标识和中断号，取消发送能力
	register_receiver	软件写发送方进程标识、中断号和接收方任务标识，注册接收能力
	unregister_receiver	软件写发送方进程标识和中断号，取消接收能力
	send	软件写接收方进程标识和中断号，发起通信

3.4　本章小结

本章提出基于执行流的任务模型，从“**任务通信**”的视角出发，将任务调度、中断处理以及任务通信组合成统一整体，利用硬件快速、高效等特点对其进行加速，设计了基于软硬协同的中断响应和任务调度机制，综合性地减小系统开销，提高系统应对应用场景复杂多样的性能需求。

第 4 章　基于软硬协同的中断响应和任务调度实现

本章根据第三章所述的任务模型和设计方案，在 FPGA 平台上实现了上述的具备中断处理、任务通信以及任务调度的控制器（TAIC，Task Aware Interrupt Controller），并且进行了相关的软件适配，实现了基于软硬协同的中断响应和任务调度机制的系统原型。

4.1　系统整体结构

图 4.1 给出了基于软硬协同的中断响应和任务调度系统的整体架构。系统由 CPU、TAIC 和外部设备三部分组成。外部设备的中断信号连接到 TAIC 的中断处理模块中，TAIC 内部维护了若干套用于不同的地址空间和特权级下的队列资源（包括就绪队列、外部中断能力槽、接收能力槽和发送能力槽），不同的队列资源之间通过任务通信模块构建硬件的通信通道。CPU、TAIC 与外部设备通过总线相连。运行于 CPU 上的处于不同地址空间和特权级之间的任务通过软硬件交互接口使用 TAIC 提供的中断处理、任务调度和任务通信功能。

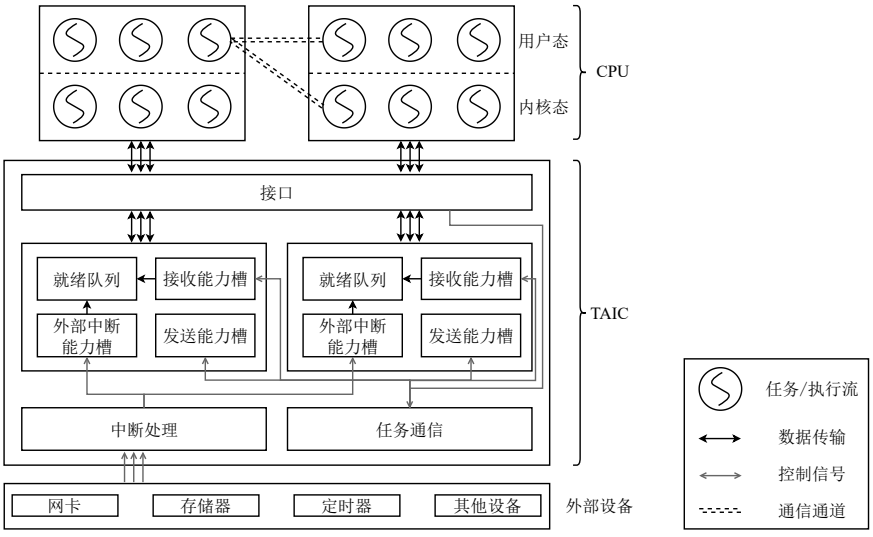


图 4.1　基于软硬协同的中断响应和任务调度系统整体架构

4.2　控制器实现

根据 TAIC 提供的功能划分，TAIC 由四个模块构成：任务调度、中断处理、任务通信和队列资源分配与回收。

4.2.1　任务调度

TAIC 内部维护了若干任务队列，并且 TAIC 能够在不同的任务队列之间迁移任务标识，维持任务队列的偏序关系，以此来向软件提供任务调度的功能。

其中就绪队列的实现直接在硬件层面提供了软件中的任务队列所需要的出队、入队、优先级排序和负载均衡等功能，能够满足软件的任务调度需求。就绪队列由一块脉冲阵列和若干个局部队列元信息组成，如图 4.2 所示。

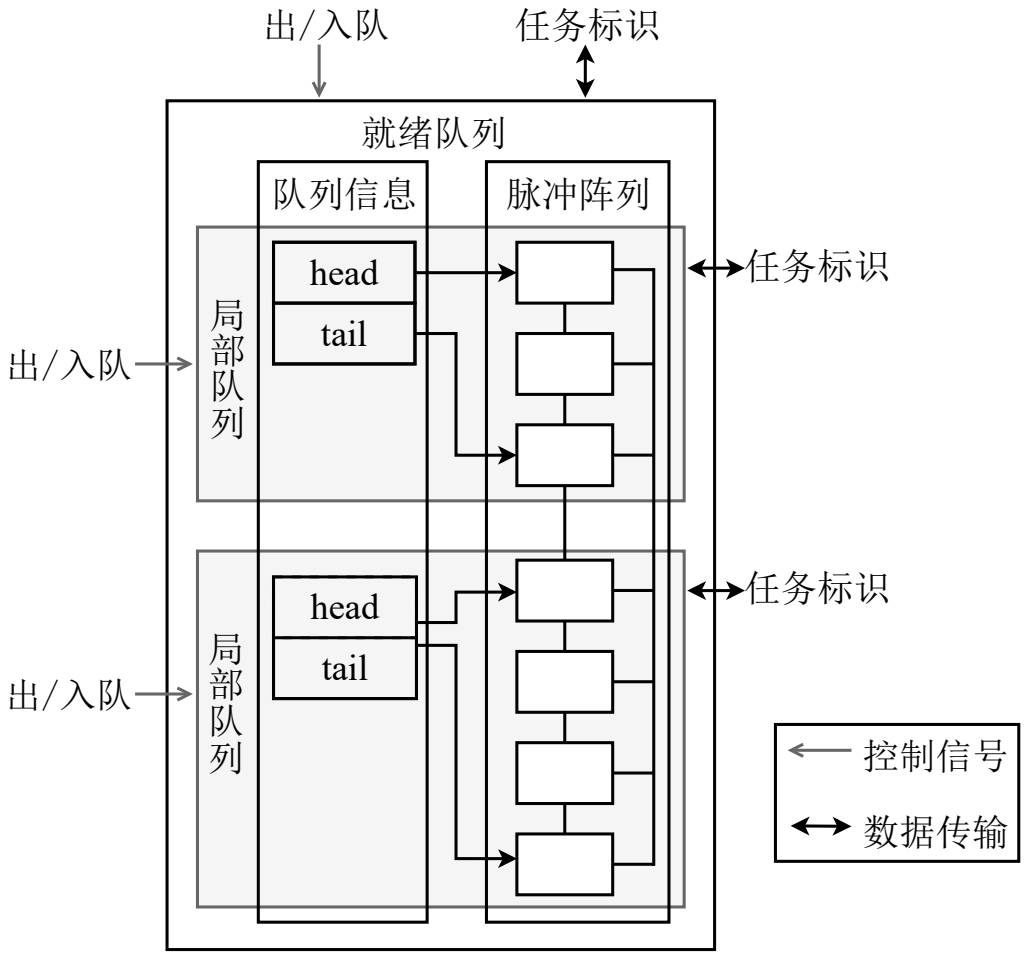


图 4.2　就绪队列

脉冲阵列用于存放任务标识，当需要删除某个位置的任务标识时，该位置后的所有任务标识自动前移即可完成删除操作；当需要在某个位置插入任务标识时，先将该位置包括后续的所有任务标识后移，然后再将任务标识插入到该位置即可完成插入操作。

基于脉冲阵列，只需要记录队列在脉冲阵列中的头部 *head* 和尾部 *tail* 位置即可实现局部队列。由于局部队列共用同一块脉冲阵列，因此当某个局部队列在进

行出/入队操作时，位于该局部队列后的所有局部队列的 *head* 和 *tail* 指针都需要更新。局部队列支持的操作为：

- 出队：取出脉冲阵列中 *head* 处的任务标识，将自己的 *tail* 指针 -1，将后续所有局部队列的 *head* 和 *tail* 指针 -1；
- 入队：将任务标识插入到脉冲阵列中 *tail* 处，将自己的 *tail* 指针 +1，将后续所有局部队列的 *head* 和 *tail* 指针 +1；

这些局部队列共同组成了一个就绪队列（全局队列），既可以单独对局部队列执行出/入队操作，也可以对全局队列执行出/入队操作。全局队列的容量存在上限（脉冲阵列可以存放的任务标识的数量），但局部队列的容量没有固定上限，当全局队列达到容量上限时，对任意局部队列进行的入队操作将会失败，无法插入任务标识。局部队列在执行出队操作时集成了负载均衡的功能，当某个局部队列为空时，会从其他具有任务的局部队列中窃取任务标识（窃取的顺序为局部队列的 *head* 指针指向的脉冲阵列位置的顺序）；只有当整个全局队列为空时，才会返回空值。

基于局部队列的窃取机制以及窃取的顺序，整个全局队列可以当作优先级队列使用，其中每个局部队列中存放优先级相同的任务标识，并且按照先进先出的规则进行排序。不同的局部队列的优先级不同，*head* 指针在脉冲阵列的位置越靠前，优先级越高。

基于上述的就绪队列实现，软件可以灵活地使用局部队列和全局队列，已实现不同的调度策略，并且减少因为固定局部队列容量导致的硬件资源的浪费。

由于硬件资源有限，阻塞队列只用于保存因为等待外部设备以及等待任务通信而进入阻塞状态的任务的标识，对应的实现为外部中断能力槽和接收能力槽，分别在 4.2.2 和 4.2.3 中描述。

4.2.2　中断处理

TAIC 快速处理中断机制依赖于任务队列中的维护的外部中断能力槽，它记录了由于等待外部设备中断而进入阻塞状态的任务的标识。由于应用程序通常只有一个任务用于处理外部中断，因此，针对每个外部中断信号只准备了一个槽位。其结构以及处理流程如图 4.3 所示。

ISR_i 表示与中断号为 *i* 相关的任务标识，在使用硬件快速处理中断能力时，需要软件向外部中断能力槽中注册任务标识（该任务为阻塞状态），注册成功即为使能了硬件外部中断处理机制。当外部设备产生中断信号时，TAIC 的中断处理模块会根据中断信号来检查对应的槽位，若其中存在任务标识，将任务标识从槽中取出，并放入到就绪队列中的合适位置，以此来唤醒因为等待外部设备中断而进

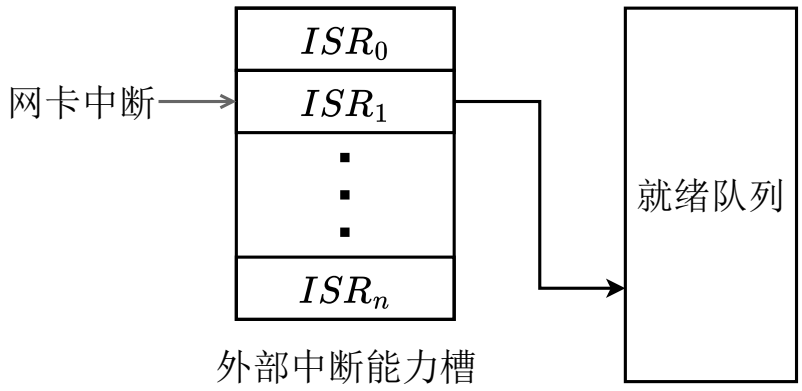


图 4.3　外部中断处理

入阻塞状态的任务。一旦完成唤醒操作，外部中断能力槽中该中断对应的槽位会被清空，TAIC 的快速处理该中断的能力会被屏蔽，直到软件再次注册该中断的处理任务重新使能该机制。

此外，若中断的处理任务的优先级较高，TAIC 会向 CPU 发送中断信号，CPU 在收到中断信号后，直接进行任务切换。TAIC 的硬件快速处理中断机制可以无视特权级，可以在用户态使用，但需要额外的机制来保证任意时刻只有一个进程或内核在使用同一个外部设备。但对于某些特殊的应用场景，例如在虚拟化场景中，多个内核使用同一个硬件定时器，则可以通过 TAIC 来实现时钟设备的虚拟化。

4.2.3　任务通信

由于任务通信的发送方和接收方是处于不同地址空间和特权级下的任务，而进程可以用于区分地址空间和特权级，且进程属于操作系统，因此使用操作系统标识和进程标识来表示任务所出的环境。因此，TAIC 基于操作系统标识和进程标识，在内部维护了若干的发送能力槽和接收能力槽，用于构建硬件支持的跨越不同环境的任务通信通道。

发送能力槽中记录了当前环境下，可以向哪些接收方 $(recv\_os_i, recv\_proc_j)$ 发起通信以及通信通道的编号 irq_k，而接收能力槽则记录了当前环境下能够接收哪些发送方 $(send\_os_l, send\_proc_m)$ 的通信请求、通信通道编号 irq_n，并且记录了当前环境下因为等待任务通信而进入阻塞状态的任务标识 isr_x。发送能力槽和接收能力槽的结构以及任务通信流程如图 4.4 所示。

在使用硬件支持的任务通信能力时，发送方需要向发送能力槽中写入接收方的操作系统标识和进程标识，以及通信通道的编号，注册成功即为使能了发送能力。接收方需要向接收能力槽中写入发送方的操作系统标识和进程标识、通信通道的编号以及任务标识，注册成功即为使能了接收能力。

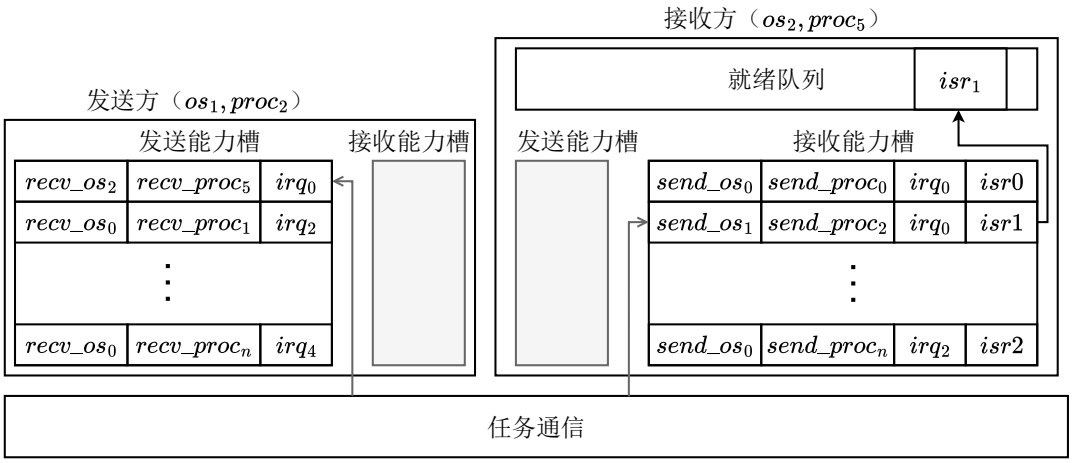


图 4.4　任务通信

发送方通过向 TAIC 的端口中写入接收方的操作系统标识、进程标识以及通信通道编号来发起通信，TAIC 的任务通信模块根据发送方提供的标识信息，首先检查发送方的发送能力槽中是否存在对应的条目，如果不存在，则通信失败；若存在条目，任务通信模块根据发送能力槽条目中记录的接收方的操作系统标识、进程标识以及通信通道标识，检查接收方的接收能力槽中是否存在对应的条目，如果不存在，则通信失败；如果存在，任务通信模块将接收能力槽中对应的条目清除，将等待通信的任务标识放入到就绪队列中的合适位置，完成任务通信。若唤醒的接收方任务的优先级较高，TAIC 会向 CPU 发送中断信号，CPU 在收到中断信号后，直接进行任务切换。

4.2.4　资源分配与回收

每套资源由就绪队列、外部中断能力槽、发送能力槽和接收能力槽组成，向位于特定地址空间和特权级下的软件提供任务调度、中断处理和任务通信功能。由于操作系统与进程可以用于区分地址空间和特权级，因此本文使用操作系统和进程的标识符 $(os\_id, proc\_id)$ 来区分不同的资源，软件要使用这些功能时，需要向 TAIC 的资源分配接口写入操作系统标识和进程标识，TAIC 会根据这些标识符来分配相应的资源，最终返回软件可以操作的句柄。资源分配的流程如下：

1. 向 TAIC 的资源分配接口依次写入 *os_id* 和 *proc_id*。
2. TAIC 根据 *os_id* 和 *proc_id* 查找是否已经分配了对应的资源，若没有，则分配一套资源，将其标记为 $(os\_id, proc\_id)$ 已使用，并在资源初始化之后执行 3；若已经分配，则执行 3；如果控制器中没有空闲资源，则分配失败。
3. 分配软件操作句柄，句柄包括了对这套资源中的就绪队列中的某个局部队列、外部中断能力槽、发送能力槽和接收能力槽的操作接口。

在相同的地址空间和特权级 ($os\_id, proc\_id$) 中，不同的句柄对中断处理、任务通信等接口的操作是等价的，都会对外部中断能力槽、发送能力槽和接收能力槽产生相同的影响，但对就绪队列中的局部队列的操作是不同的，这是为了保证软件在使用这些接口时是一致的。

资源回收的流程如下：

1. 向 TAIC 的资源回收接口依次写入 *os_id* 和 *proc_id*。
2. TAIC 根据 *os_id* 和 *proc_id* 查找是否已经分配了对应的资源，若没有，则回收失败；若已经分配，则执行 3。
3. 回收软件操作句柄，释放软件对该句柄对应的局部队列的操作权限。若这套资源所有申请的操作句柄都被回收时，将资源标记为未使用。

4.3　任务抢占

TAIC 提供了中断处理和任务通信的能力，硬件直接唤醒了对应的阻塞任务，但这不意味着被唤醒的任务会马上执行，若 CPU 上正存在其他的任务执行，而被唤醒的任务优先级较高，则会导致被唤醒任务的响应延时增加。因此，TAIC 会在被唤醒的任务优先级较高时，向 CPU 发送中断，实现了抢占式任务调度。由于被唤醒的任务可能处于内核态或用户态，因此 TAIC 根据被唤醒的任务标识中的操作系统标识和进程标识来判断任务所处的特权级，向 CPU 发送不同的中断信号。因此需要实现用户态中断扩展。本文在 rocket-chip 软核[64] 上增加了用户态中断相关的控制寄存器以及 *uret* 指令，并且将 TAIC 的中断信号与 CPU 的 *mip* 寄存器的 $SSIP$ 和 $USIP$ 位相连，实现了 RISC-V 的 N 扩展。

当被唤醒的任务处于内核态且优先级较高时，TAIC 会产生中断信号，将 *mip* 寄存器的 $SSIP$ 位拉高，CPU 跳转至内核态的中断向量寄存器 *stvec* 指向的位置，保存被打断任务的寄存器现场，从 TAIC 中取出被唤醒的任务标识并执行任务，从而减少响应延时。

当被唤醒的任务处于用户态，这里的处理比唤醒处于内核态的任务更加复杂。可以分为两类：

1. 当 CPU 正在运行与被唤醒任务处于相同进程和特权级下的其他任务，若被唤醒任务的优先级较高，则 TAIC 会产生中断信号，将 *mip* 寄存器的 $USIP$ 位拉高，CPU 跳转至用户态的中断向量寄存器 *utvec* 指向的位置，保存被打断任务的寄存器现场，从 TAIC 中取出被唤醒的任务标识并执行任务，从而减少响应延时。
2. 若当前 CPU 正在执行其他的与被唤醒任务处于不同地址空间或不同特权级

的任务时，TAIC 会暂时挂起这次唤醒所产生的中断，直到 CPU 再次回到 1 这种情况。

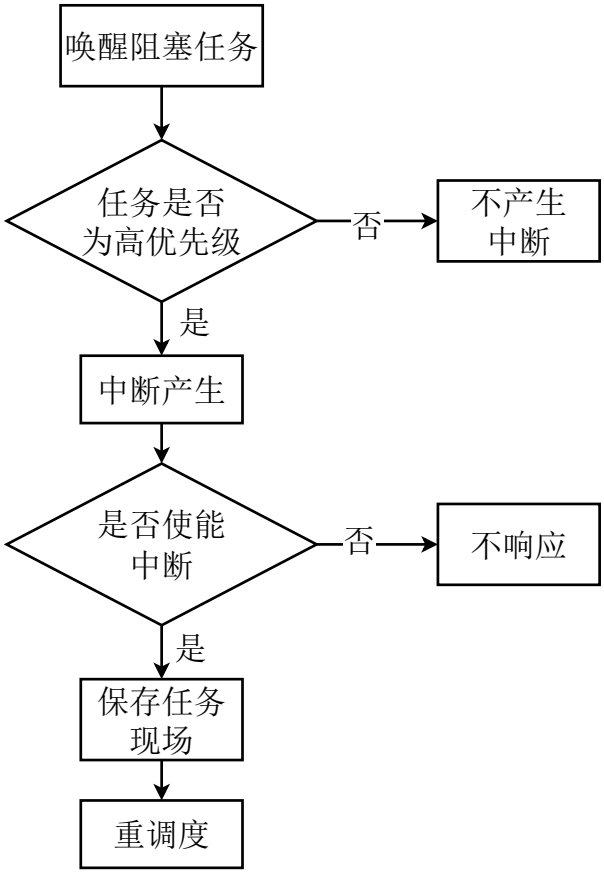


图 4.5　任务抢占的流程

4.4　软件适配

TAIC 可以与现有的进程、线程或协程任务模型进行结合，线程与协程模型建立在进程提供的地址空间隔离机制上，将 TAIC 的端口映射到各自的地址空间中，即可向 TAIC 申请硬件资源用于任务调度、中断处理和任务通信。软件与 TAIC 的交互通过申请获取的硬件资源句柄进行。

4.4.1　申请硬件资源

TAIC 中的硬件资源通过操作系统标识和进程标识进行区分，因此在申请使用 TAIC 的硬件资源时，需要向资源分配端口写操作系统标识 *os_id* 和进程标识 *pid*。在内核中使用 TAIC 时，*os_id* 为一个固定的数值 $OSID$（由内核在初始化时进行

约定），*pid* 为 0，因此，内核向 TAIC 的资源分配接口写入 $(OSID, 0)$ 即可使用 TAIC 提供的功能。在用户态使用时，则需要通过 `get_taic_handle` 系统调用，该系统调用首先向 TAIC 的资源分配端口写入 $(OSID, pid)$ 申请资源句柄，并将资源句柄对应的端口映射到用户进程的地址空间中，最终将资源句柄返回给用户态，对应的伪代码为 4.1。

系统调用 4.1 get_taic_handler

输入:
输出: handler

```
pid ←get_pid()
osid ←OSID
taic_base ←TAIC_BASE
size ←HANDLER_SIZE
handler_id ←taic_alloc(osid, pid)                    ▷ 申请 TAIC 资源，获取资源编号
if handler_id = 0 then
    handler ←−1 return handler
end if
handler_addr ←handler_addr(handler_id,
                           taic_base)   ▷ 根据 TAIC 基址和资源编号，获取资源基址
mem_fd ←open("/dev/mem", O_RDWR)
if mem_fd = −1 then
    taic_free(handler_id)
    handler ←−1 return handler
end if
handler ←mmap(NULL, size,
              PROT_READ | PROT_WRITE,
              MAP_SHARED,
              mem_fd, handler_addr)  ▷ 将 TAIC 资源句柄映射到进程地址空间中
if handler = NULL then
    taic_free(handler_id)
    handler = −1
end ifreturn handler
```

4.4.2　任务通信

当两个进程中的接收方申请到 TAIC 的硬件资源 *handler* 后，可以通过 `task_enqueue(handler, task_id)` 和 `task_dequeue(handler)` 进行任务调度，并且基于 TAIC 直接通信。伪代码 4.2 展示了两个进程单向通信的示例。同理，双向通信只需要两个进程都注册发送和接收能力即可。

当接收方处于用户态且优先级较高时，为了减少任务的响应延时，需要初始化并使能用户态中断。在收到用户态中断时，进行重调度，伪代码如 4.3 所示。

单向通信示例 4.2 communication_example

```
1: send_pid, recv_pid, irq, os_id, is_inited, has_received
2: procedure Sender                                                    ▷ 发送方
3:     send_pid ←sys_get_pid()
4:     send_handler ←get_taic_handler()                                ▷ 获取 TAIC 资源句柄
5:     register_sender(send_handler, os_id, recv_pid, irq)             ▷ 注册发送能力
6:     loop
7:         while is_inited = false do
8:         end while
9:         is_inited ←false
10:        send(send_handler, os_id, recv_pid, irq)                    ▷ 发送方发送通知
11:        while has_received = false do
12:        end while
13:        has_received ←false
14:    end loop
15: end procedure
16: procedure Receiver                                                 ▷ 接收方
17:    recv_pid ←sys_get_pid()
18:    recv_handler ←get_taic_handler()
19:    loop
20:        register_receiver(recv_handler, os_id, send_pid, irq, handle_task) ▷ 注册接收能力
21:        is_inited ←true
22:        task_id ←task_dequeue(recv_handler)
23:        while task_id = 0 do                                        ▷ 等待收到通知
24:            task_id ←task_dequeue(recv_handler)
25:        end while
26:        has_received ←true
27:    end loop
28: end procedure
29: procedure handle_task                                              ▷ 等待通知的任务
30:    xxxx
31: end procedure
```

用户态中断初始化 4.3 user_intr_init

```
1: taic_handler
2: function user_intr_init
3:     csr_write(CSR_UTVEC, uintr_entry)                               ▷ 设置用户态中断入口
4:     csr_set(CSR_USTATUS, USTATUS_UIE)
5:     csr_set(CSR_UIE, MIE_USIE)                                      ▷ 使能用户态中断
6: end function
7: function uintr_entry
8:     xxxx                                                            ▷ 保存被打断任务的上下文
9:     task_id ←task_dequeue(taic_handler)                             ▷ 取出被唤醒的任务标识
10:    switch_to_task(task_id)                                         ▷ 切换到被唤醒的任务
11: end function
```

4.5　本章小结

本章给出了基于软硬协同的中断响应和任务调度系统的整体架构，详细描述了 TAIC 内部各个模块的设计和实现，包括就绪队列、外部中断能力槽、发送能力槽和接收能力槽，解释了它们如何在硬件层面提供软件需要的能力。并且，本章描述了 TAIC 如何与已有的中断机制结合，实现任务抢占。最后给出了如何在现有的软件中使用 TAIC 提供的硬件加速能力的伪代码示例，验证了第三章设计方案的可行性。

第 5 章　性能评估

为了验证基于软硬协同的中断响应与任务调度设计的有效性，并且评估设计中的各个模块对系统性能的影响，本章在 FPGA 平台上围绕以下三个核心维度展开客观评价：

- 任务调度：对基于 TAIC 提供的任务调度功能与传统的软件调度进行多维度对比测试，评估 TAIC 在任务窃取开销等方面的影响；
- 中断处理：评估基于 TAIC 提供的中断处理机制相较于传统中断处理机制对中断延时的影响；
- 任务通信：评估基于 TAIC 提供的任务通信能力与传统任务通信机制之间的性能差异。

5.1　测试环境

性能评估实验在 ALINX AXU15EG 开发板上展开，该开发板以 Xilinx Zynq UltraScale+ XCZU15EG MPSoC 为核心板，并配备了 DDR4、双千兆以太网接口等外部设备。核心板分为两部分，分别为处理器子系统和可编程逻辑（FPGA）。其中处理器子系统集成了四核 ARM® Cortex-A53 处理器、双核 Cortex-R5 实时处理器，可以直接对开发板上的资源进行控制。FPGA 部分实现了带有 RISC-V N 扩展的 rocket-chip 软核。rocket-chip 软核通过 TileLink 协议与 TAIC 进行交互，并且将 TileLink 协议转换成 AXI 协议来访问 DDR4 内存资源。因此，软核访问 TAIC 与内存的开销相当。

表 5.1　测试硬件环境

IP 核	**配置**
RISC-V 软核	rocket chip
	四核，N 扩展
	100MHz 时钟
以太网	TAIC 中断控制器
	Xilinx AXI 1G/2.5G 以太网子系统（1Gbps）
	Xilinx AXI DMA

测试的硬件环境为 FPGA 平台，具体配置如表 5.1。软件运行于 FPGA 中的

RISC-V 软核上，在微基准测试中，首先测试了软硬件交互接口开销，后续使用了 Linux 6.6.0 版本的基于 RISC-V 指令集的内核进行部分模块的单独测试。而综合测试基于 Rust 语言对 Sel4[65] 重写的 Rel4 微内核[66] 展开，测试 TAIC 整体功能对应用程序性能的影响。

5.2　微基准测试

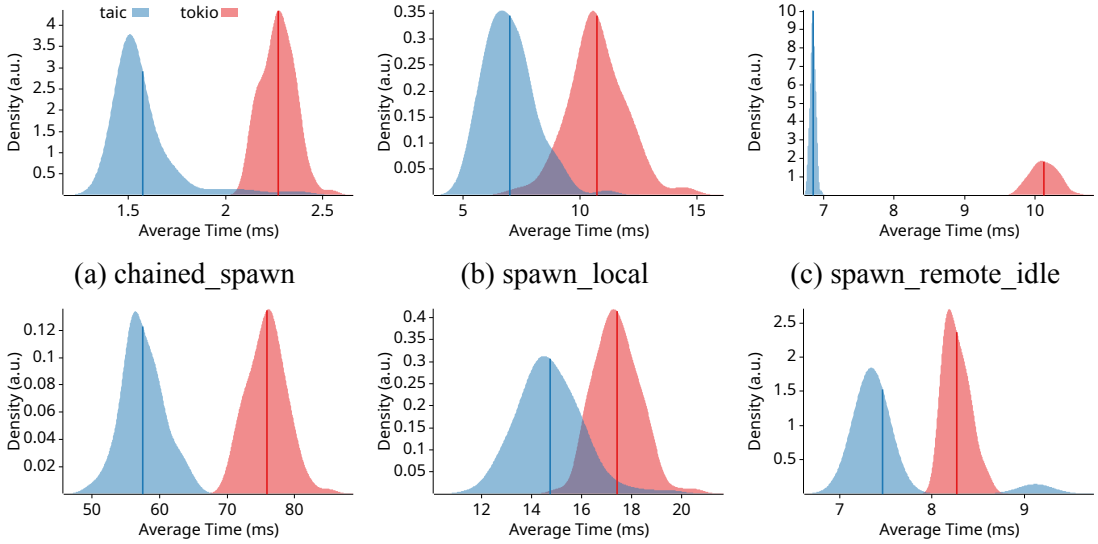
微基准测试主要用于从微观层面评估 TAIC 的性能表现。CPU 与 TAIC 之间的交互在几个时钟周期内完成。其中，申请硬件资源需要 4～6 个时钟周期，与任务调度相关的入队/出队操作在 2～4 个时钟周期内完成，与任务通信相关的注册（注销）发送方/接收方以及发送通知的操作在 8～10 个时钟周期内完成。TAIC 接收到中断信号把处于阻塞队列中的任务标识放入到就绪队列中的过程（中断处理），需要 6～8 个时钟周期。除软硬件交互接口测试外，其余测试在 Linux 6.6.0 系统环境下完成，具体通过 tokio-bench 和 ipc-bench 两个专业工具展开。

5.2.1　任务调度

tokio[67] 作为 Rust 生态中的高性能异步运行时框架，提供了从任务调度到 I/O 操作的全套异步工具链，用于构建高并发、低延时的网络服务。tokio 提供了 rt_multi_threaded 基准测试，用于评估多线程环境下的任务调度性能。tokio 的多线程调度器默认使用了任务窃取算法来动态平衡 CPU 上的任务负载，具体的规则如下：

1. 针对每个工作线程，单独维护一个有界的局部队列；
2. 所有的工作线程共用一个无界的全局队列；
3. 当局部队列达到容量上限时，任务将会被放入全局队列中；
4. 当局部队列没有任务时，工作线程会随机的从其他的工作线程的局部队列中窃取一半的任务；
5. 当工作线程连续 60 次（tokio 的默认配置）从局部队列中取出任务后，将会从全局队列中取出一个任务。

本文对 tokio 默认的多线程调度器进行了修改，将其任务窃取算法中使用的任务队列替换为 TAIC 提供的硬件任务队列。每个工作线程使用 TAIC 提供的硬件局部队列，这些硬件局部队列共同构成一个硬件全局队列，如图 4.2 所示。由于硬件的局部队列没有固定的容量限制，但其最大容量不超过全局队列的容量，因此修改后的 tokio 任务调度器无法满足规则 3，为了保证公平，测试过程中保证了不会出现局部队列溢出的情况。此外，由于硬件提供了局部队列之间的负载均衡功能，



(a) chained_spawn　(b) spawn_local　(c) spawn_remote_idle

(d) yield　(e) ping_pong　(f) spawn_remote_busy

图 5.1　任务调度对比，其中 taic 表示使用硬件任务队列，tokio 表示使用软件任务队列。

当局部队列中没有任务时，会自动从其他的局部队列中取出任务，因此，规则 4 也不再需要，但保留了计算随机数的开销来保证测试的公平性。使用上述修改后的 tokio 调度器来评估 TAIC 在任务调度方面的性能表现，最终的结果如图5.1所示。

任务窃取：chained_spawn 测试递归的生成任务，任意时刻，至多存在两个任务，当一个工作线程运行任务 ***A*** 生成一个新的任务 ***B*** 时，其他的空闲的工作线程会窃取新生成的任务 ***B*** 来执行，循环这个过程，直到结束。因此两个测试的性能差异主要取决于任务窃取的性能。taic 在任务窃取时直接从局部队列中可以窃取出其他的局部队列中的任务，而 tokio 原本的软件任务窃取需要搜索开销以及局部队列的同步互斥等额外开销，从图 5.1(a) 可以看出，taic 使得任务窃取的开销下降了 30.68%。同理，spawn_local 测试生成更多的任务，并将任务放入当前工作线程的局部队列中，尽管 tokio 原本的任务窃取会直接从其他局部队列中窃取一半的任务，但 taic 提供的硬件任务窃取功能几乎是零开销的，因此，taic 在 spawn_local 测试（图5.1(b)）中仍然将任务窃取的开销下降了 34.88%。spawn_remote_idle 测试（图5.1(c)）与 spawn_local 测试类似，不同之处在于它将生成的任务放到了远端队列（远端队列等价于局部队列，用于任务窃取）中，两者的性能差距仍然来源于任务窃取的开销（32.41%）。

任务切换：yield（图 5.1(d)）、ping_pong（图 5.1(e)）和 spawn_remote_busy（图 5.1(f)）测试了在高频任务切换时，硬件任务队列与软件任务队列的性能差异。在 yield 测试中，每个任务在主动让权若干次后结束；在 ping_pong 测试中，由于相互发送消息的两个任务需要等待对方的消息；在 spawn_remote_busy 测试中，用户态任务让权后再次运行时，会使用 `sched_yield` 系统调用让当前的工作线程进入

内核态让权。因此，使用 taic 节省的任务窃取开销被分摊到每次任务切换以及等待的过程中，使用 taic 将这三个测试的开销分别降低了 24.22%、15.37% 和 9.69%。其中 spawn_remote_busy 测试过程中，因为 Linux 内核产生随机数缓慢，导致了使用 taic 时部分测试的平均延时过高。

5.2.2　任务通信

ipc-bench[68] 是一个用于测试 Linux 上的包括信号、管道和 eventfd 等 IPC 机制的开源项目。ipc-bench 针对每种 IPC 机制使用了 ping-pong 通信模型（通信双方忙等对方的消息）来测试其性能，性能指标包括通行的总时间、平均每次通信的时间、吞吐量等。

本文参考 ipc-bench 项目中对信号、管道和 eventfd 等 IPC 机制的测试方法，实现了使用 TAIC 进行任务通信的测试程序。测试程序排除了申请 TAIC 硬件资源以及初始化的时间，测试了通信的总时间。其中，所有测试的消息大小为 1 bit，发送消息次数为 1000 次。各项 IPC 机制的性能对比如图 5.2 和表 5.2 所示。

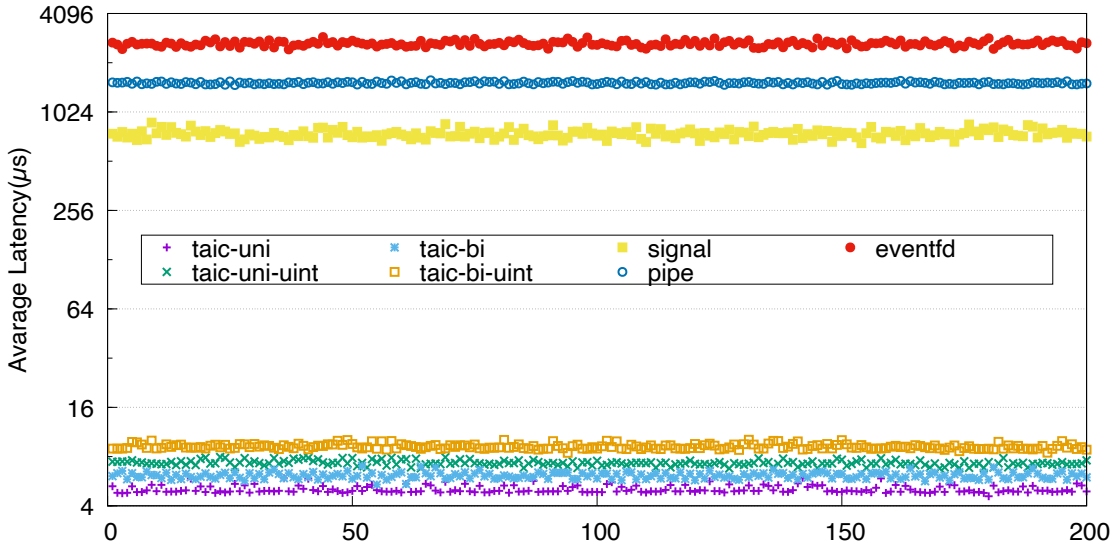


图 5.2　IPC 延迟对比，taic 表示使用硬件任务通信机制的测试。其中，bi 表示双向，双方互相发送消息；uni 表示单向；uint 表示任务优先级高，需要发送中断。eventfd、pipe、signal 均为双向；

与 taic 相关的测试保证了通信的接收方在 CPU 上运行（接收方处于在线状态）。不使用中断时，接收方不断尝试从硬件任务队列中取出被唤醒的任务，一旦取出任务则表示收到通知，结束通信的过程；当使用中断时，接收方收到中断信号后，进入用户态的中断的处理逻辑中，从硬件任务队列中取出被唤醒的任务才算收到通知，但需要从中断返回才结束通信；在双向通信中，接收方收到发送方的通知后，向发送方发起通知，直到发送方收到通知才结束通信的流程。

表 5.2　IPC 性能对比

测试	吞吐量 msg/s	平均延时 μs	相对延迟比
taic-uni	199215.0	5.02	1
taic-bi	163158.4	6.13	1.221
taic-uni-uint	137127.0	7.29	1.453
taic-bi-uint	108734.7	9.20	1.832
signal	1334.7	749.23	149.258
pipe	649.6	1539.41	306.673
eventfd	374.9	2667.38	531.382

无论是否使用中断来进行任务抢占，使用 TAIC 进行任务通信的性能都远远优于其他的 IPC 机制，在性能上有数量级的提升，能够达到微秒级的时间尺度。但在不使用中断的情况下，若接收方进程正在执行其他的任务，TAIC 内部的中断处理和中断的开销会被其他任务执行掩盖掉，若其他任务的执行时间也处于微秒级，那么即使不需要中断机制，同样可以达到微秒级的通信延迟。

在 ipc-bench 测试的基础上，本文进一步测试了接收方处于不同的特权级的中断延迟（中断延迟包括了 TAIC 中断处理开销、保存被打断任务上下文开销以及软件进行中断分发的开销）。具体的结果如表 5.3所示，括号内表示不包括 TAIC 中断处理开销的中断延迟。结果证明了使用 TAIC 不会增加中断延迟，保证被唤醒的任务可以及时抢占。

表 5.3　中断延迟对比

接收方特权级	CPU 周期
用户态	72（66）
内核态	98（92）

以上的测试没有包括极端的情况。一种极端的情况是接收方运行于用户态，但接收方的进程没有任务在用户态运行，此时，即使唤醒的任务的优先级较高，也无法进行任务抢占，需要等到接收方的进程中存在任务回到用户态运行时才能进行抢占，此时，任务通信的延时除了用户态中断的开销外，还包括了接收方进程等待被调度的时间以及从内核态返回用户态的开销。另一种极端的情况是接收方运行于内核态，但此时接收方没有任务处于内核态运行，因此任务抢占会导致一个正在运行用户态任务的 CPU 陷入内核态，因此任务通信的开销还会包括特权级切换开销（在本文的实验环境下，开销为 870 个时钟周期）。

5.3　综合测试

5.3.1　特征分析

5.4　本章小结

第 6 章　总结与展望

参考文献

[1] Adam, et al. IX: A protected dataplane operating system for high throughput and low latency [C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, 2014: 49-65.

[2] Gruss, et al. KASLR is Dead: Long Live KASLR[C]//Bodden E, Payer M, Athanasopoulos E. Engineering Secure Software and Systems. Cham: Springer International Publishing, 2017: 161-176.

[3] Livio, et al. FlexSC: Flexible system call scheduling with Exception-Less system calls[C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). Vancouver, BC: USENIX Association, 2010: 33–46.

[4] David, et al. Context switch overheads for linux on arm platforms[C]//ExpCS '07: Proceedings of the 2007 Workshop on Experimental Computer Science. New York, NY, USA: Association for Computing Machinery, 2007: 3–es.

[5] Tsafrir, et al. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)[C]//ExpCS '07: Proceedings of the 2007 Workshop on Experimental Computer Science. New York, NY, USA: Association for Computing Machinery, 2007: 4–es.

[6] Intel. Intel® Optane™ SSD DC P4800X Series (375GB, 2.5in PCIe x4, 3D XPoint™) Product Specifications[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint.html.

[7] Digital W. How to Read the Ultrastar Model Number[EB/OL]. 2024. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-nvme-series/data-sheet-ultrastar-dc-sn200.pdf.

[8] Intel. Intel® 82598EB 10 Gigabit Ethernet Controller[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/www/us/en/products/sku/36918/intel-82598eb-10-gigabit-ethernet-controller/specifications.html.

[9] Caulfield A M, De A, Coburn J, et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories[C/OL]//2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. 2010: 385-395. DOI: 10.1109/MICRO.2010.33.

[10] Yang J, Minturn D B, Hady F T. When poll is better than interrupt[C/OL]//10th USENIX Conference on File and Storage Technologies (FAST 12). San Jose, CA: USENIX Association, 2012. https://www.usenix.org/conference/fast12/when-poll-better-interrupt.

[11] Vučinić D, Wang Q, Guyot C, et al. DC express: Shortest latency protocol for reading phase change memory over PCI express[C/OL]//12th USENIX Conference on File and Storage Technologies (FAST 14). Santa Clara, CA: USENIX Association, 2014: 309-315. https://www.usenix.org/conference/fast14/technical-sessions/presentation/vucinic.

[12] corbet. Driver porting: Network drivers[EB/OL]. 2024[2024-05-12]. https://lwn.net/articles/30107/.

[13] Belay A, Prekas G, Klimovic A, et al. IX: A protected dataplane operating system for high throughput and low latency[C/OL]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, 2014: 49-65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[14] DPDK. Data plane development kit.[EB/OL]. 2024[2024-05-12]. https://www.dpdk.org/.

[15] SPDK. Storage Performance Development Kit[EB/OL]. 2024[2024-05-12]. https://spdk.io/.

[16] Intel. Intel data direct I/O technology (Intel DDIO): A primer.[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[17] Nayak R J, Chavda J B. Comparison of Accelerator Coherency Port (ACP) and High Performance Port (HP) for Data Transfer in DDR Memory Using Xilinx ZYNQ SoC[C]//Satapathy S C, Joshi A. Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 1. Cham: Springer International Publishing, 2018: 94-102.

[18] Chen X, Yu G, Cheng H. Approach to external events of real-time operating system based on polling[C/OL]//2010 Second International Conference on Computer Modeling and Simulation: Vol. 1. 2010: 459-462. DOI: 10.1109/ICCMS.2010.312.

[19] Guan H, Dong Y, Tian K, et al. Sr-iov based network interrupt-free virtualization with event based polling[J/OL]. IEEE Journal on Selected Areas in Communications, 2013, 31(12): 2596-2609. DOI: 10.1109/JSAC.2013.131202.

[20] Le Moal D. I/o latency optimization with polling[C]//Vault Linux Storage and Filesystems Conference. 2017.

[21] Stephen Bates H. Linux Optimizations for Low Latency Block Devices | SNIA[EB/OL]. 2017 [2024-05-12]. https://www.snia.org/educational-library/linux-optimizations-low-latency-block-devices-2017.

[22] AlQahtani S A. A novel hybrid scheme of interrupt enabling - disabling and polling (edp) for high-speed computer networks[C/OL]//2007 International Symposium on Communications and Information Technologies. 2007: 341-345. DOI: 10.1109/ISCIT.2007.4392042.

[23] Gao G, Hum H, Theobald K, et al. Polling watchdog: Combining polling and interrupts for efficient message handling[C/OL]//23rd Annual International Symposium on Computer Architecture (ISCA'96). 1996: 179-179. DOI: 10.1145/232973.232992.

[24] Lee G, Shin S, Jeong J. Efficient hybrid polling for ultra-low latency storage devices[J/OL]. J. Syst. Archit., 2022, 122(C). https://doi.org/10.1016/j.sysarc.2021.102338.

[25] Langendoen K, Romein J, Bhoedjang R, et al. Integrating polling, interrupts, and thread management[C/OL]//Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96). 1996: 13-22. DOI: 10.1109/FMPC.1996.558057.

[26] Yamasaki N. Responsive multithreaded processor for distributed real-time processing[C/OL]//International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems (IWIA'06). 2006: 44-56. DOI: 10.1109/IWIAS.2006.36.

[27] Suito K, Fujii K, Matsutani H, et al. Dependable responsive multithreaded processor for distributed real-time systems[C/OL]//2012 IEEE COOL Chips XV. 2012: 1-3. DOI: 10.1109/COOLChips.2012.6216589.

[28] Wada R, Yamasaki N. Fast interrupt handling scheme by using interrupt wake-up mechanism [C/OL]//2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW). 2019: 109-114. DOI: 10.1109/CANDARW.2019.00027.

[29] Lopez T A, Yamasaki N. Prioritized asynchronous calls for parallel processing on responsive multithreaded processor[C/OL]//2022 Tenth International Symposium on Computing and Networking (CANDAR). 2022: 46-55. DOI: 10.1109/CANDAR57322.2022.00014.

[30] Dodiu E, Gaitan V, Graur A. Custom designed cpu architecture based on a hardware scheduler and independent pipeline registers - architecture description[C]//2012 Proceedings of the 35th International Convention MIPRO. 2012: 859-864.

[31] Gaitan N C, Gaitan V G, Moisuc E E C. Improving interrupt handling in the nmpra[C/OL]//2014 International Conference on Development and Application Systems (DAS). 2014: 11-15. DOI: 10.1109/DAAS.2014.6842419.

[32] ZAGAN I, GAITAN N C, GAITAN V G. An approach of nmpra architecture using hardware implemented support for event prioritization and treating[J/OL]. International Journal of Advanced Computer Science and Applications, 2017, 8(2). http://dx.doi.org/10.14569/IJACSA.2017.080206.

[33] Salah K. To coalesce or not to coalesce[J/OL]. AEU - International Journal of Electronics and Communications, 2007, 61(4): 215-225. https://www.sciencedirect.com/science/article/pii/S1434841106000063X. DOI: https://doi.org/10.1016/j.aeue.2006.04.007.

[34] Ahmad I, Gulati A, Mashtizadeh A, et al. Improving performance with interrupt coalescing for virtual machine disk io in vmware esx server[J]. VMware Inc., Palo Alto, CA, 2009, 94304.

[35] Ahmad I, Gulati A, Mashtizadeh A J. vic: Interrupt coalescing for virtual machine storage device io[C/OL]//USENIX Annual Technical Conference. 2011. https://api.semanticscholar.org/CorpusID:16541915.

[36] Dong Y, Xu D, Zhang Y, et al. Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling[C/OL]//2011 IEEE International Conference on Cluster Computing. 2011: 26-34. DOI: 10.1109/CLUSTER.2011.12.

[37] Guan H, Dong Y, Ma R, et al. Performance enhancement for network i/o virtualization with efficient interrupt coalescing and virtual receive-side scaling[J/OL]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(6): 1118-1128. DOI: 10.1109/TPDS.2012.339.

[38] Tai A, et al. Optimizing storage performance with calibrated interrupts[C]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). New York, NY, USA: Association for Computing Machinery, 2021: 129-145.

[39] Gomes T, Garcia P, Salgado F, et al. Task-aware interrupt controller: Priority space unification in real-time systems[J/OL]. IEEE Embedded Systems Letters, 2015, 7(1): 27-30. DOI: 10.1109/LES.2015.2397604.

[40] Erwin J D, Jensen E D. Interrupt processing with queued content-addressable memories[C/OL]//AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, Fall Joint Computer Conference. New York, NY, USA: Association for Computing Machinery, 1970: 621–627. https://doi.org/10.1145/1478462.1478553.

[41] Scheler F, Hofer W, Oechslein B, et al. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system[C/OL]//Henkel J, Parameswaran S. Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2009, Grenoble, France, October 11-16, 2009. ACM, 2009: 167-174. https://doi.org/10.1145/1629395.1629419.

[42] Wierman A, Zwart B. Is tail-optimal scheduling possible?[J]. Operations research, 2012, 60 (5): 1249-1257.

[43] Prekas G, Kogias M, Bugnion E. Zygos: Achieving low tail latency for microsecond-scale networked tasks[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 325-341.

[44] Iyer R, Unal M, Kogias M, et al. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling[C]//Proceedings of the 29th Symposium on Operating Systems Principles. 2023: 466-481.

[45] Kaffes K, Chong T, Humphries J T, et al. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 345-360.

[46] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 361-378.

[47] Jia Y, Tian K, You Y, et al. Skyloft: A general high-efficient scheduling framework in user space[C]//Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. 2024: 265-279.

[48] Mehta S. x86 User Interrupts support[EB/OL]. 2021[2024-09-02]. https://lwn.net/Articles/869140/.

[49] Luo X, Shen W, Mu S, et al. DepFast: Orchestrating code of quorum systems[C/OL]//2022 USENIX Annual Technical Conference (USENIX ATC 22). Carlsbad, CA: USENIX Association, 2022: 557-574. https://www.usenix.org/conference/atc22/presentation/luo.

[50] von Behren R, Condit J, Zhou F, et al. Capriccio: scalable threads for internet services[C/OL]// SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2003: 268–281. https://doi.org/10.1145/945445.945471.

[51] Zhang I, Raybuck A, Patel P, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 195-211.

[52] Embassy. Modern embedded framework, using Rust and async.[EB/OL]. 2023[2024-01-04]. https://github.com/embassy-rs/embassy.

[53] Dion. Async Rust vs RTOS showdown! - Blog - Tweede golf[EB/OL]. 2022[2025-03-04]. https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown.

[54] Wikipedia. 冯诺伊曼结构[EB/OL]. 2024[2025-03-05]. https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[55] Saltzer J H. Traffic control in a multiplexed computer system[D]. Massachusetts Institute of Technology, 1966.

[56] Dijkstra E W. The structure of the “the”-multiprogramming system[J]. Communications of the ACM, 1968, 11(5): 341-346.

[57] Ritchie D M, Thompson K. The unix time-sharing system[J]. Communications of the ACM, 1974, 17(7): 365-375.

[58] Marlin C D. Coroutines: a programming methodology, a language design and an implementation: No. 95[M]. Springer Science & Business Media, 1980.

[59] Moura A L D, Ierusalimschy R. Revisiting coroutines[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2009, 31(2): 1-31.

[60] 葛文博. 众核环境下嵌入式实时操作系统中调度处理的研究[D]. 中国电子科技集团公司电子科学研究院, 2023.

[61] Lin J, Chen Y, Gao S, et al. Fast core scheduling with userspace process abstraction[C/OL]// SOSP '24: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2024: 280 – 295. https://doi.org/10.1145/3694715.3695976.

[62] RISC-V. RISC-V Ratified Specifications[EB/OL]. 2025[2025-03-06]. https://riscv.org/specifications/ratified/.

[63] Pinto S, Garlati C. User mode interrupts[Z]. 2019.

[64] chipsalliance. Rocket Chip Generator[EB/OL]. [2025-03-12]. https://github.com/chipsalliance/rocket-chip.

[65] seL4. The sel4 microkernel[EB/OL]. [2025-03-13]. https://github.com/seL4/seL4.

[66] rel4team. rel4_kernel, the rust version of sel4[EB/OL]. [2025-03-13]. https://github.com/rel4team/rel4_kernel.

[67] tokio rs. Tokio - An asynchronous Rust runtime[EB/OL]. [2025-03-13]. https://tokio.rs/.

[68] Goldsborough P. ipc-bench[EB/OL]. 2025[2025-03-13]. https://github.com/goldsborough/ipc-bench.

致　谢

衷心感谢我的导师向勇老师对本人的精心指导，他的言传身教将使我终身受益。

感谢实验室的同学廖东海，以及各位学弟。

感谢清华大学的各位老师，以及为我提供帮助的同学。

感谢我的母亲，因为她的支持和鼓励，我能够顺利完成学业。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，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享有著作权的内容。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。

签　名：_____ 日　期：_____

个人简历、在学期间完成的相关学术成果

个人简历

1999 年 01 月 09 日出生于湖南湘潭县。

2022 年 9 月考入清华大学计算机系攻读计算机科学与技术专业硕士至今。

在学期间完成的相关学术成果

学术论文：

[1] Zhao F, Liao D, Wu J, et al. Cops: A coroutine-based priority scheduling framework perceived by the operating system[C]//2024 International Conference on Computer and Information Technology. 2024.

指导教师评语

论文提出了……

答辩委员会决议书

论文提出了……

论文取得的主要创新性成果包括：

1. ……

2. ……

3. ……

论文工作表明作者在 ××××× 具有 ××××× 知识，具有 ×××× 能力，论文 ××××，答辩 ××××。

答辩委员会表决，（× 票/一致）同意通过论文答辩，并建议授予 ×××（姓名）×××（门类）学博士/硕士学位。