在 QEMU 模拟器中的共享调度器设计与实现

摘　要

协程是轻量级的任务调度单位，被广泛用于高并发场景中。但使用软件调度大量协程可能产生较高的调度开销。并且，Rust 语言中的协程只能在特定执行位置让出 CPU 资源，与可能随时打断程序执行的传统中断处理机制存在不适配。为了解决这些问题，本文在 QEMU 模拟器中实现了 QEMU-ATS-INTC，一个支持任务感知的中断控制器的硬件模拟，使得其同时具备了中断处理和协程调度的功能。本文在 ArceOS——一个基于 Rust 语言的操作系统中设计并实现了基于 QEMU-ATS-INTC 的线程和协程调度器，并使用该调度器提供的协程和中断支持，优化 ArceOS 的网络模块，从而得到了 ArceOS-ATS-INTC 系统。本文将修改后的软硬件系统——ArceOS-QEMU-ATS-INTC 与原系统进行对比实验，证明了 ArceOS-QEMU-ATS-INTC 的任务调度开销更低，且在高并发情况下的网络性能更好。

关键词：任务调度；中断处理；协程；QEMU 模拟器；Rust 语言

Design and Implementation of a Shared Scheduler in QEMU Emulator

Abstract

Coroutine is a light-weight task unit for scheduling and is widely used in high concurrency situations. However, scheduling large amounts of coroutines with software may introduce high scheduling overhead. Moreover, coroutines in the Rust language can only yield at certain points, which do not fit with traditional interrupt mechanisms that can interrupt tasks' execution anytime and anywhere. To solve these problems, we implement a task-aware interrupt controller, QEMU-ATS-INTC, in the QEMU, which handles the external interrupts and schedules the coroutines cooperatively. Using this simulated hardware, we design and implement a task scheduler for both threads and coroutines in ArceOS, a operating system written in Rust, and optimize the network module of this system with features provided by our scheduler. This modified operating system is called ArceOS-ATS-INTC. Finally, we conduct several comparative experiments between ArceOS-QEMU-ATS-INTC, the system we develop and the original system, which shows that ArceOS-QEMU-ATS-INTC has less overhead on task scheduling and high network performance at high concurrency situations.

Key Words: Task Scheduling；Interupt Handling；Coroutine；QEMU；The Rust Language

目 录

第 1 章 绪论

1.1 研究背景和意义

在操作系统的发展历程中，任务调度一直是一个重要的课题。在多个任务需要处理时，选择合适的处理顺序、实现高效的任务切换机制，可以提高计算机系统处理任务的性能。

随着计算机技术的发展，任务调度机制的发展呈现出以下几个趋势：一是适应硬件技术的进步，例如从单核任务调度到多核任务调度、从同构多核调度到异构多核调度；二是调度粒度愈发精细，例如调度的对象从进程到线程，之后又出现了比操作系统线程更精简的调度对象，称为协程/轻量级线程/用户级线程（下文中统称协程）。三是调度机制的目标更加专门化，例如应用于嵌入式实时系统的任务调度机制需要尽量缩短任务的反应时间，而应用于高性能服务器的调度机制则要求高吞吐量和低时延。

这其中，以协程为单位的调度往往出现在进程和线程都无法满足性能要求的高并发情景，具有比较苛刻的性能要求。同时因为其对任务的划分比较精细，通常具有较高的调度频率。因此，如果能减小每次调度时，任务选择和切换的开销，可以明显提升系统的性能。本课题的目标是实现一种基于硬件的协程调度器，它在保留了中心化调度的优势的前提下，一方面可以取消原有的调度器线程，使更多线程用于处理任务；另一方面可以利用硬件来加速任务的调度，从而提升高频率协程调度场景下的任务处理性能。

中断是许多实时、分时系统提供的机制，其可以使操作系统感知时间的变化并做出反应（时钟中断），可以使操作系统及时响应外部设备的事件（外部中断），也可以作为多个处理器核心间通信的手段（核间中断）。不过，中断机制与协程调度并不完全适配，其原因是：目前的协程调度机制大多在用户态实现，无法被操作系统感知。而中断的接收和处理都由操作系统完成。在中断处理的过程中，任务以操作系统线程的方式被打断和恢复，切换开销与线程相近，而远高于协程。本课题将协程调度引入操作系统，将其与中断处理更紧密地结合，使得中断处理过程中的上下文切换也可以享受协程机制带来的低开销。

Rust语言是一门适合系统编程的新兴编程语言，其拥有高于C语言的内存安全性、

更现代的语法特性和等同C语言的性能。它从语言层面上支持协程，大大方便了协程相关代码的编写。RISC-V指令集则是一套开源、简洁、模块化的指令集。因此，基于Rust语言和RISC-V指令集的操作系统是操作系统开发的新兴方向之一。这一方向上已经出现了一些教学和科研目的的操作系统，例如rCore教学操作系统[1]。因此，本课题以Rust语言和RISC-V指令集作为软硬件平台，可以丰富这一方向的软硬件生态。

1.2 国内外研究现状

本文的研究基于已有的两个研究方向：一是对任务调度机制的研究，二是对中断机制的研究。

1.2.1 任务调度机制

好的任务调度器，需要同时满足两个要求：科学的调度策略和低开销的实现机制[2]。因此，调度策略和实现机制就称为了研究任务调度的两大方向。由于本研究注重实现机制方向，因此也重点介绍实现机制方面的研究概况。

在国际上，调度器的实现机制主要有以下研究和改进：George Prekas等人[3]设计了ZygOS系统，专门优化了调度机制与网络协议栈的实现，以达到更高的网络性能。其将网络协议栈与应用程序的执行流分离，通过队列相互通信。通过该方式，空闲的应用程序可以“窃取”其它CPU核心队列中的数据包并处理，使各个核心间的负载更均衡。Kostis Kaffes等人[2]设计了Shinjuku系统，实现了一个以微秒级调度协程的调度器。因为操作系统提供的机制在微秒时间尺度下开销太大，该调度器实现了自己的抢占机制和上下文切换机制。其使用中心化调度，由一个调度线程和多个工作线程组成。中心化调度的优点是，调度器线程知道整个调度系统的信息，因此可以采用更高效的调度算法；并且，不同核心之间的负载均衡也更简单。但该方法的不足在于，设置了一个调度器线程也导致真正进行任务处理的线程比系统使用的线程少了一个。Rishabh Iyer等人[4]在这个思路上改进：一方面，采用协作式调度代替和近似抢占式调度，并使用编译器代码插桩的方式强制应用程序主动让权，在降低上下文切换开销的同时，避免了协作式调度中，占据CPU时间过长的应用程序无法被抢占的问题；另一方面，设置二级队列，在调度器线程的全局队列外设置每个CPU核心的局部队列，降低了工作线程等待调度器线程分配任务的时间。它同时让调度线程在空闲时也处理任务，降低了使用调度线程对效率的影响。该方法的优势为：在中心化调度的基础上吸收了非中心化调度的部分特点，以弥补中心化调度的不足，例如其它线程需要等待

调度线程分配任务，以及调度线程会占据一个CPU核心。然而，由于本质仍是中心化调度，因此该方法无法将这些不足完全解决。

在国内，主要有以下研究和改进：曾素华等人[5]在OSEK操作系统中，证明当低优先级的任务被高优先级的基本任务抢占时，两个任务的执行时序相当于串行执行。因此，他们将这种情况下的抢占转化为函数调用，从而减少上下文切换的开销。钱宏文等人[6]引入FPGA硬件辅助CPU计算，一方面发挥FPGA的可重构优势，将系统的FPGA资源划分为不同的资源块，从而支持多个进程的并行运行；另一方面改进进程调度机制，将一部分进程分配到FPGA上运行。其具有与软件调度统一的接口，使得用户程序不需修改即可在该系统上运行。该方法创新之处在于使用FPGA作为协处理器，研究和实现了将进程在CPU和协处理器上调度的策略；不足在于将进程传递到FPGA需要进行bit流形式的传输，可能影响效率。尹震宇等人[7]使用硬件实现线程的切换，并为此建立数学模型，设计并实现了一种更适合硬件线程的DR-EDF调度算法。在该方法中，线程的调度和切换过程均可以在硬件中实现，大幅降低了线程切换的开销，因此可以通过更频繁的调度达到更短的响应时间。不过，协程这一更细粒度、更轻量的调度单位比线程更能利用硬件机制提供的低调度开销和高调度频率。

总体而言，以往对任务调度的实现机制的研究基本上从这些方向展开：一部分研究从软件层面优化软件实现的中心化任务调度，从而提高了高并发情况下的性能表现。不过，该方向的不足在于：软件实现中心化调度需要在CPU上运行专门的调度器线程，从而挤占了任务的执行时间。而本文的研究通过（QEMU模拟的）硬件实现任务调度，从而解决了这一不足。另一部分研究在硬件上实现进程或线程的调度，从而降低了调度开销。但其不足在于：由于进程和线程的切换开销较大，硬件降低的调度开销对系统总体性能的提升不明显。本文的研究通过选用切换开销更小的协程作为硬件任务调度的单位解决了这一不足。

1.2.2 中断机制

如前文所述，时钟中断、外部中断、核间中断这三种中断里，本课题主要关注外部中断。外部中断往往会先由特定的中断芯片接收和汇总，再发往CPU进行处理。已有许多研究通过设计自定义的中断芯片，以改进外部中断的处理过程，从而优化特定指标上的性能。本课题的研究也基于这一方向。

在国际上，主要有以下研究改进：Jerry D. Erwin等人的研究[8]使中断芯片可以动

态改变各个中断源的优先级以适应外部环境的变化。其将到来的中断放入优先级队列，并只在队列存在优先级比CPU当前处理的优先级更高的中断时，将对应的中断通知CPU。不过，该设计以一组优先级决定中断接受和加入队列的顺序，以另一组优先级决定中断通知CPU的顺序，分裂的优先级设置可能导致高优先级的中断因为入队优先级低而无法及时处理。Jupyung Lee等人[9]改进了中断处理机制：一方面将中断处理函数的执行移到目标进程中，减少了上下文切换的次数；另一方面，建立中断与进程间的关联，从而将最短的中断处理路径留给优先级最高的中断-进程。通过这些机制，降低了高优先级任务的中断开销。Fabian Scheler等人[10]则为中断芯片内部、中断芯片和CPU之间设置了统一的优先级，解决了高优先级的任务会被低优先级的中断处理程序抢占（优先级倒置）的问题。它在中断到来时，直接将对应的处理程序放入内存中的调度器数据结构，而只在中断优先级高于CPU运行的任务优先级时，才会打断CPU当前的任务。这一思路将中断处理的上下文切换转化为普通任务的上下文切换，对中断与协程调度的融合具有指导意义：如果将中断处理程序与一般任务都实现为协程，就能以协程的切换开销实现中断处理的上下文切换。

在国内，主要有以下研究改进：杨媛媛等人[11]在软件层面设置处理队列并按先入先出顺序处理其中的任务，从而精简中断处理程序，使其不需执行真正的中断处理逻辑，而只需将处理逻辑所需参数加入处理队列即可退出。中断处理程序的精简可以减少中断嵌套的发生，缓解其带来的处理超时、资源抢占等问题。然而，如果先到来的低优先级中断已经入队，后到来的高优先级中断就无法抢占，只能按照队列的先入先出顺序，等待低优先级中断对应的任务执行完成后再执行，从而延长了高优先级任务的响应时间。舒生亮等人[12]为Matrix DSP平台设计和实现了专用的中断处理器，其可以接收各种来源的中断并为它们分配优先级，使CPU优先处理优先级最高的、使能的中断。其设计专门适配了平台CPU的流水线机制，使得进入中断处理例程的延时开销降低到仅4个CPU周期。张旭等人[13]在Linux操作系统上实现了一个用户态任务调度框架，其在用户态实现了任务调度、高精度时钟和中断处理的功能，并且性能均优于Linux内核的实现。任务调度方面，该系统申请的处理器线程中，一个线程用于运行定时器，其余线程均用于运行任务，且各个线程具有独立的任务调度数据结构。系统使用位图算法进行高性能的任务调度。中断处理方面，该系统在内核的中断处理程序中构建快速跳转栈，实现中断在用户态的调度框架内的处理。进入用户态后，再使用软中断机制，将较长的处理逻辑作为任务创建，保证中断处理函数的快速退出。该

研究实现了一个功能较为完整的调度系统，对本文的研究有重要的参考价值。与其基于Linux系统相比，本文的研究基于结构更加简单的软硬件系统，因此可以进一步简化部分机制的实现。

总体而言，对优化中断机制的已有研究主要有以下方向：一部分研究优化中断芯片接收中断的机制，包括设置中断源优先级、存储中断、通知CPU等机制。这些研究的优点在于解决了中断导致的优先级倒置问题，也一定程度上实现了中断处理和任务调度的结合。另一部分研究优化CPU对中断的响应机制，例如通过软中断机制将实际的中断处理过程作为普通任务创建。它们减少了中断处理程序本身的开销，也减少了中断嵌套等复杂情况的发生。但这些改进的不足在于，中断处理程序仍会打断当前任务，引入切换开销的同时，也不适配协作式任务调度的场景。而本文的研究通过将中断处理例程直接放入调度队列的方式解决了这一问题。

1.3 研究内容和关键问题

1.3.1 研究主要内容

本研究分为原型设计和对比测试两部分。首先在QEMU模拟器中实现支持任务感知的中断控制器的硬件模拟QEMU-ATS-INTC，并验证其正确性；之后在ArceOS，一个基于Rust语言、支持RISC-V指令集的操作系统中，添加基于QEMU-ATS-INTC的协程支持和外部中断支持，实现异步任务调度和网络性能优化，得到修改后的操作系统ArceOS-ATS-INTC；最后，将其和现有的任务调度机制进行性能测试，对比分析测试结果。

本研究使用QEMU模拟器实现一个模拟硬件设备，其具有中断处理和任务调度两个功能。它可以接收外部中断，并将中断向量表中对应的中断处理协程作为一个高优先级任务加入调度器中进行调度，以实现异步处理中断的效果，减少因中断处理而产生的切换开销。它维护一组不同优先级的先入先出队列，以实现协程调度的功能。软件向其中放入就绪的协程，并从中取出下一个运行的协程，其为最高优先级的非空队列中的最早加入的协程。

本研究修改了ArceOS的任务调度模块，使其能够利用本项目实现的硬件设备进行任务调度和中断处理。修改后的任务调度模块可以统一地调度线程与协程，从而既能利用协程的低切换开销，又能兼容ArceOS原有的设计，使得适用于该系统的应用程序不加修改即可运行。其能以线程或协程的形式注册外部中断处理例程，从而在外

部中断到来时，加入调度器中运行。

本研究在此基础上修改ArceOS的网络模块，利用新增的协程调度功能和外部中断机制，优化数据包发送和接收的处理流程。其采用两种优化方式，并比较优化的效果：一是创建专门的轮询任务，不断轮询网卡和网络协议栈；二是在接收到中断时，才调用一次相应的轮询函数。

本研究之后对该软硬件系统——ArceOS-QEMU-ATS-INTC进行性能测试，同时与未修改的，在QEMU上运行的ArceOS进行比较。测试结果表明，在低负载情况下，ArceOS-QEMU-ATS-INTC与原系统性能相差不大；在高负载情况下，ArceOS-QEMU-ATS-INTC优于原系统。

1.3.2 关键问题

本项研究的协程调度器具有中断处理和协程调度两个功能，其分别解决两方面的关键问题：

在中断处理方面，传统的中断处理机制需要抢占处理器现有的执行流，这会带来上下文保存和恢复的较大时间开销。本项研究通过将外部中断的处理转化为协程调度，可以利用协程机制降低上下文切换的开销，提高任务处理效率。该方法可以运用在一些对实时性要求不高的外部中断上。

在协程调度方面，协程调度频率较高，软件实现的调度器会产生较大的开销。同时，软件实现的调度机制需要一个线程来运行调度器本身，这减少了可以处理工作负载的线程数。本研究采用硬件实现调度器的方案来解决这一关键问题，尽量降低任务切换和中断处理的CPU资源开销和时间开销，同时使更大比例的CPU资源用于处理工作负载。

1.4 全文组织结构

本文的第1章介绍简要介绍了研究背景和国内外研究现状，之后引出本文研究的内容，以及解决的关键问题。第2章对本项研究涉及的技术做简单介绍，包括任务调度的机制和分类，以及协程在Rust语言中的实现。

本文的第3章描述了本研究实现的ArceOS-QEMU-ATS-INTC系统的设计，包括总体架构、QEMU-ATS-INTC的设计、ArceOS-ATS-INTC的设计三个部分。第4章介绍QEMU模拟器与本项研究相关的内容，并详细说明了本项研究的模拟硬件部分在QEMU模拟器中的实现。第5章介绍ArceOS操作系统与本项研究相关的内容，并详细

说明了本项研究的软件部分在该操作系统中的实现。

本文的第6章描述本项研究中开展的实验评估。其介绍了实验环境和各个实验的设计。之后，给出了各个实验的结果，并分析导致该结果的原因。

第 2 章 相关技术介绍

2.1 任务调度策略

2.1.1 概述

在支持多任务处理的计算机系统中，任务调度机制负责维护各个任务的运行状态（如就绪、运行、阻塞、终止等），并根据任务的状态和其它信息（例如优先级），将处理器的计算时间和计算资源按一定规则分配给这些任务，最终使每个任务（或尽可能多的任务）都执行完成。

一般来说，任务调度机制可以分为两个部分：算法部分和实现部分。任务调度算法即是将处理器资源分配给任务的规则，其分为先来先服务算法（FCFS）、轮转调度算法（RR）、基于优先级的调度算法，等等。不同的调度算法适应不同场景的使用。实现部分首先需要建立表示任务的数据结构，并且在此基础上为任务实现运行、停止、切换等操作。此外，还需实现存放任务的数据结构，例如就绪队列、阻塞队列、终止队列等。最后，实现选用的任务调度算法，从而确定触发重新调度的条件和下一个运行任务的选择标准。

从任务调度的角度改进系统的性能，有两个改进方向，对应了任务调度机制的两个部分：其一是采用更合适的任务调度算法，使各个任务以更符合性能要求的时序运行。其二是降低任务调度机制本身的运行开销。这既包括运行任务调度算法，以获取下一个运行的任务的开销，也包括任务切换产生的开销。本文所述的研究主要围绕降低任务调度机制的运行开销展开。

2.1.2 性能指标

不同的使用场景，可能对任务调度机制提出不同性能指标上的要求。广泛用于衡量任务调度性能的指标主要有以下几种[14]：

- 响应时间/时延：指任务从生成到运行完成需要的时间，一般包括任务的排队时间和运行时间。响应时间会直接影响系统的用户体验，因此对任务调度而言是重要的指标。由于系统负载等状态的变化，以及中断、抢占等随机出现的事件，调度系统的响应时间也可能出现波动。有时候不仅关注平均响应时间，也关注较坏/最坏情况下的反应时间，称为尾部时延。
- 吞吐量：指单位时间内系统能够处理的任务量。其反映了系统处理任务的能力。

在批处理系统等对于时延无要求/低要求的场景中，吞吐量是反映系统处理速度的最重要的性能指标。但即使是对时延敏感的应用场景，吞吐量也是重要的指标。

- CPU利用率：指CPU忙碌的时间占总时间的比值。该指标可以衡量调度系统是否充分利用了CPU的资源。由于大部分调度器都能充分利用CPU资源，因此又提出了更严格的利用率标准：CPU运行被调度任务的时间占总时间的比值。为了提高这一指标，需要任务调度机制的实现开销尽可能降低。

一般情况下，这几项指标的改善具有相关性，即一项指标的改善有较大可能也会改善其它的指标。然而，随着调度器应用场景的逐渐专门化和优化空间的缩小，现在的调度机制研究会专门优化一项或几项指标。例如[2]的研究针对尾部时延进行了优化，而[4]的研究在尾部时延达到要求的条件下提升系统的吞吐量。

本文的研究通过降低任务调度机制的运行开销，理论上可以降低时延，提高吞吐量和CPU利用率。本研究设计的实验会测量系统的时延和吞吐量，从而证明本项研究对系统性能的优化，并测量优化程度。

2.1.3 协作式调度和抢占式调度

根据系统是否能强制任务让出CPU资源，可以将调度机制分为协作式调度和抢占式调度。

协作式调度中，任务只能通过从运行状态进入就绪/阻塞/终止状态主动让出CPU资源，而调度系统无法强制任务让出。这种调度方式的性能依赖任务自己的行为，如果一个任务长时间占用CPU而不让出，使得其它任务无法运行，就会延长大多数任务的响应时间。不过，这种调度方式也使得任务可以完全控制自己让出的代码位置，有利于实现更高效的上下文切换机制，从而提高吞吐量和CPU利用率。

抢占式调度中，除了任务主动让出CPU资源以外，调度系统也可以强制任务让出CPU资源。这类抢占通常发生在两种情况：一是在涉及任务优先级的调度系统中，创建了更高优先级的任务，则抢占当前任务使高优先级任务立刻运行；二是在支持中断的调度系统中，中断到来，暂停当前任务的执行，上下文切换到中断处理例程，也可以看作一种抢占。抢占式调度由于实现了额外的抢占机制，并且产生了额外的上下文切换，因此吞吐量和CPU利用率都会稍低于协作式调度。不过，基于优先级的抢占使高优先级的任务尽快执行完成，基于中断的抢占使系统及时响应外部事件，它们都能明显降低任务的反应时间，特别是对于高优先级的任务。

本项研究实现的线程和协程调度机制属于协作式调度，与传统的、抢占式的中断处理机制不适配。因此，本文实现了无抢占的中断处理机制，从而将中断处理融入协作式任务调度的机制中，同时也降低了中断发生时，保存上下文的开销。

2.2 Rust 语言的协程实现

2.2.1 协程概述

在高并发的任务调度，特别是任务都涉及I/O处理的情况中，使用操作系统提供的线程模型会带来过大的时间和空间开销。因此，很多编程语言或编程框架设计了比操作系统级线程更轻量、更细粒度的调度单位，例如Kotlin语言的挂起函数、Go语言的Goroutine、Python语言的async/await、C++语言的协程、Rust语言的Future等。下文将这些调度单位统称为协程。

协程依据实现方式的不同，分为有栈协程和无栈协程。有栈协程（例如用户态线程）在上下文切换时会保存函数调用栈的内容，更类似线程切换。这导致切换开销较无栈协程更高。不过这一切换方式也使其同时支持抢占式调度和协作式调度。无栈协程（例如async/await）利用协作式调度的特性，设计了不需要调用栈的上下文切换机制。这导致其切换开销更低，但也决定了它不支持抢占式调度，只支持协作式调度。

2.2.2 Rust 语言中的协程

前文已经叙述了本研究使用Rust语言的理由。因此，下文详细描述协程在Rust语言中的实现。

Rust语言的协程是使用async/await模型的无栈协程，因此它采用协作式调度，并具有较低的开销。Rust语言使用Future特征表示协程、使用async/await关键字简化Future的创建和级联调用、使用Executor提供协程的运行环境、使用Reactor访问I/O资源。（具体的Executor和Reactor实现由第三方库提供。）

如代码2-1所示，Future特征（特征（trait）是Rust语言中的概念，可以类比为C++语言中的抽象基类或Java语言中的接口，但不完全等同）要求实现poll方法，其输入参数为Context，返回值为Poll枚举。Poll枚举有两种取值，Pending代表这次调用并未将协程运行完成，Ready则代表运行完成，并携带了运行结果一起返回。协程的运行方式即是不断地调用poll方法，直到其返回Ready。

Rust语言提供了async/await关键字以简化协程的创建和调用。async关键字可以用

于标记一个函数或代码块，使得编译器自动将它们转化为实现了Future特征的数据结构（下文中简称Future）。await关键字只能在async函数/代码块中使用，它代表调用一个Future，获取它的返回值。await的实质是调用了Future的poll方法，如果返回Pending，则使外部的async函数/代码块也返回Pending；如果返回Ready，则获取Ready中的返回值，继续执行外部async函数/代码块。通过这种方式，await关键字实现了Future的级联调用。

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) ->
Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

代码 2-1 Future 特征在 Rust 语言中的定义

Future之间通过await关键字的级联调用，可以形成一个树状结构。Executor将这样的树作为调度单位，因此可以把每一棵树看作一个协程。在这一定义下，协程和Future的关系可以类比成线程与函数的关系。Executor会管理和调度协程，在循环中不断取出就绪队列中的协程，并通过调用根节点Future的poll方法运行协程。但与一般的调度器不同，协程返回后，Executor并不会将协程重新加入就绪队列。因为协程返回Pending的意义是申请了无法立即获得的资源，因此协程应该阻塞，而协程的阻塞和唤醒是由Reactor负责的。

为了实现协程的阻塞和唤醒机制，Executor会为每个协程建立一个Waker，调用该Waker即可将对应的协程重新加入就绪队列。在通过await关键字级联调用poll方法的过程中，Waker会被封装进Context，作为poll方法的输入参数逐级传递。位于叶子节点的Future不会再使用await关键字，它们通常是手动编写，而非由async关键字生成的，以实现和Reactor的交互。叶子节点的Future将所属协程的Waker和需要进行的I/O任务一起提交给Reactor，Reactor将它们加入任务队列中，调用外部设备等资源完

成这些任务。任务完成时，Reactor会调用对应的Waker，从而唤醒相应的协程。对叶子节点的Future调用poll时，它会检查对应的Reactor中，自己任务的状态，采取对应的动作：

- 若自己的任务已经完成（可以在Reactor的任务表中找到，状态为完成），则返回Ready和执行结果。
- 如果自己的任务已提交未完成（可以在Reactor的任务表中找到，状态为未完成），则直接返回Pending。
- 若自己的任务还未提交（无法在Reactor的任务表中找到），则向Reactor提交一个任务，同时传入自己获得的Context。然后，返回Pending。

综上所述，可以描述Rust语言中，协程的执行流程：

1. 协程被Executor调用poll时，它获得了Executor为它分配的，与它自身对应的Waker。
2. 随着根节点Future使用await逐级调用下层Future，逐层向下传递的Waker也是与该协程对应的Waker。
3. poll到叶子节点Future时（假定是第一次poll，即叶子节点Future还未创建任务），它在Reactor处创建任务，同时将协程对应的Waker传入Reactor。
4. 叶子节点Future返回Pending。由于await的行为，其所有上级Future，包括根节点Future，都立即返回Pending。
5. Executor接收到协程返回的Pending结果，转而调度下一个协程。
6. 和4、5同时，Reactor执行任务。完成后，调用协程对应的Waker，将协程重新加入Executor的就绪队列。

本项研究会使用该机制实现Rust协程的运行时支持。协程的Executor由软硬件共同实现，QEMU中的硬件负责调度协程，ArceOS中的软件负责运行协程。ArceOS中的等待队列和时钟队列作为协程的Reactor，它们实现了协程的阻塞和唤醒操作。同时，硬件调度器中的中断队列也可作为Reactor，将协程放入中断队列即为阻塞操作，中断到来时，硬件将对应队列的协程加入调度队列即为唤醒操作。

2.3 本章小结

本章介绍了本项研究涉及的相关技术：任务调度方面，介绍了任务调度的概念、性能指标，以及协作式调度和抢占式调度的特点。协程方面，介绍了协程的产生原因、

实现方式、Rust语言中与协程相关的语法特性、以及Rust协程的运行流程。同时，介绍了这些技术如何在本项研究中应用。

第 3 章 ArceOS-QEMU-ATS-INTC 的设计

本项研究通过对QEMU模拟器和ArceOS操作系统两个层级的修改，实现了一个可以调度和运行协程、接收外部中断，以及使用协程优化网络性能的计算机系统，ArceOS-QEMU-ATS-INTC。其修改主要分为两个部分：在QEMU模拟器中实现的协程调度器与中断处理器QEMU-ATS-INTC，以及在ArceOS操作系统中修改任务调度模块和网络模块得到的操作系统ArceOS-ATS-INTC。下文将从整体结构和各个部分的结构两个方面，描述本项目的设计。

3.1 整体结构

本项目修改的基础是在QEMU模拟器上运行的ArceOS操作系统。在各个层级的修改如图3-1所示：

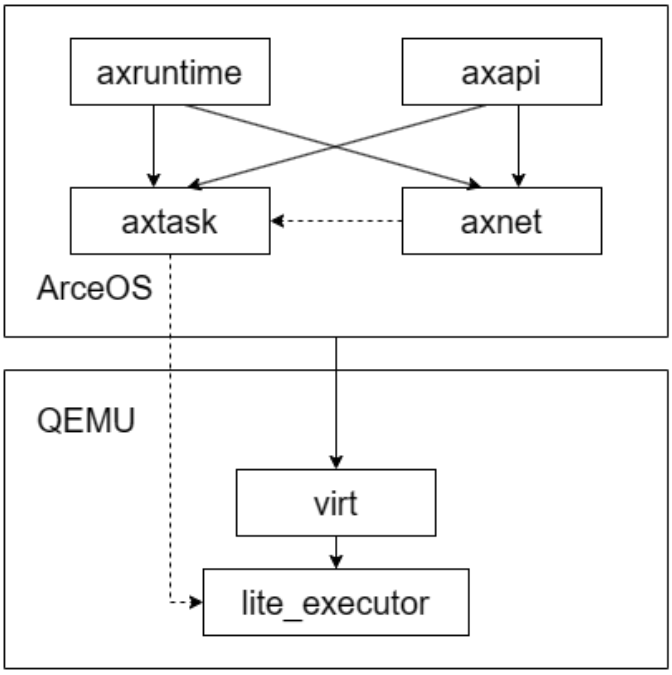

图 3-1 本项目模块的关系图

在QEMU模拟器中，本项目添加了自定义的硬件lite_executor作为中断控制器和任务调度器。它是项目硬件部分的核心模块。virt设备是QEMU中原本存在的虚拟设备，包含CPU、中断芯片、外部设备等部件，代表硬件系统整体。本项目通过修改virt

设备的代码，将实现的lite_executor硬件加入virt设备中。

lite_executor模拟硬件参考了[15]的设计。为了实现协程调度和中断处理的功能，lite_executor硬件向操作系统提供了三个接口，如表3-1所示。ps_fetch从对应进程的调度器中取出下一个运行的协程，ps_push将特定优先级的协程放入对应进程的调度器，intr_push为特定进程注册特定中断号的中断处理协程。这些接口会在3.2.1节中详细说明。

表 3-1 lite_executor 与操作系统的接口

接口名称	输入参数	输出参数
ps_fetch	进程 id	协程指针
ps_push	进程 id、协程指针、协程优先级	无
intr_push	进程 id、协程指针、中断 id	无

ArceOS操作系统可以在QEMU模拟器中的virt虚拟设备上运行。系统中的axtask模块负责任务的调度和管理，因此，是项目软件部分修改的核心模块。修改后的axtask模块可以同时支持线程和协程的调度，并且支持外部中断的处理。axtask模块将任务的调度交由lite_executor硬件管理，通过硬件提供的三个接口使用了它的任务调度和中断处理功能，从而降低了调度算法的运行开销。而任务的运行、阻塞和唤醒仍通过该模块实现的软件方式进行。该模块向用户和其它系统模块新增了两部分接口：一部分是和协程有关的接口，包括协程的创建、让出、阻塞、唤醒等；另一部分是和中断有关的接口，即注册中断处理函数。

axnet模块为ArceOS系统实现网络功能。本项目基于修改后的axtask模块提供的协程和中断功能，通过将一部分操作修改为协程和通过中断触发更新操作的方式，优化了axnet模块的网络实现。该模块仅修改了内部实现，而外部接口保持不变，其修改对用户透明。

系统的axruntime模块负责在启动时初始化各个模块，从而为应用程序提供运行时支持；axapi模块负责为应用程序提供调用接口，使其可以使用操作系统提供的功能。本项目也相应地修改了这些模块，使得本项目新增的功能既可以被操作系统使用，又可以被应用程序使用。

3.2 QEMU-ATS-INTC 的设计

在QEMU模拟器中，本项目主要的设计为lite_executor硬件。而对virt设备的修改只是为了添加lite_executor硬件，不涉及设计层面。本项目将lite_executor硬件加入到virt设备上、为lite_executor连接了外部中断线，并使用MMIO方式实现了lite_executor硬件和操作系统的接口。

3.2.1 lite_executor 硬件

本项目设计的lite_executor硬件可以将协程调度功能与中断处理功能相结合，这两项功能由相关联的一组任务调度队列和一组中断处理队列实现，如图3-2所示。该硬件有两个输入输出机制：一方面接收外部设备的中断，另一方面向操作系统提供ps_fetch、ps_push、intr_push三个接口，如表3-1所示。

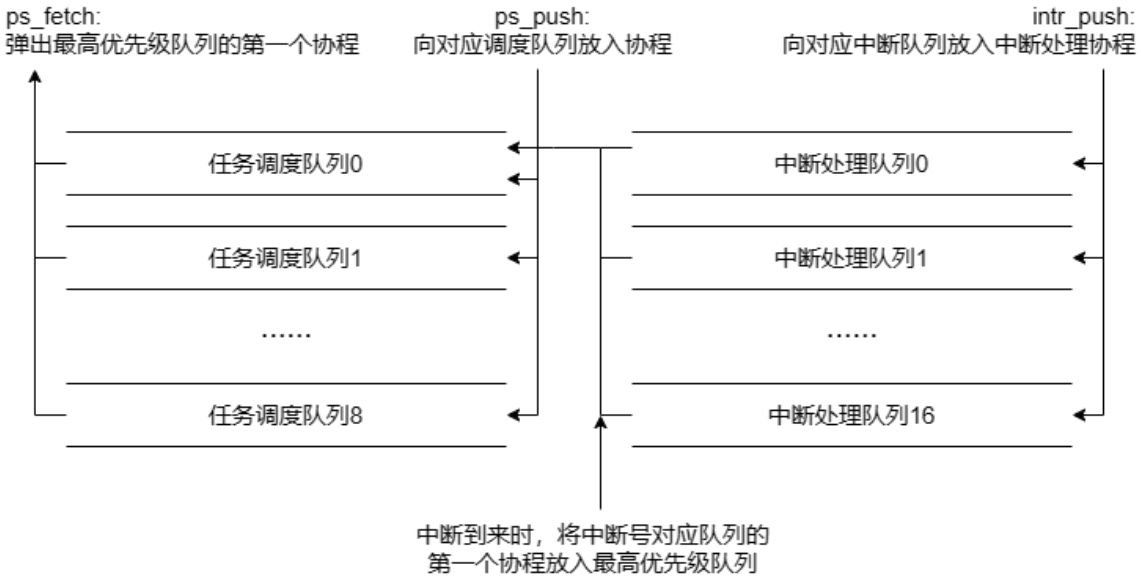

图 3-2 硬件队列示意图

在任务调度队列中，队列的序号代表任务优先级。操作系统可以调用ps_push接口将各个优先级的协程加入相应的调度队列，并调用ps_fetch接口，使硬件弹出非空的最高优先级队列中的第一个协程。通过这样的队列和接口设计，lite_executor硬件实现了基于优先级的协程调度算法，且同一优先级内采用先入先出算法。

在中断处理队列中，队列的序号代表中断号。操作系统可以调用intr_push接口，

向对应的中断队列中放入协程，从而将它们作为中断处理协程。当外部中断到来时，硬件会将中断号对应的队列的第一个协程（如果有）从中断队列移到最高优先级的任务调度队列。当操作系统下一次调用ps_fetch时，就可以获取中断处理协程并运行了。

这样的中断处理机制类似于软中断，通过将中断处理的逻辑作为新任务创建，使中断处理程序本身尽可能精简，避免中断嵌套等情况发生，降低中断开销。但本研究的方式比一般的软中断更进一步：一般的软中断，即使中断处理程序再精简，也需要在中断到来时进行上下文切换，增大开销的同时也造成了任务的抢占。而本文的情况，中断处理全部由硬件完成，不会抢占当前任务，没有上下文切换的开销，同时也与Rust协程的协作式调度更加适配。

通过以上的队列机制，lite_executor硬件实现了一种支持多优先级、支持中断处理的协作式任务调度。这样的调度机制通过对优先级和中断的支持，吸收了抢占式调度的响应时间更短的优势。同时又由于协作式调度的本质，从而保留了与Rust协程的适配性。

除了用于实现主要功能的两组队列外，lite_executor硬件还添加了多进程的支持。每个进程会获得一组任务调度队列和中断队列，并使用它们调度内部的协程，互不干扰。目前，只有代表系统内核的0号进程的中断队列有作用，所有的中断都发往0号进程，触发其中断处理机制。未来，该架构也可以支持其它进程直接进行中断处理，不需经过内核，从而降低中断处理的开销。lite_executor硬件设置了4组队列，支持4个进程同时运行。其面向操作系统的接口会将进程id作为输入参数，通过内部的映射机制，根据进程id找到对应的队列。

3.3 ArceOS-ATS-INTC 的设计

本项目在ArceOS中的修改的主要部分为axtask模块和axnet模块。下文重点说明修改这两个模块的设计。而axruntime和axapi模块仅是适应axtask模块和axnet模块的修改而调整，因此不在设计中详细描述。

3.3.1 axtask 模块

任务调度模块axtask由api、timer、ats、wait_queue、task五个子模块构成，各个模块的关系如图3-3所示。task子模块定义了任务的数据结构，ats子模块一方面调用liite_executor硬件实现任务调度，另一方面自身负责任务的运行。timer和wait_queue子模块负责任务的阻塞和唤醒。api子模块负责提供axtask模块的对外接口。

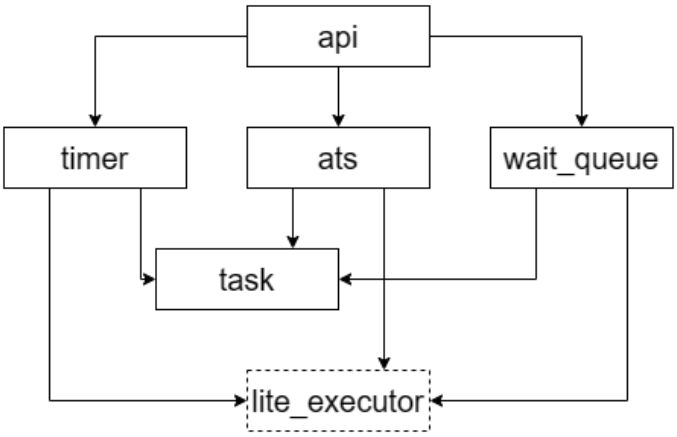

图 3-3 任务调度模块关系图

其中，task子模块定义了任务的数据结构和行为。在原本的ArceOS中，此处的任务为线程。为了保证与原有应用的兼容性，同时发挥协程调度的优势，本项目决定同时支持线程和协程的调度，并将线程和协程视为并列的调度单位。为了同时调度线程和协程，需要采用运行时多态，提供统一的上下文切换接口。本项目选择类似协程的poll方法作为上下文切换的接口。对于协程，直接调用内部Future的poll方法即可。对于线程，在poll方法内部进行线程式的上下文切换——保存和恢复寄存器。

ats子模块用于取出任务并运行，同时也提供协程运行环境（例如函数调用栈等），其会运行一个循环，不断重复以下的行为：先调用lite_executor硬件的ps_fetch接口获取需要运行的线程/协程，之后调用它们的poll方法。由于线程/协程的统一接口采用了类似协程poll方法的形式，ats模块也采用了类似协程Executor的设计。在多核情况下，每一个CPU核心都运行一个ats对象，它们从相同的硬件中取出任务，之后分别执行，互不影响。

timer子模块利用时钟中断，实现线程睡眠（sleep）的功能。由于时钟中断的实时性要求很高，而且并非由外部设备发出，因此时钟中断的处理仍沿用了原有的机制，不由lite_executor硬件处理。wait_queue子模块已经实现了阻塞队列，可以在系统的各处使用。之前的修改对这些模块的影响只是改变了任务唤醒的机制，因此将任务唤醒的机制改为调用lite_executor硬件的ps_push接口实现，即可让线程再次正常使用这些模块。然而，为了使协程使用await语法进行睡眠或阻塞操作，还需要在这两个模块中添加专用于协程的、声明为async的函数。

api子模块用于将用户需要使用的功能封装为api，之后这些api会汇总到axapi模块。但因为任务调度模块在操作系统中的广泛使用，该子模块中的函数实际上既会被应用程序使用，也会被操作系统使用。由于修改了代表任务的数据结构和部分接口，因此也要在api子模块中做对应的修改。此外，在其中添加了调用lite_executor硬件的intr_push接口，注册中断处理线程或协程的api。

3.3.2 axnet 模块

对任务调度模块的修改为ArceOS引入了协程支持和外部中断支持，本研究利用这些新实现的机制优化操作系统的网络模块。

ArceOS的网络模块使用了smoltcp第三方库提供的网络协议栈，其在socket、网络接口、网络设备等多个层次均设置了缓冲区。调用poll_interface函数，会将上层的发送缓冲区的内容传递到网络设备层，并将下层的接收缓冲区的内容传递到socket层，从而实现各个socket与网络设备之间的数据传输。该函数开销较大，且调用时需要获取网络设备、网络接口、socket集合等各个全局数据结构的互斥锁。因此，减少对该函数的调用，特别是并发调用，可以改善网络模块的性能。

在axnet模块原本的实现中，每个socket在每次调用读、写等操作时，都会先调用一次poll_interface函数，之后尝试通过内部缓冲区完成需要的操作。若操作无法完成（例如，读缓冲区空或者写缓冲区满），则让出CPU并切换到就绪态，待下次执行时再重新调用poll_interface函数，实质是一种轮询。这样的实现较为简单，也产生了许多不必要的调用次数。并且，如果在多核情况下打开了多个socket，就会导致对poll_interface的大量并发调用，从而进一步增大开销。

本项目对axnet模块的优化目标为：

1. 使用专用的任务调用poll_interface函数，socket不再直接调用该函数，从而消除并发调用的情况。
2. 将socket无法完成操作时的行为由让出改为阻塞，减少不必要的任务切换和poll_interface函数的调用。

有两种方案都可以实现该目标。

第一种方案基于专门的轮询协程。该方案参考了[16]，使用一个专门的协程处理所有socket的轮询需求，而非让它们各自轮询。由于poll_interface函数的性质，该任务只需要循环调用poll_interface函数即可。该函数具有bool类型的返回值，表示这次调

用是否有socket的可读写性发生变化。可以利用这一信息，只在调用返回true时，唤醒所有因socket阻塞的任务。该方案虽然对poll_interface函数调用次数的减少很有限，但消除了并发调用，并且网络设备的处理结果可以第一时间同步到socket，能够缩短响应时间。

第二种方案基于中断。注册相应的中断处理协程，在网络设备的中断到来时，调用poll_interface函数，并根据返回值决定是否唤醒因socket而阻塞的任务。由于网络设备中断只能通知数据的接收，而无法通知数据的发送，因此socket在发送数据时还是需要调用一次poll_interface函数。为了消除并发调用，无论是socket还是中断处理程序都不会直接调用该函数，而是唤醒一个专用的协程，由它调用poll_interface函数。

3.4 本章小结

本章介绍了本项目的设计方案，先从ArceOS-QEMU-ATS-INTC系统总体角度描述了各个模块的功能和关系，再分别从QEMU模拟硬件层次（QEMU-ATS-INTC）和软件层次（ArceOS-ATS-INTC）描述了系统的详细设计。其中，硬件层次主要介绍了lite_executor硬件的设计，软件层次主要介绍了axtask和axnet模块的设计。

第 4 章 QEMU-ATS-INTC 的实现

4.1 QEMU 模拟器介绍

QEMU是一个开源的硬件模拟器，它可以模拟不同的硬件架构，使得为一种硬件架构编写的软件可以在另一种硬件架构的计算机上运行[17]。它具有两种模拟模式：其一为系统模拟，QEMU会模拟一个完整的硬件设备，可以在其上运行为其它架构编写的操作系统。其二为用户态模拟，QEMU承担机器代码翻译等工作，从而在用户态运行为其它架构编写的应用程序。本项研究使用了QEMU的系统模拟功能，在x86架构的宿主机上运行ArceOS操作系统的RISC-V架构版本。

QEMU源代码使用C语言进行编写。在系统模拟中使用的，由软件模拟出的硬件也使用C语言实现。这些硬件的代码存放在“hw”（存放“.c”源文件）和“include/hw”（存放“.h”头文件）两个目录中，目录下的每个文件夹对应一个硬件。CPU、中断芯片、总线、外部设备、主板等组成硬件系统的各个部分，都是以这种方式模拟的。

虽然C语言本身不支持面向对象编程，但QEMU通过宏定义等方法实现了面向对象编程模型——QOM。在此基础上，每个硬件都由一个类型表示，它们共同的基类是DeviceState类型（可能不是直接基类）。硬件的初始化分为两步：create，创建一个DeviceState对象，并设置DeviceState对象的一些属性；realize，在该DeviceState对象内部执行不同种类硬件特有的初始化过程。硬件要发挥作用，就需要在realize过程中设置与外界通信的各种方式，例如，注册MMIO内存区域、连接PCI总线，或者连接中断线，并在硬件内部实现响应这些通信的功能代码。这些机制能使硬件响应外部事件，并做出反应，其输出又可能成为其它硬件的外部事件。由于实现硬件功能的过程都在realize步骤中完成，因此不会受到基类DeviceState的约束，使得硬件的功能具有很高的自由度。

4.2 lite_executor 硬件的实现

为了实现本文所述的调度器和中断处理器硬件——QEMU-ATS-INTC，首先需要创建该硬件的类型。之后，实现它与外界通信的机制，其中包括3.1节所述的硬件向操作系统提供的接口。如此搭建好硬件的框架代码后，再实现3.2节所述的硬件的核心功能，即任务调度和中断处理功能。

4.2.1 创建硬件类型

创建该硬件的类型，需要进行如下步骤：

首先需要定义结构体。该结构体表示自定义硬件的内部结构。本文所述硬件的结构体命名为RISCVLiteExecutor，定义在include/hw目录下的头文件中。

之后，在QOM框架中注册该结构体对应的类型。在hw目录下的源文件中，先使用QOM框架提供的TypeInfo结构体，在其中记录自身的类型、基类、对象大小、realize函数等信息。由于设计中，该硬件会连接在系统总线上，因此，选择TYPE_SYS_BUS_DEVICE作为该类型的直接基类。再使用type_init宏和type_register_static函数注册该TypeInfo对象。由此，本项目定义的类型已经加入到QOM框架中。

最后，实现自身的create和realize函数。在create函数中，调用qdev_new函数创建DeviceState对象，并使用qdev_prop_set系列函数，为该对象设置了uint32类型的num_sources属性。realize函数需要进行三项工作：第一，调用基类TYPE_SYS_BUS_DEVICE的realize函数；第二，初始化硬件内部结构；第三，设置外部通信方式。

4.2.2 实现外界通信机制

本项目实现的硬件需要为操作系统提供接口，同时接收外部设备的中断。因此，QEMU-ATS-INTC与外界通信的机制如下：一方面，使用MMIO机制用于与操作系统通信，另一方面使用GPIO端口连接外部设备的中断线。

MMIO是一种操作系统与硬件通信的方式。硬件将自己连接在系统总线上，将自己的输入/输出端口映射到一块物理内存区域。操作系统读写这块内存区域，就是读写硬件的输入/输出端口。

如3.2.1节所述，硬件向操作系统提供了以下三种接口：ps_fetch、ps_push和intr_push。他们的输入输出参数如第3章表3-1所示。

由于进程id、协程优先级、中断id这些输入参数的取值固定在某个范围内，因此，在MMIO实现的接口中，可以通过访问不同的内存地址代表这些参数的取值，而不需要显式传递这些参数。需要显式读写的参数只有协程指针，因此所有的接口都可以实现为：以8字节的长度（协程指针的大小）读/写一个特定的内存地址。

为了使不同的内存地址对应不同的功能、不同的进程id、不同的中断id，本文依

据[15]的设计，为该硬件连续分配了16MB的MMIO内存空间。其中，每个进程id占用4KB的连续空间，支持的进程id范围为0≤id＜4096。对每个进程内的4KB空间，如图4-1划分：

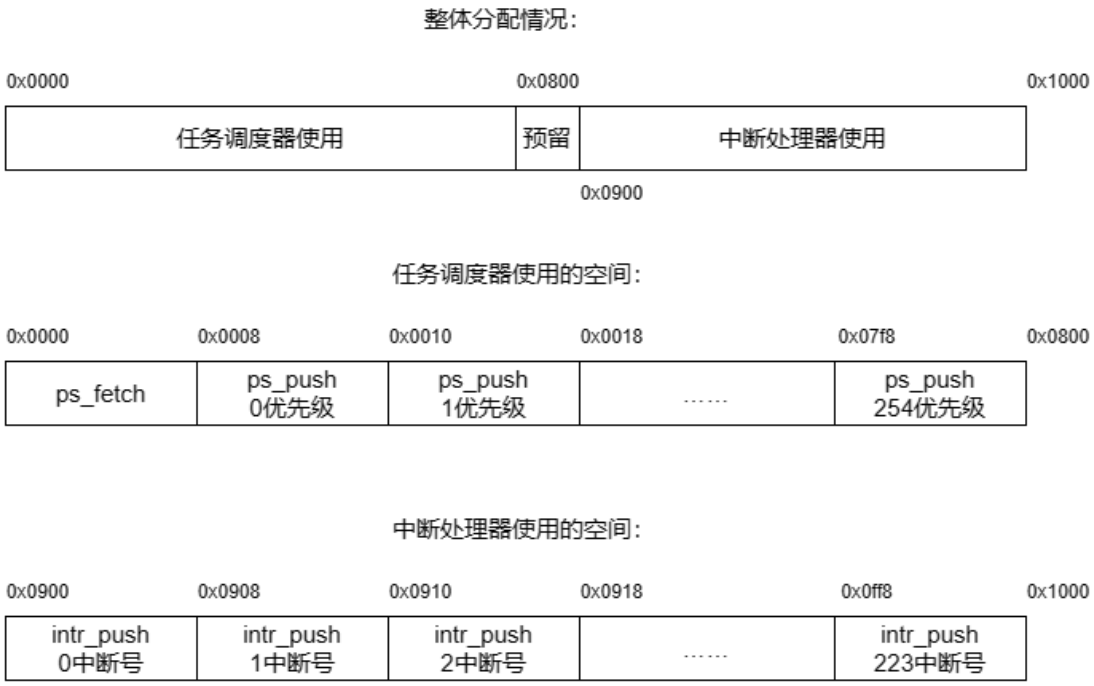

图 4-1 每个进程内 MMIO 空间的分配情况

按照空间分配规则，读写相应的内存区域，即可实现接口的调用，例如，读0x0000地址，得到的结果为调用0号进程的ps_fetch接口获得的协程指针；将协程指针写入0x1908地址，相当于调用了1号进程的intr_push接口，将相应指针放入1中断号的队列。

为了实现该MMIO机制，首先需要准备好处理MMIO读写的函数，本文将其分别命名为riscv_lite_executor_read和riscv_lite_executor_write。在两个函数中，使用条件语句判断读/写地址所在的内存区域。创建MemoryRegionOps类型的对象，在其中记录之前定义的读写函数、读写单元的大小（8字节）、大小端设置（小端）等属性，该对象即可用于响应操作系统的MMIO请求。在该硬件的realize函数中，调用memory_region_init_io 、 sysbus_init_mmio 函数， 在create函数中， 调用sysbus_mmio_map函数，从而通过基类TYPE_SYS_BUS_DEVICE的功能注册MMIO区域。经过这些操作后，该硬件注册了MMIO内存区域，在收到MMIO读写请求时，

就会调用之前创建的两个函数处理。

完成MMIO机制的实现后，还需要实现与外部设备的中断线相连的输入端口。为了实现这一点，需要首先定义用于处理中断的函数。本文将该函数命名为riscv_lite_executor_irq_request。该函数会在中断线传来信号时调用。在realize函数中，调用qdev_init_gpio_in，注册GPIO输入端口，并将之前定义的函数绑定在端口上。以上步骤之后，端口注册完成，此处注册的端口之后会连接外部设备的中断线。

4.2.3 实现任务调度和中断处理功能

如3.2.1节所述，本项目使用队列实现任务调度和中断处理的功能。其中，操作系统调用ps_fetch接口从调度队列头部取出协程，调用ps_push接口从调度队列尾部加入协程，调用intr_push从中断队列尾部加入协程。中断到来时，从中断队列头部取出协程，并加入调度队列尾部。

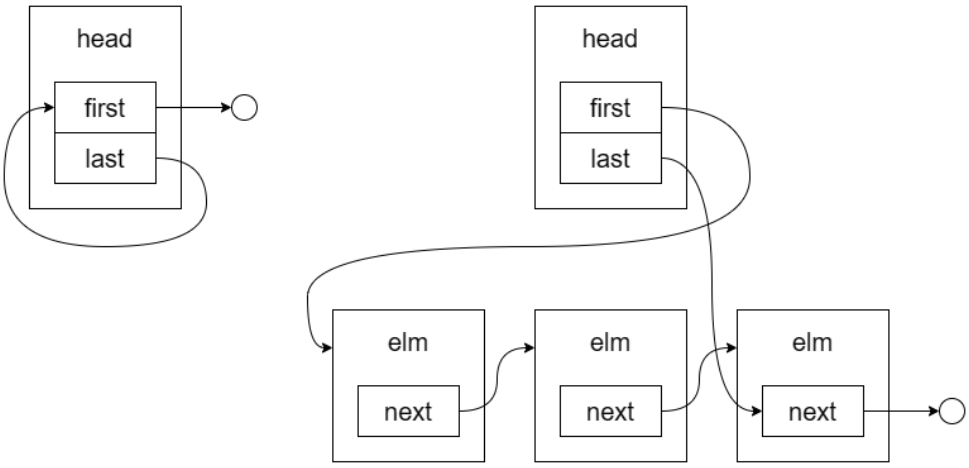

图 4-2 QEMU 简单队列的内存布局

QEMU的源代码中提供了实现各种队列类型的头文件include/qemu/queue.h。其中的简单队列（QSIMPLEQ）类型可以从队首弹出元素、从队尾添加元素，符合本文设计的需求，因此使用QEMU的简单队列实现设计中的任务调度队列和中断处理队列。简单队列的内存布局如图4-2所示。每个队列都有一个头节点（head），其代表整个队列。对队列进行的插入、删除等操作也是通过头节点完成的。队列不会管理自己节点的内存空间，需要手动创建队列节点再加入队列，并手动释放弹出节点的内存空间。

由于任务调度队列和中断处理队列都是由多条队列组成的，并且具有涉及多条队列的特殊操作（例如取出优先级最高的任务），因此对简单队列再进行一层包装，创建PriorityScheduler和ExternalInterruptHandler类型。它们分别代表任务调度队列和中断处理队列。PriorityScheduler具有ps_push方法，向指定优先级的队列放入协程，以及ps_pop方法，取出优先级最高的协程（若没有协程，则返回0）；ExternalInterruptHandler具有eih_push方法，向指定中断号的队列放入协程，以及eih_pop方法，取出指定中断号的队列的协程。这些方法同时实现了对队列节点的存储空间管理，使方法的调用者不需考虑队列节点的创建和释放。

创建了各种队列的类型后，在该硬件的realize函数中初始化这些队列。之后，在处理MMIO读写的函数riscv_lite_executor_read和riscv_lite_executor_write中，在访问对应的内存区域时调用ps_push、ps_pop、eih_push方法，从而实现了硬件与操作系统的接口。在处理GPIO输入端口的函数riscv_lite_executor_irq_request中，先后调用eih_pop和ps_push函数，从而实现在中断到来时，将中断队列中的处理协程加入调度器的功能。

4.3 virt 设备的修改

virt设备是QEMU模拟器提供的，RISC-V架构的虚拟设备，本项目将其用作运行RISC-V版ArceOS的平台。如3.2节所述，本项目对virt设备的修改目标是添加lite_executor硬件，具体进行了以下几项修改：

1. 在virt_memmap，存放MMIO地址映射信息的数组中，添加了lite_executor的映射区域。其基址为0xf000000，长度为16MB。
2. 在virt_machine_init，初始化virt设备的函数中，添加了lite_executor的初始化代码，并将virt设备结构体的irqchip成员设置为lite_executor。virt设备接下来的初始化流程会将该irqchip成员传递给各个总线和外部设备的初始化函数中，从而连接各个外部设备与lite_executor的中断线。
3. 在create_fdt_sockets，创建设备树的函数中，为lite_executor硬件创建设备树节点。此处创建的设备树节点可以使操作系统探测到lite_executor硬件。

4.4 本章小结

本章介绍了项目在QEMU中模拟硬件中的实现，即QEMU-ATS-INTC。首先创建

了自定义硬件lite_executor，在QOM中注册了该硬件的类型、设置了该硬件的MMIO内存区域和GPIO端口、使用简单队列实现了其任务调度和中断处理的功能。之后，将该硬件添加到virt设备中。

第 5 章 ArceOS-ATS-INTC 的实现

5.1 ArceOS 介绍

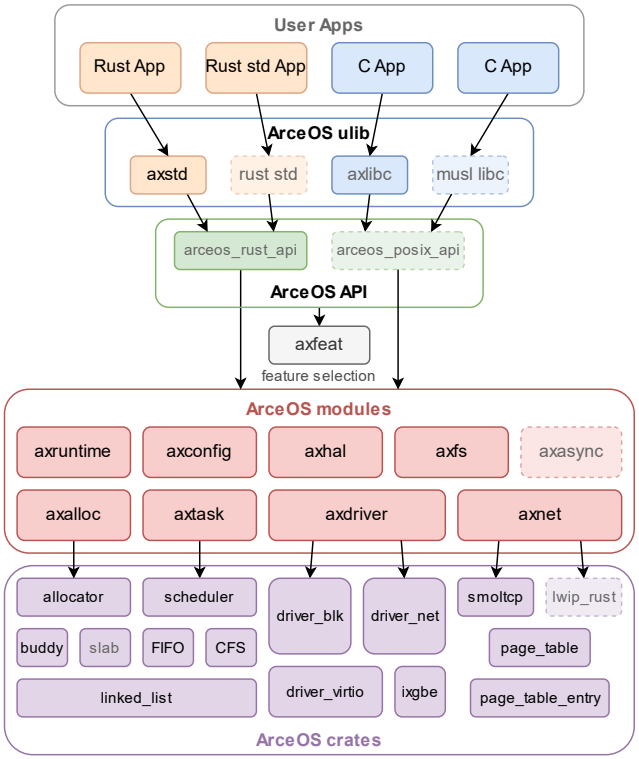

图 5-1 ArceOS 系统结构（来自[18]）

ArceOS[18]是一个实验性的、模块化的、使用Rust语言实现的操作系统。该系统通过Rust语言的feature条件编译特性实现了模块化，通过开启相应模块对应的feature，可以将这些模块对应的代码加入编译，从而使该系统分别或同时支持多线程调度、virtio设备驱动、网络、互斥锁、文件系统等功能。

ArceOS是一个unikernel系统。unikernel系统的设计目的是作为应用运行的容器，因此，其功能相较于传统操作系统进行了大量的精简，只留下了运行单个应用必要的部分。作为unikernel系统，ArceOS具有以下特点：

1. 取消了进程级的任务调度功能，整个系统只能运行一个应用（系统和应用可以视为一个进程），和运行的应用一同启动、一同关闭。
2. 取消了用户态和内核态的划分，应用程序和系统内核运行在同一特权级下。

因此，用户程序不需要系统调用，而是使用函数调用的方式访问系统api。

3. 整个系统只有一个地址空间，该空间被系统内核和用户程序共享。

这些特点为本项目的开发工作提供了方便，也简化了项目的实现，降低开销。

ArceOS的系统结构如图5-1所示，从下往上分为crates、modules、API、ulib四层。crates层包含各种实现了操作系统一部分功能的函数库，例如硬件驱动、内存分配器、网络协议栈等。这些函数库并不与ArceOS绑定，其也可以用于其它Rust语言的操作系统中。modules层为ArceOS的各个模块。ArceOS将系统的不同功能封装在不同模块中，通过条件编译控制功能的启用与否。API层为各个模块向用户暴露的api。ulib层为利用这些api实现的，类似Rust std和libc的用户函数库。如前文所述，用户程序通过ulib、api、modules的路径调用系统的功能，该路径只涉及函数调用，不进行系统调用。

5.2 axtask 模块的修改

5.2.1 task 子模块

如3.3.1节所述，该模块定义了任务的数据结构和行为。本项目需要修改代表任务的数据结构，并为它实现运行、阻塞、唤醒等行为，以使任务调度机制同时支持线程和协程。

为了使用新的任务数据结构表示线程和协程，代表任务的数据结构定义如下：

```
pub struct AxTask {
    pub task_inner: Box<dyn AbsTaskInner>,
    pub general_inner: GeneralInner,
}
```

代码 5-1 任务数据结构的定义

该数据结构有task_inner和general_inner两个成员。

task_inner的类型是特征为AbsTaskInner的特征类型，具有运行时多态的性质。因此，在这个成员中，线程和协程可以提供具有相同接口的不同实现。task_inner存放了任务与执行流直接相关的数据。对于线程而言，它们是入口点、函数调用栈和寄存器上下文。对于协程而言，它就是Future对象。AbsTaskInner特征需要控制任务的开始、保存、恢复、终止，这由它提供的poll方法实现。poll方法的定义与Future特征的

poll方法类似，执行该方法会推动任务的执行，并在执行中止/结束时得到Pending或者Ready的返回值。对于协程而言，poll方法只需调用其内部Future的poll方法即可。但线程的切换方式是寄存器的保存和恢复，涉及执行流的切换，和线程的基于函数调用和返回的切换不同。为了将其使用poll方法表示，本项目认为执行器运行的环境也是一个线程上下文，在进入poll方法后，使用寄存器保存恢复的方式切换到目标线程的上下文。在线程让出/阻塞/退出时，再使用该方式切换回执行器运行环境的poll方法中。返回poll方法的切换与一般的线程切换不同，因为它还需要返回Pending或Ready的返回值。因此，本项目实验得到了RISC-V架构中，这类返回值在寄存器中传递的规范：对于Pending，设置寄存器a0为0；对于Ready，设置a0为1，a1为Ready中包含的返回值。并且，根据该规范，编写了在上下文切换的同时向poll方法返回Ready或Pending的RISC-V汇编代码。如此，就实现了线程的poll方法。

general_inner中存储了与任务类型为线程或协程无关的信息。它们包括id、名称、状态、优先级等内容，并向AxTask类型提供了读写这些信息的接口。

为了与原有的线程API兼容，本项目为任务实现了线程的让出、阻塞、睡眠、退出等各种方法。这些方法只有在任务类型为线程时才能调用。这些方法的代码与原有的、用于线程的代码大致相同，只是将最后切换上下文使用的函数改为了返回Pending或Ready的版本。由于阻塞队列和时钟队列子模块也使用了其中的一些方法，为了使协程也可使用阻塞队列和时钟队列的功能，本项目仿照这些代码实现了协程的让出、阻塞、睡眠方法。为了使这些方法与协程的async/await机制相适应，本项目一方面为这些方法添加了async签名。另一方面，本项目实现了一种特殊的Future，其poll方法在第一次调用时返回Pending，第二次调用时返回Ready。使用await这样的Future代替线程的上下文切换代码，则协程在运行到此处时，返回的Pending会暂停任务的执行并返回执行器。而协程恢复运行时会再次运行此处代码，此时返回的Ready会使协程继续执行。由此，本项目实现了适合协程机制的让出、阻塞、睡眠方法。

5.2.2 ats 子模块

如3.3.1节所述，该模块用于取出任务并运行，并提供协程的运行环境。该模块定义了执行器、当前任务、调度器硬件的驱动等全局变量。其中，执行器和当前任务使用percpu库提供的功能，为每个CPU核心分配各自的实例。不同CPU核心的执行器和当前任务实例不会相互影响，从而使初始化完成后，各个CPU核心可以相互独立地执

行任务。相较于每个CPU核心在任务切换时都需要获取全局调度器锁的原有设计，本项目的实现可以减小多核同步的开销。多个核心使用加锁的全局硬件调度器驱动访问硬件调度器，从而避免同步问题。虽然仍引入了加锁开销，但该设计中的临界区小于原有设计。

执行器类型提供了run方法，该方法会运行一个无限循环，因此它永不退出。循环中，执行器会从硬件调度器中获取一个任务，将其状态设置为运行，之后调用它的poll方法运行它，直到任务从poll方法中返回。使用该方式，run方法可以运行所有就绪的线程和协程，直到它们全部完成。在CPU核心上调用执行器类型的run方法，即代表该CPU核心已经准备好运行用户程序。

由于本项目使各个CPU核心独立运行，不再设置加锁的全局调度器，因此也会产生一些新的同步问题。例如，当任务阻塞时，会先将自己加入阻塞队列，再进行寄存器保存和任务切换。如果加入阻塞队列后、保存寄存器之前，该任务就被其它核心唤醒，就会使用原有的寄存器上下文再次执行，造成逻辑错误。为了解决这一问题，本项目在任务的实现中增加了return_action字段，其保存一个参数为任务的函数。在执行器的run方法中，当任务返回后，执行器会以该任务为参数调用其return_action函数。该机制可以使任务注册自己返回后的行为，从而使任务先返回，再将自己加入阻塞队列，从而解决了以上的同步问题。除此以外，该机制还可用于其它情况，例如，让出的任务在return_action中将自己再次加入硬件调度器。

5.2.3 其它子模块

在定时器子模块timer和阻塞队列子模块wait_queue中，本项目主要做了以下三种修改：首先，修改了任务唤醒的方式，通过调用全局硬件调度器驱动，最终调用ps_push接口，将任务加入硬件调度器的调度队列的方式唤醒任务。之后，修改了注册任务的定时器事件/将任务加入阻塞队列的时机，使用return_action机制，在任务保存上下文并返回调度器后，再注册定时器事件或加入阻塞队列。最后，为协程增加了一些用async关键字签名的、和线程类似的接口。在已经为协程实现了async的让出、阻塞、睡眠等操作的情况下，这些与定时器、阻塞队列相关的接口也很容易实现。

在api子模块中，首先修改/新增了创建线程/协程的API，使用新的任务数据结构的new函数创建线程/协程，并调用硬件调度器的ps_push接口，将创建的任务加入硬件调度器。之后，新增了调用硬件的intr_push接口，注册中断处理线程/协程的接口。

同时，将之前为协程实现的让出、阻塞等操作也加入了api子模块。最后，修改了整个模块的初始化函数，使其初始化自己CPU核心的全局变量。对于主CPU核心，还需额外初始化各个核心共用的硬件调度器驱动。

5.3 axnet 模块的修改

如3.2.2节所述，在本模块中，需要实现使用专用的任务调用poll_interface函数和将socket无法完成操作时的行为由让出改为阻塞的修改目标，以及基于专门的轮询任务和基于中断的两种修改方案。两种方案可以通过条件编译的方式整合在该项目中，在编译时启用不同的feature以选择不同的方案。

首先，将socket的让出行为改为阻塞。axnet模块原本实现让出的方式为：各类socket操作在无法立即完成时，会返回WouldBlock错误类型。模块设计了接收这些操作的block_on函数，执行流程如下：

```
fn block_on<F, T>(&self, mut f: F) -> AxResult<T> {
    loop {
        SOCKET_SET.poll_interfaces();
        match f() {
            Ok(t) => {
                return Ok(t);
            },
            Err(AxError::WouldBlock) => {
                axtask::yield_now();
            },
            Err(e) => return Err(e),
        }
    }
}
```

代码 5-2 原本 block_on 函数的执行流程（有简化）

该函数主体是一个无限循环，在循环内首先调用一次poll_interface函数。之后，调用一次需要执行的操作，如果返回了WouldBlock，则先让出CPU，在下次运行时，返回循环开头，重复以上过程。否则，跳出循环并返回该操作的返回值。

由于socket进行的所有让出均由block_on函数执行，因此，本项目首先在模块中设置了全局变量形式的阻塞队列，并将block_on函数中让出改为阻塞在上述阻塞队列

中。但任务阻塞后还需实现唤醒机制，而在poll_interface函数执行后唤醒任务是最好的做法，因为只有poll_interface函数会更新socket的缓冲区，使它们的可读/可写状态可能变化。

poll_interface函数的调用时机以及任务的唤醒时机与使用的两种修改方案有关。

在基于专用轮询任务的方案中，本项目在网络模块初始化的过程中创建了一个任务（该任务可以是进程或协程，用于比较性能差异），该任务会在无限循环中，先调用poll_interface函数，再根据其返回值决定是否唤醒阻塞队列中的所有任务。

在基于中断的方案中，如3.3.2节所示，有两种事件都会触发poll_interface函数的调用：网卡中断到来，以及socket操作返回WouldBlock。网卡中断到来的触发，使用注册的中断处理线程/协程实现。返回WouldBlock的触发，通过修改block_on函数实现。为了消除并发调用，本项目创建了一个专门调用poll_interface函数的任务（同样可以是线程或协程），并为其配备了一个专用的阻塞队列。该函数每次调用poll_interface，并根据返回值唤醒任务后，就会阻塞在这个队列中。两种触发方式的实质是从该阻塞队列中唤醒了该专用任务。本项目通过该方式消除了poll_interface函数的并发调用。

5.4 其它模块的修改

对axruntime模块的修改体现在ArceOS的初始化流程上。在主核心和副核心初始化的过程中，需要分别调用修改后的axtask模块的初始化函数的主核心/副核心版本。而在每个核心初始化结束后，主核心需要将main线程加入硬件调度器中。之后，各个核心需要调用该CPU核心的执行器的run方法，进入run方法标志着该核心初始化完成，开始运行用户程序。而axnet模块虽然修改了初始化代码，但初始化函数的参数、返回值并未改变，因此无需在axruntime模块中修改对应内容。

在axtask模块的api子模块修改时，也需上层api进行相应的修改。其需要先使用define_api宏注册axtask新增的API，再使用axtask模块在axapi模块中实现注册的api。最后，还需在ulib层次的各个用户库中添加axapi模块中新增的api。使用这种方法，为用户程序新增了协程创建的API。

5.5 本章小结

本章讲解了本项目在ArceOS操作系统中的实现过程，即ArceOS-ATS-INTC。首

先对ArceOS及其特性做了简要介绍，之后，从任务数据结构、执行器实现、阻塞队列实现等方面介绍了对任务调度模块axtask的修改；从将让出修改为阻塞、修改poll_interface函数调用方式、设置任务唤醒方式等方面介绍了对网络模块axnet的修改；最后介绍了一些外围模块的对应修改，例如axruntime和axapi。

第 6 章 实验与评估

本项目对实现的系统，ArceOS-QEMU-ATS-INTC开展了以下实验，并且以未修改的ArceOS系统和QEMU模拟器作为对照，通过对比系统的行为验证系统实现的正确性，通过对比测量的各项性能指标分析系统的性能表现。

本文设计了三组实验。第一组实验使用编写的测试程序，仅测试系统的任务调度模块（包括QEMU中实现的调度器和ArceOS中的axtask任务模块）。第二组实验也使用编写的测试程序，测试整个系统（包括网络模块和任务调度模块）。第三组实验使用Redis数据库作为测试软件，测试系统在真实负载下的表现。

6.1 实验环境

本文开展的各项实验，均运行在Windows系统中的Linux虚拟机中。

实验的硬件环境如下：

- CPU：AMD Ryzen 5 4600U，6核心12线程，2.10GHz
- 内存：16GB
- 系统架构：x86_64

Windows宿主机的软件环境和虚拟机环境如下：

- 操作系统版本：Windows 11 家庭中文版，版本22623.1028
- 虚拟机软件：Oracle VM VirtualBox，版本7.0.10 r158379
- 虚拟机CPU：4核心
- 虚拟机内存：4GB
- 虚拟机的系统架构：x86_64
- 虚拟机使用的硬件加速：嵌套分页、KVM半虚拟化

虚拟机内运行的Linux系统软件环境如下：

- Linux内核版本：6.5.0-28-generic
- 发行版：Ubuntu 22.04.3 LTS

6.2 任务调度实验

6.2.1 实验设计

本实验测试系统的任务调度能力。本文选用并修改ArceOS自带的示例程序

apps/task/parallel作为测试程序。该程序首先会生成指定长度的随机数数组，并在当前线程计算每个元素的平方根之和。之后，程序会创建指定数量的线程，使每个线程计算该数组的一部分元素的平方根之和。最后，程序等待所有线程计算完成，收集每个线程的计算结果并求和，将最终结果与当前线程计算的结果比较。本项目对该程序的修改有以下两点：第一，在创建线程前和结果收集后增加计时代码，测量多线程计算所用的时间。第二，增加了创建协程代替原本的线程进行计算的代码。

这组实验的测试指标有两项。首先，观测多任务和单任务计算结果的比较。两者相等，则说明多任务的计算过程无误。之后，记录多任务计算所用的时间。相同条件下，时间越短，说明任务调度性能越好。

在这组实验中，本文的主要比较对象是任务调度系统的不同实现与不同的任务种类。因此，本项目设置了三个组别，比较它们的各项测试指标：第一组使用未修改的ArceOS和QEMU模拟器，创建线程计算；第二组使用本项目修改后的系统，即ArceOS-QEMU-ATS-INTC，创建线程计算；第三组使用ArceOS-QEMU-ATS-INTC，创建协程计算。

除了主要的比较对象以外，实验还更改了为QEMU模拟器分配的CPU核心数量和测试程序创建的任务数量作为变量，从而测试不同环境下，各个组别的性能变化与差异。其余的因素，例如随机数组的大小、随机数组的生成种子、线程/协程内的计算算法等，均作为无关变量，在整个实验过程中保持不变。

基于以上的设计，本项目设计了三项实验。第一项实验固定CPU核心数为4，改变创建的任务数量；第二项实验固定任务数量为60，改变CPU核心数；第三项实验固定CPU核心数为4、创建的任务数量为60，进行多次实验，获得计算时间的分布情况。第一项、第二项实验中，在每一个相同条件下测试了5组数据，取它们的平均值和中位数作为该条件的数据点，从而减小偶然因素。在第三组实验中，在每一个相同条件下测试了40组数据，以观察该条件下，数据的分布情况。

6.2.2 实验结果与分析

在实验的全过程中，本项目实现的系统始终给出了正确的结果，这可以说明该系统的正确性。

在第一项固定CPU核心数，改变创建任务数量的实验中，各个组别的计算时间如图6-1所示。

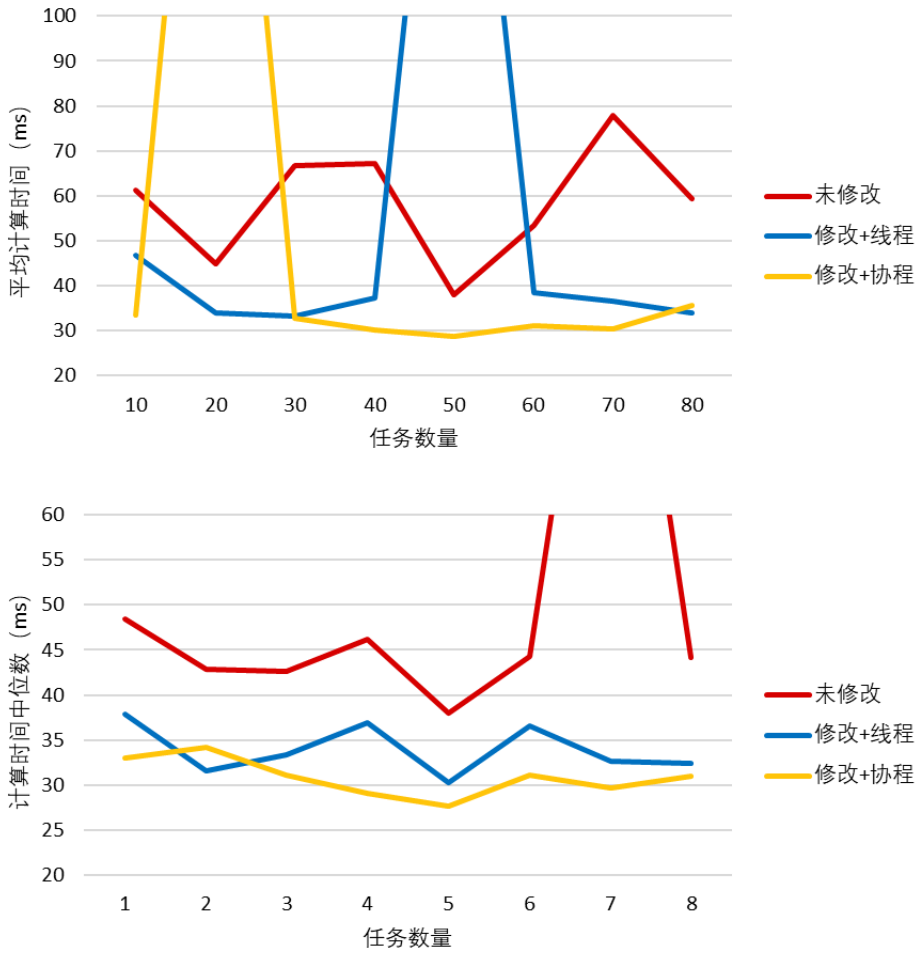

图 6-1 4CPU 核心下，计算时间随任务数量变化折线图

在测试中，大部分情况下，各个组别在各个条件下的计算时间均在50ms以下。然而，偶尔会出现计算时间高于100ms，甚至高于500ms的情况。这些偶尔出现的高时延使平均计算时间波动幅度较大，难以总结规律。因此又计算了各组数据的中位数。

由于偶发高时延的出现，图中的一些数据点的数值远高于其它数据点。为了更好展示正常情况下的数据变化，本文根据正常情况下数据的波动范围设置纵坐标轴的取值范围。这导致一些异常数据点的数值超出了显示范围（例如，在图6-1的平均计算时间折线图中，纵轴最大为100ms，而一些异常数据点达到了200ms以上），从而使一些变化曲线在图中显示为断开的折线。该情况在之后的几幅图中也有出现。

从平均值看，未修改的系统的计算时间波动较小。原数据中，其偶尔会出现高于

100ms的计算时间，但高于500ms的未出现过。该情况可能是由于多个CPU核心在同一时间申请全局调度器锁导致的。ArceOS-QEMU-ATS-INTC的测试数据中，有较低概率出现高于500ms的计算时间，该高时延情况的出现频率小于未修改的系统。因为本项目的任务调度机制只在调度器硬件驱动上加锁，全局锁更精细，CPU核心保持锁的时间更短。如果高时延情况由全局锁引起，则不仅出现频率更低，还应该具有更低的计算时间。因此，目前还未找出该高时延情况出现的原因。此外，从平均值的变化曲线还可看出，如果未出现高时延的情况，则本项目实现的任务调度机制有更低的计算时间。

由于中位数滤除了偶发的高时延的影响，从中位数曲线能更明显的看出正常情况下，各个组别的性能对比：一方面，ArceOS-QEMU-ATS-INTC的性能表现优于原系统。其原因可能有硬件实现任务调度的因素，也可能因为原系统任何的任务调度操作都需要全程持有调度器锁，而ArceOS-QEMU-ATS-INTC的各个CPU核心可以独立完成调度操作，只在调用硬件调度器驱动时需要申请和持有锁。另一方面，使用协程计算在大部分情况下优于使用线程计算。这可以证明协程的上下文切换开销低于线程。

在第二项固定创建任务数量，改变CPU核心数的实验中，各个组别的计算时间如图6-2所示。

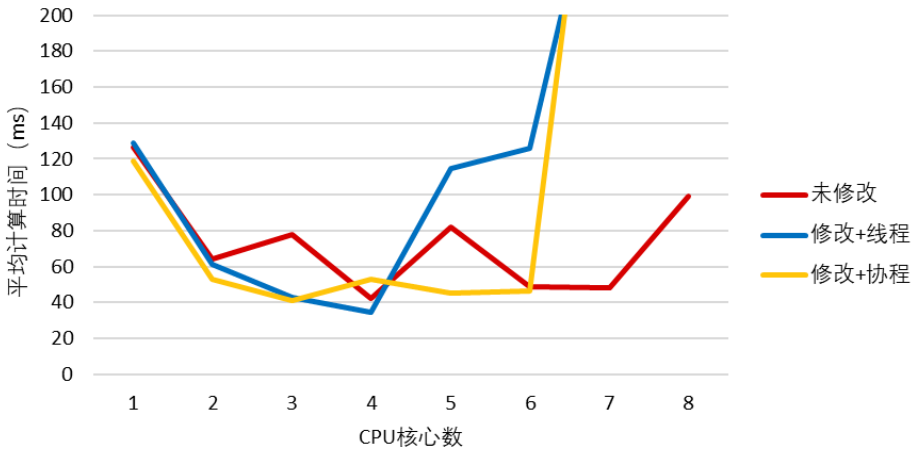

图 6-2 60 任务数量下，计算时间随 CPU 核心数变化折线图

从图中可知，在核心数为1-4时，执行时间随CPU核心数增大而减小，符合预期。本段的大致情况也是ArceOS-QEMU-ATS-INTC优于原系统、协程优于线程。与上个

实验的结果相同。

核心数>5时，各个组别的处理时间均有升高，可能是因为Linux虚拟机只有4个核心，其需要为QEMU模拟出多于4个核心，就在Linux系统中引入了任务切换开销。ArceOS-QEMU-ATS-INTC受该升高的影响更大，可能是因为ArceOS-QEMU-ATS-INTC在没有任务时，CPU也会忙等，从而在Linux虚拟机算力有限的情况下，挤占了有效任务的执行时间。

在第三项测量计算时间分布的实验中，设置任务数量为60，CPU核心数为4。各个组别的计算时间分布如图6-3所示。

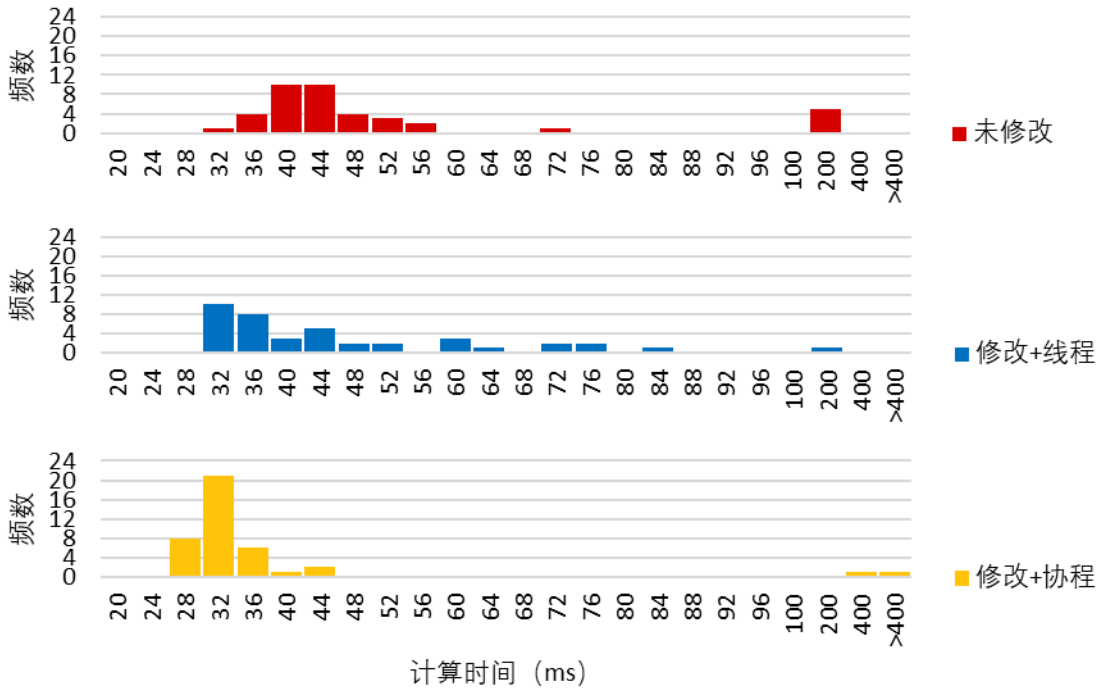

图 6-3 4CPU 核心、60 任务数量下的计算时间分布图

从图中可以看出，从总体计算时间上衡量系统的性能，则使用协程计算的ArceOS-QEMU-ATS-INTC最优，使用线程计算的ArceOS-QEMU-ATS-INTC次之，未修改的系统最差。同时，使用协程计算的ArceOS-QEMU-ATS-INTC的计算时间分布最为集中。最后，三个组别均出现了偶发的高时延情况，本文已在上文中尝试分析了其原因。

6.3 网络实验

6.3.1 实验设计

本实验测试系统的网络性能，主要涉及网络模块，但也与任务调度模块有关。本实验中，在ArceOS上运行的应用为示例应用apps/net/echoserver，其会运行一个TCP服务器，接收连接并创建线程来处理连接。之后，本项目编写了一个用于发出TCP请求、接受响应的程序，并在该程序中测量ArceOS对TCP请求的响应时间。ArceOS在Linux虚拟机的QEMU中运行，本项目编写的TCP客户端直接在Linux虚拟机上运行。它们通过QEMU的端口转发机制相互通信。

在本实验中，一方面观察ArceOS上运行的网络服务器的行为，从而验证系统的正确性；另一方面使用客户端程序测量响应时间，并以此衡量系统的性能。

实验的主要比较对象是系统的不同实现。如3.3.2和5.2节所述，本项目使用两种方案修改网络：新增专门的轮询任务，或者由网卡中断触发poll_interface函数的调用。两种方案都可以由线程或协程实现，因为轮询任务和中断处理程序可以实现为线程或协程。因此，本项目设置了六组不同的实现相互对比：第一组，未经过任何修改的系统；第二组，仅修改了任务调度模块的系统；第三到第六组进行了上文提到的所有修改，它们的网络模块分别选用了线程实现的轮询任务、协程实现的轮询任务、线程实现的中断处理例程、协程实现的中断处理例程这四种修改方案。

本文对这些不同的实现开展了三项实验，三项实验均为QEMU分配4个CPU核心。

第一项和第二项实验的测试对象均是低负载下的响应时间。TCP客户端会不断重复“建立TCP连接——发送数据包——接收服务器返回的数据包——关闭连接”的过程，且在连接关闭后才会建立下一个连接。第一项实验中比较不同的包大小对响应时间的影响。因此，本文测量不同包大小下，不同实现的响应时间，每组测量5次，取平均值和中位数。第二项实验中测量响应时间的分布情况。因此，本项目固定包大小为1024字节，对每种实现测量40次响应时间，统计分布情况。

第三项实验测试不同负载下的吞吐量。因此，本项目使TCP客户端一次性建立多条连接，统计服务器将它们全部处理完成的时间。之后，使用每秒处理的连接数这一指标表示吞吐量。该实验的变量为同时建立的连接数，本项目测量不同连接数，不同实现的处理时间，计算这些情况下吞吐量。同样每组测量5个数据，取平均数和中位数。

6.3.2 实验结果与分析

在实验运行过程中，本项目修改后的系统上的服务器的行为均正常，没有出现死锁或意料之外的连接断开。

第一项测量包大小与响应时间的关系的实验，其结果如图6-4所示。

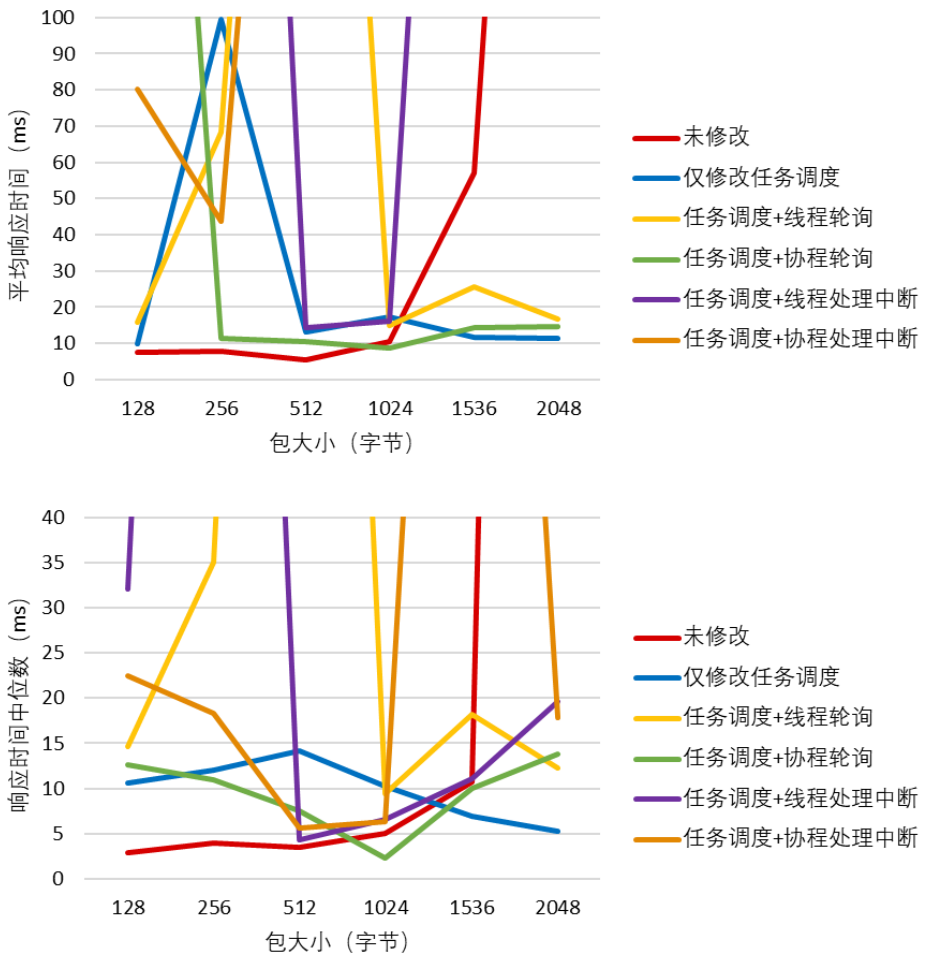

图 6-4 响应时间随包大小变化折线图

由实验结果看出，各种实现均出现了较大的波动情况。实验数据表明，正常情况下，响应时间小于50ms。偶发的高延迟大于100ms，甚至出现了大于1000ms的情况。同时，偶发的高延迟出现频率较大，以至于中位数也无法完全排除高延迟情况。由于引入了网络协议栈，因此出现该波动的原因更为复杂，不过任务调度模块的响应时间波动可能是原因之一。

总体上看，在排除了偶发高延迟的情况后，各个实现的情况呈现以下特点：未修

改的原系统的响应时间受包大小的影响较为明显，在包大小增加时，出现了明显的延迟增长情况。而修改后的系统的响应时间则与包大小的相关性不高。

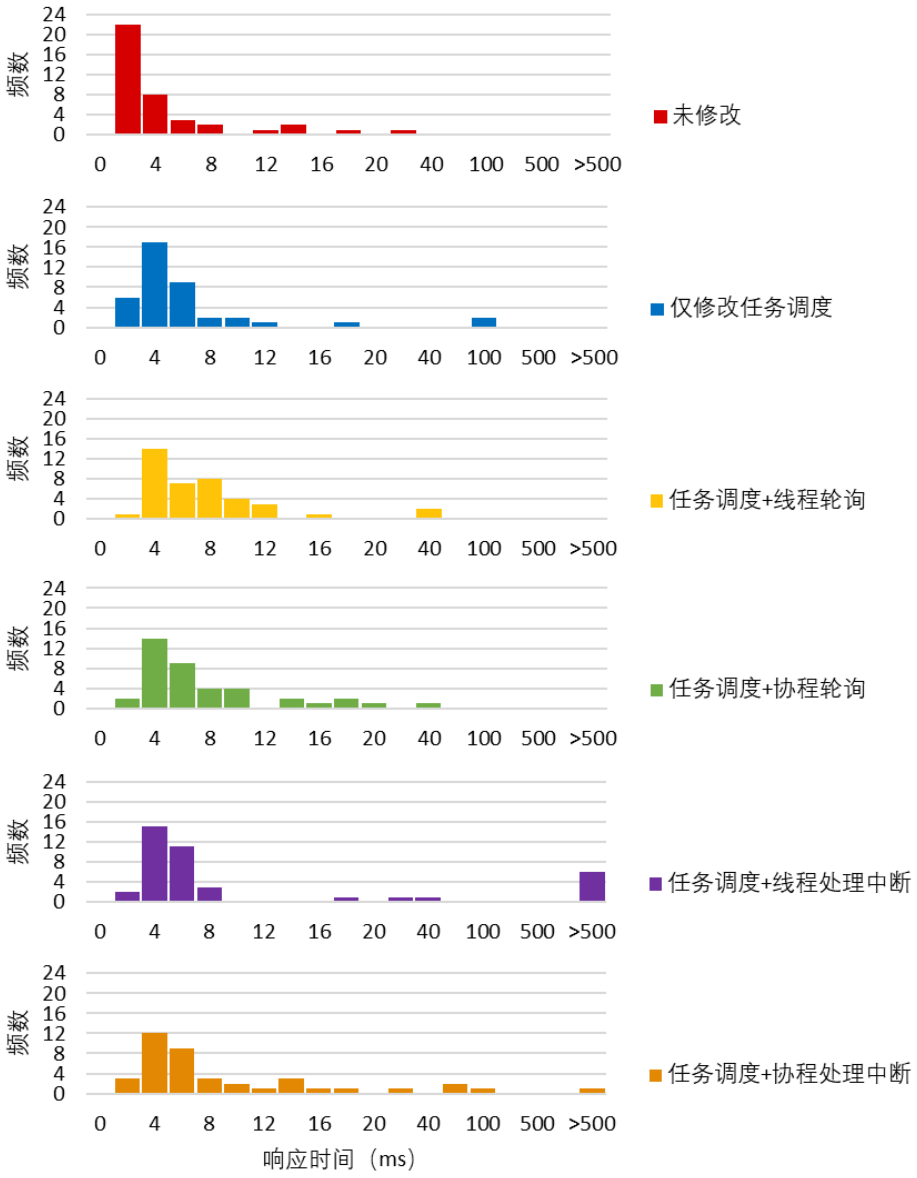

图 6-5 包大小 1024 字节的响应时间分布图

对比各种实现的响应时间，可以得出以下特点：若数据包较小，原系统的性能表现最好。若数据包较大，仅修改任务调度的系统表现最好，其次是两种使用专门的轮询任务的系统。同时，相同修改方案下的线程实现和协程实现，其响应时间的高低关系并无明显规律。其可能的原因是，poll_interface函数的调用占用了响应时间的较大

比例，导致线程或协程的上下文切换时间相较之下占比较低。

测量各种实现的响应时间分布的实验，结果如图6-5所示。

从图中可知，该情况下的响应时间分布规律为：原系统的响应时间最短，其次是仅修改任务调度的系统。最后是四种优化过网络模块的系统，它们之间相差不大。出现这一结果的可能原因是：在测试的低负载、低并发情景中，每个TCP连接自己调用poll_interface函数的开销不大，且不会出现导致性能下降的并发调用情况。而使用专用轮询任务和中断的实现，因为引入了额外任务，反而增加了开销。

测量不同负载情况下的吞吐量的实验结果如图6-6所示。

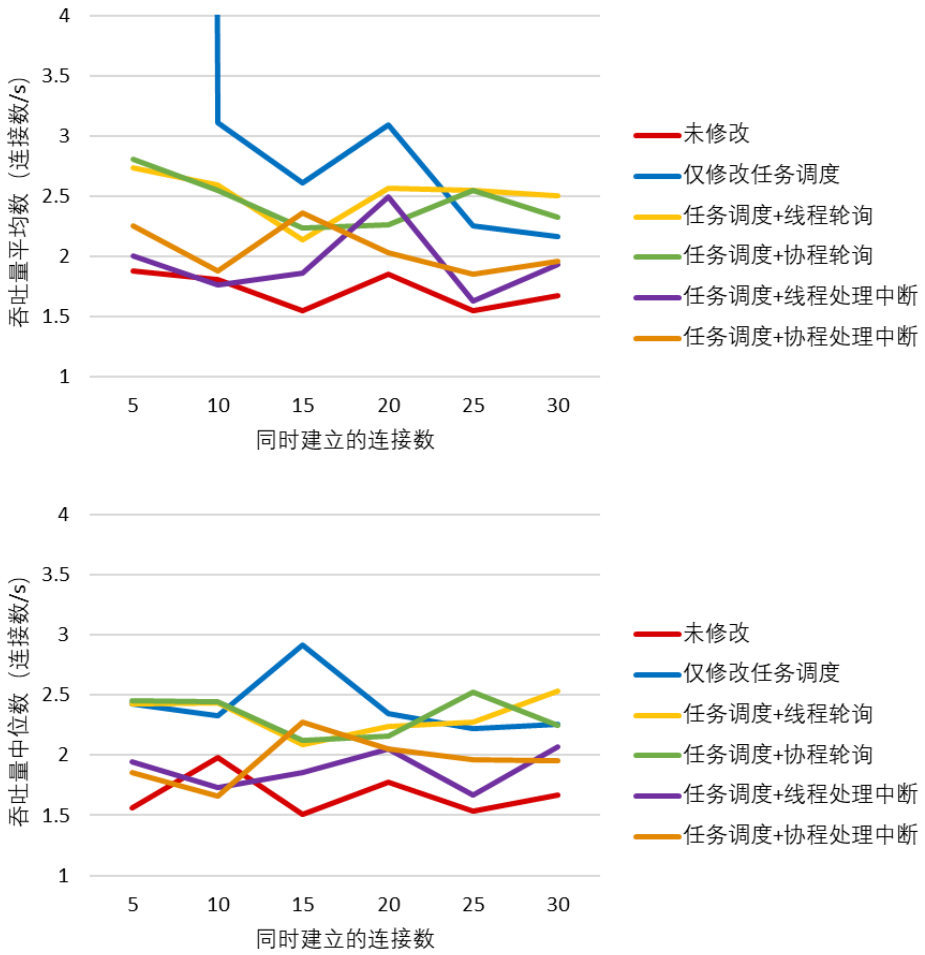

图 6-6 同时建立不同数量的连接下的吞吐量变化图

从图中可知，在同时处理多条连接时，各种修改后的系统都开始在性能方面占优。在连接数较少时，吞吐量最高的为仅修改任务调度的系统。连接数较多时，吞吐

量最高的为两个使用了专门轮询任务的系统。其次是仅修改任务调度的系统。两个使用中断的系统吞吐量较低，但仍高于原系统。

本情况出现的可能原因有以下几点：首先，同时建立多条连接使服务器同时运行多个线程（且线程数大于CPU核心数），使任务调度模块在服务器的运行中起到重要作用。其次，随着连接数的增长，原本的网络模块中，对poll_interface的过度调用和并发调用问题越发严重，导致了性能的下降。相较之下，修改后的网络模块的优势越发明显。最后，专用的轮询任务使网卡每次接收到数据包即可马上传送到各个socket处理，在处理速度上本身具有优势；而基于中断的系统从设计上就无法达到轮询的性能，尤其是在QEMU的模拟下，硬件调度器的工作也需要使用Linux虚拟机的CPU，从而挤占了ArceOS的CPU时间，从而进一步降低了中断模式的服务器的吞吐率。

6.4 Redis 实验

6.4.1 实验设计

在本组实验中，本项目使用Redis服务端作为测试程序。Redis[19]是一个基于内存的NoSQL数据库软件，已被广泛应用于许多涉及NoSQL数据库的软件项目中。与使用人工编写的测试程序的前两个实验不同，本实验中选择Redis的目的是测量本项目实现的系统在真实负载下的性能表现。

通过[20]的工作，Redis服务端已被移植到ArceOS中。因此，原本的ArceOS系统和本项目修改后的系统都可以直接运行Redis服务端。

本实验的主要比较对象仍是不同的系统实现。因此，本项目设置了和6.3.1节相同的六组系统实现参与实验：未经过任何修改的系统、仅修改了任务调度模块的系统，以及修改了任务调度模块和网络模块的四种实现：线程实现的轮询任务、协程实现的轮询任务、线程实现的中断处理例程、协程实现的中断处理例程。

本实验的测试指标是不同命令的执行速率。实验中，先在各种系统实现上运行Redis服务端。其会接收客户端的命令并完成相应的数据库操作。之后，在Linux上运行客户端测试程序redis-benchmark，其会分别连续发出各种命令并统计服务器的响应时间，从而计算服务器执行该命令的速率。本实验控制其它无关变量不变，QEMU的CPU核心数固定为4，redis-benchmark的执行参数统一为：30并行度，每个命令发送30个请求。

6.4.2 实验结果与分析

redis-benchmark测量出的各种命令执行速率如图6-7所示。

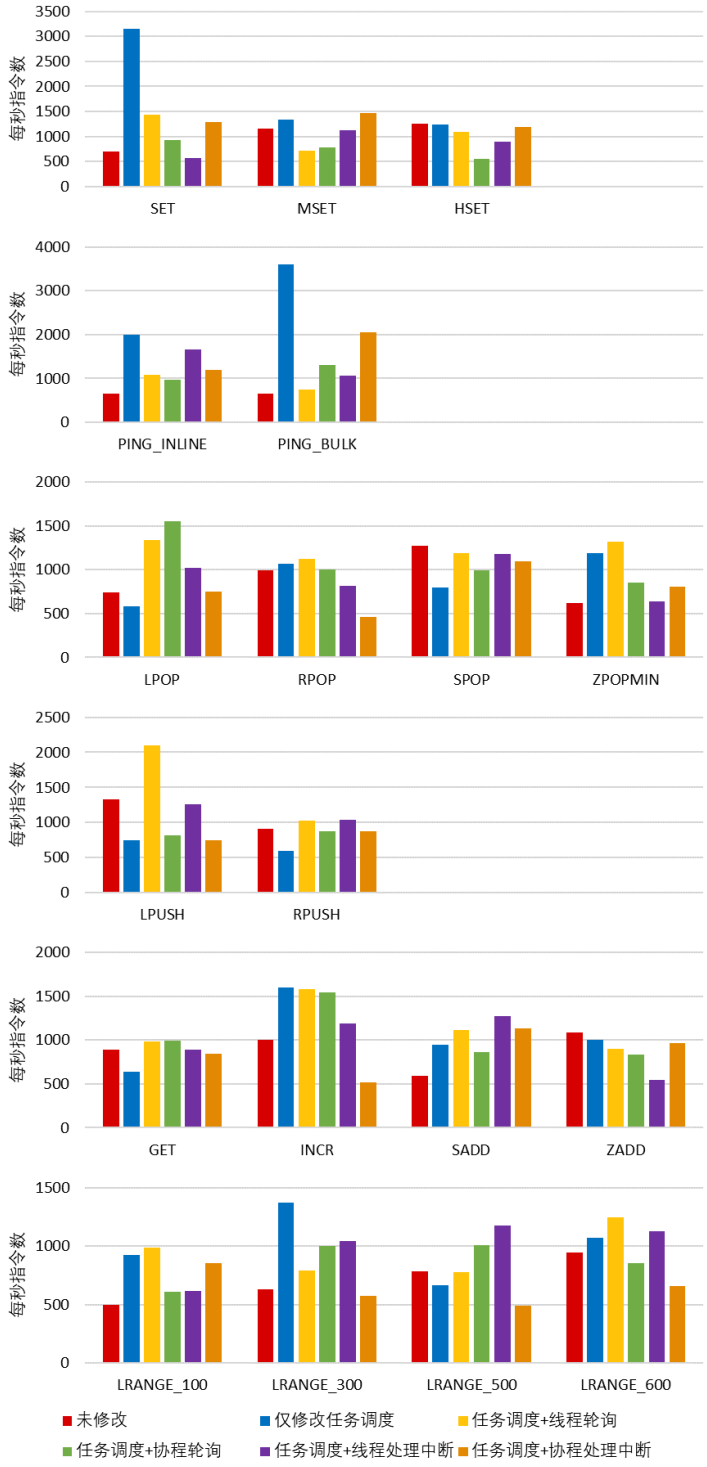

图 6-7 各种命令的执行时间

比较各种系统的执行速率，可以得到以下特点：将原系统与修改后的系统比较，则大部分情况下，修改后的系统执行速率更高。将仅修改任务调度的系统、基于专用轮询任务的系统和基于中断的系统相比较，则对于SET、PING等较简单的命令，仅修改任务调度的系统执行速率最高；对于PUSH、POP等较复杂的命令，基于轮询任务的系统执行速率最高。将线程实现与协程实现相比较，两者的执行速率在不同情况下有不同的高低关系。

在使用redis-benchmark测试的过程中，由于该测试程序在测试过程中也会输出实时计算的系统执行速率结果，因此观测到了如下现象：在任务数量较多、并行度较高的情况下，可能会出现某个任务的执行速率持续单调下降，甚至降到低于1命令/秒（正常情况的执行速率约为500~2000命令/秒），随后服务器关闭连接，测试中止。实验中涉及的各种实现方案都可能出现该情况。不过，未修改网络模块的系统（原系统和仅修改任务调度模块的系统）出现该情况的概率更大。修改网络模块的四种系统，出现该现象的概率更小，且在更高的任务数量、并行度的参数下才会观测到该现象。该现象的可能原因是：命令发送的速度高于服务器处理命令的速度，导致服务器同时处理数量逐渐增多的命令，从而使处理速率逐渐下降。在未修改网络模块的系统中，处理数量逐渐增多的命令会导致poll_interface函数的过度调用和并发调用，从而影响处理命令的速率。已修改网络模块的系统虽然避免了该情况降低的处理速率，但仍会因为命令数量的增多而加重任务调度模块的负担。因此，所有的实现都会受到该情况影响，但修改了网络模块的系统受到的影响更低，可以承载更高的负载、更高的并行度。该现象也说明了本项目对网络模块修改的作用。

6.5 本章小结

本章介绍了本项目实现的系统，ArceOS-QEMU-ATS-INTC的实验方案和实验结果。本章设计了三组实验：在任务调度实验中，实验结果表明本项目的系统相较于原系统，有更高的任务调度性能，也证明了协程的上下文切换开销低于线程。在网络实验中，结果显示，在低负载情况下，原系统的性能表现更好；在高负载情况下，本项目实现的系统性能表现更好。在Redis实验中，结果显示，在真实负载下，本项目实现的系统在大部分情况下具有更优的性能，而且可以承受更高的负载。

结　论

本文在QEMU模拟器中实现了一种硬件协程调度器与中断处理器QEMU-ATS-INTC，其可以加速协程调度，并且简化了外部中断的处理流程。本文修改了在QEMU模拟器中运行的ArceOS系统，使其具备利用该硬件进行协程调度和外部中断处理的功能，并优化了其网络模块的实现，得到ArceOS-ATS-INTC系统。实验结果表明，该机制提升了软硬件系统的任务调度性能，且优化了软硬件系统在高负载下的网络表现。

本研究的创新点为：将协程调度与中断处理结合，使用协作式调度的方法实现了外部中断处理的过程，从而既降低了中断处理的开销，又提高了中断处理与协程调度的亲和度。适合本研究的应用场景为高性能网络服务器等高并发、I/O密集的场景，因为这样的场景会进行大量的任务调度、阻塞和唤醒，从而增大采用协程的优势；并且I/O设备也可以利用本研究的中断机制，开销更低地唤醒协程。因此，本研究可以降低服务器在任务切换中花费的计算资源，提高服务器的效率和资源利用率。

未来，本研究可以在这些方向上继续推进：首先，增加该硬件可处理的中断类型，例如处理核间中断，进而为本硬件增加管理进程间通信的功能；其次，目前实现的协作式调度机制的性能和实时性仍会受到占据CPU过长时间的任务的影响，未来可以研究减弱或消除该影响的方法，例如在编译期强制插入让出CPU的代码；最后，可以在操作系统层面更广泛地利用该硬件提供的功能和性能优势，例如，将系统内核的更多部分改造为协程实现。

参考文献

[1] 王润基. rCore OS 开发文档[EB/OL]. [2019-01-22/2019-12-22]. http://rcore.gitbook.io
[2] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, 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, https://www.usenix.org/conference/nsdi19/presentation/kaffes
[3] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks[C]. the 26th Symposium on Operating Systems Principles (SOSP '17). Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780
[4] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling[C]. the 29th Symposium on Operating Systems Principles (SOSP '23). Association for Computing Machinery, New York, NY, USA, 466–481. https://doi.org/10.1145/3600006.3613136
[5] 曾素华,蒋建春. OSEK 操作系统抢占式调度策略改进[J]. 计算机工程与应用, 2010, 46(34):228-231. DOI:10.3778/j.issn.1002-8331.2010.34.068.
[6] 钱宏文,张飞,吴翼虎,等. 局部动态可重构 FPGA 进程式调度系统设计与实现[J]. 电子技术应用,2023,49(3):114-117. DOI:10.16157/j.issn.0258-7998.222818.
[7] 尹震宇,赵海,等. 一种面向硬件线程的实时调度算法研究与设计[J]. 电子学报, 2007,35(8):1467-1471. DOI:10.3321/j.issn:0372-2112.2007.08.009.
[8] Jerry D. Erwin and E. Douglas Jensen. 1970. Interrupt processing with queued content-addressable memories[C]. the November 17-19, 1970, fall joint computer conference (AFIPS '70 (Fall)). Association for Computing Machinery, New York, NY, USA, 621–627. https://doi.org/10.1145/1478462.1478553
[9] Jupyung Lee and Kyu Ho Park. 2010. Interrupt handler migration and direct interrupt scheduling for rapid scheduling of interrupt-driven tasks[J]. ACM Trans. Embed. Comput. Syst. 9, 4, Article 42 (March 2010), 34 pages. https://doi.org/10.1145/1721695.1721708
[10] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system[C]. the 2009 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '09). Association for Computing Machinery, New York, NY, USA, 167–174. https://doi.org/10.1145/1629395.1629419
[11] 杨媛媛,王晓华,李敏,等. 基于 Xilinx FPGA 的中断处理[J]. 电脑知识与技术,2021,17(5):244-245.
[12] 舒生亮,孙永节,万江华. Matrix DSP 中断处理系统的设计与实现[J]. 计算机工程与科学,2012,34(1):64-68. DOI:10.3969/j.issn.1007-130X.2012.01.011.
[13] 张旭,顾乃杰,苏俊杰.一种 Linux 用户态实时多任务调度框架[J].中国科学技术大学学报,2017,47(08):635-643.
[14] 田倬璟，黄震春，张益农. 云计算环境任务调度方法研究综述[J]. 计算机工程与应用, 2021, 57(2): 1-11.

[15] 赵方亮. the overall description documents of ATS-INTC[EB/OL]. https://ats-intc.github.io/docs/
[16] Bojie Li, Zihao Xiang, Xiaoliang Wang, Han Ruan, Jingbin Zhou, and Kun Tan. 2023. FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA[C]. the 7th Asia-Pacific Workshop on Networking (APNET '23). Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3600061.3600063
[17] QEMU 项目开发者. QEMU 项目文档[EB/OL] . https://www.qemu.org/docs/master/about/index.html
[18] ArceOS 项目开发者. ArceOS 项目[CP/OL]. https://github.com/rcore-os/arceos
[19] Redis 项目开发者. Redis 项目文档[EB/OL]. https://redis.io/docs/latest/get-started/
[20] 晏巨广. Redis on ArceOS[EB/OL]. https://github.com/syswonder/report/blob/main/docs/2023/20230730_Redis_On_ArceOS.md