注：此处无需更改。

阅后删除此文本框。



**本科生毕业设计(论文)**

**北京理工大学本科生毕业设计（论文）题目**

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

注：此处是论文中英文题目，中文题目，居中，字体：华文细黑，加黑，字号：二号，行距：多倍行距1.25，间距：段前、段后均为0行，取消网格对齐选项。英文题目，与中文题目对应，居中，字体：Times New Roman，字号：三号，加黑，行距：多倍行距1.25，间距：段前、段后均为0行，取消网格对齐选项。阅后删除此文本框。

|  |  |
| --- | --- |
| 学 院： |  |
| 专 业： |  |
| 学生姓名： |  |
| 学 号： |  |
| 指导教师： |  |

注：此处按照实际情况填写即可。打印（宋体，三号）或手写都可以。阅后删除此文本框。

20XX 年 月 日

**原创性声明**

注：页眉内容无需更改。阅后删除此文本框。

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

特此申明。

本人签名： 日 期： 年 月 日

**关于使用授权的声明**

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的,复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名： 日 期： 年 月 日

指导老师签名： 日 期： 年 月 日

**北京理工大学本科生毕业设计（论文）题目**

注：此处是中文题目，居中，字体：黑体，加黑，字号：小二，行距：单倍行距，间距：段前、段后均为1行，取消网格对齐选项。阅后删除此文本框。

摘　要

注：此处无需更改。阅后删除此文本框。

本文……。

摘要正文选用模板中的样式所定义的“正文”，每段落首行缩进2个字符；或者手动设置成每段落首行缩进2个汉字，字体：宋体，字号：小四，行距：固定值22磅，间距：段前、段后均为0行。阅后删除此段。

摘要是一篇具有独立性和完整性的短文，应概括而扼要地反映出本论文的主要内容。包括研究目的、研究方法、研究结果和结论等，特别要突出研究结果和结论。中文摘要力求语言精炼准确，本科生毕业设计（论文）摘要建议300-500字。摘要中不可出现参考文献、图、表、化学结构式、非公知公用的符号和术语。英文摘要与中文摘要的内容应一致。阅后删除此段。

**关键词：北京理工大学；本科生；毕业设计（论文）**

注：此处字体：黑体、小四号、加粗；

一般选3-8个单词或专业术语，且中英文关键词须对应。

阅后删除此文本框。

注：页脚内容为页码，宋体、五号，居中排列。阅后删除此文本框。

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

注：此处是英文题目，居中，字体：Times New Roman，加黑，字号：三号，行距：单倍行距，间距：段前、段后均为1行，取消网格对齐选项。阅后删除此文本框。

Abstract

注：此处无需更改。阅后删除此文本框。

In order to study……

Abstract正文设置成每段落首行缩进2字符，字体：Times New Roman，字号：小四，行距：固定值22磅，间距：段前、段后均为0行。阅后删除此段。

**Key Words: BIT； Undergraduate； Graduation Project (Thesis)**

注：Key Words与摘要正文之间空一行。Key Words与中文“关键词”一致（3-8个）。词间用分号间隔，末尾不加标点；Times New Roman，小四，加粗。

阅后删除此文本框。

目　录

注：在目录页面中点击鼠标右键，选择“更新域”，在弹出窗口中选择“更新整个目录”，确定即可自动生成目录。章、节标题和页码，字体：宋体，字号：小四，不加粗。阅后删除此文本框。

注：此处无需更改。阅后删除此文本框。

[摘　要 I](#_Toc165833259)

[Abstract II](#_Toc165833260)

[第1章 绪论 1](#_Toc165833261)

[1.1 研究背景和意义 1](#_Toc165833262)

[1.2 国内外研究现状 2](#_Toc165833263)

[1.2.1 任务调度机制 2](#_Toc165833264)

[1.2.2 中断机制 3](#_Toc165833265)

[1.3 研究内容和关键问题 4](#_Toc165833266)

[1.3.1 研究主要内容 4](#_Toc165833267)

[1.3.2关键问题 5](#_Toc165833268)

[1.4 全文组织结构 6](#_Toc165833269)

[第2章 相关技术介绍 7](#_Toc165833270)

[2.1 任务调度策略 7](#_Toc165833271)

[2.1.1 概述 7](#_Toc165833272)

[2.1.2 性能指标 7](#_Toc165833273)

[2.1.3协作式调度和抢占式调度 8](#_Toc165833274)

[2.2 Rust语言的协程实现 9](#_Toc165833275)

[2.2.1 协程概述 9](#_Toc165833276)

[2.2.2 Rust语言中的协程 9](#_Toc165833277)

[2.3 本章小结 11](#_Toc165833278)

[第3章 设计 12](#_Toc165833279)

[3.1 整体结构 12](#_Toc165833280)

[3.2 QEMU层的设计 13](#_Toc165833281)

[3.2.1 lite\_executor硬件 13](#_Toc165833282)

[3.2.2 virt设备 15](#_Toc165833283)

[3.3 操作系统层的设计 15](#_Toc165833284)

[3.3.1 axtask模块 15](#_Toc165833285)

[第1章 一级题目 17](#_Toc165833286)

[1.1 二级题目 17](#_Toc165833287)

[1.1.1 三级题目 17](#_Toc165833288)

[结　论 19](#_Toc165833289)

[参考文献 20](#_Toc165833290)

[参考文献 21](#_Toc165833291)

[附　录 23](#_Toc165833292)

[致　谢 24](#_Toc165833293)

# 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，因此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作出反应（时钟中断），可以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（开题报告）]。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（开题报告）]。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[6]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（开题报告）]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。不过，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（开题报告）]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，然而无法完全消除。

在国内，主要有以下研究和改进：曾素华等人[5（开题报告）]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（开题报告）]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[7（开题报告）]等人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切换开销和高调度频率。

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[1]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[3（开题报告）]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[2]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[3]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[4]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响应优先级最高的、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延时开销降低到仅4个CPU周期。张旭等人[6]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。

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模拟器中设计并实现基于硬件的共享调度器，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，添加基于该调度器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；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，使其能够利用我们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。

本研究之后对修改后的软硬件系统进行性能测试，同时与未修改的系统进行比较。测试结果表明，在低负载情况下，修改后的系统与修改前性能相差不大；在高负载情况下，修改后的系统优于修改前。

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Rust语言中的协程。

本文的第3章描述了本研究实现的系统的设计，包括总体架构、QEMU模拟硬件的设计、操作系统层次的设计。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，分析导致该结果的原因。

本文的第7章总结了全文的内容，分析优点与不足，展望了本研究在未来可能的改进方向。

# 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，从而确定重新调度的触发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后一个方向的内容。

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[7]：

* 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，吞吐量是系统最重要的性能指标。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（开题报告）]的研究针对尾部时延进行了优化，而[4（开题报告）]的研究在尾部时延达到要求的条件下提升系统的吞吐量。

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/终止状态自己让出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，除了任务自己让出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。

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coroutine、Rust语言的async/await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语言实现的协程是使用async/await模型的无栈协程，因此采用协作式调度，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，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枚举有两项元素，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。每一棵树就是一个协程。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方法来运行协程。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Context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会调用相应Context中的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，我们可以描述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。

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。

# 第3章 设计

本项研究通过对QEMU模拟器和ArceOS操作系统两个层面的修改，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。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，以及在ArceOS操作系统中修改的任务调度模块和网络模块。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：

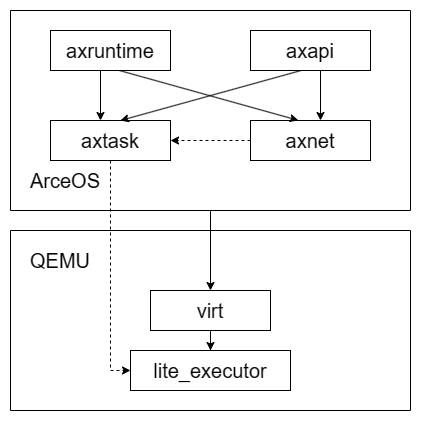


图3-1 本项目模块的关系图

在QEMU模拟器中，我们添加了自定义的硬件lite\_executor。该硬件结合了中断处理和协程调度的功能，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。我们通过修改virt设备的代码，将实现的lite\_executor硬件加入virt设备中。

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模块负责实现任务调度功能，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。它通过virt设备间接调用了lite\_executor硬件，使用了它的任务调度和中断处理功能，将其作为任务的就绪队列，降低了调度算法的运行开销。axnet模块实现网络功能，我们基于修改后的axtask模块提供的协程和中断功能，优化了axnet模块的网络实现。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。

本项目设计的系统的启动流程如下：

1. QEMU模拟器启动，在启动virt虚拟设备的过程中启动了lite\_executor硬件。
2. 加载系统映像，在virt虚拟设备中启动ArceOS操作系统。
3. ArceOS操作系统从axruntime模块开始执行，初始化axtask、axnet等各个模块。
4. 各模块初始化完成后，axruntime模块启动用户程序。

本项目设计的系统的运行流程如下：

1. 应用程序运行过程中，若调用了系统提供的api，则执行流进入系统。
2. 通过axapi模块，将应用程序调用的api分配到系统内各个模块执行。
3. 如果api调用了axtask模块，则该模块进行相关的任务调度操作，并在需要时调用lite\_executor硬件。
4. 调用lite\_executor硬件时，宿主机CPU执行的内容从模拟器中的ArceOS系统退出到QEMU模拟器中，并执行lite\_executor硬件的相关代码。
5. 各层次的调用逐层返回结果，直到返回应用程序继续执行，回到第1步。

3.2 QEMU层的设计

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：

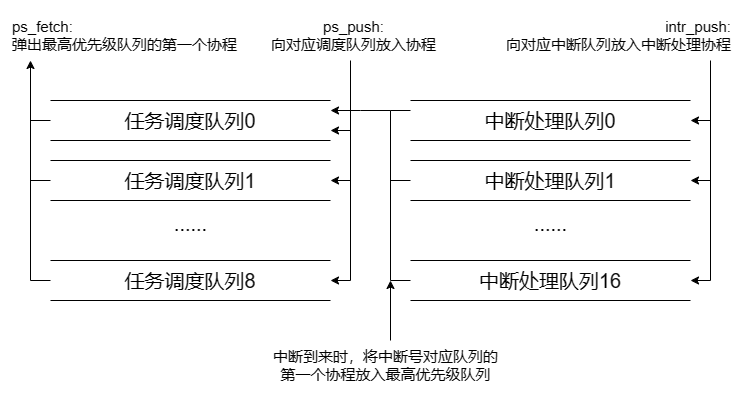


图3-2 硬件队列示意图

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，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。

3.2.2 virt设备

从设计层面描述对virt设备的修改比较简单。本项目将lite\_executor硬件加入到该设备上、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。

3.3 操作系统层的设计

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：

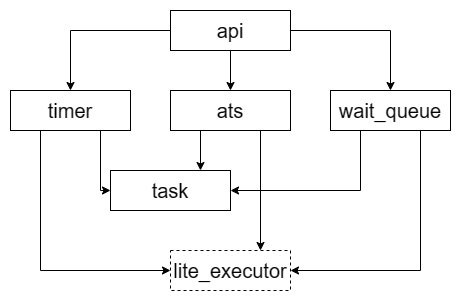


图3-3 任务调度模块关系图

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，因此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，，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、用async签名的函数。

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，注册中断处理程序的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。

我们对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。该方案参考了文献[8]，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。我们可以利用这一信息，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，能够降低反应时间。

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。

3.3.3 axruntime和axapi模块

从设计角度描述对这些模块的修改很简单，就是在axtask和axnet模块修改后，对axruntime和axapi模块做出适应性修改。

3.4 本章总结

本章介绍了本项目的设计，先从总体角度描述了各个模块的关系和执行流程，在分别从硬件层次和软件层次描述了详细设计。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 模拟硬件层次的实现

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[9]。其具有两种模拟模式：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为risc-v架构编写的ArceOS操作系统。

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，首先需要创建该硬件的类型、并实现它与外界通信的机制。如此搭建好硬件的框架代码后，才能开始实现硬件的核心功能。

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，我们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，我们实现其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表（4-1）所示：

表4-1 硬件与操作系统接口的输入输出参数

|  |  |  |
| --- | --- | --- |
| 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 |

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，我们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，我们如图（4-1）划分：

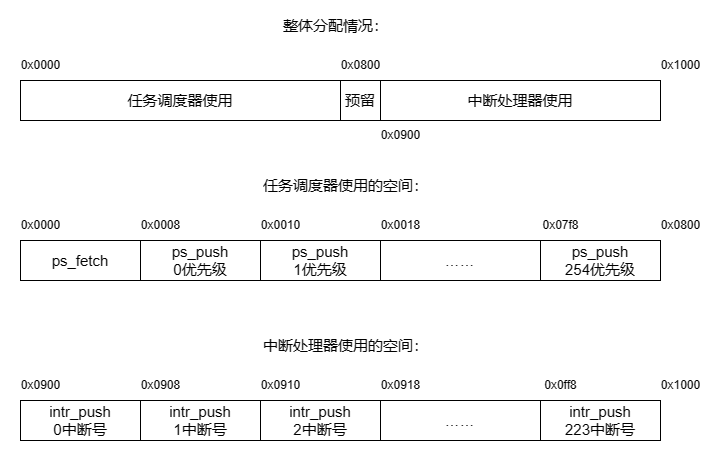


图4-1 每个进程内MMIO空间的分配情况

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，将相应指针写入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，我们将其分别命名为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。我们将该函数命名为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，我们使用队列实现任务调度和中断处理的功能。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。

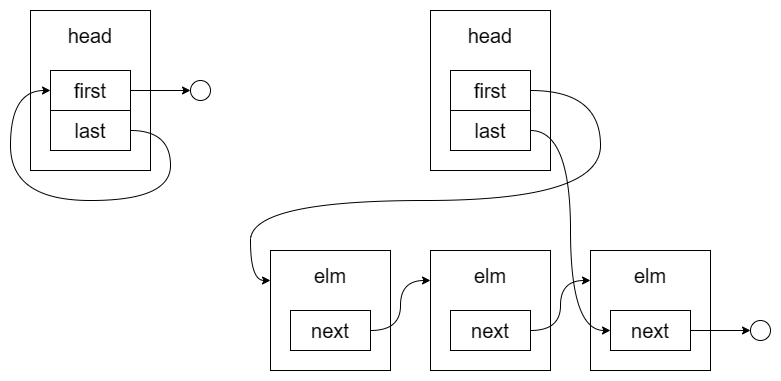


图4-2 QEMU简单队列的内存布局

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（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，在对应的内存区域处调用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。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中模拟硬件的实现。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1章 一级题目

1.1 二级题目

1.1.1 三级题目

正文……

正文部分：宋体、小四；正文行距：22磅；间距段前段后均为0行。阅后删除此段。

图、表居中，图注标在图下方，表头标在表上方，宋体、五号、居中，1.25倍行距，间距段前段后均为0行，图表与上下文之间各空一行。阅后删除此段。

注：正文页脚内容为页码，宋体、五号，居中排列。阅后删除此文本框。

图-示例：（阅后删除此段）



图1-1 标题序号

表-示例：（阅后删除此段）

表1-1 统计表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 项目 | 产量 | 销量 | 产值 | 比重 |
| 手机 | 1000 | 10000 | 500 | 50% |
| 计算机 | 5500 | 5000 | 220 | 22% |
| 笔记本电脑 | 1100 | 1000 | 280 | 28% |
| 合计 | 17600 | 16000 | 1000 | 100% |

公式标注应于该公式所在行的最右侧。对于较长的公式只可在符号处（+、-、\*、/、≤≥等）转行。在文中引用公式时，在标号前加“式”，如式（1-2）。阅后删除此段。

公式-示例：（阅后删除此段）

 (1-1)

注：此处无需更改。阅后删除此文本框。

结　论

本文结论……。

结论作为毕业设计（论文）正文的最后部分单独排写，但不加章号。结论是对整个论文主要结果的总结。在结论中应明确指出本研究的创新点，对其应用前景和社会、经济价值等加以预测和评价，并指出今后进一步在本研究方向进行研究工作的展望与设想。结论部分的撰写应简明扼要，突出创新性。阅后删除此段。

结论正文样式与文章正文相同：宋体、小四；行距：22磅；间距段前段后均为0行。阅后删除此段。

参考文献

[1] Jerry D. Erwin and E. Douglas Jensen. 1970. Interrupt processing with queued content-addressable memories. In Proceedings of 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>

[2] 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. In Proceedings of 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>

[3] 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.

[4] 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.

[5] 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.

[6] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In Proceedings of 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>

[7] 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.

[8] Bojie Li, Zihao Xiang, Xiaoliang Wang, Han Ruan, Jingbin Zhou, and Kun Tan. 2023. FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA. In Proceedings of 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>

[9]QEMU文档，https://www.qemu.org/docs/master/about/index.html

注：此部分蓝色字体为注释，阅后可删除；黑色字体为具体示例。

阅后删除此文本框。

注：此处无需更改。阅后删除此文本框。

参考文献

参考文献书写规范

参考国家标准《信息与文献参考文献著录规则》【GB/T 7714—2015】，参考文献书写规范如下：

**1. 文献类型和标识代码**

普通图书：M 会议录：C 汇编：G 报纸：N

期刊：J 学位论文：D 报告：R 标准：S

专利：P 数据库：DB 计算机程序：CP 电子公告：EB

档案：A 舆图：CM 数据集：DS 其他：Z

**2. 不同类别文献书写规范要求**

**期刊**

[序号]主要责任者. 文献题名[J]. 刊名, 出版年份, 卷号(期号): 起止页码.

[1] 余雄庆. 飞机总体多学科设计优化的现状与发展方向[J]. 南京航空航天大学学报, 2008,40(4): 417-426.

[2] Hajela P., Bloebaumj C. L., Sobieszczanski-Sobieski J. Application of Global Sensitivity Equations in Multidisciplinary Aircraft Synthesis[J]. Journal of Aircraft, 1990, 27(12):1002-110.

**普通图书**

[序号]主要责任者. 文献题名[M]. 出版地: 出版者, 出版年. 起止页码.

[3] 李成智,李小宁,田大山. 飞行之梦—航空航天发展史概论[M]. 北京: 北京航空航天大学, 2004.

[4] Raymer D. P. Aircraft Design: A Conceptual Approach[M]. Reston, Virginia: American Institute of Aeronautics and Astronautics, Inc., 2006.

**会议论文集**

[序号]析出责任者. 析出题名[A]. 见(英文用In): 主编. 论文集名[C]. (供选择项: 会议名, 会址, 开会年)出版地: 出版者, 出版年. 起止页码.

[5] 孙品一. 高校学报编辑工作现代化特征[A]. 见: 张为民编. 中国高等学校自然科学学报研究会. 科技编辑学论文集(2)[C]. 北京: 北京师范大学出版社, 1998. 10-22.

**专著中析出的文献**

[序号]析出责任者. 析出题名[A]. 见(英文用In): 专著责任者. 书名[M]. 出版地: 出版者, 出版年.起止页码.

[6] 罗云. 安全科学理论体系的发展及趋势探讨[A]. 见: 白春华,何学秋,吴宗之. 21世纪安全科学与技术的发展趋势[M]. 北京: 科学出版社, 2000. 1-5.

**学位论文**

[序号]主要责任者. 文献题名[D]. 保存地: 保存单位, 年份.

[7] 张和生. 嵌入式单片机系统设计[D]. 北京: 北京理工大学, 1998.

[8] Sobieski I. P. Multidisciplinary Design Using Collaborative Optimization[D]. United States -- California: Stanford University, 1998.

**报告**

[序号]主要责任者. 文献题名[R]. 报告地: 报告会主办单位, 年份.

[9] 冯西桥. 核反应堆压力容器的LBB分析[R]. 北京: 清华大学核能技术设计研究院, 1997.

[10] Sobieszczanski-Sobieski J. Optimization by Decomposition: A Step from Hierarchic to Non-Hierarchic Systems[R], NASA CP-3031, 1989.

**专利文献**

[序号]专利所有者. 专利题名[P]. 专利国别: 专利号, 发布日期.

[11] 姜锡洲. 一种温热外敷药制备方案[P]. 中国专利: 881056078, 1983-08-12.

**国际、国家标准**

[序号]标准代号. 标准名称[S]. 出版地: 出版者, 出版年.

[12] GB/T 16159—1996. 汉语拼音正词法基本规则[S]. 北京: 中国标准出版社, 1996.

**报纸文章**

[序号]主要责任者. 文献题名[N]. 报纸名, 出版年, 月(日): 版次.

[13] 谢希德. 创造学习的思路[N]. 人民日报, 1998, 12(25): 10.

**电子文献**

[序号]主要责任者. 电子文献题名[文献类型/载体类型]. 电子文献的出版或可获得地址(电子文献地址用文字表述), 发表或更新日期/引用日期(任选).

[14]姚伯元. 毕业设计(论文)规范化管理与培养学生综合素质[EB/OL]. 中国高等教育网教学研究, 2005-2-2.

关于参考文献的未尽事项可参考国家标准《信息与文献参考文献著录规则》（GB/T 7714—2015）

注：此处无需更改。阅后删除此文本框。

附　录

附录相关内容…

附录是毕业设计（论文）主体的补充项目，为了体现整篇文章的完整性，写入正文又可能有损于论文的条理性、逻辑性和精炼性，这些材料可以写入附录段，但对于每一篇文章并不是必须的。附录依次用大写正体英文字母A、B、C……编序号，如附录A、附录B。阅后删除此段。

附录正文样式与文章正文相同：宋体、小四；行距：22磅；间距段前段后均为0行。阅后删除此段。

注：附录和致谢页脚内容为页码，宋体、五号，居中排列。阅后删除此文本框。

注：此处无需更改。阅后删除此文本框。

致　谢

值此论文完成之际，首先向我的导师……

致谢正文样式与文章正文相同：宋体、小四；行距：22磅；间距段前段后均为0行。阅后删除此段。