北京理工大學

本科生毕业设计（论文）

跨操作系统的异步块设备驱动模块设计与实现

Design and Implementation of a Cross-Operating System Asynchronous Block Device Driver Module

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：07112005

学生姓名：董若扬

学　　号：1120202944

指导教师：陆慧梅

2024 年 5 月 30 日

原创性声明

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

特此申明。

本人签名：　　日 期：　　年　月　日

关于使用授权的声明

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

本人签名：　　日 期：　　年　月　日

指导老师签名：　　日 期：　　年　月　日

跨操作系统的异步块设备驱动模块设计与实现

摘　要

许多面向教学的操作系统需要在多种开发板上部署。这一部署过程通常要求在操作系统中编写针对特定开发板的外设驱动，这些代码通常硬编码在操作系统内核中，从而极大地限制了操作系统在不同开发板上的可移植性。此外，随着内核功能的不断增加，代码量也在不断膨胀，这不仅增加了学习者分析和理解内核的难度，也显著提高了系统的复杂性，给教学工作带来了不必要的复杂性。本研究旨在将驱动模块从操作系统中分离出来，重新封装成库，为操作系统开发者提供开箱即用的异步块设备驱动模块。

本文介绍了研究中涉及的 I/O 设备的基本知识，包括基于中断的 I/O 传输模式、基于 virtio 的 I/O 设备抽象以及异步非阻塞的执行模型。接着，阐述了基于 virtio 协议的驱动设计思路以及虚拟队列的工作原理和设计思路。基于 virtio 协议的设计思路，详细介绍了如何实现 virtio-blk 设备的驱动模块。最后，本文说明了该块设备驱动如何在裸机环境下完成 I/O 操作，以及如何在真实的操作系统环境中完成接口实现对接并正常工作。

关键词：块设备驱动；异步；跨操作系统；Virtio

Design and Implementation of a Cross-Operating System Asynchronous Block Device Driver Module

Abstract

Many educational operating systems need to be deployed on various development boards. This deployment process typically requires writing peripheral drivers for specific development boards, which are often hard-coded into the operating system kernel. This approach significantly limits the portability of the operating system across different development boards. Additionally, as the kernel's functionality expands, the codebase becomes increasingly bloated, making it more challenging for learners to analyze and understand the kernel. This also greatly increases the system's complexity, introducing unnecessary complications into the teaching process. This study aims to decouple the driver modules from the operating system, repackage them into libraries, and provide operating system developers with plug-and-play asynchronous block device driver modules.

This thesis introduces the fundamental concepts related to I/O devices involved in the study, including interrupt-based I/O transfer modes, the abstraction of I/O devices based on virtio, and the asynchronous non-blocking execution model. It then explains the design approach for drivers based on the virtio protocol, as well as the working principles and design rationale of virtual queues. Following the design principles based on the virtio protocol, the thesis details the implementation of the virtio-blk device driver module. Finally, it discusses how this block device driver performs I/O operations in a bare-metal environment and how it integrates and functions within a real operating system environment.

Key Words: Block Device Driver; Asynchronous; Cross-Operating System; Virtio

目　录

第 1 章 绪论

1.1 研究背景

驱动程序（Device Driver）是计算机系统中的关键组件，负责在操作系统与硬件设备之间建立桥梁。作为一种系统软件，驱动程序专门控制硬件设备的操作。它是操作系统核心的一部分，直接与硬件交互，为操作系统提供硬件的抽象层，使应用程序无需了解硬件的具体实现即可使用硬件资源。通过提供硬件控制和操作接口，驱动程序使操作系统能够有效地与各种硬件设备通信和控制。随着计算机技术的发展，从传统的键盘、鼠标到现代的图形处理器和网络设备，各种硬件设备层出不穷，都需要驱动程序来确保其正常工作。驱动程序不仅提高了硬件设备的兼容性和可操作性，还在系统的稳定性和性能中发挥重要作用。

块设备驱动程序是驱动程序中非常重要的一类。用于管理块设备（如硬盘、SSD、光驱等），负责大块数据的传输和存储操作，提供对块设备的高效访问接口。在 Linux 系统中，块设备驱动程序通常作为内核模块实现，开发者需要编写特定的内核模块，定义块设备的初始化、数据传输、缓存管理和中断处理函数[1]。在嵌入式系统中，块设备驱动程序的实现通常更加紧凑和高效。

当操作系统启动时，块设备驱动程序会被加载并初始化。在初始化过程中，驱动程序需要检测块设备、分配必要资源、注册设备接口，并识别设备的类型和容量。块设备驱动程序通过读取和写入固定大小的数据块来进行数据传输。操作系统将数据请求发送给驱动程序，驱动程序根据请求类型（读或写）及块的地址进行相应操作。为了提高数据访问速度，块设备驱动程序通常实现缓存机制，存储最近使用的数据块，减少对物理设备的直接访问次数，从而提升整体系统性能。块设备驱动程序需要处理多个同时发出的数据请求，通常维护一个请求队列，并采用一定的调度算法（如先来先服务、电梯算法等）来优化数据传输效率和设备利用率[2]。当数据传输完成或设备状态发生变化时，块设备会向操作系统发出中断信号。驱动程序需要处理这些中断，完成数据传输的后续工作，并准备处理下一个数据请求。

块设备驱动程序直接影响数据存储和管理的效率与可靠性。高效的驱动程序能够提高数据传输速度，减少存储设备的读写延迟，提升整体系统性能。稳定的块设备驱动程序能够减少系统崩溃和数据丢失的风险。块设备驱动程序使操作系统能够

支持多种不同的存储设备，提高系统的硬件兼容性。无论是传统的机械硬盘还是现代的固态硬盘，驱动程序都应确保其在系统中能正常工作。

随着计算机技术的发展，操作系统对设备驱动的要求也随之提高，相应地社会对于开发设备驱动的人才要求也在提高。同时社会对于国产操作系统的需求越来越高，涌现了许多开源操作系统，其中不乏一批被广泛用于教学，即帮助初学者入门学习操作系统设计的操作系统，例如基于 Rust 语言编写的 rCore[3]和 Alien[4]。当这些操作系统需要部署在开发板上时通常要求在操作系统中编写特定开发板的外设驱动模块。这导致操作系统在不同开发板上的可移植性受到极大挑战，给教学工作带来了不必要的复杂性。同时，随着操作系统内核功能的不断增加，代码量也在不断膨胀，这增加了学习者理解和分析内核的难度和成本。因此，为了降低这种复杂性，并减少学习者的负担，有必要使操作系统的各个部分之间尽可能解耦。

1.2 国内外研究现状

Rust 语言作为新兴语言，其生态中已经有不少杰出的成果。例如 Rust 中异步编程的重要工具，用于异步编程的运行时库 tokio 和旨在使异步/等待成为嵌入式开发的首选选项的 Embassy。

Embassy 提供了一系列的 Rust crates，可以一起或独立地使用，包括 Executor、硬件抽象层（HALs）、引导加载程序（Bootloader）等[5]。embassy-executor 是一个异步/等待执行器，通常在启动时执行固定数量的任务，尽管稍后可以添加更多任务。该执行器还可以提供系统定时器，用于异步和阻塞延迟。在一微秒以下的阻塞延迟应该使用阻塞延迟，因为上下文切换的成本太高，执行器将无法提供准确的定时。硬件抽象层 HALs 实现了安全的 Rust API，提供外设的使用支持，例如 USART、UART、I2C、SPI、CAN 和 USB，而无需直接操作寄存器。虽然 Embassy 不依赖标准库，但是 Embassy 异步运行时面向特定的开发板，泛用性较差，其异步运行时可能与其上运行的异步程序的运行时不兼容。

tokio 通过提供高效的异步执行引擎和丰富的异步 IO 支持，使得开发者能够编写高性能、高并发的异步代码，async/await 允许用户比以前更容易地编写异步代码[6]。它提供了一套基于 Futures 和 Tasks 的异步执行引擎，使得编写高效的异步代码变得简单而又高效[7]。Future 表示一个异步计算的结果，而 Task 则是对 Future 的执行。开发者可以通过组合和链式调用 Future 来构建复杂的异步操作，而 Task 负责调度这些

操作的执行[8]。Tokio 提供了一个异步执行引擎，基于事件驱动的模型实现了高效的并发处理。它允许开发者编写非阻塞的、异步的代码，从而避免了线程阻塞和资源浪费的问题。它使用 epoll（在 Linux 上）或者 kqueue（在 macOS 和 BSD 上）等底层系统调用来监听事件，并在事件发生时触发相应的异步操作。Tokio 允许在多个线程上执行异步任务，从而利用多核处理器的性能优势。它使用线程池来管理任务的执行，确保高效地利用系统资源。Tokio 提供了对异步 IO 操作的支持，包括文件操作、网络通信等。它使用非阻塞的 IO 操作来实现高性能的异步 IO，从而提高了系统的吞吐量和响应速度。Tokio 提供了丰富的中间件和工具，用于处理异步任务的执行过程中的各种需求，例如错误处理、超时控制、日志记录等。开发者可以根据需要选择和组合这些工具，以满足特定的应用场景。但是 tokio 依赖标准库完成其异步功能，对于操作系统来说，其本身不运行在任何操作系统之上，故无法使用任何标准库，在其运行时中不能包括任何 std，也即 tokio 无法对面向裸机的程序提供服务。

现阶段大部分代码仍是以同步的方式完成的，只有在少数特殊的代码例如 I/O 密集型的数据读写，网络访问等办法的代码使用异步等方式进行优化。采用一种特定的运行时应该是应用程序考虑的，即 application-specific choice，而对于最底层的操作系统，驱动程序使用特定的异步运行时库很可能难以避免依赖特定的异步执行器，异步代码和同步代码并不总能和睦共处，甚至于有时候，异步代码之间也存在类似的问题，一旦用户选了不同的或不兼容的运行时，就会导致不可预知的麻烦。所以作为要与用户打交道的操作系统的组成一部分——驱动程序，其设计便不能像嵌入式开发或应用程序一样，只考虑对特定设备的兼容性或对标准库的直接依赖。

1.3 研究目的和意义

为了应对这些挑战，本研究提出了将驱动模块从操作系统中剥离出来，以 crate 的形式与操作系统进行对接的解决方案。这种做法不仅便于驱动模块的开发和迭代，也能够增强其在不同系统中的可重用性。从理论上讲，基于 Rust 的操作系统可以通过简单地在 Cargo.toml 文件中导入所需的驱动模块 crate 来实现这一目标，从而降低了系统的复杂性，并提高了其可移植性。

对于一般性的操作系统而言，只需编译和调用相应的驱动 crate 库文件即可，而双方之间的对接则通过规定的接口（在 Rust 中称为 Trait）来完成。这种方法不仅使操作系统的架构更加清晰，也使得不同部分之间的交互更加简单和可靠。该研究的

目的是利用 Rust 语言模块化，异步支持，内存安全的特点，实现一个高效的跨操作系统的异步驱动模块。

这一研究将减轻操作系统开发者的开发负担，当越来越多研究投入该领域，将会形成完善的操作系统驱动生态，降低操作系统开发的门槛，相信将能极大促进开源操作系统的进展。另外本研究将块设备驱动从操作系统本身分离，降低了系统的耦合性，将为初学者学习分析理解操作系统内核代码带来极大帮助，减轻学习者的负担。本研究将对操作系统的开发者和学习者都能起到促进作用，这种帮助是正反馈的良性循环，相信这一研究对于促进基于 Rust 语言的教学操作系统的发展具有重要的理论和实际意义。

1.4 主要工作

本研究旨在基于操作系统对块设备驱动接口的共性，设计一个块设备驱动模块，使其能够在多种操作系统上对接使用。由于块设备操作是 I/O 密集型的，对性能要求较高，简单的 I/O 传输方式无法满足大多数操作系统的性能需求，因此需要考虑引入异步特性。具体研究内容包括：

1. 抽象操作系统对块设备的接口，例如读、写和刷新操作，设计相应方法，并暴露尽可能简单的接口供操作系统使用。通过这一抽象过程，封装了读写请求的过程，使操作系统只需负责将请求传递给驱动程序，由块设备驱动处理请求并与底层设备交互。

2. 引入异步特性，设置虚拟队列（virtqueue），实现块设备 I/O 操作命令和实际执行的分离。当操作系统发起 I/O 操作请求后，不必等待块设备实际完成工作，块设备驱动会先返回接收到请求的响应，从而使操作系统无需等待 I/O 操作结果。这不仅提高了操作系统的 I/O 吞吐量，还避免了 I/O 设备与 CPU 串行工作，提高了 CPU 的利用率。

3. 支持更多 virtio 协议规定的特性（例如 FLUSH、RO），为操作系统提供更多配置信息以支持高级特性，提供更细粒度的控制。同时，为后续支持新特性预留了完备的接口和健壮的架构，便于项目模块的开发迭代，满足设备的可持续发展要求。

4. 实现 Alien 操作系统和 rCore 操作系统对 virtio-blk 的支持，使其能够在实际操

作系统中运行。展示了该 virtio-blk 模块在真实操作系统中的可行性，并给出了接口对接工作的操作步骤。

1.5 文章组织结构

本文的安排如下：

第一章：给出本研究的研究背景和研究目的和意义，概述了在该领域的国内外研究现状，本文的主要工作内容和文章组织结构。

第二章：介绍 I/O 设备的相关领域知识，主要是介绍 CPU 连接 I/O 设备的方式，I/O 传输方式，I/O 设备抽象，I/O 执行模型与设备树。这些领域知识对于理解本项目的设计与实现十分重要。理清不同方式，不同模型的优点与缺点对于理解本研究的设计与实现有帮助。设备树一节的知识阐述了操作系统是如何识别设备并启动相应驱动模块的。

第三章：以 rCore 操作系统使用的文件系统为例，介绍了一种 Rust 实现的典型文件系统架构。在本章中介绍了与底层块设备关系最密切的文件系统在设计上为块设备驱动设置了哪些接口，其在实现上是如何接入操作系统内核以及对文件的操作需要块设备驱动实现哪些接口。并展示了测试文件系统的方法。

第四章：介绍了 virtio 协议，包括协议中规定的 virtio 设备的组织架构和 virtio 驱动的组成，包括设备状态域，特征位，设备配置空间以及 virtio 虚拟队列。阐述了通用 virtio 设备驱动的设计标准和原理，是所有 virtio 设备驱动的共性。

第五章：介绍了 virtio-blk 设备驱动的设计与实现。其展现的是 virtio 设备驱动共性中的 virtio-blk 设备驱动的特性。除了 virtio 协议标准中的实现还有驱动在和操作系统对接时的请求体，相应体和相应的方法函数。并且介绍了 virtio-blk 的初始化和 I/O 操作的实现和原理。

第六章：在这一章中对 virtio-blk 设备驱动进行了多层次多方面的正确性测试，包括驱动方法的单元测试，在裸机环境下的驱动集成测试，在真实操作系统 Alien 和 rCore 中的系统测试。在真实操作系统中测试中给出了和操作系统对接的操作步骤。

结论：总结了本文的主要成果，指出了本研究的主要创新点，剖析了其未来发展前景，提出了对未来工作的展望与设想。分析了其对社会的实用价值。

第 2 章 I/O 设备

2.1 CPU 连接外设的方式

随着设备种类的扩充和驱动技术的进步，从 CPU 与外设的交互方式的演变来看，随着时间的推移，CPU 能够管理的外设数量逐渐增加，而 CPU 与外设之间的数据传输性能（包括延迟和吞吐量）也逐步提升。总体而言，CPU 连接的外设经历了以下发展阶段：

简单设备

在计算机发展的早期阶段，由于 CPU 连接的设备较少且性能较低，CPU 可以直接通过 I/O 接口（例如嵌入式系统中的通用输入输出 GPIO 接口）控制 I/O 设备（例如简单的发光二极管等）。这种情况在简单的单片机和微处理器控制设备中非常常见。其特点是 CPU 发出 I/O 命令或数据，可以立即驱动 I/O 设备并产生相应的效果。

基于总线连接

随着计算机技术的发展，CPU 与 I/O 设备的连接逐渐增多，因此引入了 I/O 控制器作为中间层，例如串口控制器。通过对 I/O 控制器进行编程，CPU 可以控制各种设备，并通过访问相关寄存器获取设备的当前状态。然而，CPU 需要轮询检查设备情况，导致对于低速设备（如串口）而言，CPU 利用率较低。随着设备增多，I/O 控制器也趋向通用化，能够连接不同设备并进行集中管理。为了简化 CPU 与各种设备的连接，总线（bus）被引入。总线规定了连接设备需要共同遵循的连接方式和 I/O 时序等。

支持中断的设备

随着处理器技术的快速发展，CPU 与外设之间的性能差距逐渐增大，导致 CPU 在等待外设完成操作时的利用率下降。为了解决这一问题，I/O 控制器采用了中断机制的扩展，CPU 在发出 I/O 命令后无需忙等，而是可以执行其他任务。当外设完成 I/O 操作后，通过 I/O 控制器产生外部中断，从而促使 CPU 响应。这样，CPU 和外设可以并行执行任务，提高整个系统的执行效率。

高吞吐量设备

随着外设技术的迅速发展，一些高性能外设（如 SSD、网卡等）的性能不断提升。然而，当每次中断所产生的 I/O 数据传输量较少时，例如硬盘或 SSD 需要在短

时间内传输大量数据时，频繁中断 CPU 会导致总体中断处理开销增大，从而降低系统效率。为解决这一问题，引入了 DMA（Direct Memory Access，直接内存访问）控制器。DMA 控制器允许计算机内的某些硬件子系统在不依赖中央处理器的情况下独立地访问系统内存进行读取和/或写入。许多硬件系统都使用 DMA，包括磁盘驱动控制器、显卡、网卡和声卡[9]。外设可以在 CPU 不访问内存的时间段内，以数据块的方式直接进行外设和内存之间的数据传输，而无需 CPU 干预。这样一来，I/O 设备的传输效率大大提高。CPU 只需在数据传输开始前发出 DMA 指令，并在外设完成 DMA 操作后响应其发出的中断信息即可。

2.2 I/O 传输方式

在上述 I/O 设备的发展过程中，CPU 与外设之间主要有三种方式进行数据传输：

Programmed I/O

程序化 I/O（Programmed I/O，有时也称为忙等待）是最容易实现的一种 I/O 方法。它需要 CPU 等待输入设备准备好待处理的值或输出设备准备好接收来自 CPU 的值时执行一些指令[10]。PIO 方式可以进一步细分为基于 Memory-mapped 的 PIO（MMIO）和 Port-mapped 的 PIO（PMIO）。在 MMIO 中，I/O 设备的物理地址被映射到内存地址空间，这使得 CPU 可以通过普通的内存访问指令将数据传送到 I/O 设备在主存中的位置，从而完成数据传输。

相对而言，采用 PMIO 方式的 I/O 设备拥有独立的地址空间，与内存地址空间分离。若 CPU 要访问这些 I/O 设备，则需使用特殊的 I/O 指令，例如 x86 处理器中的 IN 和 OUT 指令。通过这些 I/O 指令，CPU 可以直接访问设备，实现 PMIO 方式的数据传输。

Interrupt based I/O

在传统的块 I/O 路径中，操作系统几乎通过中断异步完成所有 I/O 操作[11]。采用 PIO 方式让 CPU 获取外设执行结果时，I/O 软件中存在一个循环，CPU 不断读取外设相关寄存器，直到收到外设可继续执行 I/O 操作的信息后才能进行其他任务。当外设（如串口）的处理速度远低于 CPU 时，CPU 将陷入忙等状态，效率低下。

中断机制的引入显著减轻了 CPU 的负担。CPU 可以通过 PIO 方式通知外设，只要 I/O 设备有 CPU 需要的数据，便会发出中断请求信号。CPU 发送通知后，即可继续执行与 I/O 设备无关的任务。中断控制器会检查 I/O 设备是否准备好传输数据，并

向 CPU 发送中断请求信号。当 CPU 检测到中断信号时，会打断当前执行，并处理 I/O 传输。

Direct Memory Access

外设每传输一个字节都触发一次中断会显著降低系统执行效率。为解决这一问题，DMA（Direct Memory Access，直接内存访问）技术被引入到计算机系统中，以实现快速数据传输。DMA 允许外设直接将数据传输到内存，无需 CPU 直接参与。这样一来，CPU 能够从 I/O 任务中解放出来，提升系统整体性能。DMA 操作一般由 DMA 控制器管理。当 CPU 需要读取或写入设备数据时，它会向 DMA 控制器发出准备请求，然后 DMA 控制器在后续阶段直接完成数据传输到目标位置。具有 DMA 通道的计算机可以在传输数据到设备和从设备传输数据时，比没有 DMA 通道的计算机消耗更少的 CPU 资源[9]。

2.3 I/O 设备抽象

I/O 接口的交互协议

对于外设而言，其关键组成部分包括两个方面。首先是对外展示的设备 I/O 接口（hardware I/O interface），操作系统通过这一接口与外设进行通信和控制。每个设备都具有特定的接口和典型的交互协议。其次是内部结构，即对内的物理实现，包含设备内部的功能和结构。

在高度抽象的角度下，软件管理设备时，关注的焦点是简化设备接口而不是设备的内部结构。一个简化的抽象设备接口通常包括三个关键部分：状态、命令和数据。软件可以读取设备的当前状态，并基于此状态决定下一步的 I/O 访问请求；通过向设备发送一系列命令，软件可以请求设备执行特定的 I/O 访问操作；在 I/O 访问操作中，涉及将数据发送给设备或从设备接收数据。以下是 CPU 与设备之间的 I/O 接口交互协议：

```
1 while STATUS == BUSY {};    // 等待设备执行完毕
2 DATA =  data;               // 把数据传给设备
3 COMMAND = command;          // 发命令给设备
4 while STATUS == BUSY {};    // 等待设备执行完毕
```

在引入中断机制后，简化的抽象设备接口需要涵盖四个关键部分：状态、命令、数据和中断。CPU 与设备之间的 I/O 接口交互协议如下所示：

```
1 DATA =  data;           // 把数据传给设备
2 COMMAND = command;      // 发命令给设备
3 do_otherwork();         // 做其它事情
4 ...                     // I/O 设备完成 I/O 操作，并产生中断
5 ...                     // CPU 执行被打断以响应中断
6 trap_handler();         // 执行中断处理例程中的相关 I/O 中断处理
7 restore_do_otherwork();// 恢复 CPU 之前被打断的执行
8 ...                     // 可继续进行 I/O 操作
```

中断机制允许 CPU 的高速计算与外设的慢速 I/O 操作可以重叠，使得 CPU 无需等待外设完成操作，从而实现 CPU 与外设的并行执行，这是提高 CPU 利用率和系统效率的关键。从软件角度来看，引入 DMA 机制以提高大块数据传输效率并没有改变抽象设备接口的四个部分。仅仅是上面协议伪码中的 data 变成了 data block。这样，传输单个数据产生的中断频率会显著降低，进一步提高 CPU 利用率和系统效率。

基于文件的 I/O 设备抽象

在二十世纪七十到八十年代，计算机专家积极探索为 I/O 设备提供统一抽象的方法。最初将专门用于存储类 I/O 设备的文件概念进行了扩展，认为所有的 I/O 设备都可以被视为文件[12]。这一概念在传统 UNIX 系统中得到了体现，即设备文件。所有的 I/O 设备都以文件的形式呈现，可以通过诸如 `open/close/read/write` 的文件访问接口进行处理。在 Linux 系统下，可以通过执行 `$ ls /dev` 命令来查看各种设备文件。

然而，由于各种设备的功能多样化，仅仅依靠 `read/write` 这样的方式难以有效地与设备进行交互。因此，UNIX 的后续设计者提出了一个独特的系统调用，即 `ioctl`（input/output control）系统调用[13]。ifreq 是用于套接字 ioctl 的接口请求结构。`ioctl` 是专门用于设备输入输出操作的系统调用，它接受与设备相关的请求码作为参数，系统调用的功能完全取决于设备驱动程序对请求码的解释和处理。

尽管基于设备文件的设备管理表面上得到了大部分通用操作系统的支持，而且 `ioctl` 系统调用也非常灵活，但请求码的定义缺乏规律，文件接口过于面向用户应用，未能体现出操作系统在处理 I/O 设备时的共性特征。因此，文件这一抽象并未完全覆盖操作系统对设备管理的整个执行过程。

基于流的 I/O 设备抽象

在二十世纪八十到九十年代的 UNIX 操作系统发展过程中，随着网络等更加复杂的设备出现，人们提出了面向 I/O 设备管理的新抽象，即流（stream）。1984 年，

Dennis M. Ritchie 撰写了一份技术报告，名为“A Stream Input-Output System”，详细介绍了基于流的 I/O 设备抽象设计。这一设计的目标在于将 UNIX 中管道（pipe）机制拓展到内核的设备驱动中[14]。

流是用户进程和设备或伪设备之间的全双工连接，由多个线性连接的处理模块组成，类似于 shell 程序中的管道，用于数据双向流动。流中的模块通过消息传递进行通信，而模块不需要直接访问邻居模块的其他数据。每个模块只为每个邻居提供一个入口点，即一个接受消息的例程。

流的末端提供接口以与操作系统的其他部分交互。用户进程的写操作和输入/输出控制请求被转换成发送到流的消息，而读请求则从流中获取数据并传递给用户进程。流的另一端是设备驱动程序模块，它接收来自用户进程的数据并将其发送到设备；同时，它将设备检测到的数据和状态转换为消息，并发送到用户进程所在的流中。整个过程中会经过多个中间模块，这些模块以各种方式处理或过滤消息。

基于 virtio 的 I/O 设备抽象

在二十一世纪，随着互联网和云计算的兴起，数据中心的物理服务器通过虚拟机技术运行多个虚拟机成为主流。然而，存在多种虚拟机技术，如 Xen、VMware、KVM 等，它们要求支持虚拟化不同处理器架构和各种外设，并且要求让以 Linux 为代表的 guest 操作系统能够高效地运行在其上。这为虚拟机和操作系统带来了复杂性和困难。

为解决这一问题，IBM 工程师 Rusty Russell 提出了一组通用 I/O 设备抽象——virtio 规范。虚拟机提供 virtio 设备的实现，而 virtio 设备具有统一的接口，因此 guest 操作系统只需实现这些通用接口即可管理和控制各种 virtio 设备。虚拟机与 guest 操作系统之间的通信通道采用基于共享内存的异步访问方式，效率非常高。虚拟机会将相关 virtio 设备的 I/O 操作转换成物理机上的物理外设的 I/O 操作，从而完成整个 I/O 处理过程。

由于 virtio 设备的设计，虚拟机无需模拟真实的外设，因此可以设计出一种统一且高效的 I/O 操作规范，让 guest 操作系统处理各种 I/O 操作。这种 I/O 操作规范形成了基于 virtio 的 I/O 设备抽象，并逐渐成为虚拟 I/O 设备的事实标准。

2.4 I/O 执行模型

根据 Richard Stevens 的经典著作《UNIX Network Programming Volume 1: The Sockets Networking》所述，UNIX 环境中的 I/O 系统调用具有多种不同类型的执行模型，可大致分为五种 I/O 执行模型（IO Model）: 阻塞 IO（Blocking IO）、非阻塞 IO（Non-blocking IO）、多路复用 IO（IO Multiplexing）、信号驱动 IO（Signal-driven IO）和异步 IO（Asynchronous I/O）[15]。

在这些模型中，当用户进程发出一个 `read` 系统调用时，主要经历两个阶段：等待数据准备就绪和将数据从内核拷贝到用户进程中。

这五种 IO 模型在这两个阶段有不同的处理方式。阻塞与非阻塞关注的是进程的执行状态，而同步和异步关注的是消息通信机制。阻塞 IO 和非阻塞 IO 的区别在于第一阶段，即内核数据准备好之前是否阻塞用户进程；同步 IO 和异步 IO 的区别在于第二阶段，即数据从内核复制到用户空间时用户进程是否被阻塞或参与。

表 2-1 等待数据准备阶段的不同处理方式

处理方式	特征
阻塞	进程执行系统调用后会被阻塞，直到 I/O 操作完成。
非阻塞	进程执行系统调用后不会被阻塞，即使 I/O 操作未完成，进程也会继续执行。

表 2-2 数据从内核拷贝到用户进程阶段的不同处理方式

处理方式	特征
同步	用户进程与操作系统（设备驱动）之间的操作是经过双方协调的，步调一致的。
异步	用户进程与操作系统（设备驱动）之间无需协调，可以各自独立进行操作。

模型的选择取决于应用程序的需求和设计考虑，不同的场景可能会选择不同的 IO 模型以达到最佳的性能和可维护性。

阻塞 IO（Blocking IO）、非阻塞 IO（Non-blocking IO）、多路复用 IO（IO Multiplexing）、信号驱动 IO（Signal-driven IO）都属于同步 IO 模型。这些模型在第二阶段（实际 IO 操作）中需要用户进程参与，因此称为同步 IO 模型。

虽然执行非阻塞 IO 系统调用的进程在第一阶段未被阻塞，但在第二阶段（实际 IO 操作）时，内核会阻塞用户进程，因为数据需要从内核复制到用户空间。

异步 IO 不同，用户进程发起 IO 操作后立即返回，直到内核通知 IO 完成。整个过程中，用户进程不会被阻塞。

阻塞 IO（blocking IO）

阻塞 IO（blocking IO）的特点：在 I/O 执行的两个阶段（等待数据和拷贝数据）中，用户进程都处于阻塞状态。

当用户进程发出 `read` 系统调用时，如果所需数据不在 I/O 缓冲区中，内核将用户进程置于阻塞状态，并向磁盘驱动程序发出 I/O 操作请求。直到数据从磁盘传输到 I/O 缓冲区，并且内核将数据从缓冲区拷贝到用户进程的 buffer 中，并唤醒用户进程，`read` 系统调用才完成，用户进程才从阻塞状态中恢复。

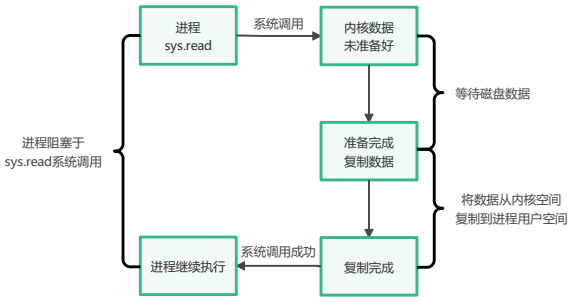

图 2-1 阻塞 I/O

非阻塞 IO（non-blocking IO）

非阻塞 IO（non-blocking IO）的特点：非阻塞系统调用在被调用之后会立即返回，而不会阻塞用户进程。

当用户进程发出 `read` 系统调用时，内核发现所需数据不在 I/O 缓冲区中，不会让用户进程处于阻塞状态，而是立即返回一个 error。用户进程收到 error 后，知道数据还没有准备好，因此可以再次发送 `read` 操作，这个过程可以重复多次。当磁盘驱动程序将数据从磁盘传输到 I/O 缓冲区并通知内核后，内核收到通知并再次收到用户进程的 system call 时，立即将数据从 I/O 缓冲区拷贝到用户进程的 buffer 中。用户进程不会被内核阻塞，而是需要不断主动询问内核所需数据是否准备好。

多路复用 IO（IO multiplexing）

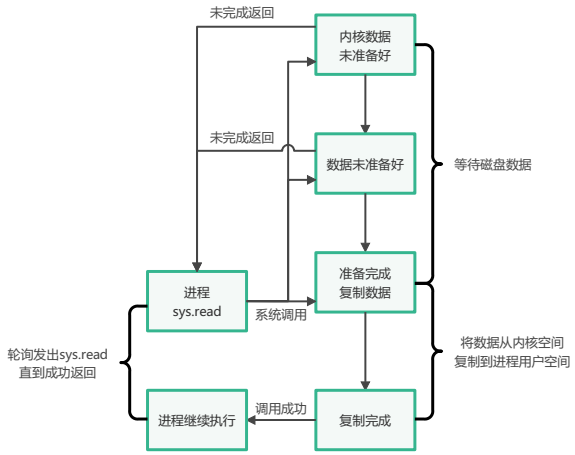

图 2-2 非阻塞 I/O

在 Linux 中，有三个主要的系统调用可以在使用非阻塞 I/O 时提供帮助，通过提供一种进行 I/O 多路复用的方式：select、poll 和 epoll[16]。

基本工作机制是通过 `select` 或 `epoll` 系统调用不断轮询用户进程关注的所有文件句柄或 socket。epoll 提供了更现代化的基础，提供了 I/O 多路复用的功能。它旨在提高可伸缩性，特别是在要监视的文件描述符方面[16]。当某个文件句柄或 socket 有数据到达时，select 或 `epoll` 系统调用会返回到用户进程，然后用户进程再调用 read 系统调用，将数据从内核的 I/O 缓冲区拷贝到用户进程的 buffer 中。

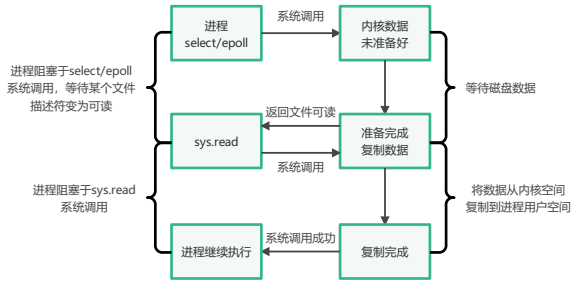

图 2-3 多路复用 I/O

信号驱动 IO（signal driven I/O）

进程发起 `read` 系统调用时，向内核注册信号处理函数后立即返回，进程不会被阻塞，而是继续执行。当内核中的 IO 数据准备就绪时，会向进程发送一个信号，进程在信号处理函数中调用 IO 读取数据。该模型的特点是采用了回调机制，但是增加

了开发和调试应用的难度。

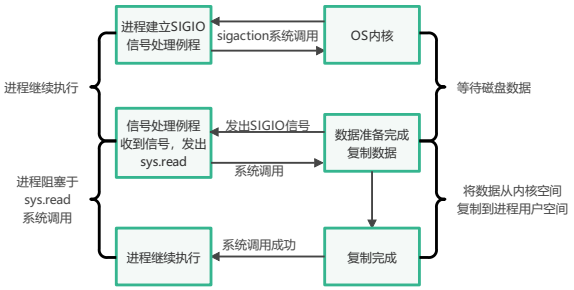

图 2-4 信号驱动 I/O

异步 IO（Asynchronous I/O）

用户进程调用 `async_read` 异步系统调用后，即可立即执行其他任务。从内核的角度看，收到 `async_read` 异步系统调用后，内核会立即返回，不会造成用户进程阻塞。然后，内核会等待数据准备就绪，将数据复制到用户内存。当数据复制完成后，内核会通知用户进程，告知读取操作已完成。

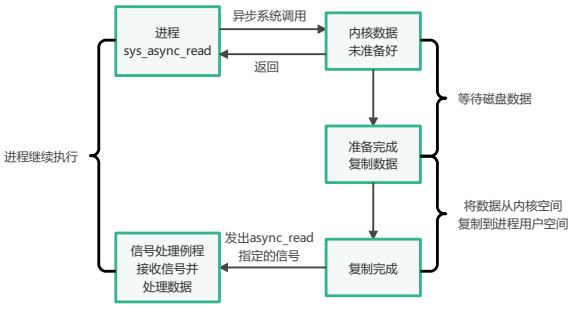

图 2-5 异步 I/O

第 3 章 文件系统

3.1 文件系统架构

为了改进单元测试和集成测试的便捷性，并实现内核各个部分的松耦合，将文件系统从内核中分离出来。文件系统的设计与开发采用应用程序库的开发模式，最终可将其直接整合到内核中，生成支持文件系统的新内核。

为避免与底层设备驱动的耦合，文件系统与底层设备驱动之间通过抽象接口 `BlockDevice` 进行连接。利用 Rust 提供的 alloc crate，文件系统实现了对操作系统内核内存管理的隔离，避免直接调用内存管理内核函数。在底层驱动方面，采用轮询方式访问 `virtio_blk` 虚拟磁盘设备，避免了访问外设中断相关内核函数。设计过程中，避免了直接访问进程相关数据和函数，从而实现了对操作系统内核进程管理的隔离。

文件系统被划分为不同层次，形成层次化和模块化的设计架构。文件系统自下而上大致划分为五个不同层次。

3.1.1 磁盘块设备接口层

设计一个 Trait 接口，以块为单位对磁盘块设备进行读写操作。

```
pub trait BlockDevice : Send + Sync + Any {
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    fn write_block(&self, block_id: usize, buf: &[u8]);
}
```

在 `fs-crate` 中，并没有具体实现了 `BlockDevice` Trait 的类型。因为块设备仅支持以块为单位进行随机读写，所以需要由具体的块设备驱动来实现这两个方法。实际上，这需要由文件系统的使用者提供并接入到 `fs-crate` 库中。`fs-crate` 库的块缓存层会调用这两个方法来管理块缓存。

3.1.2 块缓存层

为了避免频繁读写磁盘而降低系统性能，内存中缓存了磁盘块的数据。常见的做法是通过 `read_block` 将块数据从磁盘读入内存缓冲区，在缓冲区中进行读写操作。若缓冲区内容修改，则需要通过 `write_block` 将其写回磁盘。

表 3-1 磁盘布局

区域	块数	说明
超级块	一块	包含关键参数和元数据信息
索引节点位图	若干	记录索引节点的分配情况
索引节点	若干	包含文件和目录的元数据信息
数据块位图	若干	记录数据块区域的分配情况
数据块	若干	存储文件和目录的实际数据内容

为提高性能和代码鲁棒性，对这些缓冲区进行合理管理至关重要。一种常见的管理模式是：每次读写块时，创建一个临时缓冲区，操作完后可选地写回磁盘。为减少实际块读写次数的开销，需要合并块读写操作。例如，若块已在缓冲区中，则无需再次读取；若多次修改同一块，可暂时不写回磁盘，待所有修改完成后一次性写回。

复杂的磁盘数据结构使得合理规划块读写时机困难。为解决此类问题，引入了全局管理器统一管理缓冲区。当需要读写块时，先查询全局管理器是否已缓存该块。若已缓存，则在同一缓冲区内执行所有操作，避免同步性问题。全局管理器负责合并块操作并在适当时机执行实际的块读写，上层子系统无需关注此过程。

3.1.3 磁盘数据结构层

fs-crate 文件系统中，核心数据结构包括超级块、位图、索引节点、数据块和目录项。这些结构用于管理文件系统中的文件和目录，并将逻辑上的文件目录树结构映射到磁盘上。为了便于管理和更新，磁盘数据被组织为超级块、位图、索引节点、数据块和目录项这五种不同属性的连续区域。超级块存储文件系统的重要参数和元数据信息，位图记录磁盘空闲块的使用情况，索引节点包含文件和目录的元数据，数据块用于存储文件内容，目录项用于记录目录中的文件名和对应的索引节点。这种布局使得文件系统的存储结构清晰，方便进行管理和操作。

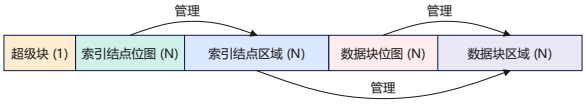

图 3-1 磁盘数据结构层示意图

3.1.4 磁盘块管理器层

从此层起的数据结构存放在内存上。

磁盘块管理模块负责整合核心数据结构和磁盘布局，并处理创建和打开文件系统以及磁盘块的分配和回收等操作。磁盘块管理模块具有以下职责：它将超级块、位图、索引节点和数据块等核心数据结构整合到文件系统中，确保它们按照磁盘布局的设计正确地组织在各个区域中。此外，磁盘块管理模块提供创建和打开文件系统的方法，确保文件系统的正确初始化和加载。作为磁盘块管理器，该模块还负责实现磁盘块的分配和回收功能，跟踪位图中的分配情况，并根据需要分配和释放数据块。通过磁盘块管理模块，整个文件系统的磁盘布局和数据管理得以统一管理和控制，确保文件系统的正确性和高效性。

3.1.5 索引节点层

磁盘块管理模块已经实现了磁盘布局并有效地管理磁盘块。然而，对于文件系统的使用者而言，通常更关注逻辑上的文件和目录，而不是磁盘布局的具体实现。因此，需要设计索引节点 `Inode` 来向文件系统的使用者公开文件和目录的操作接口。`Inode` 和 `DiskInode` 的区别可以从它们的命名中看出：`DiskInode` 存放在磁盘块中的固定位置，而 `Inode` 则是内存中记录文件索引节点信息的数据结构。磁盘块管理模块负责管理索引节点数据结构，并实现文件创建、文件打开、文件读写等成员函数，以支持文件操作相关的系统调用。

3.2 文件系统测试

`fs-crate` 架构设计的一个优点在于，它允许在 Rust 应用开发环境（包括 Windows、macOS 和 Ubuntu）中按照应用程序库的开发方式进行测试，而不需要提前将其放入内核中进行测试运行。这是因为内核运行在裸机环境上，对其进行调试很困难，而面向应用的开发环境则提供了更为完善的调试支持，从基于命令行的 GDB 到 IDE 提供的图形化调试界面都能为文件系统的开发带来很大帮助。另一个优点是，由于 `fs-crate` 需要放置在裸机上运行的内核中，因此只能使用 `no_std` 模式，无法调用标准库 `std`。但是，将 `fs-crate` 作为一个应用的库运行时，可以暂时允许使用它的应用程序调用标准库 `std`，这在开发调试过程中会带来一些方便。

从文件系统使用者的角度来看，只需要一个实现了 `BlockDevice` Trait 的块设备

来装载文件系统，之后即可使用 `Inode` 方便地进行文件系统操作。在开发环境中，为了提供这样一个块设备，可以利用 Linux（或者其他通用操作系统如 Windows/MacOS）上的一个文件来模拟块设备。Rust 标准库 std 提供了 `std::file::File`，可用于访问 Linux 上的文件。将其封装成 `BlockFile` 类型，以模拟一个块磁盘，并为其实现 `BlockDevice` 接口。

在每次清空文件 `filea` 的内容后，向其中写入一个不同长度的随机数字字符串，然后再将其全部读取出来，以验证写入的内容是否与读取的内容一致。

第 4 章 virtio 设备驱动设计

4.1 virtio 设备

4.1.1 virtio 概述

作为运行在硬件和操作系统层之间的一层，hypervisor 最早在 1960 年代被提出。hypervisor 使得计算环境能够在单个物理计算机上同时运行多个独立的操作系统，从而更有效地利用可用的计算能力、存储空间和网络带宽[17]。Rusty Russell 在 2008 年左右设计了 virtio 协议，并开发了虚拟化解决方案 lguest，形成了 VirtIO 规范（Virtual I/O Device Specification）[18]。该规范的主要目的是简化和统一虚拟机（Hypervisor）的设备模拟，并提高虚拟机环境下的 I/O 性能。virtio 协议是对 hypervisor 中一组通用模拟设备的抽象，定义了虚拟设备的输入/输出接口[19]。基于 virtio 协议的 I/O 设备被称为 virtio 设备。

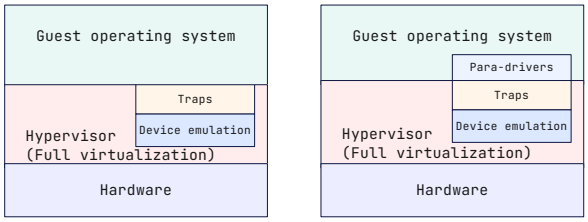

图 4-1 虚拟机解决方案

如图4-1所示，左侧的虚拟机模拟外设的传统方案中，如果 guest VM 需要使用底层 host 主机的资源，那么 Hypervisor 必须截获所有的 I/O 请求指令，并模拟这些指令的行为。然而，这种方式会导致较大的性能开销。右侧的虚拟机模拟外设的 virtio 方案中，模拟的外设实现了功能最小化的原则。也就是说，虚拟外设的数据面接口主要与 guest VM 共享内存，而控制面接口主要基于内存映射的寄存器和中断机制[20]。因此，当 guest VM 通过访问虚拟外设来使用底层 host 主机的资源时，Hypervisor 只需要处理少量的寄存器访问和中断机制，从而实现了高效的 I/O 虚拟化过程。

virtio 设备包括各种类型，如块设备（virtio-blk）、网络设备（virtio-net）、键盘鼠标类设备（virtio-input）、显示设备（virtio-gpu），它们具有共性特征和独有特征。共性特征通过统一抽象接口进行设计，而独有特征则尽量最小化各种类型设备的抽象

接口，从而屏蔽了各种 hypervisor 的差异性，实现了 guest VM 和不同 hypervisor 之间的交互过程。

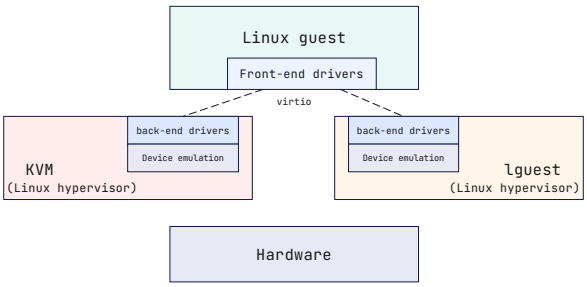

图 4-2 虚拟机交互过程

上图所描述的虚拟机模拟外设的 virtio 方案意味着，在 guest VM 上所见的虚拟设备具有简洁通用的优势。这对于运行在 guest VM 上的操作系统而言，意味着可以设计出轻量高效的设备驱动程序（即上图中的 Front-end drivers）。

在操作系统中，virtio 设备驱动程序（Front-end drivers）管理和控制着这些 virtio 虚拟设备。这些驱动程序仅需实现基本的发送和接收 I/O 数据，而位于 Hypervisor 中的 Back-end drivers 和设备模拟部分负责处理实际物理硬件设备上的设置、维护和处理，极大地减轻了 virtio 驱动程序的复杂性[21]。

4.1.2 virtio 架构

virtio 架构可以分为上、中、下三层。上层包括各种驱动程序（Front-end drivers），这些驱动程序在 QEMU 模拟器中运行的前端操作系统中；下层是在 QEMU 中模拟的各种虚拟设备 Device；而中间层则是传输（transport）层，负责驱动程序与虚拟设备之间的交互接口。传输层包含两部分：上半部是 virtio 接口定义，即定义了 I/O 数据传输机制的 virtio 虚拟队列（virtqueue）；下半部是 virtio 接口实现，即具体实现了 I/O 数据传输机制的 virtio-ring。virtio-ring 主要由环形缓冲区和相关操作组成，用于保存驱动程序和虚拟设备之间进行命令和数据交互的信息。

在操作系统中，virtio 驱动程序的主要功能包括接受来自用户进程或其他操作系统组件的 I/O 请求，将这些请求通过 virtqueue 发送到相应的 virtio 设备，并通过中断或轮询等方式查找并处理设备完成的 I/O 请求。

而在 QEMU 或 Hypervisor 中，virtio 设备的主要功能是通过 virtqueue 接受来自

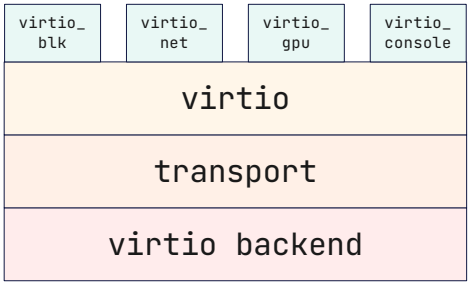

图 4-3 virtio 架构示意

相应 virtio 驱动程序的 I/O 请求，然后通过设备仿真模拟或将 I/O 操作卸载到主机的物理硬件来处理这些请求，最终通过寄存器、内存映射或中断等方式通知 virtio 驱动程序处理已完成的 I/O 请求。

在运行在 Qemu 中的操作系统中，virtio 驱动程序与 Qemu 模拟的 virtio 设备驱动之间存在以下关系：

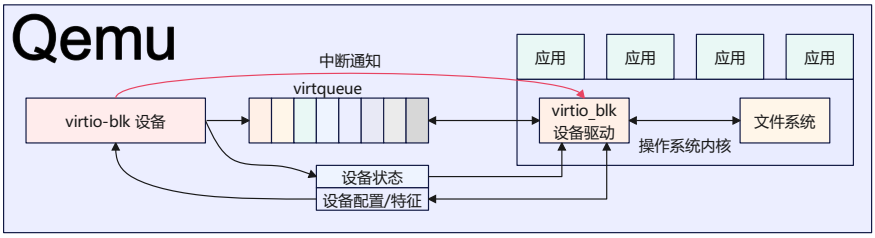

图 4-4 Qemu 模拟器上的 virtio 设备驱动

4.2 virtio 基本组成结构

表 4-1 virtio 设备的基本组成要素

组成要素	说明
设备状态域	表示 virtio 设备的当前状态，通常在设备初始化时使用
特征位	描述 virtio 设备的功能特性，由驱动程序和设备之间协商确定
通知	设备和驱动程序之间的异步事件通知，例如新数据到达或操作完成
设备配置空间	存储 virtio 设备的配置信息，包括设备的各种属性和参数
一个或多个虚拟队列	驱动程序和设备之间的数据传输和命令交互，实现高效通信的关键

virtio 设备的基本组成要素包括设备状态域（Device status field），特征位（Feature bits），通知（Notifications），设备配置空间（Device Configuration space），一个或多个虚拟队列（virtqueue）。其中，设备特征位和设备配置空间属于 virtio 设备的特征描述，用于描述设备的功能和配置；设备状态域用于初始化时表示设备的当前状态；而通知和虚拟队列则是 virtio 设备运行时的关键组成部分，用于实现设备和驱动程序之间的数据交换和通信。

4.2.1 呈现模式

virtio 设备支持三种不同的设备呈现模式：

表 4-2 呈现模式

组成要素	说明
Virtio Over MMIO	虚拟设备直接挂载到系统总线上，在内存映射的形式下呈现
Virtio Over PCI BUS	将设备挂载到 PCI 总线上，作为 virtio-pci 设备呈现
Virtio Over Channel I/O	主要用于虚拟 IBM s390 计算机，virtio-ccw

在 QEMU 模拟的 RISC-V 计算机上，采用的是 Virtio Over MMIO 的呈现模式。因此，在实现设备驱动时，只需找到相应 virtio 设备的 I/O 寄存器等内存形式呈现的地址空间，即可对 I/O 设备进行初始化和管理。

4.2.2 特征描述

virtio 设备的特征描述包括设备特征位和设备配置空间。

特征位：

特征位用于表示 VirtIO 设备具有的各种特性和功能。其中，bit0 –23 是特定设备可以使用的 feature bits，bit24 –37 用于预留给队列和特征协商机制，而 bit38 以上保留给未来其他用途。驱动程序与设备之间进行特性协商，以形成一致的共识，从而正确地管理设备。

设备配置空间：

设备配置空间通常用于配置不经常变动的设备参数（属性），或者在初始化阶段需要设置的设备参数。设备的特征位中包含一个表示配置空间是否存在的 bit 位，并

可以通过在特征位的末尾添加新的 bit 位来扩展配置空间。

在初始化 virtio 设备时，设备驱动程序需要根据 virtio 设备的特征位和配置空间来了解设备的特性，并对设备进行适当的初始化。

4.2.3 状态表示

设备状态域

在 virtio 设备初始化过程中，使用设备状态域来表示设备的状态。在设备驱动程序对 virtio 设备进行初始化过程中，会根据设备状态域的不同状态，经历一系列的初始化阶段。

表 4-3 设备状态域

状态	取值	说明
ACKNOWLEDGE	1	驱动程序发现了设备，并确认这是一个有效的 virtio 设备
DRIVER	2	驱动程序知道如何驱动这个设备
FAILED	128	由于某种错误原因，驱动程序无法正常驱动这个设备
FEATURES_OK	8	驱动程序已经与设备就设备特性达成一致
DRIVER_OK	4	驱动程序已加载完成，设备可以正常工作
DEVICE_NEEDS_RESET	64	设备触发了错误，需要重置才能继续工作

I/O 传输状态

在设备驱动程序控制 virtio 设备进行 I/O 传输过程中也有不同的状态。例如，对于 virtio_blk 设备驱动程序而言，当发出一个读设备块的 I/O 请求后，设备处于 I/O 请求状态。设备接收到请求后，进入 I/O 处理状态，处理读取操作。完成读取后，设备进入 I/O 完成状态，并通知设备驱动程序。驱动程序在接收到通知后，进行 I/O 后续处理，将数据传递给文件系统进行进一步处理，或者进行错误恢复处理。

4.2.4 交互机制

Notification 通知

在 virtio 设备的交互过程中，使用基于 Notifications 的事件通知机制。这意味着驱动程序和设备需要相互通知对方，以便处理数据。驱动程序可以通过门铃（doorbell）机制向设备发送通知，通常通过 PIO 或 MMIO 方式访问设备特定的寄存器。QEMU

表 4-4 I/O 传输状态

状态	说明
请求状态	驱动程序指示 I/O 请求队列当前位置信息，设备据此执行相应 I/O 传输操作
处理状态	设备正在处理收到的 I/O 请求
完成状态	设备已完成 I/O 请求的处理，并通知设备驱动程序
错误状态	发生了 I/O 请求处理过程中的错误
后续处理状态	设备驱动程序在接收到 I/O 完成通知后进行的后续处理

会拦截这些访问，并通知其模拟的设备。而设备通知驱动程序则通常使用中断机制。在 QEMU 中，中断被注入到 CPU 中，使其响应并执行相应的中断处理例程，从而完成对 I/O 执行结果的处理。

virtqueue 虚拟队列

为了实现批量数据传输，virtio 设备使用了 virtqueue 虚拟队列机制。每个 virtio 设备可以拥有零个或多个 virtqueue。每个 virtqueue 占用多个物理页，并可以通过各种数据结构（如数组、环形队列等）来实现。virtqueue 用于设备驱动程序向设备发送 I/O 请求命令和相关数据（例如磁盘块读写请求和读写缓冲区），同时也用于设备向设备驱动程序发送 I/O 数据（例如接收的网络包）。

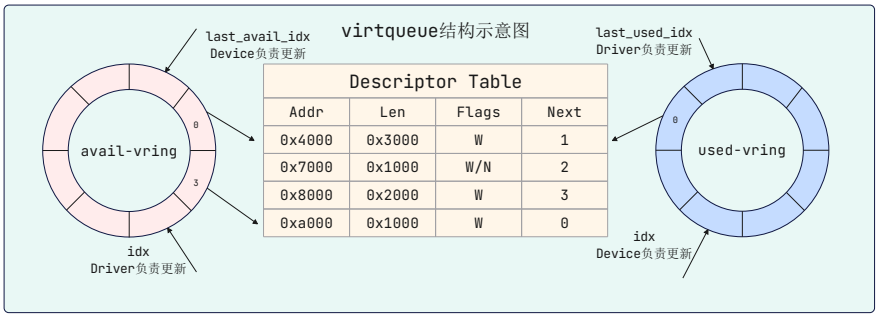

图 4-5 虚拟队列设计示意图

4.3 virtqueue 虚拟队列

在 virtio 协议中，virtqueue 是一个关键部分，用于在 virtio 设备和驱动程序之间进行批量数据传输的机制和抽象表示。在设备驱动程序的实现和 Qemu 中的 virtio 设

备模拟中，virtqueue 被视为一种数据结构，用于执行各种数据传输操作。

当描述 virtqueue 时，有时会将其与 vring（即 virtio-rings 或 VRings）等同视，也有时会将二者单独描述为不同的对象。这里将它们单独描述，因为 vring 是 virtqueues 的主要组成部分，是实现 virtio 设备和驱动程序之间数据传输的数据结构[22]。

4.3.1 描述符表（Descriptor Table）

描述符表用于指向 virtio 设备中的 I/O 传输请求缓冲区信息，通常由一系列描述符组成，数量由队列大小（Queue Size）决定。每个描述符包含物理地址，长度，下一个描述符指针。物理地址指向 I/O 传输缓冲区的物理地址。长度指定了待读取或待写入数据的大小。下一个描述符指针指向下一个描述符的指针，用于将多个描述符链接成描述符链。这样的描述符链可以表示一个完整的 I/O 操作请求，包括 I/O 操作的命令、数据块和返回结果。

设备驱动程序在初始化过程中需要分配描述符表所需的内存空间，并在其中填写描述符的信息。在后续的 I/O 操作中，设备驱动程序将在描述符表中创建描述符链，并将其提交给 virtio 设备以执行相应的 I/O 操作。

表 4-5 描述符的组成

域	长度	含义
addr	8	某段内存的起始地址
len	4	某段内存的长度, 最大为 4GB
flags	2	内存段的读写属性等
next	2	下一个内存段对应的描述符在描述符表中的索引

4.3.2 可用环（Available Ring）

可用环是一个环形队列，用于驱动程序向设备发送 I/O 操作请求。其结构包含了一系列的条目（items），这些条目只能由驱动程序写入，而设备则会读取这些条目。每个条目包含了一个描述符链的头部描述符的索引值，用于指示设备需要执行的 I/O 操作请求的位置。

可用环的头指针（idx）和尾指针（last_avail_idx）用于表示可用条目的范围。当

驱动程序写入新的描述符链时，头指针会递增，表示有新的可用条目。设备通过读取可用环中的条目来获取驱动程序发出的 I/O 操作请求对应的描述符链，然后执行相应的 I/O 操作。

描述符链通常由多个描述符组成，每个描述符指向一个缓冲区，用于存储数据或者命令。例如，在磁盘读取操作中，描述符链可能包含一个用于读取磁盘块的描述符、一个用于存储读取数据的缓冲区描述符，以及一个用于存储操作结果的描述符。设备会根据描述符链中的信息执行相应的 I/O 操作，例如读取磁盘块并将数据存储到缓冲区中。

表 4-6 可用环的组成

域	作用
flags	与通知机制相关
idx	最新放入 I/O 请求的编号
可用环数组	存放 I/O 请求的首个描述符的索引

4.3.3 已用环（Used Ring）

已用环是一个环形队列，用于 virtio 设备向驱动程序发送已完成的 I/O 操作请求。其结构与可用环类似，包含一系列的条目，每个条目表示一个已完成的描述符链的头部描述符的索引值。

已用环的头指针（idx）和尾指针（last_used_idx）用于表示已用条目的范围。当设备完成一个 I/O 操作时，会将描述符链的头部描述符的索引值添加到已用环中，并更新头指针。驱动程序通过读取已用环中的条目来获取已完成的 I/O 操作请求的信息，包括操作的结果以及使用的描述符链。

例如，在磁盘读取操作完成后，virtio 设备会将描述符链的头部描述符的索引值添加到已用环中，并在描述符链中存储操作的结果。驱动程序通过读取已用环中的条目来获取已完成的 I/O 操作请求的信息，然后进行进一步的处理。

基于 virtqueue 进行 I/O 操作有以下几个阶段

初始化阶段（由驱动程序执行）

在设备初始化过程中，virtio 设备驱动程序首先分配 virtqueue 的内存空间，包括

表 4-7 已用环的组成

域	作用
flags	与通知机制相关
idx	最新放入 I/O 响应的编号
可用环数组	存放 I/O 响应的首个描述符的索引

描述符表、可用环以及已用环。然后，驱动程序将这些部分的物理地址写入 virtio 设备的控制寄存器中，从而实现设备驱动程序和设备之间共享整个 virtqueue 的内存空间。

发起 I/O 请求阶段（由驱动程序执行）

当设备驱动程序发起 I/O 请求时，首先将请求的命令或数据存储到一个或多个 buffer 中。然后，在描述符表中分配新的描述符（或描述符链），指向这些 buffer。接着，将描述符（或描述符链的首描述符）的索引写入可用环中，并更新可用环的 idx 指针。最后，通过 kick 机制通知设备有新的请求。

完成 I/O 请求阶段（由设备执行）

当 virtio 设备接收到 kick 通知后，通过访问可用环的 idx 指针，解析出 I/O 请求。然后，设备执行 I/O 请求，并将结果存储到相应的 buffer 中。完成后，将描述符（或描述符链的首描述符）的索引写入已用环中，并更新已用环的 idx 指针。最后，通过中断机制通知设备驱动程序完成了 I/O 操作。

I/O 后处理阶段（由驱动程序执行）

设备驱动程序读取已用环的 idx 信息，并从已用环中读取描述符索引，以获取 I/O 操作完成信息。

4.4 硬件系统架构

4.4.1 设备树

设备树是描述硬件配置的数据结构。它包括关于 CPU、内存条、总线和外围设备的信息。操作系统能够在启动时解析这个数据结构，并利用它来决定如何配置内核以及加载哪些设备驱动程序[23]。虚拟计算机连接了许多外部设备，每个设备物理连接到父设备，最终通过总线等连接成设备树。

设备树与设备节点属性

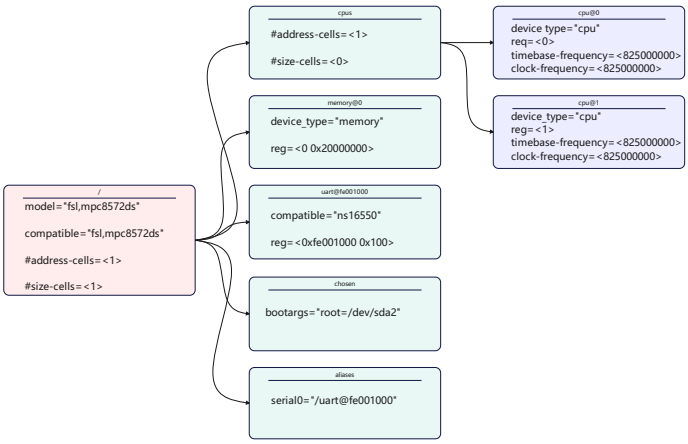

图 4-6 典型的设备树示意图

设备树是一种数据结构，用于描述硬件系统的结构和功能，并提供给操作系统使用。设备树以文本文件形式存在，包含处理器类型和数量，板载设备（如存储器、网卡、显卡等）类型和数量和硬件接口（如 I2C、SPI、UART 等）类型和地址信息。

设备树中的节点描述硬件设备的信息，每个节点包含一个或多个属性，每个属性都是键值对，用于描述设备的特定信息。操作系统通过这些节点信息识别和初始化设备[24]。

常见的设备节点属性包括：

表 4-8 常见的设备节点属性

属性	说明
compatible	设备类型，如“virtio,mmio”表示设备通过 virtio 协议、MMIO 方式驱动
reg	设备在系统中的地址空间位置
interrupts	设备支持的中断信号

设备树在嵌入式系统中广泛应用，是一种将硬件信息传递给操作系统的常用方法。在桌面和服务器系统中，PCI 总线类似于设备树，可用于访问 PCI 设备的地址空间信息。

4.4.2 传递设备树信息

在启动过程中，操作系统需要获取计算机系统中所有接入的设备信息。这一任务通常由引导加载程序（bootloader）完成，在 RISC-V 架构中，常见的引导加载程序包括 OpenSBI 或 RustSBI 固件。引导加载程序负责探测各个外设，包括物理内存，并将探测结果保存在物理内存中的某个位置，通常以设备树二进制对象（Device Tree Blob，DTB）的格式存储。随后，引导加载程序会启动操作系统，将 DTB 的物理地址加载到寄存器 a1 中，将硬件线程 ID（HART ID）加载到寄存器 a0 中，然后跳转到操作系统的入口地址开始执行。

只需为 main 函数添加两个参数（即 a0 和 a1 寄存器中的值），以便测试用例获取 bootloader 传递的放置 DTB 的物理地址。然后，init_dt 函数将此地址转换为 Fdt 类型，并遍历整个设备树以查找所有的 virtio 和 mmio 设备（通常是 QEMU 模拟的各种 virtio 设备）。接着，调用 virtio_probe 函数来显示设备信息并初始化这些设备。

4.4.3 解析设备树信息

virtio_probe 函数会进一步检查 virtio 设备节点中的 `reg` 属性，以确定 virtio 设备的具体类型（例如 DeviceType::Block 表示块设备类型），以及其他参数。通过这些信息，操作系统能够对具体的 virtio 设备进行初始化并执行相应的 I/O 操作。在 virtio_probe 函数中，会查找 virtio 设备节点的 reg 属性，以确定 virtio 设备的具体类型（例如块设备类型）。这使得操作系统能够对特定的 virtio 设备进行初始化并执行 I/O 操作。

4.5 virtio 驱动程序

4.5.1 设备的初始化

操作系统在发现 virtio 设备后，驱动程序进行设备初始化。首先，重启设备状态，将设备状态域设置为 0。接着，将设备状态域设置为 ACKNOWLEDGE，表示已识别到设备。然后，将设备状态域设置为 DRIVER，表示驱动程序了解如何操作该设备。随后，执行设备特定的安装和配置，包括协商特征位、建立 virtqueue、访问设备配置空间等，将设备状态域设置为 FEATURES_OK。最后，将设备状态域设置为 DRIVER_OK，如果出现错误就设置为 FAILED。上述步骤并非必须全部执行，但最终必须将设备状态域设置为 DRIVER_OK，以便驱动程序能够正常访问设备。

在 virtio_driver 模块中，实现了通用的 virtio 驱动程序框架，各种 virtio 设备驱动程序共同的初始化过程包括：确定协商特征位，调用 VirtIOHeader 的 begin_init 方法执行 virtio 设备初始化的步骤。接着，读取配置空间，确定设备的配置情况。然后，建立虚拟队列 1 至 n 个 virtqueue。最后，调用 VirtIOHeader 的 finish_init 方法将设备状态域设置为 DRIVER_OK。

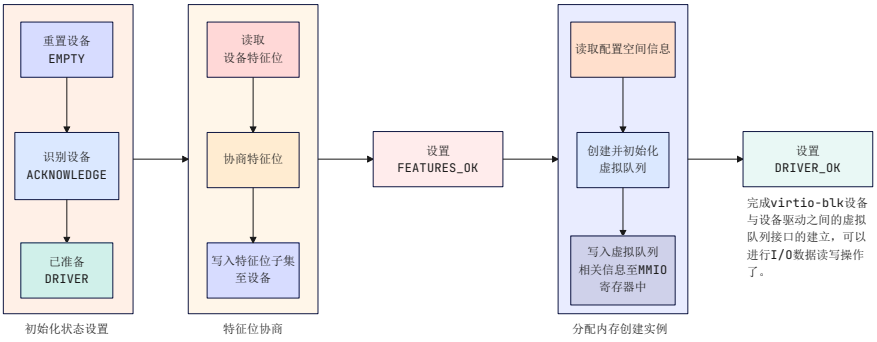

图 4-7 virtio 设备初始化示意图

4.5.2 驱动程序与设备之间的交互

驱动程序与外设通过共享的 virtqueue 进行通信，virtqueue 保存着设备驱动的 I/O 请求信息和设备的 I/O 响应信息。virtqueue 由描述符表（Descriptor Table）、可用环（Available Ring）和已用环（Used Ring）组成。在设备驱动初始化过程中，虚拟队列会被创建。

当驱动程序向设备发送 I/O 请求时，它会在 buffer 中填充命令/数据，然后将每个 buffer 的起始地址和大小信息放入描述符表的描述符中，并将这些描述符链接在一起，形成描述符链。描述符链的起始描述符的索引信息会放入一个称为环形队列的数据结构中，其中一类是可用环，包含设备驱动发出的 I/O 请求所对应的描述符索引信息，另一类是已用环，包含设备发出的 I/O 响应所对应的描述符索引信息。

用户进程发起的 I/O 操作经过层层传递到驱动程序，驱动程序将 I/O 请求信息放入 virtqueue 的可用环中，并通过通知机制通知设备。设备收到通知后，解析可用环和描述符表，取出 I/O 请求并在内部进行实际 I/O 处理。处理完成后，设备将结果作为 I/O 响应放入已用环中，并通过通知机制通知 CPU。驱动程序解析已用环，获取 I/O 响应的结果，并在进一步处理后，最终返回给用户进程。

发出 I/O 请求的过程

驱动程序向设备提供新的 I/O 请求信息首先将包含一个 I/O 请求内容的缓冲区的地址和长度信息放入描述符表中的空闲描述符中，并根据需要将多个描述符进行链接，形成一个描述符链，表示一个 I/O 操作请求。接着，将描述符链头的索引放入可用环的下一个环条目中。如果可以进行批处理（batching），则重复执行前面的操作，通过可用环将多个 I/O 请求添加到描述符链中。随后，根据添加到可用环中的描述符链头的数量，更新可用环。最后，向设备发送通知，指示“有可用的缓冲区”。

接收设备 I/O 响应的操作

一旦设备完成了 I/O 请求，形成了 I/O 响应，就会更新描述符所指向的缓冲区，并向驱动程序发送已用缓冲区通知。通常，这种通知会采用中断这种高效的机制。设备驱动程序在收到中断后，会对 I/O 响应信息进行后续处理。

第 5 章 virtio-blk 块设备驱动实现

块设备面对的是 I/O 密集型应用程序，需要高存储性能，并且其十分简单，只有简单的读取，写入和刷新命令，对于 virtio-blk 来说没有 SCSI 的开销[25]。其改进对于用户来说是感知十分显著的提升。

5.1 virtio-blk 基本组成结构

5.1.1 设备状态域

设备状态字段提供了设备初始化过程中已完成步骤的低级指示。它的作用类似于连接到控制台上的交通灯，指示每个设备的状态。设备的状态与取值如表4-3所示。设备状态字段起始值为 0，并在重置期间由设备重新初始化为 0。在 Guest 操作系统注意到该 virtio-blk 设备时设置 ACKNOWLEDGE 状态位。当 Guest 操作系统了解如何驱动该设备后，设置 DRIVER 状态位。然后检查 virtio-blk 设备支持的特征功能，将操作系统和驱动程序支持的功能位交集写入设备，然后设置 FEATURES_OK 状态位。驱动程序在此步骤之后就不能再接受新的功能位了。在执行完例如发现设备的 virtqueues，读取并可能写入设备的 virtio 配置空间等操作后，设置 DRIVER_OK 状态位，表示 virtio-blk 设备已处于活跃状态。

5.1.2 特征位

在 virtio 设备中，每个设备都提供其所理解的所有特性。在设备初始化期间，驱动程序会读取这些特性，并告诉设备它接受的子集。重新协商的唯一方法是重置设备。这样可以实现前向和后向兼容性，如果设备增加了新的特性位，旧版驱动程序将不会将该特性位写回设备。同样地，如果驱动程序增加了设备不支持的特性，会发现新特性未被提供。

设备配置空间中的新字段通过提供新的特性位来指示。为了保持特性协商机制的可扩展性，设备不应提供任何它们无法处理的特性位，即使驱动程序接受了它们（尽管根据规范，驱动程序不应接受任何未指定的、保留的或不支持的特性，即使被提供了）。同样，驱动程序不应接受它们不知道如何处理的特性位（尽管根据规范，设备不应在第一次提供任何未指定的、保留的或不支持的特性）。

驱动程序不得接受设备未提供的特性，也不得接受需要另一个未接受的特性的

表 5-1 特征位分配

位	划分
0-23 及 50-127	用于特定设备类型的特性位
24-41	保留给队列和特性协商机制的扩展的特性位
42-49 及 128 以上	保留给未来扩展的特性位

特性。驱动程序必须验证设备提供的特性位。驱动程序必须忽略并且不得接受在规范中未描述的特性位，标记为保留的特性位，不适用于特定传输的特性位和对于特定设备类型未定义的特性位。

virtio-blk 特征位

特征位可以包含关于设备容量、扇区大小、分区信息等方面的信息，这些信息可以帮助驱动程序正确地管理设备上的数据。特征位可以指示设备是否支持读写操作，以及支持的读写命令和模式。特征位可以用于指示设备是否支持缓存控制功能，比如缓存刷新命令。特征位可以包含关于设备支持的数据传输方式的信息，比如支持的最大段数、最大数据传输大小等。特征位可以指示设备是否支持安全擦除、数据加密等安全功能。特征位可以包含有关设备支持的性能优化功能的信息，比如支持的多队列、最佳 I/O 对齐方式等。特征位可以指示设备是否支持特定的协议或命令集，比如 SCSI 命令、TRIM 命令等。

通过特征位，驱动程序可以了解设备的能力和限制，并相应地配置和管理设备。这样可以确保设备和驱动程序之间的兼容性，并充分利用设备提供的功能和性能。

特征位协商过程

特征位协商过程是通用驱动程序所支持的特征与设备所支持的特征协商的过程，最终协商的结果是两者特征的交集。

特征用一个 64 位无符号数表示，理论上每一位都表示一种特征。特征位协商发生在初始化块设备的过程，首先将设备状态设置为空，然后再将设备状态设置为 ACKNOWLEDGE 和 DRIVER，表示驱动程序已经识别到了设备，并且准备好驱动设备。分别读取设备和驱动支持的特征位，做按位与操作，将最终协商结果写入 header 中。设置设备状态 `ACKNOWLEDGE`，`DRIVER`，`FEATURES_OK` 后返回协商结果到驱动程序中。

如果设备不提供它理解的特性，则驱动程序应进入向后兼容模式；否则，必须

表 5-2 virtio-blk 块设备的特征位

特征位	取值	含义
BARRIER	1 « 0	设备支持请求屏障（Legacy）
SIZE_MAX	1 « 1	单个段的最大大小
SEG_MAX	1 « 2	一个请求中的最大段数
GEOMETRY	1 « 4	指定了磁盘样式的几何结构
RO	1 « 5	设备是只读的，任何写入请求都会失败
BLK_SIZE	1 « 6	磁盘块大小。不影响协议中使用的单位（始终为 512 字节）
SCSI	1 « 7	设备支持 SCSI 包命令（Legacy）
FLUSH	1 « 9	设备支持缓存刷新命令
TOPOLOGY	1 « 10	设备提供有关最佳 I/O 对齐的信息。不影响协议中的单元
CONFIG_WCE	1 « 11	0 表示 write-through，1 表示 write-back
MQ	1 « 12	设备支持多队列
DISCARD	1 « 13	设备支持 DISCARD 命令
WRITE_ZEROES	1 « 14	设备支持写零命令
LIFETIME	1 « 15	设备支持提供存储寿命信息
SECURE_ERASE	1 « 16	设备支持安全擦除命令
ZONED	1 « 17	设备是遵循分区存储的设备

设置 FAILED 设备状态位并停止初始化。与此相反，驱动程序不得因为设备提供了它不理解的特性而失败。

设备不得提供需要另一个未提供的特性的特性。设备应接受驱动程序接受的任何有效特性子集，否则当驱动程序写入时，它必须失败以设置 FEATURES_OK 设备状态位。设备不得提供对应于如果被驱动程序接受将不支持的特性的特性位（即使规范禁止驱动程序接受这些特性位）；为了清晰起见，这指的是未在本规范中描述、保留的特性位以及特定传输或特定设备类型不支持的特性位，但这并不排除设备根据未来版本的规范提供此类特性位，如果该规范规定设备支持相应的特性。

如果设备至少成功协商了一组特性（通过在设备初始化期间接受 FEATURES_OK 设备状态位），那么在设备或系统重置后，它不应在重新协商相同的特性集时失败。否则，将干扰从挂起状态恢复和错误恢复。

5.1.3 设备配置空间

virtio-blk config

配置中的信息是驱动相关的一些参数，当设备支持某些特征时，相关的参数就会被设置为配置中的数值。基础配置信息有：capacity（容量），size_max（最大块数），

seg_max（最大段数），blk_size（块大小）。

若协商特征包含 GEOMETRY，则配置信息包含块设备的几何信息：cylinders（柱面数），heads（磁头数），sectors（单磁道扇区数）。对于操作系统和应用程序来说，了解这些信息有助于进行磁盘操作，如分区、格式化和文件系统管理。

若协商特征包含 TOPOLOGY，则配置信息包含块设备的拓扑结果信息：physical_block_size（物理块大小），alignment_offset（对齐偏移量），min_io_size（最小 I/O 规模），opt_io_size（最大 I/O 规模）。对于操作系统和应用程序来说，了解这些信息有助于优化存储访问和性能。

若协商特征包含 CONFIG_WCE，则配置信息包含 writeback，用于表示缓存模式，0 表示 write-through，1 表示 write-back。若协商特征包含 MQ，则配置信息包含 num_queues，表示虚拟队列的数量。

若协商特征包含 DISCARD，则配置信息包含块设备支持的丢弃操作的相关参数：max_discard_sectors 和 max_discard_seg。若协商特征包含 WRITE_ZEROES，则配置信息包含块设备支持的写零操作的相关参数：max_write_zeroes_sectors 和 max_write_zeroes_seg。若协商特征包含 SECURE_ERASE，则配置信息包含块设备支持的丢弃操作的相关参数：max_secure_erase_sectors 和 max_secure_erase_seg。

若协商特征包含 ZONED，则配置信息包含用于描述支持分区的存储设备的相关信息：zone_sectors(每个分区的扇区数量),max_open_zones(同时打开的最大分区数量),max_active_zones(同时处于活动状态的最大分区数量),max_append_sectors(每个分区允许追加写入的最大扇区数量),write_granularity(写入操作的粒度),model(存储设备的模型类型，NONE 表示存储设备不采用分区管理，HM 表示存储设备采用主机管理分区模型，HA 模型表示存储设备采用主机感知分区模型)。

5.2 virtio-blk 设备的关键数据结构

5.2.1 virtio-blk 设备的结构：

```
pub struct VirtIOBlk<'a, H: Hal> {
    header: &'static mut VirtIOHeader,
    queue: VirtQueue<'a, H>,
    capacity: usize,
}
```

header 成员对应着 virtio 设备的共有属性，包括版本号、设备 ID、设备特征等信息。其内存布局和成员变量的含义与此前描述 virt-mmio 设备的寄存器内存布局是一致的。virtQueue 数据结构的定义与此前的 virtqueue 表达的含义一致。

Hal trait 是 virtio_drivers 库中定义的一个 trait，用于抽象出与具体操作系统相关的操作，主要涉及内存分配和虚实地址转换等功能。

```
pub trait Hal {
    fn dma_alloc(pages: usize) -> PhysAddr;
    fn dma_dealloc(paddr: PhysAddr, pages: usize) -> i32;
    fn phys_to_virt(paddr: PhysAddr) -> VirtAddr;
    fn virt_to_phys(vaddr: VirtAddr) -> PhysAddr;
 }
```

5.2.2 请求体 BlkReq

定义一个名为 BlkReq 的结构体，表示块设备的请求。

```
#[repr(C)]
#[derive(AsBytes, Debug)]
pub struct BlkReq {
    type_: ReqType,//请求的类型
    reserved: u32,//保留字段
    sector: u64,//请求涉及的扇区号
}
```

表 5-3 ReqType 取值

状态	取值	状态	取值	状态	取值
In	0	Out	1	Flush	4
GetId	8	GetLifetime	10	Discard	11
WriteZeroes	13	SecureErase	14	Append	15
Report	16	Open	18	Close	20
Finish	22	Reset	24	ResetAll	26

5.2.3 响应体 BlkResp

定义一个名为 BlkResp 的结构体，其包装了结构体 RespStatus 表示设备的响应状态。

```rust
#[repr(C)]
#[derive(AsBytes, Debug, FromBytes, FromZeroes)]
pub struct BlkResp {
    status: RespStatus,
}
```

其中 RespStatus 的定义如下：

```rust
#[repr(transparent)]
#[derive(AsBytes, Copy, Clone, Debug, Eq, FromBytes, FromZeroes,
↪  PartialEq)]
pub struct RespStatus(u8);
impl RespStatus {
    pub const OK: RespStatus = RespStatus(0);//正常状态
    pub const IO_ERR: RespStatus = RespStatus(1);//I/O 错误
    pub const UNSUPPORTED: RespStatus = RespStatus(2);//不支持的操作
    pub const NOT_READY: RespStatus = RespStatus(3);//设备未准备好
}
```

5.3 virtio-blk 中的方法函数

request

```rust
fn request(&mut self, request: BlkReq) -> Result{}
```

该函数将块请求（BlkReq）转换为字节数组格式发送到 Virtio 设备，不带额外数据。阻塞当前线程，直到设备处理完请求并返回响应（BlkResp）。接收并解析设备的响应，将响应状态作为函数的返回值。该函数通过阻塞等待的方式，确保在收到设备响应后才返回，适用于需要同步 I/O 操作的场景。request 方法主要用于刷新 flash 操作。

请求读 request_read

```rust
fn request_read(&mut self, request: BlkReq, data: &mut [u8]) -> Result {}
```

该函数将一个包含数据缓冲区的块请求发送到 Virtio 设备，包括给定的数据，并接收设备的响应。函数的主体首先创建一个 BlkResp 类型的默认响应。然后将请求添加到队列中，并等待响应。最后，函数返回响应的状态，将其转换为 Result 类型的结果。

request_read 方法主要用于封装读操作，包括同步读和异步读。

请求写 request_write

```
fn request_write(&mut self, request: BlkReq, data: &[u8]) -> Result {}
```

该函数接收一个 BlkReq 类型的请求和一个字节切片作为数据，并返回一个 Result 类型的结果。这个函数的主要功能是将给定的请求和数据发送到设备，并等待设备的响应。

首先，函数创建了一个名为 resp 的变量，它是 BlkResp 类型的默认实例。然后，将请求和数据添加到队列中，然后通知设备有新的请求，接着等待设备的响应，最后从队列中弹出响应。

request_write 方法主要用于封装写操作，包括同步写和异步写。

刷新 flush

```
pub fn flush(&mut self) -> Result {}
```

该函数在设备支持的情况下，向设备发送一个刷新请求。

首先检查 negotiated_features 中是否包含 BlkFeature::FLUSH 特征，若包含则代表驱动和设备均支持刷新操作。如果设备支持刷新操作，那么函数会创建一个 BlkReq 类型的请求，请求类型为 ReqType::Flush，然后调用 request 方法发送这个请求。如果设备不支持刷新操作，函数返回一个 Ok(())，表示函数执行成功但没有产生有意义的结果。

阻塞读 read_blocks

```
pub fn read_blocks(&mut self, block_id: usize, buf: &mut [u8]) -> Result
↪ {}
```

该函数的主要目的是从设备中读取一个或多个块的数据，并将这些数据读入到给定的缓冲区中，阻塞直到读取完成或出现错误。

函数接受两个参数：block_id 和 buf。block_id 是一个 usize 类型的值，表示要读取的块的 ID。buf 是一个可变引用到字节切片，表示用于存储读取的数据的缓冲区。

首先使用两个断言（assert_ne! 和 assert_eq!）来检查缓冲区的长度。第一个断言确保缓冲区的长度不为零，第二个断言确保缓冲区的长度是 SECTOR_SIZE 的倍数。

这是因为设备的数据是按块存储的，每个块的大小是 SECTOR_SIZE，所以读取的数据必须是 SECTOR_SIZE 的整数倍。

然后，函数调用封装的 request_read 方法来发送一个读取请求到设备。这个请求是一个 BlkReq 结构体的实例，其中 type_ 字段被设置为 ReqType::In，表示这是一个输入（读取）请求，reserved 字段被设置为 0，sector 字段被设置为 block_id（需要转换为 u64 类型）。

阻塞写 write_blocks

```
pub fn write_blocks(&mut self, block_id: usize, buf: &[u8]) -> Result {}
```

该方法的作用是将给定的缓冲区内容写入到一个或多个块中。阻塞直到写入完成或出现错误。

这个方法接收两个参数：一个是 block_id，表示要写入的块的 ID；另一个是 buf，这是一个字节切片，包含了要写入的数据。

首先，方法使用 assert_ne! 和 assert_eq! 宏来检查 buf 的长度。assert_ne!(buf.len(), 0) 确保 buf 不为空，assert_eq!(buf.len() % SECTOR_SIZE, 0) 确保 buf 的长度是 SECTOR_SIZE 的倍数。如果这些断言失败，程序将在此处停止执行并抛出 panic。

然后，方法检查 negotiated_features 是否包含 BlkFeature::RO。如果包含，说明设备是只读的，方法将打印一条信息并返回 Ok(())。否则，方法将调用封装的 request_write 来写入数据。

在创建 BlkReq 实例时，设置 type_ 为 ReqType::Out，表示这是一个输出请求，设置 sector 为 block_id，表示要写入的块的 ID。其他的字段使用 Default::default() 方法的返回值进行初始化。

异步读 read_blocks_nb 和 complete_read_blocks

```
pub unsafe fn read_blocks_nb(&mut self,block_id: usize,req: &mut
↪  BlkReq,buf: &mut [u8],resp: &mut BlkResp,) -> Result<u16> {}
pub unsafe fn complete_read_blocks(&mut self,token: u16,req: &BlkReq,buf:
↪  &mut [u8],resp: &mut BlkResp,) -> Result<()> {}
```

read_blocks_nb 函数的作用是读取块设备的数据。它接受一个块 ID、一个 BlkReq 请求、一个用于存储数据的缓冲区以及一个 BlkResp 响应。首先，它会检查缓冲区的长度是否为 0，并且是否是扇区大小的整数倍。然后，它会创建一个新的 BlkReq

请求，其中类型为 ReqType::In，保留字段为 0，扇区为传入的块 ID。它将向 VirtIO 块设备提交请求，并返回一个标识第一个描述符在链中位置的令牌。如果没有足够的描述符可供分配，则返回 [Error::QueueFull]。然后调用者可以使用返回的令牌调用 peek_used 来检查设备是否已经完成处理请求。一旦设备完成处理，调用者必须在读取响应之前使用相同的缓冲区调用 complete_read_blocks。

complete_read_blocks 函数的作用是完成由 read_blocks_nb 启动的读取操作。它接受一个令牌、一个 BlkReq 请求、一个用于存储数据的缓冲区以及一个 BlkResp 响应。它会从队列中弹出使用过的请求和缓冲区，并将响应的状态转换为 Result 类型。

需要注意的是，当 read_blocks_nb 返回令牌时，必须再次传递相同的缓冲区，作为参数传递给 complete_read_blocks。这是因为 read_blocks_nb 和 complete_read_blocks 之间存在数据依赖，如果传递的缓冲区不同，可能会导致数据错误。即使在此方法返回后，req、buf 和 resp 仍由底层的 VirtIO 块设备借用。因此，调用者有责任确保在请求完成之前不访问它们，以避免数据竞争。

异步写 write_blocks_nb 和 complete_write_blocks

```
1 pub unsafe fn write_blocks_nb(&mut self,block_id: usize,req: &mut
  ↪  BlkReq,buf: &[u8],resp: &mut BlkResp,) -> Result<u16> {}
2 pub unsafe fn complete_write_blocks(&mut self,token: u16,req:
  ↪  &BlkReq,buf: &[u8],resp: &mut BlkResp,) -> Result<()> {}
```

write_blocks_nb 函数用于提交一个写入请求，但它会立即返回，不会等待写入操作完成。这是一个非阻塞的写入操作。函数的参数包括：要写入的第一个块的标识符。驱动程序可以用于发送给设备的请求的缓冲区，与其他缓冲区一样，直到相应的 complete_write_blocks 调用之前，它需要是有效的（并且未被其他地方使用）。内存中包含要写入块的数据的缓冲区。其长度必须是 SECTOR_SIZE 的非零倍数。调用者提供的一个可变引用，用于保存请求的状态。调用者只能在请求完成后安全地读取该变量。

write_blocks_nb 函数首先会检查 buf 的长度是否为非零且是 SECTOR_SIZE 的倍数。然后，它会创建一个 BlkReq 对象，并将其添加到队列中。最后，函数会返回一个令牌。

complete_write_blocks 函数用于完成由 write_blocks_nb 启动的写入操作。当 write_blocks_nb 返回令牌时，必须再次传递与传递给该方法的相同的缓冲区。函数会从队

列中弹出已使用的令牌，并将其状态转换为 resp.status。

5.4 初始化 virtio-blk 设备

virtio-blk 设备的初始化过程与一般的 virtio 设备初始化过程相似。

首先，将设备重置，以确保设备处于已知的初始状态。接着，设置 ACKNOWLEDGE 状态位，通知客操作系统设备已被识别。然后，设置 DRIVER 状态位，表明客操作系统已准备好驱动该设备。

在读取和配置设备特性位的过程中，首先读取设备特性位，以了解其支持的功能。然后，写入客操作系统和驱动程序能够处理的特性位子集至设备。在此过程中，驱动程序可以读取（但不得写入）设备特定配置字段，以验证驱动程序是否可以支持设备。

随后，设置 FEATURES_OK 状态位，以指示驱动程序已完成特性协商。此后，驱动程序不得接受新的特性位。接下来，重新读取设备状态，以确认 FEATURES_OK 位仍然设置。如果未设置，则表明设备不支持所选特性，设备无法使用。

在初始化过程中，virtio-blk 设备驱动程序会读取设备的配置空间，以获取设备的扇区数量、扇区大小和总容量等信息。然后，需要调用 VirtQueue::new 成员函数来创建虚拟队列 VirtQueue 数据结构的实例，以便进行后续的磁盘读写操作。该函数主要完成以下任务：设定虚拟队列的描述符条目数为 16，计算满足描述符表、可用环和已用环所需的物理空间大小，分配物理空间，创建虚拟队列，调用 VirtIOHeader.queue_set 函数将虚拟队列的相关信息（如内存地址）写入 virtio-blk 设备的 MMIO 寄存器中，并初始化 VirtQueue 实例中各个成员变量的值，主要包括 dma、desc、avail 和 used 等。

最后，设置 DRIVER_OK 状态位，指示设备已准备好使用，此时设备进入“实时”状态。驱动程序不得在设置 DRIVER_OK 位之前向设备发送任何缓冲区可用通知。

完成这些步骤后，virtio-blk 设备和设备驱动之间的虚拟队列接口就建立了，可以进行 I/O 数据读写操作了。如果任何步骤不可恢复地失败，驱动程序应设置 FAILED 状态位，表示已放弃初始化过程。驱动程序不得在设置 FAILED 位后继续初始化。不过，它可以稍后重置设备以再次尝试初始化。

5.5 virtio-blk 设备的 I/O 操作

在操作系统的 virtio-blk 驱动与 virtio-drivers crate 中 virtio-blk 裸机驱动对接的过程中，需要封装 virtio-blk 裸机驱动的基本功能，以完成以下服务：

- 读磁盘块并挂起发起请求的进程/线程；
- 写磁盘块并挂起发起请求的进程/线程；
- 处理 virtio-blk 设备发出的中断，并唤醒相关等待的进程/线程。

在此过程中，virtio-blk 驱动程序发起的 I/O 请求包含操作类型（读或写）、起始扇区（块设备的最小访问单位的一个扇区的长度 512 字节）、内存地址以及访问长度。请求处理完成后返回的 I/O 响应仅包含结果状态（成功或失败），以及读操作请求的读出扇区内容。

在内存中，一个 I/O 请求的数据结构分为三个部分：Header（请求头部，包含操作类型和起始扇区）、Data（数据区，包含地址和长度）、Status（结果状态）。这些信息分别存放在三个 buffer 中，因此需要三个描述符。

virtio-blk 设备使用 `VirtQueue` 数据结构表示虚拟队列进行数据传输。该数据结构主要由三段连续内存组成：描述符表 `Descriptor[]`、环形队列结构的 `AvailRing` 和 `UsedRing`。驱动程序和 virtio-blk 设备都能访问到这个数据结构。

描述符表由一系列固定长度为 16 字节的描述符组成，其个数与环形队列的长度相等。

对于用户进程发出的 I/O 请求，经过系统调用、文件系统等一系列处理后，最终会形成对 virtio-blk 驱动程序的调用。

完整的 virtio-blk I/O 写请求过程可以分为以下几个步骤，这些步骤包括驱动程序和设备之间的交互，以确保数据正确传输和响应。

首先，一个完整的 virtio-blk I/O 写请求由三个主要部分组成：表示 I/O 写请求信息的结构体 `BlkReq`、要传输的数据块 `buf`、以及表示设备响应信息的结构体 `BlkResp`。这三部分需要使用三个描述符来表示。在驱动程序处理阶段，首先调用 `VirtQueue.add` 函数，从描述符表中申请三个空闲描述符，每个描述符指向一个内存块。驱动程序填写上述三部分的信息，并将这三个描述符连接成一个描述符链表。

接下来，驱动程序调用 `VirtQueue.notify` 函数，通过写 MMIO 模式的 `queue_notify` 寄存器，向 virtio-blk 设备发出通知，指示有新的 I/O 请求待处理。在设备处理阶段，virtio-blk 设备接收到通知后，通过比较 `last_avail` 和 `AvailRing` 中的 `idx` 来判断是否有新的请求。如果有新的请求，设备从描述符表中找到该 I/O 请求对应的描述符链，获取完整的请求信息，并执行存储块的 I/O 写操作。设备完成 I/O 写操作后，将已完成 I/O 的描述符放入 UsedRing 对应的 ring 项中，并更新 `idx`，表明已经处理了一个请求响应。如果设置了中断机制，设备会产生中断通知操作系统响应中断。

最后，在驱动程序处理阶段，通过轮询或中断机制，驱动程序判断是否有新的响应。如果有新的响应，驱动程序取出响应，并回收完成响应的三个描述符。最终，驱动程序将结果返回给用户进程。

I/O 读请求的处理过程与 I/O 写请求的处理过程几乎一样，唯一的区别在于 BlkReq 的内容不同。在写操作中，`req.type_` 是 `ReqType::Out`，而在读操作中，`req.type_` 是 `ReqType::In`。

基于轮询的 I/O 访问方式效率相对较低，因此需要实现基于中断的 I/O 访问方式。相比于不支持中断的 `write_block` 函数，`write_block_nb` 函数更加简单，发出 I/O 请求后直接返回。`read_block_nb` 函数的处理流程与此类似。响应中断的 ack_interrupt 函数只完成了基本的 virtio 设备中断响应操作。在 virtio-drivers 中实现的 virtio 设备驱动不包含进程、条件变量等操作系统关键要素，只与操作系统内核对接，因此要完整实现基于中断的 I/O 访问方式，需要在操作系统内核中进行进一步的实现。

第 6 章 正确性测试

6.1 单元测试

在单元测试中，对块设备驱动程序中的函数和方法进行测试，以验证其在模拟输入环境下的输出正确性。总体的测试流程如图6-1所示。

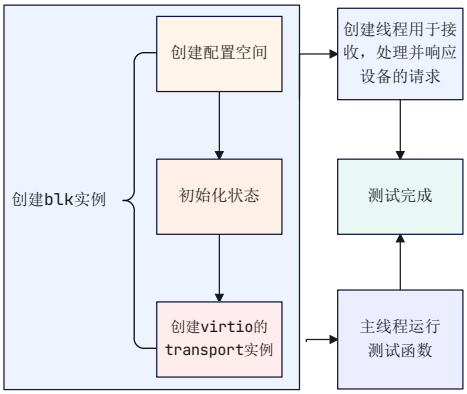

图 6-1 测试框架

读取配置测试

按照 virtio-blk 设备所需的参数（包括 config、state 和 transport）创建一个 virtio-blk 块设备。然后读取块设备的各个字段，与初始值进行比较，以判断这些字段是否相等，从而验证配置信息是否正确写入。本质上，这个过程是对 virtio-blk 块设备进行初始化和创建的过程。测试结果如图6-2所示。

```
running 1 test
test device::blk::tests::config ... ok

successes:

---- device::blk::tests::config stdout ----
Average execution time per request: 3.394 microseconds
Config test passed.


successes:
    device::blk::tests::config

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.01s
```

图 6-2 读取配置测试

获取设备号测试

首先，初始化块设备的配置空间，创建设备状态，并初始化虚拟传输层。在完成上述准备工作后，创建 virtio-blk 块设备实例，并启动线程以模拟设备的等待状态和

处理请求的能力。该线程处理一个获取设备 ID 的请求，在成功获取设备 ID 后，对其进行验证。测试结果如图6-3所示。

```
Device waiting for a request.
Transmit queue was notified.
Average execution time per request: 138.08 microseconds
Device ID test passed.


successes:
    device::blk::tests::device_id

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.14s
```

图 6-3 获取设备号测试

刷新测试

flush() 函数模拟了一个 VirtIO 块设备的刷新操作，分为设备配置、设备模拟和刷新操作三个部分。首先，初始化并配置块设备的必要参数，以便为后续操作做好准备。

在设备模拟部分，首先创建了一个代表 VirtIO 块设备的 VirtIOBlk 结构体实例 blk。接着，启动一个新线程，以模拟设备等待刷新请求的过程。在该线程中，首先打印出设备正在等待请求的信息，然后调用 State::wait_until_queue_notified 函数等待队列被通知。一旦队列被通知，打印出队列被通知的信息，然后锁定设备状态，并调用 read_write_queue 函数处理刷新请求。在处理刷新请求的过程中，首先断言请求的类型是刷新请求，然后创建一个包含响应状态的响应结构体，最后返回这个响应，以完成刷新请求的处理。

在刷新操作部分，调用 blk.flush().unwrap(); 函数以请求设备刷新数据。接着等待设备模拟线程结束，以确保刷新操作完整执行。测试结果如图6-4所示。

```
Device waiting for a request.
Transmit queue was notified.
Average execution time per request: 129.81 microseconds
Flush test passed.


successes:
    device::blk::tests::flush

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.13s
```

图 6-4 刷新测试

读写测试

read() 函数模拟了一个 VirtIO 块设备的读操作。这个函数主要分为三个部分：设备配置、设备模拟和读操作。其目的是验证 VirtIO 块设备能否正确地处理读请求并返回预期的数据。

在设备配置部分，首先创建一个 BlkConfig 结构体实例 config_space，该实例包含了 VirtIO 块设备的配置信息，如容量、大小、几何信息和拓扑信息等。接着，创建一个 State 结构体实例 state，其中包含了设备的状态信息，例如队列状态。最后，创建一个 FakeTransport 结构体实例 transport，该实例模拟了设备的传输层，包含了设备类型、最大队列大小、设备特性、配置空间和设备状态等信息。

在设备模拟部分，创建一个代表 VirtIO 块设备的 VirtIOBlk 结构体实例 blk。然后，启动一个新线程来模拟设备等待读请求的过程。在这个线程中，首先打印出设备正在等待请求的信息，然后调用 State::wait_until_queue_notified 函数等待队列被通知。一旦队列被通知，打印出队列被通知的信息，然后锁定设备状态，并调用 read_write_queue 函数处理读请求。在处理读请求的过程中，首先断言请求的类型和扇区号，然后创建一个包含读取数据和响应状态的响应结构体，最后返回该响应。

在读操作部分，首先创建一个缓冲区 buffer，然后调用 blk.read_blocks 函数，从设备读取一个块的数据到该缓冲区中。接着，断言缓冲区中的数据是否与预期的数据一致。最后，等待设备模拟线程结束，以确保整个读操作完整执行。

写操作与读操作大同小异，不同之处在于数据的流向和处理方式。在读操作中，数据从设备读取到内存，而在写操作中，数据从内存写入到设备。此外，写操作结束后通常会紧接着一个刷新操作，以确保将暂存在内存中的缓冲区数据写入到持久性存储中。通过这种方式，可以验证 VirtIO 块设备在不同操作场景下的正确性和稳定性。测试结果如图6-5，6-6所示。

```
Device waiting for a request.
Transmit queue was notified.
Average execution time per request: 226.663 microseconds
Read test passed.


successes:
    device::blk::tests::read

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.23s
```

图 6-5 读测试

```
Device waiting for a request.
Transmit queue was notified.
Average execution time per request: 142.782 microseconds
Write test passed.


successes:
    device::blk::tests::write

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.15s
```

图 6-6 写测试

异步读写测试

```
let token = unsafe { blk.read_blocks_nb(42, &mut request, &mut buffer,
↪  &mut response) }?;
assert_eq!(blk.peek_used(), Some(token));
unsafe {
blk.complete_read_blocks(token, &request, &mut buffer, &mut response)?;
}
```

首先，向 VirtIO 块设备提交请求，并返回一个标识第一个描述符在链中位置的令牌。如果没有足够的描述符可供分配，则返回 Error::QueueFull 错误。然后，调用者可以使用返回的令牌调用 peek_used 函数来检查设备是否已经完成处理请求。一旦设备完成处理，调用者必须在读取响应之前使用相同的缓冲区调用 complete_read_blocks 函数。测试结果如图6-7所示。

即使在该函数返回后，req、buf 和 resp 仍由底层的 VirtIO 块设备借用。因此，调用者有责任确保在请求完成之前不访问它们，以避免数据竞争。

```
running 1 test
test src/device/blk.rs - device::blk::VirtIOBlk<H,T>::read_blocks_nb (line 294) ... ok

successes:

successes:
    src/device/blk.rs - device::blk::VirtIOBlk<H,T>::read_blocks_nb (line 294)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7 filtered out; finished in 0.43s
```

图 6-7 异步读测试

写操作与读操作的测试环节几乎相同，主要区别在于请求类型和数据流向。在读操作中，请求类型为 ReqType::In，数据从设备读取到内存。而在写操作中，请求类型为 ReqType::Out，数据从内存写入到设备。VirtIO 会根据请求类型的不同执行不同的操作。

```
successes:
    device::blk::tests::config
    device::blk::tests::device_id
    device::blk::tests::flush
    device::blk::tests::read
    device::blk::tests::write

test result: ok. 5 passed; 0 failed;
```

图 6-8 单元测试

6.2 裸机环境集成测试

设计一个用于探测和测试 VirtIO 虚拟设备的程序，其流程如图6-9所示。从设备树（Device Tree）中提取设备信息，并对其进行遍历以识别与 VirtIO 相关的设备节点。对于发现的 VirtIO 块设备节点，程序将创建块设备驱动实例，通过循环多次写入和读取数据块来验证块设备的读写功能，并确保写入的数据和读取的数据一致。测

试过程覆盖了设备驱动程序从初始化到实际数据操作的各个关键环节，可验证驱动程序的基本功能和稳定性。

函数 init_dt(dtb: usize) 使用给定的设备树基地址（dtb）来创建一个设备树对象，并调用 walk_dt(fdt) 来遍历设备树节点。在 walk_dt(fdt) 函数中，通过检查每个节点的兼容性属性，如果检测到兼容属性为”virtio,mmio”，则调用 virtio_probe(node) 函数进行进一步处理。在 virtio_probe(node) 函数中，首先获取设备的基地址和大小，并创建一个 VirtIOHeader 对象。使用 MmioTransport::new(header) 创建 VirtIO MMIO 传输实例，然后调使用 virtio_device(transport) 函数进行设备类型识别，识别到块设备类型，调用 virtio_blk(transport) 函数创建 VirtIOBlk 实例，并进行块设备的读写测试。测试通过两种数据缓冲区进行，即一个输入缓冲区 input 和一个输出缓冲区 output，每个缓冲区大小为 512 字节。在一个循环中（共 32 次）输入缓冲区填充为当前循环索引值，调用 blk.write_blocks(i, &input) 将输入缓冲区的数据写入到第 i 个块，调用 blk.read_blocks(i, &mut output) 从第 i 个块读取数据到输出缓冲区，使用 assert_eq!(input, output) 断言输入缓冲区和输出缓冲区的数据是否一致来判断块设备驱动的工作状况。

测试结果如图6-10所示，读取遍历设备树，探测到“virtio,mmio”类型，识别到类型是“Block”，读取其配置，是大小为 16KB 的 block device，其块大小为 512B。

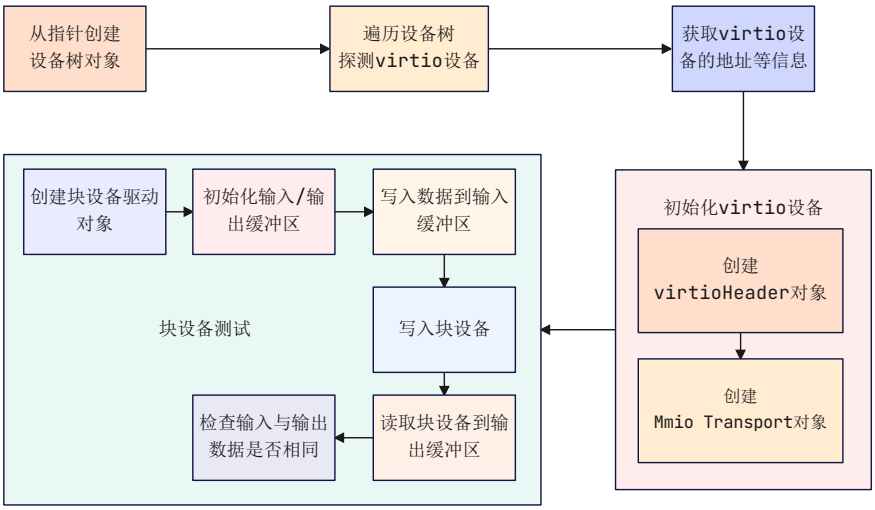

图 6-9 裸机测试框架

```
[ INFO] device tree @ 0x87000000
[ INFO] walk dt addr=0x10008000, size=0x1000
[ INFO] Device tree node virtio_mmio@10008000: Some("virtio,mmio")
[ INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Block, version Legacy
[ INFO] config: 0x10008100
[ INFO] found a block device of size 16KB
[ INFO] blk_size: 512
[ INFO] virtio-blk test finished
```

图 6-10 集成测试

6.3 Alien 系统测试

6.3.1 操作系统对接 virtio-blk 的准备

首先在 Alien 的 Cargo.toml 中添加本研究的 virtio-blk 依赖。如图6-11所示。

```
virtio-drivers = { git = "https://github.com/semidry/virtio_crate.git" }
```

图 6-11 在 Alien 中添加依赖

定义 Trait，用于满足对块设备驱动的 I/O 访问要求。

```rust
pub trait BlkDevice: Send + Sync + Any {
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    fn write_block(&self, block_id: usize, buf: &[u8]);
    // fn handle_irq(&self);
}
```

创建结构体 BLKDEVICE，用于表示块设备，其中包含设备编号和一个实现了 BlkDevice Trait 的设备。然后为其实现例如创建和获取设备号的方法以及实现 VfsFile 和 VfsInode 的特征。VfsFile 特征定义了与文件操作相关的方法，例如从指定偏移量读取数据到缓冲区，将缓冲区的数据写入指定偏移量，轮询文件的事件，设备控制操作，刷新文件的缓冲区，同步文件的缓冲区到存储设备。VfsInode 特征定义了与索引节点相关的操作，例如返回或设置索引节点的类型（例如 CharDevice 表示字符设备），获取索引节点的属性。这些特征由 Alien 操作系统定义，由该操作系统的开发者自行决定实现方式，这里仅进行最简单的实现。

```rust
pub struct BLKDEVICE {
    device_id: DeviceId,
    device: Arc<dyn BlkDevice>,
}
impl BLKDEVICE {
    ...
}
impl VfsFile for BLKDEVICE {
    ...
}
impl VfsInode for BLKDEVICE {
    ...
}
```

建立用于表示 virtio_blk 设备的全局变量 BLOCK_DEVICE。

```rust
pub static BLK_DEVICE: Once<Arc<dyn BlkDevice>> = Once::new();
```

接着创建结构体 VirtIOBlkDeviceWrapper 表示块设备驱动，其内包含着 virtio-blk 的实例。接着为 VirtIOBlkDeviceWrapper 实现 BlkDevice 的特征以及 virtio-blk 内实现的方法，当调用 I/O 访问方法时会调用 virtio 中的同名方法。例如当 Alien 中需要进行 I/O 操作时，调用 VirtIOBlkDeviceWrapper 实现的方法，其方法是调用其包裹着的 virtio-blk 驱动实例，实际上的操作都由 virtio-blk 完成。

```rust
pub struct VirtIOBlkDeviceWrapper {
    blk: Mutex<VirtIOBlk<HalImpl, MmioTransport>>,
}
impl VirtIOBlkDeviceWrapper {
    ...
}
impl BlkDevice for VirtIOBlkDeviceWrapper {
    ...
}
```

在 Makefile 中为 qemu 模拟器添加虚拟块设备

```
-drive file=$(IMG),if=none,format=raw,id=x0
-device virtio-blk-device,drive=x0
```

6.3.2 在 Alien 中启动块设备

在内核态的主函数中，devices::init_device() 函数被视为启动设备的切入点。在此初始化函数中，首先调用了 platform_dtb_ptr() 函数以获取设备树块（DTB）的指

针。随后，将该指针转换为字节指针，并传递给 Fdt::from_ptr() 函数，以创建一个 Fdt 对象。接着，调用 dtb.probe_virtio() 方法，如果探测到 Virtio MMIO 设备，则调用 init_virtio_mmio() 函数进行初始化，其中块设备属于 Virtio MMIO 设备。

在 init_virtio_mmio() 函数中，对设备树进行遍历，对每种设备执行相应的初始化操作。从 DeviceInfo 中提取设备的基地址 paddr。然后，尝试将该物理地址转换为指向 VirtIOHeader 结构体的非空指针，利用此指针创建一个 MmioTransport 实例。例如，对于块设备，其类型为 DeviceType::Block，字面值为 2，因此调用 init_block_device() 进行初始化。

对于 Virtio MMIO 设备，通过传递设备树参数，使用 VirtIOBlkDeviceWrapper::-from_mmio() 函数创建块设备，同时利用 Arc::new 创建块设备驱动。最后，利用 blk::init_blk_device 函数创建全局变量 BLK_DEVICE。

完成上述所有操作后，qemu 模拟器中设备树上的块设备以及块设备驱动已成功加载，并能够被操作系统所识别，如图6-12所示，加载出块设备的地址并初始化成功，就可以对块设备进行操作了。然后进入了 Alien 的控制台，如图6-13所示表明 Alien 正常启动。

```
[0] Init blk device, base_addr:0x10008000,irq:8
[0] Init blk device success
```

图 6-12 块设备被 Alien 所识别

```
[0] Initrd populate success
[0] Init filesystem success
[0] ++++ setup interrupt ++++
[0] ++++ setup interrupt done, enable:true ++++
[0] Init task success
[0] Begin run task...
[0] kthread_init start...
Init process is running
Alien:/#
```

图 6-13 Alien 正常启动并进入控制台

如果 virtio-blk 块设备驱动未与 Alien 对接成功，则编译时会报缺少模块错误，无法正常启动，如图6-14所示。

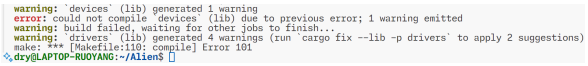

图 6-14 Alien 中接入 virtio-blk 块设备驱动失败

6.4 rCore 系统测试

6.4.1 操作系统对接 virtio-blk 设备初始化过程

在 rCore 的 Cargo.toml 中添加本研究的 virtio-blk 依赖，如图6-15所示。

```
virtio-drivers = { git = "https://github.com/semidry/virtio_old.git" }
```

图 6-15 在 rCore 中添加依赖

尚未将 virtio_drivers 模块与操作系统内核进行集成。下一步需要在操作系统中封装 virtio-blk 设备，以便操作系统内核能够识别和使用该设备。为此，首先需要建立用于表示 virtio_blk 设备的全局变量 BLOCK_DEVICE。

```rust
pub struct VirtIOBlock {
    virtio_blk: UPIntrFreeCell<VirtIOBlk<'static, VirtioHal>>,
    condvars: BTreeMap<u16, Condvar>,
}
```

在操作系统中表示 virtio_blk 设备的全局变量 BLOCK_DEVICE 的类型是 VirtIOBlock。这个全局变量封装了来自 virtio_drivers 模块的 VirtIOBlk 类型，使得操作系统内核能够通过 BLOCK_DEVICE 全局变量来访问 virtio_blk 设备。

VirtIOBlock 结构中包含了 condvars: BTreeMap<u16, Condvar> 条件变量结构，用于在进程等待 I/O 读或写操作完成之前挂起进程。每个条件变量对应着一个虚拟队列条目的编号，意味着每次 I/O 请求都会绑定一个条件变量，使得发出请求的线程/进程可以被挂起[26]。

VirtioHal 结构实现了 virtio_drivers 模块定义的 Hal trait，提供了 DMA 内存分配和虚实地址映射等操作，使得 virtio_drivers 模块中的 VirtIOBlk 类型能够得到操作系统的服务。

```
impl Hal for VirtioHal {
    fn dma_alloc(pages: usize) -> usize {
        let pa: PhysAddr = ppn_base.into();
        pa.0
    }
    fn dma_dealloc(pa: usize, pages: usize) -> i32 {0}
    fn phys_to_virt(addr: usize) -> usize {addr}
    fn virt_to_phys(vaddr: usize) -> usize {vaddr}
}
```

6.4.2 操作系统对接 virtio-blk 设备 I/O 处理

操作系统中的文件系统模块与操作系统中的块设备驱动程序 VirtIOBlock 直接进行交互。VirtIOBlock 驱动程序封装了 virtio-drivers 模块中实现的 virtio_blk 设备驱动。以下是操作系统如何与 virtio_blk 设备驱动进行对接，并完成基于中断机制的 I/O 处理过程。

首先，需要在文件系统中扩展对块设备驱动的方法。这体现在 `BlockDevice` trait 的新定义中增加了 `handle_irq` 方法。操作系统的 virtio_blk 设备驱动程序中的 `VirtIOBlock` 已实现了该方法，并且支持既支持轮询方式，也支持中断方式的块读写操作。

```
pub trait BlockDevice: Send + Sync + Any {
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    fn write_block(&self, block_id: usize, buf: &[u8]);
    fn handle_irq(&self);// 增加对块设备中断的处理
}
```

操作系统还需要对整体的中断处理过程进行调整，以支持基于中断方式的块读写操作。当系统发生中断时，`BLOCK_DEVICE.handle_irq()` 执行的实际上是 `VirtIOBlock` 实现的中断处理方法 `handle_irq()`，从而让等待在块读写的进程/线程得以继续执行[3]。

有了基于中断方式的块读写操作，当某个线程/进程由于块读写操作无法继续执行时，操作系统可以切换到其它处于就绪态的线程/进程执行，从而提升计算机系统的整体执行效率。

```rust
impl BlockDevice for VirtIOBlock {
        fn handle_irq(&self) {
            self.virtio_blk.exclusive_session(|blk| {
                  while let Ok(token) = blk.pop_used() {
                      self.condvars.get(&token).unwrap().signal();
                  }
            });
        }
        fn read_block(&self, block_id: usize, buf: &mut [u8]) {
            if *DEV_NON_BLOCKING_ACCESS.exclusive_access() { // 如果是中断方
            ↪  式
                  let mut resp = BlkResp::default();
                  let task_cx_ptr = self.virtio_blk.exclusive_session(|blk|
                  ↪  {
                      let token = unsafe { blk.read_block_nb(block_id, buf,
                      ↪  &mut resp).unwrap() };
                      self.condvars.get(&token).unwrap().wait_no_sched()
                  });
                  schedule(task_cx_ptr);// 切换线程/进程
            } else { // 如果是轮询方式，则进行轮询式的块读请求
                  self.virtio_blk
                      .exclusive_access()
                      .read_block(block_id, buf)
                      .expect("Error when reading VirtIOBlk");
            }
        }
        //write_block 与 read\_block 读操作的处理过程类似。
        fn write_block(&self, block_id: usize, buf: &mut [u8]){...}
    }
```

在 VirtIOBlock 实现的 BlockDevice 特征中的方法中，handle_irq 方法用于处理设备的中断。在中断处理过程中，它通过调用 exclusive_session 方法获取对虚拟块设备的独占访问权，然后在一个循环中处理所有已完成的请求（通过 blk.pop_used() 获取）。对于每个已完成的请求，它通过信号量通知等待的线程，表示该请求已处理完毕。read_block 方法用于读取指定块设备上的数据块。这个方法根据全局变量 DEV_NON_BLOCKING_ACCESS 决定采用哪种访问方式。若采用中断方式进行非阻塞访问。在这种情况下首先创建一个默认的 BlkResp 响应对象，然后通过独占会话调用设备的 read_block_nb 方法发起非阻塞的读请求，并获取一个令牌。该令牌用于在 condvars 中查找相应的条件变量，并等待不调度线程（wait_no_sched）。完成后，它切换到其他线程或进程继续执行（通过 schedule 方法）。若采用轮询方式进行阻塞访问。则通过独占访问方式调用设备的 read_block 方法读取数据块，CPU 处于忙等待状态并期望操作成功完成。write_block 方法的实现与 read_block 类似。

6.4.3 运行 rCore

进入 os 目录，使用命令 `make run` 启动 rCore，如图6-16所示可以看到 rCore 进入了控制台并且文件系统成功加载出了一系列文件，证明文件系统对接的块设备驱动程序工作正常。输入命令打开对应文件（例如 pipetest），可以看到 rCore 正常运行，并回到了控制台命令行。说明该 virtio-blk 块设备驱动程序可以在 rCore 上正常工作。

```
pipetest
adder_peterson_yield
huge_write_mt
gui_snake
inputdev_event
cat
eisenberg
**************/
Rust user shell
>> pipetest
Read OK, child process exited!
pipetest passed!
>>
```

图 6-16 rCore 加载出文件并进入控制台

如果块设备驱动无法正常使用，其结果应如图6-17所示，内核缺少驱动模块，无法正常编译启动。

```
warning: `os` (bin "os") generated 1 warning
error: could not compile `os` (bin "os") due to 5 previous errors; 1 warning emitted
make: *** [Makefile:60: kernel] Error 101
dry@LAPTOP-RUOYANG:~/virt/rCore-Tutorial-v3/os$
```

图 6-17 若块设备未正确接入操作系统

结　论

本研究设计并实现了一个遵循 Virtio 协议的跨操作系统异步块设备驱动模块。该模块支持更多块设备特性，既能为块设备提供通用服务，又能依据块设备支持的高级特性提供更多支持。同时，该解决方案不依赖任何特定的异步运行时，提供了一种面向底层设备的异步解决方案。模块化设计极大提高了其可移植性。

本研究或能减轻操作系统开发者的开发负担，使其只需导入依赖即可在操作系统中直接使用块设备驱动，从而增强了操作系统的可移植性并减少了开发成本。同时，得益于模块化和低耦合性的设计，本研究也能为操作系统的学习者减轻分析和理解操作系统内核的难度，促进教学工作的便利性。因此，本研究对操作系统开发者和学习者均有积极作用，能够为操作系统领域做出贡献。

由于本研究遵循 Virtio 协议，因此应能与其他 Virtio 协议驱动协调工作。然而，本研究目前仅解决了块设备驱动这一驱动类型，与其他 Virtio 驱动共同工作时，势必会产生一些冗余代码，对于用户操作系统来说会造成资源浪费，尤其是对嵌入式设备来说，这种弊端不可忽视。未来的研究方向希望能将本研究的实现中分离出 Virtio 协议依赖部分，使操作系统开发者在使用多种 Virtio 驱动时无需重复引入冗余代码。进一步分离块设备的实现可以进一步提高模块化程度。

此外，Virtio 规定的许多功能尚未在本研究中实现，这使得对块设备的许多高级特性，例如分区块设备及其独特的空间布局和拓扑信息，无法获得更优支持，可能会阻碍操作系统在调度时的进一步优化。未来的研究将分析块设备的物理特性，逐步实现 Virtio 支持的高级特性。

本研究中的异步解决方案旨在解决 I/O 设备工作时的阻塞问题。尽管如此，仍有进一步优化的空间，即 I/O 命令被挂起后携带唯一的标识符（token）在循环队列中排队。对于零散的 I/O 操作，这种方法对 CPU 利用率的提升较为显著，但当 I/O 操作较为密集时，CPU 仍会因为 I/O 操作未完成而不得不等待。因此，将来可以在队列上做进一步优化，例如多队列。该改进不属于异步本身，但可以提升异步效果。

本研究仍有大量改进工作可以继续完成，其前景十分广阔，具有巨大的实用性和经济价值。其遵循当前广受推崇的 Virtio 协议，是对跨操作系统外设驱动问题提出的一种具有实际意义的可持续改进的现实解决方案。

参考文献

[1] Corbet J, Rubini A, Kroah-Hartman G. Linux device drivers[M]. ” O’Reilly Media, Inc.”, 2005.

[2] Zhao W, Stankovic J A. Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems[C]//1989 Real-Time Systems Symposium. 1989: 156-157.

[3] Rcore-os. An OS which can run on RISC-V in Rust from scratch[Z]. https://github.com/rcore-os/rCore-Tutorial-v3. [Accessed 15-05-2024]. 2020.

[4] Godones. Alien: use modules to build a complete os[Z]. https://github.com/Godones/Alien. [Accessed 15-05-2024]. 2023.

[5] Project contributors E. Embassy Documentation[Z]. https://embassy.dev/book/dev/index.html. [Accessed 15-05-2024]. 2024.

[6] Rust-lang. The Rust RFC Book-Static async fn in traits[Z]. https://rust-lang.github.io/rfcs/3185-static-async-fn-in-trait.html. [Accessed 15-05-2024]. 2021.

[7] Sunface. Rust Course async tokio[Z]. https://course.rs/advance/async/getting-started.html. [Accessed 15-05-2024]. 2023.

[8] Rust-lang. The Rust RFC Book-async$_{a}wait$[Z]. https://rust-lang.github.io/rfcs/2394-async_await.html. [Accessed 15-05-2024]. 2018.

[9] Wadekar A, Swapnil S, Lohani R B. Design and implementation of a universal DMA controller[C]//ICWET ’11: Proceedings of the International Conference & Workshop on Emerging Trends in Technology. Mumbai, Maharashtra, India: Association for Computing Machinery, 2011: 1189-1190.

[10] Scott T A. Illustrating programmed and interrupt driven I/O[C]//Proceedings of the seventh annual CCSC Midwestern conference on Small colleges. 2000: 230-238.

[11] Yang J, Minturn D B, Hady F. When poll is better than interrupt.[C]//FAST: vol. 12. 2012: 3-3.

[12] Freeman A. Files, Streams, and IO[M]//Introducing Visual C# 2010. Berkeley, CA: Apress, 2010: 621-673.

[13] Ansari S, Rajeev S, Chandrashekar H. Packet sniffing: a brief introduction[J]. IEEE Potentials, 2003, 21(5): 17-19.

[14] Ritchie D M. The UNIX System: A Stream Input-Output System[J]. AT&T Bell Laboratories Technical Journal, 1984, 63(8): 1897-1910.

[15] Juszkiewicz K. UNIX Network Programming, Volume 1: The Sockets Networking[J]. IEEE Communications Magazine, 2004, 42(5): 20-21.

[16] Bruguera i Moriscot F. Benchmarking input/output multiplexing facilities of the Linux kernel[J]. 2019.

[17] Bauman E, Ayoade G, Lin Z. A survey on hypervisor-based monitoring: approaches, applications, and evolutions[J]. ACM Computing Surveys (CSUR), 2015, 48(1): 1-33.

[18] Yu Chen Y W. rCore-Tutorial-Book-v3 virtio 设备[Z]. https://rcore-os.cn/rCore-Tutorial-Book-v3/chapter9/2device-driver-3.html. [Accessed 15-05-2024]. 2022.

[19] Kukreja G, Singh S. Virtio based Transcendent Memory[C]//Sandhu Y H D S. Proceedings of 2010 3rd IEEE International Conference on Computer Science and Information Technology VOL.1. Institute of Electrical, 2010: 748-752.

[20] Vara Larsen M. Verifying the Conformance of a Driver Implementation to the VirtIO Specification

[C]//2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2021: 719-720.
[21] Park S, Kim K, Kim H. Ambient Virtio: IO Virtualization for Seamless Integration and Access of Devices in Ambient Computing[J]. IEEE Systems Journal, 2022.
[22] Russell R. virtio: towards a de-facto standard for virtual I/O devices[J]. SIGOPS Oper. Syst. Rev., 2008, 42(5): 95-103.
[23] Likely G, Boyer J. A symphony of flavours: Using the device tree to describe embedded hardware [C]//Proceedings of the Linux Symposium: vol. 2. 2008: 27-37.
[24] Gibson D, Herrenschmidt B. Device trees everywhere[J]. OzLabs, IBM Linux Technology Center, 2006.
[25] He A, Hat R. Virtio-blk performance improvement[C]//KVM Forum. 2012.
[26] Rcore-os. VirtIO guest drivers in Rust.[Z]. https://github.com/rcore-os/virtio-drivers/. [Accessed 15-05-2024]. 2024.

致　谢

值此论文完成之际，我怀着无比感激的心情，向在此过程中给予我帮助和支持的所有人致以诚挚的谢意。

首先，我要感谢我的导师陆慧梅教授和向勇教授。您们的悉心指导和无私奉献使我在学术上取得了长足的进步。无论是在论文的行文排版，还是在研究遇到瓶颈时，您们的指引和建议都为我指明了方向，使我能够顺利完成毕业论文。

同时，我也要感谢我的同学们。感谢你们在学习和生活中给予的帮助和陪伴。在完成毕业设计的这段时间，许多科研上、学习上和生活上的问题在你们的帮助下得以顺利解决，使我的科研工作得以稳步推进。

此外，我要特别感谢我的父母。感谢你们无微不至的关怀和默默无闻的支持。你们不仅在物质上给予了我坚实的保障，更在精神上给予了我巨大的鼓励。你们的理解和包容让我能够全身心地投入到学业中，从不曾有后顾之忧。

最后，我要感谢我的母校北京理工大学。感谢学校为我们提供了良好的学习环境和丰富的学术资源。这里浓厚的学术氛围和多元化的校园文化开阔了我的眼界，培养了我的综合素质，使我在这里得到了全面的发展。在学校的数据库中，我获取了丰富的科研资料，让我能够更好地完成课题研究。

我深知，今天的每一点进步和成就都离不开你们的帮助和支持。再次向你们表示衷心的感谢！