北京理工大學

本科生毕业设计（论文）

跨操作系统的异步串口驱动模块设计与实现

Design and Implementation of Asynchronous Serial Port Driver Module Across Operating Systems

学　　院：	计算机学院
专　　业：	计算机科学与技术
班　　级：	07112005
学生姓名：	林 晨
学　　号：	1120202738
指导教师：	陆慧梅

2024 年 5 月 13 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

跨操作系统的异步串口驱动模块设计与实现

摘　要

在操作系统发展过程中，形形色色的外设为操作系统实现了丰富的功能。而从操作系统的架构上看，驱动程序是操作系统与各种外设进行直接交互的软件组件，设计并实现一个对上层操作系统独立的硬件驱动模块，共操作系统开发人员直接调用，能够帮助其在开发的操作系统中快速实现对硬件的控制。向驱动程序中引入异步机制，能够在高 IO 场景下的提升操作系统的性能和吞吐量，而在一门新兴的系统级编程语言 Rust 中，提供了 Future 结构表示异步任务返回结果的抽象。同时 Rust 语言本身自带的包管理器 Cargo 也为我们实现的异步串口驱动更好地供其他操作系统开发人员使用提供了便利。

在本研究中，我们针对 QEMU 虚拟机中的串口设备，使用 Rust 语言开发了一个异步串口驱动模块，并在其上使用模块化操作系统 Alien 调用该异步串口驱动模块，能够在 Alien 中向串口发起异步读写请求。本研究中开发的异步串口驱动的异步运行时参考了 Embassy 的异步运行时设计，将硬件的读写任务抽象为一个 Task 结构体。读写任务创建后就会立刻返回让 Alien 执行接下来的程序，而等到接收到串口硬件中断后，异步串口驱动模块才会进行实际的读写任务操作。

关键词：异步；串口；Rust；模块化；运行时

Design and Implementation of Asynchronous Serial Port and Network Driver Module Across Operating Systems

Abstract

In the development process of operating systems, a variety of peripherals have implemented rich functions for the operating system. From the perspective of the architecture of the operating system, the driver is a software component that directly interacts between the operating system and various peripherals. Designing and implementing a hardware driver module that is independent of the upper-layer operating system can be directly called by operating system developers, which can help It quickly realizes control of hardware in the developed operating system. Introducing an asynchronous mechanism into the driver can improve the performance and throughput of the operating system in high IO scenarios. In Rust, an emerging system-level programming language, a Future structure is provided to represent the abstraction of the results returned by asynchronous tasks. At the same time, the package manager Cargo that comes with the Rust language itself also provides convenience for the asynchronous serial port driver we implemented to be better used by other operating system developers.

In this study, we used the Rust language to develop an asynchronous serial port driver module for the serial port device in the QEMU virtual machine, and used the modular operating system Alien to call the asynchronous serial port driver module and be able to initiate requests to the serial port in Alien. Asynchronous read and write requests. The asynchronous runtime of the asynchronous serial port driver developed in this study refers to Embassy's asynchronous runtime design and abstracts the hardware read and write tasks into a Task structure. After the read and write task is created, it will immediately return to Alien to execute the next program. After receiving the serial port hardware interrupt, the asynchronous serial port driver module will perform the actual read and write task operation.

Key Words: Asynchronous; serial port; Rust; modularity; runtime

目 录

摘 要 I
Abstract II
第 1 章 绪论 1
1.1 研究背景 1
1.2 国内外研究现状 2
1.3 本文研究内容与贡献 2
1.3.1 研究内容和关键问题 2
1.3.2 本文贡献 3
第 2 章 相关技术介绍 4
2.1 Rust Cargo 和 Crates.io 4
2.1.1 Rust Cargo 4
2.1.2 Crates.io 4
2.2 Rust Future 5
2.2.1 常见的并发编程方式 5
2.2.2 Rust Future 6
2.2.3 Rust 运行时 8
2.3 Embassy 9
2.3.1 Embassy 简介 9
2.3.2 Embassy 异步运行时分析 10
2.4 Alien 12
2.4.1 Alien 简介 12
第 3 章 异步串口驱动模块的设计与实现 14
3.1 串口基本机制 14
3.1.1 串口简介 14
3.1.2 串口的 FIFO 模式 15
3.1.3 串口的中断处理 15
3.2 驱动的整体架构设计 16
3.3 异步运行时设计 18
3.4 接口设计 19
3.4.1 异步串口的初始化 20
3.4.2 异步串口的读写 20
第 4 章 异步串口驱动测试 22
4.1 修改 QEMU 源码使其支持多串口 22
4.2 Alien 使用异步串口驱动 22
4.2.1 配置 QEMU 路径及启动参数 22
4.2.2 双核启动 Alien 23
4.2.3 Alien 进入异步运行时 24
4.2.4 为异步串口支持 DeviceBase 特征 24
4.3 异步串口驱动测试 25

4.3.1 异步串口驱动的初始化 25
4.3.2 异步串口读写 25
4.3 测试结果分析 26
第 5 章 总结和展望 29
5.1 论文总结 29
5.2 未来的相关工作 30
结 论 31
参考文献 32
附 录 A 修改 QEMU 源码使其支持多串口 33
致 谢 35

第 1 章 绪论

1.1 研究背景

在操作系统发展中，外设起到了非常重要的作用。为了满足不同需求而设计的外设，帮助操作系统实现了丰富的功能。从操作系统的架构上看，驱动程序是与各种外设进行直接交互的软件组件。由于外设种类多样，不同厂家对于相同外设的设计也不尽相同，这就导致了操作系统开发人员需要花大量的时间在对硬件手册的阅读以及对外设驱动程序的实现和调试上。但几乎所有的硬件外设，独立于上层的操作系统实现，因此针对于一种特定的硬件，设计并实现一个对上层操作系统独立的硬件驱动模块，供操作系统开发人员直接调用，能够帮助在其开发的操作系统中快速实现对硬件的控制。

从操作系统用户进程的角度来讲，用户进程需要的是对外设进行读写，读写的方式是通过系统调用方式。在读写系统调用执行模型的设计上，主要包括阻塞和非阻塞、同步和异步等执行模型。如果一个系统调用采用同步执行模型时，那么调用该系统调用时，在完成该系统调用的全部任务前，该系统调用都不会返回；而当一个系统调用采用异步执行模型时，那么调用该系统调用时，会立即返回，尽管该系统调用所规定的读写任务还未达成。在计算机中有些 IO 处理比较耗时。在调用这种同步代码时，如果进程在此处长时间等待，会严重影响程序的性能，因此在 IO 密集型的应用场景中，采用异步编程，能够极大程度的提升程序的运行性能。另外，采用无栈协程实现的异步，相比于传统的多线程并发设计，更是省去了操作系统对于上下文，堆栈等进行开辟和切换的开销。

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。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]。 Rust 所使用的包管理工具 Cargo 为使用 Rust 语言开发的项目提供了方便的依赖包管理，能够使操作系统开发人员很容易的使用本项目开发的异步驱动模块。另外，Rust 语言也对异步和协程的相关特性有所支持，能够使我们更加自然地实现一个异步的驱动模块。

1.2 国内外研究现状

异步 I/O 是指在进行事务处理时并发执行输入、输出以及计算操作的能力，可以将逻辑独立的 I/O 操作从指定的动作中分离出来单独执行。[2]简单来说，相较于阻塞的同步 I/O，异步 I/O 是非阻塞的，无需等到 I/O 操作返回相应的 I/O 结果，就可继续执行接下来的操作。

实现异步 I/O 的方式，包括使用操作系统提供的多线程接口，编程语言提供的函数回调、Future 等语言特性编写程序。方兴等人[3]提出了一种基于 WIN32 的多线程异步 I/O 模型，用于解决复杂的多路并发 I/O 问题，并说明了该模型的运行机制及其优越性，这是使用操作系统支持的多线程实现同步。段楠的研究[4]使用 Java 提供的“消息队列”方式进行了异步的网络通讯开发，在语言层面使用语言特性对异步进行了支持。Harris T 的研究[5]中更是将介绍了一组在 C/C++等原生语言中用于可组合异步 IO 的语言结构 AC，为开发人员提供了更方便的异步程序开发。Rust 在语言层面提供了对协程和异步的支持，使得开发人员能够方便的使用 Future 特性开发异步模块，实现异步外设驱动。

使用异步 I/O 接口，在大量 I/O 的应用场景下，能够极大的增加整个系统的吞吐量，使用无栈协程所实现的异步 I/O 相较于用多线程实现的异步 I/O，更是能够避免分配大量的堆栈等内存空间，节省了系统资源。沙泉的研究[6]在嵌入式 Linux 的串行通信中实现了异步事件驱动模型，在一定的硬件和实例测试的环境下说明了异步串口通信程序的设计思路和实现。Zhu L[7]、Kwon G[8]等人的研究分别在自己所设计的监测系统和实时数据归档系统中使用了异步驱动模块，在 I/O 量比较大的应用场景下，提升了整个系统的吞吐率和安全性。

目前，使用 Rust 语言编写操作系统组件逐渐成为趋势。Linux 已经对其众多的硬件驱动模块进行 Rust 语言改造，也进一步印证了使用 Rust 语言可以编写更安全的操作系统正在逐渐成为操作系统领域的共识。在 Rust 语言社区中，基于 Rust 语言对各种硬件设备的驱动开发也正在如火如荼地进行着。

1.3 本文研究内容与贡献

1.3.1 研究内容和关键问题

本研究的主要目标是使用 Rust 语言中对于异步的支持，参照 Embassy 中异步运

行时的实现，针对 QEMU 虚拟机模拟的串口设备，开发跨操作系统的异步串口驱动模块。

本文的研究内容主要包括以下几项：学习 Rust 在异步和模块化方面的相关支持，学习并总结 Embassy 的异步运行时，设计并实现异步串口驱动；在 QEMU 虚拟环境下对 Alien 完成适配，即能够。

本研究的关键问题包括：学习 Rust 语言对于异步的支持 Rust Future、了解 Rust 语言在模块化方面的支持 Rust Cargo、学习和总结已有串口驱动的实现、设计并实现异步串口驱动、学习 Alien 的相关结构并在 QEMU 环境下使用 Alien 调用异步串口驱动、通过正确性测试。

1.3.2 本文贡献

本文基于 Rust 语言，设计了能够在 QEMU 模拟器所模拟的 qemu-system-riscv64 虚拟环境下 AlienOS 和 ArceOS 都能够使用的异步串口驱动模块。本研究完成了异步串口驱动模块中异步运行时的设计与实现，在 QEMU 中添加了多串口的支持使得在支持多个串口的 qemu-system-riscv64 环境下的 Alien 中能够同时使用原有的同步串口驱动和我们所开发的异步串口驱动模块。我们将新创建的使用异步驱动的串口绑定到一个指定的终端，在对应的终端能够看到上层操作系统通过异步串口驱动输出的相关信息，也能通过终端输入完成 OS 创建的读字符任务。

本研究中开发的异步串口驱动模块提升了系统的并发性和效率。采用中断事件触发执行的方式，避免了 CPU 忙等占用大量的 CPU 资源，使得系统在接收到读写请求后依然可以同时执行其他任务；采用 Rust Future 的协程机制，相对于线程等并发实现需要更小的内存开销，完成了对内存资源的节约；同时我们使用 Rust 语言实现异步串口驱动模块，也使得其他开发者能够更方便地使用我们的异步串口驱动模块。

第 2 章 相关技术介绍

2.1 Rust Cargo 和 Crates.io

本部分我们将介绍 Rust 语言在包管理部分的相关技术，包括软件包管理器 Rust Cargo 以及官方的包管理平台 Crates.io，说明使用 Rust 语言开发驱动模块能够更方便地使用他人的已有成果，并使得我们的异步驱动模块能够更方便地被其他有需要的人所使用。

2.1.1 Rust Cargo

Rust 是为了解决 C/C++等底层程序设计语言存在的安全问题，设计的新一代安全系统级编程语言，通过其精心设计的所有权、生命周期等机制，保障了其不会出现 C、C++语言中悬垂指针等危险的内存操作。但除了其本身是一门安全、出色的编程语言之外，Rust 还具有一个名为 Cargo 的构建系统和软件包管理器。Rust 与 Cargo 捆绑在一起，在安装 Rust 时会自动安装 Cargo。

Rust Cargo 为 Rust 开发者提供了便捷、高效和可靠的开发工具和基础设施：在项目初始化时，开发者可以使用 `cargo new` 命令立刻创建一个新的 Rust 项目的基本结构，包括 Cargo.toml 文件和 src 目录，以及 src 目录下一个默认的 main.rs/lib.rs；在项目引用某些依赖项时，开发者可以通过 Cargo.toml 文件指定项目所依赖的外部 crate，并附带上版本信息和其他约束条件，Cargo 会负责下载和管理这些依赖，保证项目的构建和运行环境符合开发者的要求；在构建系统时，开发者可以使用 `cargo build` 命令快速地编译、构建和打包 Rust 项目，同时支持调整构建参数和选项，以及与编译器和链接器的集成；在测试支持方面，Cargo 还内置了对 Rust 项目的测试支持，开发者可以在项目中编写单元测试和集成测试，并使用 `cargo test` 命令来运行这些测试。

同时，在项目的发布方面，开发者可以通过 Cargo 轻松地发布他们编写的 Rust 项目，把他们精心设计的 crate 发布到 crates.io（Rust 包的官方仓库），使其能够提供给其他开发者引用，促进 Rust 生态的健康发展。

2.1.2 Crates.io

Crates.io 是 Rust 编程语言的官方包管理平台，也是 Rust 生态系统中最重要的组成部分之一。在 Crates.io 上，开发者可以发布、发现和共享 Rust crate，从而

促进 Rust 社区的发展与合作。该平台提供了一个中心化的位置，使得开发者们能够更加轻松地查找和使用其他人编写的 crate，同时也方便了开发者分享自己的代码作品。

通过 Rust Cargo 和 Crates.io 的集成，使得开发者能够通过 Cargo 直接安装和管理其在 Crates.io 上发现的 Crate，同时这种紧密的集成也减轻了开发者本身需要在依赖管理上投入的精力，提高了整体的项目开发效率。在本研究所开发的异步串口驱动因此能够更方便地使用其他人已经实现的 Rust crate，而其他操作系统开发者也会更方便地使用本研究的驱动 crate 进行操作系统开发。

2.2 Rust Future

本部分将介绍一些常见的并发编程方式，在对比中分析 Rust Future 所具备的优势，之后介绍有关 Rust 运行时的相关信息。

2.2.1 常见的并发编程方式

常见的并发编程的方式主要包括：线程、绿色线程、函数回调、Promise 等。

（1）使用线程实现并发编程

首先需要肯定的是，操作系统线程肯定有一些相当大的优势，使得我们在讨论“异步”和并发性的时候总是把线程摆在首位。相较于其他方法，使用操作系统提供的线程是非常简单易用的，并且由于操作系统提供的任务切换机制，在不同的线程之间的切换非常快，不需要程序员使用过多的付出便能得到很好的并发性，因此对于大量的并发问题而言，使用操作系统提供的线程通常都是最正确的解决方案。

但也存在一些场景，使用线程机制实现的并发可能会带来一些麻烦：由于操作系统级别线程的堆栈都比较大，如果在类似于负载很重的 web 服务器的情况下，服务器的内存将会被很快的用尽；操作系统有的时候会有很多事情需要处理，可能并不能像程序员所希望的那样快速切换回需要执行的线程。另外，有的操作系统本身可能并不支持线程机制，也就导致了其上的应用程序无法使用线程实现并发编程。

（2）使用绿色线程（Green Thread）实现并发编程

绿色线程是一个完全在用户空间实现的线程系统，因此也被称为用户态线程。在具体的实现上，用户空间中的线程库会把管理线程所需要的数据结构在进程地址空间内部实例化，然后就可以调用线程库来管理用户空间内的多线程。由于绿色线程完全运行在用户空间内，因此不需要内核支持线程机制，内核也并不知道这些现成的存

在。但也正因为这些线程对于内核是不可见的，所以一旦这些线程中有一个被阻塞，则其余的线程均被阻塞。

但由于该特性并不是一个零成本抽象（即在语言中加入该特性后，会对不使用该特性的其他开发者所编写的程序的性能产生影响），于是 Rust 语言的开发者在 Rust 1.0 版本删除了该特性。同时如果我们想要对绿色线程支持许多不同的平台，也很难正确实现。

（3）使用函数回调（callback）实现并发编程

基于回调所实现的方法背后的主要思想为保存指向一组指令的指针，这些指令会在我们希望执行的时候执行。而针对 Rust 语言，这将是一个闭包。

对于函数回调而言，它在大多语言中都易于实现。并且由于不需要记录线程中类似于线程上下文的数据结构，函数回调没有上下文切换的性能开销，大多数情况下在内存上的开销也相对较低。但函数回调的缺点也很明显：首先，每个任务必须保存它以后需要的状态，内存使用量也将随着一系列计算中回调的数量线性增长；并且，有的时候随着回调函数嵌套层数的增加，代码可读性会非常的差，后期并不好维护；此外，由于 Rust 语言自身的所有权机制，在 Rust 语言中使用函数回调时任务之间的状态共享也会是一个难题。

（4）使用 Promise 实现并发编程

通常，Promise 和 Future 两个名词在并发模型中是混用的，无论 Promise 还是 Future 都是对异步计算结果的抽象。对于 Promise 而言，编译器在编译时会将其重写为一个状态机。整个 Promise 中的每个异步事件执行成功/失败后都会更新状态机的状态，随着所有异步事件的依次执行完毕，Promise 会返回整个异步事件的结果。由于 Promise 与 Future 的实现比较相似，我们会在 2.2.2 Rust Future 中会详细介绍这种状态机是如何设置，又是如何工作的。

2.2.2 Rust Future

Rust 中的 Future 类似于 Javascript 中的 promise，是 Rust 语言中的一种表示异步计算结果的抽象。它允许开发者编写异步代码，处理异步任务的完成和错误，并在任务完成时收获一定的结果。Future 是 Rust 中异步编程的基础，通常与 async 和 await 关键字一起使用。

在 Rust 中，Future 作为一个接口在 rust 1.36 加入标准库中，其定义如下：

```rust
pub trait Future {
    /// The type of value produced on completion.
    #[stable(feature = "futures_api", since = "1.36.0")]
    type Output;

    /// Attempt to resolve the future to a final value, registering
    /// the current task for wakeup if the value is not yet available.
    ///
    /// # Return value
    ///
    /// This function returns:
    ///
    /// - [`Poll::Pending`] if the future is not ready yet
    /// - [`Poll::Ready(val)`] with the result `val` of this future if it
    ///   finished successfully.
    ///
    /// Once a future has finished, clients should not `poll` it again.
    ///
    /// When a future is not ready yet, `poll` returns `Poll::Pending` and
    /// stores a clone of the [`Waker`] copied from the current [`Context`].
    /// This [`Waker`] is then woken once the future can make progress.
    /// For example, a future waiting for a socket to become
    /// readable would call `.clone()` on the [`Waker`] and store it.

    #[stable(feature = "futures_api", since = "1.36.0")]
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
pub enum Poll<T> {
    /// Represents that a value is immediately ready.
    Ready(#[stable(feature = "futures_api", since = "1.36.0")] T),

    /// Represents that a value is not ready yet.
    Pending,
}
```

其中 Output 是 Future 完成时返回值关联的类型;接口中的 poll 方法为尽可能地尝试运行一次 Future 得到一个结果，该次运行有可能会返回表示 Future 运行成功的 Poll::Ready(T),或者返回表示 Future 目前还处于阻塞状态的 Poll::Pending。

如果我们想对一个具体的 IO 任务进行异步实现（即实现一个 Future 特征），我们可以创建一个具体的结构体，结构体中有一些有关 IO 任务的相关信息字段，然后

按照上面标准库中的要求为该结构体实现 Future 特征；或者将读写操作实现为一个函数，使用‘async’关键字来标识该函数为异步函数，在编译时，编译器会自动将该函数转化为一个具有对应状态转换机制的状态机，即一个匿名的 Future 类型。‘async’除了上述的在函数前面加上表示该函数为异步函数以外，还可以将一个代码块标识为 Future 类型，并且我们可以像正常处理一个 Future 类型一样，处理这个异步代码块。

有了具体的 Rust Future 对象，我们可以通过使用‘await’关键字在一个异步函数中等待另一个异步函数，即在一个 Future 中等待另一个 Future。如果要想在同步函数中使用一个异步函数（或者说等待一个 Future），我们就需要在调用一些标准库中提供的 block_on()方法，使得我们的同步函数能够阻塞在该 Future 上，等到该 Future 运行完毕，就可以接着运行同步函数了。

2.2.3 Rust 运行时

对于一个 Future 而言，在绝大多数情况下并不能一次执行到终点，为了在对应的事件被触发时让对应的 Future 继续执行，Rust 提供了一套运行时机制。而与 c#，JavaScript，Java，GO 等直接提供一个处理并发的运行时的语言不同，Rust 并没有直接提供这样的运行时，而是需要我们自己创建一个运行时，或者使用一个为我们提供运行时的库。

一个 Rust 的异步运行时可以被分为两部分：执行器（executor）和反应器（reactor），反应器负责通知 Future 可以继续执行，执行器将负责 Future 实际工作。而在一个 Future 的运行过程中，会发生三种事件：轮询、等待和唤醒。轮询指的是执行器会对未处于 Pending 状态的 Future 进行轮询，使得它们能够执行到下一个 Pending 状态或者 Ready 状态；等待指的是一个 Future 在执行器上因为运行到某个需要等待的事件时，进入 Pending 状态，并从执行器的就绪队列转移到等待队列上，等待事件的发生；唤醒指的是反应器接收到一个事件之后，唤醒等待在该事件上的 Future，将其重新放回到就绪队列中以便执行器在合适的时候执行该 Future。

当一个 Future 被创建，一般情况下会在很短的延时内立即在执行器中执行一次，直到需要等待某个事件发生，之后便进入等待队列；等到 Future 等待的事件到来，反应器通过该 Future 对应的一个 waker 的 wake 方法将该 Future 加入到就绪队列中，并由执行器开始进行轮询操作。这里提到的 waker 由一个 Future 当前执行的上

下文 data 以及一个 VTable 相关联。下面是 Waker 结构体的一种定义形式：

```
1.  struct Waker {
2.    waker: RawWaker
3.  }
4.  struct RawWaker {
5.    data: *const (),
6.    vtable: &'static RawWakerVTable
7.  }
8.  struct RawWakerVTable {
9.    clone: unsafe fn(*const ()) -> RawWaker,
10.   wake: unsafe fn(*const ()),
11.   wake_by_ref: unsafe fn(*const ()),
12.   drop: unsafe fn(*const ())
13. }
```

其中 data 指向的位置保存了该 waker 所对应的 Future 在执行过程中所需要的相关信息，vtable 则标注了一些方法的入口，这些方法会在 waker 的复制(clone)，唤醒 Future(wake、wake_by_ref)，以及 waker 的销毁(drop)时被调用。通过 waker，反应器便能在事件发生时及时的通知对应的 Future 重新进入轮询状态。

2.3 Embassy

本部分将对一个已有的 Rust 异步运行时库 Embassy 进行介绍，并分析其异步运行时的设计。

2.3.1 Embassy 简介

Embassy 是一个嵌入式的应用框架，并提供了一些在嵌入式开发板上能够直接使用的异步运行时，旨在简化嵌入式软件开发过程并提高代码的可重用性和可维护性。在 Embassy 库中提供了一组工具和库，包括但不限于 Embassy 自己设计的执行器，以及针对 stm32 板子开发的一些异步硬件驱动。

Embassy 使用事件驱动的编程范式，使得开发者能够轻松地将应用程序分解为独立的模块，每个模块针对特定类型的事件做出相应；内置了异步的任务调度器，可以管理任务的执行顺序和优先级，使得在嵌入式系统中处理并发任务变得更加容易和高效；提供了标准的驱动程序口，简化了硬件驱动程序的开发和继承过程。并且由于 Embassy 是基于 Rust 编程语言开发的，因此具有良好的跨平台和可移植性，可以轻松地在不同的硬件平台上部署和运行。

2.3.2 Embassy 异步运行时分析

Embassy 中的执行器是一个为嵌入式设计的 async/await 执行器，支持对各种外设的中断和定时器的中断进行响应。

Embassy 中的执行器具有以下几点特征：

- 无需动态分配，不需要堆，所有任务的内存都被静态分配；
- 使用非固定容量的数据结构，无需配置或调整，执行器就可以执行 1 到 1000 个任务；
- 集成的定时器队列，使用简单的方法就可以实现休眠；
- 无需一直保持轮询状态：没有任务时，配合中断和基于事件的唤醒和睡眠机制即可使 CPU 睡眠；
- 高效轮询：执行器每次只会轮询已经被唤醒的任务，而不是所有的任务；
- 公平：即使一个任务被不断地唤醒，也不会独占 CPU 资源；在一个任务的一次执行之后到下一次执行之前，所有的其他任务都会获得运行的机会。
- 支持创建多个执行器实例，并以不同的优先级运行任务。遇到优先级低的任务，较高优先级的任务可以抢占运行。

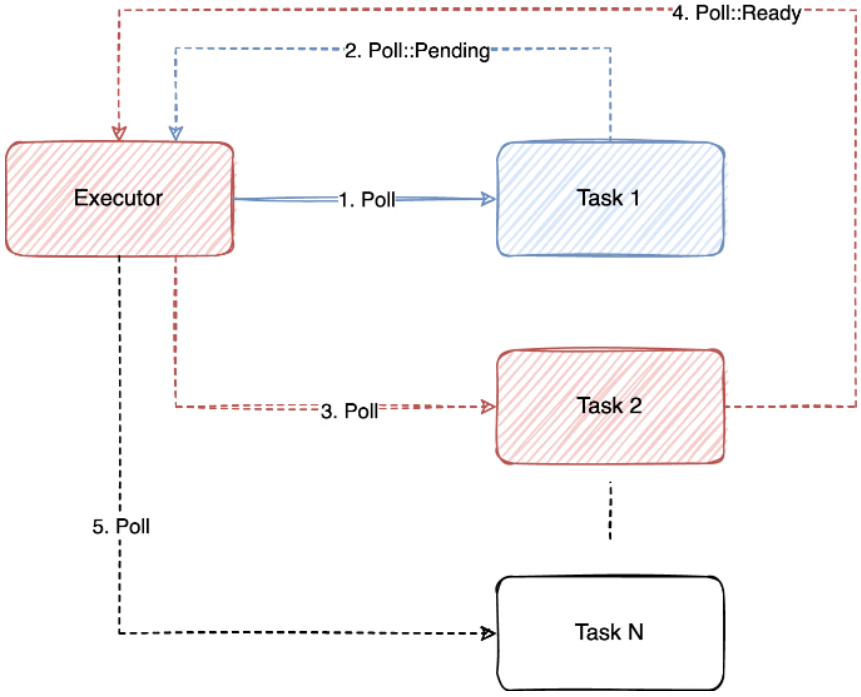

图 2-1 Embassy 执行器对任务的轮询设计

如图 2-1 所示，在执行器功能的具体设计上，Embassy 中的执行器会保存需要轮询的任务队列；（1）当一个任务被创建时，执行器对该任务执行一次 poll 方法；（2）

该任务将尝试运行，取得进展，直到被阻塞（有可能在等待某个异步事件的发生，即等待在某个 Future 上），然后向执行器返回 Poll::Pending；（3）一旦一个任务让渡 CPU，执行器就会将该任务加入到执行队列的末端，并且对执行队列中的下一个任务执行一次 poll 方法。如果一个任务被执行完成或者被取消掉，那么就不会再重新加入队列。

如果在应用程序中使用 `#[embassy_executor::main]` 宏，Embassy 将自动为我们创建一个执行器，并将该 main 入口点作为第一个任务；除此之外，我们还可以通过类似于下面的方式手动创建一个执行器，并将一个异步任务调度到执行器上轮询：

```
// 创建一个全局的 Embassy 默认的 Executor
static EXECUTOR: StaticCell<Executor> = StaticCell::new();

#[no_mangle]
// 内核入口函数
pub fn rust_main() -> ! {
    let executor = EXECUTOR.init(Executor::new());
    executor.run(|spawner| {
        // 把 spawner 作为参数放入 kernel 启动函数，为支持后面的多线程
        spawner.spawn(kernel_start(spawner)).unwrap();
    });
}

#[embassy_executor::task]
async fn kernel_start(spawner: Spawner) {
    // ...
}
```

结合具体的中断事件，一种典型的唤醒对应任务的机制如图 2-2 所示：

1）任务被轮询，并尝试取得进展；

2）任务指示外设执行某些操作，并 await 该操作的完成；

3）经过一段时间后，等待的操作完成，硬件发出中断，标识该操作已经完成；

4）然后相关的硬件抽象层会将相应的外部中断传达到相应的外设处理模块，使用操作的结果来更新外设处理模块的状态；

5）然后唤醒相应的任务，通知执行器可以继续轮询该任务。

此外，我们也可以创建多个中断处理的执行器实例，可以驱动不同优先级的任务。

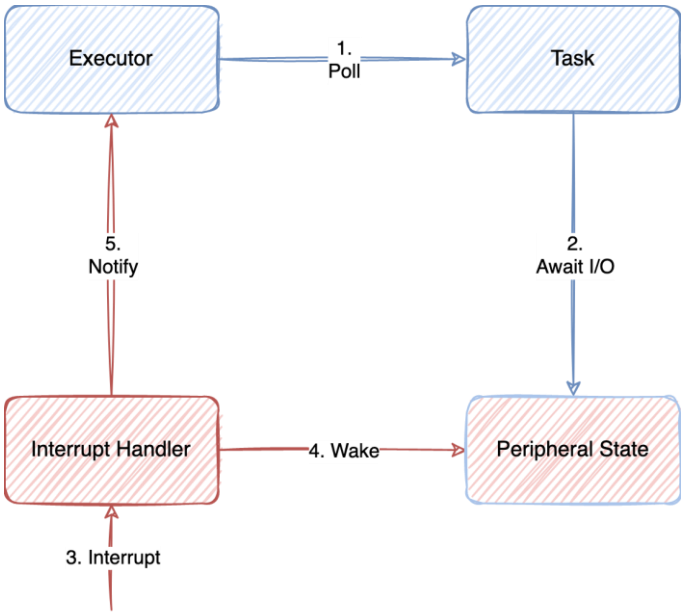

图 2-2 Embassy 处理中断的响应机制

2.4 Alien

对于很多开发者而言，复现前人所做的工作或许是需要耗费很大精力的事情。重复造轮子对自己的学习可能有很大意义，但是从行业整体的效率上看，如果众多的人都把心思放在重复造轮子上，而不能很快的把前人已有的工作利用起来，肯定会使得行业整体的开发效率有所下降。操作系统的开发亦是如此，如果所有希望开发操作系统的人都需要按照 rCore 之类的教程从头搭建一个内核，并且所有的部分都是自己开发（或者说参照别人的已有实现，进行一定的修改后放到自己的操作系统中），在这个过程中无疑是浪费了许多宝贵的时间，这些时间本可以用在更有意义的事情上去。这是操作系统模块化想法产生的重要原因之一。操作系统模块化能够使得对不同领域感兴趣、擅长不同领域的开发者能够专注在自己模块的优化和改进上，提升了操作系统开发的整体效率。

本部分将简要介绍在操作系统模块化方向上有所探索的操作系统 Alien。

2.4.1 Alien 简介

Alien 是一个基于 Rust 语言实现的类 Linux 的操作系统，并于 2023 年大学生计算机系统能力大赛内核实现赛道获得了全国赛一等奖。Alien 的目的旨在探索如何使用模块去构建一个操作系统，因此在 Alien 的实现中，会将操作系统内核分割成一个个模块，这些模块要么来自于 Rust 社区已有的比较好的现成 Crate，要么是自己

设计并实现、最终发布到相关社区中的 Crate。这些 Crate 也会被其他操作系统的开发者检索到，并且由于 Rust 在包管理方面的支持，这些 Crate 也能够很方便地被有需要的开发者使用。

在本研究中将以 Alien 作为使用异步串口驱动的上层操作系统，以此证明异步串口驱动的正确性。

第 3 章 异步串口驱动模块的设计与实现

3.1 串口基本机制

3.1.1 串口简介

串口，原名为串行接口（Serial Interface）,是一种用于连接计算机与外部设备得物理接口。在串口上，我们通常使用通用异步接收/发送器作为通信协议。通用异步接收/发送器（Universal Asynchronous Receiver-Transmiterr,简称 UART）是一种通信协议，负责在串口上实现数据的传输和接收，它定义了被传输数据的格式、速率以及停止位等。由于串口和 UART 紧密相关，一般也直接使用串口来代指 UART。

在本项研究中，我们使用的 Alien 操作系统运行在 QEMU 模拟机上，使用了 QEMU 提供的虚拟串口。QEMU 虚拟机模拟的串口兼容了 NS16550A 硬件规范。根据该规范，每个 UART 使用 8 个 I/O 字节来访问其寄存器。下表显示了 UART 中每个寄存器的地址和基本含义。表中使用的 base 表示串口设备的起始地址。在 QEMU 模拟的虚拟裸机 qemu-system-riscv64 中默认的串口设备寄存器的基址为 0x10000000。

表 3-1 UART 中每个寄存器的地址和基本含义

I/O port	Read(DLAB=0)	Write(DLAB=0)	Read(DLAB=1)	Write(DLAB=1)
Base	**RBR** reciver buffer	**THR** transmitter holding	**DLL** divisor latch LSB	**DLL** divisor latch LSB
Base+1	**IER** interrupt enable	**IER** interrupt enable	**DLM** divisor latch MSB	**DLM** divisor latch MSB
Base+2	**IIR** interrupt identification	**FCR** FIFO control	**IIR** interrupt identification	**FCR** FIFO control
Base+3	**LCR** line control	**LCR** line control	**LCR** line control	**LCR** line control
Base+4	**MCR** modem control	**MCR** modem control	**MCR** modem control	**MCR** modem control
Base+5	**LSR** line status	factory test	**LSR** line status	factory test
Base+6	**MSR** modem status	not used	**MSR** modem status	not used
Base+7	**SCR** scratch	**SCR** scratch	**SCR** scratch	**SCR** scratch

注：表中 LCR 寄存器 DLAB 位的具体设置会影响 CPU 访问的寄存器类型。例如，当 DLAB 位被设置成 0 时，读取位于 base 处的串口寄存器是 RBR 寄存器；当 DLAB 位被设置成 1 时，读取位于 base 处的串口寄存器对应的是 DLL 寄存器。

3.1.2 串口的 FIFO 模式

串口的 FIFO（First-In, First-Out）是一种缓冲区，用于临时存储串口传输的数据。它通常被实现为一个硬件缓冲区，位于 UART 控制器内部，有助于提高数据传输效率并减轻 CPU 负担。不同串口的 FIFO 可能有不同的深度，表示可以存储多少字节的数据。在 NS16550A 中，FIFO 深度被设置为 16 字节，这意味着它的 FIFO 缓冲区可以存储最多 16 个字节的数据。

由于接收器 FIFO 和发送器 FIFO 是分开控制的，因此数据的读写也会分别作用于对应的 FIFO，即读数据作用于接收器上的 FIFO，而写数据作用于发送器上的 FIFO。当有数据到达串口时，它首先被尝试写入发送器 FIFO 缓冲区。如果 FIFO 未满，则数据将成功写入；如果 FIFO 已满，则数据可能会被丢弃或者产生溢出错误，具体取决于 UART 的配置和处理方式。类似的，当有数据需要被读取时，串口控制器会尝试从接收器 FIFO 中读取数据并传送给 CPU。

在串口的硬件中使用 FIFO 能够有效地降低串口传输时的延时，并提高系统的响应速度。此外，FIFO 还具有减少 CPU 的轮询次数，降低系统负载，提高系统性能的优势。

3.1.3 串口的中断处理

串口的中断使能寄存器（IER，Interrupt Enable Register）用于单独启用或禁用 UART 能够产生的某种特定的中断请求。表 3-2 中说明了中断使能寄存器的相关字段：

表 3-2 UART 的中断使能寄存器（IER）的字段说明

Bit	字段	值类型	描述
31-8	Reserved	0	保留位
7	PTIME	R/W-0	可编程 THRE 中断模式使能，启用/禁用 THRE 中断的产生。0=禁用；1=启用
6-4	Reserved	0	保留位
3	EDSSI	R/W-0	使能调制解调器状态中断。0=禁用；1=启用
2	ELSI	R/W-0	使能接收器线状态中断。0=禁用；1=启用
1	ETBEI	R/W-0	使能发送器保持寄存器空中断。0=禁用；1=启用
0	ERBFI	R/W-0	使能接收器数据可用中断和字符超时指示中断。0=禁用；1=启用

在我们将中断使能寄存器中使能相应的串口中断后，每当有新的输入数据进入串口的接收缓存中，或者串口完成了缓存中数据的发送，又或者串口发送出现错误时，

串口都会产生一个中断，该中断的相关信息会记录在中断识别寄存器（IIR，Interrupt Identification Register）中。当上层操作系统接收到串口设备传来的中断信号后，会继续调用串口设备驱动中实现的中断处理函数，对串口设备的中断事件进行处理。表 3-3 说明了中断识别寄存器的相关字段：

表 3-3 UART 的中断识别寄存器（IIR）的字段说明

Bit	字段	值	描述
31-8	Reserved	0	保留位。
7-6	FEFLAG	R-0	FIFO 启用/禁用标志： 0b00 = 非 FIFO 模式； 0b11 = 使能 FIFO，FIFO 控制器中的 FIFOEN 为 1。
5-4	Reserved	0	保留位。
3-1	INTID	R/W-0	中断类型： 0b000 = 保留； 0b001 = 发送器保持寄存器为空（优先级 3）； 0b010 = 接收器数据可用（优先级 2）； 0b011 = 接收器线路状态（优先级 1，最高）； 0b100 = 保留； 0b101 = 保留； 0b110 = 字符超时指示（优先级 2）； 0b111 = 保留。
0	IPEND	R/W-0	中断挂起： 当任何 UART 中断产生并在 IER 中使能时，IPEND 被强制为 0。IPEND 保持为 0，直到所有挂起的中断被清除或直到发生硬件复位。如果没有启用中断，则 IPEND 不会被强制为 0。 0 = 中断挂起； 1 = 没有待处理的中断

3.2 驱动的整体架构设计

本研究开发的异步串口驱动的整体代码结构如图 3-1 所示。在 src 目录下的.rs 文件及文件的作用分别为：

- lib.rs：标识该 crate 是一个 lib 库，并声明各个子模块；
- serial.rs：定义同步串口驱动 BufferedSerial 和异步串口驱动 AsyncSerial 的数据结构，并为串口的异步读写任务 SerialReadFuture 和 SerialWriteFuture 实现 Future 特征；
- task.rs 和 waker.rs：定义与异步运行时相关的数据结构，包括对读写任务

Task、执行器 Executor、唤醒器 waker 等数据结构。

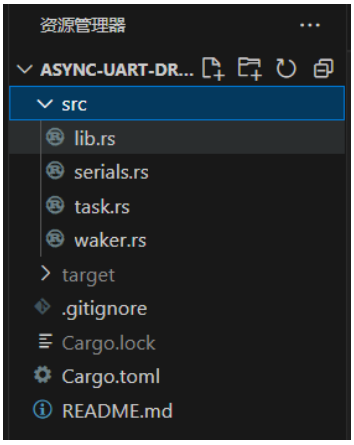

图 3-1 驱动的整体代码结构

其中最重要的数据结构异步串口 AsyncSerial 包含的字段及其说明如表 3-4 所示：

表 3-4 AsyncSerial 异步串口数据结构中的字段及其说明

字段	说明
base_address	串口的基址，并根据该地址以及偏移量可以求得某一具体寄存器的位置
rx_pro	rx 的生产者（从串口的接收 FIFO 到 rx）
rx_con	rx 的消费者（从 rx 到具体的读任务）
tx_pro	tx 的生产者（从具体的写任务到 tx）
tx_con	tx 的消费者（从 tx 到串口的传输 FIFO）
rx_count	在 rx 上进行读取操作的次数
tx_count	在 tx 上进行写入操作的次数
intr_count	接收到中断的总次数
rx_intr_count	接收到“接收器数据可用”和“字符超时”中断的次数
tx_intr_count	接收到“发送器保持寄存器为空”中断的次数
rx_fifo_count	在串口的接收 FIFO 上进行读取操作的次数
tx_fifo_count	在串口的传输 FIFO 上进行写入操作的次数
rx_intr_enable	串口发出“可读”中断的使能情况
tx_intr_enable	串口发出“可写”中断的使能情况
prev_cts	用于记录上一次串口修改 cts 时的修改结果
read_wakers	等待“可读”事件的所有任务的 waker 队列
Writ e_wakers	等待“可写”事件的所有任务的 waker 队列
executor	异步串口的执行器

3.3 异步运行时设计

本研究开发的异步串口驱动使用了自行设计和实现的异步运行时。为了使得该运行时能够达到更高的效率，我们的实现参考了 Embassy 中的异步运行时设计，采用了事件驱动的形式。

以下是有关 Task，TaskRef 以及 Executor 的相关数据结构的定义：

```
pub struct Task {
    /// detail value shown in 'TaskRef'
    pub(crate) state: AtomicU32,
    /// The task future
    pub fut: AtomicCell<Pin<Box<dyn Future<Output = i32> + 'static + Send + Sync>>>,
    /// driver
    pub driver: Arc<AsyncSerial>,
    /// IO Type
    pub iotype: AtomicU32,
}

pub struct TaskRef {
    ptr: NonNull<Task>,
}

pub struct Executor {
    tasks: Mutex<VecDeque<Arc<Task>>>,
}
```

本研究使用一个 Task 数据结构来抽象所有的串口读写任务。在该数据结构中，包含标识任务状态的 state 字段，标识实际需要被轮询操作的 fut 字段（该字段必须实现 Future 等相关特性），标识对串口驱动实例引用的 driver 字段，以及标识读写操作类型 iotype 字段。为了避免多余的 unsafe 操作，方便在 Task 和指向 Task 的裸指针之间进行切换，我们使用 TaskRef 数据结构对 Task 进行包装，并且通过 NonNull 这一数据结构，更安全地获取到一个 Task 相应的裸指针。

异步串口驱动中异步运行时的执行器，被实现为目前需要被调度以及未来需要被调度（目前处于阻塞状态）的所有任务组成的一个队列，考虑到多个线程有可能会对执行器进行互斥使用，在队列上加一把锁也是有必要的。

在图 3-2 中说明了在异步串口驱动中的异步运行时以及处理相应中断的流程。

1）由操作系统调用异步驱动模块的异步读写任务，创建一个读写任务（Task）

后，该函数会直接返回，以便操作系统执行其他的操作。

2) 新创建的读写任务将交于异步串口驱动中的执行器，并将该任务对应的唤醒器（waker）注册在相应的读写队列中；任务创建并进入运行队列之后，由于任务目前的状态为就绪状态，Executor 会首先对其进行一次轮询，直到发现该任务读取的字符数量还没有达到要求，或者任务希望写入但写入失败，此时任务的状态会被修改为阻塞，并且在下一次任务的状态被设置为就绪之前，执行器都不会对其进行轮询。

3) 等到串口中传来接收器数据可用或者字符超时指示中断（可读）以及发送器保持寄存器空中断（可写）时，相关的中断信息会被引导到中断服务例程。

4) 然后通过中断服务例程，将等待在可读事件或可写事件上的任务全部唤醒。具体的，进入相应的 waker 队列对 waker 进行唤醒操作。

5) 在中断服务例程中调用一次 run_until_idle 方法，通知执行器可以继续轮询。

6) 执行器对所有的就绪任务进行一次轮询。

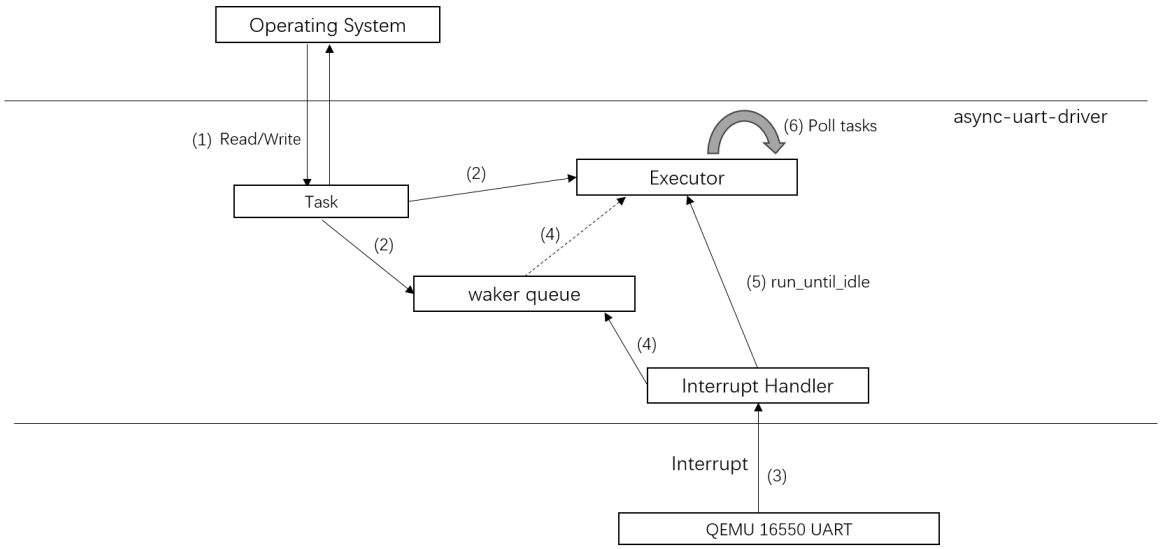

图 3-2 异步串口驱动处理中断事件的流程

3.4 接口设计

本部分主要介绍异步串口驱动模块向上层操作系统提供的相关接口，包括异步串口的初始化以及读写。

3.4.1 异步串口的初始化

当操作系统需要创建一个异步串口驱动时，需要调用 AsyncSerial::new 方法创建一个异步串口 AsyncSerial 实例。一种可能的初始化方式如下：

```
1. type RxBuffer = Queue<u8, DEFAULT_RX_BUFFER_SIZE>;
2. type TxBuffer = Queue<u8, DEFAULT_TX_BUFFER_SIZE>;
3. static mut DRIVER_RX_BUFFER: RxBuffer = RxBuffer::new();
4. static mut DRIVER_TX_BUFFER: TxBuffer = TxBuffer::new();
5. let (rx_pro, rx_con) = unsafe { DRIVER_RX_BUFFER.split() };
6. let (tx_pro, tx_con) = unsafe { DRIVER_TX_BUFFER.split() };
7. let async_serial = AsyncSerial::new(
8.     base_addr,
9.     rx_pro,
10.    rx_con,
11.    tx_pro,
12.    tx_con
13.);
14.let async_serial = Arc::new(async_serial);
15.println!("before hardware_init");
16.async_serial.hardware_init(BAUD_RATE);
```

希望使用异步串口驱动的上层操作系统需要根据自己需要的缓冲区大小，创建一个读缓冲 RX 区和写缓冲 TX 区，并将串口的基址 base_addr 以及这两个缓冲区的生产者和消费者依次提交给异步串口驱动的初始化函数。但传入相关信息后，初始化函数只是单纯对 AsyncSerial 数据结构的相关字段进行初始化，并没有对串口硬件设备进行任何的操作。我们还需要调用 hardware_init 方法（16 行），并将规定的串口传输波特率一并提供，这样 AsyncSerial 异步串口驱动就会对相应的串口硬件上的相关寄存器进行赋值，并将相应的中断使能，之后便能进行正常的读写操作。

3.4.2 异步串口的读写

创建完异步串口驱动后，上层操作系统便能直接调用 AsyncSerial 的读写函数，创建读写任务，并由异步串口驱动内部的运行时对这些读写任务进行处理。

当操作系统调用 AsyncSerial 的 read 方法后，AsyncSerial 会自动创建一个串口读任务，这个任务记录了读取的内容要被记录的位置，需要读取的字符长度，以及异步串口驱动实例的一个引用。之后，这个任务会被封装成一个 Task 结构，以便按照驱动内部设计的异步运行时获取其对应的 waker，并将其放入调度任务的执行器中，具体的实现如下所示：

```rust
pub async fn read(self: Arc<Self>, buf: &'static mut [u8]) {
    let future = SerialReadFuture {
        buf,
        read_len: 0,
        driver: self.clone(),
    };
    // 注册
    let task = Task::new(
        Box::pin(future),
        self.clone(),
        crate::task::TaskIOType::Read
    );
    self.register_readwaker(
        unsafe { from_task(task.clone())}
    );
    self.executor.push_task(Task::from_ref(task));
}
```

AsyncSerial 的 write 方法也大致相同，具体实现如下：

```rust
pub async fn write(self: Arc<Self>, buf: &'static [u8]) {
    let future = SerialWriteFuture {
        buf,
        write_len: 0,
        driver: self.clone(),
    };
    let task = Task::new(Box::pin(future), self.clone(), crate::task::TaskIOType::Write);
        self.register_writewaker(
        unsafe { from_task(task.clone()) }
    );
    self.executor.push_task(Task::from_ref(task))
}
```

第 4 章 异步串口驱动测试

本研究使用 Alien 作为上层的操作系统，在虚拟环境下对设计和实现的异步串口驱动进行测试。由于 QEMU 虚拟机并未支持 RISC-V 平台的多串口收发，本研究借鉴了前人的相关研究，修改了 QEMU 的源码使其支持多串口。之后成功在 Alien 上使用异步串口驱动，并进行了简单的测试。

4.1 修改 QEMU 源码使其支持多串口

由于 Alien 的中断已经占用了一个串口用于文字输入输出，为了不影响操作系统的运行状态并对异步串口的正确性进行验证关于如何给虚拟机添加多个虚拟串口的相关资料，大多都只提到在 qemu 模拟机启动时添加一个启动参数即可。但经过实际测试后发现添加的多个串口并没有起作用，在启动的 Alien 中输出相应的设备树也会发现确实只有一个串口设备被创建。

我们依照附录 A 的内容修改了 QEMU 源码，并对修改后的 QEMU 进行编译，得到了测试的基本虚拟环境。

注：在陈志扬学长给出的为 QEMU 添加多个串口的相关教程中，在对 QEMU 源码进行修改并编译以后，操作系统可以向串口发送消息，但并不能接收到基于中断机制的消息，因此需要对 RustSBI 中进行一定的修改。在本研究中使用默认未经修改的 OpenSBI 未发现以上异常。

4.2 Alien 使用异步串口驱动

4.2.1 配置 QEMU 路径及启动参数

在 Alien 根目录的 Makefile 文件下，定义了执行 make 相关命令时将测试文件烧写到 sdcard 中、安装相关运行环境、编译 Alien 源文件、启动 QEMU 虚拟机并将编译好的 OS 和测试文件的 bin 文件装载到虚拟机上等相关步骤的执行顺序和具体参数配置。其中，由于我们修改了 QEMU 使其支持多个串口，我们需要将默认使用的 QEMU 修改为我们修改后的 QEMU 即指出我们修改后的 QEMU 所在的路径：../os/myqemu/qemu-build/riscv64-softmmu/qemu-system-riscv64。此外，我们还需要为所有的共 5 个串口（原本 1 个加上新创建的 4 个串口）的输入输出重定向到某个终端控制台。

依照下面的代码所示，我们将 QEMU 的路径配置成修改后的 qemu-system-riscv64 所在的文件路径，并将创建的 5 个串口分别绑定在 5 个伪终端（pty）上。其中/dev/pts/4 在本测试中用于打印操作系统执行的相关信息以及执行过程中的日志文件，/dev/pts/5 所在的终端用于启动 QEMU，/dev/pts/7 为进行异步串口测试的终端。

```
1. define boot_qemu
2.  $(QEMU) \
3.          -M virt $(1)\
4.          -bios $(BOOTLOADER) \
5.          -drive file=$(IMG),if=none,format=raw,id=x0 \
6.          -device virtio-blk-device,drive=x0 \
7.          -kernel  kernel-qemu\
8.          -$(QEMU_ARGS) \
9.          -smp $(SMP) -m $(MEMORY_SIZE) \
10.  $(SERIAL_CONFIG)
11.endef
12.
13.QEMU := ../os/myqemu/qemu-build/riscv64-softmmu/qemu-system-
   riscv64
14.SERIAL_CONFIG := -serial /dev/pts/4 -serial /dev/pts/5 -
   serial /dev/pts/7 -serial /dev/pts/10 -serial /dev/pts/14
```

4.2.2 双核启动 Alien

由于异步串口驱动的接口和同步串口驱动的接口并不相同，我们无法将简单地将 Alien 中原有的同步串口直接修改为异步串口，因为这样会使得 Alien 与终端交互的逻辑出现问题。我们想到使用双核的方式，将 Alien 原本的启动流程和异步串口驱动的测试分离开。在上面的代码中，我们修改 SMP 的值为 2，这会使得我们在使用 qemu-system-riscv64 时所模拟的逻辑环境具有两个硬件核。这样，我们就可以在一个核上继续运行原来 Alien 中的相关启动流程，在另一个核上执行我们的测试。在 Alien 管理配置信息子模块中的相关脚本（/subsystems/config/build.rs）将会自动读取该 SMP 值，并在 Alien 的源文件中使用正确的核数量，从而编译出正确的操作系统可执行文件。

在正式启动 Alien 后，会根据核的启动顺序分配相应的任务：先启动的核执行 Alien 中原本的启动流程，包括初始化虚拟内存系统、中断、硬件设备、文件系统、trap 入口地址等，最终会初始化一个初始进程，执行测试文件并生成一个 She

11，供用户与 Alien 交互；后启动的核的启动过程一定在之前的核完成启动工作之后，并且在前面的核进行相关的启动工作之后，后启动的核只会简单的进行一些初始化工作，包括激活页表、允许内核访问用户内存、初始化 trap 入口地址，随后便进入异步串口驱动的测试流程。

4.2.3 Alien 进入异步运行时

在 Rust 中，异步函数调用同步函数相对简单，但同步函数调用异步函数需要借助将同步函数阻塞在异步函数上的 block_on 方法，或者使用 Embassy 的运行时，启动一个 executor 线程，在 executor 线程上，将可以调用异步函数。因此本测试中，在 Alien 双核启动完成后，执行异步串口驱动测试任务的核将借助 Embassy 的执行器，完成从同步到异步的转换。

具体代码如下：

```
fn kernel_init(hart_id: usize){
    // ...
    // 第二个核初始化结束后
    let executor = EXECUTOR.init(Executor::new());
        executor.run(|spawner| {
            spawner.spawn(async_test(spawner)).unwrap();
        });
    //
}

#[embassy_executor::task]
async fn async_test(_spawner: Spawner) {
    test().await;
    uart_driver_init().await;
    loop { }
}
```

创建的异步线程会首先执行一小段简单的异步程序，用于检验异步函数调用是否成功。之后，会正式进入异步串口测试流程。

4.2.4 为异步串口支持 DeviceBase 特征

在 Alien 中设备接口子模块中，定义了串口设备、块设备等外设的基本特征。为了在 Alien 中能够使用我们编写的异步串口驱动模块，我们将 AsyncSerial 结构封装成了 AsyncSerialBevice 结构，并为 AsyncSerialBevice 结构实现了 DeviceBase 特征。只有实现了 DeviceBase 特征的设备，才能调用 register_device_to_pli

c 方法完成该设备的中断注册。具体的，实现 DeviceBase 特征需要我们提供该设备的中断处理函数，并实现 Sync 和 Send 特征，但由于我们的异步串口驱动 AsyncSerial 本身提供了中断处理函数，因此整个步骤非常简单，如下面的代码所示：

```
pub struct AsyncSerialDevice {
    inner: Arc<AsyncSerial>,
}

unsafe impl Send for AsyncSerialDevice{ }
unsafe impl Sync for AsyncSerialDevice{ }

impl DeviceBase for AsyncSerialDevice {
    fn hand_irq(&self) {
        self.inner.interrupt_handler();
    }
}
```

4.3 异步串口驱动测试

4.3.1 异步串口驱动的初始化

在异步串口驱动的初始化阶段，有两种方式启动异步串口驱动，其一是 Alien 获取 QEMU 中的设备树信息，在设备树中找到以 Serial 开头的一个设备（经过测试，这个设备总是对应于新创建的四个串口中的最后一个串口，即起始地址为 0x10005000 的串口设备）；除此之外，我们也可以串口的 MMIO 地址和中断号硬编码在操作系统中，并利用硬编码的 MMIO 地址和中断号对异步串口进行初始化。

在本文的测试阶段，我们将使用第二种方法初始化异步串口驱动设备，固定测试的串口设备为 MMIO 在 0x10003000 的串口，并参照与 3.4.1 节中异步串口的初始化相类似方法。初始化异步串口后，我们将其赋值给一个全局的静态变量以便后续的异步串口读写操作，并将该串口的中断注册到当前运行的核上。

4.3.2 异步串口读写

Alien 启动并初始化异步串口驱动结束后，将创建两个缓冲区分别为写缓冲区和读缓冲区。写缓冲区被初始化为类型为 u8，长度为 12 的数组，里面的内容为从 A 到 L 的字符对应的 ASCII 码；读缓冲区被初始化为一个类型为 u8，长度为 12 的可变数组，里面所有的字符的值被初始化为 0。之后分别创建一个异步读任务和异步写任务，分别从串口中将数据读出写入读缓冲区，并将写缓冲区中的数据写入串

口。但由于创建异步读写任务之后，该函数立刻返回，即读写任务并非立即执行，还需要接收到串口传来的对应中断。

因此，我们需要在异步串口重定向到的终端中键入一个字符数据，此时会触发串口的接收数据可用（Received data available）中断。相应的，在异步串口驱动的中断服务例程中，会将等待串口可读事件的读任务唤醒，并使得执行器进行一次轮询，执行该读任务。

执行完读任务的一次读取后，串口会自动触发发送器保持寄存器空（Transmitter Holding Register Empty）中断，该中断指示当前串口可以写入数据。相应的，在异步串口驱动的中断服务例程中，会将等待串口可写事件的写任务唤醒，并使得执行器进行一次轮询，执行写任务。

4.3 测试结果分析

在创建读写任务之后，我们会发现 Alien 在输出启动的相关信息后分别创建异步读写任务后，输出了当前的读缓冲区状态为全零，即当前异步读任务还没有执行，如图 4-1 所示；同时我们可以发现 MMIO 地址在 0x10003000 的测试串口对应终端（以下简称测试终端）并没有接收到相应数据，即当前的异步写任务也没有执行，如图 4-2 所示。

```
[0] init sync task success
[1] ++++ setup interrupt ++++
[0] Begin run task...
[1] ++++ setup interrupt done, enable:true ++++
[0] kthread_init start...
[1] hart 1 start
[0] [INFO] user_path_at fd: -100,path:/tests/init
[1] ========= async test f3 ==============
[1] ========= async test f2 ==============
[1] ========= async test f1 ==============
[1] driver_init
[1] [devices/src] init async_uart
[1] Init async serial, base_addr:0x10003000,irq:13
[1] before hardware_init
[1] PLIC enable irq 13 for hart 1, priority 1
[1] external interrupt hart_id: 1, irq: 13
[1] [DEBUG] [Async Serial] Interrupt!
[1] [DEBUG] [SERIAL 10003000] Transmitter Holding Register Empty
[1] external interrupt hart_id: 1, irq: 13
[1] [DEBUG] [Async Serial] Interrupt!
[1] [DEBUG] [SERIAL 10003000] Transmitter Holding Register Empty
[1] async write task created
[1] async read task created
[1] current read buf: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

图 4-1 创建异步读写任务前后操作系统输出的相关信息

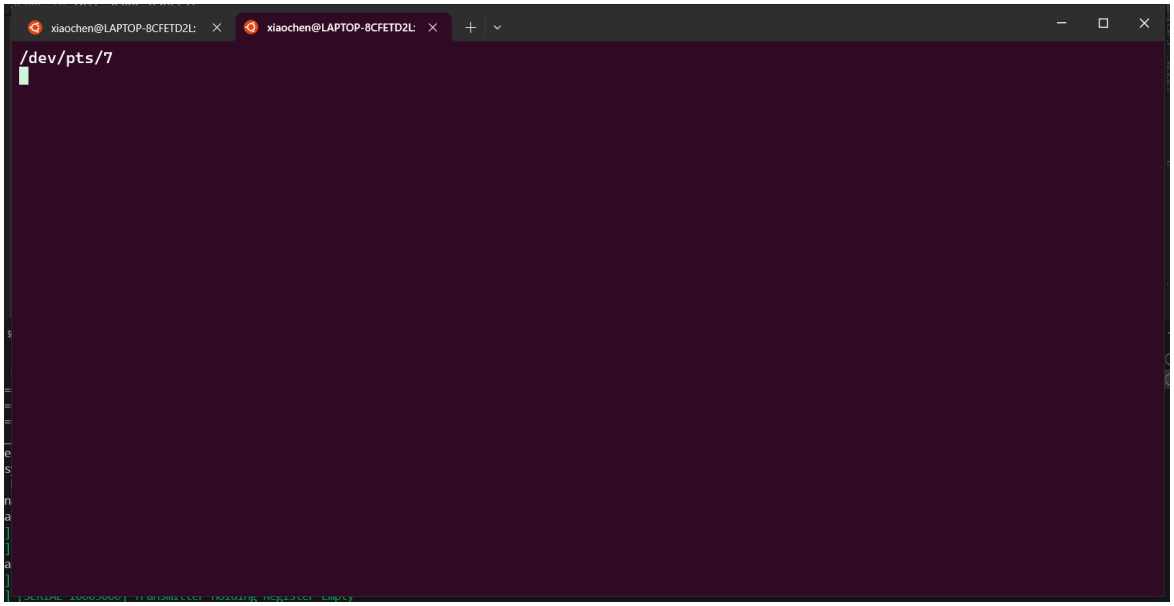

图 4-2 创建异步读写任务后测试终端的输出情况

如图 4-3 所示，在测试终端中键入“a”后，我们能够发现在接收到 Received data available 的串口中断后，异步串口驱动先唤醒了相应的异步读任务，将“a”（ASCII 码转换为十进制为 97）读入到异步读任务的缓冲区中，此时缓冲区变为[97, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]。在完成本次读操作即接收完串口中的数据后，串口又发出 Transmitter Holding Register Empty 的中断，报告当前可以向串口中写入数据，于是 TX 缓冲区向串口 FIFO 中写入相应的字符数组。该字符数组被测试终端检测到，于是输出了相应的字符数组，如图 4-4 所示。

```
[0] [INFO] pselect6: sigmask = 1018992 ---> [SIGTRAP, SIGABRT, SIGBUS, SIGSEGV, SIGUSR2, SIGSTKFLT, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP],
[1] external interrupt hart_id: 1, irq: 13
[1] [DEBUG] [Async Serial] Interrupt!
[1] [DEBUG] [SERIAL 10003000] Received data available
[1] [DEBUG] wake read task
[1] [DEBUG] tasks' len is 2
[1] [DEBUG] now 0th task
[1] [DEBUG] tasks' len is 1
[1] [DEBUG] now 0th task
[1] [DEBUG] read task receive '[97, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]'
[1] [DEBUG] read task
[1] [DEBUG] tasks' len is 1
[1] [DEBUG] now 0th task
[1] [DEBUG] [SERIAL 10003000] Transmitter Holding Register Empty
[1] [DEBUG] tx_con => serial 65
[1] [DEBUG] tx_con => serial 66
[1] [DEBUG] tx_con => serial 67
[1] [DEBUG] tx_con => serial 68
[1] [DEBUG] tx_con => serial 69
[1] [DEBUG] tx_con => serial 70
[1] [DEBUG] tx_con => serial 71
[1] [DEBUG] tx_con => serial 72
[1] [DEBUG] tx_con => serial 73
[1] [DEBUG] tx_con => serial 74
[1] [DEBUG] tx_con => serial 75
[1] [DEBUG] tx_con => serial 76
```

图 4-3 执行异步读写任务后操作系统输出的信息

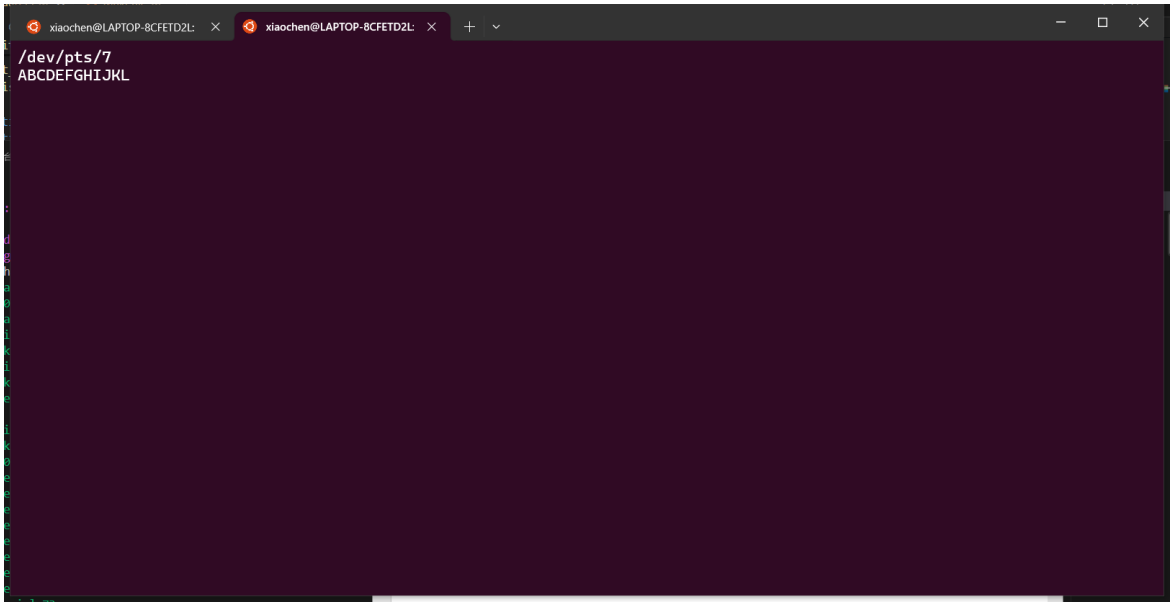

图 4-4 执行异步读写任务后测试终端输出的信息

之后再依次向测试终端中键入 A、b、C、D、e、f、g、H、I、J 字符，能够看到操作系统接收到相应的字符数组，最终异步读缓冲任务所接收到的字符数组为[97, 65, 98, 67, 68, 101, 102, 103, 72, 105, 106, 107]，如图 4-5 所示。

参照以上分析，我们能够验证异步串口驱动的正确性。

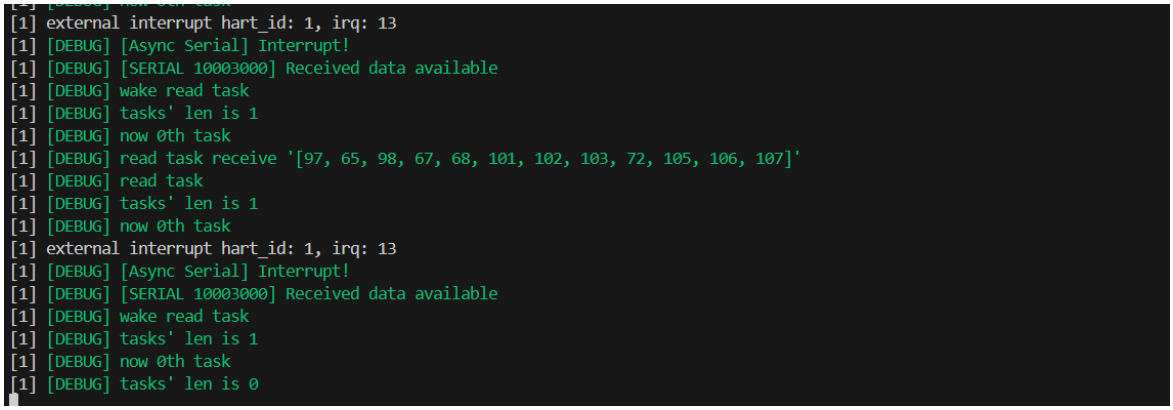

图 4-5 完成读取任务后异步读任务的缓冲区情况

第 5 章 总结和展望

5.1 论文总结

在本研究中，我们调研了 Rust 语言针对异步和模块化的相关支持，学习总结了已有串口驱动的实现，并研究了 Embassy 的异步运行时，最终我们针对 QEMU 虚拟机中的串口设备，使用 Rust 语言开发了一个异步串口驱动模块，并在其上使用模块化操作系统 Alien 调用该异步串口驱动模块。在该异步串口驱动模块中，我们设计了类似 Embassy 的异步运行时，将读写任务抽象为 Task 结构体，创建读写任务时将对应的读写任务加入到异步串口驱动的执行器中，等到接收到串口硬件中断后，会唤醒相应的读写任务并让执行器完成一次轮询。对于上层的操作系统 Alien 而言，向串口发起异步读写请求，读写任务创建后就会立刻返回让 Alien 执行接下来的操作，并且实际的读写操作将交于异步串口驱动模块执行。

理论上，使用我们的异步串口驱动模块，不仅能够提升开发者进行操作系统开发的效率，还能够提升整个操作系统的并发性和效率。采用中断事件触发执行的方式，避免了 CPU 忙等占用大量的 CPU 资源，使得系统在接收到读写请求后依然可以同时执行其他任务；采用 Rust Future 的协程机制，相对于线程等并发实现需要更小的内存开销，完成了对内存资源的节约；同时我们使用 Rust 语言实现异步串口驱动模块，也使得其他开发者能够更方便地使用我们的异步串口驱动模块。

但本论文还存在一些不足有待改进：

（1）没有在星光二实体板上进行适配。虽然当前 Alien 操作系统已经能够在板子上成功运行，但是实际的开发板和虚拟机情况存在一些不同：在 QEMU 模拟的虚拟 qemu-system-riscv64 中，我们不仅可以将几个虚拟串口的输入输出重定向到几个终端上，这样就可以对串口输出进行比较简单的可视化，并且使用该终端向串口中输入数据，并且由于具有多个串口，我们可以在某一个串口上使用原本 Alien 中的串口驱动实现，在另一个串口上使用我们实现的异步串口驱动；而在星光二 Vision Five2 开发板上启动 Alien 时使用的与 Ubuntu 进行通信的串口为默认串口，并且开发板上默认只开启一个串口。如果想要开启多串口，就需要修改硬件配置，同时使用更多的串口线连接开发板。

（2）本研究的测试和实验阶段，还只是停留在对异步串口驱动的正确性验证上，没有将异步串口驱动与传统的同步串口驱动进行性能上比较，也没有在大 IO 场景下去测试异步串口驱动的性能。这主要是考虑到 Alien 并不是纯异步的操作系统，在 Alien 上无法获取比较准确的异步性能。在目前的异步串口驱动中设计的运行时还仅仅是在异步串口驱动中使用，没有拓展到整个操作系统，以至于具体异步串口驱动的上限能够达到多少还无法测出；理论上讲，相较于 CPU 忙等型的同步串口驱动，本研究中开发的异步串口驱动的效率会有很大的提高；而和同样使用中断但采用线程实现的异步串口驱动相比，性能上则会相差不大，此时利用协程实现的异步串口驱动则会在节省内存资源的消耗上具有更大的优势。

5.2 未来的相关工作

由于目前存在的上述问题，未来的发展和工作可以分为以下几个方面：首先，将异步串口驱动移植到实体开发板上，实体开发板环境才是面向真实的生产生活环境，在其上进行的开发工作才更有实际意义。其次，还要在使用纯异步的操作系统中测试我们的异步串口驱动；如果有可能的话，还需要将异步串口驱动中的异步运行时也独立成一个单独的模块，供整体的操作系统使用，这样也便于操作系统在最上层对整个系统中所有的协程进行调度。最后，串口只是一个非常简单的硬件设备，使用 Rust 语言为其开发异步串口驱动也仅仅是众多工作中的一小步，未来还需要对更复杂的硬件设备如网络、音视频、块设备等开发异步驱动模块，更全面的提升整体操作系统的效率。

结　论

在本研究中，我们调研了 Rust 语言针对异步和模块化的相关支持，学习总结了已有串口驱动的实现，研究了 Embassy 的异步运行时，分析了 Rust Future 相较于其他并发编程方式的独特优势，得出使用 Rust 编写异步硬件驱动的可能性和意义。

最终，本研究针对 QEMU 虚拟机中的串口设备，使用 Rust 语言开发了一个异步串口驱动模块，并在其上使用模块化操作系统 Alien 调用该异步串口驱动模块，能够在 Alien 中向串口发起异步读写请求。

在理论上，使用我们的异步串口驱动模块，不仅能够提升开发者进行操作系统开发的效率，还能够提升整个操作系统的并发性和资源的利用效率。相较于 CPU 忙等型的同步串口驱动，本研究中开发的异步串口驱动的效率会有很大的提高，尤其是在 IO 量比较大的使用场景中；而和同样使用中断但采用线程实现的异步串口驱动相比，性能上则会相差不大，此时我们开发的利用协程实现的异步串口驱动会在内存资源的节约上具有更大的优势。

在未来可能的工作方面，将异步串口驱动移植到更具有实际意义的实体开发板上是必要的；同时，在纯异步的操作系统使用统一的异步运行时调度所有协程的情况下能够更真实地测试异步串口驱动的性能；除此之外，开发不限于串口的异步硬件驱动模块，也能使操作系统的开发变得更加便利，同时使得实际操作系统效率得到更好的提升。

参考文献

[1] 孙卫真,刘雪松,朱威浦,等. 基于 RISC-V 的计算机系统综合实验设计[J]. 计算机工程与设计,2021,42(4):1159-1165. DOI:10.16208/j.issn1000-7024.2021.04.037.

[2]卫一芃,杨晓宁.嵌入式实时操作系统异步 I/O 技术的研究[J].信息通信,2017(01):141-142.

[3]方兴,秦琦,刘维国.多线程异步 I/O 模型[J].舰船电子对抗,2005(04):61-64.DOI:10.16426/j.cnki.jcdzdk.2005.04.014.

[4]段楠.异步非阻塞网络通讯技术研究[J].现代计算机,2019(17):79-82.

[5] Harris T .Special Topic: AC – Composable Asynchronous IO For Native Languages[C]//Conference on Object-Oriented Programming Systems, Languages, and Applications.ACM, 2011.DOI:10.1145/2048066.2048134.

[6]沙泉.异步事件驱动模型在嵌入式系统中的应用[J].微计算机信息,2007(29):33-34+73.

[7] Zhu L , Huang L , Fu P ,et al.The upgrade to the EAST poloidal field power supply monitoring system[J].Fusion Engineering and Design, 2021, 172(10):112757.DOI:10.1016/j.fusengdes.2021.112757.

[8] Kwon G , Lee W , Lee T ,et al.Development of a real-time data archive system for a KSTAR real-time network[J].Fusion Engineering and Design, 2018, 127(feb.):202-206.DOI:10.1016/j.fusengdes.2018.01.019.

附　录 A 修改 QEMU 源码使其支持多串口

首先，从 github 上 clone QEMU 的源代码。

然后，参照下面图 1，图 2，图 3 的修改，我们分别在相应的头文件和.c 文件中为新串口添加 MMIO 地址和 IRQ 中断号。

include/hw/riscv/virt.h CHANGED

```
@@ -68,6 +68,10 @@
     VIRT_APLIC_M,
     VIRT_APLIC_S,
     VIRT_UART0,
+    VIRT_UART1,
+    VIRT_UART2,
+    VIRT_UART3,
+    VIRT_UART4,
     VIRT_VIRTIO,
     VIRT_FW_CFG,
     VIRT_IMSIC_M,
@@ -82,6 +86,10 @@ enum {
     UART0_IRQ = 10,
     RTC_IRQ = 11,
+    UART1_IRQ = 12,
+    UART2_IRQ = 13,
+    UART3_IRQ = 14,
+    UART4_IRQ = 15,
     VIRTIO_IRQ = 1, /* 1 to 8 */
     VIRTIO_COUNT = 8,
     PCIE_IRQ = 0x20, /* 32 to 35 */
```

图 1 为新添加的串口分配 IQR 号

hw/riscv/virt.c CHANGED

```
    [VIRT_ACLINT_SSWI] = {  0x2F00000,        0x4000 },
    [VIRT_UINTC] =       {  0x2F10000,        0x4000 },
    [VIRT_PCIE_PIO] =    {  0x3000000,       0x10000 },
+   // 3 PLIC context per hart: M, S, U
+   [VIRT_PLIC] =        {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 3) },
-   [VIRT_PLIC] =        {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_APLIC_M] =     {  0xc000000, APLIC_SIZE(VIRT_CPUS_MAX) },
    [VIRT_APLIC_S] =     {  0xd000000, APLIC_SIZE(VIRT_CPUS_MAX) },
    [VIRT_UART0] =       { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =      { 0x10001000,        0x1000 },
+   [VIRT_UART1] =       { 0x10002000,         0x100 },
+   [VIRT_UART2] =       { 0x10003000,         0x100 },
+   [VIRT_UART3] =       { 0x10004000,         0x100 },
+   [VIRT_UART4] =       { 0x10005000,         0x100 },
    [VIRT_FW_CFG] =      { 0x10100000,          0x18 },
    [VIRT_FLASH] =       { 0x20000000,     0x4000000 },
    [VIRT_IMSIC_M] =     { 0x24000000, VIRT_IMSIC_MAX_SIZE },
```

图 2 为新添加的串口分配 MMIO 地址

hw/riscv/virt.c CHANGED

```
    name = g_strdup_printf("/soc/uart@%lx", (long)memmap[VIRT_UART0].base);
    qemu_fdt_add_subnode(mc->fdt, name);
    qemu_fdt_setprop_string(mc->fdt, name, "compatible", "ns16550a");
@@ -1414,6 +1437,12 @@
         0, qdev_get_gpio_in(DEVICE(mmio_irqchip), UART0_IRQ), 399193,
         serial_hd(0), DEVICE_LITTLE_ENDIAN);

+    for (int uart_i = 1; uart_i <= 4; ++uart_i) {
+        serial_mm_init(system_memory, memmap[VIRT_UART0 + uart_i].base,
+            0, qdev_get_gpio_in(DEVICE(mmio_irqchip), UART1_IRQ + uart_i - 1), 399193,
+            serial_hd(uart_i), DEVICE_LITTLE_ENDIAN);
+    }
+
     sysbus_create_simple("goldfish_rtc", memmap[VIRT_RTC].base,
         qdev_get_gpio_in(DEVICE(mmio_irqchip), RTC_IRQ));
```

图 3 初始化串口

并且我们需要使用图 4 的方式修改设备树的初始化函数

hw/riscv/virt.c CHANGED

```
+    // add uart 1-4
+    for (int uart_i = VIRT_UART1; uart_i <= VIRT_UART4; ++uart_i) {
+        name = g_strdup_printf("/soc/serial@%lx", (long)memmap[uart_i].base);
+        qemu_fdt_add_subnode(mc->fdt, name);
+        qemu_fdt_setprop_string(mc->fdt, name, "compatible", "ns16550a");
+        qemu_fdt_setprop_cells(mc->fdt, name, "reg",
+            0x0, memmap[uart_i].base,
+            0x0, memmap[uart_i].size);
+        // 100M clock
+        qemu_fdt_setprop_cell(mc->fdt, name, "clock-frequency", 3686400);
+        qemu_fdt_setprop_cell(mc->fdt, name, "interrupt-parent", irq_mmio_phandle);
+        if (s->aia_type == VIRT_AIA_TYPE_NONE) {
+            qemu_fdt_setprop_cell(mc->fdt, name, "interrupts", UART1_IRQ + uart_i - VIRT_UART1);
+        } else {
+            qemu_fdt_setprop_cells(mc->fdt, name, "interrupts", UART1_IRQ + uart_i - VIRT_UART1, 0x4);
+        }
+    }
```

图 4 修改设备树初始化函数

最后对修改后的 QEMU 进行编译。

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】