

本科生毕业设计(论文)

**跨操作系统的异步串口和网络驱动模块设计与实现**

Design and Implementation of Asynchronous Serial Port and Network 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 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和网络驱动模块设计与实现

摘　要

关键词：

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

Abstract

Key Words:

目　录

[摘　要 I](#_Toc165900560)

[Abstract II](#_Toc165900561)

[第1章 绪论 1](#_Toc165900562)

[1.1 研究背景 1](#_Toc165900563)

[1.2 国内外研究现状 2](#_Toc165900564)

[1.3 本文研究内容与贡献 2](#_Toc165900565)

[1.3.1 研究内容和关键问题 2](#_Toc165900566)

[1.3.2 本文贡献 3](#_Toc165900567)

[第2章 相关技术介绍 4](#_Toc165900568)

[2.1 Rust Cargo 和 Crates.io 4](#_Toc165900569)

[2.1.1 Rust Cargo 4](#_Toc165900570)

[2.1.2 Crates.io 4](#_Toc165900571)

[2.2 Rust Future 5](#_Toc165900572)

[2.2.1常见的并发编程方式 5](#_Toc165900573)

[2.2.2 Rust Future 6](#_Toc165900574)

[2.2.3 Rust运行时 8](#_Toc165900575)

[2.3 Embassy 9](#_Toc165900576)

[2.3.1 Embassy 简介 9](#_Toc165900577)

[2.3.2 Embassy 异步运行时分析 10](#_Toc165900578)

[2.4 Alien 12](#_Toc165900579)

[2.4.1 Alien简介 12](#_Toc165900580)

[第3章 异步串口驱动模块的设计与实现 14](#_Toc165900581)

[3.1 串口基本机制 14](#_Toc165900582)

[3.1.1 串口简介 14](#_Toc165900583)

[3.1.2 串口的FIFO模式 15](#_Toc165900584)

[3.1.3 串口的中断处理 15](#_Toc165900585)

[3.2 整体架构设计 16](#_Toc165900586)

[3.3 异步运行时设计 18](#_Toc165900587)

[3.4 接口设计 19](#_Toc165900588)

[第4章 异步串口驱动测试 22](#_Toc165900589)

[4.1 修改 QEMU 源码使其支持多串口 22](#_Toc165900590)

[4.2 在 Alien 上使用异步串口驱动 22](#_Toc165900591)

[4.3 测试结果分析 22](#_Toc165900592)

[（4.4 在 ArceOS 上使用异步串口驱动 ） 22](#_Toc165900593)

[第5章 总结和展望 23](#_Toc165900594)

[5.1 论文总结 23](#_Toc165900595)

[5.2 不足和展望 23](#_Toc165900596)

[结　论 24](#_Toc165900597)

[参考文献 25](#_Toc165900598)

[附　录A 26](#_Toc165900599)

[致　谢 28](#_Toc165900600)

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语言针对QEMU虚拟机模拟的串口设备以及Vision Five2实体开发板上的串口设备，开发跨操作系统的异步驱动模块，并对该异步驱动模块的性能与传统的同步模块的性能在一定条件下进行比较，最终得出异步驱动模块相较于同步模块的优势。

本文的研究内容主要包括以下几项：设计并实现基于Rust语言的异步串口驱动；在AlienOS和ArceOS上适配该异步串口驱动；设计对比实验，对该异步串口驱动的性能与同步串口驱动的性能进行比较等。

本研究的关键问题包括：学习Rust语言对于异步的支持、了解Rust语言在模块化方面的支持、学习和总结已有串口驱动的实现、学习AlienOS和 ArceOS的相关结构并分别在QEMU环境下和开发板环境下运行、设计对比试验得出异步串口驱动的优势。

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。截至目前，本研究已经完成了异步串口驱动的设计与实现，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。

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。

1. 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。

1. 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。

1. 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：

1. pub trait Future {
2. */// The type of value produced on completion.*
3. #[stable(feature = "futures\_api", since = "1.36.0")]
4. type Output;
5. ​
6. */// Attempt to resolve the future to a final value, registering*
7. */// the current task for wakeup if the value is not yet available.*
8. *///*
9. */// # Return value*
10. *///*
11. */// This function returns:*
12. *///*
13. */// - [`Poll::Pending`] if the future is not ready yet*
14. */// - [`Poll::Ready(val)`] with the result `val` of this future if it*
15. *///   finished successfully.*
16. *///*
17. */// Once a future has finished, clients should not `poll` it again.*
18. *///*
19. */// When a future is not ready yet, `poll` returns `Poll::Pending` and*
20. */// stores a clone of the [`Waker`] copied from the current [`Context`].*
21. */// This [`Waker`] is then woken once the future can make progress.*
22. */// For example, a future waiting for a socket to become*
23. */// readable would call `.clone()` on the [`Waker`] and store it.*
24. ​
25. #[stable(feature = "futures\_api", since = "1.36.0")]
26. fn poll(self: Pin<&mut Self>, cx: &mut Context<'\_>) -> Poll<Self::Output>;
27. }
28. pub enum Poll<T> {
29. */// Represents that a value is immediately ready.*
30. Ready(#[stable(feature = "futures\_api", since = "1.36.0")] T),
31. ​
32. */// Represents that a value is not ready yet.*
33. Pending,
34. }

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。

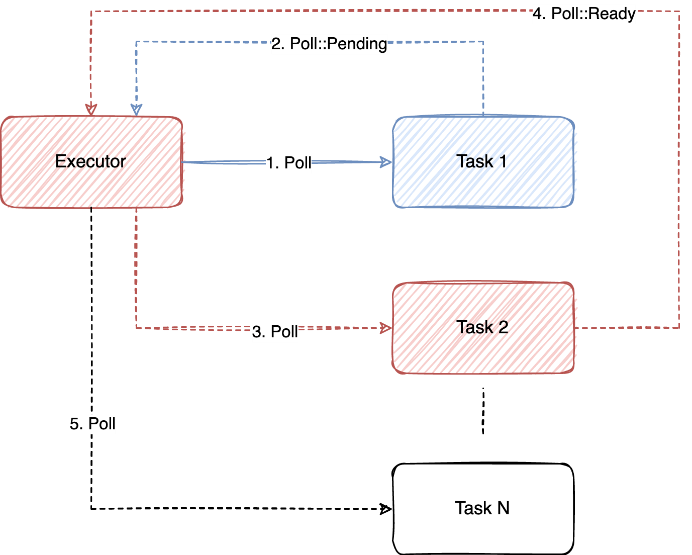


图2-1 Embassy执行器对任务的轮询设计

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：

1. *// 创建一个全局的 Embassy 默认的 Executor*
2. static EXECUTOR: StaticCell<Executor> = StaticCell::new();
3. #[no\_mangle]
4. *// 内核入口函数*
5. pub fn rust\_main() -> ! {
6. let executor = EXECUTOR.init(Executor::new());
7. executor.run(|spawner| {
8. *// 把 spawner 作为参数放入 kernel 启动函数，为支持后面的多线程*
9. spawner.spawn(kernel\_start(spawner)).unwrap();
10. });
11. }
12. #[embassy\_executor::task]
13. async fn kernel\_start(spawner: Spawner) {
14. *// ...*
15. }

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。

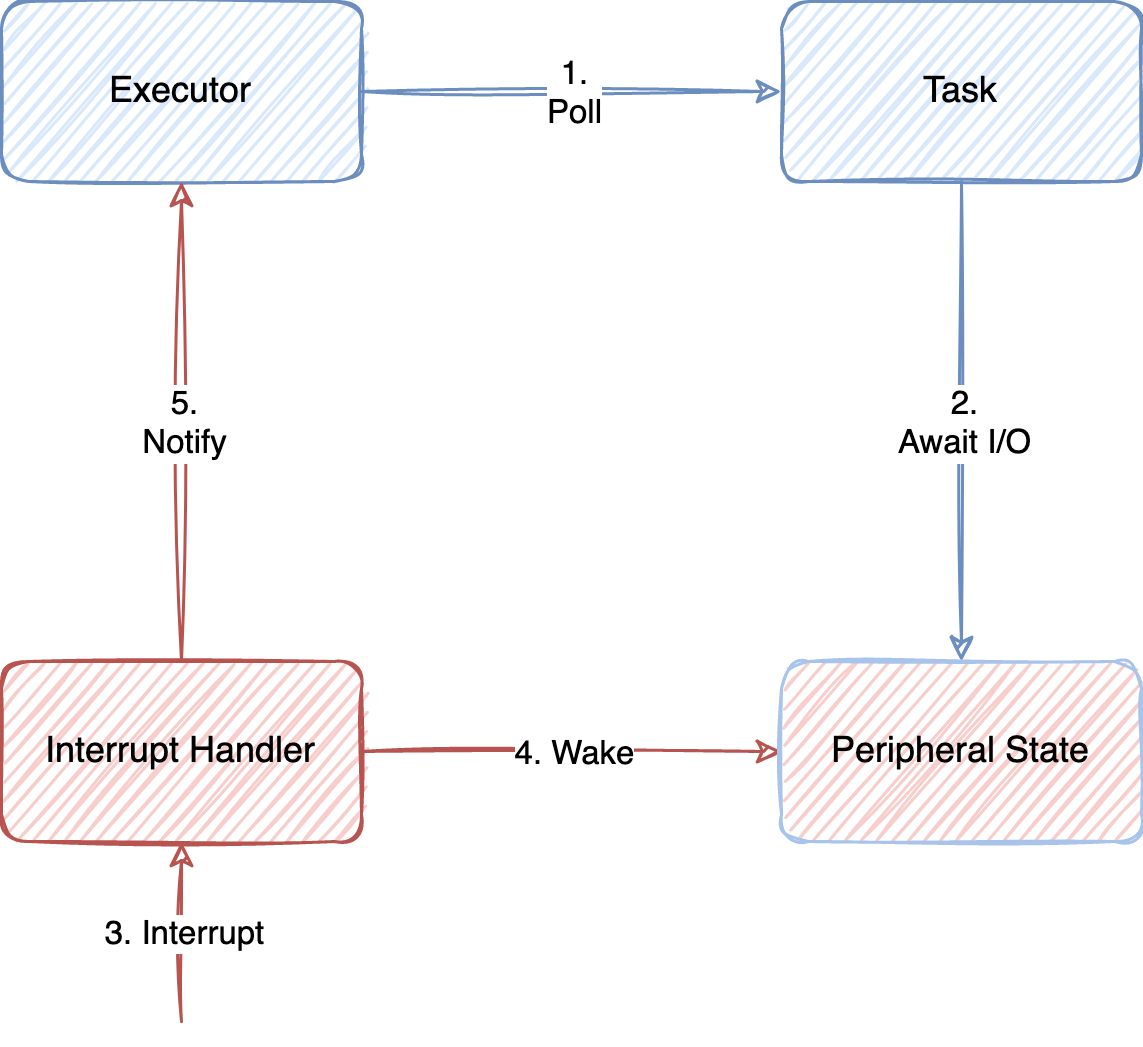


图2-2 Embassy处理中断的响应机制

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。

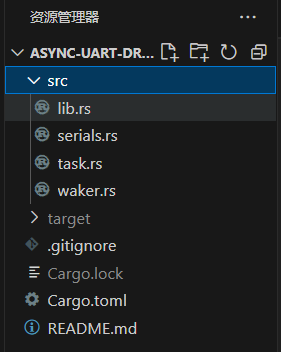


图3-1 驱动的整体代码结构

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：

1. pub struct Task {
2. */// detail value shown in 'TaskRef'*
3. pub(crate) state: AtomicU32,
4. */// The task future*
5. pub fut: AtomicCell<Pin<Box<dyn Future<Output = i32> + 'static + Send + Sync>>>,
6. */// driver*
7. pub driver: Arc<AsyncSerial>,
8. */// IO Type*
9. pub iotype: AtomicU32,
10. }
11. pub struct TaskRef {
12. ptr: NonNull<Task>,
13. }
14. pub struct Executor {
15. tasks: Mutex<VecDeque<Arc<Task>>>,
16. }

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。

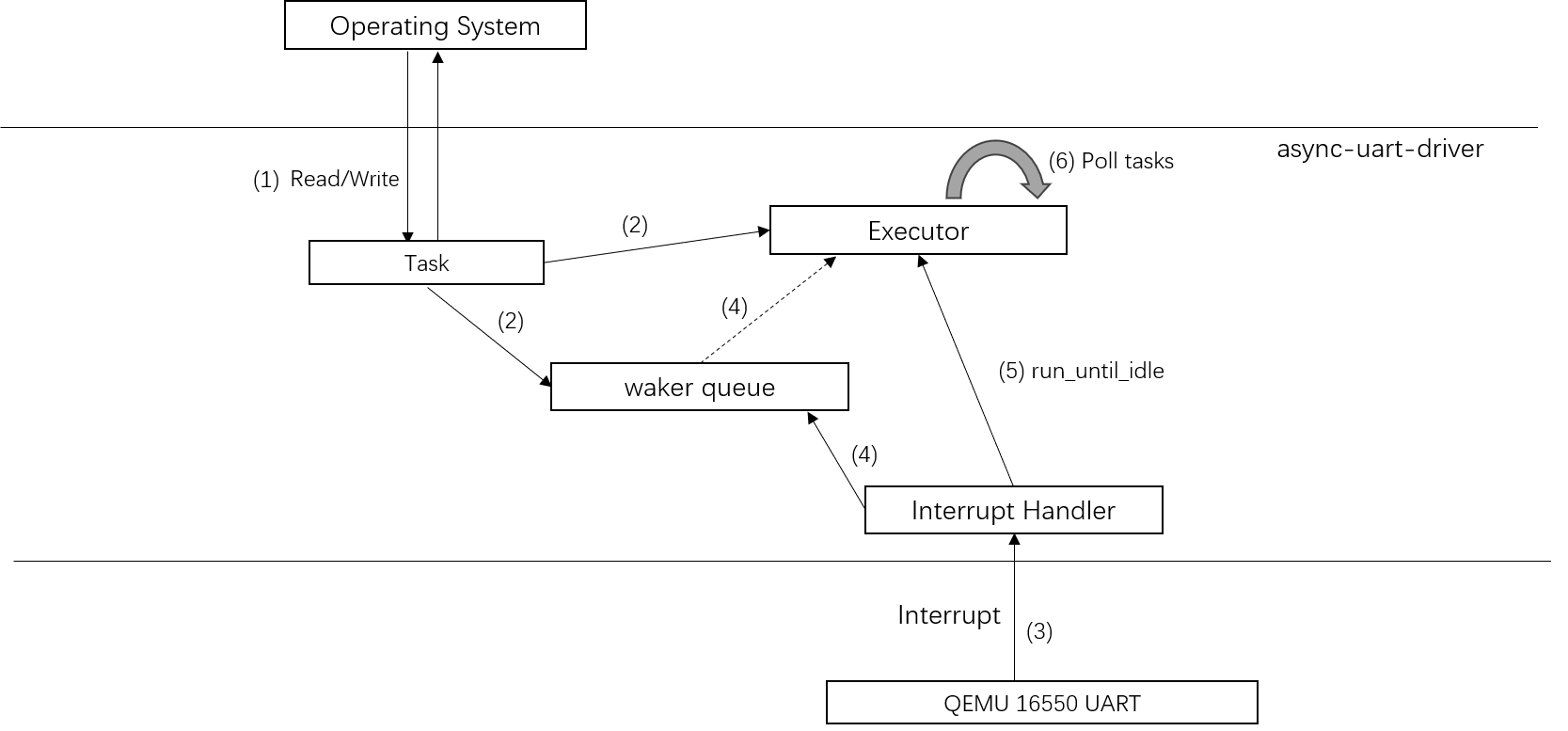


图3-2 异步串口驱动处理中断事件的流程

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：

1. pub async fn read(self: Arc<Self>, buf: &'static mut [u8]) {
2. let future = SerialReadFuture {
3. buf,
4. read\_len: 0,
5. driver: self.clone(),
6. };
7. *// 注册*
8. let task = Task::new(
9. Box::pin(future),
10. self.clone(),
11. crate::task::TaskIOType::Read
12. );
13. self.register\_readwaker(
14. unsafe { from\_task(task.clone())}
15. );
16. self.executor.push\_task(Task::from\_ref(task));
17. }

AsyncSerial的write方法也大致相同，具体实现如下：

1. pub async fn write(self: Arc<Self>, buf: &'static [u8]) {
2. let future = SerialWriteFuture {
3. buf,
4. write\_len: 0,
5. driver: self.clone(),
6. };
7. let task = Task::new(Box::pin(future), self.clone(), crate::task::TaskIOType::Write);
8. self.register\_writewaker(
9. unsafe { from\_task(task.clone()) }
10. );
11. self.executor.push\_task(Task::from\_ref(task))
12. }

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qemu-system-riscv64上启动的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后，未发现以上异常。

4.2 在 Alien 上使用异步串口驱动

4.3 测试结果分析

（4.4 在 ArceOS 上使用异步串口驱动 ）

第5章 总结和展望

5.1 论文总结

5.2 不足和展望

结　论

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。

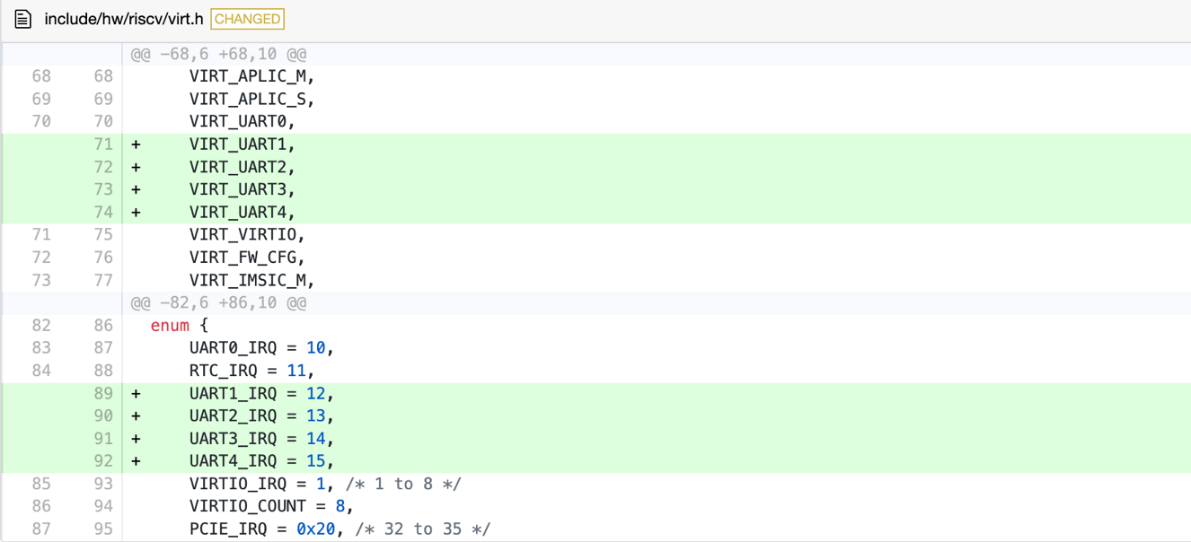


图1 为新添加的串口分配IQR号

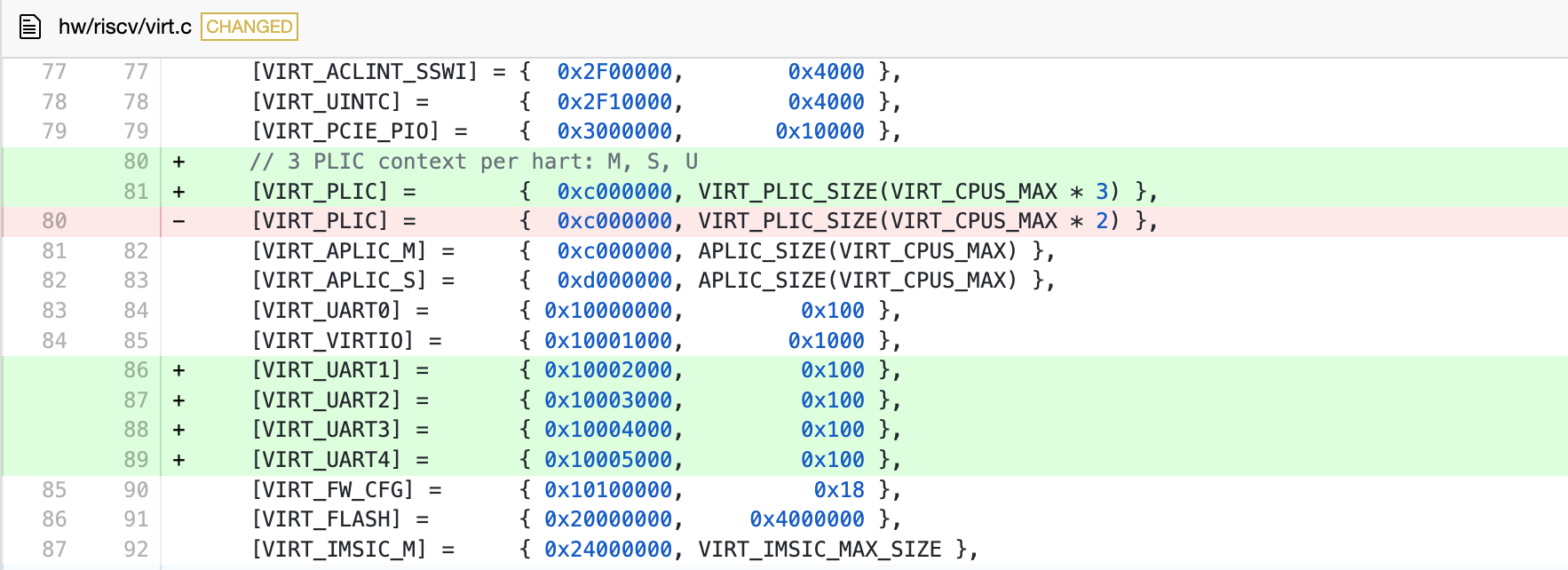


图2 为新添加的串口分配MMIO地址



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



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】