基于 Rust 语言支持的单地址空间模块隔离方法

摘　要

Alien 是使用 Rust 编程语言所开发的一个操作系统。该系统融合了基于语言的隔离机制、使用代理接口和内存隔离以保证系统健壮性与安全性、支持系统各部分的动态加载与更换。Alien 中使用的驱动程序完全使用安全的 Rust 代码编写以保证内存安全性。应用编译器保障的约束能够大幅减少由人脑保证约束可能带来的漏洞。在对驱动进行全面安全化之后，可以将其加入 Alien 或其他任一个操作系统该中，以提高系统整体的安全性。通过与不安全代码所编写的设备驱动相对比，测试对于读写性能有高要求的设备，可以看出驱动的安全化所影响的仅限于程序结构，而对性能几乎无负面影响。这证明了使用安全 Rust 代码编写驱动所带来的安全性提升不会伴随性能降低，具有实用价值。

关键词：Rust；隔离；驱动；操作系统；安全

Rust language-based modules compartmentalization for single address space operating system

Abstract

Alien is an operating system developed using the Rust programming language. The system incorporates language-based isolation mechanisms, uses proxy interfaces and memory isolation for robustness and security, and supports dynamic loading and replacement of system components. drivers used in Alien are written entirely in safe Rust code for memory security. Applying compiler-guaranteed constraints dramatically reduces the vulnerability that can result from human-guaranteed constraints. After a driver is fully secured, it can be added to Alien or any other operating system to improve the overall security of the system. By testing devices with high read/write performance requirements against device drivers written in insecure code, it can be seen that driver securitization affects only the program structure, with little to no negative impact on performance. This demonstrates the practical value of using secure Rust code to write drivers with no performance degradation.

Key Words: Rust; isolation; driver; OS; safety

目 录

第 1 章 引入 1
第 2 章 系统设计 5
2.1 模块化 5
2.2 隔离域 5
2.2.1 交互接口 6
2.2.2 内存隔离 6
2.2.3 故障隔离 7
2.2.4 动态加载 9
第 3 章 设备驱动 10
3.1 VirtIO 驱动 10
3.1.1 与系统的交互 10
3.1.2 与设备的交互 12
3.1.3 块设备 18
3.1.4 输入设备 18
3.1.5 网卡 19
3.2 Uart16550 驱动 19
3.3 在 Alien 中的适配 20
第 4 章 测试 22
4.1 功能测试 22
4.2 性能测试 23
4.2.1 块设备 23
第 5 章 总结 25
参考文献 26

第 1 章 引入

操作系统的最初目的是调度硬件资源和促进计算机上多项任务的执行，而现在已经有了长足的发展。随着计算机硬件和软件的不断进步，出现了 Windows 和 Linux 等主流操作系统，其特点是包含了操作系统所需的大部分基本功能。然而，这种功能集成带来了潜在的安全漏洞，因为操作系统中的任何错误都可能导致整个系统崩溃。操作系统中错误的来源多样，可能由于用户的误操作带来负面影响，也可能有恶意攻击者对操作系统或其中的某些部分进行攻击。[1]

多用户操作系统的出现要求了操作系统对于用户进行的权限管理和隔离，以避免不同用户之间的相互影响。而多核处理器的出现与多线程操作系统又带来了新的攻击方式、新的安全问题和挑战。操作系统开发人员需要为系统设计管理和访问的权限，如强制访问控制（mandatory access control），并对其进行验证以确保其安全方案是正确的（如使用形式化验证[]），能够满足安全性需求。MAC 架构可以对所有的下属对象（如用户和进程、内存、文件、设备、端口等）。现代处理器使用多特权级管理，把系统分为内核态（ring 0）和用户态（ring 3）。这一方案精细化了隔离的粒度大小，提供了硬件层面的保护机制，在不同特权级的切换之间需要进行权限验证，以此保护在复杂的计算机系统中保护内核和用户应用的正常运行。通常，一个进程无法修改属于更高权限的资源文件。ring0 特权级中的代码将会被视为可信任的。https://www.giac.org/paper/gsec/2776/operating-system-security-secure-operating-systems/104723

开发安全内核的基本原则是使用经过形式化定义的安全模型，满足完全、强制化、安全需求被检验满足等条件。之后，该模型需要被正确的使用编程语言实现出来。操作系统在任何计算机系统中都扮演着重要地位，因此任何在其中出现的问题都将会威胁到整个计算机系统和其上运行的所有软件。一个被入侵的应用可能导致另一个设备也被入侵。

在现代操作系统开发中，安全性的重要性日益突出。这促使开发人员寻求能最大限度减少操作系统出错的解决方案。在这种情况下，微内核的概念应运而生。微内核方法旨在通过将必要的功能（如文件系统和硬件驱动程序）委托给用户空间程序来简化操作系统本身的复杂性。微内核的主要优势在于该架构可以提供更安全和稳

[1]TRENT JAEGER. The Evolution of Secure Operating Systems[EB/OL]. https://www.cse.psu.edu/~trj1/cse543-f18/slides/cse543-secure-os.pdf.

定的操作系统。这种降低复杂性和分离功能的做法降低了开发和维护的难度，同时也减少了出现漏洞的可能性。此外，在用户空间运行操作系统模块可降低其权限，从而减轻某些类型的攻击。因为只有必要的服务在内核空间中运行，微内核操作系统拥有更小的攻击面，让攻击者更难找到其中的漏洞。并且，一个用户级的进程的崩溃不会影响到整个操作系统，因为微内核只负责管理进程和内存。微内核架构的另一个优势是它使得操作系统更加灵活和模块化。在用户态运行的服务可以更容易的被替换、移除或者重置而不影响系统的其他部分。这令操作系统可定制化，可满足特定需要。微内核的缺点则在于内核态和用户态的分离带来了大量的特权级切换，这使得微内核系统比宏内核更慢，尤其是在对系统调用性能要求高的应用场景当中。https://www.geeksforgeeks.org/microkernel-in-operating-systems/

现有的操作系统安全解决方案主要依赖于基于硬件的隔离，如 ARM TrustZone。这种方法定义了一个可信区，只有安全性要求高的程序才能在此执行，确保与普通区完全隔离。TrustZone 最初在 ARMv7 发布时提出，并逐渐被加入到芯片中。该处可信区只能被可信的系统和软件读取，其他任何调试方法和硬件访问都会被阻止。对其的访问需要通过密钥验证，密钥被写入芯片中，使得后期的攻击无法成立。用 VMFUNC 切换页表、内存保护密钥、内存标签扩展等硬件隔离技术提供了一些低负载的硬件隔离机制，其负载量与函数调用时的开销相接近。

然而，目前的硬件隔离机制存在一定的局限性。首先，它们需要特定的硬件支持，因此成本较高，而且缺乏通用性。其次，基于硬件方法的实施缺陷可能会带来不易修改的漏洞。并且由于需要其保存通用与扩展寄存器，还要切换栈空间，就算在理想的地址空间切换不耗时时，其与基于 Rust 语言的隔离相比开销仍较大。Rust 语言在实现隔离的同时避免了这两个开销，它的开销与函数调用相同。在实际的应用场景下，Rust 只比 C++实现慢 4 8%。[Understanding the Overheads of Hardware and Language-Based IPC Mechanisms]

为了解决这些不足，人们正在考虑基于软件的安全隔离解决方案。基于软件的安全隔离解决方案有望提供更大的灵活性和适应性，而且可以在各种硬件平台上实施。这些技术的开发有望提高操作系统的安全性，降低与系统崩溃和漏洞相关的风险。Nooks 提出了隔离域的思想，并给出了一些安全约束，如故障隔离的思想。奇点操作系统开创性地使用一种新的安全语言来开发操作系统内核，带来了编译器保证的系统安全研究方向。目前已有多个项目在开展操作系统层面提高系统隔离性和安全性

的尝试，包括 J-Kernel、KaffeOS、红叶操作系统、Theseus OS 和 Asterinas 等。它们旨在通过软件层面的设计和实施、安全编程语言的使用来提高安全性。

J-Kernel 认为仅靠语言安全不足以保证故障隔离和子系统更换，因此它基于 Java 语言实现了数据传输的接口和代理。其中的每个对象在传递时都会经由一层代理进行包装，并传递一个特殊的引用对象。该引用对象可以调用方法，该系统可以把各个隔离对象进行分割，但是这个过程中需要对非引用而是直接传输的参数进行深拷贝。由于内核的工作常见大量数据的处理传递，这种函数调用时增加的巨大开销是很难接受的。此外，由于其传递的引用没有做可用性检查，因此它的故障隔离是单向的：J-Kernel 可以阻止调用者的崩溃影响到引用对象的创建者，但是当引用对象创建者本身崩溃时，其调用者之后的使用将也会崩溃，并未完全做到故障隔离。

KaffeOS 没有使用代理，而是使用了写屏障技术。在 KaffeOS 中存在两种堆对象，分别是私有堆和跨域共享堆。它对于共享堆上的对象设置了称为写屏障的约束，该约束管理所有隔离部分对其的引用与垃圾回收。然而，这个模型的缺陷在于当一个隔离部分在对共享堆上的数据修改到一半却崩溃了时，共享堆上的数据将可能保留在不合法的状态并就这样被其他的隔离部分所读取，此时故障将传递给这个正在执行读取操作的部分。此外，隔离对象也需要在跨域调用的时候进行深拷贝，并且动态检查指针和垃圾回收带来的开销也无法忽视。

奇点操作系统使用了全新的模式来进行故障隔离。它重点使用了编译期即可确定的静态所有权来完成这一目标。类似 KaffeOS，奇点也使用了交换堆和私有堆。它创新地为交换堆用于在各个隔离部分之间传递数据而无需深拷贝。这些约束实现的保证都在于其开发者发明的 Sing# 语言：这种专门为安全隔离 OS 设计的语言使用了多种新颖的静态分析和验证技术，能够满足所有权约束系统。奇点规定了每个交换堆上分配的对象只能拥有单一所有权，没有所有权的域不能对其进行访问。在跨域调用的时候，交换堆上的对象的所有权会被转移到被调用者所在的域。因此，崩溃的域无法影响到其他的部分，它不会让已经存在的引用消失（因为这样的引用根本就不允许被创建），也无法让正在被引用的交换堆对象处在不合法的状态中。此外，所有权的转移实现了零复制拷贝，减少了函数调用时数据传递所带来的开销。移动语义保证了该对象是当前域专有的，因此接受者可以随意对其进行修改。奇点的系统设计严重依赖于其开发的 Sing# 语言，因此难以被广泛使用，但是其思想非常值得参考。

红叶操作系统受到了 J-Kernel、KaffeOS、奇点的启发，使用 Rust 语言原生的所有权系统和内存安全性来执行故障隔离。在数据保存和移动时，使用了 KaffeOS 和奇点的共享堆于私有堆，并且强制规定共享堆上的数据不能拥有可变引用，因此支持奇点的零拷贝通信。使用了 J-Kernel 的代理技术来规范跨域的函数调用，使用 IDL 来为各个隔离域设计接口。红叶的故障隔离规范并不依赖于 Rust，而是被其独立出来，抽象为隔离规范的集合。在红叶中，跨域执行的函数调用无需进行线程的切换，而是直接把线程移动到了另一个域中。此方法可以减少系统中的线程上下文切换，同时提高了代码的自然性。

Theseus OS 是对操作系统模块化和动态更新进行尝试的一个操作系统。它致力于减少各个组件进行交互时所需的状态，对其进行了最小的、明确的定义，并构筑边界使得这些状态不会在运行时被其他组件进行修改。此外，它也使用了 Rust 语言，借助其强大的编译器与机制来确保一些操作系统中的概念正确实现。

Asterinas 是一个用 Rust 编写的支持 Linux 适配的操作系统内核，这意味着它可以用于取代 Linux，并伴随着内存安全语言带来的一系列好处。它限制了内核中不安全代码的使用，定义了一个最小的可信代码块（TCB）用于放置必须的不安全代码。该内核的 framekernel 架构将内核划分为 TCB 和剩余部分。其中 TCB 会包装内部的低级代码并对系统其余部分提供高级的交互编程接口，而其余部分则必须完全用安全代码写就，以阻止对内存的不安全操作，减少风险。TCB 和其余部分运行在同一地址空间中，通过函数调用进行交互，因此其效率能做到与宏内核相近。微内核所提供的安全性保证则交由 Rust 编译器在编译期间完成检查。同时 Asterinas 提供了对开发者友好的工具 OSDK，用于把其各个模块的开发过程规范化，提高效率。

受到红叶、Asterinas 等系统的启发，Alien 操作系统的 isolation 版本（以下简称 Alien）尝试实现一个基于 Rust 的安全性开发，在内核中使用多个隔离域封装内核各部分功能以实现故障隔离，使用 TCB 并限制不安全代码的使用以加强内存安全性，支持对各域进行任何时候的动态加载的操作系统。通过探索基于软件的安全隔离方法，本项目旨在为操作系统中的安全设备驱动程序领域做出贡献，并深入探讨与此类解决方案相关的潜在优势和挑战。

第 2 章 系统设计

为了实现项目的目标，即提高操作系统的健壮性与安全性，Alien 使用了多种设计模式和约束。如模块化操作系统各组件、严格的隔离机制、各部分的动态加载与替换。这些新概念的引入使我们可以探究下一代操作系统可能的形式，为用户带来更加优秀的使用体验。

2.1 模块化

Alien 采用了微内核的设计，因此可以把文件系统、驱动等部分从内核中独立出来，内核中仅含有必要部分。而在这之上，微内核本身的功能也被划分为内存管理、任务调度、系统调用等多个不同的部分。

在内核中使用模块化的好处与在应用软件中使用模块化的好处一致：内核的各个部分将会高内聚和低耦合。（1）模块之间的分离带来了更小的开发和编译单元，这使得每个单元内部独立的测试会更加容易，减少了出现错误的可能性，同时在出现问题进行调试的时候，也可以快速直观的通过检查穿过接口的数据来对错误进行定位，让开发过程更有效率；（2）让代码各部分的分工更加明确，易于阅读，提高设计和实现的一致性；（3）可以拥有更加细粒度的权限赋予、功能划分。这使得完成某一功能的代码不会被授予需要之外的权限。权限的最小化降低了漏洞和恶意代码带来的风险。

Alien 中被划分出来的各模块包括内存管理、互斥锁、常量与类型定义、系统配置、架构、硬件平台、文件系统、各设备驱动、任务调度器、系统调用、日志等。其中部分模块可以在 github[os-module]上找到。

2.2 隔离域

隔离域是 Alien 中的一个隔离单位。它将会被独立编译，并且需要满足开发者规定的一些安全性约束，如内存、接口、源码层面的。在 Alien 中，大量的模块都被写成了隔离域。而剩下一些无法作为隔离域存在的模块则归入可信代码块（TCB）。

内核中对于隔离域的处理使用了一个加载器模块。同时，为了给隔离域提供它所要依赖的功能，还有专门的一个包用于隔离域和内核的交互。隔离域所需的函数有申请可交互结构（在共享堆[]上分配内存）、读取系统设置、向系统输出日志等，这部分内容由内核核心部分完成并可以被隔离域调用。

我们系统设计的目标是，每个隔离域内出现的任何问题都不会影响到系统的其他部分，从而保证系统的健壮性。为了实现这一目标，对于每个隔离域，我们要求它们满足如下约束：

1. 故障隔离：一个隔离域程序出现无法恢复的故障时，内核不会停止运行，而是可以对此情况进行处理。
2. 内存隔离：一个隔离域内不能有指针指向其他隔离域中的对象。这能让系统回收任何一个隔离域时都无需担心其他隔离域受影响。
3. 动态加载：在系统运行时，对于崩溃的隔离域需要进行重启或更换。这需要系统有在运行时加载并使用一个隔离域的能力。

为了满足以上约束，我们设计了对于接口定义和内存使用的一些机制。

2.2.1 交互接口

交互接口是每个隔离域与域外任何部分（包括内核和其他隔离域）进行通信时必须要使用的交互方式。在一个普通的操作系统中，各部分之间的交互是通过函数调用实现的。即被依赖的部分实现一个函数，内部完成所需功能，并且把函数的入口暴露出来，允许其他部分在代码中调用这个函数，从而完成功能。这一点在 Alien 中也是一样的。但是由于 Alien 为每个隔离域额外规定了安全约束，因此我们在进行域和其他部分之间的交互时，不能进行任何可能使安全约束遭到破坏的函数调用。

从具体的安全约束出发，对交互接口的规定为了满足内存隔离约束，需要在跨越域边界进行函数调用时，传递的参数和返回值只含有实现了 `Copy trait` 的类型（这使得该参数的数据可以直接通过逐字节拷贝来复制，而不影响其内容的有效性）或存在于共享堆上。任何存在于某一个域私有堆上的数据都不能通过函数调用的参数或返回值方式穿越域边界。通过这种方式，我们构造出了能够满足内存隔离的接口要求。

同时为了满足故障隔离，接口的返回值要做好收到错误的准备。我们要求所有的返回值都必须被放置于定义好的类型 `RpcResult<T>`中。因此在域崩溃而无法正常返回所需值的时候，调用者可以收到一个错误而不是跟着一起崩溃。这使得隔离域的交互接口同样满足了故障隔离要求。

2.2.2 内存隔离

不同部分所使用的内存相互暴露经常带来严重的安全问题[]。在现代操作系统中，一种常见的保护内存方式是使用虚拟地址空间，当一个用户程序尝试访问不属于

它的地址时，操作系统或硬件会检测到这个非法操作并产生错误[]。而在 Alien 中不仅对于用户程序进行了内存隔离，对于内核的部分组成模块也进行了隔离域化，为其带来了内存隔离特性。

每个隔离域都会拥有一个单独的堆作为数据存放位置，它被叫做私有堆。同时，存在一个所有隔离域都可能访问的堆，称为公有堆。任何隔离域都不被允许访问其他隔离域的私有堆，任何可能产生这一后果的行为（如在域交互接口中传递指针或引用、从地址直接构建出指针）都会被编译器拒绝。借由这点，我们的操作系统确保了各个隔离域之间堆内存的独立性。此举带来的好处是，我们在任何时候都可以安全的释放一个隔离域所拥有的所有内存，因为没有该隔离域外的指向此处的指针存在，因此把私有堆释放不会带来任何垂悬指针问题。这为域崩溃后的回收和重载提供了基础。

对于有大量数据需要在域之间进行传递的情况，系统提供了一个共享堆用于处理。借助传递指向共享堆上数据的指针，可以实现大量数据的 0 拷贝传递。共享堆上的数据也必须满足对于故障隔离的约束，即当某个域崩溃时，它所遗留下的对象不会对其他任何域产生影响。因此，每个共享堆上的对象都必须满足如下原则：（1）它在任何时刻拥有且只拥有一个所有权域。（2）它不能存在任何形式的可变引用，只能有不可变引用和所有权转移发生。

系统提供了一种独特的类型 `RRef<T>`。它实际上是共享堆中的物体在代码中的对应，该类型相关的 API 也是隔离域被允许使用共享堆的唯一方式。当隔离域尝试构建一个 `RRef<T>`对象时，系统会在共享堆中分配一块内存，构建出一个 `T` 对象，并记录该对象现在属于哪个隔离域。在跨域进行调用，使用 `RRef<T>`作为参数或者返回值时，系统会自动的更新该对象的所有权域。当一个域崩溃时，检查它拥有的所有共享堆对象，这些对象此时所有权都在崩溃的域中，并且其他域对它不可能有可变引用，只有不可变引用。因此该对象能够在所有对于它的引用都结束后安全的被释放。如果崩溃的域正在尝试对其进行修改，那么由于编译器的限制，其他域不可能拥有对它的任何引用。而如果其他的域对其有不可变引用，那么该对象的值不会改变，将会一直维持直到所有的引用消失。这整个过程中对象的值都与此时第一个不可变引用出现时对象的值相一致，满足其他域对其的读取要求。

2.2.3 故障隔离

故障隔离的主要特点是，可以在某一个隔离域崩溃时，正常地向内核其他部分返回一个 Error 类型的错误，而不是让系统出现无法运行的严重错误。我们随后可以对于该域进行重载等操作。

一个正常程序崩溃可能会有很多原因，但是一个共同点是它将被视为无法靠其自己的逻辑解决此错误。在这种情况下，rust 程序在收到崩溃的通知后，会跳转到一个特殊的函数入口，被称为 panic_handle()[]。通常情况下，这个函数会打印出崩溃时的调用栈和寄存器信息，供开发人员分析崩溃发生的原因。在 no_std 环境中，这个函数需要程序员自己实现，而 Alien 对于每个隔离域，都会自动生成一个错误处理函数，它干的事除了打印相关信息之外，还会让处理器跳转回到这次对于隔离域的调用之前的状态，这是通过记录即将进行调用时的寄存器信息并将其在错误处理函数中恢复而实现的。在恢复之后，本来处理器将要执行的程序应该是隔离域内的该函数，但是由于隔离域已崩溃，我们让处理器转而执行一个专门返回错误的函数。这样就防止了隔离域的崩溃带来的整体系统崩溃，而能让系统继续运行。

因此调用者可能收到的回复除了正常时候的返回值以外，还可能是一个错误。这通过 Rust 语言提供的 Result 类型来实现。Result 是 rust 预定义的一种 enum 类型，可以存放正常执行代码后得到的结果或者代码故障产生的错误。通过强制所有域接口都返回一个经过我们包装过的 RpcResult<T>类型，可以确保隔离域在崩溃时调用者收到合适的返回值，做到故障隔离。

当一个域崩溃后，可能还有一些部分的代码存有该域对象，并且想要使用其完成一些工作。此时由于该对象实际已经不存在并且被回收了，因此对其的任何调用都将是错误的。为了避免这种情况，系统需要在该域被调用的时候进行一次判断，验证其是否正常存在。如果不存在，则得益于上文提到的返回值约束，可以直接向调用者返回一个错误表明域已经崩溃。否则，再把调用请求传递给域让其完成功能。这部分实现我们使用域之外的一层代理来实现。该代理内会引用开发者实际编写的域，任何尝试对域的调用都会通过代理来交互。让代理和原隔离域之间感受不到差异的透明度主要依靠 Rust 的 trait 系统实现。任何域提供的函数接口都是以一个 trait 的形式展现，而只要代理实现了这个 trait，那么对于调用者来说它的行为就与原隔离域没有区别。调用者不会关注 trait 之下是什么结构如何实现的。同时，代理也需要接受域执行一个函数调用的中途可能发生的崩溃并准备好一个返回错误的函数供 panic_handle 处理时使用，向调用者返回错误。

2.2.4 动态加载

在某个隔离域发生崩溃之后，我们不仅希望系统能够继续运行，还希望能在调用者没有感知的情况下对崩溃的域进行恢复。这就要求系统具有动态卸载、加载隔离域的能力，同时还需要能够保存某次调用时传入的参数。这样在收到域崩溃的信息时，系统不仅可以选择直接向调用者返回一个错误，还可以选择重新加载一次崩溃的域或者使用其他功能相同的域对它进行替换，然后再执行之前引起崩溃的那次函数调用，向调用者返回正确的结果。

为了达到这一目标，我们使用了一个影子域，它的作用是保存调用时传入的参数并尝试恢复该崩溃的域。由于我们传入的参数的所有权在到达代理域的时候就已经被转移了，因此这一工作需要在外部影子处来完成。通过新的这一层封装，可以实现隔离域的透明故障恢复，同时也可以支持了用户在运行时对某个隔离域进行切换。

为了使各个隔离域能够动态加载和切换，对其他隔离域的依赖不能通过简单的传入一个该域的对象来完成。这是因为当想要替换该作为依赖的域时，使用它的域没有高效的方法得知这一消息并进行替换，因此 Alien 使用另一种方案，以使使用者不直接持有被依赖域的对象。我们使用一个 BTreeMap 数据结构存放所有域对象和它的标识。当某个域想要使用另一个域的功能时，只需使用系统给隔离提供的系统调用之一，从标识符从 BTreeMap 处获取当前系统中使用的该功能域的具体对象即可。这使得系统的灵活性增加，能够支持动态切换隔离域。

第 3 章 设备驱动

在 Alien 系统中，内核主要功能由其他开发者完成。本文介绍的工作在于为其开发设备驱动的隔离域部分。

为了提高安全性，同时也为了满足 Alien 隔离域的要求，我们使用了内存安全[] 的语言 Rust 重新实现了系统中的设备驱动，并且在驱动的代码中没有使用被 Rust 定义为不安全的操作[]。

Rust 中的不安全操作通常是对于直接以 `usize` 形式展示的内存地址的访问。而这种操作又是驱动程序必不可少的所需操作之一。为了解决这个矛盾，我们把目光从驱动程序移到了内核 TCB 上。操作系统内核的开发也不可避免的需要用到不安全操作。为了尽可能的减少漏洞出现的可能性，最大化安全性，Alien 只会有一个核心模块 TCB 使用不安全代码，而内核的其他部分则依赖 TCB，（以 `trait` 的形式）使用它提供的接口。那么驱动程序就可以效仿内核，把所需的不安全操作抽象成若干接口，并要求内核来实现这些接口。这样，只需要保证在内核 TCB 中的接口实现是符合语义、安全的，就可以提高安全性了。

这样做的主要好处在于，减小了开发者需要人脑考虑的安全隐患范围。当整个程序都是内存不安全的语言所写时，开发者需要自行确保其代码的安全性。而人脑是难以避免纰漏的。通过把大部分内存安全的检测工作交给 Rust 的编译器来完成，可以极大减少这种人脑产生的纰漏。

工作内容集中体现在，如何把驱动程序所需的所有不安全操作抽象为尽可能少的接口。在这些接口之上，依赖它们使用完全安全的 Rust 编程语言完成驱动程序应有功能的开发。以及在这个过程中，如何保持性能不衰减。对于这些问题，我们分别使用了对于头文件的交互接口、虚拟队列的交互接口来进行抽象和封装；使用传递引用和提供不同层次的 API 来满足使用者对高性能的需求。

3.1 VirtIO 驱动

VirtIO 设备是一种专门为虚拟环境设计的虚拟设备。Alien 现阶段在 qemu 模拟器中运行，因此我们适配了一些 qemu 提供的 VirtIO 设备。包括网卡、输入设备（鼠标、键盘、触摸板等）、块设备（硬盘等）、显示设备、控制台设备。

3.1.1 与系统的交互

VirtIO 驱动扮演一个桥梁的角色：它与设备交互，接受设备传来的信息并向设备下达指令；同时它也需要与系统交互，把设备输入传给系统，同时接受系统想要对设备进行的操作。

在驱动与系统的交互中，总共需要两种类型的接口，分别是：（1）TCB 向驱动提供的功能，即驱动需要依赖系统的部分。这部分接口允许驱动直接读写某块内存区域，或者获得位于特定内存位置处的数据结构，进行物理、虚拟地址转换等需要操作系统内核帮助才能完成的功能。（2）驱动向系统提供的功能。这部分接口运行系统通过驱动对设备进行操作，包括构建驱动进程、对设备进行设置、操作等。

3.1.1.1 对系统的依赖

为了做到能独立编译，系统向驱动提供依赖的形式以 Rust 编程中常见的 `trait` 来完成。具体地说，系统在尝试创建一个驱动对象时，需要调用其构造函数 `new()`，而这个函数最终则要求传入一个具有我们要求的某种 `trait` 的对象（`Box<dyn VirtIoDeviceIo>`）作为参数。因此驱动的使用者需要编写一个结构体，并为其实现 `VirtIoDeviceIo`。

这个 `trait` 所要求的功能是对一段固定的内存进行读写操作，而这段内存将会被操作系统以某种方式映射到设备地址空间（mmio），即与设备的寄存器相对应。对这段内存的操作就相当于对设备进行操作。

同时，构建驱动结构体的时候，还需要传入一个结构作为驱动所需的泛型类型。该结构需要实现 `Hal trait`，这是为了创建虚拟队列，让驱动能够通过虚拟队列与设备交互。虚拟队列相关内容将在 3.1.2 描述。通过 `Hal`，驱动能够要求操作系统内核为其分配一段连续的内存页来放置虚拟队列，并且在正确的地址从从 `usize` 形式的位置开始构建出队列，并传给驱动使用。

3.1.1.2 向系统提供的功能

各驱动根据所对应的设备不同，将会为系统提供不同的功能。如块设备就会提供在特定的块进行读和写的功能。此外，所有驱动都会有创建自身和对设备进行设置（如是否接受中断）的功能。

因为驱动的形式是一个结构体，所以提供功能的形式就是公开的方法。使用者可以直接调用驱动结构体所拥有的方法来完成功能。

每个驱动各自拥有的功能将在各自的章节详细描述。

3.1.2 与设备的交互

驱动与VirtIO设备的交互通过一段被映射的内存和若干个虚拟队列进行。这些结构都由 VirtIO 的常见组织规定，并且在其文档[]中描述。

3.1.2.1 内存映射（MMIO）设备寄存器

VirtIO 设备支持两种与操作系统的交互方式：通过 PCI 总线交互与通过虚拟内存映射交互（MMIO）。Alien 中采用了后一种，即把真实的寄存器区域应映射到一段内存中。被映射的内存将会连接到设备地址空间，在这里进行的操作相当于对设备寄存器进行操作。这部分由操作系统完成。

所有 VirtIO 设备的地址空间首先会有一个长度为 256 字节的结构，被命名为 `VirtIOHeader`。这个结构中有些寄存器是只读的，有些寄存器是只写的，还有一些是可读可写的。这个结构的作用包括让系统认识到这个设备的类型、使用协议版本、支持的特性、支持的队列数量、设备容量等信息，同时允许系统向设备进行通信，来商定启用的特性、队列数量、队列地址等。这个结构是所有 VirtIO 系列的虚拟设备都会拥有的，其结构也完全相同，是驱动设置区域中通用的一部分。

在 `VirtIOHeader` 之后，是每种设备不同的设置区域。这段区域的作用是对各设备特有的属性进行设置、通信、协商。它的长度和布局并不统一。

此两处区域主要用于对设备进行初始化和设置。在系统认知到虚拟设备后，就将创建其驱动。而任何一个 VirtIO 设备的驱动都需要首先对该区域进行设置，以满足设备的使用需求。该区域的头 4 个字节是表明该处地址段是一个 VirtIO 虚拟设备的映射空间的标识符，又被称为魔数。其值必须为 0x74726976，是小端序下对于字符串`"virt"`的编码。驱动只有在认知到这个数之后，才可以把该内存区域当作一个 VirtIO 虚拟设备的设置区域的映射来处理，否则驱动应该拒绝对其执行初始化操作。

在实际使用中，我们把寄存器的布局使用常量泛型的方法硬编码在了程序中。驱动程序不能直接执行这些不安全操作，因此需要操作系统向我们提供一个有 `VirtIoDeviceIo trait` 的结构。该结构允许驱动从地址空间的起始处按照某个偏移值对内存进行读写操作。在原有的不安全实现中，此处的做法是预先使用 C 风格的内存布局定义好一个设置头的结构体，该结构的空间分布与 VirtIO 设备规定的恰好一致。因此所创建出来的结构可以直接对想要执行操作的部位进行读写操作。对于其中的每一个寄存器，它都使用了一个读写权限的包装结构，用于限制失误导致的对其的非正确访问，如尝试读取一个只写寄存器，或尝试写入一个只读寄存器。我们对于设置

头也有一个结构，但是该结构并没有实际的内存布局，而是以常量泛型的方式记录了每个寄存器的位置对于设置头开头地址的偏移量。该常量泛型会存放在我们定义的读写权限包装结构中。由于常量在 Rust 编译期间会直接被解析并生成为访问正确位置的机器码，因此此方案没有额外的运行时开销。在知道了寄存器相对于设备空间起始位置的偏移量之后，读写权限包装结构所含有的泛型读写函数会使用拥有的常数泛型参数作为向系统申请对特定内存区域进行读写操作时传入的参数。实际被生成出来的代码类似于在函数调用时使用起始地址加偏移量的直接硬编码调用。

至此，我们完成了对于设置头部分的读写封装，而所需要的接口仅要 4 个：对某一内存区域特定偏移量分别进行 32 位无符号整形、8 位无符号整形的读取和写入。由于目前 Rust 还不允许把含有泛型的函数放在 trait 中作为参数或返回值被动态在函数调用中传递，因此只能使用固定的类型，产生了 8 位和 32 位两种规格但实质相同的接口。如果只使用 8 位无符号整形进行读写也可以完成全部功能，但是这会导致 32 位或 64 位无符号整形的读写需要被对应拆成 4 个或 8 个读写操作，大大减慢了操作速度，因此加入了 32 位无符号整形以保证速度。

至于各设备各有的设置区域，对其的读写结构封装与设置头大体一致。但是由于设置头中储存数据的类型只有 32 位无符号整形，但是在各设备区域中需要的类型多种多样，从 8 位无符号整形到 64 位，还有一些设备需要读写数组，因此对于读写权限结构中内含的数据类型使用了泛型。并对各个可能出现的具体类型需要分别实现不同的读写函数以把读写操作正确地转换为对系统的函数调用。目前需要对共 4 种类型的 3 种读写权限全部进行实现，该部分代码冗余较多而重复性较强，可以使用宏实现。

借由此封装，驱动可以安全的对设备的地址空间进行读写操作而只使用到了安全代码，无需担心对内存地址可能的混淆带来的负面后果。操作系统将要负责检查驱动调用函数时传入的参数是否合法，随后才进行不安全代码的执行。需要被注意的风险面就缩小到了仅仅需要被提供的 4 个函数中。

3.1.2.2 虚拟队列

一个设备可以拥有一个或多个虚拟队列（VirtQueue），虚拟队列的作用是允许驱动向设备发送具体的操作请求，并从设备处接受操作后的结果。一个虚拟队列是唯一一段连续物理页上的三个数据结构，分别是描述符表（Descripter table），可用环（Avail Ring），已用环（Used Ring）。其中描述符表是由若干个描述符（Descripter）组成的数

组，而每个描述符内含指向一段内存区域的指针、该区域的长度和该区域的属性（设备是否可读写、是否还有下一个描述符，下一个描述符的下标）。每个驱动对设备的操作请求的形式都会是一个描述符组成的链。而具体是什么类型的操作、操作所需的传入参数、所需的放置结果的内存地址，都会放在描述符所指向的区域中。可用环和已用环则是两个使用数组实现的队列，其中分别存放需要设备完成的操作和设备已完成的操作。每个操作都占据数组中恰好一个位置，表现形式是该操作的描述符链的首个节点在描述符表中的下标。

每个虚拟队列的长度都可以由驱动规定，但是不能超过设备所能处理的长度上限。该长度上限存在于设备的设置头中，可以被驱动读取和确认。当驱动准备好想要设置的虚拟队列长度之后，可以通过向设备的设置头中的一个寄存器中先写入想要设置的队列序号，此时设备将会知道现在要处理的是该虚拟队列，随后在另一个寄存器中写入队列长度来通知设备该队列的长度是多少。通过切换队列序号，可以为多个队列用此种方法设置长度。比如网卡设备就可以拥有多个虚拟队列，最少需要两个，分别是接受数据包所用的和发送数据包所用的。

3.1.2.2.1 内存布局

由于 VirtIO 设备规定虚拟队列内的三个结构必须按照特定的偏移布局放置在内存中，因此我们不能直接使用语言和系统自带的内存分配方法。具体来说，VirtIO 要求每个虚拟队列都被分配在一段连续内存物理页上。

对于每个虚拟队列，其中的描述符表将要被放置在该段内存的起始处。描述符表的总长度计算公式为 $\text{len}(Desc\ Table) = 16 \cdot Queue\ Size$，单位是字节。其中每个描述符长度为 16 个字节，分别由一个 8 字节的地址，4 字节的长度，2 字节的标识和 2 字节的下一个描述符索引构成。描述符表中所有的描述符顺序放置，并不再有其他任何成员。而描述符的个数也就是虚拟队列的长度。因此描述符表的长度就是虚拟队列的长度乘以单个描述符的大小。

在描述符表之后，紧挨着放置可用环。可用环的长度计算公式是 $\text{len}(Avail\ Ring) = 2 \cdot Queue\ Size + 6$，单位也是字节。可用环由三个 16 位无符号整形和一个长度等于队列长度的 16 位无符号整形数组构成。其中三个单独的位置所存放的分别是标识、驱动下一个将要向可用环中写入的下标序号和设备已经承认的驱动发出的请求的下标序号。

在这两个结构都于内存中放置完成之后，将他们一起按照内存页对齐。对齐后的下一个结构的开始地址就成了一个新完整内存页的起始处。通常内存页的大小是4096字节，但是这并不绝对，依赖于交互方式（Transport）。通过向设备设置头中的一个寄存器中写入队列对齐大小，驱动可以通知设备系统所采用的对齐页大小是多少。在这新的一页开头处放置已用环。已用环的大小与可用环的大小计算方法类似，公式为$\text{len}(Used\ Ring) = 8 \cdot Queue\ Size + 6$，单位也是字节。其中单独的6个字节所存放数据与可用环相似，区别仅在于已用环中的数据全部是描述设备已经往其中写入的设备完成的请求，而不是驱动发起的设备需要完成的请求。队列长度之前的系数从2变成了8，这是因为在已用环的数组中，储存的数据长度从2字节变为了8字节。

因此，一个虚拟队列所需要的空间至少为2个地址连续的物理页，准确的大小计算公式为：

$$\begin{aligned} \text{len}(Virt\ Queue) &= \underset{\text{page size}}{\text{align}}\ (\text{len}(Desc\ Table) + \text{len}(Avail\ Ring)) + \text{len}(Used\ Ring) \\ &= \underset{\text{page size}}{\text{align}}\ (18 \cdot Queue\ Size + 6) + 8 \cdot Queue\ Size + 6 \end{aligned} \tag{3-1}$$

直接操控内存地址，在特定的位置构建出结构是不安全的。因此，这部分也被移动到了系统TCB中完成。具体的方法如代码3-1所示，系统将会为驱动提供两个接口：（1）第一个接口允许驱动要求系统按照虚拟队列的内存布局为其分配若干个物理地址连续的内存页，并从该处内存上创建出虚拟队列的结构来。驱动需要自行计算出三个结构分别所在的偏移量，并通过QueueLayout传递给系统。接口会返回一个QueuePage<SIZE>结构。通过该结构我们可以直接获取虚拟队列的可变引用，以直接访问该地址处的内存；（2）第二个接口是实现了QueuePage trait的结构体需要实现的一个方法，它允许驱动获得最终被使用的虚拟队列结构。

```
pub trait QueuePage<const SIZE: usize>: DevicePage {
  fn queue_ref_mut(&mut self, layout: &QueueLayout) → QueueMutRef<SIZE>;
}
fn dma_alloc(pages: usize) → Box<dyn QueuePage<SIZE>>;
```

代码3-1　虚拟队列接口

这种方式虽然实现了我们的需求，但是由于它把虚拟队列内部的结构暴露给了使用者，要求使用者进行配合，因此并非完美。我们还提出了另一种方案：对于虚拟队列采用与寄存器映射相同的读写方式，只需提供对特定偏移处的读写功能即可。这

种方案需要更多的工作，并且需要操作系统提供的接口数量更多，但是不用把虚拟队列的内部结构暴露给使用者。目前我们没有选择实现这种方案。

如果把泛型常数用于分配空间的计算中，则可以做到既不向系统暴露具体的虚拟队列内部结构细节，又能够把需要系统提供的接口控制在最小的范围内。但是非常令人遗憾的是 Rust 语言目前并不支持泛型常数参与运算。如果之后的更新能够为 Rust 语言带来这个新特性，那么此处对于虚拟队列空间申请的接口将可以以更优雅的方式实现。

3.1.2.2.2 描述符表

在虚拟队列被构建出来之后，驱动程序还需要向设置头中写入虚拟队列的相关信息，包括虚拟队列的长度，所启用的特性与所处的物理地址。这是虚拟队列初始化的过程，同时也是对设备初始化中的一环。由于设备与内存设备之间的通信并不通过操作系统，而是直接经过 DMA 总线发送请求，因此操作系统所准备的虚拟地址空间和页表也全部失效。向设备写入的队列地址和请求地址全部都需要是物理地址。鉴于此操作系统还需要为驱动提供一个功能，即把一个虚拟地址转换为物理地址，并允许设备对该地址的访问。

初始化完成之后，驱动就可以使用虚拟队列向设备发送请求，同时设备也可以使用该队列响应请求了。VirtIO 设备的事件逻辑是所有请求都由驱动主动发起，随后由设备响应。因此一次完整的请求流程是：驱动首先在内存中分配出一段空间，向其中填入各个设备所独有的各不相同的请求结构。随后在描述符表中获取一个空闲的描述符，把新分配的该描述符的物理地址通过系统提供的接口转换虚拟地址得到并填入，再写入该描述符的长度。如果驱动想要设备进行的操作需要额外的空间，则还需要更多的描述符参与该次请求。这种情况下所有参与的描述符会形成一个链表结构。如果一个描述符的下一个节点标识被设置为 1，则描述符中指向下一个描述符在描述符表中下标的变量将有意义，设备会依据此不断找到之后的所分配的内存空间。由于分配内存是操作系统层面的工作，设备无法直接分配内存，为了保护内存，设备只会修改在描述符中指向的确定长度的内存。因此，在发出请求分配内存时，就需要事先将设备将会返回给驱动的响应所存放的空间也提前准备好，并放入某个描述符链上的节点所指向的空间处。最后，对于需要返回一个操作状态的设备来说，还需要一块空间用于存放设备对操作的完成情况，它一般作为描述符表链中的最后一个元素出现。

3.1.2.2.3 可用环

可用环是只允许驱动写入、设备读取的数据结构。它的功能是为驱动提供通知设备每个请求的描述符链的起始节点位于描述符表中下标位置的手段。借由此结构，驱动为设备指出它所需要完成的请求存放在哪里。而设备则可以从其中读取到所需的信息。

当驱动收到一个请求时，它首先构造好描述符链，随后把链头节点所在描述符表中的下标写入可用环中的下一个可用位置，并根据特性协商情况判断是否发送中断通知设备。

3.1.2.2.4 已用环

已用环的作用与可用环恰好相反。它只允许设备写入，驱动读取。设备通过这个数据结构通知驱动它所完成的请求的描述符链的起始节点位于描述符表中的哪里。

当设备完成一个驱动通过已用环通知它的请求后，它就会把所完成的那个请求的描述符链链头节点下标写入已用环中的下一个可用位置，并根据特性协商情况判断是否发送中断通知驱动。

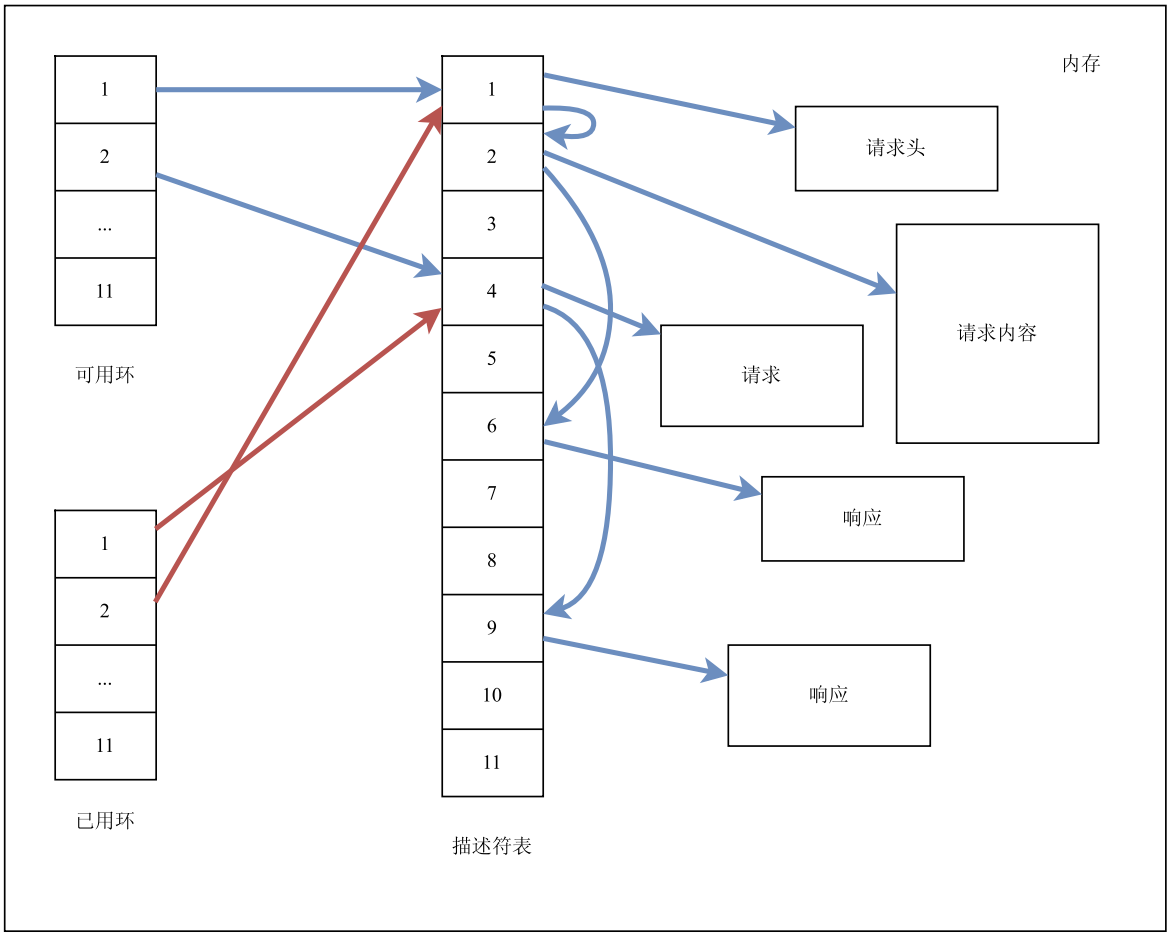

图 3-1　虚拟队列结构

如图 3-1，该图展示了一个可能出现的场景：一个块设备驱动向设备发出了两个请求，分别是：（1）对某一块的读取，因此共需要三个描述符，中间的请求内容是一块用于给设备写入读取结果的缓冲区。该请求的描述符链是1 → 2 → 6。（2）对某一块的擦除操作，因此共需要两个描述符。描述符链是4 → 9。图中蓝色箭头是驱动所需要进行的联系（赋值）操作。随后设备先完成了请求（2），并把该请求的链头下标 4 放入已用环中的第一个位置，随后完成了请求（1）并放入已用环的第二个位置。驱动会把两个请求的执行结果分别相对写入描述符表中 9 和 4 位置所指向的响应内存处。请求和响应的格式都在 VirtIO 的文档中有所规定。如果特性没有协商，设备默认不保证按请求给出的时间顺序进行响应。因此这两个任务的完成顺序可能与给出顺序相反。红色箭头是驱动需要进行的联系（赋值）操作。

3.1.3 块设备

块设备是一种用于持久储存数据的 IO 设备，包括硬盘、SD 卡、U 盘等。这种设备提供的最小读写单位是块，因此被称为块设备。对块设备可以进行的操作共有 8 种。每种操作都需要在描述符所指向的地址处放置一个 Request 结构。其中描述了操作的类型、目标块标号和数据存放缓冲区。注意这个结构可以被分开放置于多个描述符指向的地址中。在实际使用中，我们把操作的元数据（操作类型、块标号）放在首个描述符指向的区域中，并把读写操作所需的缓冲区放置在第二个描述符指向的区域中。设备在完成操作后会向驱动返回一个状态值，它将被存放在缓冲区之后的第三个（非读写操作则是第二个）描述符指向的区域中。

块设备的特点是对它的操作是由系统发出、设备接受的。这与 VirtIO 所设计的操作通信模式一致，因此较容易实现。

我们的驱动提供了阻塞式和非阻塞式的接口，使用者可以任选一种使用。非阻塞式是基础的操作方式，也是 VirtIO 设备运行的模式。驱动在发出操作请求之后，可以不时进行查询操作是否完成。如果完成，则接受操作的结果。此外，如果中断特性被协商开启，那么设备在完成操作之后也会发出一个中断来通知系统。

3.1.4 输入设备

输入设备是用于接受用户外部输入的设备，包括鼠标、键盘、触摸板等。输入设备的操作类型非常简单，只有一种，就是驱动从设备处接受一个输入事件。这个事件将会含有一些属性，如输入类型（触发了哪个按键），鼠标移动的距离，等。通信方

式是驱动向设备发出一个请求，而当输入设备接受到一个新的用户输入时，就把输入写入到请求中的储存区域，然后返回给驱动。

输入设备与之后的网卡，这两个设备与块设备之间有一点不同，就是它们需要处理的事件会包含从设备处发出而系统接受的类型。这种类型与 VirtIO 设备的操作通信模式相异，因此需要一些额外的处理。我们选择的方式是，从驱动初始化的时候开始，就保持可用环（即驱动向设备发出的请求）始终为满。因此，每当设备接受到一个输入时，总是有可用的请求用于存放输入并返回给驱动（可用请求总数为队列大小）。而系统会定期轮询驱动查询有无新的输入。此时驱动会把一个设备已经完成并返回的操作接受，取出其中的输入传给系统，并继续向可用环中插入新的请求。

3.1.5 网卡

网卡是用于通过网络通信（即发送与接受数据包）的设备。驱动在这个过程中要做的事情主要有两件，即收和发数据包。在发送数据包时，驱动从系统处获取想要发送的一个数据包，在其前方加入 VirtIO 网络包的 header，并将其传递给设备处理。在接受数据包时，驱动从设备处获取一个收到的数据包，剥离其开头的 VirtIO 网络包 header，并把内部的内容返回给系统。

网卡与输入设备虽然都有与 VirtIO 操作通信模式相异的事件，但它们之间还有不同之处，这是因为输入设备需要接受的是可以直接进行逐字节拷贝（即实现了 `Copy trait`）的基本类型数据，其大小极小（仅 64 位）使得拷贝过程不会影响性能。但是网卡所接受的内容是一整个数据包，其大小可能达到上千字节。如果逐字节拷贝这个数据，那么对性能的影响将会是不可接受的。因此可行的方案是直接保持数据在内存中的存放位置不懂，仅传递指针。在一些情况下，使用者可能想要自行决定网络收发的数据包需要放在什么地方（如 Alien 中数据包需要跨域进行传递，因此需要被放置在共享堆上）。另外的情况下使用者可能想要开箱即用而不关心数据包实际上存放在了哪里。因此我们提供了两种不同的驱动：`VirtIONet` 和 `VirtIONetRaw`。从名称上可以看出，后者允许使用者自行为数据包收发分配内存空间，并只需向驱动提供数据包的地址即可；前者则在基础驱动之上进行了更多的封装，提供了直接传入数据包进行发送，接受数据包的功能。当然这会带来一定的性能损失，对性能有追求的使用者应该使用基础的驱动。

3.2 Uart16550 驱动

这是一个物理存在的串口设备。与 VirtIO 系列的设备不同。因此它的驱动也和 VirtIO 设备的驱动分开成了两份代码。串口设备用于在多个设备间进行字节粒度的数据传输。它的通信模式也更加简单：往其某个寄存器中写入数据即是发送，而从某个寄存器中读取数据即是接受。

与 VirtIO 设备相同，对寄存器的操作也被归纳成了一个 `trait`，提供按字节对一段内存进行读写的能力。该段内存被映射到设备寄存器上。

3.3 在 Alien 中的适配

以上的驱动为满足通用性，都完全仅使用 Rust 库中所定义的结构写成，只依赖了一些在官方仓库[]中存在的包，而没有依赖任何 Alien 中的结构。因此保证了该驱动的通用性，能够被多个系统代码级的复用，而不仅限于在 Alien 中使用。为了在 Alien 中以隔离域的形式加入这些驱动，还需要对其进行包装，使其满足隔离域的约束，能够于 Alien 系统进行交互。

为了在 Alien 中新加入一个隔离域来实现驱动功能，首先需要在 Alien 中新建一个结构，作为系统的驱动域实体。该结构将会被隔离域传递给系统，系统视此结构为一个驱动程序，而它实际做的工作是在驱动的 Rust 标准结构和 Alien 中特有的结构之间提供转化层。该结构将会引用并创建一个经过安全化的标准驱动作为静态对象。系统对设备的操作通过这个结构中介，转发到静态全局变量的驱动中。随后中介结构处理驱动传递回来的结果并返回给系统。当一个 Alien 特有的结构（如 `RRef<T>`）穿越域边界来到中介结构时，它干的第一件工作是取得该参数内部的实际数据，并把拆装后的实际参数传递给自身所拥有的静态实际驱动。驱动返回的结果通常是直接的返回值或者伴随有驱动内定义的错误类型。代理结构把驱动的错误类型映射为 Alien 系统内的错误类型，如果是存在于共享堆上的数据指针，则还需要用共享堆封装结构 `RRef<T>`将其保存，随后用隔离域必须要的返回值类型 `RpcResult<T>`将其包装，最后传递回系统。这个过程提供了安全驱动和 Alien 之间的对接。

与驱动直接使用 Alien 中的结构进行编写相比，它的缺点是多了一层函数调用，这会带来额外的开销。但是它的好处是可以将驱动程序与 Alien 完全独立，提高了驱动的通用性。这被认为是非常有意义的工作，可以参与进社区的相互参考、学习、改进过程中，最终推动更优秀的成果出现。因此该方案在驱动与 Alien 的整合形式的取舍中胜出。同时，实际使用时的情形也表明，由于对驱动的操作性能瓶颈通常不在函

数的调用时间，增加的开销几乎无法感知。该方案可以作为很好的驱动与操作系统整合的示例。类似的方案可以被用于多个操作系统中。

第 4 章 测试

4.1 功能测试

对 VirtIO 驱动程序功能的测试分别使用 rCore 所提供的 riscv 架构测试程序[]的改进版本[]和 Alien 中进行测试。测试代码已在链接中给出。对于 Uart16550 设备驱动的测试仅在 Alien 中进行测试。

对于不同设备的功能测试方式分别如下：

- 块设备：对于设备的每一个块，使用一个函数计算出一个对应的均匀分布的数字，对该块的所有字节写入该数字。随后使用一个初始化为 0 的内存空间缓冲区，向其中读取目标块内的数据。如果所读取到的数据与写入的数据完全相同，则说明该块设备驱动的读写操作功能是正确的。
- 网卡：对于一个被初始化的网卡，它将不断侦听是否收到了包。如果有新收到的包，就会像系统发送中断标志。因此，在 tcp 网络协议栈未启用的情况下，使用 `ping` 命令向 qemu 模拟器所映射到虚拟机内网卡的端口发送查询命令，开启日志输出并编译后，在 Alien 虚拟机系统中可以观察到收到网络包带来的中断。输出收到的包内容，为所发出的 ping 请求包。在 tcp 网络协议栈启用的情况下，使用 `netcat / nc` 命令对其进行 tcp 连接的测试，能够看到 nc 输出 tcp 连接成功建立的消息。使用 `netcat` 发送测试数据`"hello"`，并在系统中将收到的数据逆序输出，如果观测到虚拟机输出`"olleh"`，则可以证明驱动程序对网络包的接受和发送功能正确。
- 输入设备：由于硬件条件限制，对于输入设备只测试了鼠标和键盘的设置。通过在 qemu 的运行指令里启用鼠标和键盘的 VirtIO 设备，可以把宿主机的鼠标和键盘输出传入到虚拟机的设备中。使用同一个驱动程序对此两设备进行测试。在初始化之后，将系统焦点聚集于虚拟机的窗口中，随后进行键盘按键、长按或者鼠标点击、移动、滚轮操作，并启用日志输出。对于所有的输入设备操作均能看到在测试程序中输出了接收到的输入事件，说明输入设备驱动的功能正确。
- **Uart16550** 串口设备：串口设备被用于在 Alien 中与终端进行通信。在没有启用或初始化串口设备时，虚拟机与用户的交互直接经过 qemu 完成。在使用串口之后，系统向串口输出的内容和会在终端呈现，串口收到的内容会被系统检

测到。在初始化串口驱动后使用其输出函数，向终端输出"uart test"，结果正确的在终端显示，说明串口驱动正确。

测试的结果，所开发的安全驱动，可以正常操作设备，完成所需的功能。

4.2 性能测试

通常认为使用底层的 C 语言的一个优势是其效率更高。而 Rust 的零成本抽象为其带来了与 C/C++类似的性能。使用 Rust 仅比 C++程序慢 4-8%[]，而不安全的 C 风格 Rust 速度则与 C/C++程序相差无几[]。通过对有较高性能需求的块设备驱动的速度测试，可以展现出使用安全代码编写的设备驱动与不安全代码相比性能变化如何。

以下的测试所使用的环境是： 中央处理器：13th Gen Intel(R) Core(TM) i9-13900HX 操作系统：Windows Subsystem for Linux (WSL) 2 Ubuntu 20.04 虚拟机软件：qemu 7.0.0

4.2.1 块设备

对块设备的测试采用顺序读写的方式进行。使用的设备大小为 20MB。该设备是一个由 dd 命令所创建的文件，每次写入 1MB 的源于/dev/zero 的数据，总共执行数量为 20 次。随后可以在 qemu 的运行命令中将该 20MB 的文件作为一个存储设备传递给虚拟机中。测试中分别使用本项目的安全 Rust 实现的驱动和用于对比的使用了不安全代码的设备驱动对其进行总共 200 次全盘读写，并记录每次的时间和速度。最后以此计算出两种不同驱动的平均速度和方差。写入时所使用的数据全部为 0xFF，而对于读取时的数据未进行任何初始化。200 次全盘读写会分为 20 次进行，每次会读写十倍于整个设备的数据量。这些操作按照块编号由低到高的顺序被执行，并在超出最大空间之后回到 0 号块继续进行循环。每次读写在开始前后都会读取 riscv 架构中用于计时的寄存器，将之相减后能够获知该读写操作总共所花费的时钟周期。由于 qemu 中 riscv 机器的 cpu 频率已知，因此通过计算可以获知该计时时长对应多少现实时间。块的大小通过驱动的查询容量功能获取，而非硬编码在测试程序中，可以应对不同大小的设备的测试需要。计时的区域不包括查询块设备大小所耗费的时间。但是会计入循环语句的时间与对驱动程序提供的功能进行函数调用的时间。最后通过计算每次总读写的数据量与耗时之商，可以获知该次操作的实际吞吐量。驱动所提供的查询容量所返回的单位是块。而一块在 VirtIO 块设备的规定中大小为 512 字节，因此该返回值与一块的存储大小相乘就是整个设备的容量。为了保证测试结果的精确性，在进行

测试时，不对测试机器进行任何其他操作以避免有程序与qemu抢占CPU从而导致测试结果不准确。为提高计算精度，所有数值都被转化为64位浮点数进行运算。每次单次测试出来的结果将会被存入一个数组中，此操作不会被计入所耗费时间。在所有次测试都结束后，对于所获得的每次速度数组，可以进行求平均与求方差的操作，以此减小测例结果中的误差，增大可信度。方差被用于说明设备驱动的运行速度稳定性。

测试所得的原始数据可以在此查看，测试所用代码分别在 github 代码仓库下的 `svdrivers/qemu` 和 `virtio-drivers/example/riscv` 文件夹中。测试的结果如下表：

表 4-1

操作类型	安全代码		不安全代码	
	平均速度	方差	平均速度	方差
读	19.86MB/s	0.027	19.87MB/s	0.086
写	18.04MB/s	0.129	17.40MB/s	0.038

从如上数据中可以看出，使用安全代码实现的VirtIO块设备驱动相比于不安全的设备驱动，在顺序读取操作上速度相差无几，而在顺序写入时，其速度甚至比不安全的实现快了 3.6%。

写操作普遍比读操作缓慢，但是差距并不大，只有约2MB/s的差距。读和写操作都接近一个上限，即 20MB/s。我们认为这个速度是在虚拟环境中使用单处理器对块设备进行读写的速度上限。安全驱动和不安全驱动均接近这个上限，表明在现有应用场景中驱动的性能并没有成为操作的瓶颈。

这些数据说明了在块设备驱动这方面，使用不安全代码并没有显著的速度优势。因此可以放心的使用安全的设备驱动，而不需要担心其可能对性能造成的影响。

第 5 章 总结

本文详细讲述了如何完全使用安全的 Rust 语言开发一个设备驱动，提供了可直接使用的 Rust 包，并将其作为一个隔离域加入到 Alien 中，且对这种方法实现的驱动进行了性能测试，展示其取代现有不安全驱动的可能性。此工作既是对于 Alien 系统组成部分的开发，也是使用语言机制强化系统安全的又一个实践尝试，相信这会使基于语言机制的操作系统安全领域的研究走得更远，为其他的研究者和开发者起到参考作用。

参考文献

[1] TRENT JAEGER. The Evolution of Secure Operating Systems[EB/OL]. https://www.cse.psu.edu/~trj1/cse543-f18/slides/cse543-secure-os.pdf.