清 华 大 学

综 合 论 文 训 练

题目：zCore 操作系统内核的设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：潘庆霖

指导教师：陈渝副教授

2020 年 6 月 15 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定**)**

签　名：潘庆霖　导师签名：陈涛　日　期：2020年6月15日

中文摘要

本文介绍了 zCore 操作系统内核的设计与开发过程。zCore 内核是参照 Zircon 内核的系统调用功能规范，使用 Rust 进行重新设计与实现的一个微内核，并从设计上尽可能能发挥 Rust 语言的良好特性。

Rust 语言是一门零抽象开销、内存安全、并发安全的系统级编程语言，强大的编译期检查机制帮助开发者写出更安全可靠的软件，本文由 zCore 的开发过程出发，对 Rust 在操作系统内核中的优越性与不足做出进一步的探讨。

Rust 中的 async 语法机制是 Rust 官方支持的、性能优越的异步编程机制，在用户态程序开发中，async 语法的使用日渐成熟。如今，async 机制在裸机环境下的使用兼具挑战与收益，在内核中采用 async 语法更是一个全新的尝试方向。zCore 采用一体化 async 设计，希望能够挖掘 async 机制在操作系统内核开发中的潜力，进行积极的探索。

在经历了几个月的迭代开发之后，zCore 操作系统内核已经能够支撑像 shell 这样的用户态程序完整运行，并且在性能上缩小了与 Zircon 内核的差距。本文对 zCore 进行如下方面的探讨：

- zCore 项目整体介绍
- zCore 中的 async 机制设计
- zCore 中的 Unsafe Rust 分布与分析
- zCore 与 Zircon 的简单性能比对

关键词：Rust；Kernel；Zircon；Async 协程

ABSTRACT

This paper describes the design and development of the zCore operating system kernel. zCore is a microkernel that uses Rust to redesign and implement Zircon kernel according to its system call specification, and exerts the best features of the Rust language as far as possible.

Rust language is a system-level programming language with zero-cost abstraction, memory safety, and concurrent safety. The powerful compile-time checking mechanism helps developers write safer and more reliable software. The advantages and disadvantages of Rust in development of kernel will be further discussed in this paper.

The `async` syntax mechanism is an asynchronous programming mechanism officially supported by Rust with superior performance. In the development of user-mode programs, the use of async syntax is gradually maturing. Nowadays, the use of async mechanism in the bare metal environment has both challenges and benefits, and the use of async syntax in the kernel is a new direction to explore. zCore adopts the integrated async design, hoping to tap the potential of the async mechanism in the development of the operating system kernel and actively explore it.

After several months of iterative development, the zCore operating system kernel has been able to support user-mode programs such as shell, and has narrowed the gap with the Zircon kernel in performance. This article discusses the following aspects of zCore:

- zCore project overall introduction
- Design of async mechanism in zCore
- Distribution and analysis of unsafe Rust code in zCore
- Simple performance comparison between zCore and Zircon

Keywords: Rust; Kernel; Zircon; Async Coroutine

目　录

第 1 章　背景与相关工作 1
1.1　Zircon 设计概况 1
1.1.1　内核对象 1
1.1.2　软件执行环境 2
1.1.3　隔离性设计 3
1.2　Rust 简介 3
1.2.1　所有权机制 3
1.2.2　线程安全 4
1.2.3　unsafe 语法 4
1.3　Rust Based OS 4
1.3.1　Redox 4
1.3.2　Tock OS 5
1.3.3　rCore 5
1.3.4　Biscuit 6
1.4　Rust 中的 async 协程 6
第 2 章　zCore 整体架构 7
2.1　架构设计 7
2.2　内核组件 9
2.3　实现手段 10
第 3 章　设计与实现 11
3.1　Async Rust 简介 12
3.1.1　Future 与协作式调度 12
3.1.2　async 与 await 12
3.1.3　用户态 Async 使用 12
3.2　zCore 中的 async 使用 13
3.3　Executor 概述 14
3.4　层次化 Future 对象说明 15

3.4.1 线程概况与顶层 Future 15
3.4.2 中层 Future 17
3.4.3 底层 Future 19
3.5 zCore 的用户态运行支持 23
3.5.1 修改 VDSO 23
3.5.2 调整地址空间范围 23
3.5.3 HAL 层接口设计 24
第 4 章 zCore 测试与分析 27
4.1 系统调用实现情况 27
4.2 zCore 中的 Unsafe Rust 28
4.2.1 Unsafe Rust 在 zCore 中的分布 29
4.2.2 Unsafe Rust 在 zCore 中的安全性 30
4.3 基于 microbenchmarks 的性能测试与调优 33
4.3.1 硬件环境 33
4.3.2 测试结果及分析 33
4.3.3 优化过程记录 35
第 5 章 项目总结 37
插图索引 38
表格索引 39
参考文献 40
致 谢 41
声 明 42
附录 A 外文资料的书面翻译 43
附录 B 性能对比测试结果 56

主要符号对照表

OS	操作系统
usize	Rust 中的基本数据类型之一，宽度与地址总线宽度一致的非负整数类型。
API	应用程序编程接口

第 1 章　背景与相关工作

Fuchsia 是 Google 近几年重新开发的一款操作系统，从内核到应用程序的移植都由 Google 从头开始进行，最大程度摆脱了对 Linux 的内核的支持，也因此备受外界瞩目。Zircon 是为 Fuchsia 操作系统量身打造的微内核，整体采用 C++编写，代码规模约在 10 万行，目前能够同时支持 x64 及 ARM64 两个硬件架构，支持包括 Intel NUC 和 Google Chromebook、Hikey960 开发板等硬件的正确驱动。根据目前 Google 对外开源的情况以及 zCore 项目组进行的调研来看，Zircon 是一个全新的微内核，从系统调用的设计、系统调用层的 VDSO 实现，到相关的用户态驱动机制以及不同硬件架构支持等方面来看，都具有很好的学习借鉴意义。

本文将详细介绍 zCore 内核的设计与开发过程，并对其与 zircon 内核的比较测试进行分析。zCore 内核采用 Rust 作为主要开发语言，以少量汇编代码辅助最底层的裸内存操作/直接寄存器操作，参考 Zircon 内核的系统调用设计文档与系统调用层代码，在内核态对 Zircon 中的内核对象进行重新实现，最终达到支撑 Fuchsia 用户态程序运行的目的。

Rust 是一门新兴的系统级编程语言，关注内存安全、类型安全以及并发安全。在性能接近 C 的情况下，Rust 通过特定的语法设计配合编译期检查，在禁止使用 Unsafe 代码的情况下避免了内存安全问题，在允许正常使用 Unsafe 的情况下减少了绝大多数内存安全问题。近年来，采用 Rust 语言进行内核开发成为一个新的热点，以 Redox、Tock OS 在内的一系列 Rust 内核的成功开发证明了用 Rust 进行内核开发的可行性与优越性，也为进一步探讨 Rust 在开发内核过程中的不足提供基础。这也是 zCore 设计的初衷。

在本章内容中，我们将对 Zircon 内核、zCore 涉及到的相关概念及已有项目进行简单介绍，作为文章后续的背景知识。

1.1　Zircon 设计概况

1.1.1　内核对象

在 zircon 中，将用户态看到的软硬件资源描述为若干个内核对象，仅允许用户程序通过系统调用对某些内核对象进行修改、传递、新建、销毁等操作。从而达到让用户程序能够在权限允许的情况下调整软硬件资源、进行进程间协作的效

果。基于内核对象这一基本设计理念，为用户态驱动提供了稳定的 ABI 接口，避免了驱动影响内核稳定性和复杂性。同时，对中断、异常这样的特殊资源的封装，让用户态能够更自然地参与到中断和异常的处理中来，父进程对于子进程的控制力度增大，安全性也显著增强。

在整体的内核设计上，Zircon 中的内核对象的设计天然对引用计数有着需要，绝大多数内核对象的析构都发生在句柄数量为 0 时，并且通过句柄对内核对象进行操作的模式，与 Rust 的所有权机制、Rust 核心库自带的 Arc 指针等，都有着很大程度的契合；而这正是我们决定用 Rust 来开发 zCore 的一大出发点。

1.1.2 软件执行环境

在 zircon 中，用户程序的运行环境如图 1.1中所示。每一个用户进程的地址空间中在某个随机位置映射了一个 VDSO，VDSO 中以函数接口对 syscall 指令进行了封装，这部分内存区域对于用户态代码而言是只读的。用户程序可以通过函数调用访问 VDSO 中的接口，进而进行系统调用。通过系统调用进入内核态之后，内核将会进行 pc 位置检查，只允许用户程序在 VDSO 的对应位置进行 syscall 指令的调用。

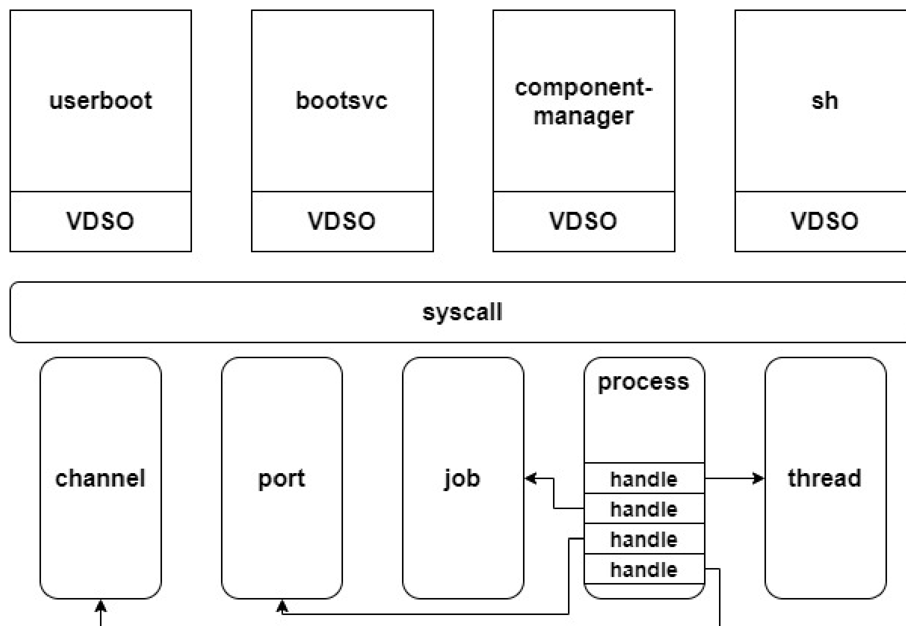


图 1.1　Zircon 用户态程序执行环境

进入到内核态之后，内核将系统调用分发到对应的处理例程中，各个系统调用的具体实现代码被封装到 zircon-syscall 库中。在实现 zircon-syscall 库中的系统

调用逻辑时，将会默认可以获取到指向当前进程的内核对象的句柄。从当前进程所拥有的内核对象句柄、当前进程拥有的权限等方面进行对应的检查。然后在权限允许的情况下，用户程序可以通过系统调用对内核中的某些内核对象进行创建、状态修改、传递、销毁等操作，从而对对应的资源进行调节。

zircon-object 库中，借助 HAL 层提供的接口，实现 zircon 设计中各个内核对象的具体功能，并对外暴露出相应的接口，供系统调用层使用。zircon-object、zircon-syscall 两个库与 zircon 中对应的系统调用层和内核对象的概念基本一致，为比较 C++ 与 Rust 在内核开发中的差异，提供更接近的代码。

1.1.3 隔离性设计

同时，Zircon 中对于隔离性有更彻底的设计，将所有用户态程序能接触到的软硬件资源全部抽象为内核对象，只允许用户态程序在特定权限范围内通过相关句柄进行资源的处理。句柄 + 内核对象的处理机制为 Zircon 的用户程序提供了天然的隔离性支持，官方称之为原生沙盒机制。这一机制对于当前已有的内核中暴露出的安全问题来说，有非常积极的作用。

1.2 Rust 简介

作为一门注重性能的系统级编程语言，Rust 最早被 Graydon Hoare 设计时，计划用于取代 C/C++。该语言最初的设计目标是提供与 C/C++ 同样的性能，但允许更多编译时的安全检查以及少量的运行时检查。为了达成上述目标，Rust 提供了一整套类型系统，在语法设计上引入了所有权机制，并通过编译期检查来保证并发安全。同时，由于过多的语法限制可能对于用 Rust 实现很多底层操作造成阻碍，Rust 使用了 unsafe 语法来帮助开发者更灵活地绕过一些编译期的检查，获得与 C 同样灵活的内存操作能力，同时需要开发者自行承担相应的风险。

1.2.1 所有权机制

在 Rust 中，对每一个变量，规定了该变量的所有权。一个变量可以同时多个位置不可变引用，或在任一时刻被一个位置进行可变引用。要取得一个变量的可变引用，需要当前作用范围能够获取到该变量的所有权。同时，允许对变量进行借用，但借用的引用必须在变量销毁之前归还。以上特性在 Rust 的实际编程中体现，用于帮助配合编译器的检查，从而在合适的位置插入内存回收的代码，最终保证用 Rust 写出来的不使用 unsafe 语法的程序能够做到内存安全。

1.2.2 线程安全

多个线程间的可能存在数据竞争，影响并发安全。这是每一个支持多线程的现代编程语言都需要考虑的问题。在 Rust 中，要求可能被不同线程访问到的变量必须实现 Sync Trait，或者用实现了 Sync Trait 的互斥锁对相关变量进行封装。从而保证在多线程环境下不存在数据竞争，让编译器的检查帮助开发者写出高效、并发性强的程序。

1.2.3 unsafe 语法

unsafe 语法是 Rust 出于对自身限定性强的语法设计和编译期检查的考虑，为开发者提供的另外一种选择。在 Rust 中，如果完全遵循我们前面提到的所有权机制和线程安全的限定，在类似系统内核这样的底层领域，有部分功能将会无法实现。Rust 给出的 unsafe 语法，允许开发者使用 unsafe 关键字划定代码块，在 unsafe 代码块内部，开发者可以绕过部分编译期检查，灵活地使用裸指针操作内存，使用嵌入汇编对代码进行优化、对硬件进行控制，等等。同时，由于 unsafe 块内部的操作灵活程度堪比 C 代码，编译器也无法完全保证 unsafe 内部的代码安全，需要开发者对这部分代码足够小心，并按照开发规范写下 safety 说明。

1.3 Rust Based OS

在 zCore 项目正式启动前，学术界与工业界已经存在许多优秀的 Rust OS 项目，这些项目为 zCore 的设计与开发提供了一定的参考。同时，MIT 开发的 Biscuit 内核[1] 创新性地在开发中使用 Go 语言，这一项目对高级语言开发操作系统内核的利弊进行了深入的探讨，对使用其他像 Rust 这样的高级语言进行内核开发也具有重要的借鉴意义。

1.3.1 Redox

Redox 是一个用 Rust 语言编写的类 UNIX 操作系统。除了用 Rust 开发一个现代的微内核之外，Redox 同时将 Rust 语言的创新特性带入到了一组全系列的用户程序中。在微内核之上，Redox 提供了常见的 Unix 命令，对 Rust 标准库的完整支持，以及用 Rust 开发的 GUI——Orbital。

相比其他采用 Rust 开发的 OS，Redox OS 具有更强的易用性和更完备的用户程序支持，其在微内核设计理念的指导下，将驱动放在用户态的做法，相比 Linux 这样的传统宏内核，更加安全、现代化。到目前为止，Redox 已经基本实现自举，

并在 System76 公司的某些笔记本上可以成功运行起来，正确驱动包括键盘，触摸板在内的常用硬件。

1.3.2 Tock OS

Tock OS 借助 Rust 的类型系统，在硬件隔离机制支持较为简单的低功耗微控制器上，构建系统内核。借助 Rust 的类型系统，Tock OS 在相同的硬件环境下，相比其他用 C 开发的内核，在性能和隔离性方面都有更好的表现。在 Tock OS 的设计[2] 中，对内核构建需要的 Unsafe 代码进行了深入的讨论，采用 Cell 与 Take Cell 的设计方法，满足了内核中常见的保持多个可变引用的需求。

在特定的硬件环境下，结合在内核中维持 Rust 类型系统的特定需要，Tock OS 重新设计了一套简便可靠的系统调用，在微控制器硬件环境下，为用户程序提供了更好的隔离性和开发便捷性。在特殊设计的系统调用支持下，允许用户态使用不同语言进行开发，相比其他面向微控制器的内核，更好地支持用户程序的开发和调试。

1.3.3 rCore

rCore 是使用 Rust 开发的内核的另一个尝试，整体设计采用宏内核，系统调用遵循 POSIX 标准，部分实现逻辑参考了 Linux 中的实现。在 rCore 的开发过程中，更大胆地引入了许多可用易用的第三方库，减少内核人员工作量的同时满足了敏捷开发的需要。在较短时间内实现了对 GCC，Nginx 等程序的支持，这一点同时也展示了 Rust 在内核开发中另一个可能超越 C 的地方，成熟的库机制能够使不同内核中尽可能复用已有的第三方代码，在提高第三方代码可靠性的同时，内核的开发将更便捷高效。

同时，大量引入第三方库也会引入风险，需要内核开发人员对第三方库的代码具有较强的甄别能力。就 Unsafe Rust 的角度来看，引入第三方库要求对第三方库中的 Unsafe 代码足够信任，而信任更多的 Unsafe 代码往往导致了出现内存安全问题和类型安全问题的风险升高。

zCore 的主要开发人员之一同时也是 rCore 项目的发起人与长期维护者，在 zCore 中，从 rCore 继承了部分硬件相关代码，节约了 zCore 开发者在开发前期搭建内核开发环境的时间。而 rCore 中由内核开发者封装成库的大量代码在 zCore 中可以不经修改直接调用，也进一步验证了 rCore 对库的封装与调用，是具有较高的可行性与高效性的。

1.3.4 Biscuit

Biscuit 是由 MIT 开发的宏内核，采用 Go 而非 Rust 作为主要开发语言，搭配少量汇编代码。在 Biscuit 之前也有许多在系统内核中使用高级语言的先例，相比之下，Biscuit 的最大亮点是：在 Kernel 中包含着一个 Go 的运行时。Biscuit 的开发过程中最大程度依赖了 Go 的运行时，完全使用 Go 语言自带的垃圾回收机制来辅助内存管理，使用 Go 的协程机制来实现内核中的线程，尽可能最大化 Go 语言为内核开发带来的便利性。

在应用程序方面，Biscuit 完整支持了 Nginx，并用来与 Linux 这样成熟的 C 语言内核进行比较。结果表明，像 Go 这样的高级语言在开发内核方面带来的性能损失在可接受范围内，而在安全性和方便程度上则有一定优势。Biscuit 作为高级语言开发系统内核的一次尝试与探讨，为其他使用高级语言进行内核开发的探索提供了很好的借鉴。

1.4 Rust 中的 async 协程

协程是 Rust 中对高并发特性的一个有力支持，Rust 通过编译器检查结合运行时内存分配，实现了一套无栈协程机制。在语法上，Rust 给出了 async/await 关键字和 Future Trait 接口，开发者可以通过实现对应的接口，来封装底层 Future 对象；Future 对象可以层层嵌套，通过组合形成更大的 Future 对象，最外层的 Future 对象将会作为协程调度器的基本调度单位进行调度。另一方面，社区的第三方库帮助封装了多数底层 Future，并给出了第三方调度器的实现，开发者也可以直接通过组合不同的 Future，形成自己想要的 Future，交给协程调度器进行调度。由于有编译期检查的支持，Rust 实现的协程是无栈协程，在内存使用上天然具有一定优势。

在本次的 zCore 开发中，我们使用 no_std 情况下的 async 相关语法，来将用户线程封装为内核态中的 async 协程，真实目的是希望能够借助 Rust 强大的编译期检查，将传统的线程用内核态的协程来实现，不仅减少了内存占用，同时也创造更好的内核开发环境，真正在内核的开发中体现 Rust 强大的并发特性。实际应用中，我们除了能够直接借助的 async 语法支持，还需要自行提供不依赖于 std 的协程调度器，这一开销相比 async 语法机制为 zCore 带来的好处来说，完全在可接受的范围内，文章后续将会详细介绍这一设计过程。

第 2 章　zCore 整体架构

本章主要描述 zCore 的整体架构，对 zCore 在开发过程维护的软件架构进行介绍。同时，zCore 是一个复合项目，项目的目标并不只是 zCore 微内核的实现，而是基于 zircon 设计概念的一次重新思考。在 2.1将对具体架构以及本毕设相关的部分进行更详细的介绍。

2.1　架构设计

在实际的架构规划中，zCore 项目在 zircon 的设计基础上进行了更深入的思考。Zircon 的内核对象这一设计思路，引导我们对整个 zCore 项目的构建进行了重新思考。并基于以下几点考虑，来设计整个 zCore 项目的框架。

- 内核对象在 C++ 中实现为类，在 Rust 中实现为结构。这部分代码应当被封装为一个库，使其具有易被复用的特点。
- 内核对象的实现需要操作硬件，但应当可以独立于硬件，这样我们能够更方便地支持 libos 的运行。可以通过设计一套硬件接口，向下封装具体的硬件操作代码，向上提供统一、抽象的 API 接口，这种做法同时有利于我们将直接操作硬件的 Unsafe Rust 代码进行单独封装，方便 Rust Safety 的检查与保证。

从以上两个出发点，我们将 zCore 项目设计为图 2.1所示的架构。下层更靠近硬件，上层更远离硬件。Bare Metal 可以认为是硬件架构上的寄存器等硬件接口。在软件架构的设计中。首先将直接操作硬件的代码进行一层封装，对应模块为 kernel-hal-bare；kernel-hal-bare 对上提供 HAL 层所定义的接口。即 kernel-hal 库仅负责接口定义，具体的实现可以有多种形式，完全基于实际硬件的 kernel-hal-bare 能够让上层 OS 运行在实际硬件上，基于宿主 OS 提供的系统调用进行实现的 kernel-hal-unix 能够让上层 OS 内核运行在 Linux 这样的类 Unix 系统上。

kernel-hal 层将向上提供一套硬件相关 API，这一套 API 给出上层实现所需要的所有操作硬件的接口，上层的库实现所能够看到的硬件环境完全由这套 API 给出。API 之内的实现和所依赖的环境，对于上层实现而言是透明的。

基于 kernel-hal 层提供的虚拟硬件 API，我们能够参照 Zircon 文档进行内核对象的实现，即 zircon-object 层。并依赖于 zircon-object 层所实现的各类内核对

象，实现各个系统调用的处理例程，将所有的处理例程封装为一个系统调用库，即 zircon-syscall。出于同样的设计理念，我们也可以依赖 HAL 层接口来实现 linux-object，使用 linux-object 作为基础对 Linux 的系统调用规范进行实现。或者直接使用 zircon-object 中已经实现好的内核对象，作为实现 Linux 系统调用的基础，来实现服务例程。

以上所描述的 kernel-hal-bare、kernel-hal、zircon-object、zircon-syscall 在具体实现中都被封装为对应的库，被上一层所调用。zircon-loader 层中封装的则是区别于内核对象与系统调用层的、更高级的一层逻辑。zircon-loader 库中，将内核中初始化内核对象、设定系统调用入口、运行首个用户态程序等逻辑进行封装，形成一个库函数。这一部分逻辑将既能够在裸机环境下被 zCore 调用，也能够在模拟执行的 Zircon LibOS 中被从用户态调用。在最顶层，zCore 与 Zircon LibOS 都最终调用了 zircon-loader 库中的初始化逻辑，以相类似的顺序进入第一个用户态程序执行。在 zCore 中，由于内核是由 bootloader 启动，移交控制权给 zircon-loader 之前，仍需要进行部分硬件相关的初始化，以保证最下层的 HAL 部分能够正常工作。

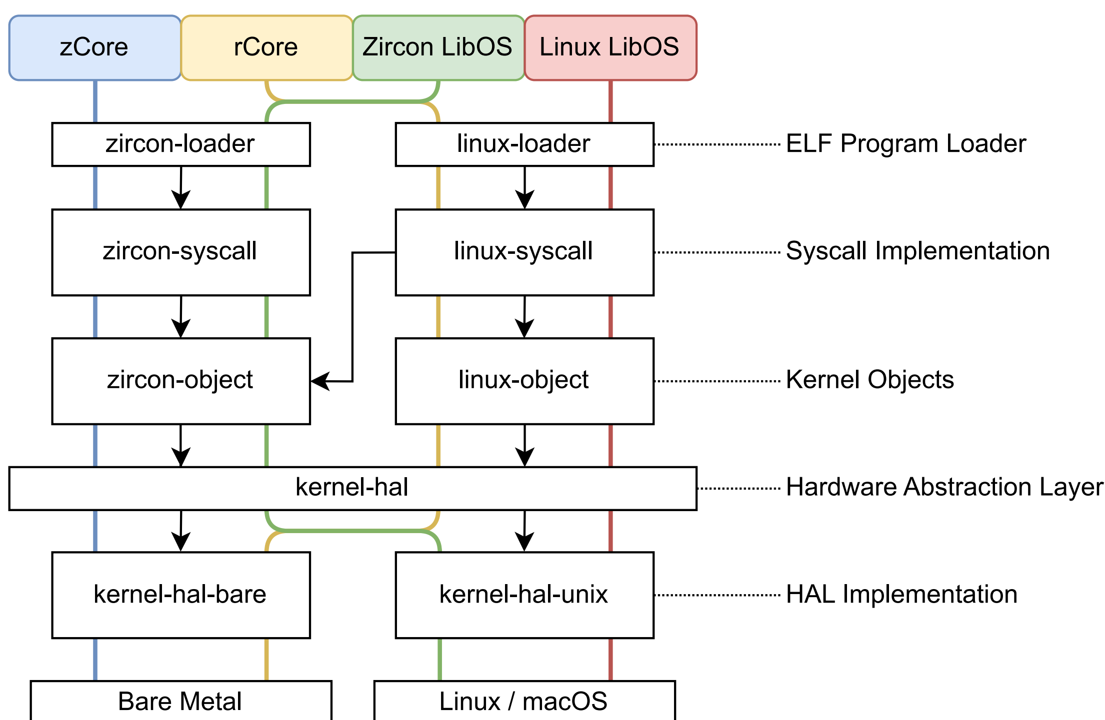


图 2.1　zCore 项目设计图

本毕设的主要工作涉及图中左侧蓝色线部分，即整个 zCore 微内核的设计实

现部分。因此本文只涉及对 zCore 内核的介绍，文章后续部分提到的 zCore 特指能够运行在内核态的微内核 zCore。zCore 内核是一个支持 x86_64 架构，运行在内核态的微内核软件，设计理念上与 zircon 文档所体现的内核对象概念一致。在项目架构中，与其他 libos 版本及 linux 系统调用的相关实现有部分共用代码。

2.2 内核组件

在编译、链接时，zCore 内核中的各个传统组件如堆内存管理器、物理页帧分配器、线程调度器等，会将提供的 API 绑定到 HAL 层声明的各个函数。在实际运行过程中，内核组件的接口通过 HAL 层接口在内核中被调用，并发挥相应的作用。

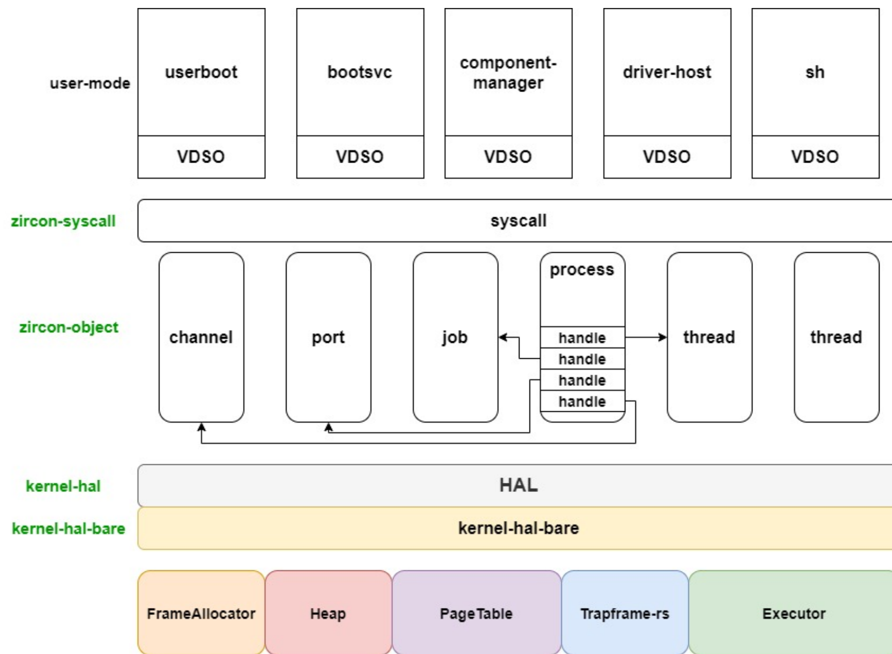


图 2.2　zCore 内核运行概况

图 2.2中给出了当 zCore 正常运行、为用户态程序提供服务时，内核各组成部分的层次。在 zCore 启动过程中，会初始化物理页帧分配器、堆分配器、线程调度器等各个组成部分。并委托 zircon-loader 进行内核对象的初始化创建过程，然后进入用户态的启动过程开始执行。每当用户态触发系统调用进入内核态，系统调用处理例程将会通过已实现的内核对象的功能来对服务请求进行处理；而对应的内核对象的内部实现所需要的各种底层操作，则是通过 HAL 层接口由各个内核组件负责提供。关于图中的 Executor，我们在此简单称为调度器，但由于 zCore

在设计过程中采用了 async 机制，所以实际上的 Executor 与一般意义上的线程调度器有一定区别，该区别将在章节 3.3中进行详细描述。

2.3 实现手段

根据本章开头的描述，zCore 在实际代码层面将代码进行了目录结构的调整，为了便于将具体设计与代码相结合，在此给出更具体的目录结构说明。在图 2.2中，绿字直接对应项目中的具体目录名称。

在目录结构的设计上，充分考虑 Rust 在模块化设计上的优势和链接过程中体现的灵活性，将可以被独立出来的模块作为子目录，共同组成整个 cargo 工程目录。在 HAL 接口层的设计上，我们借助 Rust 的能够指定函数链接过程的特性。在 kernel-hal 中规定了所有可供 zircon-object 库及 zircon-syscall 库调用的虚拟硬件接口，以函数 API 的形式给出，内部均为未实现状态，并设置函数为弱引用链接状态。在 kernel-hal-bare 中给出裸机环境下的硬件接口具体实现，在 zCore 项目编译、链接的过程中将会替换 kernel-hal 中未实现的同名接口，从而达到能够在编译时灵活选择 HAL 层的效果。另外 HAL 层的接口设计文档也可以借助 Rust 的自动生成文档功能，自然地在 kernel-hal 中给出。

第 3 章　设计与实现

Async 机制是由 Rust 官方进行语法支持的异步编程机制，通过协程间的协作式调度提升性能。Rust 的无栈协程不仅能够减少线程切换的时间开销，还能减少多线程带来的内存占用；在占据更少的内存空间的情况下，保证并发性。Rust 的 async 机制在用户态程序的开发中已经日渐成熟，标准库提供的各项机制与来自社区的成熟的第三方库，对于使用 async 开发异步程序的开发者而言都是福音。我们希望能够突破现有 OS 设计的思路，进行在内核中使用 async 语法的尝试，并希望借此获得内核开发的高效与良好性能。

本章中，我们将对 zCore 中的 async 机制实现与 zCore 在用户态运行的支持两方面进行分析，作为对 zCore 设计方案的详细说明。在 zCore 中，创新性地引入了 Rust 的 Async 语法作为内核中的基础设施，在基于 Async 语法机制进行内核设计的过程中，我们对传统内核中的线程实现进行了重新考虑。在本章开头，我们将首先介绍 Async Rust 的相关概念，对 Rust 中的协程支持进行简单介绍。在章节 3.2对于 zCore 中目前使用 Async 语法的情况进行简要介绍，说明我们如何使用内核中的协程来为用户态提供用户线程机制的支持。

在 no_std 情况下，Rust 核心库并不提供协程调度器，我们在 zCore 中使用了自己设计的 naive-executor，在章节 3.3中对这一调度器的运行机制进行说明。

在开发 zCore 的过程中，我们将 zCore 中的 Async 语法使用分为三层 Future 对象：顶层 Future、中层 Future、底层 Future，这三类 Future 对象也将在后续章节被详细介绍。然后，我们将通过若干个简单的例子说明如何在 zCore 中用 Async 语法实现一个阻塞式系统调用。

LibOS 是能够运行在用户态的系统内核，通过使用宿主 OS 上的用户态代码模拟硬件底层的手段，在用户态支撑内核的运行。本质上 LibOS 是一个运行于宿主 OS 上的用户程序，因此相比直接运行于裸机环境的内核而言，具有快速调试、方便定位 bug 的作用。对 zCore 而言，支持 zCore 的 libos 版本在用户态正常运行，能够帮助在用户态对内核组件进行开发和测试，节省直接在裸机环境运行、调试需要的时间。

同时，为了兼容 zCore 的 libos 版本，在 zCore 的设计中对此做出了一定调整，导致我们需要对 Fuchsia 用户程序进行一些微小的修改，这些修改将在 3.5进行详

细说明。

3.1 Async Rust 简介

3.1.1 Future 与协作式调度

为了支持协程，Rust 在核心库中引入了用于协作式调度的 Future Trait。每一个 Future 对外暴露出一个 poll 接口，对 poll 的每一次调用可能返回执行成功或需要阻塞的结果，来将 Future 内部的状态反映给调度器。同时，poll 接口接收代表当前 Future 上下文的 context 参数，该传入的上下文引用可以通过 wake 接口提供对当前 Future 的唤醒操作。

Future 内部本身是可组合的，在用户态 Rust 程序中，一种常见的设计模式是引入标准库中已经封装好的底层 Future，经过多重嵌套和组合，形成自己的 Future，然后传递给调度器进行调度。在编译期，Rust 编译器将会计算每一个 Future 所需要的内存大小，将最顶层的 Future 变成一个带有多个状态的状态机；在运行时，每一次调度器对一个 Future 进行 poll 操作，都将使得该 Future 的状态机发生变化，进入另一个中间状态或者终态。

3.1.2 async 与 await

async 关键字与 await 关键字是 Rust 官方对于协程的语法支持，与 Future 配合使用。async 关键字用于标记某个函数为阻塞调用类型，该 async 函数将不会被立即执行，而是会返回一个 Future 对象。await 关键字则表示异步地等待当前 Future 返回。

在调用 async 标记的函数时，加上.await 关键字，表明该函数的调用可能被阻塞。在编译器看来，一个标记了 async 的函数本质上也是一个将会在运行时被构造的 Future 对象，可以参与其他 Future 的内部组成，也可以由某些 Future 组合来实现内部逻辑。借助 async 关键字与 await 关键字，Rust 开发者可以更自然地实现一个可能在中途多次阻塞的函数调用链，而不需要使用传统的手写回调函数的方法。对于已有非协作式调度实现，借助 async 关键字也可以快速、低成本地转换到协作式调度。

3.1.3 用户态 Async 使用

在实际的用户态程序开发过程中，Rust 官方并不提供 async runtime，只提供 Future Trait 与 async/await 语法，以及与之对应的编译期检查和状态机生成。

图 3.1是一个在用户态使用 async 的例子，tokio 是由社区提供的第三方实现的 async runtime，依赖于操作系统的系统调用来实现线程池。每一个 Executor 可以在一个线程中运行，负责调用 poll 接口，并将 poll 接口返回 Pending 的 Future 对象交给 Reactor 进行管理，Reactor 则负责在 Future 对象被唤醒时重新将其交由 Executor 进行调度。

一个贴近实际的例子是在一个 Future 对象内部进行 TCP 包的发送并等待回应。在发包结束之后，Future 对象内部将因为等待回复包而陷入阻塞，对外返回 Pending。并委托 Reactor 在目标事件发生时重新将自身 Future 对象唤醒，返回 Ready。

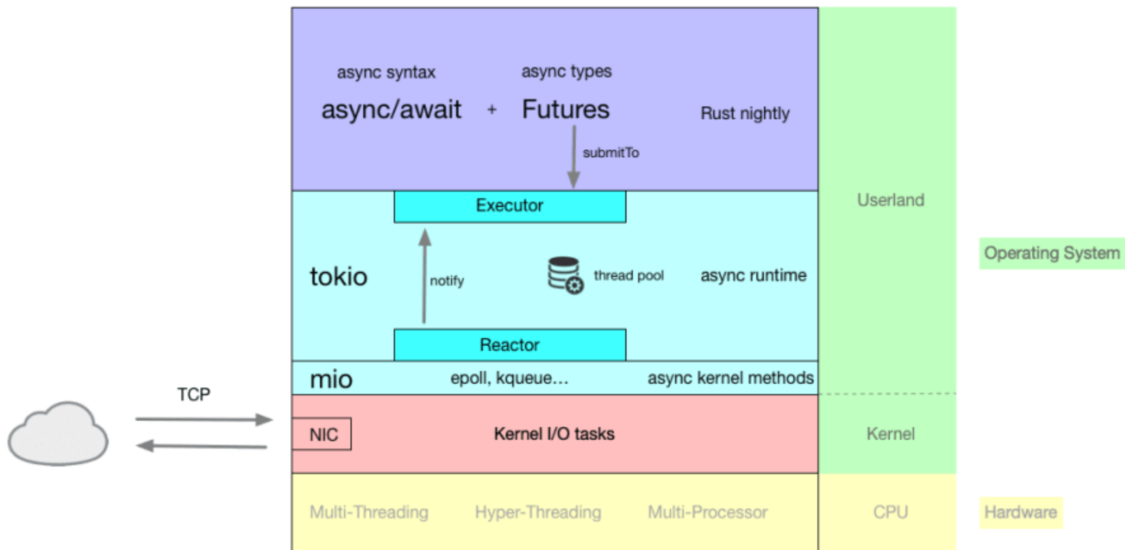


图 3.1 用户态 async 实例[3]

3.2 zCore 中的 async 使用

从 3.1可以看出，async 语法在代码中具有较强的传播性，即采用协程进行协作式调度的 Rust 程序更强调整体的 async 语法使用。在 zCore 的设计中，这一特点也有明显的体现，为了能够在内核中更好地使用 async 语法，并且与内核-用户的切换无缝结合，我们从 zCore 的根本设计上顺应了 async 语法的要求。

在内核态，我们目前无法像图 3.1中的用户程序一样获得 async runtime 的支持，即现有的第三方库无法为我们提供 Executor、Reactor 等基础设施。在 zCore 中，我们自己设计了调度器（naive executor），将每一个内核线程封装在一个顶层 Future 中，由 naive executor 对顶层 Future 进行调度。每一个顶层 Future 由若

干个中层 Future 参与组成，这些中层 Future 内部主要对阻塞式系统调用、分时调度、实时性要求高的系统调用等进行了实现、封装。中层 Future 内部需要进行阻塞的部分，我们采用手写底层 Future 的方式来实现。

从内核开发者的角度来看，每一个用户线程都是一颗向下生长的 Future 树，最顶层节点被传递给调度器 naive-executor，调度过程中对顶层 Future 进行的 poll 实际上是不断尝试对对应的 Future 树进行的深度优先遍历。

3.3 Executor 概述

当我们不考虑切换到用户态的情形时，zCore 可以被认为是一个简单的内核态 async 程序；在 no_std 的情况下，Rust 官方只提供了特定的语法支持和 Future Trait，不提供调度器。我们自行为 zCore 在单核情况下的调度实现了调度器 naive-executor。

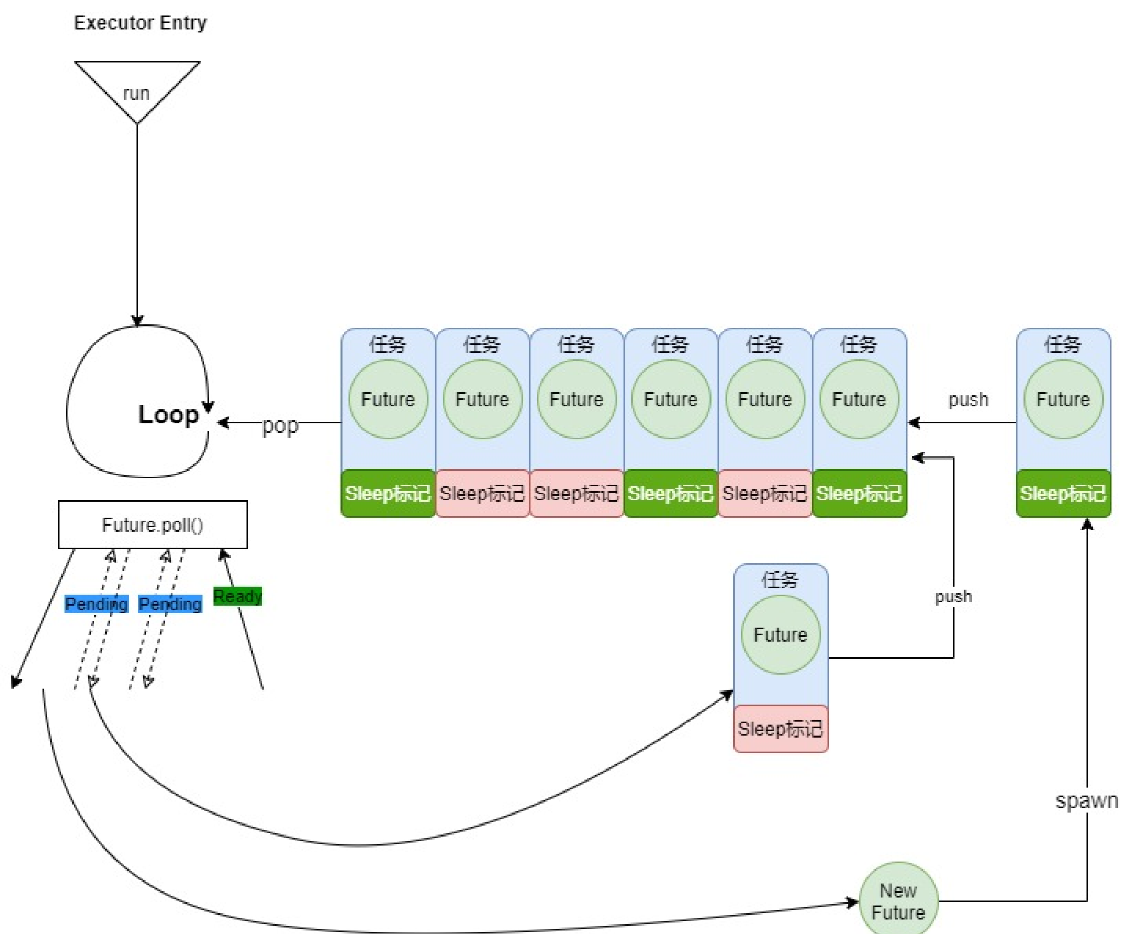


图 3.2 naive-executor 执行简图

naive-executor 采用较为简单的实现逻辑，其中维护了一个任务队列。任务队列中的每一项持有一个 Future，以及一个 sleep 标记。在执行过程中，naive-executor 将会不断从任务队列中取出队头，检查 sleep 标记，如果该任务不在休眠状态，则调用对应 Future 的 poll 接口，将执行权交给这一 Future 内部封装的逻辑进行执行。poll 的结果将会显示该 Future 执行完毕或陷入阻塞状态，对于一次执行后陷入阻塞状态的任务，打上 sleep 标记之后重新放回队尾，等待下次调度。

naive-executor 与传递给 poll 的上下文引用 context 是唯一有权修改 sleep 标记的两个代码区域，对于一个打算返回阻塞的 Future，实现该 Future 的开发者将必须自行考虑如何保存上下文引用 context，何时调用 context 的 wake 接口来取消当前任务的 sleep 标记。在 poll 函数返回前，Future 的实现者必须将 context 中的 waker 对象注册到某个事件的回调函数中，使得当目标事件发生时，能够通过 waker 唤醒当前任务。waker 的 wake 接口由 executor 的实现者给出。在 naive-executor 中，我们简单将其实现为取消对应任务的 sleep 标记。

在 zCore 运行时，将会在 HAL 层相关的硬件初始化结束之后，封装第一个用户线程的顶层 Future，提交给 naive-executor，然后立即将执行权移交给 naive-executor。naive-executor 进行协作式调度的过程，可以看作是 zCore 内核的 IDLE 线程，这也是目前 zCore 使用的唯一一个内核态线程。从 executor 的角度来看，zCore 内核为 Fuchsia 用户程序提供服务的过程，实际上是 naive-executor 不断维护任务队列，进行协作式调度的过程。初始状态任务队列中只有一个任务，即 Fuchsia 的第一个用户程序 Userboot 的首个线程。

3.4 层次化 Future 对象说明

3.4.1 线程概况与顶层 Future

在操作系统中，线程通常代表着一个最小可调度任务，一个线程的具体内容是一个可以被切换的执行上下文。在 zCore 的设计中，用户线程代表一个在用户态执行的任务所需要的上下文、通过系统调用请求服务的途径、出现异常时控制权移交给内核的正确入口。以上的线程实现逻辑，被我们统一封装到顶层 Future 中。

代码块 3.1是 zCore 中现在顶层 Future 内部的主要执行逻辑。其中，由 HAL 层提供的 context_run 接口，是根据不同的硬件架构对内核态/用户态之间的上下文切换的具体实现，对上表现为一个函数的形式。当内核打算将执行权移交给用

户态时，通过调用该函数切换上下文，并在内核栈中保留内核的执行上下文；当用户态发生中断、异常、主动进行系统调用时，进入内核态，从内核的执行角度来看，是 context_run 函数返回。context_run 函数并不涉及 async/await 关键字，但是可以看作是一种仅在顶层 Future 内部调用的、由于进入用户态而阻塞的特殊阻塞函数。调用该函数会将执行权交给用户态代码。

```
loop{
    let mut cx = thread.wait_for_run().await;
    kernel_hal::context_run(&mut cx);
    let mut exit = false;
    match cx.trap_num {
        0x100 => exit = handle_syscall(&thread, &mut cx.
   general).await,
        0x20..=0x3f => {
            kernel_hal::irq_handle(cx.trap_num as u8 - 0x20);
        }
    }
    if exit {
        break;
    }
}
```

Listing 3.1　顶层 Future 内部实现

上下文切换函数返回之后，CPU 的状态为内核态，可以看作是运行到了内核中一个 Future 的中间部分，因此我们可以通过实现带 async 关键字的处理例程，来处理可能需要阻塞的系统调用，或者实现分时调度。例如代码块中的 handle_syscall 接口，是 zCore 中处理系统调用的入口点，在其中进行了系统调用的分发、处理、返回。某些系统调用在处理的过程中需要等待至特定时间或系统中某一类资源可用，在进入内核态之后需要暂时阻塞当前线程；在设计时，我们将该处理例程标记为 async 函数，其中存在.await 的位置可能将会发生阻塞，然后 naive-executor 会将执行权交给其他顶层 Future，即其他线程，从而实现并发。

从 naive-executor 的角度来看，每一个用户线程都是其任务队列中的一个任务，即一个被进一步封装的顶层 Future。从内核态到用户态的来回切换可以很自然地发生在这一顶层 Future 内部，而该切换过程对 naive-executor 是透明的。即对于 naive-executor 而言，一个顶层 Future 可能用于封装用户线程，也可能用于封装内核线程。如果将 context_run 函数内部允许传入的上下文参数代表一个内核态执行的线程，那么内核线程可以与用户线程在顶层 Future 的设计上完全统一起来。在当前情况下，由于我们对内核线程没有需求，暂时没有进行实现。

3.4.2 中层 Future

在顶层 Future 中，当内核处理某些系统调用时，会面临系统调用的实时性需求以及系统调用过程中与其他进程间的交互。在未使用 Async 语法的传统内核设计中，常见的做法可能要求添加更多回调函数或采用轮询的做法。在 zCore 中，借助 Rust 提供的 async 语法支持，我们需要有别于传统做法的实现方法，也由此获得更高的效率。

3.4.2.1 支持系统调用超时的实现

Zircon 本身是一个对实时性支持较为完善的内核，部分 Zircon 系统调用支持传入 deadline 参数，由用户程序规定该系统调用的完成时限，以此促进用户态程序之间、用户程序与内核之间的强实时性交互。对于传入 deadline 参数来规定某一系统调用完成时限的需求。在不使用 async 语法的内核中，一种可能的做法是在系统调用例程中直接与线程调度器交互，并在全局时钟管理队列中设置对应时钟，同时还需要在 deadline 到来之前多次对要操作的内核对象进行轮询，比如对于当前线程等待的资源进行多次轮询。如果希望尽可能减少轮询次数，那么需要添加更多的回调函数，同时将使线程调度器、回调函数队列、时钟队列等不同组件的耦合性进一步增强；对实际的代码维护来说，并不利于低耦合设计和出错时的调试。

在 zCore 中，由于使用了 async 语法，对于这类需求我们可以有更方便的解决办法。Rust 官方支持的 select 宏可以很好地帮助我们实现这一操作。对于 zCore 而言，带有时限要求的阻塞型系统调用，本质上可以认为是带有时限要求的中层 Future 实现需求，更进一步，我们可以同时将两个 Future 绑定在一起。两个 Future 中，一个负责等待 deadline 到来，然后返回结果；另一个则可以专注于阻塞型系统调用的本身逻辑，不需管 deadline。select 宏可以帮我们将两个 Future 进行组合，形成一个新的 Future，当其中一个 Futre 返回时，其结果作为合并 Future 的返回结果返回。

```
/// Run async future and change state while blocking.
pub async fn blocking_run<F, T, FT>(
    &self,
    future: F,
    state: ThreadState,
    deadline: Duration,
) -> ZxResult<T>
where
    F: Future<Output = FT> + Unpin,
    FT: IntoResult<T>,
```

```
{
    let ret = select_biased! {
        ret = future.fuse() => ret.into_result(),
        _ = sleep_until(deadline).fuse() => Err(ZxError::
  TIMED_OUT),
    };
    ret
}
```

Listing 3.2　中层 Future 支持强实时性系统调用

在代码片段 3.2中展示了我们在 zCore 中用于支持系统调用的主要接口。其中 future 是封装了阻塞型系统调用代码的 Future 对象，sleep_until 接口可以用于产生一个底层 Future，该 Future 将阻塞，直到 deadline 时间到来时返回。在 async 语法和 select 宏的支持下，我们使用很自然而高效的语法完成了这个需求，避免了大量的回调函数的使用和复杂的代码逻辑。

3.4.2.2　支持可取消型系统调用

Zircon 对于系统调用的可取消特性这一需求，主要源自于内核对象的基本设计概念，当一个阻塞型系统调用带时限要求地对某一个内核对象进行等待，希望在特定时间内得到该内核对象的某个特定信号时；这一内核对象的所有者可能并不是当前用户进程，因此该内核对象可能在等待时间内由于所在进程退出而被销毁，这时需要等待者有多元化的返回结果。至少包含等待超时、成功等待、操作被取消这三种返回结果。这三种结果的返回互不影响且任意一个结果产生将直接终止其余两个等待过程。本质上，阻塞型系统调用可取消的特性，要求一个异步的协程执行过程可以被打断。

对于这样的需求，select 同样展现了非常好的契合度，我们需要同时等待三类结果，在 zCore 中可以认为是同时等待三个 Future，select 可以自然地将三个 Future 绑定后放入 naive-executor 中，并以最先返回的 Future 对象的结果来作为系统调用返回结果。在这里，我们借助社区已有的 Future 生态中的 oneshot channel 来实现 cancel token。oneshot channel 帮助我们生成一对 sender 与 receiver，并支持对 receiver 进行阻塞式监听，直到 sender 发送消息。我们为每一个内核对象维护一个 sender 队列，对每一个因等待内核对象而阻塞的系统调用，在 select 中多加入一个协程，监控对应的 receiver。当取消事件发生时，拥有内核对象的线程将调用 sender 的发送接口，正监控对应 receiver 的协程将不再阻塞，阻塞型系统调用也将退出并返回对应结果。

在 zCore 中，系统调用的可取消通常与实时性同时出现，在上一小节中我们展示了对系统调用实时性的具体实现，对比两个代码块 3.3与 3.2，不难发现使用 async 语法结合 select 宏对于这种复杂系统调用的功能添加来说，并没有导致系统的耦合度增强，这对于开发的便利和后续代码维护都将有很好的促进。

```
/// Run a cancelable async future and change state while
   blocking.
pub async fn cancelable_blocking_run<F, T, FT>(
    &self,
    future: F,
    state: ThreadState,
    deadline: Duration,
    cancel_token: Receiver<()>,
) -> ZxResult<T>
where
    F: Future<Output = FT> + Unpin,
    FT: IntoResult<T>,
{
    let ret = select_biased! {
        ret = future.fuse() => ret.into_result(),
        _ = sleep_until(deadline).fuse() => Err(ZxError::
   TIMED_OUT),
        _ = cancel_token.fuse() => Err(ZxError::CANCELED),
    };
    ret
}
```

Listing 3.3　中层 Future 支持可取消系统调用

3.4.3　底层 Future

顶层 Future 与中层 Future 的设计与实现，仍旧停留在对更低层的 Future 对象的选择与组装上。但在裸机内核态环境下，我们无法利用社区中已有的 Future 基础设施，仅通过组装来设计一个内核。在使用 async 语法对 zCore 进行一体化设计的过程中，对底层 Future 的设计是一个重要且不可避免的部分。底层 Future 的可靠性与稳定性直接影响所有直接或间接依赖该 Future 对象的所有上层 Future。因此底层 Future 的开发成本和试错成本较高，在开发过程中占据了大量的时间进行排错。

在这一小节中我们对目前 zCore 中实现的几类底层 Future 进行说明，底层 Future 的实现都是通过手动实现 Future Trait，这也是 zCore 中对各个 poll 接口的具体实现过程。

3.4.3.1 Yield

Yield 操作是多数内核都会支持的一个简单调度操作，允许当前线程主动让出执行权限，将 CPU 资源暂时交出，但是不进入睡眠状态，可以在之后继续执行。在常见的内核设计中，yield 是一个容易实现的系统调用类型，一种常见的做法是：在 yield 的系统调用处理函数中调用线程调度器的相应接口，将当前线程移动到调度队列的末端。然后切换到内核 IDLE 线程，由内核重新从调度队列的头部取出任务执行。

在使用 async 语法实现线程支持的 zCore 中，async 语法要求我们对 yield 这一操作提出适合协程的解释。从 Future 对象的角度对 yield 进行思考，如果一个协程希望能够自己让出执行权限，本质上是希望能够在不影响后续执行的情况下，让本次的 poll 结果返回 Pending。即借助 Future 对象与 Executor 天然存在的交互，通过返回一个阻塞的结果，来让调度器暂时放弃继续执行该协程。

在实际的实现中，我们还需要考虑何时唤醒当前线程的问题。对于 yield 操作，当前协程让出 CPU 执行权时，不应影响调度器在下一轮调度中唤醒当前协程。所以在返回阻塞的结果时，同时应该通过上下文引用对当前协程进行唤醒操作。如代码段 3.4所示，当前协程在让出执行权的同时，对自身的上下文引用进行了唤醒操作。根据我们在之前章节 3.3对 naive-executor 的介绍，唤醒操作将任务队列中的任务重新标记为活跃状态，即可以参与下一轮调度。

```
struct YieldFuture {
    flag: bool,
}

impl Future for YieldFuture {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context) ->
   Poll<Self::Output> {
        if self.flag {
            Poll::Ready(())
        } else {
            self.flag = true;
            cx.waker().clone().wake();
            Poll::Pending
        }
    }
}
```

Listing 3.4　底层 Future 支持 Yield

3.4.3.2 Sleep

Sleep 操作是与 Yield 类似且同样普遍的操作，指在一定时间内让当前线程进入休眠状态。在传统的内核实现中，对于 Sleep 的实现有过多种可靠成熟的实现方式。在 rCore 中，依靠时钟事件队列与线程调度器的配合来实现在特定的时间节点唤醒特定线程，从而为用户态提供 Sleep 服务。每次内核从调度器取出线程执行之前，都将首先查询时钟队列，是否有之前定义的事件到期，需要执行，使时钟队列的优先级高于调度器的任务队列。而进行 Sleep 系统调用时，则可以在时钟队列尾部加入对应的事件之后，将当前线程标记为休眠状态。

对于 zCore 的设计而言，当一个线程所在的协程被标记为休眠状态，那么只有通过上下文引用进行唤醒操作，才能让该协程再次被调度器调度。因此，Sleep 问题在 zCore 中可以认为是一个简单的回调问题，内核维护一个全局的回调事件队列，在发生时钟中断时检查队列、进行回调。计划让自身 Sleep 的协程在回调时间队列中注册一个唤醒自身的回调操作之后，返回阻塞即可。

```
pub struct SleepFuture {
    deadline: Duration,
}

impl Future for SleepFuture {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<
   Self::Output> {
        if timer_now() >= self.deadline {
            return Poll::Ready(());
        }
        if self.deadline.as_nanos() < i64::max_value() as u128
    {
            let waker = cx.waker().clone();
            crate::timer_set(self.deadline, Box::new(move |_|
   waker.wake()));
        }
        Poll::Pending
    }
}
```

Listing 3.5　底层 Future 支持 Sleep

在代码块 3.5中，展示了目前 zCore 中对 Sleep 这一底层 Future 的实现。在简单比较当前时间，确认需要睡眠之后，通过注册一个回调函数即可在之后确定的时间将自身任务标记为活跃，重新参与 naive-executor 的调度。而首次 poll 返回的 Pending 结果将会直接借用调度器的支持，使当前协程暂时退出执行。

3.4.3.3 实现 object.wait

wait 操作是 Zircon 的系统调用中的常用操作，用户态程序所能够看到的所有资源都被封装为内核对象，对于某个内核对象进行信号监听，阻塞当前线程直到监听结果返回，是 Zircon 用户程序中常用的一种交互手段，普遍用于监控 IPC 通信。实现这种信号监听机制的高效手段是在具体信号发生时使用回调函数唤醒对应线程，而在信号发生前将该线程设置为休眠，不占用 CPU 资源。

在 zCore 中，实现信号监听操作同样借助了回调函数这一手段，但 async 语法的使用在语法上简化了回调函数的内容，也让唤醒线程的操作更自然。在 SignalFuture 这一底层 Future 的 poll 接口中，首先对目前的信号状态进行判断，如果有符合条件的信号，则直接返回 Ready。如果需要进行等待，则在目标对象的回调队列中注册一个回调函数，函数内判断该对象发生的信号，并在符合条件时唤醒当前线程。真实的 Future 对象设计在代码段 3.6中给出。

```
struct SignalFuture {
    object: Arc<dyn KernelObject>,
    signal: Signal,
    first: bool,
}

impl Future for SignalFuture {
    type Output = Signal;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->
    Poll<Self::Output> {
        let current_signal = self.object.signal();
        if !(current_signal & self.signal).is_empty() {
            return Poll::Ready(current_signal);
        }
        if self.first {
            self.object.add_signal_callback(Box::new({
                let signal = self.signal;
                let waker = cx.waker().clone();
                move |s| {
                    if (s & signal).is_empty() {
                        return false;
                    }
                    waker.wake_by_ref();
                    true
                }
            }));
            self.first = false;
        }
        Poll::Pending
    }
```

```
}
```

Listing 3.6　底层 Future 支持 Wait

3.5　zCore 的用户态运行支持

libos 版 zCore 的开发与裸机版 zCore 同步进行，两个版本的 zCore 共用除了 HAL 层之外的所有代码。为了支持 libos 版 zCore 的正常运行，zCore 在地址空间划分方面对 Zircon 的原有设计进行了一定的修改，并为此对 Fuchsia 的源码进行了简单的修改、重新编译；另外，libos 版 zCore 需要的硬件相关层（HAL）将完全由宿主 OS 提供支持，一个合理的 HAL 层接口划分也是为支持 libos zCore 做出的重要考虑。本小节的余下内容中，为了方便起见，将用 libos 来指代 libos 版本的 zCore，用 zCore 来指代裸机版 zCore。

3.5.1　修改 VDSO

VDSO 是由内核提供、内核负责映射的动态链接库，以函数接口形式提供系统调用接口。原始的 VDSO 中将会最终使用 syscall 指令从用户态进入内核态。但在 libos 环境下，内核和用户程序都运行在用户态，因此需要将 syscall 指令修改为函数调用，也就是将 sysall 指令修改为 call 指令。在 libos 内核初始化环节中，将 VDSO 中的 syscall 指令修改为 call 指令，并指定跳转的目标地址，重定向到内核中处理 syscall 的特定函数，从而实现模拟系统调用的效果。

3.5.2　调整地址空间范围

在 libos 中，使用 mmap 来模拟页表，所有进程共用一个 64 位地址空间。因此，从地址空间范围这一角度来说，运行在 libos 上的用户程序所在的用户进程地址空间无法像 Zircon 要求的一样大。对于这一点，我们在为每一个用户进程设置地址空间时，手动进行分配，规定每一个用户进程地址空间的大小为 0x100_0000_0000，从 0x2_0000_0000 开始依次排布。0x0 开始至 0x2_0000_0000 规定为 libos 内核所在地址空间，不用于 mmap。图 3.3给出了 libos 在运行时若干个用户进程的地址空间分布。

与 libos 兼容，zCore 对于用户进程的地址空间划分也遵循同样的设计，但在裸机环境下，一定程度上摆脱了限制，能够将不同用户地址空间分隔在不同的页表中。如图 3.4所示，zCore 中将三个用户进程的地址空间在不同的页表中映射，

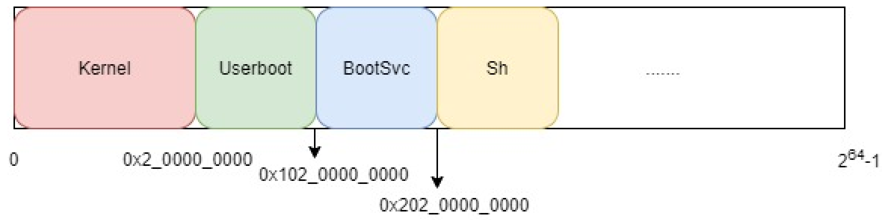


图 3.3　libos 地址空间分布

但是为了兼容 libos 的运行，每一个用户进程地址空间中用户程序能够真正访问到的部分都仅有 0x100_0000_0000 大小。

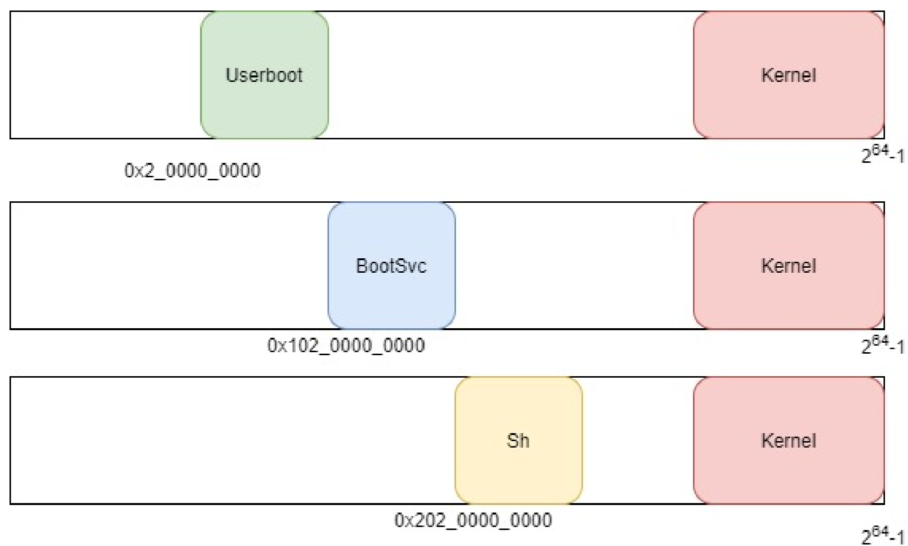


图 3.4　zCore 用户进程地址空间分布

3.5.3　HAL 层接口设计

HAL 层的设计与 zCore 和 libos 的开发是同步进行的，在开发过程中将硬件实现相关的接口，比如页表、物理内存分配等进行封装，暴露给上层的内核对象层使用。在 kernel-hal 模块中，给出空的弱链接实现，由 zCore 或 libos 的开发者对相应的接口进行相应的实现，并用设定函数链接名称的方式，替换掉预设的弱链接的空函数。在整个开发过程中，不断对 HAL 层提出需求并实现，目前形成了第一版 HAL 层接口，在设计上能够满足现有的内核对象实现所需要的功能。

对内核对象层而言，所依赖的硬件环境不再是真实硬件环境中能够看到的物理内存、CPU、MMU 等，而是 HAL 暴露给上层的一整套接口。这一点从设计上来说，是 zCore 与 Zircon 存在差异的一点。Zircon 将 x86_64 和 ARM64 两个硬件架构进行底层封装，但是没有给出一套统一的硬件 API 供上层的内核对象直接使用，在部分内核对象的实现中，仍然需要通过宏等手段对代码进行条件编译，从

而支持同时面向两套硬件架构进行开发。而在 zCore 的内核对象层实现中，可以完全不考虑底层硬件接口的实现，使一套内核对象的模块代码可以同时在 zCore 和 libos 上运行，之后如果 zCore 进一步支持 ARM64 架构，只需要新增一套 HAL 的实现，无需修改上层代码。

表 3.1　zCore HAL 层部分接口概况

接口名称	功能描述
Thread::spawn	将一个线程加入调度
Thread::set_tid	设定当前线程的 id
Thread::get_tid	获取当前线程的 id
context_run	切换到特定的线程上下文
PageTable::current	获取当前页表
PageTable::new	新建一个页表
PageTable::map	将一个物理页帧映射到一个虚拟地址中
PageTable::unmap	解映射某个虚拟地址
PageTable::query	查询某个虚拟地址对应的页表项状态
PageTable::table_phys	获取对应页表的根目录表物理地址
PageTable::map_many	同时映射多个物理页帧到连续虚拟内存空间
PageTable::map_cont	同时映射连续的多个物理页帧到虚拟内存空间
PageTable::unmap_count	解映射某个虚拟地址开始的一片范围
PhysFrame::alloc	分配一个物理页帧
PhysFrame::alloc_contiguous	分配一块连续的物理内存
PhysFrame::zero_frame_addr	返回零页的物理地址
PhysFrame::drop	回收该物理页帧
pmem_read	读取某特定物理页帧的内容到缓冲区
pmem_write	将缓冲区中的内容写入某特定物理页帧
frame_copy	复制物理页帧的内容

在表格 3.1中列出了 HAL 层的部分接口设计，覆盖线程调度与内存管理两方面。在线程调度方面，Thread 结构体相关的接口与 context_run 接口主要用于将一个线程加入调度、手动进行上下文切换等基本操作。在 zCore 的相关实现中，线程调度的各接口使用 naive-executor 给出的接口以及 trapframe-rs 给出的接口来进行实现，二者都是我们为裸机环境的协程调度与上下文切换所封装的 Rust 库。libos 中，线程调度的相关接口依赖于 Rust 的用户态协程支持以及 libos 开发者实现的用户态上下文切换。

在内存管理方面，HAL 层将内存管理分为页表操作与物理页帧管理两方面，

并以此设计接口。在 zCore 实现中，物理页帧的分配与回收由于需要设计物理页帧分配器，且可分配范围大小与内核刚启动时的内存探测密切相关，我们将其直接在总控模块 zCore 中进行实现。而在 libos 中，页表对应操作依赖 mmap 进行模拟，物理页帧的相关操作则直接使用用户态物理内存分配器进行模拟。

在 Zircon 的设计中，内存的初始状态应该设置为全 0，为了在内核对象层满足该要求，我们为 HAL 层设计了零页接口，要求 HAL 层保留一个内容为全 0 的物理页帧，供上层使用。上层负责保证该零页内容不被修改。

第 4 章　zCore 测试与分析

Rust 是一门安全、高效的语言，使用包括所有权机制和生命周期在内的一系列语法，为开发者的代码进行更强的安全性约束，由此也造成了学习成本高、开发时较难通过编译器检查等特点。在给出较强语法约束的同时，Rust 也学习其他语言的特点，给出了对 Unafe Rust 的支持，允许开发者在使用 unsafe 标记代码的情况下，在 unsafe 块内部写可能导致内存不安全的代码。因此，Unsafe 代码在 Rust 程序中需要被谨慎使用，并且 Unsafe 代码出现错误的概率远高于其他普通代码。

本章将首先对 zCore 实现的系统调用的功能正确性进行说明，然后对 zCore 中的 Rust 代码进行 Safety 说明，探讨在 zCore 中使用 Unsafe 代码的必要性以及在 zCore 中对 Unsafe 代码进行约束的做法。另外，对于 Rust 在一定的安全性语法约束下所表现出的性能，也是我们重点关注的讨论点，我们采用 Fuchsia 官方编写的小型性能测试 microbenchmarks，对 zCore 和 Zircon 进行性能对比测试，并对结果进行分析，以此对 zCore 开发的成果进行衡量，验证用 Rust 开发内核的可行性。

4.1　系统调用实现情况

zCore 对照 Zircon 的系统调用设计文档，主要目标是能够支持尽可能多的 zircon 系统调用，支撑运行 Fuchsia 用户程序。目前支持的系统调用数量为 93 个，约占全部系统调用的 60%。zCore 目前能够在用户态运行简单的 shell 程序和其他像 ps 这样的简单用户程序。

在系统调用实现的可靠性上，经过对官方源码的调研，选用官方提供的 core-tests 作为系统调用实现的单元测试，来保证系统调用实现的功能正确性。已实现的系统调用目前均可以通过多数测试。具体的测试情况如表格 4.1所示。

表 4.1　zCore 系统调用单元测试情况

测试名称	通过情况	测试名称	通过情况
Bti	7/8	ConditionalVariableTest	3/3
C11MutexTest	5/5	C11ThreadTest	6/6
ChannelInternalTest	2/2	ChannelTest	38/38

续下页

续表 4.1　zCore 系统调用单元测试情况

测试名称	通过情况	测试名称	通过情况
ChannelWriteEtcTest	27/27	ClockTest	2/2
ProcessDebugUtilsTest	1/1	ProcessDebugTest	0/4
ExecutableTlsTest	12/12	EventPairTest	8/8
FifoTest	9/9	FPUTest	1/1
FutexTest	14/14	HandleCloseTest	2/3
HandleDup	4/4	HandleInfoTest	4/4
HandleTransferTest	2/2	HandleWaitTest	2/2
InterruptTest	5/7	JobTest	8/26
MemoryMappingTest	5/8	ObjectChildTest	1/1
ObjectGetInfoTest	4/4	JobGetInfoTest	12/39
ProcessGetInfoTest	25/69	TaskGetInfoTest	9/12
ThreadGetInfoTest	28/41	VmarGetInfoTest	19/21
ObjectWaitOneTest	5/5	ObjectWaitManyTest	5/5
Pager	0/76	PortTest	8/18
ProcessTest	4/26	SchedulerProfileTest	0/14
Pthread	6/6	PThreadBarrierTest	3/3
PthreadTls	1/1	Resource	0/11
SocketTest	24/26	StackTest	2/2
StreamTestCase	0/11	SyncCompletionTest	4/11
SyncCondition	2/2	SyncMutex	3/3
SystemEvent	0/9	Threads	5/36
TicksTest	1/1	Vmar	4/33
VmoCloneTestCase	6/6	VmoClone2TestCase	29/41
VmoCloneDisjointClonesTests	2/2	VmoCloneResizeTests	3/4
ProgressiveCloneDiscardTests	2/2	VmoSignalTestCase	3/3
VmoSliceTestCase	13/15	VmoZeroTestCase	12/12
VmoTestCase	7/27		

4.2　zCore 中的 Unsafe Rust

Rust 注重安全和高效，这两点都通过编译器的检查和编译期优化来实现。Rust 与 C 或 C++ 在性能和安全性上有较大的差别。在 Rust 的设计理念中，所有权机制与生命周期检查将对程序的安全性有更好的保障，而 Unsafe 代码的使用可能破坏这些安全性保障。另外，在性能优化方面，Unsafe 代码在具有破坏安全性的风险的同时，也是优化的一个主要手段，通过适当的 Unsafe 代码使用，可以明显

提高 Rust 程序的性能。因此，对 Unsafe 代码进行必要的 Safety 说明，明确 unsafe 代码应该出现的范围，是一个成熟的 Rust 库/程序应该做到的。而 zcore 作为一个用 Rust 主力开发的内核，设定 Unsafe 代码的出现范围，既是对普通内核开发者的有力约束，也是在代码规模增大的过程中保证安全性不被轻易破坏的条件。

在这里对于 Rust 程序安全性的说明中，我们引入关键代码（Trusted Code）的概念。多数靠近底层的 Rust 程序或需要注重性能优化的 Rust 程序，将会不可避免出现 unsafe 代码，或者主动添加 unsafe 代码进行优化。在系统内核这样的编程环境下，我们希望能够借助对 unsafe 代码的正确使用，高效开发内核，也希望对 unsafe 代码进行约束，避免因 unsafe 代码的滥用而破坏系统稳定性。因此，设计上我们以 HAL 层作为边界，HAL 层本身的实现或其依赖的第三方库，通常需要用到像直接使用指针操作内存这样的 unsafe 操作，对 unsafe 代码的使用不可避免。而在 HAL 层之上的代码中，由于下层已经封装了足够多的 unsafe 操作，所以 unsafe 代码应该被明确禁止。我们认为允许出现 unsafe 代码块的模块是关键代码，即我们信任该模块代码的开发人员，允许他们使用 unsafe 语法，但需要对每一处 unsafe 代码的使用进行确定的 Safety 说明，并尽量减少 unsafe 代码的使用。而不允许出现 unsafe 代码块的代码被认为是非关键代码，该类模块的代码不应出现 unsafe 语法，我们不对这部分模块的开发者予以主观上的信任，而是由 Rust 强大的编译期检查来约束开发者写出内存安全的代码，从而减少错误。

4.2.1 Unsafe Rust 在 zCore 中的分布

在 zCore 中，unsafe 代码中的绝大多数都被封装在了 HAL 层中，被认为是关键代码。但由于在内核对象层和系统调用层使用了核心库中一些与内存操作相关的底层函数，在使用这些函数时需要添加 unsafe 标记。HAL 层及以下的部分代码中，使用 unsafe 语法是必不可少的，因此在这里我们不对 HAL 层及以下的代码进行 safety 分析；在表格 4.2中，我们对 HAL 层之上的 unsafe 代码进行统计和简单的 safety 说明，对相关的这些 safety 说明，将在下一小节给出描述。

表 4.2　zCore HAL 层之上 unsafe 分布情况

对应模块	使用原因	使用次数
zircon-object	强制裸指针转换 Arc::get_mut_unchecked transmute/transmute_copy 读/写 union core::slice::from_raw_parts	14
zircon-syscall	读取随机数	1

4.2.2　Unsafe Rust 在 zCore 中的安全性

4.2.2.1　强制裸指针转换

裸指针转换在这里主要指将 usize 直接转换为具有可变引用权限的指针，这一操作在 Rust 的 unsafe 操作中危险程度可谓名列前茅。类似的操作在 C/C++ 中是常见操作，并且在多数时候提高效率，但其带来的安全漏洞一直被许多开发者诟病。在 zCore 中，对裸指针转换一直保有非常谨慎的态度，但确实有不可避免需要进行转化的地方。最常见的场景是在解析 elf 文件、进行动态链接库重定位时，需要进行从整数直接到可变引用指针的转换，然后修改该指针指向的内容。关于该类操作的安全性，这类操作都出现在 zCore 的内核初始化早期环节，比如 VDSO 的映射，与内核的正常初始化密切相关，如果该操作出现错误，内核将无法正常初始化。因此我们认为这类操作在 zCore 中可以由实际运行情况来确定其安全性。

同时，我们已经尽可能缩减这类操作的使用，在后续开发过程中，随着 HAL 层和内核初始化环节的不断完善，此类操作在 zCore 中将会被极力避免。

4.2.2.2　Arc 指针转变为可变引用 &mut

Arc 指针是 Rust 中的线程安全引用计数指针，Weak 指针是与其相对应的弱引用指针，每一个 Arc 指针可以认为是目标结构体实例的一个不可变引用，而每一个 Weak 指针则可以认为是目标结构体的不可变弱引用。指向某一个结构体对象的 Weak 指针必须通过 Arc 指针的可变引用才能获得。而 Weak 指针需要通过 upgrade 接口升级为 Arc 指针，才能处理对应的结构体对象，如果目标结构体已经由于 Arc 指针全部销毁而被回收，那么 upgrade 接口升级将会失败。

正常情况下，Arc 指针对目标结构体实例持有不可变引用。在 zCore 中，通过 Arc 指针获得可变引用这一需求，主要用于实现 IPC 对应的成对的内核结构体。zCore 设计中的 IPC 如 Channel、Fifo 等，都是由一对结构体组成的双端通信。在

代码块 4.1中，我们给出了 Channel 的简化版结构体定义，两端的结构体分别持有对方的 Weak 引用，并且两端的结构体将分别通过 Arc 引用，作为内核对象而被内核中的其他数据结构使用。

```
pub struct Channel {
    peer: Weak<Channel>,
    recv_queue: Mutex<VecDeque<T>>,
    //...
}
```

Listing 4.1　IPC 结构体定义示例——Channel

因此，在同时初始化该类 IPC 的两端的同时，将必须对某一端的 Arc 指针进行获取可变引用的操作，即代码段 4.2中的 get_mut_unchecked 接口。当 Arc 指针的引用计数不为 1 时，这一接口是非常不安全的，但是在当前情境下，我们使用这一接口进行 IPC 对象的初始化，安全性是可以保证的。

```
#[allow(unsafe_code)]
pub fn create() -> (Arc<Self>, Arc<Self>) {
    let mut channel0 = Arc::new(Channel {
        peer: Weak::default(),
        recv_queue: Default::default(),
        //...
    });
    let channel1 = Arc::new(Channel {
        peer: Arc::downgrade(&channel0),
        recv_queue: Default::default(),
        //...
    });
    // no other reference of `channel0`
    unsafe {
        Arc::get_mut_unchecked(&mut channel0).peer = Arc::
   downgrade(&channel1);
    }
    (channel0, channel1)
}
```

Listing 4.2　IPC 结构体创建实例——Channel

4.2.2.3　transmute/transmute_copy

transmute/transmute_copy 是 Rust 核心库中提供的两个对内存进行操作的底层函数，主要用于内存布局转换和内存直接复制。对于 transmute 接口，给入的参数是一个结构体实例，将会被直接转换为对应大小的字节数组并返回，数组与结构体实例对应的内存区域完全相同。transmute 接口在 zCore 中主要用于将初始化

好的结构体实例转为字节数组，然后将数组复制到用户态地址空间或通过 IPC 传递。transmute_copy 则更多用于接受指向某一个结构体实例的不可变引用，复制对应的内存区域，然后返回另一类结构体实例。

transmute 函数与 transmute_copy 函数被 Rust 编译器认为是不安全的原因，主要是由于这两个函数对 Rust 构建的类型系统可能会具有破坏性。在使用这两个函数的安全性保证上，一方面我们通过足够多的单元测试来对类型转换前后的内存布局等进行验证；另一方面，使用这两个函数的代码位置都是 zCore 中经常运行到的代码，我们在内核实际运行过程中也比较容易发现其导致的问题。目前来看，这两个函数的使用还未暴露出错误。

4.2.2.4 读写 union

Union 结构体的概念在 C/C++ 中同样存在，Rust 中的 Union 相比 C/C++ 中的 Union，在灵活性上更受限制。同时，在 Rust 中，关于 Union 的全部操作被认为是 unsafe 的。这也是我们在使用到 Union 的过程中必须用到 unsafe 关键字的原因。但在 zCore 中，Union 结构内都是纯数据类型，对应的变量都经过了初始化；并且所使用的的 Union 对象都仅由内核管理，用户程序的数据不在此范围。因此，我们认为 Union 相关的读写操作在 zCore 中是安全的。

4.2.2.5 from_raw_parts

from_raw_parts 是 Rust 核心库提供的基于常数指针的类型转换函数，支持将一个指向特定结构体的常数指针转为一个指向同类型结构体数组的不可变引用。该函数被认为是不安全的主要原因在于，使用者如果不能确定常数指针指向的内存情况，可能发生运行时错误。并且该函数没有维护类型系统的完整性。在 zCore 中，我们仅使用了一次该函数，具体用处是为了实现内核中的程序计数器 kcounter。

kcounter 的实现过程：我们通过编写特定的宏，帮助指定内核中的某些 static 变量将被链接的段位置。在运行时，我们通过将特定的段符号转换为 usize，获取段的始末位置，然后将 usize 转换为计数器结构体类型的常量指针。最终通过 from_raw_parts 获取对应的计数器数组引用。

在上述应用场景中，基于开发者对对应数组的了解情况，可以规避 Rust 设计中所担心的 from_raw_parts 的不安全情景。

4.2.2.6 读取随机数

Zircon 的设计中有专门用于读取随机数的相关系统调用，在该系统调用的实现例程中，我们直接使用了 Rust 核心库支持的 _rdrand32_step 函数，该函数本身被标记为 unsafe。这一操作使用 unsafe 语法不可避免，即使由我们自己手动进行实现，也需要对底层寄存器进行直接操作，同样需要 usnafe 语法。该操作目前由于时间问题尚未被移入 HAL 层，在之后将会移入到 HAL 层中。

4.3 基于 microbenchmarks 的性能测试与调优

microbenchmarks 是 Fuchsia 官方设计的小型性能测试，涵盖虚拟内存性能测试、多线程性能测试、用户态内存分配器测试、IPC 测试等多个方面。我们选取其中的虚拟内存性能测试与 IPC 测试两方面，在支持 KVM 的 QEMU 上对 Zircon 与 zCore 进行测试，比对结果，并记录我们对 zCore 的调优过程。

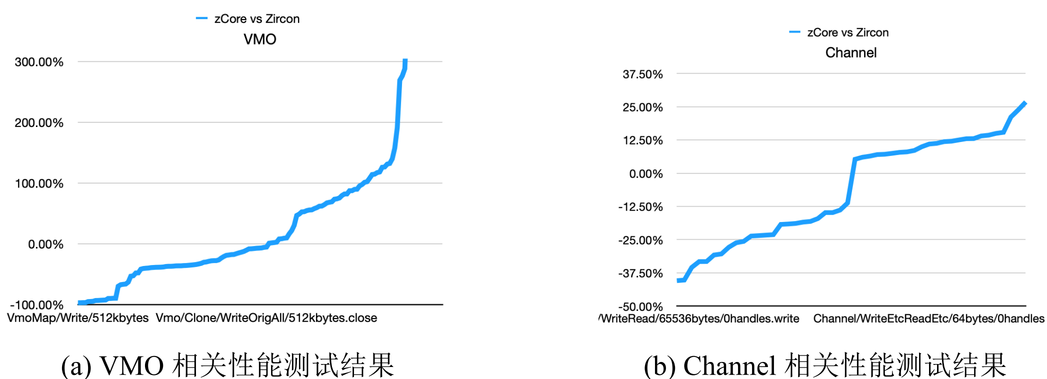
4.3.1 硬件环境

硬件环境选用 Ubuntu 上带 KVM 支持的 QEMU 5.0，内存大小为 4G。

4.3.2 测试结果及分析

完整的测试结果在附录 B的表格中列出，包括虚拟内存对象（Virtual Memory Object）和 Zircon 中最常用的 IPC——Channel 的测试。在这里，我们给出根据表格生成的图形。

在图片 4.1(a)和图 4.1(b)中，分别给出了大部分数据的对比结果。对于少部分差异较大的数据，由于数据截断在图中不可见，之后的分析同时将对差异较大的数据给出说明。



(a) VMO 相关性能测试结果　　(b) Channel 相关性能测试结果

图 4.1　性能测试结果

从图 4.1(a)中可以看出，大部分数据表明 zCore 与 Zircon 完成测试所用的时间有接近 1-3 倍的差距。在对实验结果的分析中，我们推测的主要原因有：

- zCore 仅从最基本的需求上实现了功能正确性，但是没有对性能相关的一些边界条件进行优化。
- 在 VMO 内部使用的数据结构与算法具有较大的影响，在之前我们将 VMO 内部的 BTreeMap 数据结构换成 HashMap 之后，提升了几倍的性能。因为查询效率从 O(logN) 提升到了 O(1)，在图 4.2中给出了前后测试结果比对。后续仍可以对相关数据结构进行优化，比如直接使用 Vec。
- VMO 内部被组织成一颗树，每个节点管理若干个物理页帧，根据树的各个节点之间的页帧共享实现 Copy On Write 机制。但在 zCore 的默认实现中，Copy On Write 机制没有被默认启用，而 Zircon 默认则是启用了 Copy On Write 机制。因此在部分性能测试上呈现较为极端的结构。在需要更快访问内存对应部分的测试中，zCore 占优，而在需要快速分配、映射、回收的测试中，Zircon 占优。

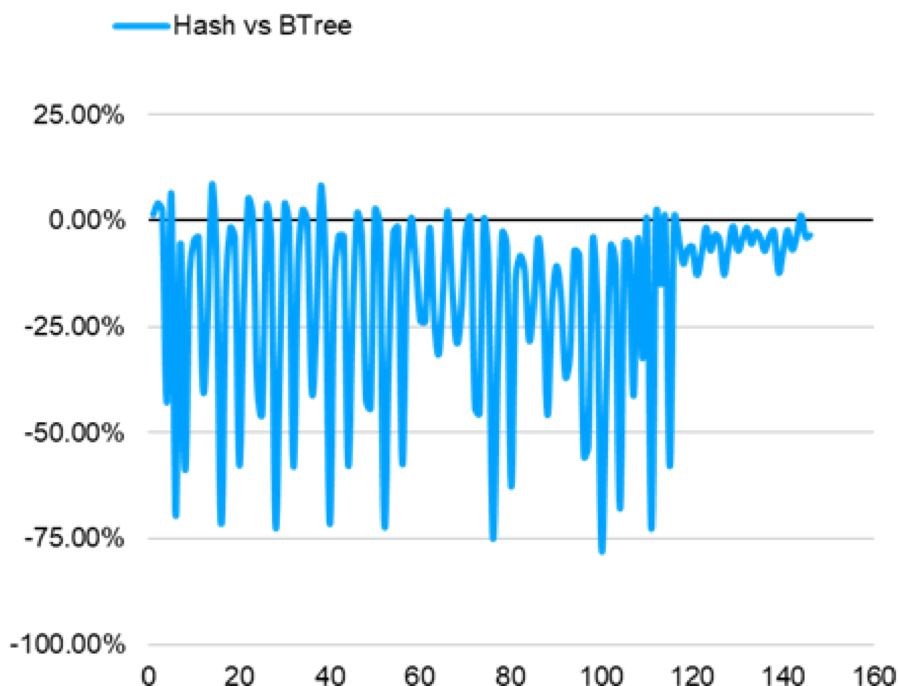


图 4.2　使用 BTreeMap 与使用 HashMap 的 VMO 测试对比

以上原因是基于本毕设开展过程中对 Zircon 的分析与对 zCore 的理解进行的推测，在实际尝试对 zCore 进行的优化过程中，我们同时发现了 Rust 自带的 rlibc 中的 memcpy 函数对性能造成了一定的影响， 4.3.3小节将详细说明这一优化过

程。

对 Channel 的性能测试结果相比 VMO 的测试结果来说更加正常。Channel 是在 Zircon 的设计中最基本、最底层的 IPC 机制，采用报文式传输，每一则消息中允许包含一块内存区域和若干个内核对象句柄。关于 Channel 的测试结果，经过分析我们发现，在消息长度接近 64Byte 大小的情况下，zCore 速度逐渐慢于 Zircon。我们推测主要原因在于 Zircon 中对 Cache 进行了适配，而这一点在当前的 zCore 中还未进行。

另一方面，在内核/用户之间的内存复制也是可能影响 VMO 与 Channel 的测试结果的原因，现在 zCore 在接受用户态传入的指针时，会先将指针所指向的内容进行复制，然后再传递给内核对象层进行操作；在内核对象层中，VMO 相关的接口会将对应内容写到物理页帧中。在此期间有两次复制。而 Zircon 中，借用 C++ 对指针操作的灵活性，仅需要一次复制。Channel 的消息收发过程中同样有类似的过程，因此当前这一内存复制机制同时可能影响 Channel 的测试效果。

4.3.3 优化过程记录

在对 zCore 的性能进行分析优化的过程中，我们发现 zCore 的内存复制比预想的要缓慢，最终经过调研，发现在 rlibc 中存在几乎没有进行优化的 memcpy 代码。如代码块 4.3所示，这部分 Rust 代码在设计上存在低效问题，编译器将 Rust 代码直接翻译为汇编的过程中，没有对这部分进行足够的优化。导致在汇编层面仍然是按字节复制。

```
#[cfg_attr(all(feature = "mem", not(feature = "mangled-names")
   ), no_mangle)]
pub unsafe extern "C" fn memcpy(dest: *mut u8, src: *const u8,
    n: usize) -> *mut u8 {
    let mut i = 0;
    while i < n {
        *dest.offset(i as isize) = *src.offset(i as isize);
        i += 1;
    }
    dest
}
```

Listing 4.3 rlibc 中的低效 memcpy

在之后，我们使用自行开发的 rlibc-opt 库，从 musl libc 中直接参考了一份 memcpy 的实现，即在代码块 4.4中显示的汇编代码。经过这一优化，对内存复制相关的测试有一定的提升。结果在表格 4.3中体现。

```
.global memcpy
```

```
memcpy:
__memcpy_fwd:
    mov %rdi,%rax
    cmp $8,%rdx
    jc 1f
    test $7,%edi
    jz 1f
2:  movsb
    dec %rdx
    test $7,%edi
    jnz 2b
1:  mov %rdx,%rcx
    shr $3,%rcx
    rep
    movsq
    and $7,%edx
    jz 1f
2:  movsb
    dec %edx
    jnz 2b
1:  ret
```

Listing 4.4 rlibc-opt 中的高效 memcpy

表 4.3 使用 rlibc 与使用 rlibc-opt 性能对比测试结果

Case	rlibc	rlibc-opt	rlibc-opt vs rlibc
VmoMap/Read/Kernel/128kbytes	108923	25711	-76%
VmoMap/Read/Kernel/2048kbytes	1732137	484787	-72%
VmoMap/Read/Kernel/512kbytes	434396	103447	-76%
VmoMap/Write/Kernel/128kbytes	144539	24347	-83%
VmoMap/Write/Kernel/2048kbytes	2269986	404771	-82%
VmoMap/Write/Kernel/512kbytes	577157	99784	-83%
VmoMapRange/Read/Kernel/128kbytes	108681	25309	-77%
VmoMapRange/Read/Kernel/2048kbytes	1731293	483149	-72%
VmoMapRange/Read/Kernel/512kbytes	434299	103591	-76%
VmoMapRange/Write/Kernel/128kbytes	144039	23658	-84%
VmoMapRange/Write/Kernel/2048kbytes	2269785	405616	-82%
VmoMapRange/Write/Kernel/512kbytes	571828	100635	-82%

第 5 章　项目总结

到目前为止，zCore 项目已经能够支持相当数量的系统调用，支持部分用户态程序的正常运行。zCore 项目在开发过程，不仅是对上一代 Rust OS 的充分借鉴与反思，也是对使用 Rust 开发内核的又一次新的尝试。在对 zCore 的设计和开发过程中，我们更加坚定了 Rust 这门零抽象开销、内存安全、线程安全的语言，能够在系统底层的编程中有所贡献。

在对 zCore 的设计中，我们借助 async 的异步编程机制简化了操作系统内核开发，为在内核开发中使用 async 机制进行了一次积极的尝试。目前，zCore 在性能上与 Zircon 仍有一定差距，在架构设计、功能完善程度上也需要进一步改进，但截至目前的 zCore 开发让我们看到了 Rust 巨大的潜力，相信 Rust OS 的不断迭代将会让 Rust 在操作系统内核开发中正式占据一席之地！

插图索引

图 1.1　Zircon 用户态程序执行环境 2
图 2.1　zCore 项目设计图 8
图 2.2　zCore 内核运行概况 9
图 3.1　用户态 async 实例[3] 13
图 3.2　naive-executor 执行简图 14
图 3.3　libos 地址空间分布 24
图 3.4　zCore 用户进程地址空间分布 24
图 4.1　性能测试结果 33
图 4.2　使用 BTreeMap 与使用 HashMap 的 VMO 测试对比 34
图 A-1　一个用于随机数生成器的系统调用接口的软件体系结构 45

表格索引

表 3.1 zCore HAL 层部分接口概况 25
表 4.1 zCore 系统调用单元测试情况 27
表 4.2 zCore HAL 层之上 unsafe 分布情况......30
表 4.3 使用 rlibc 与使用 rlibc-opt 性能对比测试结果 36
表 B-1 zCore 与 Zircon 性能对比测试结果 56

参考文献

[1] Cutler C, Kaashoek M F, Morris R T. The benefits and costs of writing a POSIX kernel in a high-level language [C/OL] // Arpaci-Dusseau A C, Voelker G. 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. USENIX Association, 2018: 89-105. https://www.usenix.org/conference/osdi18/presentation/cutler.

[2] Levy A, Campbell B, Ghena B, et al. Multiprogramming a 64kb computer safely and efficiently [C/OL] // Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. ACM, 2017: 234-251. https://doi.org/10.1145/3132747.3132786.

[3] Gruber B. Explained: How does async work in rust? [EB/OL]. (2019-04-02)[2019-05-02]. https://dev.to/gruberb/explained-how-does-async-work-in-rust-46f8.

致　谢

感谢陈渝老师和向勇老师，在毕设完成过程中对我悉心指导，让我能够顺利完成开题时的预期工作。二位老师在我学习操作系统各方面知识的过程中也给予了我非常多的指导与帮助，让我对操作系统各方面理论有了初步的了解。

感谢王润基学长在我进行毕设过程中的大力支持，他对整个 zCore 项目的立项、开发、调试都起到了关键性的作用，也在我撰写论文期间给予了我很多指导。

zCore 从立项起便是一个开源项目，随着代码规模的壮大，更多的开发者开始注意到 zCore、为 zCore 贡献代码。感谢孙文厚、张晨两位同学，他们在本学期的课程设计中选择了 zCore，并为实现 zCore 的用户态驱动机制做出了贡献。

感谢我的家人和朋友们，疫情期间在家开展毕设多有不便，是他们给了我美好的心情和不愁吃穿的生活，让我得以安心工作。

感谢本科期间各位老师的认真教导与不挂之恩，感谢身边所有同学四年来的陪伴与帮助，让我拥有了这段安逸快乐的时光。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：潘庆霖　日　期：2020年6月15日

附录 A　外文资料的书面翻译

用 Rust 编写操作系统内核的案例[1]

A.1　摘要

关于在操作系统内核中增加安全机制的研究已经开展了数十年，但在多数实际系统中都失败了。尤其是牺牲性能的方案，通常会直接被拒绝。但是，现代编程语言的隔离技术能够提供安全性并且避免性能问题。更进一步地，一门不包含垃圾回收或其他运行时服务的类型安全语言，能够避免原先最大部分的受信任代码库。我们将汇报我们用 Rust 编写一个资源高效的嵌入式内核的经历，再吃过程中我们发现在内核构建中只需要一小部分的 unsafe 抽象。并且，我们认为 Rust 使用类型系统来避免运行时内存管理的做法，将会孕育下一代安全的操作系统。

A.2　介绍

绝大多数操作系统认为内核代码是受信任的。在一定程度上，这是由于系统构建者依赖于硬件提供的强制内存保护，尤其是进程抽象来提供隔离。然而，进程是一种重量级抽象：它包含栈、文件描述符、以及许多内存中的数据结构。更进一步地，虚拟内存地址间的切换开销巨大。像 Nooks 和 seL4 这样的系统已经探索了使用比进程更好的地址空间来进行隔离，显示了大幅减少额外开销的途径。

本文采用了更为极端的方法：完全不依赖内核中的硬件保护，而是用内存安全的编程语言来编写内核。这种方法在之前已经被尝试过，并取得了不同的成功：Spin 允许应用程序拓展和优化内核，通过下载使用 Modula-3 开发的模块来优化内核性能；而 Singularity 是使用 Sign#（C# 的一个变种）进行开发，并且提供了软件隔离进程（SIP）抽象机制。然后，Spin 和 Singulariy 都是用了带垃圾回收的语言，这类语言为内核带来了许多问题。垃圾回收让内存布局变得复杂，背后的各类锁带来了时间不确定性，并且引入了导致整个 OS 暂停的全局间隔。更进一步，Spin 和 Singularity 都依赖于一个庞大、不安全的基础代码，即 Spin 内核和 Singularity 的语言运行时。

受类型安全编程语言的新近优点启发，本论文提出，用一门类型安全且不带

垃圾回收机制的编程语言来开发整个内核。这样的内核能够同时满足线程安全并满足内核开发人员所需要的的内存管理控制和内存确定性行为。一门新兴语言，Rust，满足这一需求。在 Rust 中，值是被生命周期监测的，在超出作用范围时被自动释放。

虽然开发任何内核都需要一些不安全代码，但我们认为内核的主要设计目标是最小化必须信任的 unsafe 代码数量。我们依据这一设计目标在低功耗微处理器上构建了一个内核。在我们的内核中，unsafe 代码只有两类：语言开发者提供的 Rust 库代码、内核开发者编写的内核代码。需要的 Rust 库代码只包含少量底层操作，比如浮点、边界检查、类型转换。内核中的受信任代码也是非常少的。包括上下文切换的标准抽象，系统调用陷入，中断处理，内存映射 I/O 端口，以及一个新的抽象 TakeCell，起源于 Rust 的内存模型。TakeCell 允许内核代码安全地使用静态编译、无额外开销的内联闭包来调整复杂的数据结构。

我们在第 1 章中描述 Rust 内存模型和内核代码中引入这一内存模型所面临的挑战。然后我们将描述我们用来构建内核的最小的受信任代码抽象的集合，并提供一些例子说明我们的内核如何借助这些抽象来提供一般的 OS 特性。

A.3 Rust

Rust 是一门为系统软件设计的类型安全语言 [1]。最初，它的提出是由于编写 Firefox 的布局引擎既要快速又高度并行的挑战。从那时起，它就被成功应用于像 Dropbox 后端存储这样的大型项目。对底层系统而言，它是一门特别吸引人的语言，因为它保留了类型安全且提供了类似 C 的运行时特征。本小节将会给出 Rust 的内存管理在开发过程中所引入的挑战；有兴趣的读者可以选择阅读 [2] 获取更多细节。

A.3.1 单可变引用

Rust 使用名为所有权（一个仿射类型系统 [3]）的理念来在编译时确定内存什么时候被释放。所有权在内核软件中引入的最主要挑战是无法同时存在两个指向同一片内存的可变引用（C 中的非常数指针）[2]。这是必要的，因为允许可变别名会允许程序规避类型系统 [4]。例如，考虑 Rust 的 enum 类型，该类型允许不同的独立类型共享同一片内存区域，就像 C 中的 union。在这个例子中，该 enum 既可以是 32 位无符号整数，也可以是一个指向 32 位无符号整数的可变引用：

```
// Rust
```

```
enum NumOrPointer {
  Num(u32),
  Pointer(&mut u32)
}
// Equivalent C
union NumOrPointer {
  uint32_t Num;
  uint32_t* Pointer;
};
```

不像 C 中的 union，Rust 的 enum 是类型安全的。语言将确保党编译器认为一个 NumOrPointer 是 Pointer 时，该值不可能被当成 Num 处理，反之亦然。

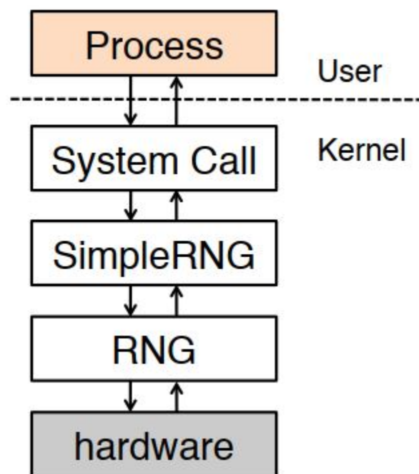


图 A-1　一个用于随机数生成器的系统调用接口的软件体系结构

存在指向同一个内存区域的两个可变引用将会破坏 NumOrPointer 的安全性，并允许代码构造任意的指针、处理任意区域内存。假定一个 NumOrPointer 现在是 Pointer，如果它的其中一个引用指向一个 Pointer，而另外的引用可以将其修改为一个 Num，那么它能够用来构造任意的指针：

```
// Rust
// n.b. will not compile
let external : &mut NumOrPointer;
match external {
  Pointer(internal) => {
  // This would violate safety and
  // write to memory at 0xdeadbeef
    *external = Num(0xdeadbeef);
    *internal = 12345; // Kaboom
},
...
}
```

```
// Equivalent C
// compiles without warning
union NumOrPointer* external;
uint32_t* numptr = &external->Num;
*numptr = 0xdeadbeef;
*external->Pointer = 12345;
```

A.3.2　内核需要多个引用

操作系统内核大量依赖回调函数和其他事件驱动编程机制。经常情况下，多个组件必须能够修改一个共享的数据结构。考虑这个例子，随机数生成器的软件栈在图 A-1中给出。RNG 为下层硬件 RNG 提供了一个接口，比如 Intel 的 RDRAND/RDSEED 或者 ARM 处理器上的一个 TRNG。

SimpleRng 在 RNG 和系统调用层之间。它在用户空间系统调用和 RNG 之间转换接口。它在一个进程请求随机数时调用 RNG，并在随机数生成之后回调系统调用层，将随机数传递给用户进程。实现该软件栈的一个自然方式是系统调用层和 RNG 都持有指向 SimpleRng 的一个引用。

```
pub struct SimpleRNG {
    busy: bool,
    ...
}
impl SimpleRng {
  fn command(&mut self) { self.busy = true; ... }
  fn deliver(&mut self, rand: u32) { self.busy = false; ... }
}
impl SysCallDispatcher {
  fn dispatch(&mut self, num: u32) {
    match num {
      // ...
      43 => self.simple_rng.command(),
    }
  }
}
impl RNG {
  fn done(&mut self, rand: u32) {
    self.simple_rng.deliver(rand);
  }
}
```

Rust 的所有权机制不允许两个结构体持有指向 SimpleRng 的引用（由于 command 标记 SimpleRng 为 busy，并且 deliver_rand 标记它为非 busy，都修改了 SimpleObject 的内部状态，所以引用必须为可变引用）。先前的工作表明，这个问题

意味着编写内核需要更改 Rust 语言 [2]。然而，下一小节我们将描述一个解决方案，由最少的受信任代码集合组成，代码中包含两个软件抽象，Cell 和 TakeCell。

A.4 面向 Rust 内核

使用 Rust 编写操作系统不需要进行任何操作语言的更改，但确实需要信任两种类型不安全的代码。第一类由 Rust 语言团队编写的 Rust 语言机制和库组成，提供了安全的接口，但是其底层实现使用不安全的代码。第二类受信任代码是内核代码的一部分，由内核开发人员编写，他们使用不安全的代码来实现基本操作系统功能，但同样可以提供安全接口。

本节的其余部分描述了哪些代码属于这两类代码。它们共同组成了内核中的所有 unsafe 代码，并且非常少，由所有语言和内核需要的机制组成。但是，还有两个附加的抽象接口。一个是由 Rust 提供的 Cell，它允许多个引用可变对象，但公开了有限的 API，需要进行数据复制才能访问，而是不是就地访问。第二个是由内核提供的接口，被称为 TakeCell。TakeCell 允许内核代码安全地运行共享复杂的结构，而没有运行时的开销（例如复制）和非常简单的编程抽象接口。

A.4.1 受信任 Rust 代码

Rust 具有大量可用的库，包括数据结构，Web 浏览器引擎，JavaScript 编译器和 I/O。一个内核仅需要 libcore，它支持原始类型，例如数组和编译器接口（LLVM[5]）固有操作。内核如果想要使用通用的动态内存分配器，也需要 liballoc。

这两个库都包含一些受信任的代码, 因为它们必须破坏类型系统（内存管理需要类型转换）或是为了进行性能优化。具体来说，内核依赖于以下四个使用 unsafe 代码的 Rust 抽象：

边界检查：数组是受边界检查的，所以 unsafe 代码使用长度域来确保操作是安全的。

迭代器优化：在 Rust 数组之间进行操作的规范方法是使用迭代器，迭代器通过 unsafe 代码避免不必要的边界检查。

编译器内部函数和基本类型转换：浮点数，易变负载/存储和原始类型之间的转换具有架构相关的特殊细节，这些实现细节 Rust 依赖 LLVM 来实现。

Cell：一种封装数据的抽象，使得内部引用无法转义，并且可以使用不可变引用来操作。

Cell 为事件驱动代码需要保存多个可变引用的问题（章节 A.3.2）给出了部分解决方案。Rust Cell 是一个可以被代码复制进/出的不透明的存储容器，但不能在内部引用。Cell 的关键特征是不可变参考可以复制到其中。出现的不安全类型转换, 在第 A.3.1节中使用 enum 引起的不安全性不能在 Cell 中发生，因为多个引用对共享数据的单独副本进行操作。

```
pub struct SimpleRng {
    busy: Cell<bool>,
    ...
}

impl Syscall for SimpleRng {
    fn command(&self) {
      ...
      self.busy.set(true);
    }
}
```

如上，我们展示了 Cell 能够解决在随机数生成器例子中的问题（图 A-1）。RNG 和系统调用分发器都持有一个指向 SimpleRng 的不可变引用。通常情况下，这表明来自两边的调用都无法改变 SimpleRng 的内部状态。然而，就像下面展示的，Cell 允许 command 方法，在 &self 是不可变引用的情况下，能够设置 busy 为 true。

不幸的是，Cell 只能是部分解决方案。它需要的内存拷贝引入了巨大开销，这对于复杂或大型的内核数据结构来说是不可接受的额外开销。下一小节将会介绍一个新的抽象机制，TakeCell，它允许安全、高效的内核接口实现。

A.4.2 受信任内核代码

上面描述的 Rust 抽象都为内核程序员提供了安全的接口。假设这些抽象都被正确实施，不允许调用者违反类型安全。然而，内核本身必须做一些根本上不安全的事情（例如上下文切换），并且必须将这些 unsafe 实现包装到 safe 接口中。惊喜的是，这只需要很少的 Rust 代码来实现。以下六种 unsafe 内核代码需要被信任：

上下文切换：在线程上下文之间切换需要保存和还原程序计数器和堆栈指针以及在保护和非保护模式之间可能会更改的进程状态。

内存映射的 I / O 和结构：处理器提供通过内存映射寄存器进行 I/O，因此内核需要将原始内存地址（例如 0x40008000）转换并放入类型化的寄存器和位字段的能力，而文件系统要求将磁盘控制块转换为结构体。

内存分配器：内核定义的特定内存分配器应当具有从裸指针转换到响应类型的能力。

用户空间缓冲区：因为用户空间可能传递无效的缓冲区，不安全的代码必须检查缓冲区有效。

中断/异常处理程序：处理程序本质上是不安全，因为它们会抢占运行代码，而内存可能处于不一致状态。

TakeCell：允许多个引用的抽象，像 Cell，但没有内存副本。

在其中，TakeCell 是独一无二的，因为它纯粹是设计用于允许高效，安全内核代码的软件抽象。Cell 适用于原始类型和较小值，对此，复制语义不会增加任何开销。在这些类型上，Cell 会优化成仅加载到一个寄存器并存储到内存中。TakeCell 适用于更大或更复杂的数据对象。TakeCell 是程序通过闭包传入的代码，而不是将值复制。例如，系统调用接口通过名为 App 的数据结构保留调用者/进程的状态，通过以下方式访问它：

```
struct App { /* many variables */ }
app: TakeCell<App>

self.app.map(|app| {
  // code can read/write app's variables
});
```

像 Rust 的 Cell 一样，TakeCell 在内部拥有共享的数据，例如可变的引用，并且多个调用者可以共享一个 TakeCell。但是，其 API 由 map（f）方法组成，而不进行向 TakeCell 之外的值复制。通常情况下，map（f）将调用闭包 f，传递 TakeCell 内部数据的引用。但是，如果已经存在对内部数据的引用，例如递归调用 map（f）是禁止操作，闭包不会被执行。

就结果而言，TakeCell 是互斥的一种形式。像互斥锁，TakeCell 确保只有一个可变引用指向其内部的值。但是，与互斥锁不同，它跳过操作而不阻塞。编译后，TakeCell 与未经检查的 C 代码一样快。例如，以下 TakeCell 代码段

```
struct App {
    count: u32,
    tx_callback: Callback,
    rx_callback: Callback,
    app_read: Option<AppSlice<Shared, u8>>,
    app_write: Option<AppSlice<Shared, u8>>,
}

pub struct Driver {
    app: TakeCell<App>,
```

```
}
driver.app.map(|app| {
    app.count = app.count + 1
});
```

生成了以下 ARM 汇编，可以安全地检查 app 是否是空指针，仅当它是具体值时对 app.count 进行操作，并存储结果：

```
/* Load App address into r1, replace with null */
ldr r1, [r0, 0]
movs r2, 0
str r2, [r0, 0]
/* If TakeCell is empty (null) return */
cmp r1, 0
it eq
bx lr
/* Non-null: increment count */
ldr r2, [r1, 0]
add r2, r2, 1
str r2, [r1, 0]
/* Store App back to TakeCell */
str r1, [r0, 0]
bx lr
```

注意，因为闭包被放到了栈上，所以不需要额外的内存分配。

A.5 实例探究

在本节中，我们在第 A.4节的基础上展示如何用 Rust 构建三个内核抽象。这些例子来自于我们的内核。内核的受信任代码基础还包括 Rust 核心库，在超过 6000 行代码中不超过 1000 行的 unsafe 代码。

A.5.1 直接内存访问 (DMA)

直接存储器访问（DMA）是今天内核中违规操作的常见来源。因为无论给定什么地址，硬件都会使用，使用 DMA 的内核代码可以规避虚拟内存和其他保护机制。

基于 Rust 的内核将内存映射的寄存器公开为类型化的数据结构。以这种方式安全地暴露它们，确保内核代码无法将任意值写入他们。例如，使用 Rust 切片的 DMA 接口（动态调整大小的数组）

```
struct DMAChannel {
    ...
    enabled: Cell<bool>,
```

```
    buffer: TakeCell<&'static mut [u8]>,
}
```

强制 buffer 字段是指向一个内存块的有效指针。此外，它可以使用缓冲区长度来确保它不会写到超出块的末尾的位置。对于传递&'static [u8] 的调用者，则必须已被授予该静态分配的字节缓冲区的访问权。该接口还突出 Rust 强制执行的一个有趣的安全问题。Rust 不能确定 DMA 操作需要多长时间，但是需要确保缓冲区在操作完成时仍将处于活动状态（未释放）。唯一可以满足这一要求的内存类型是静态（全局）分配的缓冲区和堆缓冲区，因此它要求缓冲区是静态的。DMA 实现中唯一的 unsafe 代码是将寄存器到内存中，将缓冲区和长度写入 DMA 寄存器并使能全部中断的代码。

A.5.2 通用串行总线

通用串行总线（USB）使用程序员指定的内存中的描述符来配置和控制 USB 端口。硬件假定这些描述符以一种特殊的方式被放置在内存中。在 Rust 中，用结构体表示这些硬件内存描述符是简单明了的。

下面的示例显示了依赖于需要特定布局和引用完整性两个级别的数据结构的硬件接口。USBRegisters.in_endpoints 是对端点描述符数组的引用。InEndpoint 精确地按照处理器希望这些描述符在内存中的布置，来设置内存布局。最后，EpCtl 类型定义了在编译时检查的有效值。值得的注意与指针不同，引用不能为空；当且仅当 dma_address 指向一个有效的的 DMADescriptor 时，一个 InEndPoint 能够存在；当且仅当 in_endpoints 指向一个有效的 16 个 InEndpoints 组成的数组时，一个 USBRegisters 能够存在。

```
enum EpCtl {
    ...
    Enable = 1 << 31,
    ClearNak = 1 << 26,
    Stall = 1 << 21
}
struct InEndpoint {
    control: Cell<EpCtl>,
    dma_address: Cell<&'static DMADescriptor>,
    ...
}
struct USBRegisters {
    ...
    // There can be 16 endpoints
    in_endpoints: Cell<&[InEndpoint; 16]>,
    ...
```

```
}
```

A.5.3 复杂数据结构

像缓冲区缓存、页表、文件系统这样的内核组件，通常依赖像双向链表、树这样带有循环引用的数据结构。这通常需要对同一可变数据使用多个别名，但是这些别名可以合乎逻辑。例如，缓冲区高速缓存条目引用磁盘块，但这被编码为设备 id 和扇区号，因此不需要管理 Rust 的所有权语义。数据结构使用双向指针的情况（例如双向链接列表）可以用与处理内核组件之间的循环依赖的相同原理来处理：Cell 和 TakeCell。例如：

```
struct ListLink<T>(Cell<Option<&T>>);
struct BufferHead {
  state: BufferState,
  next: &ListLink<BufferHead>,
  prev: &ListLink<BufferHead>,
  page: &Page,
  ...
}
```

A.5.4 多核

为了简洁起见，本文对在单线程设置中使用 Rust 进行了检验。支持多核需要系统在内核中管理并发。Rust 最初是为并行系统设计的。因此，它具有允许程序员安全地管理并发的语言机制。例如，Sync 关键字指定一种数据结构可以在线程和 Channel（一种跨线程传递数据的机制，仅允许 Sync 类型）之间安全地共享。多核 Rust 内核将使用这种机制来维护跨核心的共享数据的安全性。

A.6 后续工作

到目前为止，我们的工作讨论了内核机制和内核相关的驱动程序，面向低功耗单处理器的应用程序，例如 USB 应用程序和 DMA。我们相信其他操作系统组件，例如文件系统或视频缓冲区将能够使用类似的技术，尽管实际的实现工作中可能会发现更多的挑战。特别是，尽管我们展示了内存中的数据结构像链表和树可以在 Rust 中安全建模，未来的工作应该探索如何针对在磁盘上或在硬件中（例如页表）的数据结构进行建模。

此外，我们没有在多处理器设置中评估我们的设计。其他类型安全语言构建的系统（例如 Singularity），已成功地在语言环境中实现了并发计算。未来的工作

应该探索如何避免在并发服务中增长受信任的的计算基础。

最后，使用类型安全语言编写低开销内核，发掘了新颖应用程序在安全性和并发性上的潜力。尽管在这两个领域已有广泛的研究，他们很少被用于内核开发。我们尤其相信未来工作中一个有前途的领域是动态信息流控制。

信息流控制（IFC）[6] 是一种安全机制，可通过用安全标记来标记输入和输出，保证无干扰，并跟踪整个系统的标签。通常，基于 IFC 构建的系统可以使用静态标签 [7-8]，这些标签必须在编译时，或者通过牺牲粒度来最小化资源消耗 [9]。最近，有前途的工作 [10-12] 利用 Haskell 等功能语言的纯净性获得两全其美：动态的 IFC 标签以及最少内存开销。

在 Rust 内核中实施动态信息流控制是一个令人兴奋的前景。但是，虽然 Balasubramanian[13] 等人简要提出了一种基于 Rust 的静态 IFC 语言，尚不清楚这样的实现是否合理，甚至还不清楚线性类型是否足以代替纯度或效果系统来实施动态 IFC。未来的工作应该探讨这些原语是否可以在语言级别用 Rust 来强制执行，如果有的话，它如何影响内核设计。

A.7 总结

数十年的研究尝试将安全机制添加到操作系统内核中，但是在绝大多数实际系统中均告失败。纯语言技术可以缓解由于硬件强制内存隔离引起的性能和粒度问题。此外，使用没有垃圾收集器或其他运行时服务的类型安全语言（例如 Rust）可以避免原本应该是受信任代码库的最大部分的代码。之前的使用此类语言进行内核开发的努力，得到的结论是需要更改语言，我们发现该语言已经足够，只有一小部分不安全的抽象对于构建通用内核模块是必要的，希望能够实现下一代安全的操作系统。

参考文献

[1] LEVY A, CAMPBELL B, GHENA B, et al. The case for writing a kernel in rust[C/OL]// Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017. ACM, 2017: 1:1-1:7. https://doi.org/10.1145/3124680.3124717.

书面翻译对应的原文索引

[1] FELDMAN M, TAFT S T. Proceedings of the 2014 ACM sigada annual conference on high

integrity language technology, HILT 2014, portland, oregon, usa, october 18-21, 2014[C/OL]. ACM, 2014. https://doi.org/10.1145/2663171.

[2] LEVY A A, ANDERSEN M P, CAMPBELL B, et al. Ownership is theft: experiences building an embedded OS in rust[C/OL]//LU S. Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015. ACM, 2015: 21-26. https://doi.org/10.1145/2818302.2818306.

[3] TOV J A, PUCELLA R. Practical affine types[C/OL]//BALL T, SAGIV M. Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. ACM, 2011: 447-458. https://doi.org/10.1145/1926385.1926436.

[4] GROSSMAN D. Existential types for imperative languages[C/OL]//MÉTAYER D L. Lecture Notes in Computer Science: volume 2305 Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. Springer, 2002: 21-35. https://doi.org/10.1007/3-540-45927-8_3.

[5] LATTNER C, ADVE V S. LLVM: A compilation framework for lifelong program analysis & transformation[C/OL]//2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. IEEE Computer Society, 2004: 75-88. https://doi.org/10.1109/CGO.2004.1281665.

[6] MYERS A C, LISKOV B. Protecting privacy using the decentralized label model[J/OL]. ACM Trans. Softw. Eng. Methodol., 2000, 9(4):410-442. https://doi.org/10.1145/363516.363526.

[7] LOSCOCCO P, SMALLEY S. Integrating flexible support for security policies into the linux operating system[C/OL]//COLE C. Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA. USENIX, 2001: 29-42. http://www.usenix.org/publications/library/proceedings/usenix01/freenix01/loscocco.html.

[8] APPEL A W, AIKEN A. POPL '99, proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages, san antonio, tx, usa, january 20-22, 1999[C/OL]. ACM, 1999. http://dl.acm.org/citation.cfm?id=292540.

[9] ZELDOVICH N, BOYD-WICKIZER S, KOHLER E, et al. Making information flow explicit in histar[J/OL]. Commun. ACM, 2011, 54(11):93-101. https://doi.org/10.1145/2018396.2018419.

[10] BUIRAS P, VYTINIOTIS D, RUSSO A. HLIO: mixing static and dynamic typing for information-flow control in haskell[C/OL]//FISHER K, REPPY J H. Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. ACM, 2015: 289-301. https://doi.org/10.1145/2784731.2784758.

[11] RUSSO A, CLAESSEN K, HUGHES J. A library for light-weight information-flow security in haskell[C/OL]//GILL A. Proceedings of the 1st ACM SIGPLAN Symposium on Haskell,

Haskell 2008, Victoria, BC, Canada, 25 September 2008. ACM, 2008: 13-24. https://doi.org/10.1145/1411286.1411289.

[12] STEFAN D, RUSSO A, MITCHELL J C, et al. Flexible dynamic information flow control in haskell[C/OL]//CLAESSEN K. Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011. ACM, 2011: 95-106. https://doi.org/10.1145/2034675.2034688.

[13] BALASUBRAMANIAN A, BARANOWSKI M S, BURTSEV A, et al. System programming in rust: Beyond safety[C/OL]//FEDOROVA A, WARFIELD A, BESCHASTNIKH I, et al. Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017. ACM, 2017: 156-161. https://doi.org/10.1145/3102980.3103006.

附录 B　性能对比测试结果

表 B-1　zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
HandleCreate_Vmo.create	852	627	-26.41%
HandleCreate_Vmo.close	827	634	-23.34%
Vmo/Clone/128kbytes.clone	1595	834	-47.71%
Vmo/Clone/128kbytes.close	1293	2424	87.47%
Vmo/Clone/2048kbytes.clone	1475	884	-40.07%
Vmo/Clone/2048kbytes.close	1232	25102	1937.50%
Vmo/Clone/512kbytes.clone	1394	814	-41.61%
Vmo/Clone/512kbytes.close	1150	6649	478.17%
Vmo/Clone/ReadCloneAll/128kbytes	10032	16245	61.93%
Vmo/Clone/ReadCloneAll/128kbytes.clone	1336	876	-34.43%
Vmo/Clone/ReadCloneAll/128kbytes.read	7550	12779	69.26%
Vmo/Clone/ReadCloneAll/128kbytes.close	1146	2590	126.00%
Vmo/Clone/ReadCloneAll/2048kbytes	129816	281990	117.22%
Vmo/Clone/ReadCloneAll/2048kbytes.clone	1351	1247	-7.70%
Vmo/Clone/ReadCloneAll/2048kbytes.read	127073	255828	101.32%
Vmo/Clone/ReadCloneAll/2048kbytes.close	1392	24915	1689.87%
Vmo/Clone/ReadCloneAll/512kbytes	34346	65224	89.90%
Vmo/Clone/ReadCloneAll/512kbytes.clone	1372	843	-38.56%
Vmo/Clone/ReadCloneAll/512kbytes.read	31610	57582	82.16%
Vmo/Clone/ReadCloneAll/512kbytes.close	1364	6799	398.46%
Vmo/Clone/ReadCloneSome/128kbytes	4586	6833	49.00%
Vmo/Clone/ReadCloneSome/128kbytes.clone	1306	833	-36.22%
Vmo/Clone/ReadCloneSome/128kbytes.read	2223	3467	55.96%
Vmo/Clone/ReadCloneSome/128kbytes.close	1056	2533	139.87%
Vmo/Clone/ReadCloneSome/2048kbytes	36951	76866	108.02%
Vmo/Clone/ReadCloneSome/2048kbytes.clone	1342	857	-36.14%
Vmo/Clone/ReadCloneSome/2048kbytes.read	34457	52575	52.58%
Vmo/Clone/ReadCloneSome/2048kbytes.close	1152	23434	1934.20%
Vmo/Clone/ReadCloneSome/512kbytes	10741	20985	95.37%
Vmo/Clone/ReadCloneSome/512kbytes.clone	1307	828	-36.65%

续下页

续表 B-1　zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
Vmo/Clone/ReadCloneSome/512kbytes.read	8367	13574	62.23%
Vmo/Clone/ReadCloneSome/512kbytes.close	1067	6582	516.87%
Vmo/Clone/ReadOrigAll/128kbytes	9969	16400	64.51%
Vmo/Clone/ReadOrigAll/128kbytes.clone	1367	878	-35.77%
Vmo/Clone/ReadOrigAll/128kbytes.read	7473	12963	73.46%
Vmo/Clone/ReadOrigAll/128kbytes.close	1129	2558	126.57%
Vmo/Clone/ReadOrigAll/2048kbytes	128791	281173	118.32%
Vmo/Clone/ReadOrigAll/2048kbytes.clone	1379	1265	-8.27%
Vmo/Clone/ReadOrigAll/2048kbytes.read	126011	254932	102.31%
Vmo/Clone/ReadOrigAll/2048kbytes.close	1401	24977	1682.80%
Vmo/Clone/ReadOrigAll/512kbytes	34317	65101	89.70%
Vmo/Clone/ReadOrigAll/512kbytes.clone	1367	836	-38.84%
Vmo/Clone/ReadOrigAll/512kbytes.read	31575	57490	82.07%
Vmo/Clone/ReadOrigAll/512kbytes.close	1375	6775	392.73%
Vmo/Clone/ReadOrigSome/128kbytes	4615	6772	46.74%
Vmo/Clone/ReadOrigSome/128kbytes.clone	1314	829	-36.91%
Vmo/Clone/ReadOrigSome/128kbytes.read	2238	3485	55.72%
Vmo/Clone/ReadOrigSome/128kbytes.close	1063	2458	131.23%
Vmo/Clone/ReadOrigSome/2048kbytes	36573	78297	114.08%
Vmo/Clone/ReadOrigSome/2048kbytes.clone	1305	850	-34.87%
Vmo/Clone/ReadOrigSome/2048kbytes.read	34083	53873	58.06%
Vmo/Clone/ReadOrigSome/2048kbytes.close	1186	23573	1887.61%
Vmo/Clone/ReadOrigSome/512kbytes	11249	21082	87.41%
Vmo/Clone/ReadOrigSome/512kbytes.clone	1295	817	-36.91%
Vmo/Clone/ReadOrigSome/512kbytes.read	8868	13565	52.97%
Vmo/Clone/ReadOrigSome/512kbytes.close	1086	6700	516.94%
Vmo/Clone/WriteCloneAll/128kbytes	23373	16411	-29.79%
Vmo/Clone/WriteCloneAll/128kbytes.clone	1435	924	-35.61%
Vmo/Clone/WriteCloneAll/128kbytes.write	19798	13794	-30.33%
Vmo/Clone/WriteCloneAll/128kbytes.close	2139	1694	-20.80%
Vmo/Clone/WriteCloneAll/2048kbytes	3916443	270562	-93.09%
Vmo/Clone/WriteCloneAll/2048kbytes.clone	1729	1157	-33.08%
Vmo/Clone/WriteCloneAll/2048kbytes.write	3740093	254561	-93.19%
Vmo/Clone/WriteCloneAll/2048kbytes.close	15906	14844	-6.68%
Vmo/Clone/WriteCloneAll/512kbytes	85559	61669	-27.92%

续下页

续表 B-1　zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
Vmo/Clone/WriteCloneAll/512kbytes.clone	1416	871	-38.49%
Vmo/Clone/WriteCloneAll/512kbytes.write	79566	56697	-28.74%
Vmo/Clone/WriteCloneAll/512kbytes.close	4577	4101	-10.40%
Vmo/Clone/WriteCloneSome/128kbytes	6435	6572	2.13%
Vmo/Clone/WriteCloneSome/128kbytes.clone	1287	822	-36.13%
Vmo/Clone/WriteCloneSome/128kbytes.write	3774	3557	-5.75%
Vmo/Clone/WriteCloneSome/128kbytes.close	1374	2193	59.61%
Vmo/Clone/WriteCloneSome/2048kbytes	69305	74879	8.04%
Vmo/Clone/WriteCloneSome/2048kbytes.clone	1429	850	-40.52%
Vmo/Clone/WriteCloneSome/2048kbytes.write	62704	54928	-12.40%
Vmo/Clone/WriteCloneSome/2048kbytes.close	5173	19101	269.24%
Vmo/Clone/WriteCloneSome/512kbytes	18492	20048	8.41%
Vmo/Clone/WriteCloneSome/512kbytes.clone	1317	793	-39.79%
Vmo/Clone/WriteCloneSome/512kbytes.write	15079	13856	-8.11%
Vmo/Clone/WriteCloneSome/512kbytes.close	2096	5399	157.59%
Vmo/Clone/WriteOrigAll/128kbytes	24186	19959	-17.48%
Vmo/Clone/WriteOrigAll/128kbytes.clone	1379	866	-37.20%
Vmo/Clone/WriteOrigAll/128kbytes.write	20353	17072	-16.12%
Vmo/Clone/WriteOrigAll/128kbytes.close	2454	2021	-17.64%
Vmo/Clone/WriteOrigAll/2048kbytes	4095883	416495	-89.83%
Vmo/Clone/WriteOrigAll/2048kbytes.clone	1731	1253	-27.61%
Vmo/Clone/WriteOrigAll/2048kbytes.write	3886813	394192	-89.86%
Vmo/Clone/WriteOrigAll/2048kbytes.close	19175	21051	9.78%
Vmo/Clone/WriteOrigAll/512kbytes	90906	74240	-18.33%
Vmo/Clone/WriteOrigAll/512kbytes.clone	1424	867	-39.12%
Vmo/Clone/WriteOrigAll/512kbytes.write	83786	67976	-18.87%
Vmo/Clone/WriteOrigAll/512kbytes.close	5696	5397	-5.25%
Vmo/Clone/WriteOrigSome/128kbytes	6472	7074	9.30%
Vmo/Clone/WriteOrigSome/128kbytes.clone	1271	823	-35.25%
Vmo/Clone/WriteOrigSome/128kbytes.write	3825	3932	2.80%
Vmo/Clone/WriteOrigSome/128kbytes.close	1377	2319	68.41%
Vmo/Clone/WriteOrigSome/2048kbytes	72596	84531	16.44%
Vmo/Clone/WriteOrigSome/2048kbytes.clone	1455	902	-38.01%
Vmo/Clone/WriteOrigSome/2048kbytes.write	65173	60464	-7.23%
Vmo/Clone/WriteOrigSome/2048kbytes.close	5967	23165	288.22%

续下页

续表 B-1 zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
Vmo/Clone/WriteOrigSome/512kbytes	18857	22965	21.79%
Vmo/Clone/WriteOrigSome/512kbytes.clone	1346	830	-38.34%
Vmo/Clone/WriteOrigSome/512kbytes.write	15224	15472	1.63%
Vmo/Clone/WriteOrigSome/512kbytes.close	2287	6663	191.34%
Vmo/CloneMap/128kbytes.map	12608	4679	-62.89%
Vmo/CloneMap/128kbytes.clone	1594	3703	132.31%
Vmo/CloneMap/128kbytes.close	1158	2486	114.68%
Vmo/CloneMap/128kbytes.unmap	2350	3066	30.47%
Vmo/CloneMap/2048kbytes.map	168446	55210	-67.22%
Vmo/CloneMap/2048kbytes.clone	1727	36908	2037.12%
Vmo/CloneMap/2048kbytes.close	1162	23798	1948.02%
Vmo/CloneMap/2048kbytes.unmap	3715	35485	855.18%
Vmo/CloneMap/512kbytes.map	43974	14902	-66.11%
Vmo/CloneMap/512kbytes.clone	1632	10453	540.50%
Vmo/CloneMap/512kbytes.close	1174	6817	480.66%
Vmo/CloneMap/512kbytes.unmap	2545	9582	276.50%
Vmo/Read/128kbytes	6605	11595	75.55%
Vmo/Read/2048kbytes	116730	230751	97.68%
Vmo/Read/512kbytes	28967	52022	79.59%
Vmo/Write/128kbytes	5610	9394	67.45%
Vmo/Write/2048kbytes	107626	187399	74.12%
Vmo/Write/512kbytes	26988	41766	54.76%
VmoMap/Read/128kbytes	60092	20089	-66.57%
VmoMap/Read/2048kbytes	9104023	300831	-96.70%
VmoMap/Read/512kbytes	2321911	77405	-96.67%
VmoMap/Read/Kernel/128kbytes	51653	24532	-52.51%
VmoMap/Read/Kernel/2048kbytes	7649363	422970	-94.47%
VmoMap/Read/Kernel/512kbytes	1941181	96699	-95.02%
VmoMap/Write/128kbytes	59669	18042	-69.76%
VmoMap/Write/2048kbytes	8897544	276903	-96.89%
VmoMap/Write/512kbytes	2306391	71511	-96.90%
VmoMap/Write/Kernel/128kbytes	51260	23958	-53.26%
VmoMap/Write/Kernel/2048kbytes	7357273	382766	-94.80%
VmoMap/Write/Kernel/512kbytes	186350	97046	-47.92%
VmoMapRange/Read/128kbytes	29871	19734	-33.94%

续下页

续表 B-1 zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
VmoMapRange/Read/2048kbytes	3913832	298668	-92.37%
VmoMapRange/Read/512kbytes	1038261	76968	-92.59%
VmoMapRange/Read/Kernel/128kbytes	28944	24659	-14.80%
VmoMapRange/Read/Kernel/2048kbytes	4018803	423603	-89.46%
VmoMapRange/Read/Kernel/512kbytes	103420	96177	-7.00%
VmoMapRange/Write/128kbytes	26870	18248	-32.09%
VmoMapRange/Write/2048kbytes	3856962	276594	-92.83%
VmoMapRange/Write/512kbytes	98324	71228	-27.56%
VmoMapRange/Write/Kernel/128kbytes	27771	23969	-13.69%
VmoMapRange/Write/Kernel/2048kbytes	3727972	390326	-89.53%
VmoMapRange/Write/Kernel/512kbytes	96146	97129	1.02%
Channel/WriteEtcReadEtc/1024bytes/0handles	1099	1253	14.01%
Channel/WriteEtcReadEtc/1024bytes/0handles.write_etc	554	637	14.98%
Channel/WriteEtcReadEtc/1024bytes/0handles.read_etc	545	616	13.03%
Channel/WriteEtcReadEtc/1024bytes/1handles	1368	1456	6.43%
Channel/WriteEtcReadEtc/1024bytes/1handles.write_etc	682	733	7.48%
Channel/WriteEtcReadEtc/1024bytes/1handles.read_etc	687	723	5.24%
Channel/WriteEtcReadEtc/32768bytes/0handles	3833	2830	-26.17%
Channel/WriteEtcReadEtc/32768bytes/0handles.write_etc	2105	1359	-35.44%
Channel/WriteEtcReadEtc/32768bytes/0handles.read_etc	1728	1472	-14.81%
Channel/WriteEtcReadEtc/32768bytes/1handles	3769	3043	-19.26%
Channel/WriteEtcReadEtc/32768bytes/1handles.write_etc	1938	1483	-23.48%
Channel/WriteEtcReadEtc/32768bytes/1handles.read_etc	1830	1559	-14.81%
Channel/WriteEtcReadEtc/64bytes/0handles	1075	1193	10.98%
Channel/WriteEtcReadEtc/64bytes/0handles.write_etc	552	607	9.96%
Channel/WriteEtcReadEtc/64bytes/0handles.read_etc	523	586	12.05%
Channel/WriteEtcReadEtc/64bytes/1handles	1368	1464	7.02%
Channel/WriteEtcReadEtc/64bytes/1handles.write_etc	685	740	8.03%
Channel/WriteEtcReadEtc/64bytes/1handles.read_etc	682	723	6.01%
Channel/WriteEtcReadEtc/65536bytes/0handles	6302	4388	-30.37%
Channel/WriteEtcReadEtc/65536bytes/0handles.write_etc	3460	2069	-40.20%
Channel/WriteEtcReadEtc/65536bytes/0handles.read_etc	2842	2319	-18.40%
Channel/WriteEtcReadEtc/65536bytes/1handles	6036	4641	-23.11%
Channel/WriteEtcReadEtc/65536bytes/1handles.write_etc	3085	2226	-27.84%
Channel/WriteEtcReadEtc/65536bytes/1handles.read_etc	2950	2415	-18.14%

续下页

续表 B-1 zCore 与 Zircon 性能对比测试结果

Case	Zircon(ns)	zCore(ns)	zCore vs Zircon
Channel/WriteRead/1024bytes/0handles	1120	1253	11.88%
Channel/WriteRead/1024bytes/0handles.write	561	624	11.23%
Channel/WriteRead/1024bytes/0handles.read	559	629	12.52%
Channel/WriteRead/1024bytes/1handles	1299	1401	7.85%
Channel/WriteRead/1024bytes/1handles.write	654	710	8.56%
Channel/WriteRead/1024bytes/1handles.read	645	691	7.13%
Channel/WriteRead/32768bytes/0handles	3684	2826	-23.29%
Channel/WriteRead/32768bytes/0handles.write	2024	1351	-33.25%
Channel/WriteRead/32768bytes/0handles.read	1661	1475	-11.20%
Channel/WriteRead/32768bytes/1handles	3709	3008	-18.90%
Channel/WriteRead/32768bytes/1handles.write	1903	1453	-23.65%
Channel/WriteRead/32768bytes/1handles.read	1805	1555	-13.85%
Channel/WriteRead/64bytes/0handles	1048	1198	14.31%
Channel/WriteRead/64bytes/0handles.write	533	615	15.38%
Channel/WriteRead/64bytes/0handles.read	516	583	12.98%
Channel/WriteRead/64bytes/1handles	1161	1438	23.86%
Channel/WriteRead/64bytes/1handles.write	579	734	26.77%
Channel/WriteRead/64bytes/1handles.read	582	705	21.13%
Channel/WriteRead/65536bytes/0handles	6310	4368	-30.78%
Channel/WriteRead/65536bytes/0handles.write	3458	2060	-40.43%
Channel/WriteRead/65536bytes/0handles.read	2852	2307	-19.11%
Channel/WriteRead/65536bytes/1handles	6078	4519	-25.65%
Channel/WriteRead/65536bytes/1handles.write	3229	2156	-33.23%
Channel/WriteRead/65536bytes/1handles.read	2849	2363	-17.06%

综合论文训练记录表

<table>
<tr><td>学生姓名</td><td>潘庆霖</td><td>学号</td><td>2016011388</td><td>班级</td><td>计 61</td></tr>
<tr><td>论文题目</td><td colspan="5">zCore 的设计与实现</td></tr>
<tr><td>主要内容以及进度安排</td><td colspan="5">主要内容：
参考 Google 的 Fuchsia 操作系统的内核 zircon，用 Rust 对 zircon 的设计理念进行重新实现。

进度安排
第 03 周：完成部分 KernelOjbect 实现，能够运行 userboot
第 06 周：完全全部 KernelObject 实现，能够运行 user shell
第 09 周：上板调试
第 12 周： zCore 与 zircon 进行性能对比测试
第 14 周：尝试进行 zCore 的性能优化
第 16 周：完成论文

指导教师签字：
考核组组长签字：
2020 年 1 月 7 日</td></tr>
<tr><td>中期考核意见</td><td colspan="5">工作进展符合预期。

考核组组长签字：
2020 年 4 月 3 日</td></tr>
</table>

<table>
<tr><td>指导教师评语</td><td>潘庆霖的研究工作是研究以 zircon 新一代操作系统内核的设计思路，用 rust 语言设计并实现了以 object 为抽象基础的 zCore 操作系统内核。通过与其他同学的合作，潘庆霖深入分析了 zircon 的 C++实现，并在 KVM/物理硬件上设计实现了 zCore。在具体实现中，把 rust 的 async 机制引入到内核的 CPU 调度中，形成了无栈协程的底层调度与运行机制，对于一部分测试用例，提高了执行性能。并且通过对 zCore 中的 Unsafe Rust 分布进行了分析，说明了 Unsafe Code 的必要性和相对很小的比例，从一个侧面说明了 zCore的安全可靠性。通过zCore 与 Zircon 的简单性能比对，发现了 zCore 的部分性能瓶颈，进行了优化改进，部分测试用例的性能超过 Zircon。。论文相关的工作量大，有相当的难度，是一篇不错的论文。

指导教师签字：陈渝

2020 年 6 月 14 日</td></tr>
<tr><td>评阅教师评语</td><td>rust 语言是一种正在被内核开发人员重视和开始使用的新编程语言。尝试用异步机制来实现微内核操作系统，具有挑战和意义。作为微内核教学操作系统 zCore 的一部分，论文参考 Zircom，在 zCore 内核中设计和实现了基于 rust 的内核异步机制和基于 VDSO 的系统调用机制；并对 zCore 的系统调用支持情况和系统调用的性能进行了初步的分析。工作有一定新意。论文结构合理，写作规范，表述清楚，达到综合论文训练要求。

评阅教师签字：

2020 年 6 月 14 日</td></tr>
<tr><td>答辩小组评语</td><td>潘庆霖同学完成了综合论文训练预定计划，论文答辩过程中阐述清楚，回答问题正确。答辩小组同意通过论文答辩，并建议授予潘庆霖工学学士学位。

答辩小组组长签字：

2020 年 6 月 10 日</td></tr>
</table>

总成绩： B+

教学负责人签字：

2020 年 6 月 15 日