

清华大学

综合论文训练

题目: Rust 语言操作系统的设计
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: 王润基

指导教师: 陈康 副教授

2019年6月24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： 王润基 导师签名： 陈康 日 期： 2019.6.24

中文摘要

目前主流的操作系统大部分使用 C 语言编写。C 语言简洁高效，但容易产生内存和线程安全问题，已成为系统安全漏洞的主要来源。近年来，出现了一种新兴的系统级编程语言 Rust，它引入了所有权和生命周期等编译期检查机制，能够避免上述安全问题。并且，Rust 引入了现代编程语言的优良特性，提升了系统软件的开发效率和体验。这些使得 Rust 语言成为编写操作系统的理想选择。

本文中我们用 Rust 语言实现了一个完整的小型操作系统 rCore。它能够运行实际的 Linux 程序，支持四种指令集平台。本文从软件架构、子系统模块、系统调用和开发模式等方面介绍了 rCore 的设计实现过程。评估结果显示，rCore 在 Bug 数量、开发体验方面明显优于传统的 C 语言系统，但在性能上仍有少量差距。通过此次实践，本文还从各方面分析了 Rust 语言编写操作系统的能力，得出了它相比 C 语言更加适合编写操作系统的结论。

关键词：Rust 语言；操作系统

ABSTRACT

At present, most of the mainstream operating systems are written in C language. The C language is simple and efficient, but it is easy to cause memory safety and thread safety problems, which have become the main source of system security vulnerabilities. In recent years, a new system-level programming language, Rust, was born. It introduces compile-time checking mechanisms such as ownership and lifetime to avoid these safety problems. Moreover, Rust introduces the excellent features of modern programming languages and enhances the efficiency and experience of system software development. All of these make Rust an ideal choice for writing operating systems.

In this thesis, we implement a small but full-featured operating system rCore in Rust. It runs real world Linux programs and supports four instruction set architectures. We introduce the design and implementation of rCore from the aspects of software architecture, subsystem modules, system calls and development methods. The evaluation results show that rCore is significantly better than the traditional C language system in terms of the number of bugs and development experience, but there is still a small gap in performance. Through this practice, we also analyze the ability of Rust programming language to write operating system from various aspects, and draw the conclusion that it is more suitable to write operating system than C.

Keywords: Rust; Operating System

目 录

第 1 章 引言	1
1.1 研究动机	2
1.2 工作概述	5
1.3 相关工作	6
第 2 章 Rust 语言特性	9
2.1 内存安全和线程安全	9
2.2 接口和泛型	11
2.3 高级枚举类型和函数式风格	12
2.4 unsafe 代码和 C 语言互操作	12
2.5 模块系统和包管理机制	13
2.6 标准库和核心库	13
第 3 章 Rust 语言操作系统 rCore 的设计与实现	14
3.1 系统架构与项目结构	14
3.1.1 站在前人的肩膀上	14
3.1.2 类 Unix 的单体内核架构	15
3.1.3 低耦合的项目结构	16
3.2 构造可复用模块：以线程调度为例	17
3.2.1 内核线程与用户线程	18
3.2.2 内部数据结构和接口的设计	19
3.2.3 底层依赖与对外提供的接口设计	22
3.2.4 展望：与 Async 无栈协程机制的对比	24
3.3 Linux 系统调用层	26
3.3.1 Hello world：支持基于 musl libc 的 Linux 程序	28
3.3.2 Busybox + GCC：实现文件和进程相关系统调用	29
3.3.3 Nginx + Redis：实现网络功能	30
3.3.4 Rustc：实现多线程与同步互斥	32
3.4 小结：操作系统设计实现经验谈	36

3.4.1 迭代式开发	36
3.4.2 测试驱动开发	36
3.4.3 遵守行业标准	37
3.4.4 面向接口编程	37
3.4.5 持续推进重构	38
第 4 章 Rust 编写操作系统的能力分析.....	40
4.1 底层控制能力	40
4.2 抽象与优化能力.....	41
4.3 平台适配能力	42
4.4 高质量开发能力.....	43
第 5 章 结论	44
插图索引	45
表格索引	46
参考文献	47
致 谢	48
声 明	49
附录 A 外文资料的书面翻译.....	50

主要符号对照表

ABI	应用程序二进制接口
SMP	对称多处理
GC	垃圾回收
OS	操作系统
RAII	资源获取即初始化

第 1 章 引言

操作系统作为直接管理计算机软硬件资源的底层程序，在整个计算机系统中有着至关重要的地位。

目前主流的操作系统主要使用 C 语言编写。但是，由于 C 语言需要程序员手动管理内存，因此容易产生各种内存安全问题，如访问空指针、缓冲区溢出等，这些问题是现代操作系统内核中安全漏洞的主要来源。

近年来，诞生了一门全新的系统级编程语言——Rust。Rust 语言主打高性能、高可靠和高生产力。它引入了独特的语言机制，可以在编译期进行内存安全检查，突破性地解决了系统软件中的内存安全问题。同时它还具有现代语言中常见的编程范式，摒弃了历史上的糟粕特性。这些使它成为了编写操作系统的一个诱人选择。

为了检验 Rust 编写操作系统的能力，本工作历时一年时间，基于 Rust 社区的操作系统教程项目 BlogOS，参考清华大学教学操作系统 uCore OS，用 Rust 语言从零开始实现了一个完整的小型操作系统——rCore OS。目前 rCore 支持运行原生 Linux 程序，如 GCC、Nginx、Rustc 等，并且支持四种指令集平台：x86_64、RISC-V 32/64、ARM 64、MIPS 32，能够在物理硬件上运行。

另外，本工作得到了清华大学计算机系操作系统课教学团队的大力支持，已经在本科教学中进行应用推广，吸引了很多感兴趣的同学加入 Rust 语言操作系统的探索 and 开发。rCore 项目在开源平台 GitHub 上也获得了广泛关注和好评^①。

本工作的主要贡献有：

- 用 Rust 语言编写了一个完整的小型操作系统项目 rCore。
- 通过实践检验了 Rust 语言操作系统的可行性、可用性和优越性。
- 全面分析了在编写操作系统过程中，Rust 语言的能力和潜力。
- 总结了从零开始编写一个支持实际 Linux 程序的小型操作系统的过程和经
验。

^① <https://github.com/rcore-os/rCore>

截至 2019 年 6 月 24 日，rCore 主仓库共获得 204 stars 和 42 forks。

1.1 研究动机

目前主流的实用操作系统，如 Windows, Linux, macOS 等，主要使用 C 语言编写。C 语言于二十世纪七十年代伴随 Unix 操作系统而生，具有高效、简洁、可移植性强等特点。在之后的数十年中逐渐成为了使用最广泛的编程语言。直到今天，C 语言依然是编写系统软件的首选语言。

C 语言的内存安全问题

C 语言需要程序员手动管理内存，它的好处是可以对内存进行精确控制，但弊端是由于程序员疏忽导致的错误操作，会造成严重的内存安全问题。常见的问题有以下几种：

空指针 (Null Pointer) 访问一个空指针。原因往往是忘记初始化指针，或忘记指针已被置空。由于一般的程序在 0 地址不会映射页，因此访问空指针会立刻导致段错误。

野指针 (Wild Pointer) 和 **悬空指针 (Dangling Pointer)** 访问一个非法指针。一个未初始化的指针可能指向任何地方，被称为野指针。一个指针指向的内存已被释放并被别人使用，被称为悬空指针。野指针和悬空指针所指向的数据都是无效的。访问它们可能导致段错误，也可能毫无征兆地修改程序的某处数据，并在继续运行一段时间后产生神秘的问题。

重复释放 (Double Free) 同一个指针被连续释放两次。原因可能是释放指针后没有清零，之后再次释放。一般会导致未定义行为。

越界访问 (Out of Range) 访问一片连续内存时超出了边界。原因是没有对访问范围进行边界检查。其中一个著名的表现是缓冲区溢出 (**Buffer Overflow**)，即允许输入数据写到缓冲区之外的位置覆盖掉其它数据，从而破坏系统状态并夺取系统控制权。这是 C 语言程序中常见的安全漏洞，被黑客利用成为广泛使用的攻击手段。

数据竞争 (Data Race) 多线程并发读写共享内存。这个问题在多线程程序中非常常见，原因是没有正确地对共享数据进行加锁，或对线程进行合理的同步。数据竞争会造成共享数据损坏，进而导致未定义行为。

Linux 系统使用 C 语言编写，因此也广受这些安全问题的危害。一份 2018 年对 Linux 内核的漏洞分析表明，一半以上的安全漏洞是由以上问题导致的^[1]。

自动化管理内存

正是由于手动管理内存带来的巨大安全隐患，使得更高层的编程语言几乎都是自动管理内存，不同程度上避免了上述问题。但它们所使用的垃圾回收机制（Garbage Collection）会周期性地暂停整个程序的运行，并且清理内存时有很大的开销，这对于实时性和性能要求很高的操作系统而言是不可接受的。

还有一种半自动的内存管理方式，例如 C++ 语言的 RAII 惯用法^①。它引入了生命周期的概念，让资源的生命周期与一个对象的生命周期绑定：在对象构造时获取资源，在对象析构时释放资源。由语言保证所有对象一定会被构造和析构，从而实现了资源（包括内存资源）的自动化分配和释放。C++ 的 RAII 惯用法及其实现的智能指针，很大程度上解决了程序中内存管理的需求，同时又避免了垃圾回收带来的性能开销，是系统级软件管理内存的优秀解决方案。

C 语言的语言特性匮乏

除开内存管理，C 语言的另一大短板是语言特性的匮乏。我们必须承认，C 语言的设计短小精悍，语言特性简单直接，同时对底层有强大的控制能力。但是，它毕竟是一个有着近 50 年历史的古老语言。从现代编程语言的角度看，它仅支持简单的过程式编程，抽象和表达能力不足：

缺少面向对象和接口的语言级支持 面向对象编程符合人们的思维方式，已经成为现代编程语言的标配。但在 C 语言中只能用 `struct` 结构体来实现对象，无法方便地调用成员函数。另一个重要的特性——接口，也就是动态的函数调用，C 语言中只能自己定义函数指针的结构体（也就是虚函数表）来实现。

缺少必要的标准库基础设施 例如高级语言中最基本的容器类型变长数组 `vector`，在 C 语言标准库中就不存在。这直接导致 C 语言程序经常要自己实现侵入式链表^②充当容器，这在 Linux 的代码中体现得尤其明显。这不仅让代码复杂难懂，而且可能导致缓存命中率降低，从而影响性能。

缺少模块和包管理机制，编译配置复杂 现代编程语言都有统一的编译工具链，具有中心化的软件包仓库，可以方便地进行模块导入和发布，极大促进了代码复用。而 C 语言受制于沉重的历史包袱，至今没有标准的编译工具和模块系统。使得在开发时，难以复用现成的代码，提高了开发成本。

^① 全称 Resource Acquisition Is Initialization，资源获取即初始化

^② “侵入”是指把链表节点嵌入在数据节点中

由于上述 C 语言的局限性，也有一些操作系统全部或部分使用 C++ 语言编写，例如 Windows 和 Fuchsia^①。C++ 具有更丰富的语言特性和极强的零开销抽象能力，按说很适合写操作系统。但它的语言特性过于繁杂，潜在的问题更多，对程序员的心智要求很高，不是经验丰富的开发人员很难驾驭的了。因此，在大型 C++ 项目中都会禁用部分语言特性，C++ 也未在主流操作系统中被广泛应用。

此外，目前国内外高校的操作系统教学也是以 C 语言为主。在平时的学习中，我观察到很多同学对操作系统敬而远之，原因就是程序逻辑复杂、细节琐碎，C 语言过于底层、很容易出 Bug，并且调试起来异常困难。可以说，编程语言是制约系统软件开发体验的重要因素。

Rust 语言的诞生

为了解决系统软件开发的种种问题，Mozilla 研究院于 2010 年推出了 Rust 语言。与 Rust 相伴而生的是名为 Servo^② 的实验性并行浏览器引擎项目。得益于 Rust 语言的特性，Servo 引擎取得了更好的性能和安全性，目前已经成为 Firefox 浏览器内核的一部分。Rust 语言于 2015 年发布了第一个稳定版本，此后逐渐获得了更多的关注和应用。

Rust 语言的杀手级特性是引入了所有权、生命周期和借用机制，从而在编译期进行内存安全检查，避免出现空指针、野指针、悬空指针。针对多线程的数据竞争问题，Rust 为对象引入了 Send 和 Sync 标记，限制它们是否允许跨线程传递和多线程访问，实现了线程安全。本文会在第二章中进一步介绍 Rust 语言的语法和核心特性。

Rust 语言与操作系统

内存安全和线程安全，是操作系统内核的核心诉求。随着单核处理器逐渐达到技术瓶颈，现代 CPU 正在向多核发展，充分利用多处理器资源也成为了现代操作系统的基本要求。在这方面，相比毫无安全保护的 C 语言，Rust 语言有着天然的优势。

历史上，不乏有用其它语言编写操作系统的尝试，但都难以挑战 C 语言的统治地位。目前也已经有不少使用 Rust 语言的尝试，但大部分都是玩具项目，不能运行实际的应用程序。并且国内 Rust 语言的生态也才刚刚起步，还没有写操作系统的先例。本工作的目的不是做出多么大的创新，而是从头开始实践 Rust

^① Google 公司正在开发的全新操作系统，全部用 C++ 编写

^② <https://servo.org>

语言操作系统的开发过程，全面完整地总结经验，弥补国内在相关领域的空白，起到普及推广 Rust 语言和计算机系统知识的效果。

1.2 工作概述

本文的工作是，从零开始用 Rust 语言编写了一个小型操作系统 rCore。

rCore 是一个类 Unix 的单体内核系统。它支持运行原生 Linux 程序：从最基本的实用工具 Busybox，到多进程的 C 语言编译器 GCC，再到网络服务器 Nginx 和内存数据库 Redis，最后是多线程的 Rust 编译器 Rustc。

rCore 目前支持四种指令集，均能够在物理硬件上运行，并在部分平台上支持多核处理器。

表 1.1 rCore 支持的指令集和物理硬件

指令集	物理硬件
x86_64	PC 机（4 核），Intel Xeon 服务器（64 核）
RISC-V 32/64	HiFive Unleashed ^① ，Kendryte K210 ^② ，Rocket Chip FPGA ^③
ARM64	树莓派 3B+
MIPS32	TrivialMIPS ^④

① SiFive 公司生产的可以运行 Linux 的 RISC-V 处理器。

<https://www.sifive.com/boards/hifive-unleashed>

② 国产嵌入式处理器。<https://kendryte.com>

③ 陈嘉杰修改过的开源 CPU。<https://github.com/jiegec/fpga-zynq>

④ 陈晟祺等实现的 FPGA CPU。<https://github.com/Harry-Chen/TrivialMIPS>

本工作开题之初，我首先参考了 Rust 社区的 *Writing an OS in Rust*^[2]，这是一系列从零开始用 Rust 编写操作系统的教程，作者伴随文章实现了一个名为 BlogOS^③ 的简单内核。它运行在 x86_64 平台上，支持输出信息到 VGA 和串口、处理中断和异常、简单修改页表。在 BlogOS 代码基础上，我参考清华大学教学操作系统 uCore，实现了基础的内存管理、进程调度、文件系统功能，能够运行 uCore 的用户程序。随后我又将它移植到了 RISC-V 32 指令集上。以上工作由本人独立完成，基本证实了 Rust 语言操作系统的可行性。

此后，陆续有更多同学加入到了这个项目的开发中来，实现了更多功能特性。在底层适配方面，我们将其移植到了更多的指令集和硬件平台上，优化了系统架构和跨平台能力。在上层接口方面，我们确定了兼容 Linux 的开发路线，在

③ https://github.com/phil-opp/blog_os

选好了要支持的目标应用程序后，逐一实现它们依赖的系统调用，在此过程中补充和完善内核的功能。在大家的共同努力下，rCore 已经发展成为一个功能丰富、支持广泛的小型操作系统，证实了 Rust 语言操作系统的可用性。

表 1.2 rCore 后续实现的重要特性及其开发者

功能特性	开发者
多核的启动和调度	王润基
移植到 RISC-V 64	戴臻旻, 王润基
移植到 ARM64 (树莓派)	贾越凯
移植到 MIPS32	陈晟祺, 周聿浩
支持 Linux musl libc 程序	王润基
网卡驱动, 网络协议栈和 Socket 接口	陈嘉杰
内核可加载模块	郭敬哲

最后，我们对 rCore 进行了初步的性能测试，与 Linux 和 Go 语言操作系统 Biscuit 进行了对比分析。我们还总结了开发操作系统过程中遇到的经验教训，并分析了 Rust 编写操作系统的优势和潜在问题。我们希望通过 rCore 项目的实践过程，展示 Rust 语言的优越性，从而吸引更多人加入 Rust 和操作系统的探索和开发中来。

1.3 相关工作

玩具和教程项目

自 Rust 语言诞生以来，用它编写操作系统的尝试就从未停止。2013 年，Rust 还处于测试版时，zero.rs^① 项目就开始探索用 Rust 编写裸机程序。2014 年，rust-boot^② 项目受到它的启发，实现了第一个可在 QEMU 模拟器中运行的内核程序。此后，用 Rust 语言实现的内核如雨后春笋般涌现，先后诞生了 intermezzOS^③、Tifflin^④ 等源于个人兴趣的实验性操作系统内核。

2015 年，随着 Rust 1.0 版本的发布，Philipp Oppermann 在个人博客上发表了 *Writing an OS in Rust*^⑤ 的第一篇文章。这是一个手把手教你用 Rust 写操作系统的

① <https://github.com/pcwalton/zero.rs>

② <https://github.com/charliesome/rustboot>

③ <https://github.com/intermezzOS>

④ https://github.com/thepowersgang/rust_os

⑤ <https://os.phil-opp.com>

系列教程，作者在文章中逐步实现了一个基于 x86_64 的简单内核——BlogOS^①。此后，作者对它持续更新，并陆续开发了 x86_64, bootloader, bootimage 等^②方便 Rust 写 OS 的实用工具和软件包。目前，这个教程已经更新到第二版。由于文章和代码质量很高，加之作者的持续投入和改进，这一项目受到 Rust 社区和读者们的广泛认可，成为了 Rust 语言编写操作系统的必读之作。

教学操作系统

除了个人项目以外，也有不少将 Rust 编写操作系统用于高校教学的尝试。最早的应用是在美国弗吉尼亚大学 2013 年秋季学期的操作系统课上。在课程结束时，学生们对 Rust 语言的评价两极分化，但大部分都给予了正面评价。教学团队在课程评估中指出了将 Rust 用于系统教学的两大问题：语言不成熟，学习曲线陡峭。^[3]站在 2019 年的时间点来看，第一个问题已经不复存在，而在第二个问题上，虽然官方一直在持续改进语言的友好度，但 Rust 语言的高难度依然是初学者面临的一大障碍。

2015 年，美国布朗大学的一名本科生将他们的教学操作系统 Weenix 用 Rust 语言重新实现，并命名为 Reenix^③。和清华大学的 uCore 类似，Weenix 也是用 C 语言编写的类 Unix 系统。作者在他的本科毕业论文中记录了新系统的实现过程，并分析了遇到的挑战：包括结构体不能继承、生命周期的困扰、堆内存分配失败等，作者还对 Rust 与 C 语言系统的性能进行简单对比，结果是平均要慢 2-3 倍左右。^[4]

2018 年，美国斯坦福大学新开设了代号 CS140e 的实验性操作系统课程^④，用 Rust 编写系统并在树莓派 3B 上运行。这门课其中一位讲师是 Rust 社区的专家，因此课程实验代码质量较高。值得一提的是，这个课程在国内社区也受到了关注^⑤，不少爱好者同步跟进了课程学习。不过遗憾的是，由于时间所限，课程实验缺失了内存管理部分，并且这门课在 2019 年又换回了 C 语言。

严肃的社区项目

Rust 社区也十分重视操作系统领域的应用。目前社区中完成度最高的系统是 Redox^⑥ 项目。Redox 是一个类 Unix 的微内核系统，首发于 2015 年，到如今

① https://github.com/phil-opp/blog_os

② <https://github.com/rust-osdev>

③ <https://github.com/scialex/reenix>

④ <https://cs140e.sergio.bz>

⑤ 如何评价斯坦福的新操作系统课程 CS140e? <https://www.zhihu.com/question/265653828>

⑥ <https://www.redox-os.org>

已经建立了庞大的生态，包括 `libc`、GUI、文件系统等，目前维护者有近 40 人。`Redox` 项目代表了 Rust 语言在操作系统上的最前沿成就，是我们前进的标杆。

相比操作系统，Rust 社区更注重在嵌入式领域的应用。在这方面，有一个用 Rust 编写的嵌入式系统——`Tock`^①项目。嵌入式设备资源有限，注重实时性和安全性，因此 `Tock` 项目提出了 `Capsule`（胶囊）内核模块机制，并将每个用户进程所用的内核内存集中存放，起到隔离的效果。`Tock` 项目的上述机制都充分利用了 Rust 语言特性，来对内核代码进行限制，以保证安全可靠。^[5]

其它高级语言操作系统

和 Rust 同时期诞生的还有 Google 推出的 Go 语言。Go 语言的核心特性是名为 `goroutine` 的用户态协程机制，可以轻松地支持并发编程。Go 是一门带有垃圾回收的语言，其它特性类似增强版的 C 语言。

Go 诞生后也被用来编写操作系统，不过它有一大阻碍是垃圾回收和 `goroutine` 带有不小的语言运行时。2017 年，`Gopher`^②项目诞生，验证了 Go 语言操作系统的可行性。不过它仅仅是个雏形，并且禁用了以上两个语言特性。

2018 年，MIT 公布了 `Biscuit`^③ 项目，这是一个用 Go 编写的完整 POSIX 兼容系统，能够在它自己定制的 `libc`^④ 上运行 `Nginx` 和 `Redis` 等实际程序。`Biscuit` 的论文主要探讨了操作系统中合理使用垃圾回收机制的方法，指出了 Go 语言内存安全的好处，并详细测试了为了这些安全所要付出的代价：相比 C 语言性能下降 10% 左右。^[6]

`Biscuit` 项目为我们的 Rust 系统提供了很好的参考和对比目标：它为我们列举了支持 `Nginx` 和 `Redis` 所必须的 58 个系统调用，为我们支持实际应用程序指明了方向。另外，从语言上讲，Rust 不需要 Go 的垃圾回收和 `goroutine` 机制。如果 Go 语言都可以做到只损失 10% 的性能，那 Rust 一定可以在性能上做得更好。

① <https://www.tockos.org>

② <https://github.com/achilleasa/gopher-os>

③ <https://github.com/mit-pdos/biscuit>

④ 一个 3000 行的精简实现，称为 `litc`

第 2 章 Rust 语言特性

本章列举和简单介绍 Rust 重要的语言特性，并概述它们在编写系统软件中的作用。

Rust 是一门注重性能、可靠性和生产力的系统级编程语言^[7]。它通过独特的所有权和生命周期机制，保证内存安全和线程安全，实现高可靠性。Rust 集成了现代编程语言的优良特性，同时支持面向对象和函数式编程范式，拥有友好的编译工具链，大幅提高了系统编程的效率。借助强大的接口和泛型机制，Rust 实现了零开销抽象，保证程序的高性能。

2.1 内存安全和线程安全

Rust 继承并发扬了 C++ 语言中的 RAII 模式，引入了所有权 (*ownership*) 的概念，并在此基础上建立了引用 (*reference*) 和借用 (*borrow*) 规则。此外，每个所有者变量和引用变量都有自己的生命周期 (*lifetime*)。引用变量的生命周期不能超过其引用目标，这样就避免了悬空引用。

具体而言，所有权和借用机制有如下规则^[8]：

所有权规则 每一个值 (*value*) 唯一地对应一个所有者 (*owner*) 变量。每个变量都有自己的作用域 (*scope*) 范围。当所有者变量离开作用域时，其值将被丢弃。一个值的所有权可以从一个变量移动到另一个变量，移动完成后原有的变量将不能访问。

引用规则 一个所有权变量 (*T*) 在任何时刻，要么有一个可变引用 (`&mut T`)，要么有多个不可变引用 (`&T`)。代码可以通过引用使用值，但不能获取它的所有权。引用在任何时刻总是有效的。

下面我们借用一张图来直观地理解这些规则：

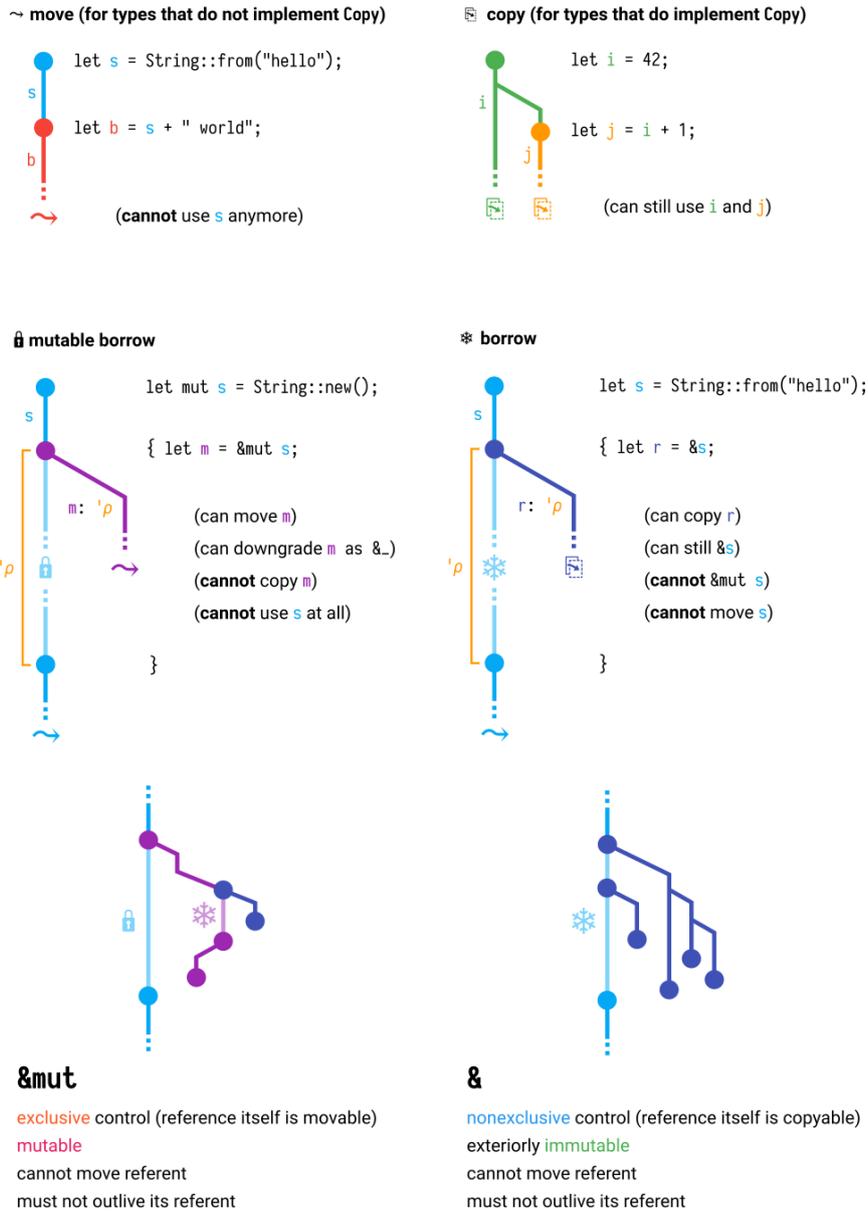


图 2.1 Rust 所有权和借用模型[©]

- 图 1: 移动语义。所有权从变量 `s` 移动到变量 `b`。
- 图 2: 复制语义。对于实现 `Copy` 特性类型的变量，会自动复制。
- 图 3: 可变借用。变量 `m` 可变借用 `s`。
- 图 4: 不可变借用。变量 `r` 不可变借用 `s`。
- 图 5: 可变引用。
- 图 6: 不可变引用。

上述机制基本解决了单线程环境下的内存安全问题。但到了多线程中，我们需要对共享变量增加更多的约束，以防止数据竞争。Rust 语言引入了两个标

记特性（marker trait）：`Send` 和 `Sync`。

Send 表明类型的所有权可以安全地在线程间传递。

Sync 表明可以安全地在多个线程中拥有其值的引用。

以上定义可能有些抽象，下面我们列举一些推论来帮助理解这两个特性：

- 只有满足 `Sync` 的类型才能定义全局变量^③
- 只有满足 `Send` 的类型才能被移动到闭包函数内^④
- 锁类型 `Mutex<T>` 是满足 `Send` 和 `Sync` 的^⑤
- 对于任意类型 `T`，如果 `&T` 满足 `Send`，那么 `T` 满足 `Sync`^⑥
- 如果一个类型的所有成员变量都满足 `Send / Sync`，那么这个类型就自动满足 `Send / Sync`^⑦
- 手动标记类型为 `Send / Sync` 是不安全的^⑧

最后我们总结一下 `Rust` 语言如何避免各种安全问题：

表 2.1 `Rust` 语言保证内存安全和线程安全的机制

安全问题	<code>Rust</code> 语言的应对机制
空指针，野指针	不存在未初始化的引用
悬空指针	生命周期机制保证引用目标一定存在
重复释放	仅在所有权的生命周期结束时释放内存
越界访问	运行时边界检查
数据竞争	<code>Send</code> 和 <code>Sync</code> 标记

2.2 接口和泛型

`Rust` 具有部分面向对象的编程范式，并在其中属于面向协议（`prototype-based`）的派别，即关注类的公共行为而不是它们的继承关系。

`Rust` 使用 `struct` 定义一个类，类有字段（`field`）和方法（`method`）。每个类可以有析构函数（使用 `Drop` 特性实现），但没有特殊的构造函数，构造函数都用普通静态函数实现。和 `Go` 语言一样，`Rust` 摒弃了“继承”特性，避免了它带来

③ 全局变量是可以被多线程引用的。

④ 闭包函数可以被任意线程执行，因此跨越闭包的变量必须可以安全地在线程间传递。

⑤ 锁保证了内部变量一定被独占访问。

⑥ 在线程间传递引用就意味着它被共享。

⑦ 比较显然的传递性质。

⑧ 需要程序员自己保证满足相关性质，不会发生数据竞争。

的复杂性。

作为面向协议的体现，Rust 使用 特性 (*trait*) 表示共享行为，类似其它语言中接口 (*interface*) 的功能。trait 有两种使用方式：基于 特性对象 (*trait-object*) 的动态派发，和基于泛型的静态派发。与 C++ 类比：前者是虚函数调用，有运行时开销；后者是模板，在编译期生成代码，没有运行时开销。在泛型中我们需要使用特性约束 (*trait bound*) 来描述泛型类的行为，类似 C++20 中 *Concept* 的概念。

2.3 高级枚举类型和函数式风格

Rust 还引入了部分函数式编程特性。这些特性提高了语言的表达能力，使之更加符合人的思维方式。

相比 C 语言，Rust 中的枚举 (*enum*) 类型中不仅有种类，而且每个种类还可以附带一些值，这种设计也被称作代数数据类型。对于这种类型，一般通过模式匹配的方式处理它的各种不同情况，它有一个好处是可以保证不会遗漏任何一种情况。

Rust 不允许出现空指针，取而代之的是一个名为 *Option* 的枚举类型，可以表示有值 (*Some*) 或空值 (*None*)。类似地，Rust 中没有抛出异常的机制，而是像 C 语言一样通过返回值表示错误。Rust 中使用名为 *Result* 的枚举类型描述一个可能是错误的返回值。错误处理过程是：外层函数对其进行模式匹配，检测错误并继续向外传递。并且这一过程可以配合 ? 语法优雅地表达出来。

Rust 语言支持创建闭包 (*closures*)，也就是可以捕获环境中变量的匿名函数。有了闭包的支持，Rust 实现了强大的迭代器 (*iterator*)，配合 *map* 和 *filter* 等内建函数，我们就可以方便地对序列数据进行处理。这些额外的小函数都会被 Rust 强大的编译器优化掉，从而达到和手写循环语句相同甚至更好的性能。

2.4 unsafe 代码和 C 语言互操作

在编写系统软件尤其是操作系统时，需要直接读写内存的操作，这在正常的 Rust 代码中是不可实现的。为此 Rust 引入了 *unsafe* 语句块，可在其中绕过编译器限制，通过裸指针直接访问内存。强制的 *unsafe* 语句起到了对安全和不安全的代码进行隔离的目的。在 Rust 语言哲学中，安全与不安全是对立统一的特性。我们应尽量避免 *unsafe* 代码，并将其封装成安全的接口。

在用新语言编写操作系统时，另一个很实际需求是与现有 C 语言代码进行互操作。类似 C++，Rust 支持使用 `extern "C"` 语句导出 Rust 函数和导入外部 C 语言函数。Rust 在内部使用自己的 ABI，当与 C 语言交互时，需要对结构体添加 `#[repr(C)]` 标注，以使用 C 语言标准的内存布局。显然，Rust 无法保证外部函数的安全性，因此所有导入函数都是 `unsafe` 的。

2.5 模块系统和包管理机制

Rust 语言使用模块对代码进行组织归类，一般而言一个文件就是一个模块，模块之间的层级关系通过文件夹的树状结构来体现。在模块中，默认所有的函数和结构体都是内部的，仅在模块内可见。如需对外暴露，则要加 `pub` 关键字修饰。引用其它模块需要在文件头部显式声明。

一个 Rust 的二进制或库项目称为一个包 (*crate*)。包是代码复用的最小单位。Rust 具有中心化的包发布平台和文档平台，配合官方发布的包管理工具 `cargo`，能够自动完成包的下载、更新和发布。

2.6 标准库和核心库

Rust 语言提供一个精简的标准库 `std`，其背后依赖 `libc` 完成和系统相关的功能。此外，Rust 还提供了一个核心库 `core`，仅包含和语言特性紧密相关的实现，可用于编写裸机程序。我们的 `rCore` 操作系统就是依赖核心库编写的。

由于 Rust 语言的一个主要应用场景就是编写嵌入式程序，官方也成立了嵌入式工作组来促进相关基础设施的建设。因此在软件包平台上有相当多的 `no_std` 包可供我们使用。

第 3 章 Rust 语言操作系统 rCore 的设计与实现

本章中我们将详细介绍 rCore 的设计思路和实现过程。

首先我们在 3.1 中介绍 rCore 的历史背景和整体系统架构，描述如何利用 Rust 语言特性实现更好的软件结构。之后在 3.2 中我们以线程调度为例，介绍 rCore 中一个完整可复用模块的设计实现过程。接下来的 3.3 节，在这些基础功能模块的基础上，我们逐步实现了 Linux 系统调用层，让 rCore 支持了各种实际的应用程序。最后，在 3.4 中我们将回顾 rCore 的设计实现过程，总结和梳理小型操作系统项目的开发经验。

3.1 系统架构与项目结构

3.1.1 站在前人的肩膀上

rCore 的命名清晰地表明它是用 Rust 语言重新实现的 uCore 系统。这在项目启动之初确实是我们全部的努力方向，但如今我们欣喜地看到，它所达到的高度已经逐渐超越了这一目标。应该说，rCore 是脱胎于 uCore 的一个更为先进的系统。

uCore^① 是清华大学操作系统课中所使用的教学系统。它基于 MIT 的 xv6^②，以及哈佛大学的 OS/161^③ 教学操作系统，并参考了 Linux 的部分设计。它于 2012 年由王乃峥同学实现，随后经过了一批批学生的使用和改进，目前已经实现了丰富的功能并支持诸多指令集平台。^[9]

在使用 Rust 语言编写系统之初，Philipp Oppermann 的 BlogOS 是一个极好的起点：它已经完成了最艰难的 boot 工作，为 Rust 代码建立了良好的执行环境，并且每一步都有详细的文档说明。鉴于此，rCore 直接基于 BlogOS 第一版第 10 章的代码^④完成后续工作。图 3.1 显示了 rCore 系统的参考和继承关系。

① https://github.com/chyyuu/ucore_os_lab

② <http://pdos.csail.mit.edu/6.828/xv6>

③ <http://os161.eecs.harvard.edu>

④ https://github.com/phil-opp/blog_os/tree/first_edition_post_10

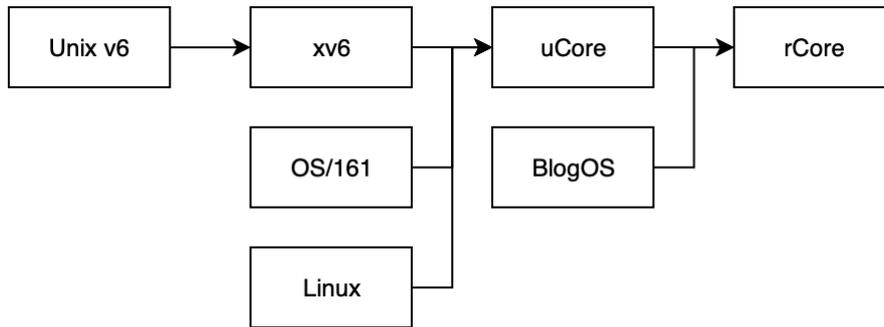


图 3.1 rCore 系统的参考和继承关系

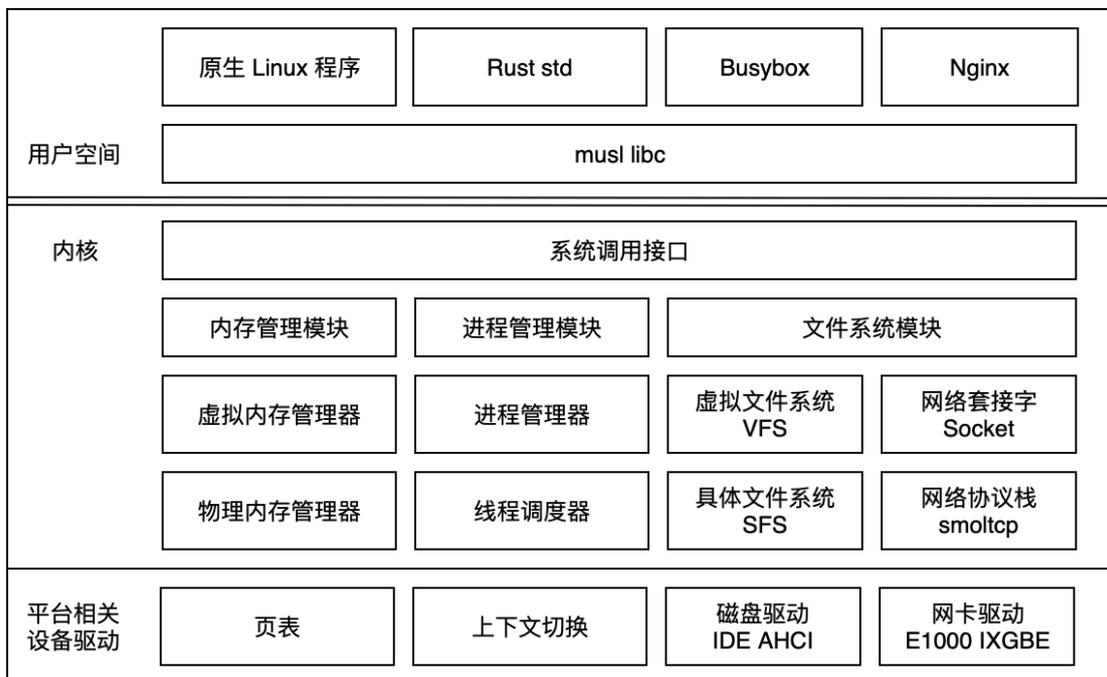


图 3.2 rCore 的系统架构

3.1.2 类 Unix 的单体内核架构

rCore 的主要参考对象 uCore 是一个用 C 语言编写的单体内核系统。包括内存管理、进程管理、文件系统三大子系统。其中内存管理又分为以连续内存分配为主的物理内存管理，和以页为单位分配的虚拟内存管理；进程管理又分为内核线程的调度和用户进程的管理；文件系统又分为虚拟文件系统（*Virtual File System*）和具体实现的简单文件系统（*Simple File System*）^①。

rCore 基本继承了 uCore 的架构，并在此基础上进行了完善和改进，如图 3.2 所示。在系统的最底层，对于不同的指令集，rCore 总结出一套最小化的平台相关接口。在内核平台无关层，rCore 增加了网络协议栈并将之封装成 Socket 接口，

① 继承自 OS/161

它进一步与虚拟文件系统的 `Inode` 接口一起，封装成文件提供给用户程序。在系统调用层面，`rCore` 主要针对 Linux 的 `musl libc`^① 实现了它所用到的接口。因此在用户空间，`rCore` 理论上支持运行所有使用 `musl libc` 编译的 C 语言程序（静态或动态链接均可）。

3.1.3 低耦合的项目结构

操作系统内核的功能模块众多，且各部分之间互相依赖，耦合度极高。雪上加霜的是，传统的 C 语言抽象能力低下，代码难以泛化和复用。并且由于没有统一的包管理工具和发布平台，C 语言几乎没有软件包生态，形成了“一切轮子都要自己造”的局面。一个著名的例子是 Linux 系统。作为世界上规模最大的单体内核系统，Linux 代码复杂难以阅读，各模块间藕断丝连，是一个难以触动的庞然大物。

在我们的项目实践中，`rCore` 充分探索和利用了 Rust 的语言特性，致力于优化项目结构，降低各部分代码间的耦合性。一言蔽之，我们希望编写操作系统能拥有和编写普通用户程序一致的体验，包括方便的调试、全面的测试和灵活地拆分组合。具体方式体现为以下几个方面：

大量使用外部软件包 Rust 包发布平台上有大量可用于裸机程序的软件包。此外 Philipp Oppermann 在编写 BlogOS 的过程中，也发布了很多专用于编写操作系统的实用工具。`rCore` 充分利用了这些资源，不仅免去了“重复造轮子”的过程，而且由于它们使用人数众多，质量更有保障。表 3.1 中列举了 `rCore` 主要依赖的一些软件包。

最小化平台相关代码 `rCore` 对平台相关和无关代码进行了严格划分。相比 `uCore`^②，大幅减少了平台相关代码的规模^③，使得移植工作变得更加容易。

子系统模块化 `rCore` 对各个子系统做了更彻底的分割，将它们作为独立的包发布出来。这些包可以进行单元测试，甚至可以被用户程序或其它 Rust OS 项目所引用。表 3.2 中列举了从 `rCore` 主项目独立出去的软件包。在下一节中我们以其中的线程调度模块为例，介绍一个可复用模块的实现过程。

① 一个轻量级的 C 语言标准库。<https://www.musl-libc.org>

② `uCore_plus`: https://github.com/oscourse-tsinghua/uCore_plus

③ 每个平台独有的代码量都在 2000 行以内

表 3.1 rCore 主要依赖的一些软件包

软件包名	主要功能
log	日志模块
spin	自旋锁
lazy_static	用来定义延迟初始化的全局变量
bitflags	按标志位处理整形数据
x86_64	对 x86_64 底层指令和数据结构的封装和抽象
pc-keyboard	解析键盘扫描码
xmas-elf	解析 ELF 文件
linked_list_allocator ^①	堆内存分配器（简单链表数据结构）
smoltcp	网络协议栈

① 这是 BlogOS 的一个副产品。早期 rCore 使用这个包，后期由陈嘉杰在此基础上改写实现了伙伴算法，因此目前 rCore 使用他实现的 `buddy_system_allocator` 作为分配器。

表 3.2 rCore 子系统独立出去的软件包

软件包名	主要功能
rcore-memory	虚拟内存管理
rcore-thread	线程调度
rcore-fs-*	文件系统（进一步分为 VFS, SFS 等子项目）
isomorphic_drivers	设备驱动集合（可同时在内核态和用户态使用）

3.2 构造可复用模块：以线程调度为例

本节我们考察 rCore 中线程调度模块的设计与实现。这个模块直接运行于裸机之上，用软件^①实现有栈线程^②的上下文切换。它支持多核调度，支持自定义线程对象和上下文切换函数，支持自定义的调度算法。它对外提供和 Rust 标准库中线程模块 `std::thread` 完全一致的接口，同时提供底层接口以支持更精细的控制。除了为 rCore 实现线程调度外，这个模块还能被应用在 BlogOS 中，甚至还能用于普通的用户程序，实现用户态的有栈协程机制。这个模块展示出了强大的可定制性和泛用性，而这些都离不开 Rust 语言的支持。

① 与软件切换相对的是硬件切换，例如 x86 指令集中的 TSS 原本就用于硬件切换线程。但由于硬件切换开销大、灵活性差，因此目前的操作系统都使用软件切换。

② 与有栈线程相对的是无栈协程，是一种常见于用户态的轻量级任务对象，后文会进一步提到。

3.2.1 内核线程与用户线程

想要写好代码，我们首先要对目标对象有足够深刻的认识。让我们梳理一下线程的相关概念：

- 线程是顺序控制流的抽象，它是操作系统进行调度的最小单位。
- 每个线程有自己的栈，用来保存函数栈帧等运行时数据。
- 线程分时占用 CPU 资源，操作系统通过上下文切换（*Context Switch*）让 CPU 从一个线程切换到另一个线程。
- 线程切换有两种策略：协作式^①和抢占式。

协作式 线程需要主动进行上下文切换交出 CPU 控制权，否则就一直执行
抢占式 线程运行中会被打断，切换到其它线程。

- 用户线程都是抢占式调度，当它的时间片用尽时，会触发时钟中断，陷入内核态执行上下文切换；内核线程一般是协作式调度，在运行时往往会关闭中断，需要线程主动调用切换函数。
- 与进程的概念相对，线程不拥有系统资源，狭义上线程的组成部分只有寄存器状态。

另一个值得探讨的话题是用户线程和内核线程的区别和联系。

编写多线程用户程序离不开系统线程库的支持，如 Linux 系统中著名的 `pthread`。那么可不可以直接在内核中使用 `pthread` 构造线程呢？答案是否定的，因为 Linux `pthread` 的实现就是用内核线程实现的用户线程。具体而言，每一个用户线程唯一对应一个内核线程，这也被称为多线程的一对一模型。除此之外，还有多对一模型和多对多模型，如图 3.3 所示。

在一对一模型中，每次用户线程切换都要进入内核执行上下文切换，开销较大。为了避免这种开销，诞生了多对一模型，也就是纯用户态线程，上下文切换都在用户态完成，操作系统感知不到这些线程。这种设计非常高效，但线程就只能协作式调度，如果其中一个线程执行了系统调用，那么其它线程都会被阻塞掉。多对多模型结合了二者的优点，我们开启若干个内核线程作为工作线程（*Worker*），将所有用户线程放到一个线程池（*Thread Pool*）中，工作线程独立地从线程池中取线程执行。这种模型没有阻塞问题，切换又很高效，是现代多线程程序的常用做法。

^① 也叫非抢占式

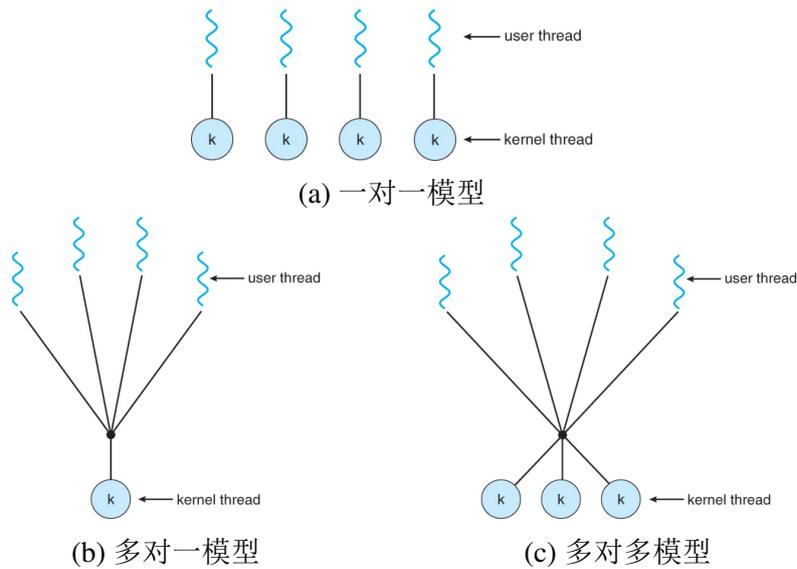


图 3.3 三种用户线程和内核线程的对应模型^①

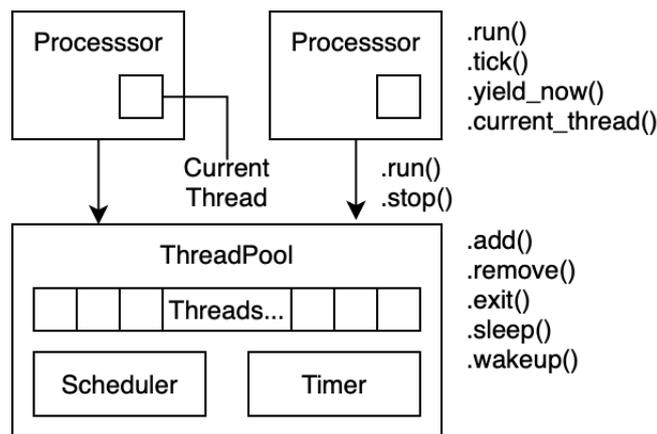


图 3.4 线程调度模块的主要数据结构和接口

3.2.2 内部数据结构和接口的设计

有了这些背景知识，就不难理解 rCore 线程调度模块的结构了。因为我们即将实现的内核线程，它与多核 CPU 的关系，正好对应了上节多对多模型中，用户线程与内核线程的关系，也就是任务和执行者的关系：每个 CPU 核都是线程的执行者，它们并行且独立地从公共的线程池中取一个线程执行。此外，我们还需要有调度器来选取每次让哪个线程执行，以及一个简单的定时器来适时唤醒睡眠的线程。

据此，我们设计了如图 3.4 所示的对象结构。Processor 对象代表了一个逻辑处理器，我们将其定义为全局变量，以便任何时候都能访问到它。由于每个 CPU 核只能访问自己对应的 Processor 对象。因此它是 CPU-Local 的数据结

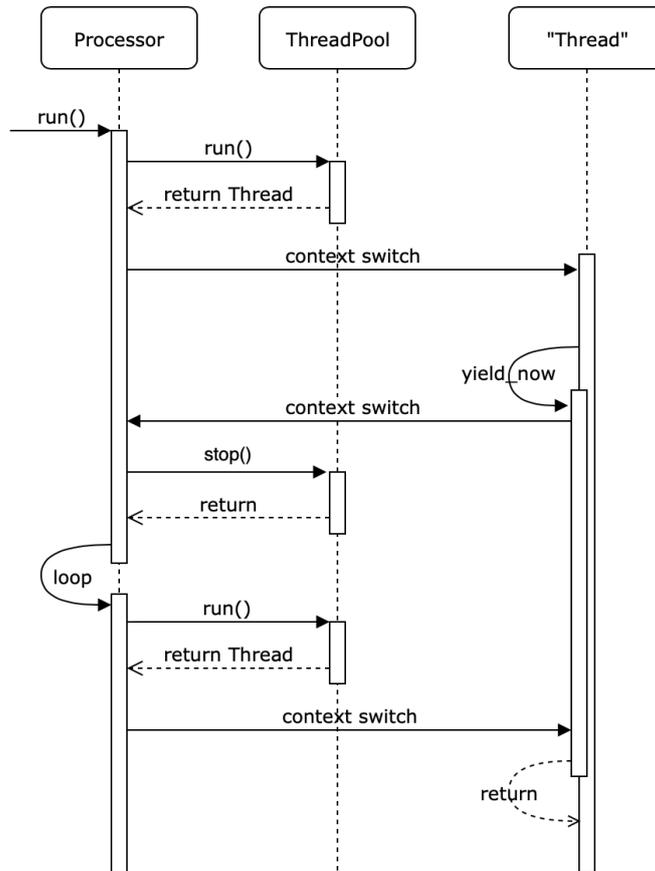


图 3.5 线程切换时的函数调用流程图

构，不用加锁保护。ThreadPool 是全局唯一的线程池，它是线程的容器，负责管理所有线程的状态，其中包括了线程的上下文信息。所有的 Processor 对象都会引用这个线程池，它向前者提供两个接口 run 和 stop，分别用来取走一个线程执行和结束一个线程的执行。

当操作系统结束初始化工作，CPU 开始执行线程任务时，会调用 Processor 的 run 函数，进入调度循环。在循环中，Processor 首先调用 ThreadPool 的 run 函数，拿到一个线程对象并保存为成员变量。然后对其中的上下文执行 switch 操作，切换到目标线程。目标线程在运行过程中，可访问 Processor 全局变量，调用 current_thread 函数获取自己线程的信息，例如线程号 tid 等。当线程主动让出 CPU 时，调用 Processor 的 yield_now 函数，再次进行上下文切换，回到调度循环。最后，在循环中调用 ThreadPool 的 stop 函数归还线程对象，结束这个线程的执行。整个流程如图 3.5 所示。

在调度过程中，还有两种特殊情况：第一种是在线程运行过程中发生了时钟

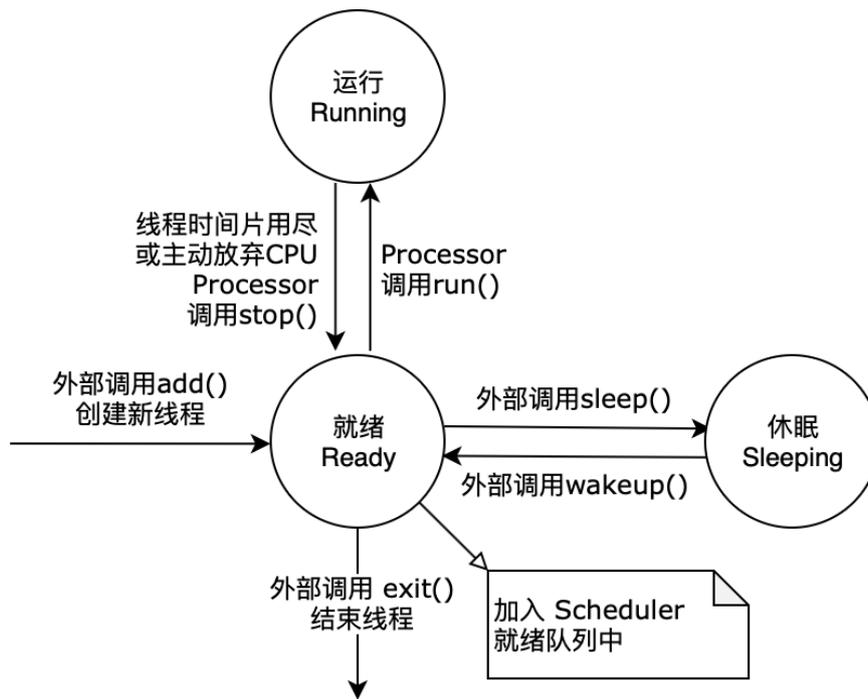


图 3.6 线程状态转移图

中断，那么需要调用 Processor 的 tick 函数，它会进一步向 ThreadPool 询问当前线程是否时间片用尽、需要调度，如果是的话则调用 yield_now。另一种情况是 Processor 发现当前没有待执行线程，也就是 ThreadPool::run 返回 None，此时就会开启中断并让 CPU 休眠，等待下一次中断时再重新开始循环。

作为线程的管理者，ThreadPool 对外提供了 add 和 remove 函数用来新增和移除线程。系统还可以调用它的 exit、sleep、wakeup 等函数改变线程状态。我们为每个线程设计了 3 种状态：运行，就绪，休眠。它们的状态转移过程如图 3.6 所示。

当线程进入就绪状态时，我们将它加入调度器 Scheduler 中。当使用 run 函数运行一个线程时，我们调用 Scheduler 的 pop 函数取出下一个要运行的线程。在其它情况下，当线程离开就绪状态时，我们需要将其从调度器中移除。Scheduler 的接口如下所示，在此接口下可以实现不同的调度策略。

```

1  /// The scheduler for a ThreadPool
2  pub trait Scheduler: 'static {
3      /// Push a thread to the back of ready queue.
4      fn push(&self, tid: Tid);
  
```

```

5     /// Select a thread to run, pop it from the
6     ↪ queue.
7     fn pop(&self, cpu_id: usize) -> Option<Tid>;
8     /// Got a tick from CPU.
9     /// Return true if need reschedule.
10    fn tick(&self, current_tid: Tid) -> bool;
11    /// Set priority of a thread.
12    fn set_priority(&self, tid: Tid, priority: u8);
    }

```

最后需要注意的是，由于 ThreadPool 是多线程共享对象，需要对其加锁以实现 Sync 特性。为了提高并发度，我们将其进一步分解成细粒度的锁，也就是 Scheduler, Timer 和每一个 Thread 对象都加了锁。Rust 语言可以坦然面对这种大量细粒度锁的情况，而不会出现 C 语言中忘记上锁解锁的问题。

3.2.3 底层依赖与对外提供的接口设计

设计好模块的内部结构后，还要考虑两个重要问题：这个模块底层依赖哪些函数？对外暴露什么样的接口？在良好的设计中，底层依赖应尽可能少，以提高模块的泛用性；上层接口应在功能完整、易于扩展的前提下，尽量精简并且正交。

上下文切换

首先我们考虑线程调度的底层机制，也就是上下文切换。上下文 (*Context*) 指的是线程的寄存器状态，当发生上下文切换时，首先将当前线程的寄存器内容保存到内存，然后从内存中恢复目标线程的寄存器内容。在不同的指令集上，寄存器内容和上下文切换过程都各不相同。不过在上一节中，我们并没有关心这些细节，而仅仅使用到了“上下文切换”这一抽象的行为，在代码中的体现就是一个简单的 trait:

```

1 pub trait Context {
2     /// Switch to target context
3     unsafe fn switch_to(&mut self, target: &mut
4     ↪ Context);
    }

```

在新建线程时，我们先为其构造好一个初始的上下文，也就是 Context 对象，然后把它加入 ThreadPool 中。接下来我们的线程模块就能通过它的 switch_to 方法进行线程切换了。

```
1 impl ThreadPool {
2     /// Add a new thread
3     pub fn add(&self, context: Box<Context>) -> Tid
4     ↪ { ... }
5 }
```

当线程退出时，会同时销毁掉 Context 对象。因此这里可以应用 RAI 模式来将线程的资源与 Context 对象的生命周期绑定，以达到线程退出即释放资源的效果。在实践中，我们将线程的所有资源集中到一个 Thread 类中，然后为其实现 Context 特性：

```
1 pub struct Thread {
2     context: ArchContext, // 平台相关的寄存器上下文
3     kstack: KernelStack, // 内核栈
4     proc: Arc<Mutex<Process>>, // 共享其所在进程的资源
5 }
6
7 impl Context for Thread { ... }
```

std::thread 接口

在目前的接口下，为了新建一个线程，我们需要根据函数入口地址创建一个 Context 对象，再将其加入 ThreadPool 中。这样一来有些麻烦，二来不能直接用闭包函数创建线程（闭包没有静态的入口地址）。因此我们需要一套更高层的线程操作接口，幸运的是 Rust 标准库中已经有了很好的设计，那就是 std::thread 模块。下面是它的主要用法演示，可以看到非常优雅。

```
1 use std::thread;
2 use std::time::Duration;
3
4 // 从闭包函数创建新线程
```

```

5 let handle = thread::spawn(|| {
6     // 休眠当前线程
7     thread::park();
8     // 结束并返回值
9     return 1;
10 });
11
12 // 让当前线程休眠一段时间，以等待新线程
13 thread::sleep(Duration::from_millis(10));
14
15 // 唤醒新线程
16 handle.thread().unpark();
17
18 // 等待新线程结束，并获取返回值
19 let ret = handle.join().unwrap();
20 assert_eq!(ret, 1);

```

我们全盘接受了它的设计，将底层接口进一步包装成和 `std::thread` 一模一样的接口。于是使用这个模块的开发者完全可以把它当作裸机环境下的线程标准库来用，大幅降低了使用者的学习成本。

在整个包装过程中，唯一有点难办的问题是：如何让新线程运行这个动态的闭包函数。我们的解决方法是，利用一些泛型的技巧，为每个闭包函数创建一个对应的静态函数，然后将闭包放在堆空间上，并把它的指针作为静态函数的参数传入，新线程进入静态函数以后，先解析参数再调用闭包。最后用类似的方法，将闭包函数返回值放在堆上，并把它的指针作为线程退出码传递出来。

3.2.4 展望：与 Async 无栈协程机制的对比

以上，我们完整实现了一个在裸机环境中构造线程的模块。值得一提的是，在当下的主流编程语言中，还存在着另一种任务机制，称为无栈协程（*Stackless Coroutine*）。它的本质是一个状态机，将任务状态存储于对象中，每次运行都是一次函数调用，需要让出 CPU 时先更新对象状态，然后直接函数返回。这样每个任务都不需要保存自己的函数栈，无需上下文切换，任务切换开销很小。无栈协程适合有大量 IO 任务的场景，因此近年来在网络服务器中被大量使用。

为了配合无栈协程机制，编程语言普遍加入了 `async` 语法来帮助我们将普通函数转化成状态机对象，并配合 `await` 语法处理子任务。有了 `async-await` 语法，我们就能用同步的风格编写异步逻辑，以实现高性能 IO。`Rust` 语言也引入了同样的机制，并计划于 2019 年下半年稳定这一语言特性，目前已经有部分关键接口的定义进入了核心库^①。

通过研究相关代码，我发现实现无栈协程和有栈线程的代码结构几乎相同，唯一的区别是切换任务的方式：前者是函数调用，后者是上下文切换（实际上也被封装成函数调用）。这意味着二者在任务调度上本质是相同的。那么，未来是否可以统一协程和线程的调度框架？

进一步回到操作系统。OS 中大量异步 IO 的需求非常适合用无栈协程来实现。但由于目前的 OS 都用 C 语言编写，而 C 语言是没有这套机制的，因此还没有在 OS 内部使用无栈协程的先例。而 `Rust` 语言的出现为这件事创造了独特的机遇。据了解，`BlogOS` 已经开始计划使用 `async` 机制实现内核线程，但具体方法还不明晰。在操作系统内部，无栈协程相比有栈线程是否还有性能优势？能否应用无栈协程实现 IO 子系统？甚至是整个内核系统？这些都是十分值得探索的问题。

^① `core::task` 和 `core::future`

3.3 Linux 系统调用层

上一节中我们介绍了一个子系统模块的实现过程，rCore 中其它的子系统如内存管理和文件系统，也是通过类似的方法实现的。接下来我们将基于这些底层模块实现标准的 Linux 系统调用，以支持实际的 Linux 程序。

目前 rCore 共处理了约 130 个系统调用，但其中只有 90 个真正实现了功能，剩下都是的空实现。空实现主要有以下两种情况：

- 直接返回成功。欺骗用户程序，使其认为系统正常处理了请求。一般用于不会影响主要功能，例如用户、信号相关的系统调用。
- 直接返回错误。由 libc 检测到错误后使用其它的系统调用代替，或直接忽略错误。最典型的是用于动态内存分配的 brk，当返回错误后，musl libc 会继续尝试使用 mmap 分配内存。

在真正实现了的系统调用中，也有很多功能类似，只是参数不同的接口。这些一般是出于历史原因，在古老的接口上添加新的参数以拓展功能。但为了保持向后兼容，原始的函数都予以保留，不过在一些新的指令集中就被删除了。例如在 Linux 2.6.16 以后，文件类系统调用中加入了大量以 at 结尾的新接口，它们都添加了 dirfd 参数，用来指定相对路径的起始目录（原来默认以当前工作目录为起点）。比如在 x86_64 中，同时存在 open 和 openat 函数，但在新的 RISC-V 中，只保留了 openat 函数。在 rCore 中，这些系统调用都被实现，但很多只需简单包装即可。

表 3.3 中列出了目前 rCore 处理的全部系统调用。我们对它们按功能进行了划分，并进一步分为核心的、类似的和空实现三种情况，分别使用粗体、正体、斜体表示。

尽管看上去很多，但其实我们只需重点实现加粗的部分，并且这些也是在测试大量程序的过程中逐渐积累出来的。我们实现这些系统调用的过程分为若干阶段，每一阶段都有一个明确的目标程序：

Hello world 首先我们选定了 musl 作为待支持的 C 标准库，然后尝试运行基于 musl 静态链接编译的 Hello World 程序，这一过程中我们需要实现 musl 建立运行环境时不可或缺的系统调用。

Busybox 是 Linux 系统下的实用工具箱，集成了如 ls, cat 和 shell 等最基本的命令和程序。这些小程序很多都是对文件相关系统调用的简单封装，十分方便实现和测试。其中最重要的程序就是 shell，它会依赖很多进程相关的系

表 3.3 rCore 处理的系统调用清单

类别	系统调用
文件	<code>read, write, openat, close, fstatat, lseek, ioctl, fcntl, fsync, fdatasync, ftruncate, getdents64, getcwd, chdir, renameat, mkdirat, linkat, unlinkat, readlinkat, dup2, pipe, sendfile, select, poll, open, stat, lstat, fstat, pread64, pwrite64, readv, writev, truncate, mkdir, rmdir, link, unlink, readlink, dup3, pipe2, copy_file_range, ppoll, symlink, symlinkat, flock, chmod, fchmod, fchmodat, chown, fchown, fchownat, access, faccessat, utimensat, epoll_create, epoll_create1</code>
文件系统 进程	<code>sync, statfs, fstatfs, mount, umount2, fork, clone, execve, exit, exit_group, kill, wait4, nanosleep, set_tid_address, futex, sched_yield, sched_getaffinity, vfork, tkill</code>
内存 信号	<code>mmap, munmap, mprotect, brk, madvise, sigaction, sigprocmask, sigaltstack, sigqueueinfo</code>
网络	<code>socket, connect, accept, sendto, recvfrom, recvmsg, bind, listen, getsockname, getpeername, getsockopt, setsockopt, accept4</code>
时间	<code>gettimeofday, clock_gettime, time, alarm, setitimer</code>
内核模块 系统	<code>init_module, delete_module, finit_module, gettid, getpid, getppid, uname, getrusage, sysinfo, times, setpriority, prlimit64, reboot, getrandom, arch_prctl, umask, getuid, getgid, setuid, geteuid, getegid, setpgid, setsid, getpgid, getgroups, setgroups, prctl, membarrier</code>

统调用。支持 Busybox 将大大方便系统运行后的测试环节。

GCC 是最常用的 C 语言编译器。它大部分计算在用户态进行，对系统的依赖主要是文件操作，同时它的整个编译流程是通过多进程配合完成的，因此 GCC 可以对文件和进程部分的实现进行综合检验。

Nginx + Redis 是现代高性能网络程序的代表。它们需要使用 Socket 接口完成网

络操作，还会用到 `select` 和 `poll` 等 IO 多路复用接口。支持这两个程序的主要目的是和 `Biscuit` 以及 `Linux` 进行性能对比。

Rustc 是 `Rust` 语言官方编译器。与 `GCC` 不同的是，它使用多线程并配合动态链接完成整个编译流程。作为更现代的程序，它还使用了很多最新的系统调用。`Rustc` 可以检验系统对多线程以及 `Rust` 标准库的支持情况，此外还有一个更宏大的目标是实现 `rCore` 系统的自举，不过这是一个相当大的挑战。

我们的实现过程采用完全测试驱动的开发方式。首先编译好一份可在 `Linux` 中正常运行的程序，然后直接放到 `rCore` 当中跑，逐个实现缺失的系统调用，并修复以前留下来的 `Bug`。对于需要实现的新功能，我们遵循“`quick and dirty`”的原则，采用最快最简单的方案去实现。一切以尽早让程序正常运行为奋斗目标。当一个阶段的快速开发告一段落后，我们再对新代码进行反思重构，以便下一阶段的开发能更顺利地进行。

3.3.1 Hello world: 支持基于 `musl libc` 的 `Linux` 程序

`libc` 的选择

`C` 语言标准库 (`libc`) 是用户程序的最底层 `API`。目前几乎所有的用户程序都建立在某个 `libc` 之上，极少直接和系统调用打交道。而不同的 `libc` 所使用的系统调用也有细微区别，因此选择一个合适的 `libc` 就尤其重要。

我们平时最常用的是 `glibc`，它功能全面但是过于复杂，一个从零实现的系统难以支持。另一个轻量级的版本是 `musl libc`，它实现简单并且常用于静态链接。`Linux` 发行版 `Alpine` 就是完全基于 `musl` 以及 `Busybox` 构建的。最后是我们的参考对象 `Biscuit`，它使用自己剪裁的 `libc` (称为 `lirc`)，全部代码只有不到 4000 行。但它并不完全兼容 `Linux` 接口规范，有很多自定义的成分。

综合考虑下来，虽然 `lirc` 最为简单，但其不兼容 `Linux` 的特点可能导致后续大量程序的移植成本。相比之下 `musl` 可能更加复杂，但它支持几乎全部的 `Linux` 程序。因此我们选择适配 `musl`，这是一件一劳永逸的事情。

处理 `syscall` 指令

标准的 `Linux x86_64` 程序使用 `syscall` 指令进行系统调用，而 `rCore` 之前的用户程序使用软中断 `int 0x80`。在默认情况下，执行 `syscall` 会触发一个指令异常，我们只需在异常处理中判断一下指令码即可。

在后期我们也实现了真正的 `syscall` 指令中断处理，从而优化系统调用的性能。首先要打开一个 `CPU` 开关，然后重新用汇编实现一个保存和恢复用户现场

的操作。

通过 strace 分析系统调用

在将目标程序放到 rCore 上运行前，我们可以先在 Linux 上使用 strace 工具分析它用到的系统调用。例如下面就是对一个最简单静态链接的 Hello World 程序的分析结果：

```
1 execve("./hello", ["/hello"], 0x7ffdd56772f0 /* 6 vars */)
  = 0
2 arch_prctl(ARCH_SET_FS, 0x4065f8)          = 0
3 set_tid_address(0x4067f8)                 = 86
4 ioctl(1, TIOCGWINSZ, {ws_row=25, ws_col=80, ws_xpixel=0,
  ws_ypixel=0}) = 0
5 writev(1, [{iov_base="Hello, world!", iov_len=13}, {iov_base
  =NULL, iov_len=0}], 2Hello, world!) = 13
6 exit_group(0)                             = ?
7 +++ exited with 0 +++
```

值得关注的是 arch_prctl。经过与 musl 代码的比对，我们发现它会通过这个系统调用将 FSBASE 寄存器设置为 pthread 中描述本线程的数据结构指针。FSBASE 在 x86_64 下用作线程局部存储的起始地址，如果不设置的话会导致访问 0 地址。

支持线程局部存储 (TLS)

以上分析告诉我们，支持线程局部存储 (Thread Local Storage) 是 musl libc 运行的必要条件。它在内核中的实现分为以下三个部分：

- 实现 arch_prctl 系统调用，在其中使用 wrmsr 指令设置 FSBASE
- 在发生中断时保存 FSBASE 到中断帧中
- 将 ELF 头的地址放在初始栈中传递给 musl。具体而言，需要从 ELF 中解析 PHDR 段的虚拟内存地址，将其作为辅助向量 (Auxiliary Vector) 的 AT_PHDR 参数。musl 通过读取这一地址解析 ELF 头，找到 TLS 数据段的起始地址，在线程初始化时从此处复制数据。

有了这些以后，我们的程序就可以顺利地 rCore 上输出 Hello World 了！

3.3.2 Busybox + GCC：实现文件和进程相关系统调用

Busybox 中自带 sh 作为简单的 Shell 程序。它首先会使用 ioctl 获取一些 tty 信息，这些我们需要正确处理。然后通过 poll + read 读取用户输入并使用 writev 回显，因此我们需要为标准输入文件（也就是串口或键盘设备）实

现 `poll` 方法，这里底层使用了条件变量，用来在有新的输入时唤醒等待线程。此处的另一个问题是需要对键盘和 VGA 设备正确处理 ANSI 转义符，例如将键盘上下左右按键转化成输入序列，并识别处理光标移动等控制序列。在收到输入后，`sh` 使用标准的 `fork + execve` 模式启动新进程，新进程在 `execve` 之前，会有很多信号相关的系统调用，我们可以直接忽略它们。

在支持 `sh` 以后，我们就可以把它作为系统的默认启动程序。进一步，如果我们实现了文件系统的符号链接，就可以直接使用 Alpine 的 `rootfs`^① 作为系统的磁盘镜像了。

对于 GCC 这种稍大一些的程序，往往有浮点数运算，并且会使用 SIMD 指令。这时就会用到浮点寄存器（x86 下是 `xmm`），我们需要在中断时保存它们。

3.3.3 Nginx + Redis: 实现网络功能^②

为内核添加网络功能是一个庞大的工程，它分为三个主要部分：网卡驱动、网络协议栈和 Socket 系统调用接口。

网卡驱动

在 x86 平台下，识别设备首先要对 PCI 总线进行枚举探测，我们发现已经有人写好了相关的 Rust 库^③，就直接拿来使用。在找到正确的网卡设备后，获取它的 BAR 内存范围，并为其设置中断号。

接下来我们构造一个驱动对象，并将它的 MMIO 内存地址范围传入。驱动内部可以合法地使用 `unsafe` 访问这片内存，并自行保证内存安全。此外，驱动普遍都有分配连续物理内存用于直接内存访问（DMA）的需求，因此我们为其抽象出了 `Provider` 接口，提供全部的底层支持。这样，我们就可以把驱动从内核中分离出来，形成独立的包。并且，这个包还可以直接在用户态使用，形成用户态驱动，在 Linux 下只需把设备的 BAR 通过 `mmap` 映射到用户态即可。我们将这个包命名为 `isomorphic_drivers`（同构驱动），已经发布到了 GitHub 上。

具体实现时，我们参考了 `Biscuit` 和 `JudgeDuck`^④ 项目，确定了两种典型的网卡 E1000（千兆）和 IXGBE（万兆），将它们的驱动代码从 Go 和 C 语言翻译到了 Rust。

① <https://alpinelinux.org/downloads/>

② 这部分工作主要由陈嘉杰同学完成

③ <https://gitlab.com/robigalia/pci>

④ <https://github.com/wangyisong1996/JudgeDuck-OS>

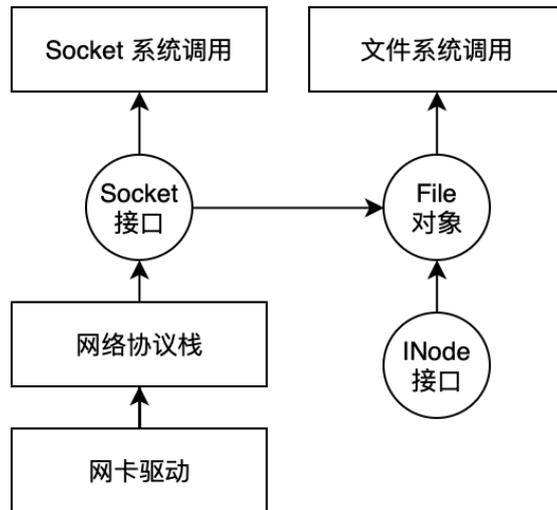


图 3.7 网络子系统结构图

网络协议栈

网络协议栈是整个网络子系统最为复杂的部分。幸运的是，Rust 社区已经有了比较完整的用于嵌入式设备的网络协议栈包 `smoltcp`。它支持 TCP、UDP 和 IP 协议，定义好了底层驱动需要实现的收发包接口，对外提供一种简化的 Socket 接口。它的局限性是只为单线程设计，性能较差。

我们仅用了几天时间，就将 `smoltcp` 引入 `rCore` 项目，并完成了和网卡驱动的对接，能够在 QEMU 中运行一个简单的 HTTP 服务器了。

Socket 系统调用

最后一步是将网络协议栈提供的功能封装成 Socket 接口，作为文件提供给用户程序使用。由于 `smoltcp` 提供的接口与真正的 Socket 之间还有一定差距，并且考虑到未来替换协议栈的需要，我们在系统内部抽象出了 Socket 接口，隔开了系统调用层和具体协议栈的实现。整个网络子系统的结构如图 3.7 所示。

表 3.4 Socket 和 INode 接口的对比

Socket	INode
<code>read, write, poll, io_control</code>	
<code>connect, bind, listen, shutdown, accept, endpoint, remote_endpoint, setsockopt</code>	<code>metadata, set_metadata, sync_all, sync_data, resize, create, link, unlink, move, find, get_entry</code>

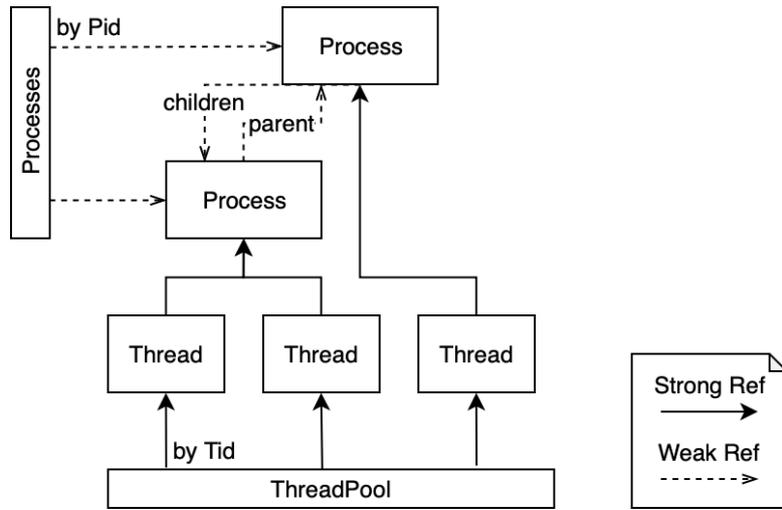


图 3.8 进程对象结构图

Socket 和 INode 接口有很多共性，也有很多差异，因此我们将二者设计为不同的接口，但都可以作为文件对象的后端，提供 read、write、poll 等操作，如表 3.4 所示。Socket 最大的特点是必须支持 poll 操作，即有能力在网络事件发生时唤醒等待线程。一个理想的设计是给 Socket 和 INode 添加注册回调函数的 register 接口以及非阻塞询问状态的 poll 接口。在 sys_poll 的实现中，如果没有 poll 到期望的状态，就通过 register 向设备注册回调函数，在发生指定事件时唤醒自己，然后进入休眠。不过目前，我们暂时使用一个全局的条件变量简单粗暴地实现这个需求。

3.3.4 Rustc: 实现多线程与同步互斥

多线程

实现用户态多线程是一件麻烦的事情。它涉及到内部结构的调整，系统调用的实现，在多核下还有 TLB Cache 一致性的问题。

我们首先对进程子系统的对象结构进行了一次重构，分离出线程和进程。如图 3.8 所示，实线表示强引用，虚线表示弱引用。我们让线程对象 Thread 强引用进程对象 Process。这乍一看上去反转了线程和进程概念上的从属关系，但在实践中这是一种自然的表示。因为每个线程必定有它从属的进程，且大部分的访问模式都是从当前线程找到当前进程。除此之外，其它的关系都是弱引用，包括进程的父子关系和通过 Pid 查找进程。这样的设计保证了当所有线程退出后，它们所属的进程对象也会自动被销毁，从而释放进程的资源。

Linux 通过 clone 系统调用创建新线程。它的主要参数包括新线程的入口

地址、栈地址和 TLS 地址。此外还有 flags 参数包含了大量的可定制选项，不过 musl 只会使用固定的选项组合，因此我们只对这一种组合的语义进行了实现。clone 和 fork 的内部实现都需要创建新的 Thread 对象，唯一区别是后者需要复制 Process 对象和中断帧，而前者只需引用现有的 Process 并根据系统调用参数创建新的中断帧即可。

另外需要注意的是，改成多线程后受影响的还有 execve 系统调用。根据规范，多线程程序中的任意一个线程调用 execve 都会导致其它线程立即退出，接着整个进程被替换。尽管这种情况在真实场景下很少见。

在多核环境下，多线程还会产生多核共享同一个页表的需求，进而导致 TLB Cache 一致性问题。因此在每次修改页表后，还需要通过处理器间中断 (IPI) 机制通知其它核刷新自己的 TLB。

Futex

在调用 clone 创建新线程后，musl 会立刻调用 futex。Futex 全称快速用户空间互斥锁 (*Fast Userspace muTEX*)，是一种内核态和用户态混合的高效同步机制。Futex 基于用户空间共享内存中的一个原子整型变量，主要有两个操作等待 (*Wait*) 和唤醒 (*Wake*)：

Wait 在指定 futex 变量上睡眠等待^①

Wake 唤醒指定 futex 变量上的若干个等待者

需要注意的是，futex 还有一个隐蔽的调用点，在每个线程退出的时刻。Linux 为每个线程规定了一个 clear_child_tid 属性，这是一个指向用户内存的指针，当线程退出时会隐式地在这个地址上进行 futex 唤醒操作。还记得在 3.3.1 中出现的 set_tid_address 系统调用吗？它就是用来设置这个属性的。clone 系统调用的 ctid 参数也会设置这个属性。具体细节可参阅文档^②。

musl 使用 futex 机制完成线程间的同步：当线程 A join 线程 B 时，就会在 B 的 futex 上等待，当线程 B 退出时，会唤醒自己 futex 上所有的等待线程。此外，musl 还会将所有线程的 clear_child_tid 属性设置为进程唯一的线程列表锁 (*Thread List Lock*) 地址，利用它在线程退出时自动解锁的特性，保证正确的互斥访问。

Futex 在 rCore 中的实现利用了条件变量，比较平凡，此处就不再赘述。因为相比之下，准确理解 Futex 的语义和系统行为才是最重要的。

^① 完整的 wait 还会检测 futex 的值以防止竞争现象，此处略过。

^② http://man7.org/linux/man-pages/man2/set_tid_address.2.html

动态链接

实现了 `clone` 和 `futex` 后，就可以运行基于 `musl pthread` 的简单多线程程序了。不过此时距离 `Rustc` 还差最后一步——动态链接。动态链接需要一个用户态的装载器配合（在 `Linux` 下是 `ld.so`）：操作系统先运行装载器，由它在运行时将目标程序及其依赖的动态链接库通过 `mmap` 系统调用映射到内存中，最后跳转到目标程序执行。

由上面的分析可知，动态链接需要操作系统支持 `mmap` 文件，并在 `execve` 时读取 `ELF` 文件中的链接器地址，然后转而加载链接器程序。这里 `mmap` 文件的最简单实现方式是，一次性分配所有物理内存并将文件内容复制进来，并且不需要将内存中的修改写回文件。

优化性能：延迟映射物理内存

做完了这些，理论上就可以运行 `Rustc` 了。不过实际上还有一个问题：动态链接加载程序的过程太慢了。原因是我们一上来就读取并复制了文件的所有内容到内存中。对于 `Rustc` 而言，大大小小的动态库加起来有几百 `MB`，全部加载进来需要几分钟时间，而实际运行中很有可能只用到其中的一小部分。

此时我们有必要进行一步重要优化：延迟映射用户程序的物理内存。首先我们不在页表中进行映射，只有当它访问了一个页，触发缺页异常（`Page Fault`）时，再实际分配物理内存，并从文件中复制数据或将其清零。这个优化对 `fork` 的性能也有提升，对于只读页和未被修改过的数据页（`Dirty` 位为 0），也都无需立即建立映射和复制数据了。

修复 Bug：延迟映射导致的死锁

实现延迟映射特性后，系统出现了一个 `Bug`：在内核态处理系统调用时访问用户内存，可能出现缺页异常（这个页还没有被用户访问过），此时系统会重入中断处理函数，尝试获取内存管理对象的锁，而这个对象可能已经被之前的系统调用函数锁住了，于是就出现了死锁。

对此问题，我们提出了多种可能的解决方案，例如使用可重入锁，细化锁的粒度等。最终我们参考了 `Linux` 在内核态访问用户内存的策略，发现它没有进行任何用户地址合法性检查，而是直接在一个特殊函数中访问用户内存，在此期间不会锁住内存管理对象。如果发生了缺页异常，并且属于非法地址，就修改本次异常时的指令指针。当异常返回后，程序执行一段新的汇编代码（称为 `fixup` 函

数), 使原函数返回错误。这一系列精巧的操作相当于实现了高级语言中的“try {...} catch (PageFault) { return EFAULT; }”。

这里我们以此 Bug 为例, 为了说明实现新功能往往伴随着新问题的暴露, 事实上这种情况在日常开发中经常遇到。在进一步修复了各种大小 Bug 后, rCore 终于可以正常地用 Rustc 编译一个 Hello world Rust 程序, 并且编译出来的结果也能在 rCore 中正常运行!

3.4 小结：操作系统设计实现经验谈

本章我们全面介绍了 rCore 操作系统的技术特性：从它的整体架构，到一个具体的底层模块，再到建立在这些模块基础上的 Linux 系统调用层。但是，罗马不是一天建成的，rCore 的上万行代码也不是一朝一夕就码出来的。在一年多的时间里，我们进行了大量的实践与尝试，积累了丰富而宝贵的开发经验。在本节中，我们就结合具体的实践经历，谈谈编写一个小型操作系统的经验和收获，总结高质量开发的方法和教训，希望能对后人有所启发。

3.4.1 迭代式开发

rCore 的开发过程属于增量式或迭代式，每次先定下一个小目标，然后依次进行方案设计、代码实现、测试和重构。在设计实现新功能以及修 Bug 的阶段，我们采用 quick and dirty 的原则，一律使用最简单的实现方案。例如在加载用户程序时，我们先采用立即映射，再实现了延迟映射。这些 dirty 的代码一般都会在不久的将来被重构、重写，甚至彻底废弃。但它们有独特的存在价值，那就是用最快速度完成目标。

不过，在快速挺进时，也要为日后巩固阵地留下必要的保障：例如每个函数签名处必须有描述文档，尚未实现的特性必须写 TODO 或 FIXME，可能出问题的地方也必须用 WARNING 提示出来。它们将为下一阶段的重构标明雷区、指明方向。

3.4.2 测试驱动开发

推进开发进度最快的手段就是以测试驱动。在有了测试程序后，通过所有测试就成为了最明确的开发目标。例如在我们完成 musl libc 的适配后，瞬间解锁了海量的测试程序。于是在此后的一个月时间里开发进度高歌猛进，依次补全了 20 多个文件相关的系统调用，陆续支持了 Nginx、Redis、GCC 等实际应用。但是当这些目标达成后，由于没有了新的目标程序，开发速度就渐渐慢了下来。

虽然测试驱动的效果十分显著，但在实践中大家却普遍很难坚持下来，原因就是写测试这件事是大家都不愿意干的。我们之所以能有上面的成就，重要原因是已经有了现成的测试程序。这就引出了下一个话题：为了充分利用已有资源，我们必须遵守行业标准。

3.4.3 遵守行业标准

所谓行业标准就是一个领域内公认的行为规范。而由于计算机领域发展迅猛，标准的制定往往落后于技术的更迭，因此行业标准往往就是几年前的事实标准。例如 POSIX 接口就是操作系统的行业标准，而 Linux 系统调用接口则是当下的事实标准。

在白手起家、一无所有的时期，遵守行业标准可以带来巨大的资源加成。具体到软件开发领域，就是丰富的文档、测试以及对比目标。在 rCore 发展初期，uCore 是主要的参考目标，因此这一阶段 uCore 的系统调用就成为了我们的标准，我们甚至为 x86_64 的系统实现了运行 i386 程序的功能^①。兼容 uCore 用户程序使得我们可以直接使用它自带的 10 余个测试例程，在两周时间里实现了大部分测试要求的功能。

当 rCore 的完成度已经和 uCore 追平，需要进一步支持实际应用时，我们意识到需要确定一个新的标准。这就回到了 3.3 节中，对 Biscuit 的 libc 和 musl libc 的选择。事实上，我们当时还有第三条路可以走，那就是在必要时简化 musl，在用户空间而不是内核里面进行适配。这样考虑的原因是 TLS 对于单线程程序来说并不是必须的，而 musl 却必须依赖它才能运行。后面发生的事情大家也知道了，我们十分庆幸当时选择了完全兼容 Linux 的路线，坚持突破了 TLS 的门槛，使得大量预编译的用户程序可以直接运行，少数需要重新编译的程序也能在 alpine 环境下方便地完成。

除了系统调用，rCore 在其它方面也尽量遵守事实标准和最佳实践。例如在接口设计上，INode 参考了标准库中 File 的命名风格，线程模块封装成了和 `std::thread` 完全一致的接口。我们还参考其它 Rust 包为代码添加文档并使用 `cargo fmt` 统一代码风格。

3.4.4 面向接口编程

rCore 项目一直以来的一个努力目标，就是让操作系统内核这样一个高度耦合的整体，尽量地模块化。因此我们采用了面向接口编程的指导思想，将各个子系统之间的交互抽象成接口。

接口的设计方法可以总结为：将抽象的逻辑转化为接口函数，并根据实际情况进行微调。所谓抽象逻辑，可以通过简练的自然语言描述得到。例如在描述线程调度时，我们说“首先从就绪队列中取出一个线程，然后进行上下文切

^① uCore Lab 只有 i386 版本

换”。那么“就绪队列”就是一个接口 (Scheduler)，其中有一个方法“取出线程” (pop)；“上下文”也是一个接口 (Context)，其中有一个方法“切换” (switch_to)。我们并没有描述这些动作的细节，因此在面向接口编程时也不关心它们的内部实现。

不过在实践中我们发现，往往是先有具体实现，然后再抽象出接口。这样确保接口的设计是实际的、可落地的。一个反例是我们曾经从缺页置换算法出发，设计了一套页表接口。这些代码可以顺利地为用户态跑单元测试，但集成到内核的过程却困难重重，很多实际需求不知如何在这套接口下实现。此外，我们还观察到接口是会随着需求动态演化的。这些事实都说明，接口是自然归纳出来的，而不是刻意设计出来的。

3.4.5 持续推进重构

重构指的是在不改变软件功能的前提下改进代码的内部结构。由于我们在设计实现阶段采用 quick and dirty 的风格，复杂度在一段时间的积累后必然导致代码结构的腐化，使其不能及时地适应新需求。因此持续地进行重构是让软件健康发展的必要举措。

在 rCore 的开发过程中，重构一般分为大小两类。小型重构诸如重命名、移动代码、消除警告、改进写法等，就像打扫屋子一样，是一种日常行为。大型重构涉及整体结构或功能的变化，往往需要大面积修改代码。rCore 历史上几次典型的大型重构如下表所示：

表 3.5 rCore 历史上的大型重构

时间	重构内容	目的
2018.05	统一中断处理函数	支持创建新进程
2018.05	统一内存管理器 MemorySet	支持多进程的虚拟内存管理
2018.07	分离平台无关代码	支持跨平台
2018.10	重写线程调度模块	支持多核处理器
2018.12	抽象出页表映射策略接口 MemoryHandler	支持页表延迟映射
2019.03	分离进程和线程对象	支持用户态多线程
2019.05	自映射改为线性映射	简化页表和 DMA 操作

大型重构一般都有明确的动机，那就是现有的代码结构阻碍了新功能的实现。一次完整的重构从分析策划到具体实施，再到回归测试，一般要花上几天的

时间，可以说代价很大。但同时回报也是巨大的，包括重要功能的实现或性能的大幅提升。

从 **BlogOS** 到 **uCore** 再到 **Linux**，一路走来，**rCore** 已经脱胎换骨，代码被改得面目全非。没有人能一开始就设计出最合理的软件架构，架构是根据需求的推动而自然演化的。这样看来，**rCore** 就像是一个有机生命体，开发者们持续的维护是它活力的根源。

第 4 章 Rust 编写操作系统的能力分析

本章中，我们从几个不同的角度分析 Rust 语言编写操作系统的能力，指出它的优势和不足之处。

4.1 底层控制能力

操作系统需要能够完全地掌控它的运行环境，包括精确地访问内存、执行指令、控制资源使用。这些称为语言的底层控制能力。

和 C 语言等价的底层控制能力

虽然安全的 Rust 代码不允许直接内存操作，但在 `unsafe` 块中，我们可以通过裸指针访问特定的内存地址。Rust 语言中没有 `volatile` 关键字，但核心库中有相关的函数，并且也有第三方库封装了 C 风格的 `Volatile<T>` 类型可供使用。Rust 语言不支持 C 中的结构体位域，但存在第三方库 `bit_field` 和 `bitflags` 帮助我们简化位操作。对于原子操作，Rust 核心库封装了 LLVM 提供的相关接口，无需再用汇编手写原子指令。而 C 语言中直到 C11 标准后才提供了类似的原子操作库。

Rust 代码可以内联汇编，语法和 C 语言很像。Rust 项目也可以集成 C/C++ 代码，或链接静态/动态库。在和这些外部代码交互时，需要精确控制内存布局和函数调用行为。Rust 内部的 ABI 是没有明确定义的，因此这时就要使用通用的 C 语言 ABI。在使用特定的标注后，程序的行为就变得精准可控了。

Rust 使用基于 RAII 的半自动内存管理。动态内存的分配和回收基于对象的生命周期或引用计数自动完成。相比垃圾回收而言，它对内存资源的使用已经相当可控了。如果需要更精准的控制，也可以退回到 C 风格的 `alloc` 和 `dealloc` 接口手动管理内存。

总结下来，虽然一般的 Rust 代码对应的底层实现是不可控的，但如有需要一切都可以精确控制。因此我们认为 Rust 具有和 C 语言等价的底层控制能力。

对运行时要求极小

Rust 语言对运行时环境的要求极小：在 `no_std` 环境编写裸机程序时，除了函数栈以外，只需实现一个 `panic` 处理函数，就能顺利编译运行了。如果程序依赖动态内存分配，还需要引入 `alloc` 库，并实现一个 `oom`（内存耗尽）处理函数，再定义一个全局内存分配器（`GlobalAlloc`）。分配器可以直接使用第三方库，例如 `BlogOS` 作者写的 `linked_list_allocator` 等。

不适合编写过于底层的代码

虽然 Rust 支持直接内存访问，但它并不鼓励这种行为，体现就是 `unsafe` 代码在设计上十分啰嗦，裸指针用起来处处不爽。像驱动程序这种特别底层的代码中，会有大量直接访问内存的操作，并且全部使用 C 语言 ABI。因此这种场景下还是用 C 语言更加合适。

4.2 抽象与优化能力

在编写操作系统这种逻辑复杂的软件时，语言的抽象能力就十分重要。并且由于 OS 对性能的苛刻要求，这些抽象必须没有额外的开销，要求语言有很强的优化能力。

语言抽象方便了系统的实现

Rust 语言最基本的抽象是面向对象，这种描述事物的范式非常符合操作系统的特点：OS 中有各种各样的内核对象，如进程、文件、信号量等等。相比 C 语言的 Linux 中一切都用“xx 控制块”来表示，Rust 中直接用对象来描述就显得更加自然。

内核对象的另一个特点是它们都表示某种资源。在 Rust 中，可以进一步使用 `RAII` 机制，把资源的生命周期和对象的生命周期进行绑定，以实现资源的自动化分配和释放。大大简化了系统资源管理。

Rust 语言引入的更强大抽象是函数式风格。代数数据类型的枚举以及模式匹配，大幅简化了系统状态的表示和处理。另外最重要的特性是匿名函数与闭包，方便了事件和回调机制的实现，让创建内核线程变得异常简单。函数成为一等公民，促进了代码复用。

Rust 语言最强大的抽象是接口和泛型。它们最大的作用是帮助代码的解耦和分离。利用这些特性，我们先后将文件系统、线程调度和内存管理都抽离成了

独立的模块，并在其它 OS 中复用。这在 C 语言中几乎是不可想象的。

强大而激进的优化能力

上面提到的这些抽象在 Rust 中几乎都是零开销的。这都离不开 Rust 强大的优化能力。相比 C++，Rust 充分利用了不可变约束，做到更深入的优化。测试表明 Rust 甚至可以内联虚函数。

但是，强大到激进的优化策略也会带来一些问题。尤其是编写 `unsafe` 代码时，很容易错误地打破编译器假设，导致优化后出现未定义行为。在实践中我们发现了以下值得注意的问题：

内部可变性 内部可变指的是对象的一个不可变方法 (`&self`) 改变了内部数据的值，常见于并发对象 (*Concurrent Object*) 中，例如互斥锁、环形缓冲等。

Rust 语言规定，所有内部可变的数据最终都要包在一层 `UnsafeCell` 中，否则编译器看到不可变就可能把所有写操作优化掉。

数据合法性 Rust 语言规定所有类型的变量在任何时刻都有合法值。例如一个引用背后必须是一个合法指针。但在 OS 中经常会有不对变量进行初始化的场景，例如作为缓冲区的数组，此时需要正确使用 `MaybeUninit`。另一点需要注意的是，即使在 `unsafe` 代码中，也不能同时出现两个可变引用 (`&mut`) 指向同一个数据。这违反了 `&mut` 的语义，可能导致未定义行为。

栈对齐 这是一个比较隐蔽且危害很大的问题。每种指令集的函数调用 ABI 对于函数栈指针的对齐有不同的要求。编译器会基于栈对齐假设，对栈上变量的访问做优化。如果栈没对齐，会导致隐蔽的数据读写错误。OS 中所有需要手动控制栈指针的地方都要注意，包括 `boot` 后的入口函数、上下文切换和中断帧保存。

4.3 平台适配能力

Rust 语言后端基于 LLVM，广泛支持各种指令集平台。但相比 GCC，它们对小众平台的支持还是普遍落后一些，有些地方并不完美。例如 Rust 和 LLVM 对 RISC-V 指令集的支持直到今年年初才基本可用，目前还在完善中。rCore 几乎是第一时间用上了这些新特性，同时也踩了很多的坑。Rust 标准库在这些小众平台上也有一些没有考虑的特殊情况：例如 RISC-V 的原子变量最小是 32 位的，而核心库中的 `AtomicBool` 是 8 位的。我们不得不手动修改库代码，以实现正确的原子操作。

4.4 高质量开发能力

保证软件的安全性

我们在长期实践中，确实体会到了 **Rust** 提升软件安全性的好处：我们几乎没有遇到 C 语言中常见的内存安全问题，在多核物理机上也没有遇到任何数据竞争问题。在合理使用 **unsafe** 的前提下，只要编译通过，我们就能放心地认为代码不会出太大的差错，离完工也就不远了。

接下来要处理的就是软件逻辑上的问题。**Rust** 会在运行时及时把它们暴露出来：任何意料之外的行为，例如 `unwrap` 一个 `None` 值，越界访问数组等，都会立刻触发 **panic**。在并发问题上，由于强制共享对象上锁，所有的数据竞争都转化为了死锁，更容易定位和解决。

初学者较难上手

虽然 **Rust** 的这些机制保证了内存安全，但对于初学者而言，它们太过严格、难以驾驭，以至于经常陷入和编译器的斗争，无法写出编译通过的代码。此时，他们会倾向于使用 **unsafe** 操作绕过这些限制，使代码尽快通过编译。例如最让人头疼的生命周期约束，大部分人会把引用转成裸指针甚至整型变量。但这样做就完全丧失了 **Rust** 保证安全的好处，使代码退回 C 风格，并且写起来还十分啰嗦。

这里暴露的问题是：为了保证内存安全，**Rust** 提供了很强大的工具，但使用这些工具的门槛过高。然而这是一个难以解决的现实问题，因为 **Rust** 的高门槛是由内存安全问题固有的难解性决定的。在这一点上，**Rust** 已经给出了历史上最好的解决方案，能否进一步改进有待未来的探索。

第 5 章 结论

本工作利用新兴的 Rust 语言，编写了一个小型操作系统 rCore。

rCore 在单体内核架构下，重点考虑了模块间低耦合构建方法与实现，充分利用 Rust 语言特性，构造可复用内核模块。在此基础上，rCore 实现了 Linux 系统调用层，能够运行基于 musl libc 的原生 Linux 程序。rCore 项目的开发过程验证了测试驱动开发、面向接口编程、持续推进重构等软件工程方法的合理性，并论证了一个操作系统软件遵守行业标准的重要意义。

通过编写 rCore，我们检验了 Rust 语言应用在操作系统场景下的各方面能力。Rust 的底层控制和抽象优化能力满足操作系统的苛刻要求，所有权和生命周期机制大幅减少了系统的 Bug 数量。不过，Rust 还不是一个完美语言，在编写最底层代码时依然存在平台支持不足、语言特性复杂、容易误用导致优化出错等问题，语言对初学者的高门槛也是阻碍 Rust 普及推广的一大难题。

综合来看，我们认为 Rust 语言是编写操作系统的理想选择。

插图索引

图 2.1	Rust 所有权和借用模型	10
图 3.1	rCore 系统的参考和继承关系	15
图 3.2	rCore 的系统架构	15
图 3.3	三种用户线程和内核线程的对应模型	19
图 3.4	线程调度模块的主要数据结构和接口	19
图 3.5	线程切换时的函数调用流程图	20
图 3.6	线程状态转移图	21
图 3.7	网络子系统结构图	31
图 3.8	进程对象结构图	32
图 A-1	x86 虚拟地址结构	50
图 A-2	64 位下虚拟地址到物理地址的转换	51
图 A-3	通过递归映射访问 P1 页表	58
图 A-4	通过递归映射访问 P3 页表	59
图 A-5	通过递归映射访问 P4 页表	59

表格索引

表 1.1	rCore 支持的指令集和物理硬件	5
表 1.2	rCore 后续实现的重要特性及其开发者	6
表 2.1	Rust 语言保证内存安全和线程安全的机制	11
表 3.1	rCore 主要依赖的一些软件包	17
表 3.2	rCore 子系统独立出去的软件包	17
表 3.3	rCore 处理的系统调用清单	27
表 3.4	Socket 和 INode 接口的对比	31
表 3.5	rCore 历史上的大型重构	38

参考文献

- [1] Corporation M. Cve linux kernel vulnerability statistics, 2018[EB/OL]. 2017[2017]. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.
- [2] Oppermann P. Writing an os in rust[EB/OL]. 2019[2019-06-03]. <https://os.phil-opp.com>.
- [3] Evans D. Using rust for an undergraduate os course[EB/OL]. 2014[2014]. <http://rust-class.org/0/pages/using-rust-for-an-undergraduate-os-course.html>.
- [4] Light A. Reenix: Implementing a unix-like operating system in rust[J]. 2015.
- [5] Levy A, Campbell B, Ghena B, et al. Multiprogramming a 64kb computer safely and efficiently [C]//Proceedings of the 26th Symposium on Operating Systems Principles. [S.l.]: ACM, 2017: 234-251.
- [6] Cutler C, Kaashoek M F, Morris R T. The benefits and costs of writing a POSIX kernel in a high-level language[C/OL]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 89-105. <https://www.usenix.org/conference/osdi18/presentation/cutler>.
- [7] Rust programming language[EB/OL]. 2019[2019-05-23]. <https://www.rust-lang.org>.
- [8] The rust programming language book[EB/OL]. 2019[2019-05-23]. <https://doc.rust-lang.org/book/>.
- [9] Yu Chen Y X. ucore 历史[EB/OL]. 2018[2018]. https://chyyuu.gitbooks.io/simple_os_book/content/zh/supplement/ucore-history.html.

致 谢

本文的工作得到了很多老师和同学们的大力支持。

首先感谢导师陈康老师的悉心指导。康总积极乐观的心态使我深受鼓舞，平日里的支持和鼓励都转化为我持续前进的动力！

本文的工作要特别感谢隔壁的陈渝老师。陈老师最早鼓励我完成 Rust OS 的探索，此后持续积极地关注 rCore 项目的进展，对我的工作提出了大量宝贵意见。陈老师是 rCore 发展的引路人。

此外，还要感谢操作系统教学团队的向勇老师，以及参与 rCore 项目开发的 20 余名同学，是你们的支持让 rCore 逐渐成长为今天的模样。其中要特别感谢陈嘉杰、贾越凯、陈晟祺等同学，对 rCore 的开发做出了重大贡献。

最后感谢沈游人师兄、戴臻昶师哥、谭闻德院士，以及王逸松、朱书聪同学，平日与我交流讨论相关工作和技术。这些优秀的同学是我持续奋斗的目标和榜样。

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： 王润基 日 期： 2019.6.24

附录 A 外文资料的书面翻译

用 Rust 语言编写操作系统：页表

原文：Writing an OS in Rust: Page Tables^①

在这篇文章中，我们将创建一个页表模块，用来访问和修改拥有四个层级的页表。我们会探索页表递归映射机制，并利用 Rust 语言的一些特性安全地实现它们。到最后，我们将会实现若干函数，完成虚拟地址转换、映射和取消映射页等功能。

您可以在 GitHub^② 上找到这篇文章以及它的源代码。如果您有任何问题或改进建议，请及时创建 issue。另外本文的末尾还有评论区。请注意，以下内容都需要使用当前最新版本的 Rust nightly 工具链^③。

分页机制

分页是一种内存管理机制，它将内存分为虚拟内存和物理内存两部分。它将地址空间划分为相同大小的页，并用页表来指定每个虚拟页映射到哪个物理帧。对于分页的详细介绍可以参考《操作系统：三大简易元素》^④ 这本书的对应章节^⑤。

x86 架构在 64 位模式下使用一个四级页表。一个虚拟地址有如下结构：

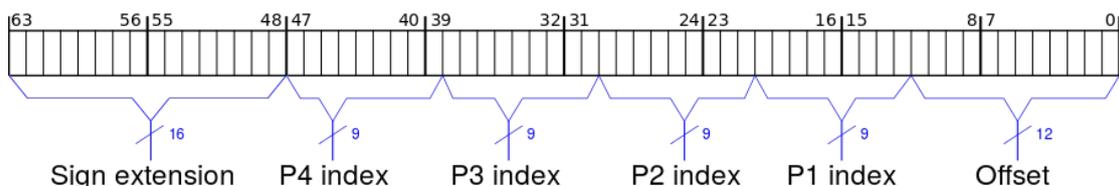


图 A-1 x86 虚拟地址结构

其中第 48-63 位称为符号扩展位，必须和第 47 位保持一致。接下来的 36 位决定了它在页表中的索引（每个表中 9 位），最后 12 位表示它在 4 KiB 页中的

① <https://os.phil-opp.com/page-tables/>

② https://github.com/phil-opp/blog_os/tree/post_6

③ 译者注：也就是文章发表日期 2015-12-09

④ <http://pages.cs.wisc.edu/~remzi/OSTEP/>

⑤ <http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf>

偏移。

每个页表中包含 $2^9 = 512$ 个页表项，每项占用 8 字节。因此一个页表正好占用了 4 KiB 的空间。

在转换一个地址时，CPU 首先从 CR3 寄存器中读取第四级页表 P4 的物理地址。然后使用索引来遍历整个页表：

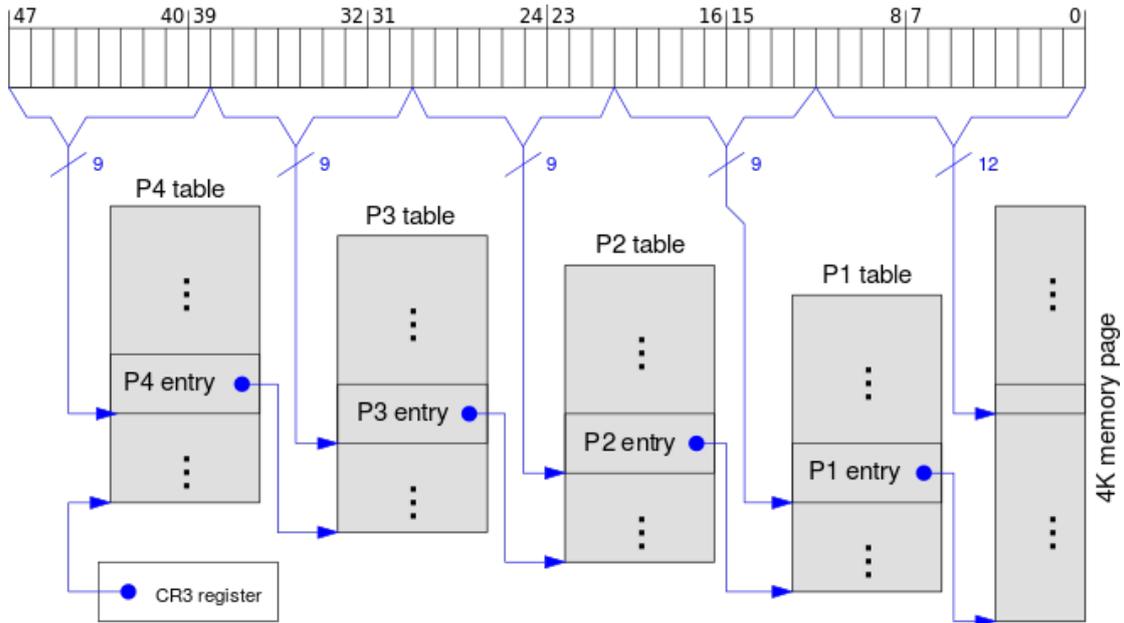


图 A-2 64 位下虚拟地址到物理地址的转换^①

第四级页表项 (P4 entry) 指向一个第三级页表 (P3)，然后用地址中接下来的 9 位选择一个页表项。同理，第三级页表项 (P3 entry) 指向第二级页表 (P2)，第二级页表项 (P2 entry) 指向第一级页表 (P1)。第一级页表项 (P1 entry) 由地址中第 12-20 位所决定，最终指向一个物理帧。^②

一个基础的分页模块

接下来让我们创建一个简单的分页模块，它位于 `memory/paging/mod.rs` 文件中：

```
1 use memory::PAGE_SIZE; // 之后会用到
2
3 const ENTRY_COUNT: usize = 512;
```

^② 译者注：下文中均使用 P1-P4 表示各级页表

```

4
5 pub type PhysicalAddress = usize;
6 pub type VirtualAddress = usize;
7
8 pub struct Page {
9     number: usize,
10 }

```

我们导入 `PAGE_SIZE` 并定义一个常数 `ENTRY_COUNT` 表示每个页表中页表项的个数。为了让后面的函数签名看起来更加直观，我们可以使用类型别名（type aliases）`PhysicalAddress` 和 `VirtualAddress` 表示物理地址和虚拟地址。`Page` 结构和上一篇文章^①中的 `Frame` 结构类似，只是表示的是虚拟页而非物理帧。

页表项

为了给页表项建模，我们创建一个新的子模块 `entry`：

```

1 use memory::Frame; // 之后会用到
2
3 pub struct Entry(u64);
4
5 impl Entry {
6     pub fn is_unused(&self) -> bool {
7         self.0 == 0
8     }
9
10    pub fn set_unused(&mut self) {
11        self.0 = 0;
12    }
13 }

```

我们定义一个全 0 的项为未使用项。这样之后就能把它和不存在项（`present = 0`）区别开。举个例子，以后我们会用一个空闲位表示被换出，即这个页已经被交换到了磁盘上。

^① <https://os.phil-opp.com/allocating-frames/#a-memory-module>

接下来我们将为它包含的物理地址和状态位建模。请记住，页表项有如下格式：

位	名称	含义
0	存在	此页当前在内存中
1	可写	允许向此页中写入数据
2	用户可访问	如果不设置的话，只有内核态可访问此页
3	写直达	直接写入到内存中（不缓存）
4	禁用缓存	对此页不使用缓存
5	访问标记	当此页被访问时，CPU 置此位为 1
6	脏页标记	当此页被写入时，CPU 置此位为 1
7	大页	表示一个 1GiB 或 2MiB 大小的页
8	全局	当切换地址空间时，不会从缓存中清除此页
9-11	空闲	OS 可自由使用
12-51	物理地址	页对齐的 52 位物理地址，表示数据帧或下一级页表
52-62	空闲	OS 可自由使用
63	禁止执行	禁止执行此页中的代码

为了表示众多的状态位，我们将会使用 `bitflags`^① 库。在 `Cargo.toml` 中加入以下代码来将其添加到项目依赖中：

```
1 [dependencies]
2 bitflags = "0.9.1"
```

为了使用其中的宏，我们需要在 `extern crate` 语句前面添加 `#[macro_use]` 标注：

```
1 // in src/lib.rs
2 #[macro_use]
3 extern crate bitflags;
```

现在我们可以表示这些状态位了：

```
1 bitflags! {
2     pub struct EntryFlags: u64 {
3         const PRESENT = 1 << 0;
```

① <https://github.com/rust-lang-nursery/bitflags>

```

4     const WRITABLE =          1 << 1;
5     const USER_ACCESSIBLE = 1 << 2;
6     const WRITE_THROUGH =    1 << 3;
7     const NO_CACHE =         1 << 4;
8     const ACCESSED =         1 << 5;
9     const DIRTY =            1 << 6;
10    const HUGE_PAGE =        1 << 7;
11    const GLOBAL =           1 << 8;
12    const NO_EXECUTE =       1 << 63;
13 }
14 }

```

为了获取页表项中的这些状态位，我们创建一个 `Entry::flags` 方法，它包装了 `from_bits_truncate`^① 函数：

```

1 pub fn flags(&self) -> EntryFlags {
2     EntryFlags::from_bits_truncate(self.0)
3 }

```

这样我们就可以通过 `contains()` 函数来检查状态位。例如，`flags().contains(PRESENT | WRITABLE)` 表示是否同时包含这两个状态。

为了获取物理地址，我们添加一个 `pointed_frame` 方法：

```

1 pub fn pointed_frame(&self) -> Option<Frame> {
2     if self.flags().contains(PRESENT) {
3         Some(Frame::containing_address(
4             self.0 as usize & 0x000ffff_ffff000
5         ))
6     } else {
7         None
8     }
9 }

```

^① <https://doc.rust-lang.org/bitflags/bitflags/index.html#methods-1>

如果这一项是存在的 (present), 我们就 mask 第 12-51 位并返回相应的物理帧。如果这一项不存在, 它就不会指向一个合法的物理帧, 因此我们返回 None。

为了修改页表项, 我们添加一个 set 方法来更新标志位和物理帧:

```
1 pub fn set(&mut self, frame: Frame, flags: EntryFlags)
   ↪ {
2     assert!(frame.start_address() & !0x000ffff_fffff000
   ↪ == 0);
3     self.0 = (frame.start_address() as u64) |
   ↪ flags.bits();
4 }
```

物理帧的起始地址应该是页对齐的, 并且小于 2^{52} (x86 使用 52 位物理地址)。考虑到设置非法地址会搞乱整个页表项, 我们在这里添加一个断言。最后, 把起始地址和标志位进行或运算, 就得到了页表项的实际值。

还剩一个很简单的 `Frame::start_address` 方法:

```
1 use self::paging::PhysicalAddress;
2
3 fn start_address(&self) -> PhysicalAddress {
4     self.number * PAGE_SIZE
5 }
```

我们把它添加到 `memory/mod.rs` 中的 `impl Frame` 语句块中。

页表

为了建模页表, 我们创建一个基础的 `Table` 结构, 放在新的子模块 `table` 中:

```
1 use memory::paging::entry::*;
2 use memory::paging::ENTRY_COUNT;
3
4 pub struct Table {
5     entries: [Entry; ENTRY_COUNT],
6 }
```

它就是一个包含 512 个页表项的数组。

为了让 Table 支持索引语法，我们可以为它实现 Index 和 IndexMut 特性：

```
1 use core::ops::{Index, IndexMut};
2
3 impl Index<usize> for Table {
4     type Output = Entry;
5     fn index(&self, index: usize) -> &Entry {
6         &self.entries[index]
7     }
8 }
9
10 impl IndexMut<usize> for Table {
11     fn index_mut(&mut self, index: usize) -> &mut
12     ↪ Entry {
13         &mut self.entries[index]
14     }
15 }
```

这样就可以通过诸如 `some_table[42]` 获取第 42 个页表项了。当然我们也可以把这里的 `usize` 换成 `u32` 甚至 `u16`，不过这会导致更多的数字类型转换（例如 `x as u16`）。

让我们添加一个方法来将所有页表项置为未使用。它之后会在创建新页表时被用到。这个方法长这样：

```
1 pub fn zero(&mut self) {
2     for entry in self.entries.iter_mut() {
3         entry.set_unused();
4     }
5 }
```

现在我们可以读取页表并获得其中的映射信息，并通过 `IndexMut` 特性和 `Entry::set` 方法更新它们。但该如何引用到其它众多的页表呢？

我们可以读取 CR3 寄存器获取 P4 的物理地址，然后读取它的页表项获取 P3 的地址。P3 的页表项又会指向 P2，以此类推。但这种方法只适用于对等映射的页。我们之后会创建新的页表，到那时就不再是对等映射了。这样我们就无法通过物理地址访问到它们，因此需要一种方法将它们映射到虚拟地址。

映射页表

我们该怎么映射页表自己呢？当前的 P4，P3，P2 页表都位于对等映射区域中，因此不用考虑这个问题，但我们需要为未来访问页表寻找一个方法。

一种解决方案是让所有页表都对等映射。这样我们就不需要区分它们的虚拟和物理地址，访问它们就很简单了。不过这种方法会让虚拟地址空间碎片化。并且创建一个页表也会变得更加麻烦，因为还要保证它所在物理帧对应的页没有被其他人使用。

另一种方法是只暂时映射这个页表。我们把它映射到某个空闲的地址区域，对它进行读写，完成后再恢复。我们可以指定一小片地址空间专门做这件事，对不同的页表重复使用。这种方法只占用少量的虚拟地址空间，因此在 32 位系统下是个很好的方案，因为此时地址空间比较小。但考虑到我们访问一个页时，需要最多临时映射 4 个页，事情就更加复杂了。而且在临时映射一个页时，还需要修改其它页表，它们也需要被映射啊。

我们最终通过另一种方式解决这个问题，它使用了一个叫做递归映射^①的巧妙手段。

递归映射

这里面的诀窍是递归地映射 P4：页表中的最后一项不再指向 P3，而是指向 P4 自己。我们可以通过这种手段去掉一层映射，这样最终就落到了一个页表上。例如，我们可以循环一次来访问 P1：

^① 译者注：也称为自映射

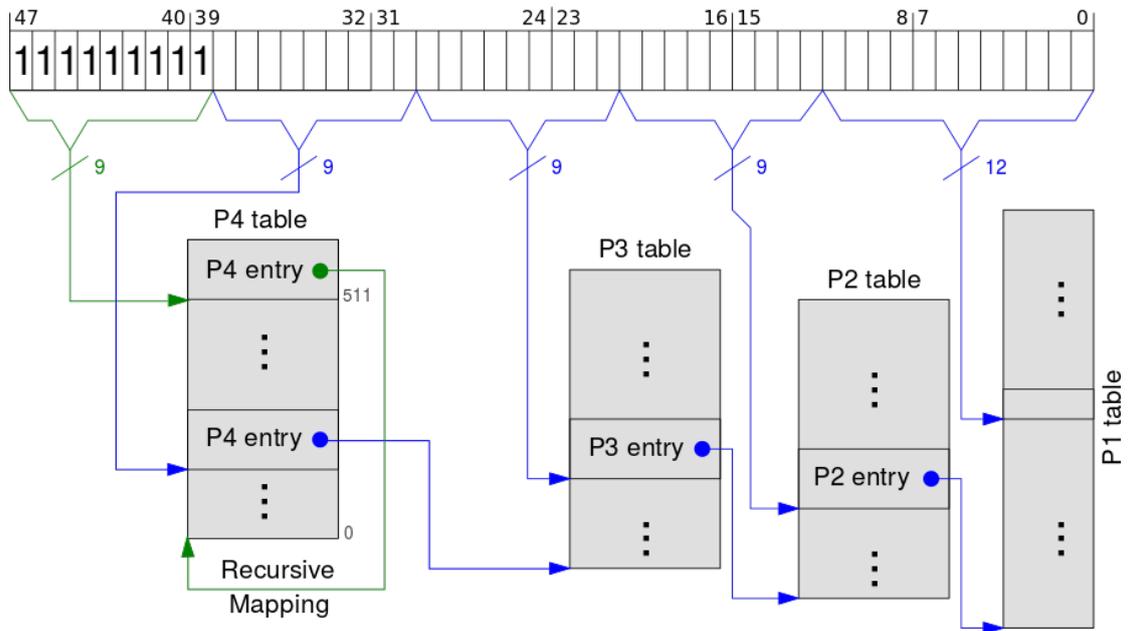


图 A-3 通过递归映射访问 P1 页表

通过选中 P4 中的最后一项（它指向 P4 自己），P4 就可以被当作 P3 使用。类似的，P3 被当成了 P2，P2 被当成了 P1。因此，P1 就成为了映射的目标页，可以通过给定偏移来访问内容。

当然也可以循环两次来访问 P2。如果我们选中最后一项三次，就可以访问和修改 P3 了：

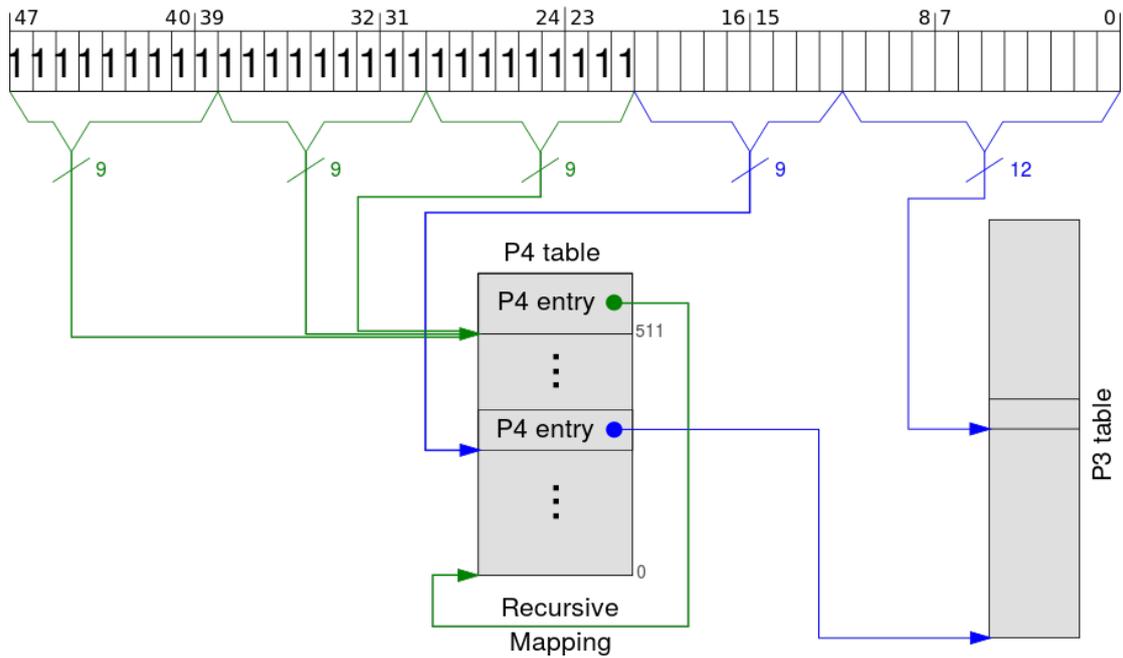


图 A-4 通过递归映射访问 P3 页表

所以只需要在地址的 P1 索引处指定我们想要的 P3 编号就好了。通过多次选中最后一项，我们可以一直停在 P4，直到地址的 P1 索引变成了实际的 P4 索引。

为了访问 P4 自己，我们只需再多循环一次，始终留在同一个物理帧上：

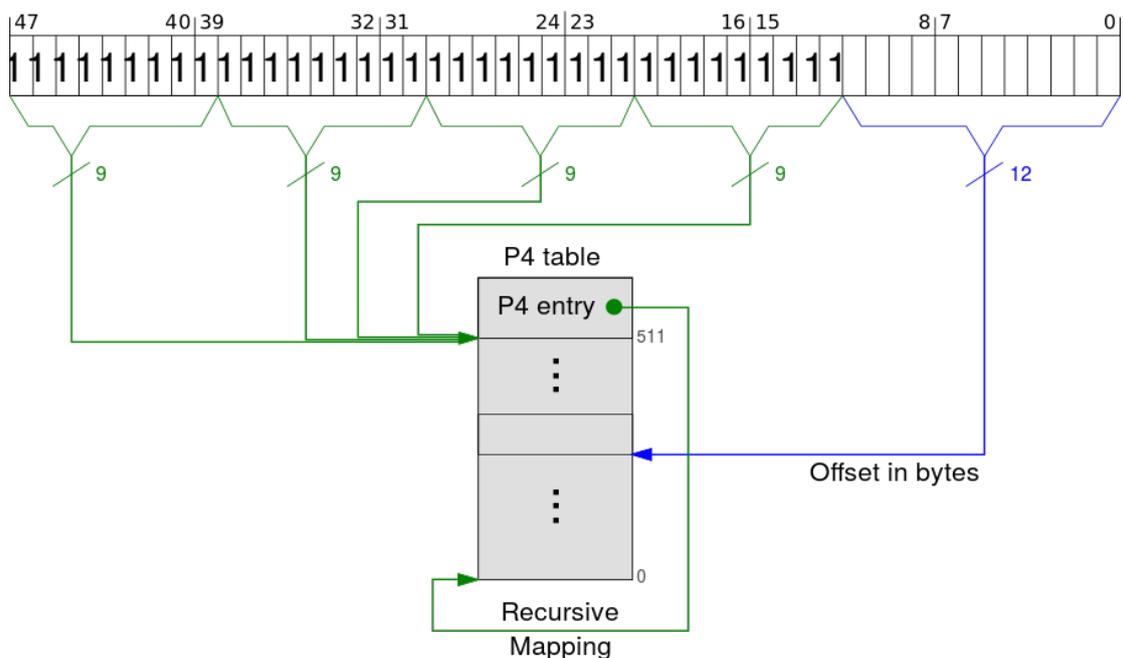


图 A-5 通过递归映射访问 P4 页表

这样我们就可以访问和修改各个层级的页表，只需要设置一个 P4 页表项一次即可。大部分的工作都由 CPU 完成，我们只负责选择递归项来去掉一层或多层映射。这个方法乍一看有些奇怪，但一旦你充分理解了它，就能体会到其中的简洁了。

通过使用递归映射，每个页表都可以经由唯一的虚拟地址访问到。这里做一些简单的数学计算：假设所有页表都被使用了，那么就有 1 个 P4，511 个 P3（最后一项被用作递归映射了）， 511×512 个 P2， $511 \times 512 \times 512$ 个 P1，合计 134217728 个页表。每个页表占用 4 KiB 空间，于是我们需要 $134217728 \times 4\text{KiB} = 512\text{GiB}$ 来保存它们。这正好就是通过一个 P4 页表项可以访问到的空间大小，算一下就知道： $4\text{KiB} \times 512 \times 512 \times 512 = 512\text{GiB}$ 。

当然，递归映射也有一些缺点。它占用了一个 P4 项，也就是 512 GiB 的虚拟地址空间。不过考虑到我们位于长模式（Long mode），有 48 位地址空间，那么仍然有 225.5 TiB 可用。更大的问题其实是我们默认只能修改生效的页表。如果要访问其它页表，递归项就需要被临时替换掉。我们将会在下篇文章中处理这个问题，因为那时我们要切换到新的页表。

实现

为了递归映射 P4 页表，只需要将它的最后一项指向自己。我们在 Rust 中当然可以做这件事，只是这需要操作不安全的指针。而在 boot 汇编代码中加几行要更加容易：

```
1 mov eax, p4_table
2 or eax, 0b11 ; present + writable
3 mov [p4_table + 511 * 8], eax
```

我把它放在了 `set_up_page_tables` 符号后面，但你可以随便把它放在什么地方。

现在我们就可以用一个特殊的虚拟地址来访问页表，其中 P4 位于 `0xfffffffff000`。我们在 `table` 子模块中添加一个常量 P4：

```
1 pub const P4: *mut Table = 0xffffffff_ffff000 as *mut _;
```

下面我们将改用八进制表示它们，因为这能更好地体现出其它特殊地址的意义。上面 P4 的地址用八进制表示就是

00177777_777_777_777_777_0000。可以看到它在所有页表中的索引都是 777，偏移是 0000。最左边的 177777 是符号扩展，它们都是第 47 位的拷贝。这是因为 x86 只使用 48 位虚拟地址。

其它的页表可以通过以下地址被访问到：

页表	地址	索引
P4	00177777_777_777_777_777_0000	-
P3	00177777_777_777_777_XXX_0000	XXX 是 P4 索引
P2	00177777_777_777_XXX_YYY_0000	同上，且 YYY 是 P3 索引
P1	00177777_777_XXX_YYY_ZZZ_0000	同上，且 ZZZ 是 P2 索引

如果仔细看就会发现，P3 的地址等于 $(P4 \ll 9) | XXX_0000$ ，P2 的地址等于 $(P3 \ll 9) | YYY_0000$ 。因此我们可以通过“左移 9 位再加索引”的方法，获得下一级页表的地址。公式表示如下：

$$\text{next_table_address} = (\text{table_address} \ll 9) | (\text{index} \ll 12)$$

next_table 方法

让我们把上面的公式写成一个 Table 的方法：

```

1 fn next_table_address(&self, index: usize) ->
  ↪ Option<usize> {
2     let entry_flags = self[index].flags();
3     if entry_flags.contains(PRESENT) &&
  ↪ !entry_flags.contains(HUGE_PAGE) {
4         let table_address = self as *const _ as usize;
5         Some((table_address << 9) | (index << 12))
6     } else {
7         None
8     }
9 }

```

下一级页表地址合法，仅当它所对应的页表项存在位为 1，并且不是一个大页。然后我们就可以将指针转换成地址，并使用上述公式计算下一级页表的地址。

如果索引越界的话，函数会触发 `panic`，因为 Rust 有数组越界检查。这个 `panic` 是我们期望出现的，因为错误的索引会导致 Bug，它不应该出现。

我们添加两个函数来将地址转换为引用：

```
1 pub fn next_table(&self, index: usize) ->
  ↪ Option<&Table> {
2     self.next_table_address(index)
3         .map(|address| unsafe { &*(address as *const
  ↪ _) })
4 }
5
6 pub fn next_table_mut(&mut self, index: usize) ->
  ↪ Option<&mut Table> {
7     self.next_table_address(index)
8         .map(|address| unsafe { &mut *(address as
  ↪ *mut _) })
9 }
```

我们首先通过 `as` 将地址转换到裸指针，再通过 `&mut *` 将它们转换到 Rust 的引用。后者是不安全操作，因为 Rust 无法保证这个指针是合法的。

需要注意，只要函数返回的引用还存在 (`&mut Table`)，`self` 就会一直保持被借用的状态。这是由 Rust 的生命周期省略^①规则所导致的。简单来说，这些规则规定：默认情况下，函数输出引用的生命周期和输入引用的生命周期保持一致。因此，上面函数的签名可以被扩展为：

```
1 pub fn next_table<'a>(&'a self, index: usize)
2     -> Option<&'a Table> {...}
3
4 pub fn next_table_mut<'a>(&'a mut self, index: usize)
5     -> Option<&'a mut Table> {...}
```

注意那些额外的生命周期参数 `'a`，它对于输入和输出引用是相同的。这正是我们想要的。它保证了我们在持有下级页表的引用时，不能修改当前页表。举例来说，假如我们在写一个 P2 时还能取消它上面的 P3 的映射，那就很不妙了。

^① <https://doc.rust-lang.org/book/lifetimes.html#lifetime-elision>

安全性

现在我们可以从常量 P4 开始，使用 `next_table` 函数依次访问下级页表。我们甚至不需要 `unsafe` 就可以做到这些！不过这里有必要敲一个警钟。多亏了 Rust，到此为止我们做的一切都是完全安全的。不过我们刚刚引入了两个 `unsafe` 块来告诉 Rust：某个特殊的地址上存在合法的页表。我们真能确定这一点吗？

首先，这些地址合法的前提条件是 P4 页表实现了递归映射。考虑到 `paging` 模块是唯一可以修改页表的地方，我们可以为它引入一个约束：

> 活跃页表 P4 的最后一项（第 511 项）一定被映射到它自己。

因此当我们切换到另一个 P4 页表的时候，需要它也提前完成递归映射。只要我们遵守这一约束，就可以安全地使用那些特殊地址。然而即便如此，上面两个函数仍存在一个很大的问题：

> 假如我们在 P1 上调用它们会发生什么？

好吧，此时函数会计算下一级页表的地址（然而并不存在），然后把它当作一个页表。这会导致要么构造出一个非法地址^①（如果 `xxx < 400`），要么访问到被映射的数据页本身。这样的话，我们就会轻易地破坏内存，或者意外地触发 CPU 异常。因此按照 Rust 规范，这两个函数是不安全的。我们就不得不将它们标记为 `unsafe`，除非我们能找到更优雅的方案。

一种优雅的方案

我们可以利用 Rust 的类型系统来静态地（在编译时）保证 `next_table` 方法只能被 P4, P3, P2 页表调用，而不能被 P1 页表调用。这一想法是向 `Table` 类型中加入 `Level` 参数表示页表的层级，并只对 `Level4`、`Level3` 和 `Level2` 实现 `next_table` 方法。

我们使用特性（`trait`）和空枚举（`empty enums`）对层级建模：

```
1 pub trait TableLevel {}
2
3 pub enum Level4 {}
4 pub enum Level3 {}
```

^① 如果地址的 `xxx` 部分小于 `0o400`，那么它的二进制表示中最高位就不是 1。然而地址中的符号扩展位都是 1，与最高位不一致。因此这个地址是非法的。

```

5 pub enum Level2 {}
6 pub enum Level1 {}
7
8 impl TableLevel for Level4 {}
9 impl TableLevel for Level3 {}
10 impl TableLevel for Level2 {}
11 impl TableLevel for Level1 {}

```

空枚举类型的大小是 0，并且会在编译后完全消失。和空结构不同，空枚举不可被实例化^①。由于我们会在导出类型中使用 `TableLevel` 和 `Level*`，因此它们必须是 `public` 的。

为了将 P1 页表和其它页表区别开，我们引入 `HierarchicalLevel`（中间级）特性，它是 `TableLevel` 的子特性。我们只对 `Level4`、`Level3` 和 `Level2` 实现这一特性：

```

1 pub trait HierarchicalLevel: TableLevel {}
2
3 impl HierarchicalLevel for Level4 {}
4 impl HierarchicalLevel for Level3 {}
5 impl HierarchicalLevel for Level2 {}

```

现在我们为 `Table` 类型添加层级参数：

```

1 use core::marker::PhantomData;
2
3 pub struct Table<L: TableLevel> {
4     entries: [Entry; ENTRY_COUNT],
5     level: PhantomData<L>,
6 }

```

我们需要添加一个幻影数据（*PhantomData*）^② 字段，因为 `Rust` 不允许出现未使用的类型参数。

由于我们修改了 `Table` 类型的定义，它每一个被用到的地方也需要做修改：

^① 译者注：不能构造一个此类型的变量

^② <https://doc.rust-lang.org/core/marker/struct.PhantomData.html#unused-type-parameters>

```

1 pub const P4: *mut Table<Level4> = 0xffffffff_ffff000 as
    ↪ *mut _;
2 ...
3 impl<L> Table<L> where L: TableLevel {
4     pub fn zero(&mut self) {...}
5 }
6
7 impl<L> Table<L> where L: HierarchicalLevel {
8     pub fn next_table(&self, index: usize)
9         -> Option<&Table<???\>> {...}
10    pub fn next_table_mut(&mut self, index: usize)
11        -> Option<&mut Table<???\>> {...}
12    fn next_table_address(&self, index: usize)
13        -> Option<usize> {...}
14 }
15
16 impl<L> Index<usize> for Table<L>
17     where L: TableLevel {...}
18
19 impl<L> IndexMut<usize> for Table<L>
20     where L: TableLevel {...}

```

现在 `next_table` 方法只能被 P4, P3, P2 页表使用了。但它们的返回值类型 `Table<???\>` 还不完全。我们该往 `???` 中填什么呢？

对于 P4 页表我们期望返回 `Table<Level3>`，对于 P3 返回 `Table<Level2>`，对于 P2 返回 `Table<Level1>`。也就是我们希望返回下一级的页表。

我们可以通过为 `HierarchicalLevel` 特性添加关联类型 (*associated type*) 来定义下一级是什么：

```

1 trait HierarchicalLevel: TableLevel {
2     type NextLevel: TableLevel;
3 }

```

```

4  impl HierarchicalLevel for Level4 {
5      type NextLevel = Level3;
6  }
7  impl HierarchicalLevel for Level3 {
8      type NextLevel = Level2;
9  }
10 impl HierarchicalLevel for Level2 {
11     type NextLevel = Level1;
12 }

```

现在我们可以把 `Table<???` 类型替换成 `Table<L::NextLevel>`，并且代码会按照我们期望的方式运行。你可以通过一个简单的函数来测试它：

```

1  fn test() {
2      let p4 = unsafe { &*P4 };
3      p4.next_table(42)
4          .and_then(|p3| p3.next_table(1337))
5          .and_then(|p2| p2.next_table(0xdeadbeaf))
6          .and_then(|p1| p1.next_table(0xcafebabe))
7  }

```

上面的大部分索引都是越界的，因此当调用它会触发 `panic`。然而我们都不需要调用它测试，因为它在编译时就已经报错了：

```

1 error: no method named `next_table` found for type
2   `&memory::paging::table::Table<memory::paging::table::
3   Level1>`
4   in the current scope
5 错误：类型 `&memory::paging::table::Table<memory::paging::
6   table::Level1>`
   在当前作用域中找不到 `next_table` 方法

```

要记得这可是裸机内核代码。我们刚刚通过类型系统的魔法，让底层页表操作更加安全。Rust 真是太棒了！

地址转换

现在让我们在新的模块中做一些有用的事情。我们将会创建一个函数，将虚拟地址转换到对应的物理地址。我们把它加入 `paging/mod.rs` 模块中：

```
1 pub fn translate(virtual_address: VirtualAddress)
2     -> Option<PhysicalAddress>
3 {
4     let offset = virtual_address % PAGE_SIZE;
5
6     ↪ translate_page(Page::containing_address(virtual_address))
7     .map(|frame| frame.number * PAGE_SIZE +
8     ↪ offset)
9 }
```

这里面用到了两个我们还没有定义的函数：`translate_page` 和 `Page::containing_address`。我们先从后一个开始：

```
1 pub fn containing_address(address: VirtualAddress) ->
2     ↪ Page {
3     assert!(address < 0x0000_8000_0000_0000 ||
4     address >= 0xffff_8000_0000_0000,
5     "invalid address: 0x{:x}", address);
6     Page { number: address / PAGE_SIZE }
```

这里需要一个断言，因为传入的地址可能是非法的。x86 中的地址只有 48 位长，其它位只是符号扩展，也就是最高位的拷贝。举个例子：

1 非法地址：0x0000_8000_0000_0000
2 合法地址：0xffff_8000_0000_0000

于是整个地址空间被分为两半：高一半包含所有符号扩展为 1 的地址，低一半包含所有符号扩展为 0 的地址。中间的所有地址都是非法的。

既然已经有了 `containing_address`，我们也加入相反的过程（也许之后会用到）：

```

1 fn start_address(&self) -> usize {
2     self.number * PAGE_SIZE
3 }

```

另一个函数 `translate_page` 看起来是这样的：

```

1 use memory::Frame;
2
3 fn translate_page(page: Page) -> Option<Frame> {
4     use self::entry::HUGE_PAGE;
5
6     let p3 = unsafe { &*table::P4
↪ }.next_table(page.p4_index());
7
8     let huge_page = || {
9         // TODO
10    };
11
12    p3.and_then(|p3| p3.next_table(page.p3_index()))
13        .and_then(|p2| p2.next_table(page.p2_index()))
14        .and_then(|p1|
↪ p1[page.p1_index()].pointed_frame())
15        .or_else(huge_page)
16 }

```

我们使用一个 `unsafe` 块将 `P4` 指针转化为引用，然后使用 `Option::and_then`^① 函数来遍历四级页表。如果过程中某一个页表项是 `None`，我们就通过 `huge_page` 闭包函数（尚未实现）来检查它是不是一个大页。

`Page::p*_index` 这些函数返回不同层级页表的索引。它们看起来是这样的：

```

1 fn p4_index(&self) -> usize {
2     (self.number >> 27) & 0o777

```

① https://doc.rust-lang.org/nightly/core/option/enum.Option.html#method.and_then

```

3 }
4 fn p3_index(&self) -> usize {
5     (self.number >> 18) & 0o777
6 }
7 fn p2_index(&self) -> usize {
8     (self.number >> 9) & 0o777
9 }
10 fn p1_index(&self) -> usize {
11     (self.number >> 0) & 0o777
12 }

```

安全性

我们使用 `unsafe` 块将裸指针 `P4` 转化为一个共享引用。这是安全的，因为除此之外我们既没有创建 `&mut` 引用，也没有切换到其它的 `P4` 页表。不过一旦我们做了这些，就有必要重新审视这个方法。

大页

`huge_page` 闭包函数会在使用大页的情况下，计算对应的物理帧地址。它看起来是这样的：

```

1 p3.and_then(|p3| {
2     let p3_entry = &p3[p3_index()];
3     // 1GiB 页?
4     if let Some(start_frame) = p3_entry.pointed_frame()
5     ↪ {
6         if p3_entry.flags().contains(HUGE_PAGE) {
7             // 地址必须 1GiB 对齐
8             assert!(start_frame.number % (ENTRY_COUNT *
9             ↪ ENTRY_COUNT) == 0);
10            return Some(Frame {
11                number: start_frame.number +
12                ↪ page.p2_index() *
13                ENTRY_COUNT + page.p1_index(),

```

```

11         });
12     }
13 }
14 if let Some(p2) = p3.next_table(page.p3_index()) {
15     let p2_entry = &p2[page.p2_index()];
16     // 2MiB 页?
17     if let Some(start_frame) =
↪ p2_entry.pointed_frame() {
18         if p2_entry.flags().contains(HUGE_PAGE) {
19             // 地址必须 2MiB 对齐
20             assert!(start_frame.number %
↪ ENTRY_COUNT == 0);
21             return Some(Frame {
22                 number: start_frame.number +
↪ page.p1_index()
23                 });
24         }
25     }
26 }
27 None
28 })

```

这个函数比起 `translate_page` 本身就复杂得多了。为了在以后避免这一复杂性，我们从现在开始只考虑 4 KiB 页的情况。

总结

这篇文章真是足够长了。让我们总结一下都做了哪些事情：

- 我们创建了一个页表模块，定义了页表和页表项
- 我们为 P4 页表建立了递归映射，并创建了 `next_table` 方法
- 我们使用空枚举和关联类型，保证了 `next_table` 函数的安全
- 我们写了一个函数来将虚拟地址转化为物理地址
- 我们创建了安全的函数来映射页和取消映射页

- 并且我们修复了栈溢出和 TLB 相关 Bug

下集预告

在下一篇文章^①中我们将扩展这个模块，为其添加一个函数来修改不活跃页表。通过这个函数，我们将创建一个新的页表，它使用 4 KiB 页妥善地映射内核部分的内存空间。然后我们会切换到这个新的页表，使得内核运行环境更加安全。

然后，我们将会用当前分页模块来构造一个堆分配器。这样我们就可以动态分配内存空间，从而使用诸如 Box 和 Vec 等容器类型。

^① <https://os.phil-opp.com/remap-the-kernel/>

综合论文训练记录表

学生姓名	王润基	学号	2015011279	班级	计53
论文题目	Rust语言操作系统的设计与实现				
主要内容以及进度安排	<p>使用新兴的系统级编程语言Rust实现一个简单的操作系统。</p> <p>目前的已有工作是参考uCore完成了一个基础框架。接下来的任务是对其内存管理、进程管理、文件系统部分的功能进行完善，实现mmap、共享内存、多核调度算法、IPC机制、文件缓存和挂载等功能。目标是能在x86平台物理机上运行，支持实际的用户程序。最后通过此项目检验Rust语言的高性能、高安全、高抽象特性，为其编写操作系统探索和积累经验。</p> <p>进度安排如下：</p> <ul style="list-style-type: none"> • 第一学期内：文献和源码阅读 • 第2周：完成内存部分功能 • 第4周：完成进程部分功能 • 第6周：完成文件部分功能 • 第8周：完成真机调试 • 第14周：完善系统调用与性能优化 • 第16周：完成论文 <div style="text-align: right; margin-top: 20px;"> <p>指导教师签字： </p> <p>考核组组长签字： </p> <p>2019年1月15日</p> </div>				
中期考核意见	<p style="font-size: 24px; font-weight: bold;">符合预期。</p> <div style="text-align: right; margin-top: 20px;"> <p>考核组组长签字： </p> <p>2019年4月3日</p> </div>				

<p>指导教师评语</p>	<p>使用新的语言尝试架构一个新的操作系统是一项有益的尝试，该工作具有很大的工作量，该生按时完成任务，达到了本科综合论文训练的要求</p> <p style="text-align: right;">指导教师签字: </p> <p style="text-align: right;">2019年6月2日</p>
<p>评阅教师评语</p>	<p>使用 Rust 语言, 参照 μCore, 本文设计实现了一个简单的操作系统, 支持内存共享、多核调度、文件挂载等功能。论文书写规范, 任务有难度和很大的工作量, 完成结果良好, 达到了本科综合论文训练的要求。</p> <p style="text-align: right;">评阅教师签字: </p> <p style="text-align: right;">2019年6月14日</p>
<p>答辩小组评语</p>	<p>该生在综合论文训练期间用 Rust 语言实现了一个新的操作系统。该项工作难度和工作量大, 任务完成的好; 答辩过程中讲述清楚, 问题回答准确, 达到了论文训练的要求, 是一篇优秀的本科综合训练论文。</p> <p style="text-align: right;">答辩小组组长签字: </p> <p style="text-align: right;">2019年6月17日</p>

总成绩: 95

教学负责人签字:



2019年6月19日