

北京理工大学

本科生毕业设计(论文)

rCore 模块化改进的设计与实现

Design and Implementation of rCore Modular Improvement

学 院： 计算机学院

专 业： 计算机科学与技术

班 级： 07111703

学生姓名： 石文龙

学 号： 1120173592

指导教师： 陆慧梅

2023 年 5 月 23 日

原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名: _____ 日期: _____ 年 月 日

关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名: _____ 日期: _____ 年 月 日

指导老师签名: _____ 日期: _____ 年 _____ 月 _____ 日

rCore 模块化改进的设计与实现

摘 要

rCore 操作系统是一个用 Rust 语言写的基于 RISC-V 基础架构的类 linux 教学操作系统，其每个章节都是一个历史发展进程中的操作系统。目前在用 Rust 系统编程语言写 rCore 操作系统内核的实验中，前一章实验的成果作为后一章实验的基础，在进行后一章实验时必须要把前一章的代码复制过来，代码的重复利用率很低，完成 rCore 操作系统的模块化能够显著提高代码的重复利用率。

本文首先调研了 Rust 编程、RISC-V 处理器的相关内容，配置了本文工作所需环境，即 Linux 操作系统、Rust 开发环境、Qemu 模拟器和 VSCode 编译器。本文对实验室模块化的代码进行分析与复现，将模块化之前每章都要重复实现的内容封装为一个可以被所有章节的内核调用的模块，以提高代码的重复利用率。本文亦为各功能模块增加用户态单元测试的代码，使开发者在一个模块内部能进行用户态单元测试，相当于将一个完成操作系统内核的实验分成了用户态部分的实验和内核态部分的实验，能够降低其他想要完成此实验的同学的工作难度。本文亦将不同仓库中的模块组合成一个不断完善的教学操作系统。

关键词：Rust；操作系统；模块化；单元测试

Design and Implementation of rCore Modular Improvement

Abstract

The rCore operating system is a Linux-like teaching operating system written in the Rust language based on RISC-V infrastructure. Each chapter is an operating system in a historical development process. Currently, in experiments using the Rust system programming language to write the rCore operating system kernel, the results of the previous chapter's experiments serve as the basis for the subsequent chapter's experiments. When conducting the next chapter's experiments, it is necessary to copy the code from the previous chapter, The code reuse rate is very low, and completing the modularization of the rCore operating system can significantly improve the code reuse rate.

This article first investigated the relevant content of Rust programming and RISC-V processors, and configured the required environment for this article's work, namely Linux operating system, Rust development environment, Qemu simulator, and VSCode compiler. This article analyzes and replicates the modular code in the laboratory, encapsulating the content that needs to be repeatedly implemented in each chapter before modularization into a module that can be called by the kernel of all chapters to improve code reuse. This article also adds user mode unit testing code for each functional module, enabling developers to conduct user mode unit testing within a module. This is equivalent to dividing an experiment that completes the operating system kernel into user mode and kernel mode experiments, which can reduce the difficulty of other students who want to complete this experiment. This article also combines modules from different warehouses into a continuously improving teaching operating system.

Key Words: Rust; operating system; modular; unit test

目 录

摘 要	I
Abstract	II
第 1 章 绪论	1
1.1 研究背景	1
1.2 研究现状	1
1.3 研究意义	2
1.4 研究内容	3
1.5 论文结构	3
第 2 章 rCore 模块化的相关技术及准备工作	5
2.1 Rust 语言	5
2.2 RISC-V 系统结构	5
2.3 rCore 操作系统	6
2.4 桩函数	6
2.5 配置实验环境	7
第 3 章 rCore 模块化的分析	9
3.1 rCore 整体分析	9
3.2 console 模块的分析	10
3.3 linker 模块的分析	10
3.4 kernel-context 模块的分析	12
3.5 kernel-alloc 模块的分析	13
3.6 kernel-vm 模块的分析	14
3.7 syscall 模块的分析	16
3.8 task-manage 模块的分析	17
3.9 easy-fs 模块的分析	19
3.10 signal-defs、signal 和 signal-imple 模块的分析	22
3.11 sync 模块的分析	24
第 4 章 rCore 模块化的测试	27
4.1 console 模块的用户态单元测试	27

4.2 linker 模块的用户态单元测试	28
4.3 kernel-context 模块的用户态单元测试	29
4.4 kernel-alloc 模块的用户态单元测试	29
4.5 kernel-vm 模块的用户态单元测试	30
4.6 task-manage 模块的用户态单元测试	31
4.7 easy-fs 模块的用户态单元测试	32
4.8 signal-imple 模块的用户态单元测试	33
4.9 sync 模块的用户态单元测试	34
结 论	36
参考文献	37
致 谢	38

第 1 章 绪论

1.1 研究背景

中国的操作系统市场以 Windows 的闭源架构作为主导，但以 Linux 为代表的开源操作系统正在不断对其进行挑战，于此同时，操作系统国产化的浪潮也正在兴起^[6]。当前时期是操作系统国产化的风口时期，也是 Linux 等开源系统蓬勃发展的时期。在此阶段对操作系统进行学习与研究，即有利于国家政策的实施，亦更易获得外界的支持。

Rust 是一种相对年轻的编程语言，直到 2015 年 Rust 的 1.0 版本才被发布，但 Rust 的受欢迎程度丝毫没有受到影响。根据超过 65000 名开发人员的反馈，Rust 已连续五年被列为 Stack Overflow 上“最受欢迎”的编程语言。与此同时，Rust 在 Redmonk 的编程语言排名中也跻身前 20 名之列。考虑到 Java、C 和 JavaScript 等语言几乎不可动摇的地位，这无疑是 Rust 的一项重大成就。

Rust 语言致力于解决高并发性和高安全性问题。目前，开源社区中的许多开发人员都试图开发基于 Rust 语言的操作系统，清华大学的教学操作系统 rCore 的实现就具有很好的代表性^[2]。本文基于清华大学的 rCore 进行研究，因为 rCore 是一个逐步完善的操作系统能够每完成一个阶段就出一个阶段成果，且清华大学关于 rCore 有一个夏令营活动，笔者以 rCore 为基础遇到问题时能够获得更多的帮助。

1.2 研究现状

Rust 语言能够在追求速度的同时追求稳定性，这表明用 Rust 语言重写操作系统的优势非常明显。2021 年 3 月，允许在 Linux 内核中使用 Rust 程序的初始基础架构落入 Linux-Next 树，以便在 Rust 包含在主线内核中之前，进行更广泛的测试。4 月，Linux 内核邮件列表上围绕 Linux 内核的 Rust 代码的前景启动了“请求评论”。

谷歌安卓团队工程师韦森·阿尔梅达·菲略亦发布了对“将 Rust 引入 Linux 内核”的看法，韦森·阿尔梅达·菲略表示：因为 C 语言高标准的代码审查和精细的防护措施，近半个世纪以来，C 语言一直作为编写内核的首选语言。但内存安全漏洞还是经常发生。谷歌安卓团队认为 Rust 现在已经能够实现内核开发。引入 Rust 可以帮助减少特权代码中潜在 BUG 和安全漏洞，同时保留其性能特征。

在于以色列特拉维夫召开的 BlueHat IL 2023 大会上，微软企业和操作系统安全副总裁 David Weston 介绍了他们正对 Win11 进行的内核级改造，即宣布 Rust 将正式入驻 Windows 系统内核。Pydantic 公司创始人兼 Python/Rust 开发者 Samuel Colvin 表示“对于我们这些依赖 Rust 的开发者来说，微软使用并支持 Rust 的决定真的很令人兴奋。”微软的认可、支持以及代码贡献正在令 Rust 变得愈发强大。

1.3 研究意义

在 rCore 教学操作系统中，随着操作系统的功能实现的增多，其复杂程度亦在增加。值得注意的是在对操作系统的学习过程中，各个章节是被 git 分支隔离开的，因此完成前一个章节的实验后，在下一个章节有关前一个章节的实现内容需要复制代码过来，同样的代码用一次就需要写一次，此外如果改了某一章节的内容，后续的所有章节相同的地方都需要重新改一边，代码的复用性差。为了解决上述的问题，进行了 rCore 操作系统的模块化设计与实现。

rCore 操作系统的模块化编程能够很好的解决上述问题，之前做不同章节的课后实验需要切换到对应的 lab 分支去写代码，现在只需要封装一个 crate 加入 workspace 就可以了。模块化之后，能够以最少的代码实现尽可能多的个性化设计，之前每个实验都会有大量的代码需要重复的写在每一个章节中，模块化之后这些重复的代码就可以直接引用了，如第二章的批处理系统同样需要第一章的 print 宏的实现，现在可以通过复用 console 模块来避免重复编写代码，减少了开发时间和开发成本，大大提高了代码的复用性。同时能够防止修改了前一章节中某一模块的内容，然而后续章节没有同步修改造成错误的情况出现，减少了开发者出错的可能性，提高了实验项目的可维护性^[1]。

每个模块实际上亦是一个完整的项目，可以进行单独的编译、调试。将 rCore 操作系统的每个模块看作一个独立的单元进行测试，不仅减少测试的时间和成本，同时，为每个模块增加单元测试提高了操作系统的可维护性。当其他的开发者想要对一个独立的模块的接口的实现进行修改时，或者当操作系统出现问题时，在对应的模块中完成了调试或修改之后，开发者不再需要将修改后的模块放入整个操作系统内核中来测试自己的修改是否正确，可以直接通过这个模块的单元测试来判断自己的修改是否有错误，能够让其他开发者更专注于一个模块中的内容，方便了其他

开发者实现一个模块的接口。

1.4 研究内容

本文理解了实验室当前对内核模块化的成果，即将所有的章节中复用的代码形成了单独 package，各个章节对于这些复用的代码只需要在 `cargo.toml` 的依赖中加上需要使用到的 package 就可以了，不需要再像之前一样需要将这些已经写过的代码在每一章节中都重新写一遍。

且本文成功利用了 Rust 语言的 workspace 和 crate，使得每一个章节是一个预期目标不同的 package，做不同的章节实验的时候，只需要封装一个 crate 到对应的 package 中，在运行的时候运行指定的 package，不需要像之前一样做不同章节的课后实验需要到不同的分支中写代码。

此外本文实现了系统调用接口的模块化，即系统调用的分发封装到一个 crate 中，这个 crate 就是 `syscall/src/kernel/mod.rs`。使得添加系统调用的模式不是为某个 match 增加分支，而是实现一个分发库要求的 trait 并将实例传递给分发库^[1]。

与此同时，本文在实验室工作的基础上完善实验室现有的模块化的 rCore 操作系统内核。针对目前实验室的版本只是实现了操作系统内核模块化的基本功能的问题，提出了在用户态对每一个模块进行单元测试的优化方向。增加了单元测试之后，能够在隔离的环境中一次测试一个模块或者测试模块的接口，能够更加方便的检测出来是代码的哪个模块甚至是哪个接口出了问题。将增加测试前的一个写 rCore 内核的复杂任务分解为了写用户态部分的任务和写内核态的任务，能够降低其他开发者的工作难度。

1.5 论文结构

本文的结构安排如下：

第一章介绍了 rCore 模块化的研究背景与意义，并对本文涉及的相关工作、国内外研究现状进行了综述，最后介绍了本文的研究内容与贡献。

第二章介绍了本文研究内容的相关技术及准备工作，包括 Rust 语言、RISC-V、rCore 操作系统、桩函数及配置实验环境。

第三章分析了 rCore 操作系统的整体结构、console 模块、linker 模块、

kernel-context 模块、kernel-alloc 模块、kernel-vm 模块、task-manage 模块、easy-fs 模块、signal-defs 模块、signal 模块、signal-imple 模块及 sync 模块。

第四章介绍了 console 模块、linker 模块、kernel-context 模块、kernel-alloc 模块、kernel-vm 模块、task-manage 模块、easy-fs 模块、signal-imple 模块及 sync 模块的用户态单元测试的设计思路及实现方法。

结论总结全文，展望了本文工作在未来的发展方向。

第 2 章 rCore 模块化的相关技术及准备工作

2.1 Rust 语言

在编程语言设计中，上层编程效率和下层细粒度控制往往无法同时实现，Rust 则能挑战这一困难。因此 Rust 语言与其他编程语言相比，Rust 能够编写更快、更稳定的软件。

Rust 努力将高级编程语言特性编译成低级代码，并以与手写代码相同的速度运行，这个概念被称为零成本抽象，该概念可以使代码获得高级编程语言的编写速度及人工低级语言的运行速度。

Rust 编译器会检查代码以确保稳定性。因此 Rust 是高效的协作工具，在多人进行协作开发时，开发者所拥有的不同层次系统编程知识，会导致底层代码中出现不易察觉的漏洞，对其他编程语言而言，为寻找漏洞，只能加强开发人员对代码的审查力度，并设计大量测试。但在 Rust 中，编译器扮演着守门员的角色，若代码中存在漏洞，例如并发错误，编译器将拒绝编译。只要使用 Rust 语言，团队就可以花更多的时间专注于程序逻辑，而不用担心发现错误。

2.2 RISC-V 系统结构

RISC-V 是一种基于精简指令集（RISC）原理的开源指令集体系结构（ISA）。与大多数 ISA 不同，RISC-V 指令集体系结构可以在所有设备中免费使用，允许任何人设计、制造和销售 RISC-V 芯片和软件。RISV 是第一个可以根据特定场景选择适当指令集的指令集架构。

RISC-V 的体系结构简单。处理器领域的主流架构是 x86 和 ARM 架构，x86 和 ARM 体系结构的发展亦伴随着现代处理器架构技术的不断发展和成熟。然而作为商业架构，为了保持向后兼容性，x86 和 ARM 架构不得不保留许多过时的定义，导致拥有大量的指令、严重的指令冗余和大量的文档。因此，在这些体系结构上开发新的操作系统或直接开发应用程序的门槛很高。然而，RISC-V 体系结构可以完全消除负担，并利用计算机体系结构已经成为一项相对成熟的技术的优势，使 RISC-V 体系结构重量轻且有效。RISC-V 基本指令集加上其他模块化扩展指令总共有几十条指令。

现代操作系统已经将特权级指令与用户级指令分离。特权指令只能由操作系统

调用，而用户级指令只能在用户模式下调用，确保了操作系统的稳定性。RISC-V 同时提供特权级和用户级指令，以及有关 RISC-V 特权级指令规范和 RISC-V 用户级指令规范的详细信息，便于开发人员将 Linux 和 Unix 系统移植到 RISC-V 平台。

2.3 rCore 操作系统

rCore 操作系统是由 Rust 语言实现的，以简洁的 RISC-V 基本架构为底层硬件基础的，遵循操作系统开发的历史背景的一个逐步完善的操作系统。rCore 操作系统这个系统软件的功能是向下管理 CPU、内存和各种外设等硬件资源，并形成软件执行环境来向上管理和服务应用软件。rCore 操作系统的主要组成部分包括：

- (1) 内存管理：内核负责管理系统的内存，分配和回收内存空间，并保证进程之间的内存隔离。
- (2) 进程/线程管理：内核负责管理系统中的进程或线程，创建、销毁、调度和切换进程或线程。
- (3) 文件系统：内核提供文件系统接口，负责管理存储设备上的文件和目录，并允许应用访问文件系统。
- (4) 同步互斥：内核负责协调多个进程或线程之间对共享资源的访问。同步功能主要用于解决进程或线程之间的协作问题，互斥功能主要用于解决进程或线程之间的竞争问题。
- (5) 系统调用接口：内核为应用程序提供了访问系统服务的入口点，应用程序通过系统调用接口调用操作系统提供的服务，如文件系统、进程管理等。

2.4 桩函数

桩函数是指用来代替关联函数或者未实现函数的函数。桩函数能够实现隔离、补齐和控制等功能。隔离功能是指用桩函数来代替测试任务之外的与测试任务相关的代码，以达到分离测试任务的目的。补齐功能是指要测试的函数 `func_a` 调用了函数 `func_b`，但如果 `func_b` 没有实现，则可以用桩函数来代替函数 `func_b`，使函数 `func_a` 能够运行并测试。控制功能是指在测试过程中，手动设定相关的代码行为，以满足测试需求。桩函数原型应该与原函数保持一致，只是实现不同，这样测试代码才能正确链接到桩函数。

本文使用桩函数的目的是隔离和控制，例如在 `console` 模块中为了运行并测试

函数 `put_char()` 就用桩函数 `print` 宏代替了 `ch1` 模块对 `put_char()` 的实现函数 `console_putchar()`。

2.5 配置实验环境

在用 Rust 写 rCore 操作系统之前，首先需要完成环境配置并成功运行 rCore-Tutorial。整个流程分为下面几个部分：OS 环境配置、Rust 开发环境配置、Qemu 模拟器安装、其他工具安装、试运行 rCore-Tutorial。

本文在电脑上配置了一个 ubuntu 双系统。在配置好 ubuntu 系统之后，需要配置 Rust 开发环境。首先安装 Rust 版本管理器 Rustup 和 Rust 包管理器 cargo，可以使用官方的安装脚本：

```
curl https://sh.rustup.rs -sSf | sh
```

在安装过程中可能因为网络问题通过命令行下载脚本失败，因此最好设置科学上网代理，安装过程中全部选择默认选项即可。安装完成后重新打开一个终端来让新设置的环境变量生效，最后通过：

```
Rustc -version
```

确认是否正确安装了 Rust 工具链。完成 Rust 开发环境的配置之后，可以通过 Visual Studio Code 搭配 Rust-analyzer 和 RISC-V Support 插件来进行代码阅读和开发。

其次下载并安装 Qemu 模拟器，需要使用 Qemu 7.0.0 以上版本进行实验，为此，从源码手动编译安装 Qemu 模拟器，如图 2-1 所示：

```
# 安装编译所需的依赖包
sudo apt install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev \
gawk build-essential bison flex texinfo gperf libtool patchutils bc \
zlibg-dev libexpat-dev pkg-config libglib2.0-dev libpixman-1-dev git tmux \
python3 ninja-build

# 下载源码包
# 如果下载速度过慢可以使用我们提供的百度网盘链接: https://pan.baidu.com/s/1z-iWIPjxjxbdFS2Qf-NKxQ
# 提取码 8woe
wget https://download.qemu.org/qemu-7.0.0.tar.xz
# 解压
tar xvjf qemu-7.0.0.tar.xz
# 编译安装并配置 RISC-V 支持
cd qemu-7.0.0
./configure --target-list=riscv64-softmmu,riscv64-linux-user
make -j$(nproc)
```

图 2-1 编译安装 Qemu 模拟器

```
qemu-system-riscv64 --version
qemu-riscv64 -version
```

在完成所有的实验环境的配置之后，可以通过试着运行 rCore-Tutorial 操作系统内核来检查一下环境配置是否正确且完全。首先从 github 上克隆一个 OS 实验的仓库到本地，其次通过 Visual Studio Code 打开这个仓库，本文至此运行不需要处理用户代码的裸机操作系统 os1:

运行结果如图 2-2 所示, 说明环境配置正确。

```
[rustsbi] RustSBI version 0.2.2, adapting to RISC-V SBI v1.0.0  
.  
| _ _ \ | | | | | / | _ _ _ _ _ | _ _ _ _ _ | | _ _ \ | | | | |
| |_) | | | | | | (----) | (----) | (----) | |_) || |  
| | / | | | | | \ \ | | | | \ \ | | < | |  
| | \ ----. | ---- ) | | | .----) | |_) || |  
|_| \_ _ _ | \_ _ _ / | _ _ _ / | | | _ _ _ / | |_  
[rustsbi] Implementation : RustSBI-QEMU Version 0.1.1  
[rustsbi] Platform Name : riscv-virtio,qemu  
[rustsbi] Platform SMP : 1  
[rustsbi] Platform Memory : 0x80000000..0x88000000  
[rustsbi] Boot HART : 0  
[rustsbi] Device Tree Region : 0x87000000..0x87000ef2  
[rustsbi] Firmware Address : 0x80000000  
[rustsbi] Supervisor Address : 0x80200000  
[rustsbi] pmp01: 0x00000000..0x80000000 (-wr)  
[rustsbi] pmp02: 0x80000000..0x80200000 (---)  
[rustsbi] pmp03: 0x80200000..0x88000000 (xwr)  
Hello, world!  
[DEBUG] .rodata [0x80203000, 0x80205000)  
[ INFO] .data [0x80205000, 0x80206000)  
[ WARN] boot_stack [0x80206000, 0x80216000)  
[ERROR] .bss [0x80216000, 0x80217000)  
Panicked at src/main.rs:48 Shutdown machine!
```

第3章 rCore 模块化的分析

3.1 rCore 整体分析

本小节对 rCore 模块间的相互关系、各实验涉及的模块做了详细的描述。

在划分好的各模块之中，console 模块、linker 模块、kernel-context 模块、kernel-alloc 模块、kernel-vm 模块、task-manage 模块、easy-fs 模块的独立性比较高，除了被各个章节的操作系统依赖之外没有与其他的模块产生依赖关系。除此之外，syscall 模块有对 signal-defs 模块的依赖，signal 模块有对 kernel-context 模块和 signal-defs 模块的依赖，signal-imple 模块有对 kernel-context 模块和 signal 模块的依赖，sync 模块对 rcore-task-manage 模块有依赖。

rCore 操作系统各章节与独立模块的依赖关系如表 3-1 所示：

表 3-1 各章实验涉及的模块

操作系统	外部引用	实现的模块	模块作用
裸机输出	sbi-rt	console	实现标准输出
批处理系统	riscv	linker	提供链接脚本
		kernel-context	实现特权级机制
		syscall	提供系统调用
多道程序与分时多任务		kernel-context(foreign)	实现特权级机制
地址空间	xmas-elf	kernel-alloc	提供内存管理
		kernel-vm	实现虚拟内存
进程及进程管理	spin	task-manage(proc)	实现进程
简易文件系统	virtio-drivers	easy-fs	实现文件系统
进程间通信		signal-defs	定义信号标号
		signal	定义信号模块
		signal-impl	实现信号模块
同步互斥		sync	实现同步互斥
		task-manage(thread)	实现线程

3.2 console 模块的分析

console 模块的功能是：在移除标准库依赖的情况下实现 `print` 宏和 `println` 宏，可以向控制台输出自己想要输出的内容。实现了日志功能，需要注意目前日志只能提供基础的彩色功能。

为了实现 console 模块想要的功能，需要在这个 console 模块的 `Cargo.toml` 里面添加需要的依赖项：`log = "0.4.17"`、`spin = "0.9.4"`，对 `spin` 的依赖是为了使用其提供的 `Once` 结构，其功能是创建一个只生成一次元素迭代器。对 `log` 的依赖是为了实现 console 模块的日志功能，通过调用 `log::set_max_level()` 设置全局最大的日志级别来实现 `set_log_level()` 的功能，还提供了 `log::log trait` 接口，为日志提供分级功能和 `log::set_logger()` 接口，用来设置全局记录器，值得注意的是在调用 `set_logger` 完成之前发生的任何日志事件都将被忽略。console 模块的对外接口及作用如表 3-2 所示：

表 3-2 console 模块的对外接口及作用

对外接口	作用
<code>print</code> 宏	格式化打印。
<code>println</code> 宏	格式化打印并换行。
<code>Console trait</code>	定义向控制台“输出”这件事。
<code>init_console(console: &'static dyn Console)</code>	设置输出的方法, 即初始化 console。
<code>set_log_level(env: Option<&str>)</code>	根据环境变量设置日志级别
<code>test_log()</code>	打印一些测试信息

3.3 linker 模块的分析

linker 模块的功能是为内核提供链接脚本的文本，同时依赖于定制链接脚本把应用程序的二进制镜像文件作为数据段链接到内核里以使内核运行在 qemu 虚拟机上。在每一章的操作系统模块中的 `build.rs` 文件可依赖此模块，并将作为 [SCRIPT] 文本常量的链接脚本写入链接脚本文件中。有了 linker 模块后内核链接脚本的结构将完全由这个模块控制。所有链接脚本上定义的符号都不会泄露出这个

板块，内核二进制模块可以基于标准纯 Rust 语法来使用模块，而不用再手写链接脚本或记住莫名其妙的 `extern "C"`。

linker 模块的对外接口及作用如表 3-3 所示：

表 3-3 linker 模块的对外接口及作用

对外接口	作用
boot0 宏	设置一个启动栈，并在启动栈上调用高级语言入口。
KernelLayout	定位、保存和访问内核内存布局
KernelRegion	内核内存分区
KernelRegionIterator	内核内存分区迭代器
AppMeta	应用程序元数据

KernelLayout 结构体的接口及作用如表 3-4 所示：

表 3-4 KernelLayout 的接口方法及作用

方法	作用
locate()	定位内核布局
start(&self)	得到内核起始地址
end(&self)	得到内核结尾地址
len(&self)	得到内核静态二进制长度
zero_bss(&self)	清零 .bss 段
iter(&self)	得到内核区段迭代器

内核所在的内核区域被定义为了 4 个部分，分别为代码块，只读数据段，数据段和启动数据段，启动数据段放在最后，以便启动完成后换栈。届时可放弃启动数据段，将其加入动态内存区。KernelRegion 结构有两个成员，分别为 KernelRegionTitle 枚举类型的分区名称 title 和分区地址范围 range。KernelRegionIterator 结构有两个成员，一个成员是 KernelLayout 类型的引用 layout，一个成员是 Option<KernelRegionTitle>类型的 next。

此外 linker 模块的子模块 app 有两个结构，分别为应用程序元数据 AppMeta 和

应用程序迭代器 AppIterator。 App 子模块的接口如表 3-5 所示：

表 3-5 app 子模块的接口及作用

AppMeta	locate()	定位应用程序
	iter(&'static self)	遍历链接进来的应用程序
AppIterator	next(&mut self)	对应用程序进行迭代

3.4 kernel-context 模块的分析

应用程序出错是在所难免的，然而如果应用程序出错会导致操作系统出错那就太令人崩溃了，因此为了保护操作系统不受应用程序的出错的破坏，开发者引入了特权级机制，来实现用户态和内核态的隔离。kernel-context 模块的内核上下文控制就和特权级切换机制密切相关。

kernel-context 模块添加了 spin 的依赖，是为了使用 spin::Lazy，这个接口的作用是对惰性值和静态数据的一次性初始化。

kernel-context 模块主要的结构包括：LocalContext：线程上下文、PortalCache：传送门缓存、ForeignContext：异界线程上下文即不在当前地址空间的线程上下文、TpReg：从 tp 寄存器读取一个序号、MultislotPortal：包含多个插槽的异界传送门。

LocalContext 结构体的成员有 5 个，需要特别注意的是 supervisor 和 interrupt，这两个成员的含义分别是是否以特权态切换和线程中断是否开启。LocalContext 的接口方法如表 3-6 所示：

表 3-6 LocalContext 的接口方法及作用

empty()	创建空白上下文
user(pc: usize)	初始化指定入口的用户上下文，切换到用户态时会打开内核中断
thread(pc:usize, interrupt:bool)	初始化指定入口的内核上下文
move_next(&mut self)	将 pc 移至下一条指令
execute(&mut self)	执行此线程，并返回 sstatus,将修改 sscratch、sepc、sstatus

	和 stvec
x(&self, n: usize)	读取用户通用寄存器
a(&self, n: usize)	读取用户参数寄存器
ra(&self)	读取用户栈指针
sp(&self)	读取用户栈指针
pc(&self)	读取当前上下文的 pc
x_mut(&mut self, n: usize)	修改用户通用寄存器
a_mut(&mut self, n: usize)	修改用户参数寄存器
sp_mut(&mut self)	修改用户栈指针
pc_mut(&mut self)	修改上下文的 pc

此外 kernel-context 的子模块 foreign 有 PortalCache 结构体，该结构的定义是传送门缓存，即映射到公共地址空间，在传送门一次往返期间暂存信息。该结构的方法一共有 2 种，分别是 init(&mut self, satp: usize, pc: usize, a0: usize, supervisor: bool, interrupt: bool) 和 address(&mut self)。这两个方法的作用是初始化传送门缓存和返回缓存地址。ForeignContext 结构体有两个成员，分别是 LocalContext 类型的 context：目标地址空间上的线程上下文和 usize 类型的 satp：目标地址空间。该结构的作用是异界线程上下文，即不在当前地址空间的线程上下文。该结构只有一种方法，是 execute()，该方法的作用是执行异界线程。TpReg 结构体只有一种方法，是 index(self)，这个方法的作用是从 tp 寄存器读取一个序号并转化为插槽序号。

MultislotPortal 结构体的含义是包含多个插槽的异界传送门。该结构有 2 个方法，分别是 calculate_size(slots: usize) 和 init_transit(transit: usize, slots: usize)。这两个方法的作用是计算包括 slots 个插槽的传送门总长度和初始化公共空间上的传送门，其中参数 transit 必须是一个正确映射到公共地址空间上的地址。

3.5 kernel-alloc 模块的分析

kernel-alloc 模块的功能是为内核提供内存管理，即管理内核内存的分配与回收，内核不必区分虚存分配和物理页分配的条件是虚地址空间覆盖物理地址空间，

换句话说，内核能直接访问到所有物理内存而无需执行修改页表之类其他操作。

`kernel-alloc` 模块需要调用 `alloc::alloc::handle_alloc_error` 函数，`core::alloc::Layout` 结构体和 `core::alloc::GlobalAlloc` 特征，`core::ptr::NonNull` 结构体此外需要在 `Cargo.toml` 中对 `customizable_buddy = 0.03` 进行依赖，与此同时调用 `customizable_buddy` 中的 `BuddyAllocator`，`LinkedListBuddy`，`UsizedBuddy` 结构体。

`handle_alloc_error` 函数，在全局分配器分配内存时如果响应分配错误而希望终止计算则调用该函数，该函数的默认行为是将一条消息打印到标准错误并终止该进程。通过 `BuddyAllocator` 的 `new()` 方法可以创建一个全局内存分配器 `HEAP`；`NonNull` 结构体就是非零且协变 `*mut T`，调用 `NonNull::new()` 方法如果 `ptr` 不为空，则创建一个新的 `NonNull`；创建一个结构体 `Global`，为这个结构体实现 `GlobalAlloc` 特征需要的 `alloc()` 方法和 `dealloc()` 方法，体重参数 `layout` 的类型直接使用 `Layout`，通过使用 `#[global_allocator]` 属性将 `Global` 结构体分配为标准库的默认内存分配器。

`kernel-alloc` 模块有两个对外接口：`init(base_address: usize)` 和 `transfer(region: &'static mut [u8])`。

`init(base_address: usize)` 用于初始化内存分配。用户需要告知内存分配器参数 `base_address`，`base_address` 表示动态内存区域的起始位置，用于计算伙伴分配器的参数基址。

`transfer(region: &'static mut [u8])` 用于将一个内存块托管到内存分配器。`region` 内存块的所有权将转移到分配器，因此需要调用者确保这个内存块与已经转移到分配器的内存块都不重叠，且未被其他对象引用。这个内存块必须位于初始化时传入的起始位置之后。值得注意的是这个函数是不安全的。

3.6 kernel-vm 模块的分析

`kernel-vm` 模块的功能是管理内核的虚拟内存，实现了地址空间（`Address Space`）抽象，并在内核中建立虚实地址空间的映射机制，给应用程序提供一个基于地址空间的安全虚拟内存环境，让应用程序简单灵活地使用内存。

`kernel-vm` 模块的实现首先需要在 `Cargo.toml` 中添加依赖项 `spin = "0.9.4"` 和 `page-table = "0.0.6"`，调用 `page-table` 中的结构体

kernel-vm 模块的对外接口是 PageManager 特征和 AddressSpace 结构体。满足 PageManager 特征的结构体的功能是管理物理页，而满足这个特征需要实现的方法及作用如表 3-7 所示：

表 3-7 PageManager 的接口方法及作用

new_root() -> Self	新建根页表
root_ptr(&self) -> NonNull<Pte<Meta>>	获取根页表
p_to_v<T>(&self, ppn: PPN<Meta>) -> NonNull<T>	计算当前地址空间上指向物理页的指针
v_to_p<T>(&self, ptr :Nonnull<T>)->PPN<Meta>	计算当前地址空间上的指针指向的物理页
check_owned(&self,pte:Pte<Meta>)->bool	检查是否拥有一个页的所有权
allocate(&mut self,len:usize,flags:&mut VmFlags<Meta>)->Nonnull<u8 >	为地址空间分配 len 个物理页
deallocate(&mut self, pte: Pte<Meta>, len: usize) -> usize	从地址空间释放 pte 指示的 len 个物理页
drop_root(&mut self)	释放根页表

此外这个特征还提供了一个方法 root_ppn(&self) -> PPN<Meta>来获取根页表的物理页号。

AddressSpace 结构体是地址空间的抽象，该结构体有一个公有成员 areas，areas 成员的类型是 Vec<Range<VPN<Meta>>>，含义是虚拟地址块。AddressSpace 结构体的方法如表 3-8 所示：

表 3-8 AddressSpace 的接口方法及作用

new() -> Self	创建新地址空间
root_ppn(&self) -> PPN<Meta>	得到地址空间根页表的物理页号
root(&self) -> PageTable<Meta>	得到地址空间根页表
map_extern(&mut self,range: Range<VPN<Meta>>, pbase: PPN<Meta>,flags: VmFlags<Meta>)	向地址空间增加映射关系
map(&mut self,range:Range<VPN<Meta>>,	分配新的物理页，拷贝数据并建立

<code>data:&[u8],offset:usize,flags:VmFlags<Meta>)</code>	映射
<code>translate<T>(&self,addr:VAddr<Meta>,flags: VmFlags<Meta>) -> Option<NonNull<T>></code>	检查 flags 的属性要求，将地址空间中的一个虚地址翻译成当前地址空间中的指针
<code>cloneself(&self, new_addrspace: &mut AddressSpace<Meta, M>)</code>	遍历地址空间，将其中的地址映射添加进自己的地址空间中，重新分配物理页并拷贝所有数据及代码

3.7 syscall 模块的分析

这个库封装了提供给操作系统和用户程序的系统调用。

syscall 模块需要在 Cargo.toml 中添加依赖：spin = "0.9.4"、bitflags = "1.2.1"、signal-defs，而作为依赖的 signal-defs 模块需要在 Cargo.toml 中添加依赖 numeric-enum-macro = "0.2.0"。此外需要在 Cargo.toml 中设置两个 features：user 和 kernel。只有当 feature = user 时，才调用 user 子模块的内容，当 feature = kernel 时调用 kernel 子模块的内容。

syscall 模块定义的结构体有 ClockId、SignalAction、SyscallId、TimeSpec，其中 SignalAction 结构体是信号处理函数的定义，SyscallId 结构体是系统调用号，实现了 From 特征，这个实现为包装类型，是为了在不损失扩展性的情况下实现类型安全性。系统调用号从 Musl Libc for RISC-V 源码生成，因为找不到标准文档。TimeSpec 结构体有两个成员：tv_sec 和 tv_nsec，都是 usize 类型，分别代别了秒和纳秒，该结构体定义了几个分别代表 0、1 秒、1 毫秒、1 微秒、1 纳秒的 ZERO、SECOND、MILLSECOND、MICROSECOND、NANOSECOND 常数及 from_millisecond() 函数，该函数能将输入的毫秒转化成该结构体下的秒和纳秒。

syscall 模块对外暴露的枚举有 SignalNo，代表了信号的编号，枚举的内容是从 0 到 63，从 32 开始的部分为实时信号：SIGRT，其中 RT 表示 real time，但目前实现时没有通过 ipi 等手段即时处理，而是像其他信号一样等到 trap 再处理。同时还暴露了几个常量：MAX_SIG、STDDEBUG、STDIN、STDOUT，分别代表最大的信号编号、标准调试、标准输入和标准输出。

当 features = user 时，对外接口增加了 OpenFlags 结构体和 close()、

exit()、fork()等对外接口函数。

当 features = kernel 时，对外接口增加了 Caller 结构体，用来管理系统调用的发起者信息，Caller 结构体有两个成员：entity:usize 和 flow: usize，分别代表发起者拥有的资源集的标记，相当于进程号；发起者的控制流的标记，相当于线程号。还增加了 SyscallResult 枚举和 Clock、IO、Memory、Process、Scheduling、Signal、SyncMutex、Thread 等 trait 及 handle()、init_thread() 等接口函数。

3.8 task-manage 模块的分析

rCore 操作系统第五章将开发一个命令行(俗称 Shell)，形成用户与操作系统进行交互的命令行界面，为此，开发者将对任务建立新的抽象：进程，并实现若干基于进程的强大系统调用。rCore 前几章的任务抽象是这里提到的进程抽象的初级阶段，与任务相比，进程能在运行中创建子进程、用新的程序内容覆盖已有的程序内容、可管理更多物理或虚拟资源。

线程是进程的组成部分，进程可包含 1 至 n 个线程，属于同一个进程的线程共享进程的资源，如地址空间、打开的文件等。线程是可以被操作系统或用户态调度器独立调度和分派的基本单位。在有了线程后，进程是线程的资源容器，线程成为了程序的基本执行实体。而 task-manage 模块的功能就是对进程和线程进行管理。

task-manage 模块没有依赖的对象，与之不同的是使用了 proc 和 thread 两个 features，当没有使用 features 参数进行编译时，只会定义 ProcId、ThreadId 两个结构，和 Manage、Schedule 两个特征。进程号 ProcId 结构体的方法及作用如表 3-9 所示：

表 3-9 ProcId 的接口方法及作用

new() -> Self	创建了一个进程编号自增的进程 id 类型
from_usize(v: usize) -> Self	根据输入的 usize 类型参数可以获得一个以参数为 id 的 ProcId
get_usize(&self) -> usize	需要输入的参数是一个 ProcId 的引用，返回该 ProcId 结构对应的 id

ThreadId 结构体与 ProcId 结构体的含义和方法相似，只是把进程换成了线程。

Manage 特征对标数据库增删改查操作，Schedule 特征通过在队列中保存需要调度的任务 Id 来调度任务，这两个特征定义的方法如表 3-10 所示：

表 3-10 Manage 及 Schedule 的接口方法及作用

Manage	insert(&mut self, id: I, item: T)	插入 item
	delete(&mut self, id: I)	删除 item
	get_mut(&mut self, id: I) -> Option<&mut T>	获取可变的 item
Schedule	add(&mut self, id: I)	向调度队列加入一个任务
	fetch(&mut self) -> Option<I>	从调度队列中取出一个任务

当 features = proc 时，增加了 ProcRel 和 PManager 两个结构体。ProcRel 结构体封装了进程与其子进程之间的关系，通过进程的 Id 来查询这个关系；PManager 结构体用来管理进程以及进程之间的父子关系。

ProcRel 的公有成员有 ProcId 类型的 parent、Vec<ProcId>类型的 children、及 Vec<(ProcId, isize)>类型的 dead_children，这几个成员分别代表父进程 id，子进程列表和已经结束的进程列表。ProcRel 结构体的方法有下面五种。

```

new(parent_pid: ProcId) -> Self
add_child(&mut self, child_pid: ProcId)
del_child(&mut self, child_pid: ProcId, exit_code: isize)
wait_any_child(&mut self) -> Option<(ProcId, isize)>
wait_child(&mut self, child_pid: ProcId) -> Option<(ProcId, isize)>

```

- (1) new() 方法在创建一个新的进程的时候使用，用来创建一个新的进程关系。
- (2) add_child() 方法的作用是在进程关系中增加一个子进程。
- (3) del_child() 方法的作用是令子进程结束，子进程 Id 被移入到 dead_children 队列中，等待 wait 系统调用来处理。
- (4) wait_any_child() 方法的作用是等待任意一个结束的子进程，直接弹出 dead_children 队首，如果等待进程队列和子进程队列为空，返回 None，如果等待进程队列为空、子进程队列不为空，则返回 -2。
- (5) wait_child 方法的作用是等待特定的一个结束的子进程，弹出 dead_children

中对应的子进程，如果等待进程队列和子进程队列为空，返回 None，如果等待进程队列为空、子进程队列不为空，则返回 -2。

与此同时，PManager 结构体的方法及作用如表 3-11 所示：

表 3-11 PManager 的接口方法及作用

new() -> Self	新建 PManager
find_next(&mut self) -> Option<&mut P>	找到下一个进程
set_manager(&mut self, manager: MP)	设置 manager
make_current_suspend(&mut self)	阻塞当前进程
make_current_exited(&mut self, exit_code: isize)	结束当前进程，只会删除进程的内容，以及与当前进程相关的关系
add(&mut self, id: ProcId, task: P, parent: ProcId)	添加进程，需要指明创建的进程的父进程 Id
current(&mut self) -> Option<&mut P>	获取当前进程
get_task(&mut self, id: ProcId) -> Option<&mut P>	获取某个进程
wait(&mut self, child_pid: ProcId) -> Option<(ProcId, isize)>	wait 系统调用，返回结束的子进程 id 和 exit_code，正在运行的子进程不返回 None，返回 (-2, -1)。

当 features = thread 时，只是在进程中加入了代表进程、子进程以及进程地址空间内的线程之间的关系的结构体 ProcThreadRel，用来管理进程、子进程以及进程地址空间内的线程之间关系的结构体 PThreadManager，这两个方法与对应的进程的结构的方法基本相同，这里就不详细展开了。

3.9 easy-fs 模块的分析

easy-f 模块将实现一个简单的与内核隔离的文件系统 easy-fs，能够对持久存储设备 I/O 资源进行管理。将设计两种文件：常规文件和目录文件，这两种文件均以文件系统所维护的磁盘文件形式被组织并保存在持久存储设备上。

easy-fs 模块的对外的接口有 EasyFileSystem、FileHandle、Inode、

OpenFlags、UserBuffer 五个结构体及 BlockDevice、FSManager 两个特征。结构体的含义分别是块上的简单文件系统、内存中的缓存文件元数据、easy-fs 上的虚拟文件系统层、打开文件标志、用户与 os 通信的 u8 切片数组。BlockDevice 是以块为单位读写数据的块设备的特性，FSManager 是对文件进行管理的特性。

BlockDevice 特征需要两个方法：

```
read_block(&self, block_id: usize, buf: &mut [u8])
write_block(&self, block_id: usize, buf: &[u8])
```

这两个方法的功能分别是将编号为 block_id 的块从磁盘读入内存中的缓冲区 buf 和将数据从缓冲区写入块。

FSManager 特征定义的五种方法如表 3-12 所示：

表 3-12 FSManager 的接口方法及作用

open(&self, path: &str, flags: OpenFlags) -> Option<Arc<FileHandle>>	打开文件
find(&self, path: &str) -> Option<Arc<Inode>>	查找文件
link(&self, src: &str, dst: &str) -> isize	创建到源文件的硬链接
unlink(&self, path: &str) -> isize	删除硬链接
readdir(&self, path: &str) -> Option<Vec<String>>	列出目标目录下的 inode

EasyFileSystem 结构体的公有成员有：block_device: Arc<dyn BlockDevice> inode_bitmap: Bitmap data_bitmap: Bitmap，这三个成员的含义分别为真实设备、索引节点位图、数据位图。EasyFileSystem 结构体的对外接口函数如表 3-13 所示：

表 3-13 EasyFileSystem 的接口方法及作用

create(block_device: Arc<dyn BlockDevice>)	从块设备创建文件系统
total_blocks: u32, inode_bitmap_blocks: u32) -> Arc<Mutex<Self>>	将块设备作为文件系统打开
open(block_device: Arc<dyn BlockDevice>) -> Arc<Mutex<Self>>	获取文件系统的根索引节点
root_inode(efs: &Arc<Mutex<Self>>) -> Inode	通过 id 获取索引节点、
get_disk_inode_pos(&self, inode_id: u32) -> (u32, usize)	通过 id 获取数据块
get_data_block_id(&self, data_block_id: u32) -> u32	分配新索引节点

北京理工大学本科生毕业设计（论文）

<code>alloc_inode(&mut self) -> u32</code>	分配一个数据块
<code>alloc_data(&mut self) -> u32</code>	释放一个数据块

Inode 结构体代表 easy-fs 上的虚拟文件系统层，该结构体的对外接口函数如表 3-14 所示：

表 3-14 Inode 的接口方法及作用

<code>new(block_id: u32, block_offset: usize, fs: Arc<Mutex<EasyFileSystem>>)</code>	创建一个 vfs 索引节点
<code>block_device: Arc<dyn BlockDevice>) -> Self</code>	按名称查找当前索引节点下的索引节点
<code>find(&self, name: &str) -> Option<Arc<Inode>></code>	通过名称创建当前索引节点下的索引节点
<code>create(&self, name: &str) -> Option<Arc<Inode>></code>	列出当前索引节点下的索引节点
<code>readdir(&self) -> Vec<String></code>	从当前索引节点读取数据
<code>read_at(&self, offset: usize, buf: &mut [u8]) -> usize</code>	对当前索引节点写数据
<code>write_at(&self, offset: usize, buf: &[u8]) -> usize</code>	清除当前索引节点中的数据

FileHandle 结构体表示内存中的缓存文件的元数据，该结构体的公有成员有 `Inode: Option<Arc<Inode>>`、`read: bool`、`write: bool` 及 `offset: usize`，这些成员的含义是文件系统索引节点、打开选项：能读、打开选项：能写及当前偏移量。该结构体的对外接口函数如表 3-15 所示：

表 3-15 FileHandle 的接口方法及作用

<code>new(read: bool, write: bool, inode: Arc<Inode>) -> Self</code>	创建一个新的 FileHandle
<code>empty(read: bool, write: bool) -> Self</code>	创建一个空的 FileHandle
<code>readable(&self) -> bool</code>	判断 FileHandle 是否能读
<code>writable(&self) -> bool</code>	判断 FileHandle 是否能写
<code>read(&mut self, buf: UserBuffer) -> isize</code>	从缓冲区读数据
<code>write(&mut self, buf: UserBuffer) -> isize</code>	向缓冲区写数据

OpenFlags 结构体是打开文件的标志，定义了一些分别表示只读、只写、读写、允许创建、清除文件并返回一个空文件的常数 `RONLY`、`WRONLY`、`RDWR`、

CREATE、TRUNC，及接口函数：

```
read_write(&self) -> (bool, bool)
```

这个函数的功能是返回（可读、可写），用来表示文件是否可读、可写。

UserBuffer 结构体代表用户与 os 通信的缓冲区，该缓冲区的公有成员有一个：buffers: Vec<&'static mut [u8]>，类型为一个 u8 切片数组。该结构体的对外接口函数如表 3-16 所示：

表 3-16 UserBuffer 的接口方法及作用

new(buffers: Vec<&'static mut [u8]>) -> Self、	通过参数创建一个用户缓冲区
len(&self) -> usize,	得到用户缓冲区的长度

3.10 signal-defs、signal 和 signal-imple 模块的分析

signal-defs 模块需要在 Cargo.toml 中添加依赖 numeric-enum-macro = "0.2.0"。通过 #[derive] 属性，编译器能够提供某些 trait 的基本实现，对于 SignalNo 通过 #[derive] 属性实现了 Eq, PartialEq, Debug, Copy, Clone 特征，

signal-defs 模块对外暴露的枚举有 SignalNo，代表了信号的编号，枚举的内容是从 0 到 63, 从 32 开始的部分为实时信号：SIGRT，其中 RT 表示 real time，然而目前实现时没有通过 ipi 等手段即时处理，而是像其他信号一样等到 trap 再处理。SignalNo 实现了 From<usize> trait，从输入类型转换为此结构体类型。同时还暴露了一个常量：MAX_SIG 代表最大的信号编号。最后，还定义了 SignalAction 结构体，是信号处理函数的定义；SignalAction 通过 #[derive] 属性实现了 Debug, Clone, Copy, Default 特征；SignalAction 有两个公有成员 handler、mask。

signal 模块是信号的管理和处理模块，信号模块的实际实现在 signal_impl 模块，需要依赖 kernel-context 模块和 signal-defs 模块，signal 模块在依赖 signal-defs 模块的基础上，增加了代表信号处理函数返回得到的结果的 SignalResult 枚举，SignalResult 的变体及作用如表 3-17 所示：

表 3-17 SignalResult 的变体

变体	作用
NoSignal	没有信号需要处理
IsHandlingSignal	目前正在处理信号，因此无法接受其他信号
Ignored	已经处理了一个信号，正常返回用户态即可
Handled	已经处理了一个信号，并修改了用户上下文
ProcessKilled(i32)	需要结束当前进程，并给出退出时向父进程返回的 <code>errno</code>
ProcessSuspended	需要暂停当前进程，直到其他进程给出继续执行的信号

signal 模块还增加了 Signal 特征：一个信号模块需要实现的功能。想要实现这个 trait 需要满足一些函数：

```

from_fork(&mut self) -> Box<dyn Signal>
clear(&mut self)
add_signal(&mut self, signal: SignalNo)
is_handling_signal(&self) -> bool
set_action(&mut self, signum: SignalNo, action: &SignalAction) -> bool
get_action_ref(&self, signum: SignalNo) -> Option<SignalAction>
update_mask(&mut self, mask: usize) -> usize
handle_signals(&mut self, current_context: &mut LocalContext)->
SignalResult
sig_return(&mut self, current_context: &mut LocalContext) -> bool

```

- (1) from_fork 函数的功能是：当 fork 一个任务时(在通常的 linux syscall 中，fork 是某种参数形式的 sys_clone)，需要继承原任务的信号处理函数和掩码。此时 task 模块会调用此函数，根据原任务的信号模块生成新任务的信号模块。
- (2) clear 函数的功能是实现 sys_exec，然而 sys_exec 不会继承信号处理函数和掩码。
- (3) add_signal 函数的功能是：添加一个信号
- (4) is_handling_signal 函数的功能是：是否当前正在处理信号
- (5) set_action 函数的功能是：设置一个信号处理函数，返回设置是否成功。
sys_sigaction 会使用。（不成功说明设置是无效的，需要在 sig_action 中返回 EINVAL）

- (6) `get_action_ref` 函数的功能是：获取一个信号处理函数的值，返回设置是否成功。`sys_sigaction` 会使用。（不成功说明设置是无效的，需要在 `sig_action` 中返回 `EINVAL`）
- (7) `update_mask` 函数的功能是：设置信号掩码，并获取旧的信号掩码，`sys_procmask` 会使用。
- (8) `handle_signals` 函数的功能是：进程执行结果，可能是直接返回用户程序或存栈或暂停或退出。
- (9) `sig_return` 函数的功能是：从信号处理函数中退出，返回值表示是否成功，`sys_sigreturn` 会使用。

`signal-imple` 模块需要对 `kernel-context` 模块和 `signal` 模块进行依赖。`signal-imple` 模块是一种对信号模块的实现，对外暴露的接口有 `SignalImpl` 结构体和 `HandlingSignal` 枚举，`SignalImpl` 结构体实现 `signal` 模块的 `Signal` 特征。`SignalImpl` 结构体的公有成员如表 3-18 所示：

表 3-18 `SignalImpl` 及 `HandlingSignal` 的成员

SignalImpl	received: SignalSet	已收到的信号
	mask: SignalSet	屏蔽的信号掩码
	handling: Option<HandlingSignal>	在信号处理函数中，保存之前的用户栈
	actions: [Option<SignalAction>; 32]	当前任务的信号处理函数集
HandlingSignal	Frozen	内核信息，需要暂停当前进程
	UserSignal(LocalContext)	用户信息，需要保存当前的用户栈

3.11 sync 模块的分析

当多个线程共享同一进程的地址空间时，每个线程都可以访问属于这个进程的数据（全局变量）。如果每个线程使用到的变量都是其他线程不会读取或者修改的话，那么就不存在一致性问题。如果变量是只读的，多个线程读取该变量亦不会有一致性问题。与之相反的是，当一个线程修改变量时，其他线程在读取这个变量时，可能会看到一个不一致的值，这就是数据不一致性的问题。`sync` 模块是同步互斥模块，就是用来解决这中问题的。

sync 模块需要在 Cargo.toml 中添加依赖 `riscv = "0.8.0"`、`spin = "0.9.4"` 和 `thread` 特征下的 `task-manage` 模块。

对外暴露了：`Condvar`、`MutexBlocking`、`Semaphore`、`UPIntrFreeCell`、`UPIntrRefMut`、`UPSafeCellRaw` 及 `IntrMaskingInfo` 结构体，分别代表条件变量、互斥阻塞、信号、具有动态检查借用规则的可变内存位置，允许内核开发者在单核上安全使用可变全局变量、从 `RefCell` 可变借用值的包装器类型、内部可变性、内部屏蔽信息。以及代表互斥的 `Mutex` 特征。

`Condvar` 结构体的接口函数如表 3-19 所示：

表 3-19 `Condvar` 的接口函数

<code>new() -> Self</code>	创建一个新的条件变量结构
<code>signal(&self) -> Option<ThreadId></code>	唤醒某个阻塞在当前条件变量上的线程
<code>wait_no_sched(&self, tid: ThreadId) -> bool</code>	将当前线程阻塞在条件变量上
<code>wait_with_mutex(&self, tid: ThreadId, mutex: Arc<dyn Mutex>) -> (bool, Option<ThreadId>)</code>	从 <code>mutex</code> 的锁中释放一个线程，并将其阻塞在条件变量的等待队列中，等待其他线程运行完毕，当前的线程再试图获取这个锁

`MutexBlocking` 结构实现了互斥。该结构的接口函数如表 3-20 所示：

表 3-20 `MutexBlocking` 的接口函数

<code>new() -> Self</code>	创建一个新的 <code>MutexBlocking</code> 结构
<code>lock(&self, tid: ThreadId) -> bool</code>	获取锁，如果获取成功，返回 <code>true</code> ，否则会返回 <code>false</code> ，要求阻塞对应的线程
<code>unlock(&self) -> Option<ThreadId></code>	释放锁，释放之后会唤醒一个被阻塞的进程，要求重新进入调度队列

`Semaphore` 结构体的接口函数如表 3-21 所示：

表 3-21 `Semaphore` 的接口函数

<code>new(res_count: usize) -> Self</code>	创建一个新的 <code>Semaphore</code> 结构
---	----------------------------------

<code>up(&self) -> Option<ThreadId></code>	当前线程释放信号量表示的一个资源，并唤醒一个阻塞的线程
<code>down(&self, tid: ThreadId) -> bool</code>	当前线程试图获取信号量表示的资源，并返回结果

UPSafeCellRaw 结构体的接口如表 3-22 所示：

表 3-22 UPSafeCellRaw 的接口函数

<code>new(value: T) -> Self</code>	创建一个新的 UPSafeCellRaw 结构
<code>get_mut(&self) -> &mut T</code>	获得该结构内部成员的可变引用

IntrMaskingInfo 有三个接口函数 `new() -> Self`、`enter(&mut self)`、`exit(&mut self)` 这三个函数为 `UPIntrFreeCell` 的接口函数和 `UPIntrRefMut` 的 trait 需要的函数的实现提供了基础。

UPIntrFreeCell 的接口函数如表 3-23 所示：

表 3-23 UPIntrFreeCell 的接口函数

<code>new(value: T) -> Self</code>	创建新的结构
<code>exclusive_access(&self) -> UPIntrRefMut<'_, T></code>	独占访问，如果数据已被借用，会 panic
<code>exclusive_session<F, V>(&self, f: F) -> V</code>	独占会话

UPIntrRefMut 实现了三个 trait: `Deref`、`DerefMut`、`Drop`，这三个 trait 需要的函数分别如表 3-24 所示：

表 3-24 UPIntrRefMut 定义的函数

<code>deref(&self) -> &Self::Target</code>	取消引用该值
<code>deref_mut(&mut self) -> &mut Self::Target</code>	可变地取消引用该值
<code>drop(&mut self)</code>	将数据删除

需要注意的是 `Target` 为取消引用后的结果类型。

第 4 章 rCore 模块化的测试

4.1 console 模块的用户态单元测试

在 Rust 中，测试是通过函数的方式实现的，测试函数可以用于验证被测试代码的正确性。测试函数往往依次执行以下三种行为：1. 设置所需的数据或状态 2. 运行想要测试的代码 3. 判断 (assert) 返回的结果是否符合预期。

在进行 console 模块的单元测试时，需要实现 put_char() 函数向控制台输出字符的功能，笔者开始想得是找到了一个输出的函数，看这个函数调用了哪些函数，想着找到了最后就能找到不需要标准库支持的输出字符的方法，很明显这个思路是错误的，之后想利用 sbi 的功能来实现 put_char() 函数，然而此方法增加了新的依赖且没有成功。最后在导师的帮助下，找到了正确的方法，应该直接在核心库 core 里面找到一个能够向控制台输出字符的功能函数，如使用 assert_eq() 函数，如果里面的两个参数不相等或者不能进行比较，在进行测试的时候哪怕没有标准库仍然会有信息在控制台上显示，提示 assert_eq() 的参数不相等，或者这个类型的参数不能进行比较之类的信息。然而笔者在核心库里 assert_eq() 测试失败的地方找源代码并没有发现这样一个函数，最后，笔者使用了标准库中的输出函数来作为桩函数。

在用户态测试 console 之前，首先需要定义一个满足 console 模块定义的 Console trait 的结构体 Console1，即需要利用桩函数实现 put_char() 函数，该函数的功能是向控制台输出一个字符，输出的字符对应的 ASCII 码是输入的参数 c: usize。该函数的实现步骤为：首先将作为参数的 ASCII 码放入数组 buffer 中，其次调用标准库中的 std::str::from_utf8 函数将含有 ASCII 码的数组 buffer 转化成含有一个字符的字符串 s，之后调用标准库中的 print 宏输出该字符串即可实现向控制台输出一个字符的操作。

至此，在用户态测试 console 模块已经完成了设置所需要的数据或者状态的任务，可以依次运行想要测试的代码了。在这里的测试用例设置的是很常见的输出 hello world 以及输出各种日志级别的 Hello world!。在这里就是依次运行 init_console() 函数，put_char() 函数、put_str() 函数、set_log_level() 函数、

test_log()函数以及 print、println 宏；最后比较控制台上是否是自己预期的输出。console 模块单元测试结果如图 4-1 所示：

```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/console$ cargo test --package rcore-console --test test -- --nocapture
   Compiling rcore-console v0.0.0 (/home/swl/os/crate_module/console)
   Finished test [unoptimized + debuginfo] target(s) in 2.44s
   Running tests/test.rs (target/debug/deps/test-b9b2524329fff092)

running 1 test
F
___`abc

=====
[TRACE] LOG TEST >> Hello, world!
[DEBUG] LOG TEST >> Hello, world!
[ INFO] LOG TEST >> Hello, world!
[ WARN] LOG TEST >> Hello, world!
[ERROR] LOG TEST >> Hello, world!
test_hello_world is OK

hello world
test test_println ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/console$
```

图 4-1 console 模块单元测试结果

4.2 linker 模块的用户态单元测试

linker 模块的用户态测试比较简单，只需要依次调用 linker 模块里的 KernelLayout 结构：代表内核地址信息；KernelRegion 结构：内核内存分区；KernelRegionIterator 结构：内核内存分区迭代器和 KernelRegionTitle 枚举：内核内存分区名称。与此同时设置一个非零初始化的 KernelLayout，依次运行结构体的方法，将初始化的结果和预期值进行比较就可以了。linker 模块单元测试结果如图 4-2 所示：

```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/linker$ cargo test --package linker --lib -- tests --nocapture
   Compiling linker v0.0.1 (/home/swl/os/crate_module/linker)
   Finished test [unoptimized + debuginfo] target(s) in 1.30s
   Running unittests src/lib.rs (target/debug/deps/linker-890dba53af58e019)

running 4 tests
test tests::test_app ... ok
test tests::test_kernel_layout ... ok
test tests::test_kernel_region ... ok
test tests::test_kernel_region_iterator ... ok

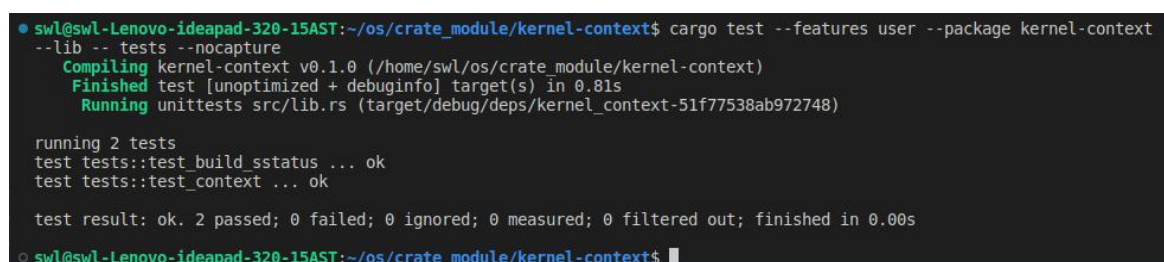
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/linker$
```

图 4-2 linker 模块单元测试结果

4.3 kernel-context 模块的用户态单元测试

kernel-context 模块的用户态测试亦比较简单，就是调用该模块自己的线程上下文结构 LocalContext，与此同时依次运行 LocalContext 结构的方法，通过 assert_eq() 函数判断返回的结果是否符合预期，综上所述，已经完成 kernel-context 模块的用户态单元测试。

具体来说就是首先运行 LocalContext 结构的 empty() 函数来创建空白上下文，其次通过 assert_eq() 函数来比较创建的空白上下文的两个成员 supervisor 及 interrupt 是否为 false。如果这两个成员的值都为 false，比较通过 pc() 方法得到的 sepc 成员是否与预期的 0 相等，如果相等则表明 empty() 函数的运行结果与预期结果相同，empty() 函数的测试就完成了。该结构的其他方法亦是一样，先运行 user(pc: 04) 方法，判断 supervisor 是否等于 false，interrupt 是否等于 true，pc() 函数是否返回 04。运行 thread(04, false) 方法，判断 supervisor 是否等于 true，interrupt 是否等于 false，pc() 函数是否返回 04。运行 thread(04, true) 方法，判断 supervisor 是否等于 true，interrupt 是否等于 true，pc() 函数是否返回 04。运行 x()、a()、ra()、sp() 及 pc() 等读取类函数，判断他们是否分别与 0、0、0、0 及 04 相等，运行 move_next() 函数，判断是否将 pc 移动到下一条指令，即 pc() 函数的返回值是否为 08。最后还需要测试 x_mut(1)、a_mut(1)、sp_mut() 及 pc_mut() 等修改类函数，在测试之前写一个 LocalContext 结构，以这个 LocalContext 结构为参数运行这些修改类方法，判断得到的结果是否和预期结果相同。至此，LocalContext 这个结构以及其中的方法就测试完成了。kernel-context 模块单元测试结果如图 4-3 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-context$ cargo test --features user --package kernel-context
--lib -- tests --nocapture
Compiling kernel-context v0.1.0 (/home/swl/os/crate_module/kernel-context)
Finished test [unoptimized + debuginfo] target(s) in 0.81s
Running unittests src/lib.rs (target/debug/deps/kernel_context-51f77538ab972748)

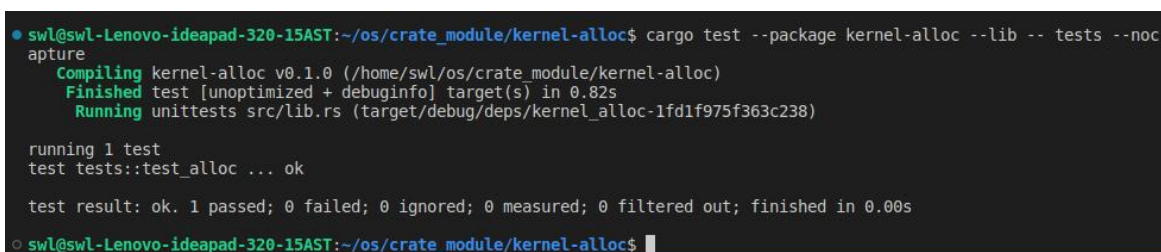
running 2 tests
test tests::test_build_sstatus ... ok
test tests::test_context ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-context$
```

图 4-3 kernel-context 模块单元测试结果

4.4 kernel-alloc 模块的用户态单元测试

单独运行 kernel-alloc 模块的时候会报错，因此在 Global 结构及其方法之上添加了#[cfg(feature = “kernel”)], 其次在测试的时候定义一个内核地址信息结构 KernelLayout, 该结构只需要有代表开始地址的 text 及代表结束地址的 end 两个成员，以及实现 start(), end(), len() 三个方法，分别得到内核开始地址、内核结束地址及内核静态二进制长度。最后调用初始化函数 init(), 将结果和预期结果进行比较。kernel-alloc 模块单元测试结果如图 4-4 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-alloc$ cargo test --package kernel-alloc --lib -- tests --nocapture
   Compiling kernel-alloc v0.1.0 (/home/swl/os/crate_module/kernel-alloc)
   Finished test [unoptimized + debuginfo] target(s) in 0.82s
   Running unittests src/lib.rs (target/debug/deps/kernel_alloc-1fd1f975f363c238)

running 1 test
test tests::test_alloc ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

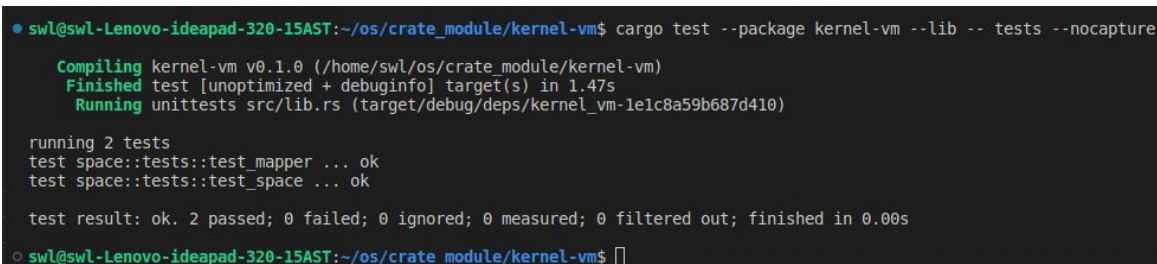
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-alloc$
```

图 4-4 kernel-alloc 模块单元测试结果

4.5 kernel-vm 模块的用户态单元测试

AddressSpace 结构体的参数 Meta 需要满足 VmMeta 特征，M 参数需要满足 PageManager<Meta>特征，由此可知在测试 AddressSpace 结构体之前，需要定义实现了 VmMeta 特征的 SV39 结构和实现了 PageManager<Meta>特征的 Sv39Manager 结构体。VmMeta 特征其实就是 'static + MmuMeta + Copy + Ord + core::hash::Hash + core::fmt::Debug 这几个特征的集合，可以直接使用 #[derive(Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash, Debug)] 来满足其他的特征，与此同时为 SV39 结构体构建 is_leaf(value: usize) -> bool 函数来满足 MmuMeta 特征，则 SV39 就实现了 VmMeta 特征。

其次根据 SV39 结构体定义一个 Sv39Manager(NonNull<Pte<Sv39>>) 结构体且依次实现 PageManager<Sv39>特征需要的函数。最后还需要定义一个 KernelLayout 结构体，并实现 start() 和 end() 方法，赋值一个物理页 range1: Range<PPN>和 pbase1: PageNumber<Sv39, Physical>, 这样测试需要的准备工作就做好了。依次运行 AddressSpace 的几个方法，并和预期值进行比较。kernel-vm 模块单元测试结果如图 4-5 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-vm$ cargo test --package kernel-vm --lib -- tests --nocapture
Compiling kernel-vm v0.1.0 (/home/swl/os/crate_module/kernel-vm)
Finished test [unoptimized + debuginfo] target(s) in 1.47s
Running unittests src/lib.rs (target/debug/deps/kernel_vm-1e1c8a59b687d410)

running 2 tests
test space::tests::test_mapper ... ok
test space::tests::test_space ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/kernel-vm$
```

图 4-5 kernel-vm 模块单元测试结果

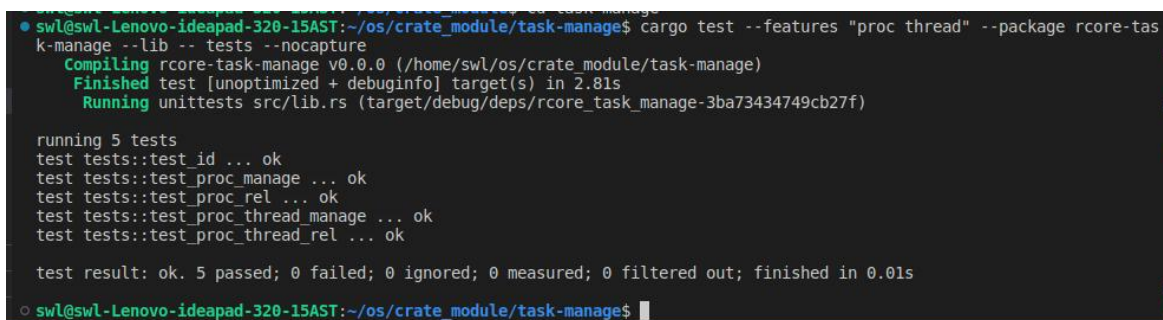
4.6 task-manage 模块的用户态单元测试

在 `feature = proc` 的时候，调用 `proc_manage::PManage` 和 `proc_rel::ProcRel`，在 `feature = thread` 的时候，调用 `thread_manage::PThreadManage` 和 `proc_thread_rel::ProcThreadRel`。了解了这些就可以开始进行 task-manage 模块的测试。

首先，测试进程 id 结构 `ProcId` 的 `new()` 方法，`from_usize(v:usize)` 方法，`get_usize(&self)` 方法，分别运行这三个函数，比较第一次 `new()` 函数的结果是否与 `from_usize(0)` 的结果相同，比较第二次 `new()` 函数的结果是否与 `from_usize(1)` 的结果相同，若两次结果都相同则表明 `new()` 函数的编号自增功能已经实现，此外 `from_usize(v)` 函数亦可以得到对应的进程 id；比较对应的数字与 `get_usize()` 的结果是否相同来测试 `get_usize()` 的功能是否实现。线程 id 结构 `ThreadId` 和其方法的测试与进程 id 的测试一样。

在测试 `proc_rel` 和 `proc_manage` 的时候，需要在测试函数之前加上 `#[cfg(feature = "proc")]` 以便在运行这两个测试函数的时候能够调用 `PManage` 结构和 `ProcRel` 结构，在测试 `proc_rel` 时，首先创建一个父进程 id 和一个子进程 id，与此同时根据父进程 id 创建一个进程关系 `procrel` 且以进程关系 `procrel` 为参数运行等待子进程结束的函数 `wait_any_child()` 和 `wait_child()`，将该函数的返回值和 `None` 进行比较。其次运行添加子进程的函数 `add_child(&mut procrel, child_id)`，比较 `child_id` 和 `procrel` 里 `child` 队列中的 `child_id` 是否相同，比较等待子进程结束的函数返回值是否和预期值 `Some((ProcId::from_usize(-2 as _), -1))` 相同。最后运行子进程结束函数 `dead_child()`，子进程 id 将会被转移到 `dead_children` 队列中，比较进程关系 `procrel` 的子进程队列成员是否和预期的空队列相同，至此 `proc_rel` 测试完成。

在测试 `proc_manage` 时，首先需要创建一个结构 `Process`，该结构只包含一个进程号，`Process` 结构需要能够根据 `proc_id` 创建进程。且创建一个满足 `MP: Manage<P, ProcId>+ Schedule<ProcId>` 特征的 `ProcManage` 结构，该结构包含一个任务列表和一个准备队列，这个 `ProcManage` 结构需要满足 `Manage<Process, ProcId>` 特征，即能够 `insert()` 插入一个任务，`delete` 删除任务实体和 `get_mut()` 根据 `id` 获取对应的任务，同时还需要满足 `Schedule<ProcId>` 特征，即实现添加 `id` 进入调度队列的 `add()` 函数及从调度队列中取出 `id` 的 `fetch()` 函数，至此测试需要的准备工作完成了。与此同时新建一个满足 `MP` 的 `procmanage` 结构和管理父进程和子进程的 `pmanage` 结构，依次运行设置 `manage`、添加子进程和获取指定进程的函数，将 `get_task()` 和 `find_next()` 函数的返回值和预期结果进行比较。线程的测试和进程的测试方法是一样的，这里就不重复说明了。`task-manage` 模块单元测试结果如图 4-6 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/task-manage$ cargo test --features "proc thread" --package rcore-task-manage --lib -- tests --nocapture
Compiling rcore-task-manage v0.0.0 (/home/swl/os/crate_module/task-manage)
Finished test [unoptimized + debuginfo] target(s) in 2.81s
Running unittests src/lib.rs (target/debug/deps/rcore_task_manage-3ba73434749cb27f)

running 5 tests
test tests::test_id ... ok
test tests::test_proc_manage ... ok
test tests::test_proc_rel ... ok
test tests::test_proc_thread_manage ... ok
test tests::test_proc_thread_rel ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/task-manage$
```

图 4-6 `task-manage` 模块单元测试结果

4.7 easy-fs 模块的用户态单元测试

对 `easy-fs` 模块进行用户态单元测试，首先需要调用标准库中的文件操作系统 `fs` 模块中代表对文件系统上打开的文件的引用的 `File` 结构体、可用于配置文件打开方式的选项和标志的 `OpenOptions` 结构。标准库中 `io` 模块的提供用于读取和写入、输入和输出的最通用接口的 `Read` 及 `Write` 特征、建立在 `reader` 的顶部，以控制读取的方式，`Seek` 可以控制下一个字节的来源的 `Seek` 特征及列举可能在 `I/O` 对象中进行搜索的方法的 `SeekFrom` 枚举。标准库中同步互斥 `sync` 模块中代表线程安全的引用计数指针的 `Arc` 结构及用于保护共享数据的互斥原语 `Mutex` 结构体。

需要自己定义一个结构体 `BlockFile`：用于将文件转换为 `BlockDevice` 的包装

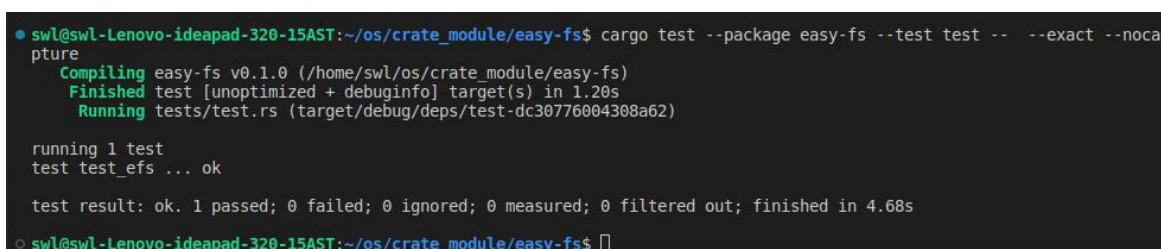
器，他的成员是 `Mutex<File>`，来满足 `BlockDevice` 特征，即定义的这个结构体 `BlockFile` 需要实现下列的两个方法：

```
read_block(&self, block_id: usize, buf: &mut [u8])
write_block(&self, block_id: usize, buf: &[u8])
```

为了实现 `read_block()`，首先使用 `lock().unwarp()`，用来保证没有两个线程或进程同时在他们的关键区域且通过文件的 `seek()` 来更改文件内部包含的逻辑游标，在流中寻找以字节为单位的偏移量。在这里的偏移量是 512 字节的倍数来确保将一个内容从文件中读入缓冲区的操作，可以模拟将一个块从磁盘读入内存中的缓冲区的操作。其次利用 `read()` 从文件中提取一些字节到指定的缓冲区中，返回读取的字节数，值得注意的是需要通过比较返回的字节数是否等于 512，来判断其能否作为从磁盘中读取块到内存中的缓冲区的操作。

实现 `write_block()` 的操作与 `read_block()` 的操作一样，只是将 `read()` 替换成了 `write()`，至此结构体 `BlockFile` 已经实现了 `BlockDevice` 特征需要的所有函数功能了，测试的准备工作完成了。

最后就是测试的正式工作了，先创建一个 `Arc<BlockFile>` 类型的值，将其作为参数来创建 `Arc<Mutex<EasyFileSystem>>`，之后就是依次运行接口函数，和预期值进行比较。例如创建根索引节点后，根据名称创建一些文件且使用 `readdir()` 函数，将得到的文件名称和预期得到的文件名称进行比较。`easy-fs` 模块单元测试结果如图 4-7 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/easy-fs$ cargo test --package easy-fs --test test -- --exact --nocapture
   Compiling easy-fs v0.1.0 (/home/swl/os/crate_module/easy-fs)
   Finished test [unoptimized + debuginfo] target(s) in 1.20s
   Running tests/test.rs (target/debug/deps/test-dc30776004308a62)

running 1 test
test test_efs ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 4.68s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/easy-fs$
```

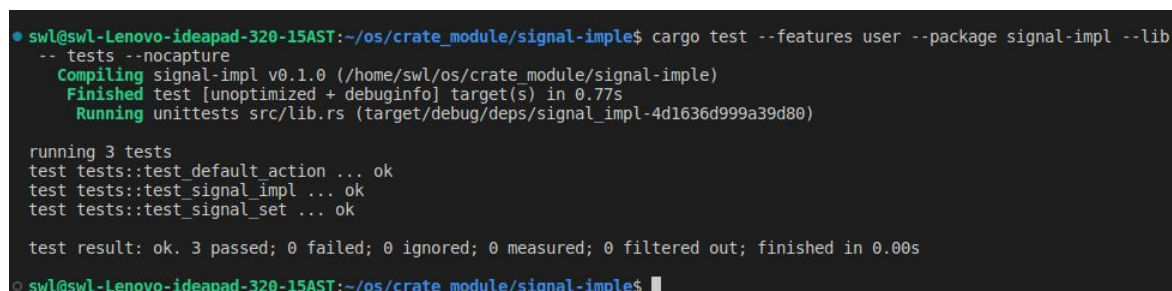
图 4-7 `easy-fs` 模块单元测试结果

4.8 `signal-imple` 模块的用户态单元测试

测试 `signal-imple` 模块需要依赖 `signal-defs` 模块、`signal` 模块及 `kernel-context` 模块，`signal` 模块和 `signal-imple` 模块对 `kernel-context` 模块的依赖会

导致运行出错，由此可知在进行测试之前需要将这两个模块对 kernel-context 模块的依赖注释掉；此外在 signal 模块增加 user 和 kernel 的 features，在 signal 调用 LocalContext: use kernel-context::LocalContext 上面加上#[cfg(feature = "kernel")], 添加代码#[cfg(feature = "user")], 在 user 特征下定义一个线程上下文结构 LocalContext，实现该结构一系列和特权级无关的方法，来代替运行操作系统内核时调用的 kernel-context 模块中的线程上下文结构。至此，测试 signal-imple 模块的准备工作完成，可以进行测例设置。

首先通过函数 new() 创建一个可变的信号管理器 sig1 和一个不可变的信号管理器 sig2，与此同时以 (&mut sig1) 为参数运行 fetch_signal() 函数，获取一个没有被 mask 屏蔽的信号，并从已收到的信号集合中删除该信号，如果没有这样的信号，则返回空。此时 sig1 是新建的空的信号管理器，这表明要判断 fetch_signal() 是否实现其功能，在这里应该将返回值和 None 进行比较，且运行 fetch_and_remove() 函数，将返回值与 false 进行比较。其次依次运行 from_fork()、ckear()、add_signal()、is_handling_signal() 及 update_mask() 函数，将返回值与预期值进行比较。signal-imple 模块单元测试结果如图 4-8 所示：



```
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/signal-impl$ cargo test --features user --package signal-impl --lib
-- tests --nocapture
Compiling signal-impl v0.1.0 (/home/swl/os/crate_module/signal-impl)
Finished test [unoptimized + debuginfo] target(s) in 0.77s
Running unittests src/lib.rs (target/debug/deps/signal_impl-4d1636d999a39d80)

running 3 tests
test tests::test_default_action ... ok
test tests::test_signal_impl ... ok
test tests::test_signal_set ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/signal-impl$
```

图 4-8 signal-imple 模块单元测试结果

4.9 sync 模块的用户态单元测试

sync 模块中的 let sie = sstatues::read().sie() 需要读取和修改寄存器的值，涉及到了内核特权级的操作，由此可知需要修改与寄存器相关的代码，否则在单独编译这一模块的代码的时候会进行报错。在关于 sstatues 寄存器的代码上方加上#[cfg(feature = "kernel")], 此外添加#[cfg(feature = "user")] let

sie = true;用无关的变量来代替 kernel 特征下对寄存器 sie 进行的操作。至此，同步互斥模块 sync 可以在 user 特征下进行单独一个模块的编译测试工作。

运行 up 模块中 UpSafeCellRaw 结构的 new()、get_mut() 方法，将预期值和返回值进行比较，其次运行 new() 函数创建一个 IntrMaskingInfo 类型的变量 intr1 且以 intr1 为参数运行 enter()、exit() 函数以及运行具有动态检查借用规则的可变内存位置 UpIntrFreeCell 结构的 exclusive_access() 方法，能够顺利通过。最后测试 condvar 模块，通过 new() 函数创建一个新的条件变量 condvar1, 创建一个线程号 tid2。sync 模块单元测试结果如图 4-9 所示：

```
● swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/sync$ cargo test --features user --package sync --lib -- --exact -
nocapture
Compiling sync v0.1.0 (/home/swl/os/crate_module/sync)
Finished test [unoptimized + debuginfo] target(s) in 1.07s
Running unittests src/lib.rs (target/debug/deps/sync-a023e6823456a127)

running 4 tests
test condvar::test_condvar ... ok
test semaphore::test_semaphore ... ok
test up::test_up ... ok
test mutex::test_mutex ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

图 4-9 sync 模块单元测试结果

结 论

针对目前高校计算机系统各课程实验衔接不紧密的问题^[2]，清华大学开发了基于 Rust 语言和 RISC-V 的操作系统内核实验。但其每章都是一个独立的操作系统，前一章实现的内容在后一章要重复实现，代码利用率很低。将 rCore 操作系统模块化之后，各章中相同的功能只实现一遍并作为库被各章的操作系统调用，提高了代码的利用率，但模块化之后写 rCore 内核的实验难度依然不低。

针对上述问题，本文首先完成了对 rCore 操作系统内核模块化的代码的探索，分析了模块化的 rCore 的整体结构及各模块的接口。

其次，本文对根据功能拆分的各个模块进行了用户态的单元测试，在学习如何写教学级的操作系统内核的时候，代码可以分成两个部分，一部分是只会和用户态相关的，一部分是需要读取修改寄存器等和内核的特权级切换相关的。现在有了各个模块的用户态单元测试，其他开发人员就不必像之前一样一上来就需要直接关注内核态的内容，可以先将用户态的内容做完，通过了用户态测试之后再去完成内核态的内容，将之前的整个任务分成了用户态和内核态两部分，相当于降低了整个任务的工作难度。

整体而言，本文完成了对 rCore 操作系统模块化的改进，通过增加每个模块的用户态单元测试，在一定程度上降低了其他开发者的工作难度。

但本文工作仍有可扩展的研究方向与不足之处：为每一个模块增加内核态单元测试。以同步互斥 sync 模块为例，就是要写一个假的操作系统内核来代替 rCore 处理寄存器和特权级的相关函数来进行测试。

参考文献

- [1] 杨德睿. 单核环境的模块化 Rust 语言参考实现[Z/OL]. <https://github.com/YdrMaster/rCore-Tutorial-in-single-workspace/>, 2023-2-13.
- [2] 孙卫真, 刘雪松, 朱威浦, 向勇. 基于 RISC-V 的计算机系统综合实验设计[J]. 计算机工程与设计, 2021, 42(04):1159-1165.
- [3] 苏铅坤, 颜庆茁, 郭晓曦. 操作系统课程实验平台设计与实践[J]. 福建电脑, 2023, 39(05):77-82.
- [4] 冯依嘉, 王雷, 孟丽平. 在 Windows 上使用虚拟机安装 Linux 操作系统[J]. 电脑编程技巧与维护, 2023(04):60-63+73.
- [5] 孙旺朝, 赵娅雯, 葛旭晴等. 基于 Linux 的《操作系统》实验教学改革[J]. 中国新通信, 2022, 24(19):119-121.
- [6] guomeng. 2021 年操作系统的商业化应用国内操作系统现状与前景分析[Z/OL]. <https://www.chinairn.com/news/20211020/104724656.shtml>, 2021-10-20.
- [7] 洛佳. 使用 Rust 编写操作系统（四）：内核测试[J/OL]. <https://zhuanlan.zhihu.com/p/90758552>, 2019-11-07.
- [8] 张润宇, 杨朝树. Linux 操作系统设备仿真教学探索与实践[J]. 福建电脑, 2023, 39(04):112-116. DOI:10.16707/j.cnki.fjpc.2023.04.023.
- [9] 李荣会, 李攀. 嵌入式 Linux 操作系统设备驱动程序设计与实现[J]. 计算机光盘软件与应用, 2012, No. 193(10):194-195.
- [10] 杨艳. Linux 操作系统在嵌入式设计中的分析与实现[J]. 电子世界, 2012, No. 410(20):108.
- [11] 陈枝清, 王雷. 基于 SCI 的 Linux 操作系统扩展研究[J]. 计算机应用研究, 2004(12):82-84.
- [12] 薛筱宇. 基于 Linux 内核的操作系统实验系统[D]. 西南交通大学, 2003.
- [13] Codie Wells. R68-52 The Structure of the "The"-Multiprogramming System. [J]. IEEE Trans. Computers, 1968, 17(12).
- [14] John Criswell, Andrew Lenharth, Dinakar Dhurjati, Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems[J]. Operating systems review, 2007, 41(6).
- [15] Grant Ayers, Heiner Litz, Christos Kozyrakis, Parthasarathy Ranganathan. Classifying Memory Access Patterns for Prefetching[P]. Architectural Support for Programming Languages and Operating Systems, 2020.
- [16] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems[P]. Operating systems design and implementation, 2006.
- [17] Mendel Rosenblum, Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. [J]. IEEE Computer, 2005, 38(5).

致 谢

值此论文完成之际，首先要向我的导师表示诚挚的感谢，因为我的导师对我进行了全面而细致的指导与帮助，从论文的选题、研究工作的展开到论文的完成都倾注了我导师的大量心血，在我因为思路出错而工作没有进展之时，尽管导师的平时工作非常繁忙，但他仍然会抽出时间通过线上视频的方式仔细帮助我分析哪里出现了问题，应该怎么修改才能解决问题，使得我在日常的工作学习中少走了很多弯路。

同时，我的导师亦非常关心我的身体健康，在我身体不舒服之后，会叮嘱我需要早睡早起，养成良好的作息习惯，要每天锻炼身体，多出去晒晒太阳。同时，亦要感谢我的同学，在遇到一些问题的时候，向同学请教他亦会耐心地帮助我，为我提供了一些技术上的支持。

最后感谢我的家人，是他们多年来对我学业的支持才让我走到这一步，才使我得以顺利完成学业。