

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 Realization of rcore Modular Improvement

学 院： 计算机学院

专 业： 计算机科学与技术

学生姓名： 石文龙

学 号： 1120173592

指导教师： 陆慧梅

2023 年 月 日

原创性声明

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

特此申明。

本人签名: _____ 日期: _____ 年 _____ 月 _____ 日

关于使用授权的声明

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

本人签名: 日期: 年 月 日

指导老师签名: 日期: 年 月 日

rcore 模块化改进的设计与实现

摘 要

本文主要针对目前用 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很难迁移到后一章的问题，发挥 Rust 语言 workspace/crates/traits 的先进设计理念，完成操作系统的模块化。本文的研究方法是：首先我们需要学习 Rust 编程和 RISC-V 处理器的相关内容，然后需要配置实验所需要的环境，主要包括 Linux 操作系统、Rust 开发环境配置、Qemu 模拟器安装；最后对实验室模块化的工作进行分析，然后将根据功能拆分的模块从一个工作空间中不同的 crates 进一步拆分到不同的工作空间，使只对一个操作系统内核某一部分感兴趣的同学能够只用关注那一部分的代码即可，不必像之前一样需要关注整个内核的代码；并且分别对各个模块进行用户态的单元测试，使得在只有一个模块的情况下也能够进行测试，能够直接对自己修改的代码能否正常工作作出判断，而不需要将自己修改后的代码放入整体的内核中进行测试。

关键词：系统编程语言；操作系统；模块化；单元测试

Design and Realization of rcore Modular Improvement

Abstract

This paper mainly aims at the problem that it is difficult to transfer the experimental results of the previous chapter to the next chapter in the current experiment of writing the rcore operating system kernel in the rust system programming language, and uses the advanced design concept of the Rust language workspace/crates/traits to complete the operating system. Modular. The research method of this paper is: first we need to learn the relevant content of Rust programming and RISC-V processor, and then we need to configure the environment required for the experiment, mainly including Linux operating system, Rust development environment configuration, Qemu simulator installation; Analyze the work of room modularization, and then further split the modules split according to functions from different crates in one workspace to different workspaces, so that students who are only interested in a certain part of an operating system kernel can only focus on That part of the code is enough, you don't need to pay attention to the code of the town kernel as before; and perform user-mode unit tests on each module, so that you can test even if there is only one module, and you can directly modify the Whether the code can work normally can be judged without putting the modified code into the overall kernel for testing.

Key Words: system programming language; operating system; Modular; unit test

目 录

摘 要.....	I
Abstract.....	II
第 1 章 绪论.....	1
1.1 研究背景和意义.....	1
1.2 研究现状.....	2
1.3 论文的主要研究内容.....	2
1.4 论文结构.....	3
第 2 章 用 rust 写 rcore 操作系统的准备工作.....	1
2.1 配置实验环境.....	1
第 3 章 rcore 操作系统模块化的实现与改进.....	2
3.1 将模块进一步分隔.....	2
3.2 console 模块.....	2
3.3 linker 模块.....	4
3.3.2 app.rs.....	5
3.4 kernel-context 模块.....	5
3.4.1 lib.rs.....	6
3.4.2 foreign/mod.rs.....	6
3.5 kernel-alloc 模块.....	8
对外接口.....	8
3.6 kernel-vm 模块.....	9
3.7 syscall 模块.....	10
3.8 task-manage 模块.....	11
3.9 easy-fs 模块.....	14
3.10 signal-def、signal 和 signal-imple 模块.....	14
3.11 sync 模块.....	15
3.12 在工作中遇到的问题和解决方法.....	17
3.12.1. 编译 task-manage 模块的时候报错.....	17
3.12.2. 运行时出错.....	18
结 论.....	19
参考文献.....	20

附 录.....	22
致 谢.....	23

第1章 绪论

1.1 研究背景和意义

操作系统是计算机的灵魂，目前国外操作系统品牌几乎垄断了巨大的中国场，其中在桌面端、移动端的市占率分别超过 94.75%、98.86%。根据 Gartner 的统计数据，2018 年中国的操作系统市场容量在 189 亿以上，其中国外操作系统品牌几乎在中国市场处于垄断地位。截至 2019 年 8 月，在中国的桌面操作系统市场领域，微软 Windows 的市占率 87.66%，苹果 OSX 的市占率为 7.09%，合计 94.75%；在中国的移动操作系统市场领域，谷歌 Android 的市占率为 75.98%，苹果 iOS 的市占率为 22.88%，合计为 98.86%。

虽然当前中国的操作系统市场依旧是微软的 windows+intel 占据了主导地位，但是 windows 的闭源架构正面临着以 linux 为代表的开源操作系统的挑战。中国的操作系统国产化浪潮起源与二十世纪末，目前正依托于开源操作系统的开源生态以及政策东风正在快速崛起，涌现出了中标麒麟、银河麒麟、深度 Deepin、华为鸿蒙等各种国产操作系统。所以在这个时期，我们进行操作系统的学习和研究是非常可行的。但是由于随着操作系统的功能实现的增多，其复杂程度也在增加；并且在对操作系统的学习过程中，因为各个章节是被 git 分支隔离开的，所以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，并且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，代码的复用性非常不好。

模块化编程，是强调将计算机程序的功能分离成独立的和可相互改变的“模块”的软件设计技术，它使得每个模块都包含着执行预期功能的一个唯一方面所必需的所有东西，复杂的系统被分割为小块独立代码块。在 rcore 内核设计中运用模块化编程之前，使用的是分支隔离的形式，做不同的课后实验时需要切换到不同的 lab 分支写代码，并且若要修改某一章的实现，就需要手动同步到后续所有章节，使用模块化编程之后，做不同的课后实验时只需要分别封装一个 crate 加入 workspace，然后运行指定的 package 即可。

rust 的模块化编程就能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，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，大大提高了代码的复用性。防止修改了某一模块的内容，但是后续章节没有修改造成的错误出现，减少了开发者出错的可能性。

1.2 研究现状

研究目标：在实验室工作的基础上，实现对于 `rcore` 内核模块化的改进与优化。主要完善实验室现有的模块化的 `rcore` 操作系统内核，针对现有的 `rcore` 操作系统内核模块化，提出了针对每一个章节模块的独立测试增加单元测试用例和在用户态对每一个模块进行单元测试的优化方向

主要内容：首先需要了解 `rust` 语言的使用和 `RISC-V` 架构，并且完成用 `rust` 写操作系统内核的 5 个实验，理解实验室当前对内核模块化的成果，在当前成果的基础上各个章节的模块进行内核态和用户态的单元测试。

目前实现的模块化，主要有在所有的章节中复用的代码形成了单独的 `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。

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`，做不同的章节的实验的时候，只需要封装一个 `crate` 到对应的 `package` 中，然后在运行的时候运行指定的 `package` 就可以了，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。

并且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。

1.3 论文的主要研究内容

首先，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，增加了每个章节的模块化完成之后的单元测试。增加了单元测试之后，我们能够进行小而集中的测试能够在隔离的环境中一次测试一个模块或者测试模块的私有接口，能够更加方便的检测出来是代码的哪个模块甚至是哪个函数出了问题。

在 `Rust` 中一个测试函数的本质就是一个函数，只是需要使用 `test` 属性进行标注或者叫做修饰，测试函数被用于验证非测试代码的功能是否与预期一致。在测试的函数体里经常会进行三个操作，即准备数据/状态，运行被测试的代码，断言结果。

在各章节的 `src` 目录下，每个文件都可以创建单元测试。标注了 `#[cfg(test)]` 的模块就是单元测试模块，它会告诉 `[Rust]` 只在执行 `cargotest` 时才编译和运行代码。在 `library` 项目中，添加任意数量的测试模块或者测试函数，之后进入该目录，在终端中输入 `cargo test` 运行测试。

1.4 论文结构

本篇论文在之后的结构如下：

（1）第二章主要讲述了用 `Rust` 系统编程语言写简单的操作系统内核需要的准备工作有哪些。主要是配置实验环境：安装 `Linux` 双系统，配置 `Rust` 开发环境：下

载安装 Rust 版本管理器 `rustup` 和 Rust 包管理器 `cargo`，下载并安装 Qemu 模拟器。

（2）第三章主要讲述了将 `crates` 拆分到不同的工作空间 `workspcae`，并且将每一个模块分别上传到一个单独的仓库中，在 QEMU 模拟器中自己写的简单操作系统内核任然能够工作；并且对实现 `print` 宏的 `console` 模块、提供链接脚本的 `linker` 模块、对内核上下文进行控制的 `kernel-context` 模块、对内存进行分配的 `kernel-alloc` 模块、对内核的虚拟内存进管理的 `kernel-vm` 模块、对进程和线程进行管理的 `task-manage` 模块、实现简易文件系统的 `easy-fs` 模块、处理信号的 `signal-imple` 模块以及实现同步互斥 `sync` 模块进行了分析并且写了对每一个模块进行用户态单元测试的设计思路 and 实现方法；最后写了在进行实验的过程中遇到的问题和解决方法。

（3）最后我主要讲述了本片论文的创新点，即将每一个模块拆分到不同的 `github` 仓库中，对每一个模块进行用户态单元测试，以及我的这次工作有什么意义，并且之后继续进行研究的方向。

第2章 用 rust 写 rcore 操作系统的准备工作

2.1 配置实验环境

在用 rust 写 rcore 操作系统之前，我们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。

目前，实验主要支持 Ubuntu18.04/20.04/22.04 操作系统。使用 Windows10 和 macOS 的读者，可以安装一台 Ubuntu18.04 虚拟机、使用 wsl2 或者配置 ubuntu 双系统。

我自己是在电脑上配置了一个 linux 双系统，

Rust 开发环境配置

首先安装 Rust 版本管理器 rustup 和 Rust 包管理器 cargo，可以使用官方安装脚本：

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

Qemu 模拟器安装

我们需要使用 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：

```
# 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 \
zlib1g-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

第3章 rcore 操作系统模块化的实现与改进

3.1 将模块进一步分隔

原本所有的模块是在一个工作区间，是不同的 package，然后将其每一个模块都分别放入一个 workspace 中，如果别人只对这个模块的内容感兴趣就可以只用在 github 上下载这一个模块而不需要和之前一样要把整个操作系统的内容全部下载下来。

运行系统需要进行的操作：`cargo qemu -ch n`，`n` 是章节号，选择范围是从 1 到 8，意思是在 qemu 模拟器中运行第 `n` 章的操作系统。后面可以选择的参数是：`--lab`，只对 `ch1` 有效，意思是运行 `ch1-lab` 的操作系统，`--features <features>`，该参数只对 `ch3` 有效，传入的参数是 `features = coop`。在 `ch5` 之后，我们实现了终端

```
[rustsbi] pmp02: 0x80000000..0x80200000 (---)
[rustsbi] pmp03: 0x80200000..0x84000000 (xwr)
[rustsbi] pmp04: 0x84000000..0x00000000 (-wr)

          _ _ _ _ _
         / / / / /
        / / / / /
       / / / / /
      / / / / /
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/ / / / /

[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

[ INFO] .text ----> 0x80200000..0x80256000
[ INFO] .rodata --> 0x80256000..0x80263000
[ INFO] .data ----> 0x80263000..0x80263850
[ INFO] .boot ----> 0x80264000..0x80284000
[ INFO] (heap) ---> 0x80284000..0x83200000

[ INFO] MMIO range -> 0x10001000..0x10002000
[ INFO] device features: SEG_MAX | GEOMETRY | BLK_SIZE | SCSI | FLUSH | TOPOLOGY | CONFIG_WCE | DISCARD | WRITE_Z
EROES | NOTIFY_ON_EMPTY | RING_INDIRECT_DESC | RING_EVENT_IDX
[ INFO] config: BlkConfig { capacity: Volatile(131072), size_max: Volatile(0), seg_max: Volatile(254), cylinders:
Volatile(130), heads: Volatile(16), sectors: Volatile(63), blk_size: Volatile(512), physical_block_exp: Volatile
(0), alignment_offset: Volatile(0), min_io_size: Volatile(0), opt_io_size: Volatile(0) }
[ INFO] found a block device of size 65536KB
Rust user shell
>> |
```

图 3-1 分隔后在 Qemu 上运行的结果

3.2 console 模块

console 模块的功能是提供可以定制的 `print!`、`println!` 和 `log::Log`；目前日志只能提供基础的彩色功能。

console 模块提供的宏定义有两个，分别是 `print`：格式化打印；和 `println`：格式化打印并换行。

console 模块的对外接口有 4 个，分别是一个 trait 和三个函数：

```
pub trait Console: Sync
```

这个接口定义了向控制台“输出”这件事，同时满足 console 特征。

```
pub fn init_console(console: &'static dyn Console)
```

这个函数的作用是用户能够调用这个函数来设置输出的方法, 初始化 console。

```
pub fn set_log_level(env: Option<&str>)
```

这个函数的作用是根据环境变量设置日志级别。

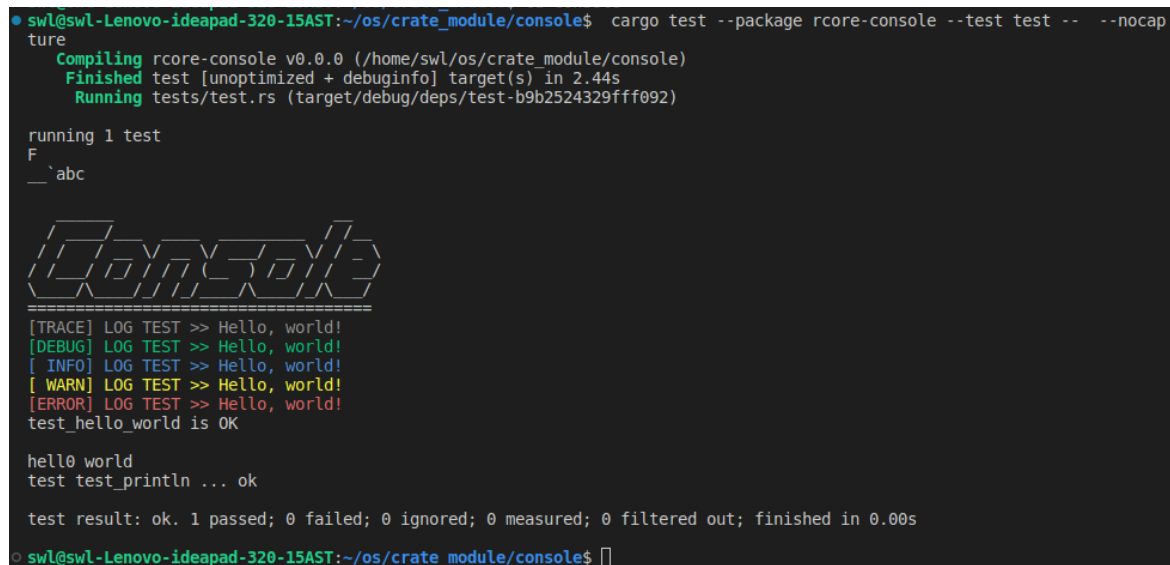
```
pub fn test_log()
```

这个函数的作用是打印一些测试信息。

在 Rust 中，测试是通过函数的方式实现的，它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。

在用户态测试 console 之前，首先需要定义一个结构 Console1，并且需要为 Console1 结构实现 console 模块定义的 trait Console: Sync，为了实现 Console trait 需要实现 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，然后调用标准库中的 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，我们就可以依次运行想要测试的代码了，在这里就是依次运行 init_console() 函数来初始化 console 结构，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。



```
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 3-1 console 模块单元测试结果

3.3 linker 模块

linker 板块为内核提供链接脚本的文本，以及依赖于定制链接脚本的功能。`build.rs` 文件可依赖此板块，并将 `[SCRIPT]` 文本常量写入链接脚本文件

```
use std::{env, fs, path::PathBuf};

let ld =
    &PathBuf::from(env::var_os("OUT_DIR").unwrap()).join("linker.ld");
fs::write(ld, linker::SCRIPT).unwrap();

println!("cargo:rerun-if-changed=build.rs");
println!("cargo:rustc-link-arg=-T{}", ld.display());
```

内核使用 `boot0` 宏定义内核启动栈和高级语言入口：

定义内核入口，即设置一个启动栈，并在启动栈上调用高级语言入口。
内核链接脚本的结构将完全由这个板块控制。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"`。

`macro_rules! boot0`

- `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。

`KernelLayout` 的结构为：`pub struct KernelLayout { text: usize, rodata: usize, data: usize, sbss: usize, ebss: usize, boot: usize, end: usize, }`

该结构有 6 个方法，分别为 `pub fn locate()` 该方法作用为定位内核布局。

`pub const fn start(&self)` 该方法作用为得到内核起始地址。

`pub const fn end(&self)` 该方法作用为得到内核结尾地址。

`pub const fn len(&self)` 该方法作用为得到内核静态二进制长度。

`pub unsafe fn zero_bss(&self)` 该方法作用为清零 `.bss` 段。

`Pub fn iter(&self)` 该方法作用为得到内核区段迭代器。

`KernelRegion` 结构为：

```
pub struct KernelRegion {
    /// 分区名称。 pub title: KernelRegionTitle,
    /// 分区地址范围。 pub range: Range, }
```

该结构的含义是内核内存分区。该结构存在 `fmt` 方法。

`fn fmt(&self, f: &mut fmt::Formatter<'_>)` 该方法的作用是使用给定的格式化程序格式化值。

KernelRegionIterator 结构为：

```
pub struct KernelRegionIterator<'a> {  
    layout: &'a KernelLayout, //内核内存分区名称  
    next: Option, } 该结构的含义是内核内存分区迭代器。该结构存在 next 方法。  
fn next(&mut self) 该方法的作用是得到迭代器中下一位的值。
```

3.3.2app.rs

AppMeta: 应用程序元数据。

AppIterator: 应用程序迭代器。

AppMeta 结构为：

```
pub struct AppMeta { base: u64, step: u64, count: u64, first: u64, } 该结构的含义是应用程序元数据。该结构有 2 个方法，分别为：
```

```
pub fn locate() 该方法的作用是定位应用程序。
```

```
pub fn iter(&'static self) 该方法的作用是遍历链接进来的应用程序。
```

AppIterator 结构为：

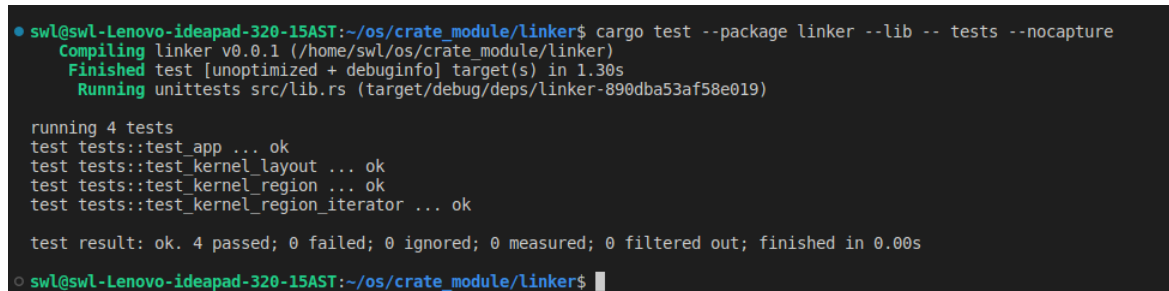
```
pub struct AppIterator { meta: &'static AppMeta, i: u64, } 该结构的含义是应用程序迭代器。该结构有一个 next 方法：
```

```
fn next(&mut self) 该方法的作用是对应用程序进行迭代。
```

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。



```
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 3-2 linker 模块单元测试结果

3.4 kernel-context 模块

kernel-context 的实现

该模块实现了内核上下文的控制，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。
- PortalText: 传送门代码。
- MultislotPortal: 包含多个插槽的异界传送门。

3.4.1 lib.rs

```
pub struct LocalContext {
    sctx: usize,
    x: [usize; 31],
    spec: usize,
    /// 是否以特权态切换。
    pub supervisor: bool,
    /// 线程中断是否开启。
    pub interrupt: bool,
}
```

该结构包含 14 个方法，其分别为：

pub const fn empty() 该方法的作用是创建空白上下文。

pub const fn user(pc: usize) 该方法的作用是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。

pub const fn thread(pc: usize, interrupt: bool) 该方法的作用是初始化指定入口的内核上下文。

pub fn x(&self, n: usize) pub fn a(&self, n: usize) pub fn ra(&self) pub fn sp(&self) pub fn pc(&self) 该方法的作用分别是读取用户通用寄存器；读取用户参数寄存器；读取用户栈指针；读取用户栈指针；读取当前上下文的 pc。

pub fn x_mut(&mut self, n: usize) pub fn a_mut(&mut self, n: usize) pub fn sp_mut(&mut self) pub fn pc_mut(&mut self) 该方法的作用分别是修改用户通用寄存器；修改用户参数寄存器；修改用户栈指针；修改上下文的 pc。

pub fn move_next(&mut self) 该方法的作用是将 pc 移至下一条指令。

pub unsafe fn execute(&mut self) 该方法的作用是执行此线程，并返回 sstatus, 将修改 sscratch、sepc、ssstatus 和 stvec。

3.4.2 foreign/mod.rs

```
pub struct PortalCache {
    a0: usize,          // (a0) 目标控制流 a0
    a1: usize,          // 1*8(a0) 目标控制流 a1      (寄存，不用初始化)
    satp: usize,        // 2*8(a0) 目标控制流 satp
```

```

    sstatus: usize, // 3*8(a0) 目标控制流 sstatus
    sepc: usize,    // 4*8(a0) 目标控制流 sepc
    stvec: usize,   // 5*8(a0) 当前控制流 stvec    (寄存, 不用初始化)
    sscratch: usize, // 6*8(a0) 当前控制流 sscratch (寄存, 不用初始化)
}

```

该结构是传送门缓存, 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 `pub fn init(&mut self, satp: usize, pc: usize, a0: usize, supervisor: bool, interrupt: bool)` 该方法的作用是初始化传送门缓存。 `pub fn address(&mut self)` 该方法的作用是返回缓存地址。

```

pub struct ForeignContext {
    /// 目标地址空间上的线程上下文。
    pub context: LocalContext,
    /// 目标地址空间。
    pub satp: usize,
}

```

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。该结构一共有 1 中方法, 其是

`pub unsafe fn execute(&mut self, portal: &mut impl ForeignPortal, key: impl SlotKey)` 该方法的作用是执行异界线程。

```

struct PortalText(&'static [u16]);

```

该结构的作用是传送门代码。该结构一共有 3 种方法, 分别是: `pub fn new()` `pub fn aligned_size(&self)` `pub unsafe fn copy_to(&self, address: usize)`

foreign/multislot_portal.rs

```

pub struct MultislotPortal {
    slot_count: usize,
    text_size: usize,
}

```

该结构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:

`pub fn calculate_size(slots: usize)` 该方法的作用是计算包括 `slots` 个插槽的传送门总长度。

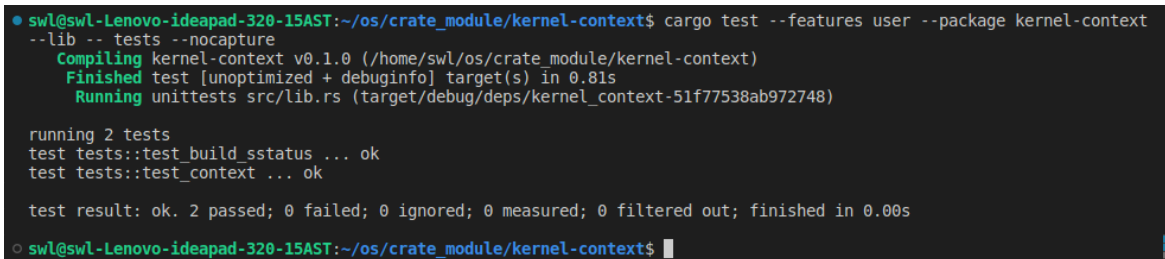
`pub unsafe fn init_transit(transit: usize, slots: usize)` 该方法的作用是初始化公共空间上的传送门。其中参数 `transit` 必须是一个正确映射到公共地址空间上的地址。

kernel-context 模块的用户态测试也比较简单, 就是调用该模块自己的结构:

LocalContext 线程上下文。则测试所需要的依赖环境就已经完成了, 接下来只需要依次运行 LocalContext 结构的方法, 并且通过 `assert_eq()` 函数判断回的结果是否符合预期就完成 kernel-context 模块的用户态单元测试了。

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, 然后通过 `assert_eq()` 函数来比较创建的空白上下文两个成员 `supervisor`: 是否以特权态切换, `interrupt`: 线程中断是否开启; 是否为 `false`, 如果这两个成员的值都为 `false`, 并且比较通过 `pc()` 方法得到的 `sepc` 成员: 当前上下文的 `pc` 地址是否与预期的 0 相等; 则 `assert_eq` 函数顺利通过, 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。该结构的其他方法也是一样，先运行 user(pc: 04) 方法，然后判断 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，然后判断 supervisor 是否等于 true，interrupt 是否等于 false，pc() 函数是否返回 04；thread(04,true) 方法，然后判断 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(),pc() 等读取类函数，判断他们是否分别与 0,0,0,0,04 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(),pc_mut() 等修改类函数，自己在测试之前写一个 LocalContext 结构，然后以这个 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。



```

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 3-3 kernel-context 模块单元测试结果

3.5 kernel-alloc 模块

内核内存管理

这个模块提供 #[global_allocator]。

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。

测试 transfer() 时，参数正确还是失败，准备分开测试一下其中的代码。

HEAP.transfer(ptr, region.len()); 失败 Heap::new() 创建的是空分配器，总容量为 0，所以分配失败。测试 alloc() 时，会运行 handle_alloc_error(layout)。

失败原因：memory allocation of 5 bytes failed; 5 个字节的内存分配失败（信号：6，SIGABRT：进程中止信号）

kernel-alloc 的实现

对外接口

pub fn init(base_address: usize)

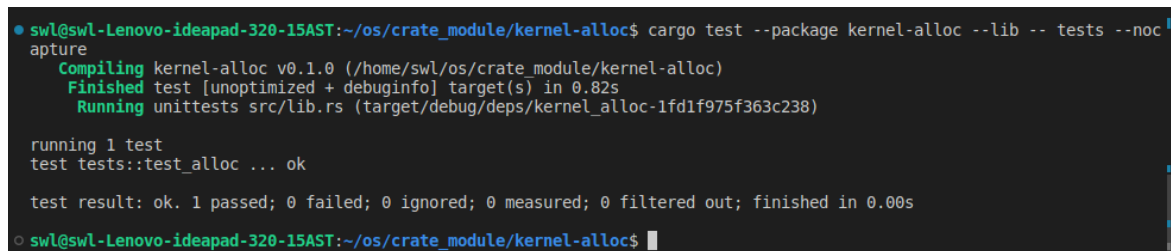
这个函数用于初始化内存分配。用户需要告知内存分配器参数 base_address

表示的动态内存区域的起始位置;内存区域的起始位置用于计算伙伴分配器的参数基址。

`pub unsafe fn transfer(region: &'static mut [u8])`

这个函数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。并且需要注意这个函数是不安全的。

单独运行 `kernel-alloc` 模块的时候，会对报错，所以在会报错的 `Gobal` 结构及其方法之上添加了 `#[cfg(feature = "kernel")]`，然后在测试的时候定义一个内核地址信息结构 `KernelLayout`，该结构有 `text`：开始地址和 `end`：结束地址两个成员，以及实现 `start()`, `end()`, `len()` 三个方法，分别得到内核开始地址，内核结束地址和内核静态二进制长度；



```

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 3-4 `kernel-alloc` 模块单元测试结果

3.6 `kernel-vm` 模块

`cargo test --package kernel-vm --lib -- tests --nocapture` 定义了实现 `VmMeta` 特征的 `SV39` 结构和实现了 `PageManager` 特征的 `Sv39Manager` 结构体。

需要赋值一个物理页 `range: Range<PPN>`;

`kernel-vm` 的实现

内核虚存管理 `kernel-vm` 模块的主要内容是内核虚拟存储的管理。

`space/mod.rs`

- `AddressSpace`：地址空间结构。

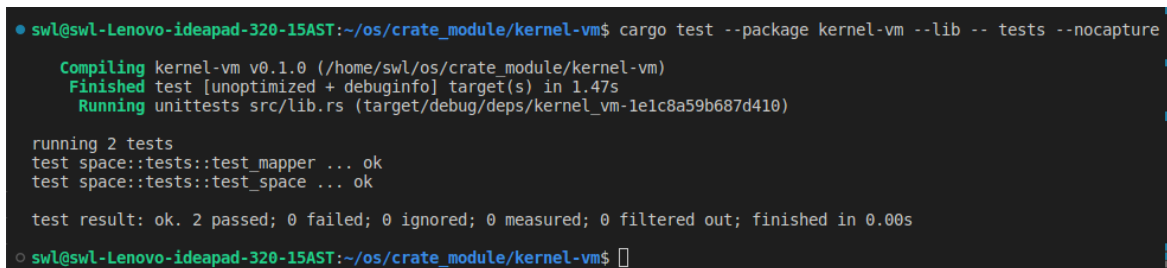
`pub struct AddressSpace<Meta: VmMeta, M: PageManager> {` /// 虚拟地址块
`pub areas: Vec<Range<VPN>>, page_manager: M, }` 该结构共有 7 个方法，分别为：
`pub fn new()` 该方法的作用是创建新地址空间。
`pub fn root_ppn(&self)` 该方法的作用是得到地址空间根页表的物理页号。
`pub fn root(&self)` 该方法的作用是得到地址空间根页表
`pub fn map_exten(&mut self, range: Range<VPN>, pbase: PPN, flags: VmFlags)` 该方法的作用是向地址空间增加映射关系。
`pub fn map(&mut self, range: Range<VPN>, data: &[u8], offset: usize, mut flags: VmFlags,)` 该方法的作用是分配新的物理页，拷贝数据并建立映射。
`pub fn translate(&self, addr: VAddr, flags: VmFlags)` 该方法的作用是检查 `flags` 的属性要求，然后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。 pub fn cloneself(&self, new_addrspace: &mut AddressSpace<Meta, M>) 该方法的作用是遍历地址空间，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。

space/mapper.rs

```
pub(super) struct Mapper<'a, Meta: VmMeta, M:
PageManager<Meta>> {
    space: &'a mut AddressSpace<Meta, M>,
    range: Range<PPN<Meta>>,
    flags: VmFlags<Meta>,
    done: bool,
}
```

该结构有 5 个方法，分别是： pub fn new(space: &'a mut AddressSpace<Meta, M>, range: Range<PPN>, flags: VmFlags,) 该方法的作用是创建一个新的 Mapper。 pub fn ans(self) 该方法的作用是得到 Mapper 结构的 done 值。 fn arrive(&mut self, pte: &mut Pte, target_hint: Pos) fn meet(&mut self, _level: usize, pte: Pte, _target_hint: Pos,) fn block(&mut self, _level: usize, pte: Pte, _target_hint: Pos)



```
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 3-5 kernel-vm 模块单元测试结果

3.7 syscall 模块

系统调用

这个库封装了提供给操作系统和用户程序的系统调用。

系统调用号从 Musl Libc for RISC-V 源码生成，因为找不到标准文档。

syscall 系统调用的实现

对外接口

pub struct SyscallId(pub usize);

Structs

ClockId

SignalAction

信号处理函数的定义

SyscallId

系统调用号。

TimeSpec
Enums
SignalNo
信号编号。
Constants
MAX_SIG
最大的信号编号
STDDEBUG
STDIN
STDOUT

3.8 task-manage 模块

我们将开发一个用户终端(Terminal)或命令行(CommandLineApplication, 俗称 Shell), 形成用户与操作系统进行交互的命令行界面, 为此, 我们要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。

任务是这里提到的进程的初级阶段, 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。

任务管理

任务 id 类型, 自增不回收, 任务对象之间的关系通过 id 类型来实现

- ProcId
- ThreadId
- CoroId 结构 ProcId 的方法有 3 个, 分别是 new(),from_usize(),get_usize;
- new()方法创建了一个进程编号自增的进程 id 类型,
- from_usize()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()方法需要输入的参数是一个 ProcId 的引用, 返回该 ProcId 结构对应的 id。

结构 ThreadId 的方法与 ProcId 的方法相同。

任务对象管理 manage trait, 对标数据库增删改查操作

- insert
- delete
- get_mut

任务调度 schedule trait, 队列中保存需要调度的任务 Id

- add: 任务进入调度队列
- fetch: 从调度队列中取出一个任务

封装任务之间的关系, 使得 PCB、TCB 内部更加简洁

- ProcRel: 进程与其子进程之间的关系
- ProcThreadRel: 进程、子进程以及它地址空间内的线程之间的关系

ProcRel 结构包含父进程 id, 子进程列表和已经结束的进程列表。ProcRel 结构有 new()方法, add_child(),del_child(),wait_any_child(),wait_child()5 种方

法。

- `new()`方法需要输入父进程 `ProcId`，返回一个 `ProcRel` 结构，其中父进程 `ProcId` 是输入的参数，子进程列表和已经结束的进程列表使新创建的动态数组。
- `add_child()`方法的参数是一个 `ProcRel` 的可变引用和一个子进程 `ProcId`，该方法的作用是将参数子进程 `id` 放入到输入的 `ProcRel` 的子进程列表中。
- `del_child()`方法的参数有一个 `ProcRel` 的可变引用、一个子进程 `ProcId` 和一个退出码 `exit_code`，该方法的作用是：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。
- `wait_any_child()`方法的参数是一个 `ProcRel` 的可变引用，该方法的作用是：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`。
- `wait_child` 方法的参数有一个 `ProcRel` 的可变引用和一个子进程 `ProcId`，该方法的作用是：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`。

封装任务之间的调度方法

- **PManager**：管理进程以及进程之间的父子关系
 - **PThreadManager**：管理进程、子进程以及它地址空间内的线程之间的关系
- `PManager` 结构为：`pub struct PManager<P, MP: Manage<P, ProcId> + Schedule> { // 进程之间父子关系 rel_map: BTreeMap<ProcId, ProcRel>, // 进程对象管理和调度 manager: Option, // 当前正在运行的进程 ID current: Option, phantom_data: PhantomData`
`, }`

该结构办函的方法共有 9 个，分别为：`pub const fn new() -> Self` 此方法用于新建 `PManager` `pub fn find_next(&mut self) -> Option<&mut P>` 此方法用于找到下一个进程 `pub fn set_manager(&mut self, manager: MP)` 此方法用于设置 `manager` `pub fn make_current_suspend(&mut self)` 此方法用于阻塞当前进程 `pub fn make_current_exited(&mut self, exit_code: isize)` 此方法用于结束当前进程，只会删除进程的内容，以及与当前进程相关的关系 `pub fn add(&mut self, id: ProcId, task: P, parent: ProcId)` 此方法用于 添加进程，需要指明创建的进程的父进程 `Id` `pub fn current(&mut self) -> Option<&mut P>` 此方法用于获取当前进程 `#[inline] pub fn get_task(&mut self, id: ProcId) -> Option<&mut P>` 此方法用于获取某个进程 `pub fn wait(&mut self, child_pid: ProcId) -> Option<(ProcId, isize)>` 此方法用于 `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)`

运行所有满足 `features = proc` 的测试：`cargo test --features proc --package rcore-task-manage --lib -- tests --nocapture` 运行所有满足 `features = thread` 的测试：`cargo test --features thread --package rcore-task-manage --lib -- tests --nocapture`

运行 `id` 模块的测试：`cargo test --package rcore-task-manage --lib -- tests::test_id --exact --nocapture` 运行某一个特定的测试，以 `test_proc_rel` 为例：

```
cargo test --features proc --package rcore-task-manage --lib -- tests::test_proc_rel --exact --nocapture
```

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`，没有 `feature` 的时候调用 `id`，`manage` 和 `Schedule`；至此，测试的依赖环境搭建完成。

然后，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，然后比较第一次 `nw()` 函数的结果是否与 `from_usize(0)` 的结果相同，比较第二次 `nw()` 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，并且 `from_usize(v)` 函数也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`，然后根据父进程 `id` 创建一个进程关系 `procrel`，然后以进程关系 `procrel` 为参数运行等待子进程结束的函数 `wait_any_child()` 和 `wait_child()`，并且将他们的返回值和 `None` 进行比较；然后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，并且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；然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，接着等待子进程后，`dead_children` 队列会为空，再运行一次等待子进程的结果等于 `dead_children` 队列的头部等于 `None`；在添加子进程后再结束子进程，此时等待子进程的结果等于 `Some((child_id, 1))`；至此，`proc_rel` 测试完成。

在测试 `proc_manage` 时，首先需要创建一个进程结构 `Process`，该结构只包含一个进程 `id`，并且 `Process` 结构需要能够根据 `proc_id` 创建进程，然后创建一个满足 `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 `add()` 添加 `id` 进入调度队列，`fetch()` 从调度队列中取出 `id`；至此，测试需要的准备工作完成了。然后新建一个满足 `MP` 的 `procmanage` 结构和管理父进程和子进程的 `pmanage` 结构，然后依次运行设置 `manage`，添加子进程和获取指定进程的函数，并且将 `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。

```

● 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 3-6 task-manage 模块单元测试结果

3.9 easy-fs 模块

本章我们将实现一个简单的文件系统 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，它们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。

3.10 signal-def、signal 和 signal-imple 模块

测试 signal-imple 模块需要依赖 signal-def 模块和 signal 模块，并且 signal 模块和 signal-imple 模块对 kernel-context 模块的依赖会导致运行出错，所以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；并且在 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")], 并且添加代码#[cfg(feature = "user")], 在 user 特征下定义一个线程上下文结构 LocalContext，并且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了。

然后，运行想要测试的函数，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；接着以 (&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它；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，所以要判断 fetch_signal() 是否实现其功能，在这里应该将返回值和 None 进行比较；接着运行 fetch_and_remove() 函数，将返回值与 false 进行比较；然后依次运行 from_fork()、ckear()、add_signal()、is_handling_signal() 函数，然后将 is_handling_signal() 函数的返回值与 false 进行比较。然后运行两次 update_mask() 函数，并且参数分别为 0001 和 0002，比较两次的返回值是否分别为 0000 和 0001。


```

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 3-7 signal-impl 模块单元测试结果

3.11 sync 模块

到本章开始之前，我们好像已经完成了组成应用程序执行环境的操作系统的三个重要抽象：进程、地址空间和文件，让应用程序开发、运行和存储数据越来越方便和灵活。有了进程以后，可以让操作系统从宏观层面实现多个应用的并发执行，而并发是通过操作系统基于处理器的时间片不断地切换进程来达到的。到目前为止的并发，仅仅是进程间的并发，对于一个进程内部还没有并发性的体现。而这正是线程（Thread）出现的起因：提高一个进程内的并发性。

有了进程以后，为什么还会出现线程呢？考虑如下情况，对于很多应用（以单一进程的形式运行）而言，逻辑上存在多个可并行执行的任务，如果其中一个任务被阻塞，将会引起不依赖该任务的其他任务也被阻塞。举个具体的例子，我们平常用编辑器来编辑文本内容的时候，都会有一个定时自动保存的功能，这个功能的作用是在系统或应用本身出现故障的情况前，已有的文档内容会被提前保存。假设编辑器自动保存时由于磁盘性能导致写入较慢，导致整个进程被操作系统挂起，这就会影响到用户编辑文档的人机交互体验：即软件的及时响应能力不足，用户只有等到磁盘写入完成后，操作系统重新调度该进程运行后，用户才可编辑。如果我们把一个进程内的多个可并行执行任务通过一种更细粒度的方式让操作系统进行调度，那么就可以通过处理器时间片切换实现这种细粒度的并发执行。这种细粒度的调度对象就是线程。

3.11.1 sync 的实现

该模块是同步互斥模块

condvar.rs

- Condvar: Condvar 条件变量
- CondvarInner: Condvar 条件变量的内部结构 `pub struct CondvarInner { /// block queue pub wait_queue: VecDeque, }`
`pub struct Condvar { /// UPIntrFreeCell pub inner: UPIntrFreeCell, }` 该结构有 4 个方法，分别为：`pub fn new()` 该方法的作用是创建一个新的条件变量结

构。 `pub fn signal(&self)` 该方法的作用是唤醒某个阻塞在当前条件变量上的线程。 `pub fn wait_no_sched(&self, tid: ThreadId)` 该方法的作用是将当前线程阻塞在条件变量上。 `pub fn wait_with_mutex(&self, tid: ThreadId, mutex: Arc,)` 该方法的作用是从 `mutex` 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。

`mutex`

- `MutexBlockingInner`:

- `MutexBlocking`: 该结构实现了互斥。

`pub struct MutexBlockingInner { locked: bool, wait_queue: VecDeque, }`
`pub struct MutexBlocking { inner: UPIntrFreeCell, }` 该结构有 3 个方法，分别为
`pub fn new()` 该方法的作用是创建一个新的 `MutexBlocking` 结构。 `fn lock(&self, tid: ThreadId)` 该方法的作用是获取锁，如果获取成功，返回 `true`，否则会返回 `false`，要求阻塞对应的线程。 `fn unlock(&self)` 该方法的作用是 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`

- `SemaphoreInner`:

- `Semaphore`:

`pub struct SemaphoreInner { pub count: isize, pub wait_queue: VecDeque, }`
`pub struct Semaphore { /// UPIntrFreeCell pub inner: UPIntrFreeCell, }` 该结构有 3 个方法，分别是：
`pub fn new(res_count: usize)` 该方法的作用是创建一个新的 `Semaphore` 结构。
`pub fn up(&self)` 该方法的作用是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。
`pub fn down(&self, tid: ThreadId)` 该方法的作用是前线程试图获取信号量表示的资源，并返回结果。

`up.rs`

- `UPSafeCellRaw`: 内部可变性
- `IntrMaskingInfo`: 中断屏蔽信号
- `UPIntrFreeCell`: 具有动态检查借用规则的可变内存位置
- `UPIntrRefMut`: 从 `RefCell` 可变借用值的包装器类型

`pub struct IntrMaskingInfo { nested_level: usize, sie_before_masking: bool, }`

使用方式

有些模块在有特却及不能在用户态测试

有关寄存器与特权级用无关变量代替 模拟存储单元写一个和 `sstaus` 的接口一样的模块 根据编译条件的不同

将 `mod up` 等前面加了 `pub`

`cargo test --features user --package sync --test test -- --nocapture`

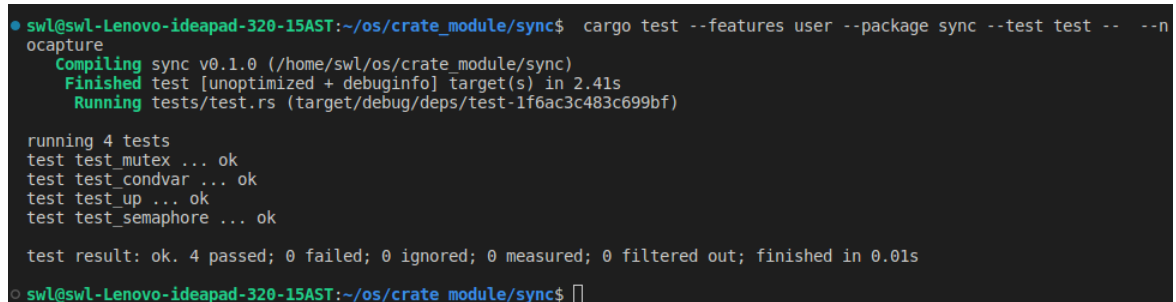
`cargo test --features user --package sync --test test -- test_up --exact --nocapture`

`cargo test --features user --package sync --test test -- test_condvar --exact --nocapture`

`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，所以需要修改这一部分的代码，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”)], 并且添加#[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。

首先运行 up 模块中 UpSafeCellRaw 结构的 new()、get_mut()方法, 并且将预期值和返回值进行比较; 然后运行 new()函数创建一个中断屏蔽信息 IntrMaskingInfo 结构: intr1, 然后以 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。然后测试 condvar 模块, 通过 new()函数创建一个新的条件变量 condvar1, 并且创建一个线程 idtid2,



```

swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/sync$ cargo test --features user --package sync --test test -- --n
ocapture
   Compiling sync v0.1.0 (/home/swl/os/crate_module/sync)
   Finished test [unoptimized + debuginfo] target(s) in 2.41s
   Running tests/test.rs (target/debug/deps/test-1f6ac3c483c699bf)

running 4 tests
test test_mutex ... ok
test test_condvar ... ok
test test_up ... ok
test test_semaphore ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/sync$ 

```

图 3-8 sync 模块单元测试结果

3.12 在工作中遇到的问题 and 解决方法

3.12.1. 编译 task-manage 模块的时候报错

error[E0554]: ‘#![feature]’ may not be used on the stable release channel

编译的时候报错, 是因为当前编译使用的 channel 是稳定版本的, 还没有包含#![feature]功能, 需要换成 nightly 版。

具体的操作步骤是: 要使用 beta 和 nightly 版首先要看下有没有安装: rustup toolchain list

以安装 nightly 为例: rustup toolchain install nightly 安装好后怎么使用呢?

方式一: 比较简单的方式是直接安装加更改当前系统默认的 channel rustup default nightly

方式二: 使用 rustup run 指定 channel rustup run nightly cargo build

方式三: 使用 rustup overwrite 设置当前项目使用的 channel 进入项目目录执行: rustup override set nightly

3.12.2. 运行时出错

Blocking waiting for file lock on package cache

如果确定没有多个程序占用，可以用下面的命令删除 package-cache 缓存文件

```
rm -rf ~/.cargo/.package-cache
```

然后重新运行就可以了。

结 论

本篇论文进行了对 rcore 操作系统内核模块化的代码分析和实现，以及在现有的模块化的基础上进一步将一个工作空间中不同的模块分别放入不同的工作空间中并且每个模块都上传到各自的 github 仓库中，以便之后只对操作系统内核某一个模块感兴趣：比如只对简易文件系统感兴趣但是对其他的模块不感兴趣的同学就可以只用下载这一个模块的内容并加以修改就可以了，而不必和之前一样需要将整个操作系统内核的内容都下载下来才能对自己感兴趣的模块进行修改，降低了整个操作系统内核各个模块之间的耦合。并且之后如果要写更复杂的操作系统的话，就不可能像现在关注教学这样一个人会对操作系统内核的所有模块都有研究，势必要进行更加细致的分工，很有可能一个人只会负责一个模块，并且因为有保密需求，除了总负责人，负责一个模块的人不能了解到其他模块的具体实现过程，细化分隔之后能够很好的满足这些要求。

本篇论文完成的工作还包括了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。涉及到特权级切换与内核有紧密关联的部分没有变，但是与内核关联不那么深的部分就降低了工作的难度。

本篇论文今后进一步在本研究方向进行研究工作的设想是完成对每一个模块的内核态进行测试，以 console 模块为例就是在测试的部分写一个假的操作系统内核并且测试 console 模块在假内核上能否正常工作，具体的做法就是调用 Rust 的核心库 core 里面能够向控制台输出的函数来实现 put_char()；以同步互斥模块 sync 模块威力就是要写一个假的操作系统内核但是要能够正确处理寄存器和特权级的相关问题。

参考文献

参考文献书写规范

参考国家标准《信息与文献参考文献著录规则》【GB/T 7714—2015】，参考文献书写规范如下：

1. 文献类型和标识代码

普通图书：M 会议录：C 汇编：G 报纸：N
期刊：J 学位论文：D 报告：R 标准：S
专利：P 数据库：DB 计算机程序：CP 电子公告：EB
档案：A 舆图：CM 数据集：DS 其他：Z

2. 不同类别文献书写规范要求

期刊

[序号]主要责任者. 文献题名[J]. 刊名, 出版年份, 卷号(期号): 起止页码.

[1] 余雄庆. 飞机总体多学科设计优化的现状与发展方向[J]. 南京航空航天大学学报, 2008,40(4): 417-426.

[2] Hajela P., Bloebaumj C. L., Sobieszczanski-Sobieski J. Application of Global Sensitivity Equations in Multidisciplinary Aircraft Synthesis[J]. Journal of Aircraft, 1990, 27(12):1002-110.

普通图书

[序号]主要责任者. 文献题名[M]. 出版地: 出版者, 出版年. 起止页码.

[3] 李成智, 李小宁, 田大山. 飞行之梦—航空航天发展史概论[M]. 北京: 北京航空航天大学, 2004.

[4] Raymer D. P. Aircraft Design: A Conceptual Approach[M]. Reston, Virginia: American Institute of Aeronautics and Astronautics, Inc., 2006.

会议论文集

[序号]析出责任者. 析出题名[A]. 见(英文用 In): 主编. 论文集名[C]. (供选择项: 会议名, 会址, 开会年)出版地: 出版者, 出版年. 起止页码.

[5] 孙品一. 高校学报编辑工作现代化特征[A]. 见: 张为民编. 中国高等学校自然科学学报研究会. 科技编辑学论文集(2)[C]. 北京: 北京师范大学出版社, 1998. 10-22.

专著中析出的文献

[序号]析出责任者. 析出题名[A]. 见(英文用 In): 专著责任者. 书名[M]. 出版地: 出版者, 出版年. 起止页码.

[6] 罗云. 安全科学理论体系的发展及趋势探讨[A]. 见: 白春华, 何学秋, 吴宗之. 21 世纪安全科学与技术的发展趋势[M]. 北京: 科学出版社, 2000. 1-5.

学位论文

[序号]主要责任者. 文献题名[D]. 保存地: 保存单位, 年份.

[7] 张和生. 嵌入式单片机系统设计[D]. 北京: 北京理工大学, 1998.

[8] Sobieski I. P. Multidisciplinary Design Using Collaborative Optimization[D]. United States -- California: Stanford University, 1998.

报告

[序号]主要责任者. 文献题名[R]. 报告地: 报告会主办单位, 年份.

[9] 冯西桥. 核反应堆压力容器的 LBB 分析[R]. 北京: 清华大学核能技术设计研究院, 1997.

[10] Sobieszczanski-Sobieski J. Optimization by Decomposition: A Step from Hierarchic to Non-Hierarchic Systems[R], NASA CP-3031, 1989.

专利文献

[序号]专利所有者. 专利题名[P]. 专利国别: 专利号, 发布日期.

[11] 姜锡洲. 一种温热外敷药制备方案[P]. 中国专利: 881056078, 1983-08-12.

国际、国家标准

[序号]标准代号. 标准名称[S]. 出版地: 出版者, 出版年.

[12] GB/T 16159—1996. 汉语拼音正词法基本规则[S]. 北京: 中国标准出版社, 1996.

报纸文章

[序号]主要责任者. 文献题名[N]. 报纸名, 出版年, 月(日): 版次.

[13] 谢希德. 创造学习的思路[N]. 人民日报, 1998, 12(25): 10.

电子文献

[序号]主要责任者. 电子文献题名[文献类型/载体类型]. 电子文献的出版或可获得地址(电子文献地址用文字表述), 发表或更新日期/引用日期(任选).

[14] 姚伯元. 毕业设计(论文)规范化管理与培养学生综合素质[EB/OL]. 中国高等教育网教学研究, 2005-2-2.

关于参考文献的未尽事项可参考国家标准《信息与文献参考文献著录规则》（GB/T 7714—2015）

附 录

附录相关内容...

附录是毕业设计（论文）主体的补充项目，为了体现整篇文章的完整性，写入正文又可能有损于论文的条理性、逻辑性和精炼性，这些材料可以写入附录段，但对于每一篇文章并不是必须的。附录依次用大写正体英文字母 A、B、C……编序号，如附录 A、附录 B。阅后删除此段。

附录正文样式与文章正文相同：宋体、小四；行距：22 磅；间距段前段后均为 0 行。阅后删除此段。

致 谢

值此论文完成之际，首先向我的导师……

致谢正文样式与文章正文相同：宋体、小四；行距：22 磅；间距段前段后均为 0 行。阅后删除此段。