

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.....	1
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.....	2
第 2 章 用 rust 写 rcore 操作系统的准备工作.....	3
2.1 配置实验环境.....	3
第 3 章 rcore 操作系统模块化的实现与改进.....	5
3.1 将模块进一步分隔.....	5
3.2 console 模块.....	6
3.2.1 对 console 模块的分析.....	6
3.2.2 对 console 模块的用户态单元测试.....	7
3.3 linker 模块.....	8
3.3.1 对 linker 模块的分析.....	8
3.3.2 对 linker 模块的用户态单元测试.....	9
3.4 kernel-context 模块.....	9
3.4.1 对 kernel-context 模块的分析.....	9
3.4.2 对 kernel-context 模块的用户态单元测试.....	10
3.5 kernel-alloc 模块.....	11
3.5.1 对 kernel-alloc 模块的分析.....	11
3.5.2 对 kernel-alloc 模块的用户态单元测试.....	12
3.6 kernel-vm 模块.....	12
3.6.1 对 kernel-vm 模块的分析.....	12
3.6.2 对 kernel-vm 模块的用户态单元测试.....	13
3.7 syscall 模块.....	14
3.7.1 对 syscall 模块的分析.....	14
3.8 task-manage 模块.....	15

3.8.1 对 task-manage 模块的分析.....	15
3.8.2 对 task-manage 模块的用户态单元测试.....	17
3.9 easy-fs 模块.....	18
3.9.1 对 easy-fs 模块的分析.....	18
3.9.2 对 easy-fs 模块的用户态单元测试.....	19
3.10 signal-defs、signal 和 signal-imple 模块.....	20
3.10.1 对 signal-defs、signal、signal-imple 模块的分析.....	20
3.10.2 对 signal-imple 模块的用户态单元测试.....	21
3.11 sync 模块.....	22
3.11.1 对 sync 模块的分析.....	22
3.11.2 对 sync 模块的用户态单元测试.....	23
3.12 在工作中遇到的问题和解决方法.....	24
3.12.1 编译 task-manage 模块的时候报错.....	24
3.12.2 运行时出错.....	24
3.12.3 测试时的错误思路.....	25
结 论.....	26
参考文献.....	27
附 录.....	28

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` 的同学，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。

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，所以最好设置科学上网代理，安装过程中全部默认选项即可。安装完成后，我们可以重新打开一个终端来让新设置的环境变量生效，最后我们需要确认一下是否正确安装了 Rust 工具链：`rustc -version`。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。

接着我们需要下载并安装 Qemu 模拟器。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

安装完成后，可以重新打开一个新的终端，输入 `qemu-system-riscv64 --version` 和 `qemu-riscv64 -version` 来确认 Qemu 的版本。

在完成所有的实验环境的配置之后，我们可以通过试着运行 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，然后通过 Visual Studio Code 打开这个仓库，然后我们先运行不需要处理用户代码的裸机操作系统 `os1`：`cd os1 LOG=DEBUG make run`，如果运行的结果与下图相同，则说明环境配置是正确的。

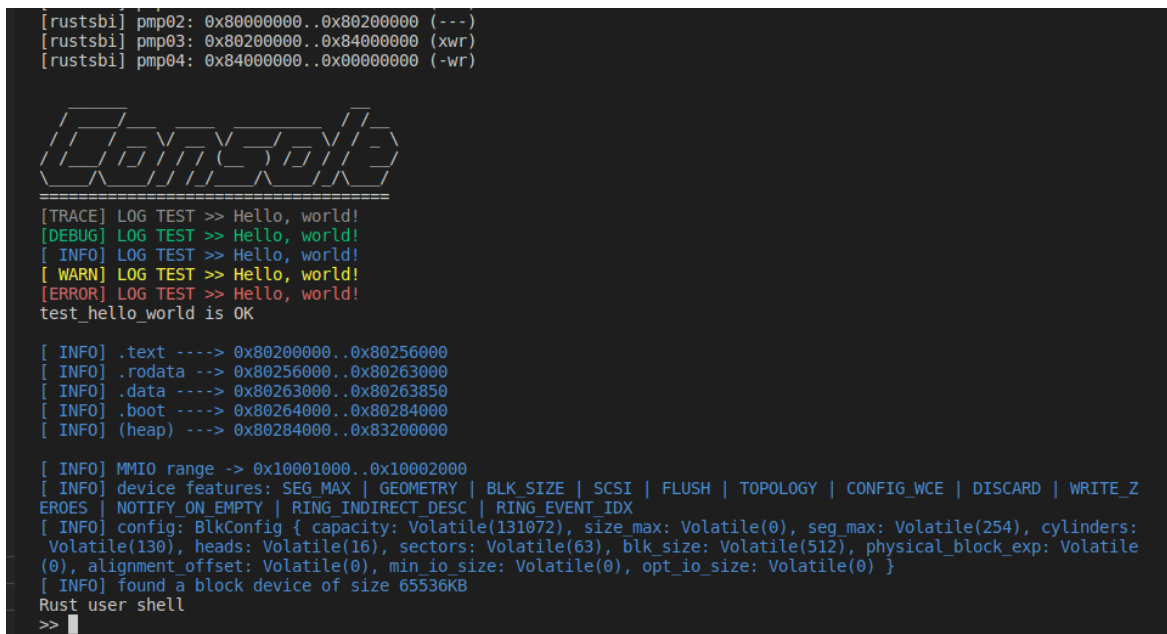
图 2-2 环境配置正确时预期的结果

3.1 将模块进一步分隔

同时，因为是经过慎重考虑才将整个操作系统内核按照功能拆分为了不同的模块，所以它的接口是不会轻易发生变化的，之后的同学们就可以在考虑接口之后就能够专注于模块内部的实现了，不必在考虑其他的模块中的工作。就比如在模块化之前，做到了同步互斥的部分，因为之前只是实现了进程的抽象，没有实现线程的部分，所以要做同步互斥的部分还要修改之前进程的部分，之前进程是程序的基本执行实体，在有了线程后，对进程的定义就要进行调整了，进程是线程的资源容

器，线程成为了程序的基本执行实体。但是在模块化之后，task-manage 模块同时定义了进程和线程，只是分别在 proc 的 feature 下和 thread 的 feature 下，在不需要线程的操作系统中，调用 task-manage 模块时确定 features = proc 就可以了，在需要线程的操作系统中，只需要将 task-manage 的 features 改为 features = thread 就可以了。

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，可以在终端里输入需要运行的程序，如果不知道程序名称，可以随便输入一些字符，如果错误，终端会告诉你正确的程序名称，然后就可以输入正确的程序名称了。



```

[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

3.2.1 对 console 模块的分析

console 模块的功能是可以在移除标准库依赖的情况下实现了 print 宏和 println 宏，可以向控制台输出自己想要输出的内容；并且实现了日志功能，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` 的依赖主要是为了使用其提供的 `Once` 结构，其功能是创建一个迭代器，但是这个迭代器只生成一次元素。对 `log` 的依赖主要是为了实现 `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，设置全局记录器，并且在调用 `set_logger` 完成之前发生的任何日志事件都将被忽略。

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

3.2.2 对 `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，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

3.3.1 对 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。

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"。

linker 模块没有需要的依赖，对外提供的接口有 boot0 宏，KernelLayout 结构，AppMeta 结构和 KernelRegionIterator 结构。boot0 宏的功能是定义内核入口，即设置一个启动栈，并在启动栈上调用高级语言入口。KernelLayout 结构的功能是定位、保存和访问内核内存布局；KernelRegion 结构：内核内存分区；KernelRegionIterator 结构：内核内存分区迭代器；AppMeta 结构是应用程序元数据。

KernelLayout 的结构有 6 个方法，分别为 pub fn locate()、pub const fn start(&self)、pub const fn end(&self)、pub const fn len(&self)、pub unsafe fn zero_bss(&self)、pub fn iter(&self) 这些方法的作用分别为定位内核布局、得到内核起始地址、得到内核结尾地址、得到内核静态二进制长度、清零 .bss 段、得到内核区段迭代器。

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。该结构的含义是内核内存分区，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。

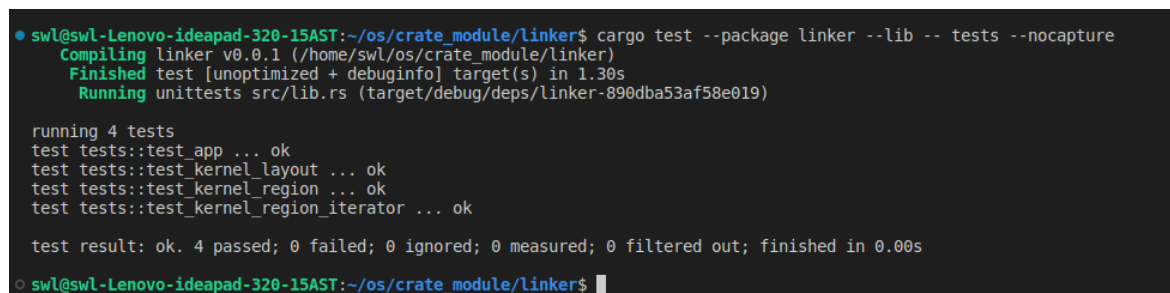
KernelRegionIterator 结构有两个成员，一个成员是 KernelLayout 类型的引用 layout，一个成员是 Option<KernelRegionTitle>类型的 next。该结构的含义是内核内存分区迭代器。该结构存在 next 方法，该方法的作用是得到迭代器中下一位的值。

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，通过枚举 KernelRegionTitle 来进行区分；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。

linker 模块还有一个子模块 app，app 子模块中有两个结构，分别为 AppMeta：应用程序元数据和 AppIterator：应用程序迭代器。AppMeta 结构有 2 个方法，分别为：pub fn locate()、pub fn iter(&'static self)这两个方法的作用是定位应用程序和遍历链接进来的应用程序。AppIterator 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。

3.3.2 对 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。然后设置一个非零初始化的 KernelLayout，之后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。



```
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

3.4.1 对 kernel-context 模块的分析

应用程序出错是在所难免的，但是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，所以为了保护操作系统不受应用程序的出错的破坏，我们引入了特

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，主要是为了使用 `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、**PortalText**：传送门代码、**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`，它们的含义分别是是否以特权态切换和线程中断是否开启。这个结构包含 14 个方法，分别为：`empty()` 方法、`user(pc: usize)` 方法、`thread(pc: usize, interrupt: bool)` 方法、这三个方法的作用是创建空白上下文、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、初始化指定入口的内核上下文。`x(&self, n: usize)`、`a(&self, n: usize)`、`ra(&self)`、`sp(&self)`、`pc(&self)` 这些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`。`x_mut(&mut self, n: usize)`、`a_mut(&mut self, n: usize)`、`sp_mut(&mut self)`、`pc_mut(&mut self)` 这些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`。`move_next(&mut self)` 该方法的作用是将 `pc` 移至下一条指令。`execute(&mut self)` 该方法的作用是执行此线程，并返回 `sstatus`，将修改 `sscratch`、`sepc`、`sstatus` 和 `stvec`。

`kernel-context` 模块还有一个子模块 `foreign`，在这个子模块里有 `PortalCache` 结构体，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 `init(&mut self, satp: usize, pc: usize, a0: usize, supervisor: bool, interrupt: bool)` 方法和 `address(&mut self)` 方法，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(&mut self, portal: &mut impl ForeignPortal, key: impl SlotKey)` 该方法的作用是执行异界线程。

`PortalText` 结构体的作用是传送门代码。该结构一共有 3 种方法，分别是：`new()`、`aligned_size(&self)`、`copy_to(&self, address: usize)`。`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)` 方法和 `init_transit(transit: usize, slots: usize)` 方法，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。

3.4.2 对 `kernel-context` 模块的用户态单元测试

`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的两个成员 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

3.5.1 对 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 结构体并且需要在 Cargo.toml 中对 customizable_buddy = 0.03 进行依赖，然后调用 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 *mut T 但是非零且协变，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 模块有恋歌对外接口，分别为 init(base_address: usize) 和 transfer(region: &'static mut [u8])。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。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。

3.5.2 对 kernel-alloc 模块的用户态单元测试

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；然后调用初始化函数 init()，并且将结果和预期结果进行比较。

```
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

3.6.1 对 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, 而想要满足这个特征就需要能够调用这个特征的所有方法, 即 new_root() -> Self、root_ptr(&self) -> NonNull<Pte<Meta>>、p_to_v<T>(&self, ppn: PPN<Meta>) -> NonNull<T>、v_to_p<T>(&self, ptr : NonNull<T>)->PPN<Meta>、check_owned(&self,pte:Pte<Meta>)->bool、allocate(&mut self,len:usize,flags:&mut VmFlags<Meta>)->NonNull<u8>、deallocate(&mut self, pte: Pte<Meta>, len: usize) -> usize、drop_root(&mut self);这些方法的功能分别为新建根页表页、获取根页表、计算当前地址空间上指向物理页的指针、计算当前地址空间上的指针指向的物理页、检查是否拥有一个页的所有权、为地址空间分配 len 个物理页、从地址空间释放 pte 指示的 len 个物理页、释放根页表。并且这个特征还提供了方法 root_ppn(&self) -> PPN<Meta> 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 结构共有 7 个方法, 分别为: new() -> Self、root_ppn(&self) -> PPN<Meta>、root(&self) -> PageTable<Meta>、map_extern(&mut self,range: Range<VPN<Meta>>, pbase: PPN<Meta>,flags: VmFlags<Meta>)、map(&mut self,range: Range<VPN<Meta>>,data: &[u8],offset: usize,flags: VmFlags<Meta>)、translate<T>(&self,addr: VAddr<Meta>,flags: VmFlags<Meta>) -> Option<NonNull<T>>、cloneself(&self, new_addrspace: &mut AddressSpace<Meta, M>)这些方法的作用分别是创建新地址空间、得到地址空间根页表的物理页号、得到地址空间根页表、向地址空间增加映射关系、分配新的物理页, 拷贝数据并建立映射、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、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。

3.6.2 对 kernel-vm 模块的用户态单元测试

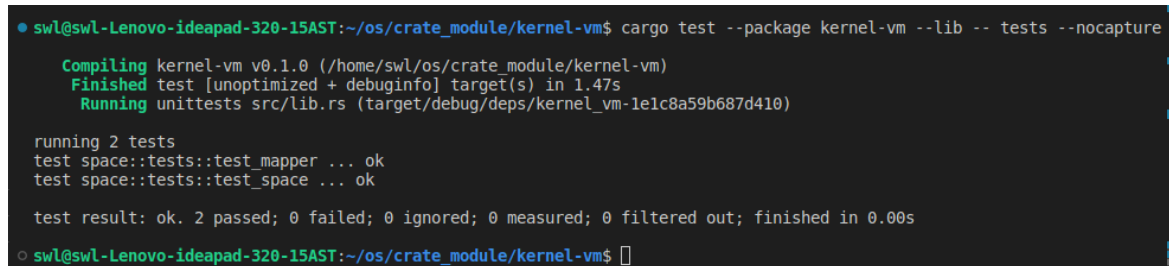
因为测试的函数不全部是对外接口, 所以 kernel-vm 模块的测试代码没有放在 tests 文件夹或者 lib.rs 中, 而是放在子模块 space 的 mod.rs 中。AddressSpace 结构体的参数 Meta 需要满足 VmMeta 特征, M 参数需要满足 PageManager<Meta>特征, 所以在测试 AddressSpace 结构体之前, 需要定义实现了 VmMeta 特征的 SV39 结构和实现了 PageManager<Meta>特征的 Sv39Manager 结构体。VmMeta 特征其实就是 'static + MmuMeta + Copy + Ord + 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, 然后为 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。

接下来根据 SV39 结构体定义一个 Sv39Manager(NonNull<Pte<Sv39>>)结构体, 然后依次实现 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。然后依次运行 AddressSpace 的几个方法，并和预期值进行比较。

最后，可以在终端输入 cargo test --package kernel-vm --lib -- tests --nocapture 来查看测试的结果。



```

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

3.7.1 对 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"。并且需要在 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。因为找不到标准文档, 所以系统调用号从 Musl Libc for RISC-V 源码生成。TimeSpec 结构体有两个成员: tv_sec 和 tv_nsec, 都是 usize 类型, 分别代表了秒和纳秒, 该结构体下还定义了几个常数 ZERO、SECOND、MILLSECOND、MICROSECOND、NANOSECOND 分别代表 0、1 秒、1 毫秒、1 微秒、1 纳秒, 还有 from_millisecond() 函数, 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

clock_gettime()、close()、condvar_create()、condvar_signal()、condvar_wait()、exec()、exit()、fork()、getpid()、gettid()、kill()、mutex_create()、mutex_lock()、mutex_unlock()、open()、read()、sched_yield()、semaphore_create()、semaphore_down()、semaphore_up()、sigaction()、sigprocmask()、sigreturn()、thread_create()、wait()、waitpid()、waittid()、write()等对外接口。

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 以及

handle()、init_clock()、init_io()、init_memory()、init_process()、init_scheduling()、init_signal()、init_sync_mutex()、init_thread()等函数。

3.8 task-manage 模块

3.8.1 对 task-manage 模块的分析

我们将开发一个用户终端(Terminal)或命令行(俗称 Shell)，形成用户与操作系统进行交互的命令行界面，为此，我们要对任务建立新的抽象：进程，并实现若干基于进程的強大系统调用。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。

线程是进程的组成部分，进程可包含 1 - n 个线程，属于同一个进程的线程共享进程的资源，比如地址空间、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（Scheduling）和分派（Dispatch）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，但是使用了 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。ProcId 结构体代表进程 id，有 new(), from_usize(), get_usize() 三个方法，他们的功能分别是创建了一个进程编号自增的进程 id 类型，根据输入的 usize 类型参数可以获得一个以参数为 id 的 ProcId，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，所以需要三个方法：insert(&mut self, id: I, item: T)、delete(&mut self, id: I)、get_mut(&mut self, id: I) -> Option<&mut T>，他们的功能分别是插入 item，删除 item 和获取可变的

item。Schedule 特征的功能是任务调度，在队列中保存需要调度的任务 Id，Schedule 特征需要实现方法：add(&mut self, id: I)、fetch(&mut self) -> Option<I>，他们的功能分别是任务进入调度队列、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 parent: ProcId、children: Vec<ProcId>、dead_children: Vec<(ProcId, isize)> 分别代表：父进程 id，子进程列表和已经结束的进程列表。ProcRel 结构有 new(), 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，该方法的作用是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。

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。

PManager 结构体的方法共有 9 个，分别为：new() -> Self 此方法用于新建 PManager、find_next(&mut self) -> Option<&mut P> 此方法用于找到下一个进程、set_manager(&mut self, manager: MP) 此方法用于设置 manager、make_current_suspend(&mut self) 此方法用于阻塞当前进程、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、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、current(&mut self) -> Option<&mut P> 此方法用于获取当前进程、get_task(&mut self, id: ProcId) -> Option<&mut P> 此方法用于获取某个进程、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)。

当 features = thread 时，只是在进程中加入了线程，ProcThreadRel：进程、子进程以及它地址空间内的线程之间的关系，PThreadManager：管理进程、子进程以及它地址空间内的线程之间的关系，他们的方法与对应的进程的结构的方法就基

本相同，在这里就不详细展开了。

3.8.2 对 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`，没有 `feature` 的时候调用 `id,manage` 和 `Schedule`；了解了这些就可以开始进行 task-manage 模块的测试了。

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，然后比较第一次 `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，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

3.9.1 对 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，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。

easy-fs 模块的对外的接口有

EasyFileSystem、FileHandle、Inode、OpenFlags、UserBuffer 这五个结构体，他们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。还有两个特征：BlockDevice、FSManager，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；以及一个常数 BLOCK_SZ = 512，使用 512 字节的块大小。

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 特征需要满足五个方法：open(&self, path: &str, flags: OpenFlags) -> Option<Arc<FileHandle>>、find(&self, path: &str) -> Option<Arc<Inode>>、link(&self, src: &str, dst: &str) -> isize、unlink(&self, path: &str) -> isize、readdir(&self, path: &str) -> Option<Vec<String>>，这五个方法的功能分别为打开文件、查找文件、创建到源文件的硬链接、删除硬链接、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有 create(block_device: Arc<dyn BlockDevice>, total_blocks: u32, inode_bitmap_blocks: u32) -> Arc<Mutex<Self>>、open(block_device: Arc<dyn BlockDevice>) -> Arc<Mutex<Self>>、root_inode(efs: &Arc<Mutex<Self>>) -> Inode、

`get_disk_inode_pos(&self, inode_id: u32) -> (u32, usize)`、`get_data_block_id(&self, data_block_id: u32) -> u32`、`alloc_inode(&mut self) -> u32`、`alloc_data(&mut self) -> u32`、`dealloc_data(&mut self, block_id: u32)`，这些函数的功能分别为从块设备创建文件系统、将块设备作为文件系统打开、获取文件系统的根索引节点、通过 id 获取索引节点、通过 id 获取数据块、分配新索引节点、分配一个数据块、释放一个数据块。

`FileHandle` 结构体表示内存中的缓存文件的元数据，它的公有成员有：`inode: Option<Arc<Inode>>`、`read: bool`、`write: bool`、`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、当前偏移量。它的对外接口函数有 `new(read: bool, write: bool, inode: Arc<Inode>) -> Self`、`empty(read: bool, write: bool) -> Self`、`readable(&self) -> bool`、`writable(&self) -> bool`、`read(&mut self, buf: UserBuffer) -> isize`、`write(&mut self, buf: UserBuffer) -> isize`。这些函数的功能分别是创建一个新的 `FileHandle`，创建一个空的 `FileHandle`，判断 `FileHandle` 是否能读，判断 `FileHandle` 是否能写，从缓冲区读数据，向缓冲区写数据。

`Inode` 结构体代表 `easy-fs` 上的虚拟文件系统层，他的对外接口函数有：`new(block_id: u32, block_offset: usize, fs: Arc<Mutex<EasyFileSystem>>, block_device: Arc<dyn BlockDevice>) -> Self`、`find(&self, name: &str) -> Option<Arc<Inode>>`、`create(&self, name: &str) -> Option<Arc<Inode>>`、`readdir(&self) -> Vec<String>`、`read_at(&self, offset: usize, buf: &mut [u8]) -> usize`、`write_at(&self, offset: usize, buf: &[u8]) -> usize`、`clear(&self)`，这些函数的功能分别是创建一个 `vfs` 索引节点、按名称查找当前索引节点下的索引节点、通过名称创建当前索引节点下的索引节点、列出当前索引节点下的索引节点、从当前索引节点读取数据、对当前索引节点写数据、清除当前索引节点中的数据。

`OpenFlags` 结构体是打开文件的标志，定义了一些常数

`RONLY`、`WRONLY`、`RDWR`、`CREATE`、`TRUNC`，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。它还有一个对外接口函数 `read_write(&self) -> (bool, bool)`，返回（可读、可写）。

`UserBuffer` 结构体代表用户与 `os` 通信的缓冲区，它的公有成员有一个：`buffers: Vec<&'static mut [u8]>`，类型为一个 `u8` 切片数组；它的对外接口函数有 `new(buffers: Vec<&'static mut [u8]>) -> Self`、`len(&self) -> usize`，它们的功能分别是通过参数创建一个用户缓冲区、得到用户缓冲区的长度。

3.9.2 对 easy-fs 模块的用户态单元测试

3.10 signal-defs、signal 和 signal-impl 模块

3.10.1 对 signal-defs、signal、signal-impl 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，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。`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；自动实现了 `RefUnwindSafe`、`Send`、`Sync`、`Unpin`、`UnwindSafe`；`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 `SignalResult` 枚举：信号处理函数返回得到的结果，`SignalResult` 的变体有 `NoSignal`, `IsHandlingSignal`, `Ignored`, `Handled`, `ProcessKilled(i32)`, `ProcessSuspended`, 它们分别表示没有信号需要处理、目前正在处理信号，因而无法接受其他信号、已经处理了一个信号，接下来正常返回用户态即可、已经处理了一个信号，并修改了用户上下文、需要结束当前进程，并给出退出时向父进程返回的 `errno`、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：一个信号模块需要对外暴露的接口。想要实现这个 trait 需要满足一些函数：

`from_fork(&mut self) -> Box<dyn Signal>`; 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。

`clear(&mut self)`; `sys_exec` 会使用，但是 `sys_exec` 不会继承信号处理函数和掩码。

`add_signal(&mut self, signal: SignalNo)`; 添加一个信号

`is_handling_signal(&self) -> bool`; 是否当前正在处理信号

`set_action(&mut self, signal: SignalNo, action: &SignalAction) -> bool`; 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 `sigaction` 中返回 `EINVAL`）

`get_action_ref(&self, signal: SignalNo) -> Option<SignalAction>`; 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 `sigaction` 中返回 `EINVAL`）

`update_mask(&mut self, mask: usize) -> usize`; 设置信号掩码，并获取旧的信号掩

码，`sys_procmask` 会使用。

`handle_signals(&mut self, current_context: &mut LocalContext) -> SignalResult`; 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。

`sig_return(&mut self, current_context: &mut LocalContext) -> bool`; 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` 结构体的公有成员有 `received: SignalSet`，`mask: SignalSet`，`handling: Option<HandlingSignal>`，`actions: [Option<SignalAction>; 32]`，它们分别表示已收到的信号、屏蔽的信号掩码、在信号处理函数中，保存之前的用户栈、当前任务的信号处理函数集。`SignalImpl` 结构体实现 `signal` 模块的 `Signal` 特征并且对外暴露了一个 `new()` 方法用来创建一个新的信号管理器。`HandlingSignal` 枚举表示正在处理的信号，有两个变体：`Frozen`、`UserSignal(LocalContext)`，分别代表内核信息，需要暂停当前进程、用户信息，需要保存当前的用户栈。

3.10.2 对 `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-imple$ cargo test --features user --package signal-impl --lib
-- tests --nocapture
Compiling signal-impl v0.1.0 (/home/swl/os/crate_module/signal-imple)
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-imple$

```

图 3-7 signal-imple 模块单元测试结果

3.11 sync 模块

3.11.1 对 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也不会有一致性问题。但是，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**、**IntrMaskingInfo** 结构体，它们分别代表条件变量、互斥阻塞、信号、具有动态检查借用规则的可变内存位置，允许我们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 结构体的接口函数有 4 个，分别为：

new() 该方法的作用是创建一个新的条件变量结构。

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。

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。

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。

MutexBlocking 结构实现了互斥。该结构的接口函数有 **new()**，该方法的作用是创建一个新的 **MutexBlocking** 结构。该结构还满足了 **Mutex** 特征，即实现了 **lock(&self, tid: ThreadId)** 函数，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。

unlock(&self) 函数，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 结构有三个接口函数，分别是：`new(res_count: usize)`、`up(&self)`、`down(&self, tid: ThreadId)` 这些方法的作用分别是创建一个新的 Semaphore 结构、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、当前线程试图获取信号量表示的资源，并返回结果。

UPSafeCellRaw 结构体有两个接口：`new(value: T)`、`get_mut(&self)`，他们的功能分别是：创建一个新的 UPSafeCellRaw 结构、获得该结构内部成员的可变引用。

IntrMaskingInfo 有三个接口函数 `new()`、`enter(&mut self)`、`exit(&mut self)`。它们为 UPIntrFreeCell 的接口函数的实现提供了基础。

UPIntrFreeCell 有三个接口函数 `new(value: T)`、`exclusive_access(&self)`、`exclusive_session<F, V>(&self, f: F)`。创建新的结构、独占访问，如果数据已被借用，会 panic、独占会话

UPIntrRefMut 实现了三个 trait：`Deref`、`DerefMut`、`Drop` 需要的函数 `deref(&self)`、`deref_mut(&mut self)`、`drop(&mut self)`，取消引用该值、可变地取消引用该值、将数据删除。

3.11.2 对 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，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 `id: tid2`。

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

```

● swl@swl-Lenovo-ideapad-320-15AST:~/os/crate_module/sync$ cargo test --features user --package sync --lib -- --exact --nocapture
    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 3-8 sync 模块单元测试结果

3.12 在工作中遇到的问题解决方法

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，如果没有安装，则需要安装。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 override 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

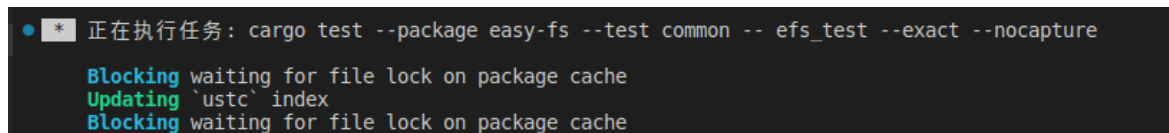


图 3-9 多个程序占用错误

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。

图 3-10 console 运行错误

3.12.3 测试时的错误思路

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

- [1]杨德睿 单核环境的模块化 Rust 语言参考实现
- [2]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[OL]. 中研网, 2021-10-20.
- [3]洛佳. 使用 Rust 编写操作系统（四）：内核测试[OL]. 知乎, 2019-11-07.
- [4]Open-Source-OS-Training-Camp-2022 文档[OL]
- [5]孙卫真 基于 RISC-V 的计算机系统综合实验设计[D]. 北京：首都师范大学, 2021-04-16.

附 录

如何进行测试，以 task-manage 模块为例进行说明。运行所有的测试，需要 features = “proc thread”：

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

--features feature_name，表示的是这个模块中有部分代码使用了 features 属性，在测试是是否编译并测试在这个 features 下的代码。

--package package_name，表示需要运行 package 里的测试代码。

--lib – tests，是测试代码在这个 package 里的具体位置。

--nocapture，默认情况下，测试工具隐藏或捕获测试函数中的 print 语句而不发，以使测试结果更整洁，并且只显示测试工具的输出。如果我们想在测试中查看打印语句，就需要使用--nocapture。