

rCore Tutorial Refactoring

计 75 班 涂轶翔

计 75 班 赵成钢

2020.05.16, 作为最终报告

文档/代码地址

点开看看

- 代码仓库：
 - <https://github.com/os20-rCore-Tutorial/rCore-Tutorial>
 - <https://git.tsinghua.edu.cn/os20-rcore-tutorial/rcore-tutorial>
- 文档部署： <https://os20-rcore-tutorial.github.io/rCore-Tutorial-deploy>

成果概述

代码统计对比

指标	原框架	新框架
Warning 数量	32	0
cargo doc	不能生成	全部内容覆盖
Rust 行数	2178	1677
Rust 注释	21	696
Assembly 行数	185	134
Assembly 注释	2	52
unsafe 数量	57	23

成果概述

随便打开一个文件

```
lazy_static! {  
    /// 全局的 [`Processor`]  
    pub static ref PROCESSOR: UnsafeWrapper<Processor> = Default::default();  
}  
  
/// 线程调度和管理  
#[derive(Default)]  
pub struct Processor {  
    /// 当前正在执行的线程  
    current_thread: Option<Arc<Thread>>,  
    /// 线程调度器, 记录所有线程  
    scheduler: SchedulerImpl<Arc<Thread>>,  
}  
  
impl Processor {  
    /// 获取一个当前线程的 `Arc` 引用  
    pub fn current_thread(&self) -> Arc<Thread> { self.current_thread.as_ref().unwrap().clone() }  
  
    /// 第一次开始运行  
    ///  
    /// 从 `current_thread` 中取出 [`Context`], 然后直接调用 `interrupt.asm` 中的 `__restore`  
    /// 来从 `Context` 中继续执行该线程。  
    ///  
    /// 注意调用 `run()` 的线程会就此步入虚无, 不再被使用  
    pub fn run(&mut self) -> ! {  
        // interrupt.asm 中的标签  
        extern "C" {  
            fn __restore(context: usize);  
        }  
        // 从 current_thread 中取出 Context
```

```
1  # 操作系统启动时所需的指令以及字段  
2  #  
3  # 我们在 linker.ld 中将程序入口设置为了 _start, 因此在这里我们将填充这个标签  
4  # 它将会执行一些必要操作, 然后跳转至我们用 rust 编写的入口函数  
5  #  
6  # 关于 RISC-V 下的汇编语言, 可以参考 https://rv8.io/asm.html  
7  # %hi 表示取 [12,32) 位, %lo 表示取 [0,12) 位  
8  
9  .section .text.entry  
10 .globl _start  
11 # 目前 _start 的功能: 将预留的栈空间写入 $sp, 然后跳转至 rust_main  
12 _start:  
13     # 计算 boot_page_table 的物理页号  
14     lui t0, %hi(boot_page_table)  
15     li t1, 0xffffffff00000000  
16     sub t0, t0, t1  
17     srli t0, t0, 12  
18     # 8 << 60 是 satp 中使用 Sv39 模式的记号  
19     li t1, (8 << 60)  
20     or t0, t0, t1  
21     # 写入 satp 并更新 TLB  
22     csrw satp, t0  
23     sfence.vma  
24
```

没有注释会产生警告

成果概述

注释生成文档



Crate os

See all os's items

Modules

Macros

Functions

Crates

algorithm

bare_metal

bit_field

bitflags

bitvec

buddy_system_allocator

cfg_if

device_tree

either

lazy_static

log

os

radium

rcore_fs

rcore_fs_sfs

Crate **os**

[-]

全局属性

- `#![no_std]` 禁用标准库
- `#![no_main]` 不使用 `main` 函数等全部 Rust-level 入口点来作为程序入口
- `#![deny(missing_docs)]` 任何没有注释的地方都会产生警告：这个属性用来压榨写实验指导的学长，同学可以删掉

一些 **unstable** 的功能需要在 **crate** 层级声明后才可以使用

- `#![feature(alloc_error_handler)]` 我们使用了一个全局动态内存分配器，以实现原本标准库中的堆内存分配。求我们同时实现一个错误回调，这里我们直接 `panic`
- `#![feature(llvm_asm)]` 内嵌汇编
- `#![feature(global_asm)]` 内嵌整个汇编文件
- `#![feature(panic_info_message)]` `panic!` 时，获取其中的信息并打印
- `#![feature(naked_functions)]` 允许使用 `naked` 函数，即编译器不在函数前后添加出入栈操作。这允许我们在函数级汇编使用 `ret` 提前结束，而不会导致栈出现异常

Modules

console

实现控制台的字符输入和输出

drivers

驱动模块

fs

文件系统

interrupt

中断模块

memory

内存管理模块

panic

代替 `std` 库，实现 `panic` 和 `abort` 的功能

process

管理进程 / 线程

sbi

调用 `Machine` 层的操作

Macros

print

实现类似于标准库中的 `print!` 宏

print!

实现类似于标准库中的 `print!` 宏



Module mapping

Re-exports

Modules

os::memory

Modules

address

config

frame

heap

mapping

range

Functions

init



All crates

Click or press 'S' to search, '?' for more options...



Module **os::memory::mapping**

[-]

内存映射

每个线程保存一个 **Mapping**，其中记录了所有的字段 **Segment**。同时，也要追踪为页表或字段分配的所有物理页，目的是 `drop` 掉之后可以安全释放所有资源。

Re-exports

```
pub use mapping::Mapping;
pub use memory_set::MemorySet;
pub use page_table::PageTable;
pub use page_table::PageTableTracker;
pub use page_table_entry::Flags;
pub use page_table_entry::PageTableEntry;
pub use segment::MapType;
pub use segment::Segment;
```

Modules

mapping

Rv39 页表的构建 **Mapping**

memory_set

一个线程中关于内存空间的所有信息 **MemorySet**

page_table

单一页表页面（4K） **PageTable**，以及相应封装 **FrameTracker** 的 **PageTableTracker**

page_table_entry

页表项 **PageTableEntry**

segment

映射类型 **MapType** 和映射片段 **Segment**

成果概述

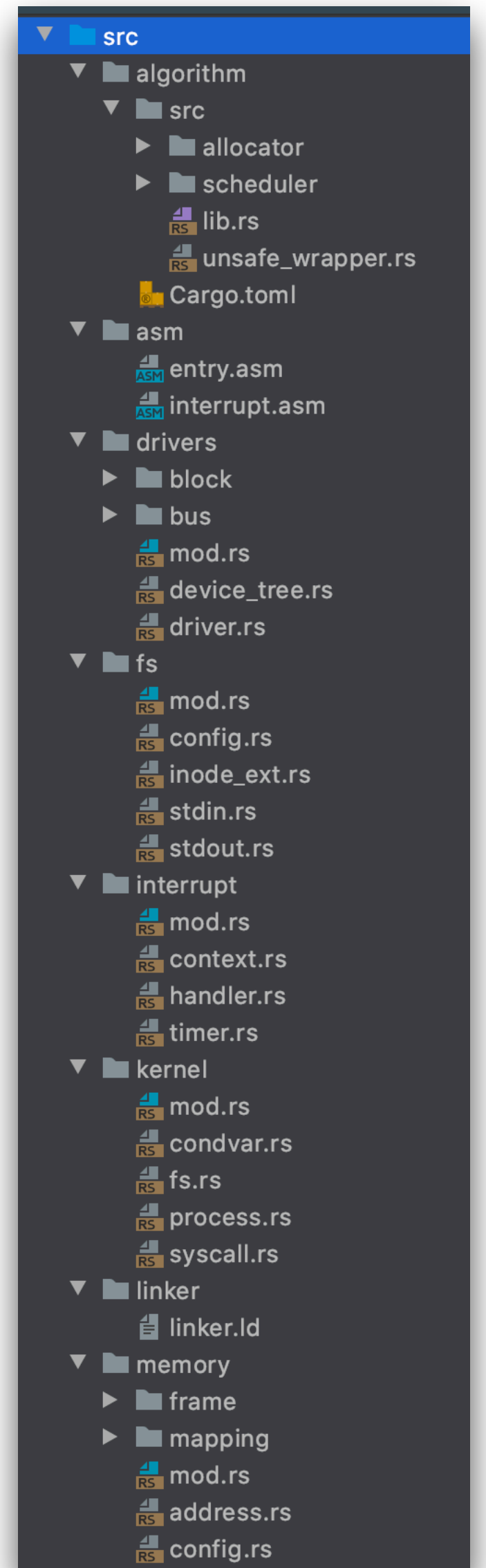
更加 Rust

- 更严格和安全的区分
 - 物理/虚拟 - 地址/页
- 更方便的接口设计和封装
 - 更多 Rust 语法糖运用
- 借助语言实现自动化内存管理
 - 资源自动销毁
- 更少的 unsafe
 - 加锁、引用、线程安全
 - 基本上只有 Raw 地址访问、内嵌汇编
- 随着近期 Rust 工具链更新

成果概述

结构设计

- 命名、文件组织和模块设计更加符合软件工程规范
- 相比原来精简很多
 - e.g. TrapFrame、Context、ContextContent 统一抽象为 Context
 - e.g. 不再有 Idle 线程
 - e.g. 页表重写，不再调库套很多壳



成果概述

功能只多不少

- 没有任何裁剪而且更加完善
 - 杀死线程 / 输入输出中文 等等
- 为物理页分配器和调度器抽象出了算法库
- 区分进程和线程
- 页表重新实现，不再调用库代码
- 增加了设备驱动，用户线程将不再和内核链接在一起，而是通过 QEMU 模拟的设备读取
- （可能）未来增加真机支持和多核

成果概述

文档内容

- 不造轮子
 - 很多知识性的内容从原来的教程中借鉴
 - 覆盖的知识面将会是原来的超集
 - 把原来文档评论区一些内容都整合进去了
- 一个章节对应一个 Lab
- 不再是练习题和章节分开
- （未来）补全结合课内练习补全 Lab 题目设计

成果概述

文档设计

- 设计上更加明确和精简
 - 明确目标是给做实验的同学们开发的
 - 精简很多不重要的实现部分
 - 专注于操作系统原理、架构设计和关键代码
- 文档内有 15 个思考题
 - 也是我们在开发时不断权衡设计、对操作系统的理解
 - 也结合了我们在做原来实验时的一些疑问
- 更加循序渐进：用到什么写什么
 - e.g. 中断处理部分的 `__restore` 不需要判断用户/内核态
 - e.g. 开始的 linker script 不需要对齐
 - 做实验只需要从对应版本 checkout 出来

思考

运行下面的代码：

```
os/src/main.rs

1  /// Rust 的入口函数
2  ///
3  /// 在 `_start` 为我们进行了一系列准备之后，这是第一个被调用的 Rust 函数
4  #[no_mangle]
5  pub extern "C" fn rust_main() -> ! {
6      // 初始化各种模块
7      interrupt::init();
8      memory::init();
9
10     // 物理页分配
11     match memory::frame::FRAME_ALLOCATOR.lock().alloc() {
12         Result::Ok(frame_tracker) => frame_tracker,
13         Result::Err(err) => panic!("{}", err)
14     };
15
16     loop{}
17 }
```

思考，和上面的代码有何不同，我们的设计是否存在一些语法上的设计缺陷？

Click to show

成果概述

文档规范

- 更加规范
 - 格式有比较严格的成文规定
- 做的非常细节
 - 代码路径标签
 - 思考题隐藏答案

Context

我们把在中断时保存了各种寄存器的组构体叫做 `Context`，他表示原来和 `main` 函数相关的寄存器（`main` 函数在 `main` 函数中保存了 `main` 函数的寄存器，这个概念在后面线程的部分还会用到），这里我们和 `scause` 以及 `stval` 一起保存 `Context` 而仅仅被看做一个临时的变量（在后面会被用到），`Context`

os/src/interrupt/context.rs

```
1 use riscv::register::{sstatus::Sstatus, scause::Scause};
2
3 #[repr(C)]
4 pub struct Context {
5     pub x: [usize; 32], // 32 个通用寄存
6     pub sstatus: Sstatus,
7     pub sepc: usize
8 }
```

这里我们使用了 rCore 中的库 `riscv` 封装的一些寄存器

os/Cargo.toml

```
1 [dependencies]
2 riscv = { git = "https://github.com/rcore-
```

思考

可以看到我们的设计中用了大量的锁结构，很多都是为了让 Rust 知道我们是安全的，而且大部分情况下我们仅仅会在中断发生的时候来使用这些逻辑，这意味着，只要内核线程里面不用，就不会发生死锁，但是真的是这样吗？即使我们不在内核中使用各种 `Processor` 和 `Thread` 等等的逻辑，仅仅完成一些简单的运算，真的没有死锁吗？

Click to show

```

- x86_64
- RISC-V 64
- 其他一些名词
- ABI
- GitHub
- virtio
- Rust 相关
- rustup
- cargo
- rustc
- 其他软件
- QEMU
- Homebrew

### 内容控制

- 我们的目标针对于「做实验的同学」，对想完整实现一个 rCore 的同学来说可能不太友好；
- 但是我们也相信，这部分想完整实现的同学也不会因为我们在文档中少了一些非常细节的诸如模块调用的内容就放弃，而且从头复制理解；
- 所以，在文档开发过程中，我们需要对清晰和全面做很多的权衡和考虑，需要省略掉大量语法层面而 OS 无关的代码来带来更多的可读性
- 所以，在文档中引用的代码，只需要写主体的函数，不需要把一系列调用、头部注释全部加入进去。在最后，可能会再利用代码折叠的方法

### 书写格式

- 在数字、英文、独立的标点或记号两侧的中文之间要加空格，如：
  - 安装 QEMU

```

名、变量名、行间输出、编译选项、路径和文件名需要使用反引号`记号`，而不是 `$$a_0$$`，这是为了和 ``sep`` 统一。反引号`记号`，并在两侧加入空格，如：

ary 'pkg-config' not found` 时

成果概述

文档结构

- 环境部署：原第零章（实验环境说明）
- 实验指导零：原第一章（独立可执行程序）+ 原第二章（最小化内核）
- 实验指导一：原第三章（中断）
- 实验指导二：原第四章（内存管理）
- 实验指导三：原第五章（内存虚拟化）
- 实验指导四：原第六章（内核线程）+ 原七章（线程调度）
- 实验指导五：新内容（设备驱动）+ 原九章（文件系统）
- 实验指导六：原第八章（用户进程）

具体设计

实验之前：环境部署

- 整合了原来评论区
- 多平台
- 安装
 - Rust 环境
 - 把原来第一章的 QEMU 配置移动到了这里

具体设计

实验指导零 - 目录

- 摘要
- 创建项目
- 移除标准库依赖
- 移除运行时环境依赖
- 编译为裸机目标
- 生成内核镜像
- 调整内存布局
- 重写程序入口点
- 使用 QEMU 运行
- SBI 接口封装
- 小结

具体设计

实验指导零 - 变化

- 合并了原来的第一二章
- 精简了通过链接器参数来编译的介绍
- 直接走编译为生成裸机目标的路线
- 原来的 linker script 是祖传的
 - 发现了一个潜在的 bug
 - .sbss 也要加，否则后面会被分配出去
 - 对齐还用不上，后面结合物理页才用的上，删去
 - .stack 段没有用（其实就是在 .bss 里面）

具体设计

实验指导一 - 目录

- 摘要
- 什么是中断
- RISC-V 中的中断
- 程序运行状态
- 状态的保存与恢复
- 进入中断处理流程
- 时钟中断
- 小结

具体设计

实验指导一 - 变化

- TrapFrame 改成 Context（运行状态）+ scause + stval
- Context 后面在线程那里可以继续用
 - TrapFrame、Context 和 ContextContent 精简为一个
- handle_interrupt 直接返回一个要运行的 Context，比较直观
- 部分设计参考了 zCore
- 状态恢复的时候不需要判断是用户态还是内核态
 - 在后面涉及和修改

```
/// 中断的处理入口
///
/// `interrupt.asm` 首先保存寄存器至 Context，其作为参数和 scause 以及 stval 一并传入此函数
/// 具体的中断类型需要根据 scause 来推断，然后分别处理
#[no_mangle]
pub fn handle_interrupt(context: &mut Context, scause: Scause, stval: usize) -> *mut Context {
    match scause.cause() {
        // 断点中断 (ebreak)
        Trap::Exception(Exception::Breakpoint) => breakpoint(context),
        // 时钟中断
        Trap::Interrupt(Interrupt::SupervisorTimer) => supervisor_timer(context),
        // 其他情况，终止当前线程
        _ => fault(_context: context, scause, stval),
    }
}
```

具体设计

实验指导二 - 目录

- 摘要
- 动态内存分配
- 物理内存探测
- 物理内存管理
- 小结

具体设计

实验指导二 - 变化

- 物理/虚拟 - 地址/页不再是 `usize`
 - 都有单独的类
 - 支持一些 `from` 的 Rust 特性
- 分配和回收使用 `FrameTracker`
 - 析构时自动释放
- 分配算法分离出来一个算法库
 - 在 OS 中只有 `trait`

```
/// 虚拟地址
#[derive(Copy, Clone, Debug, Default, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct VirtualAddress(pub usize);

/// 物理地址
#[derive(Copy, Clone, Debug, Default, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct PhysicalAddress(pub usize);

/// 虚拟页号
#[derive(Copy, Clone, Debug, Default, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct VirtualPageNumber(pub usize);

/// 物理页号
#[derive(Copy, Clone, Debug, Default, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct PhysicalPageNumber(pub usize);

// 以下是一大堆类型的相互转换、各种琐碎操作

/// 从指针转换为虚拟地址
impl<T> From<*const T> for VirtualAddress {
    fn from(ptr: *const T) -> VirtualAddress {
        VirtualAddress(ptr as usize)
    }
}

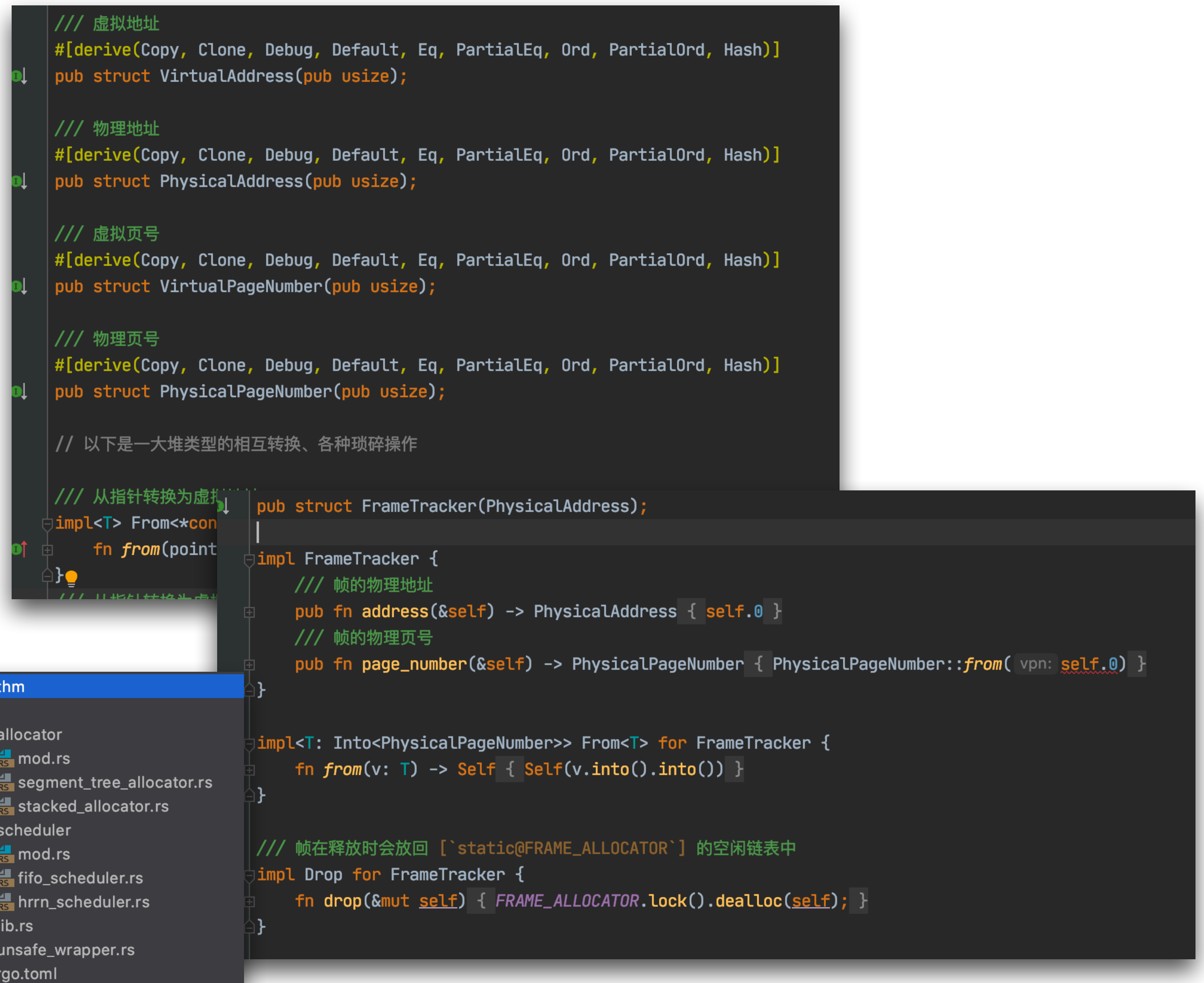
/// 从指针转换为物理地址
impl<T> From<*const T> for PhysicalAddress {
    fn from(ptr: *const T) -> PhysicalAddress {
        PhysicalAddress(ptr as usize)
    }
}

pub struct FrameTracker(PhysicalAddress);

impl FrameTracker {
    /// 帧的物理地址
    pub fn address(&self) -> PhysicalAddress { self.0 }
    /// 帧的物理页号
    pub fn page_number(&self) -> PhysicalPageNumber { PhysicalPageNumber::from(vpn: self.0) }
}

impl<T: Into<PhysicalPageNumber>> From<T> for FrameTracker {
    fn from(v: T) -> Self { Self(v.into().into()) }
}

/// 帧在释放时会放回 [`static@FRAME_ALLOCATOR`] 的空闲链表中
impl Drop for FrameTracker {
    fn drop(&mut self) { FRAME_ALLOCATOR.lock().dealloc(self); }
}
```



具体设计

实验指导三 - 目录

- 摘要
- 从虚拟地址到物理地址
- 修改内核
- 实现页表
- 实现内核重映射
- 小结

具体设计

实验指导三 - 变化

- 修正了启动线程的页表的一个 bug
- 重新实现了一个清爽的页表
 - 不再套奇怪的壳
- 页表分配回收使用 PageTableTracker
 - 自动回收
- 循序渐进：linker script 加入对齐并解释

```
/// 一个进程所有关于内存空间管理的信息
pub struct MemorySet {
    /// 维护页表和映射关系
    pub mapping: Mapping,
    /// 每个字段
    pub segments: Vec<Segment>,
    /// 所有分配的物理页面映射信息
    pub allocated_pairs: Vec<(VirtualPageNumber, FrameTracker)>,
}
```

```
#[derive(Default)]
/// 某个进程的内存映射关系
pub struct Mapping {
    /// 保存所有使用到的页表
    page_tables: Vec<PageTableTracker>,
    /// 根页表的物理页号
    root_ppn: PhysicalPageNumber,
}
```

```
/// 一个映射片段 (对应旧 tutorial 的 `MemoryArea`)
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub struct Segment {
    /// 映射类型
    pub map_type: MapType,
    /// 所映射的虚拟地址
    pub page_range: Range<VirtualPageNumber>,
    /// 权限标志
    pub flags: Flags,
}
```

具体设计

实验指导四 - 目录

- 摘要
- 线程和进程
- 线程的创建
- 线程的切换
- 内核栈
- 线程调度
- 小结

具体设计

实验指导四 - 变化

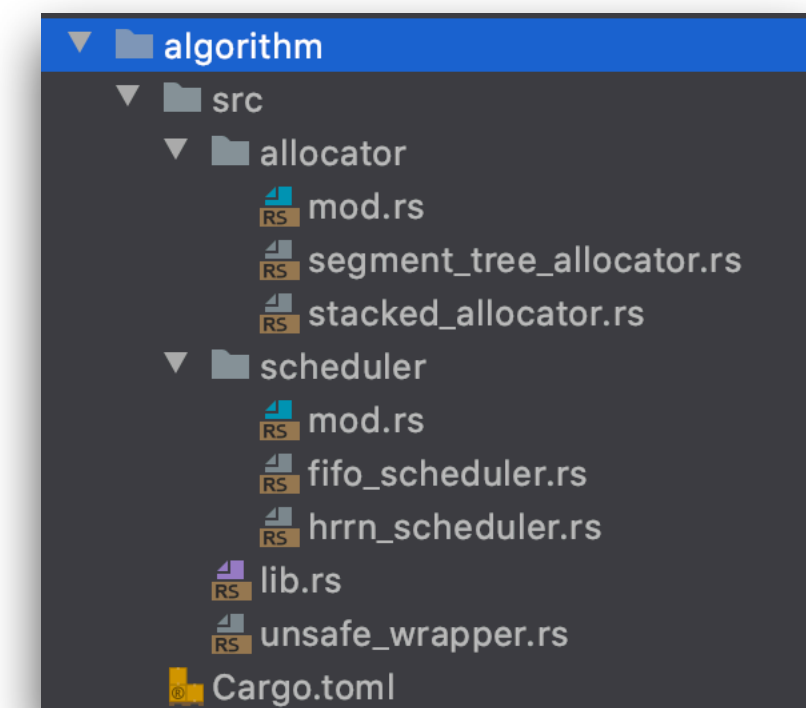
- 合并了原来的第六章（内核线程）和第七章（线程调度）
- 更加清爽
 - TrapFrame、Context 和 ContextContext 统一为 Context
 - 不再有 Idle 线程
 - 时钟中断直接 __restore 到一个新的 Context
- 区分进程和线程
- 线程的栈分配物理页
 - 不再是动态分配
- 用户线程共用一个内核栈（为了处理中断）
- 内核线程用自己的栈
- 调度算法分离到库中

```
/// 进程的信息
pub struct Process {
    /// 是否属于用户态
    pub is_user: bool,
    /// 进程中的线程公用页表 / 内存映射
    pub memory_set: MemorySet,
}
```

```
/// 线程的信息
pub struct Thread {
    /// 线程 ID
    pub id: ThreadID,
    /// 线程的栈
    pub stack: Range<VirtualAddress>,
    /// 线程执行上下文
    ///
    /// 当且仅当线程被暂停执行时, `context` 为 `Some`
    pub context: Mutex<Option<Context>>,
    /// 所属的进程
    pub process: Arc<RwLock<Process>>,
}
```

```
/// 处理时钟中断
fn supervisor_timer(context: &mut Context) -> *mut Context {
    timer::tick();
    PROCESSOR.get().tick(context)
}

/// 出现未能解决的异常, 终止当前线程
fn fault(_context: &mut Context, scause: Scause, stval: usize) -> *mut Context {
    println!(
        "{:?} terminated with {:?}",
        PROCESSOR.get().current_thread(),
        scause.cause()
    );
    println!("stval: {:x}", stval);
    PROCESSOR.get().kill_current_thread();
    // 跳转到 PROCESSOR 调度的下一个线程
    PROCESSOR.get().current_thread().run()
}
```



具体设计

实验指导五 - 目录

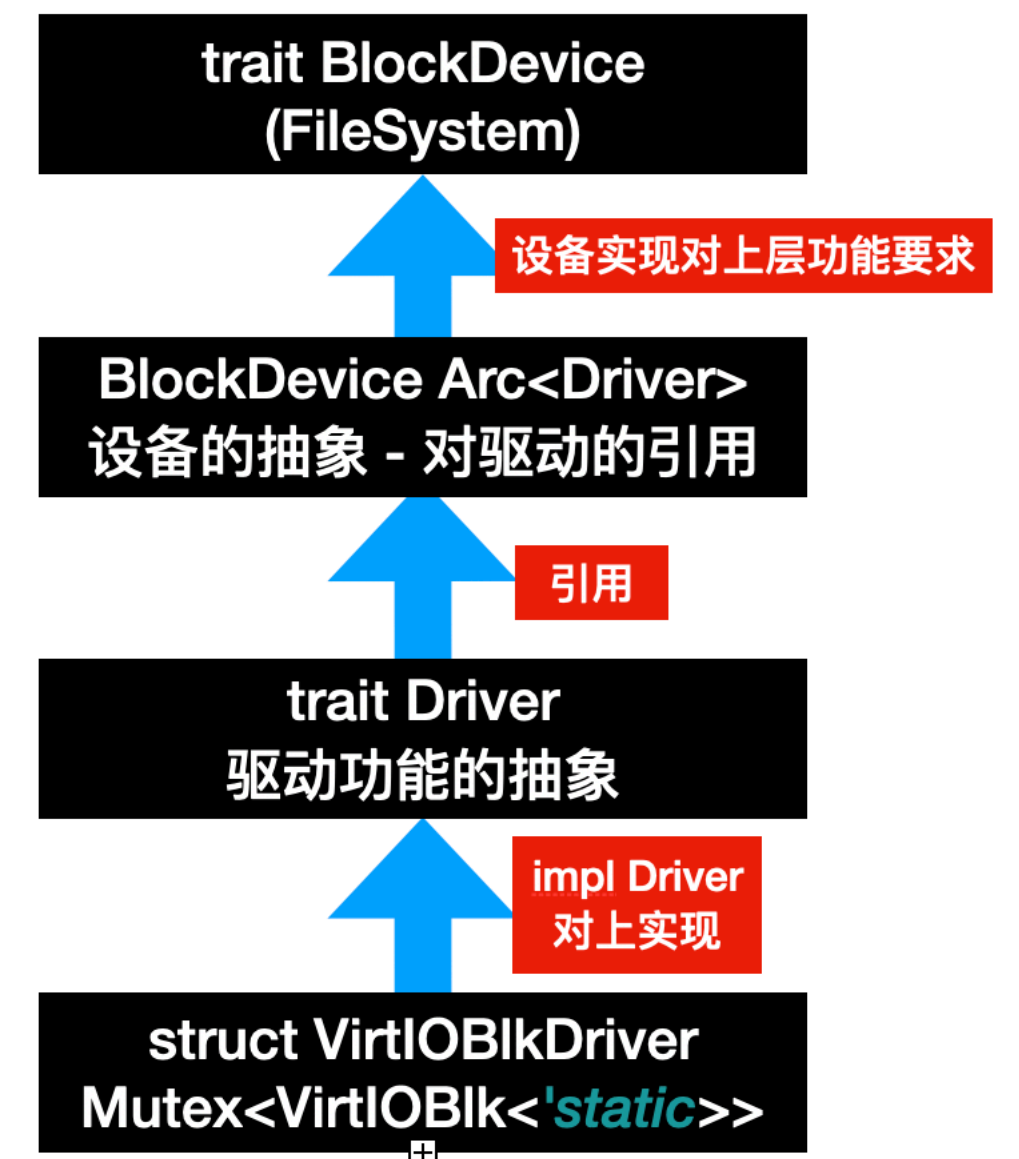
- 摘要
- 设备树
- virtio
- 驱动和块设备驱动
- 文件系统
- 小结

具体设计

实验指导五 - 变化

- 原来的第九章（文件系统）
- 用户态程序编译出的文件不再链接到内核中
- 挂载到 QEMU 虚拟的存储设备上
- 增加了真实的设备驱动
- 驱动设计参考了 rCore
- 文件系统不再挂载在内存上，挂在 virtio 块设备上

```
lazy_static! {  
    /// 根文件系统的根目录的 INode  
    pub static ref ROOT_INODE: Arc<dyn INode> = {  
        // 选择第一个块设备  
        for driver in DRIVERS.read().iter() {  
            if driver.device_type() == DeviceType::Block {  
                let device = BlockDevice(driver.clone());  
                // 动态分配一段内存空间作为设备 Cache  
                let device_with_cache = Arc::new(BlockCache::new(device, BLOCK_CACHE_CAPACITY));  
                return SimpleFileSystem::open(device_with_cache)  
                    .expect("failed to open SFS")  
                    .root_inode();  
            }  
        }  
        panic!("failed to load fs")  
    };  
}
```



具体设计

实验指导六 - 目录

- 摘要
- 构建用户程序框架
- 打包为磁盘镜像
- 解析 ELF 文件并创建线程
- 实现系统调用
- 条件变量
- 小结

具体设计

实验指导六 - 变化

- 原来的第八章（进程）
- stdin stdout 均统一使用文件接口
- 代码结构更有逻辑
- 输入输出均可一次处理多个字符，包括中文字符

```
<notebook>  
Hello world from user mode program!  
Thread 1 exit with code 0  
在记事本中  
书写中文  
Hello, rCore-Tutorial
```

感受

从零到一

- 收获非常大
- 三件事情
 - 做操作系统实验
 - 理解原理
 - 写操作系统
 - 理解实现
 - 写操作系统教程
 - 理解过程

致谢

感谢一起讨论、交流和帮忙的小伙伴

- 老师：向勇、陈渝
- 助教：陈嘉杰、王润基、刘丰源、吴一凡
- What's UR problem：刘润达、曹鼎原（另一个做 uCore 组的同学）
- 帮忙看文档的同学：陈海天、郑逢时

谢谢