杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

RongOS 设计与实现文档

项目成员：林杰克
莫佳洋

指导老师：章复嘉

2022.3 – 2022.8

目录

1 概述

RongOS 是使用 Rust 语言编写的基于 RISC-V64 的微型操作系统，实现了进程控制、内存管理、文件系统这三大操作系统基础模块，且支持在 QEMU 虚拟环境和 K210 硬件平台上运行。

1.1 比赛准备和调研

RongOS 为了避免重复“造轮子”，因此在开发前研究了许多高校的教学 OS，最后的抉择在于是基于 MIT 6.828 课程的 xv6 教学系统，还是基于清华大学的 rCore-Tutorial-v3，对于这一点我们做了很多思考与比较。

主要由以下三点原因最终决定了我们选择 rCore-Tutorial-v3 作为基石。

1) 编程语言的选择

xv6 采用的是传统的 C 语言，而 rCore-Tutorial-v3 则是使用新型的 Rust。由于 C 语言因为历史原因导致其在一些方面并没有很好的处理机制，如指针、内存泄漏、并发漏洞等，而 Rust 语言首先具有与 C 语言一样的硬件控制能力，其次它大大强化了安全编程和抽象编程能力。从某种角度上看，新出现的 Rust 语言的核心目标是解决 C 的短板，取代 C。所以用 Rust 写操作系统内核具有很好的开发和运行体验，这是 rCore-Tutorial-v3 采用 Rust 语言的原因。对于我们来说，出于好奇心与强烈的挑战欲望，我们更喜欢崭新的、带有独特机制的、以及目前被多家顶尖科技公司看好的 Rust 语言。

2) 社区支持

xv6-book 是一个 pdf 文件，最新更新时间为 2021 年 9 月 6 日。rCore-Tutorial-Book-v3 是以网站的形式部署在 Github Pages 上，相比之下具有较好的可读性，知识点的寻找也更为便捷。并且，rCore-Tutorial-v3（包括 rCore-Tutorial-Book-v3）的更新非常频繁，清华大学的老师与学生在不断地完善功能，修正错误。最为重要的是，rCore-Tutorial-Book-v3 在每一章节下方设定了讨论区，对于不理解的地方可以随时提出，且回复效率极高。通过密切地与其他学习者、开发者进行讨论交流，我们在项目开发途中解决问题的难度会大幅下降。

3) 自然语言环境

诚然，每个合格的计算机学子应该要具备阅读外文文献的能力。但我们对于自身的英语能力仍感到心有余而力不足。对于一个需要从头学到尾、时刻跟踪的项目来说，我们觉得采用母语的教程会大幅提高我们的学习与开发效率。

最后，非常感谢清华大学陈渝老师与吴一凡等同学开发编写的 rCore-Tutorial-v3！

1.2 需求分析

鉴于 2022 年全国大学生计算机系统能力大赛操作系统赛内核实现赛道的赛题要求，RongOS 最终支持了 26 个系统调用，具体如下：

```
const SYSCALL_GETCWD:   usize = 17;
const SYSCALL_DUP:      usize = 23;
const SYSCALL_DUP3:     usize = 24;
const SYSCALL_MKDIRAT:  usize = 34;
const SYSCALL_UNLINKAT: usize = 35;
const SYSCALL_UMOUNT2:  usize = 39;
const SYSCALL_MOUNT:    usize = 40;
const SYSCALL_CHDIR:    usize = 49;
const SYSCALL_OPENAT:   usize = 56;
const SYSCALL_CLOSE:    usize = 57;
const SYSCALL_PIPE:     usize = 59;
const SYSCALL_GETDENTS64: usize = 61;
const SYSCALL_READ:     usize = 63;
const SYSCALL_WRITE:    usize = 64;
const SYSCALL_FSTAT:    usize = 80;
const SYSCALL_EXIT:     usize = 93;
const SYSCALL_NANOSLEEP:usize = 101;
const SYSCALL_YIELD:    usize = 124;
const SYSCALL_KILL:     usize = 129;
const SYSCALL_TIMES:    usize = 153;
const SYSCALL_UNAME:    usize = 160;
const SYSCALL_GET_TIME: usize = 169;
const SYSCALL_GETPID:   usize = 172;
const SYSCALL_FORK:     usize = 220;
const SYSCALL_EXEC:     usize = 221;
const SYSCALL_WAITPID:  usize = 260;
```

但从项目完整性的角度出发，在支持大量的系统调用之外，我们还需要考虑如何正确地加载内核代码、如何做到内核与运行平台的无关性，这都需要我们在

机器级上做不小的努力。此外，由于需要支持文件系统，因此我们还需要实现持久化设备的数据读写（如 K210 上的 SD 卡驱动）等。

1.3 开发计划

由于我们最初对操作系统知识点的掌握大部分只停留在理论阶段，基本没有实践经验，并且 Rust 语言对我们来说也是完全陌生的，因此我们对项目总体的开发思路是稳中求进，先学习后创新。

具体的开发计划如表 1-1 所示：

表 1-1 开发计划表

起止时间	进度安排
2022.3.1 – 2022.4.10	学习 rCore-Tutorial-v3，同时学习 Rust 语言
2022.4.11 – 2022.4.30	莫佳洋：修改 rCore-Tutorial-v3、实现部分系统调用 林杰克：学习、编写 FAT32 文件系统
2022.5.1 – 2022.5.20	莫佳洋：实现进程管理与内存管理相关的系统调用 林杰克：构建测试环境，实现文件系统相关的系统调用
2022.5.21 – 2022.6.5	撰写开发文档，优化代码结构

2 系统设计

2.1 系统整体架构

如图 2-1 所示，系统整体根据 RISC-V 的三个特权级（这里没有考虑 H 模式）进行设计，自上而下分别是：

- U 模式（用户模式）：用于用户应用程序
- S 模式（管理员模式）：用于操作系统内核
- M 模式（机器模式）：用于引导加载程序等

在 RISC-V 中，各模式使用 ecall 与 eret 进行特权级切换，对于用户模式（即用户态）切换到管理员模式（即内核态），我们提供了 ABI（Application Binary Interface）接口，主要是 RongOS 实现的系统调用。对于管理员模式切换到机器模式，我们选择使用洛佳等开发者编写的 RustSBI，它为内核控制机器级提供了便捷的接口。

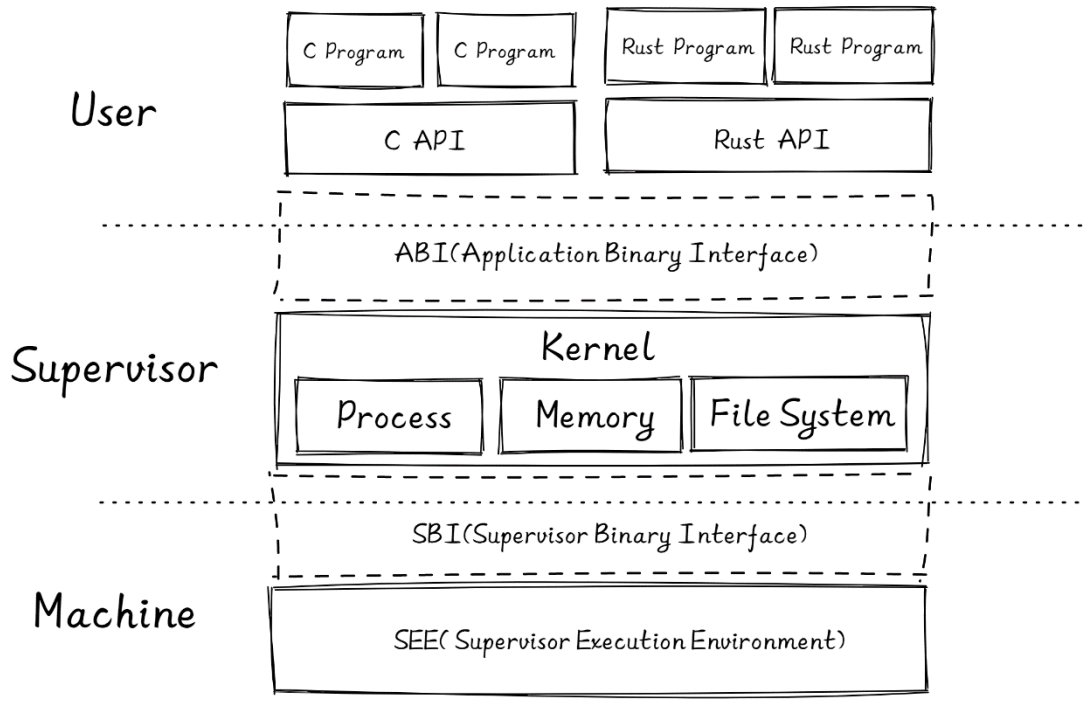

图 2-1 RongOS 系统整体架构

2.2 子模块设计

对于最重要的内核部分，也就是 S 特权级，我们借鉴了《Operating Systems: three easy pieces》的撰写想法（即虚拟化、并发、持久化），将 RongOS 也分成了三个主体部分以及各自的子模块：

1. 内存管理

- 地址数据类型模块：实现了地址类型定义与类型相互之间的转换规则
- 物理页帧管理器模块：管理空闲物理页帧，包括页帧的分配与回收等
- 地址空间模块：描述一个程序所拥有的地址空间
- 页表模块：实现与管理三级页表结构以及页表项
- 虚拟地址空间模块：管理线性虚拟地址空间的映射关系。
- 内核堆模块：负责内核空间的堆分配与回收

2. 进程管理

- 进程控制模块：实现进程控制块抽象进程信息
- 处理器管理模块：实现处理器控制块，提供调度等接口
- 进程调度模块：负责全局进程的调度，支持不同的调度算法
- 进程标识符模块：管理全局的进程标识符，确保标识符的唯一性

- 应用内核栈模块：管理内核虚拟地址空间中应用内核栈区域
- 上下文切换模块：通过内联汇编实现上下文切换功能。

3. 文件系统

- 内核虚拟文件系统：提供用户视角的文件系统抽象
- FAT32 文件系统：实现内核虚拟文件系统接口的一个实例
  - 磁盘块设备接口模块：定义磁盘块接口
  - 块缓存模块：实现内存中的磁盘块缓存
  - 磁盘布局模块：定义 FAT32 磁盘布局数据结构
  - 文件系统管理模块：实现文件系统管理器，提供 FAT32 管理接口
  - 虚拟文件模块：抽象 FAT32 文件结构，为内核使用提供接口

3 系统实现

3.1 内存管理

内存管理主要分为地址空间和页表结构。其作用就是对物理内存增加一层抽象，为操作系统上层的应用提供一个抽象接口，同时操作系统还能有效检测并阻止非法读取内存数据的行为。RongOS 为了实现对有限内存尽可能高效的利用引入了分页内存管理功能和 Lazy 机制，为了支持分时多任务并发内存安全的特性引入了地址空间的概念。

3.1.1 分页系统

RongOS 以页为单位进行内存管理，默认页面大小为 4KiB，支持 RISC-V SV39 多级页表，相比分段内存管理，分页内存管理的粒度更小且大小固定，这样页内部没有被用到的内碎片的大小也更小，提高了内存利用率。

为了方便实现虚拟页面到物理页帧的地址转换，我们给每个虚拟页面和物理页帧一个编号，分别称为虚拟页号 (VPN, Virtual Page Number)和物理页号 (PPN, Physical Page Number)。每个应用都有一个表示地址映射关系的页表 (Page Table) ，里面记录了该应用地址空间中的每个虚拟页面映射到实际物理内存对应的物理页帧。我们可以用页号来代表二者，因此如果将页表看成一个键值对，其

键的类型为虚拟页号，值的类型则为物理页号。

在页表中，还针对虚拟页号设置了一组保护位，它限制了应用对转换得到的物理地址对应的内存段的使用方式。最典型的如 RWX，R 表示当前应用可以读取该内存段的数据；W 表示当前应用可以向该内存段写入数据；X 则表示当前应用可以从该内存段取指令用来执行。一旦违反了这种限制则会触发异常，并被内核捕获到。通过适当的设置，可以检查一些应用在运行时的明显错误：比如应用试图修改只读的代码段，或者尝试从数据段取指令来执行。

3.1.2 地址空间设计

地址空间作为内存的抽象，可以为每个程序（包括内核代码）创建一个相互隔离的内存空间，从而解决了以下两个问题：

- 地址暴露问题：如果用户程序可以寻址内存的每个字节，它们就可以很容易地（故意地或偶然地）破坏操作系统，从而使系统错误运行或停止运行。即使在只有一个用户程序运行的情况下，这个问题也是存在的。引入地址空间后用户程序无法访问其他程序的内存空间，可以防止数据泄露和恶意代码的破坏。
- 程序重定向问题：在未引入地址空间之前，每段程序必须加载到不同的内存地址上，虽然可以通过重定位的方式加载，但这是一个缓慢且复杂的行为；引入地址空间之后，程序可以加载到位于不同地址空间的同一个虚拟地址上，从而省去了重定向的过程。

同时地址空间也引入了按需分配的思想，即地址空间为每个程序提供了 512GiB 的可寻址虚拟内存大小，但是只有在程序需要读写某块虚拟内存时相应的物理内存才会被分配给这一地址空间，这一特性将在 mmap 和 COW 的实现中详细说明。

（1） 内核地址空间

当计算机控制权由 RustSBI 交给 RongOS 后，内核便会开始创建内核虚拟空间，在内核空间中除了跳板和应用内核栈外其余内存地址均采用恒等映射，如图 3-1 所示。

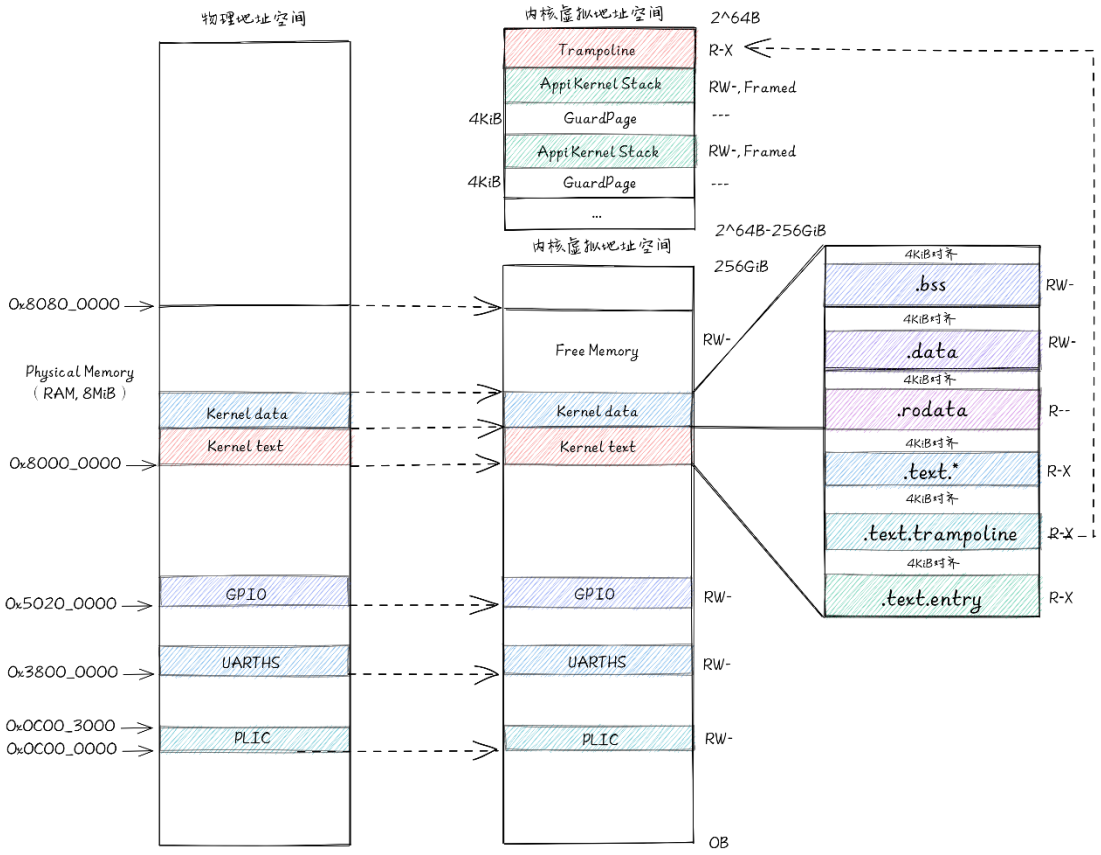

图 3-1 内核虚实空间映射

其中，物理地址范围 [0x0C00_0000,0x5020_0000] 内的若干段内存空间是硬件 MMIO 区域；物理地址范围 [0x8000_0000,0x8080_0000] 内的内存空间为 K210 的 8MiB 内存，内核数据区结束地址 ekernel 会在内核链接脚本中给出；物理地址范围 [ekernel, 0x8080_0000] 会在恒等映射后全部交给全局物理页帧管理器 FRAME_ALLOCATOR 由其负责页面的分配和回收。

（2） 应用地址空间

应用地址空间会在应用加载时创建，其中：

- .text 段, .rodata 段, .data 段, .bss 段的范围与数据由 ELF 文件或另一应用地址空间给出
- user stack 段使用一页 GuardPage 与 .bss 段分离，用作用户栈空间
- user heap 段使用一页 GuardPage 与 user stack 段分离，用作用户堆空间

应用的地址空间如图 3-2 所示：

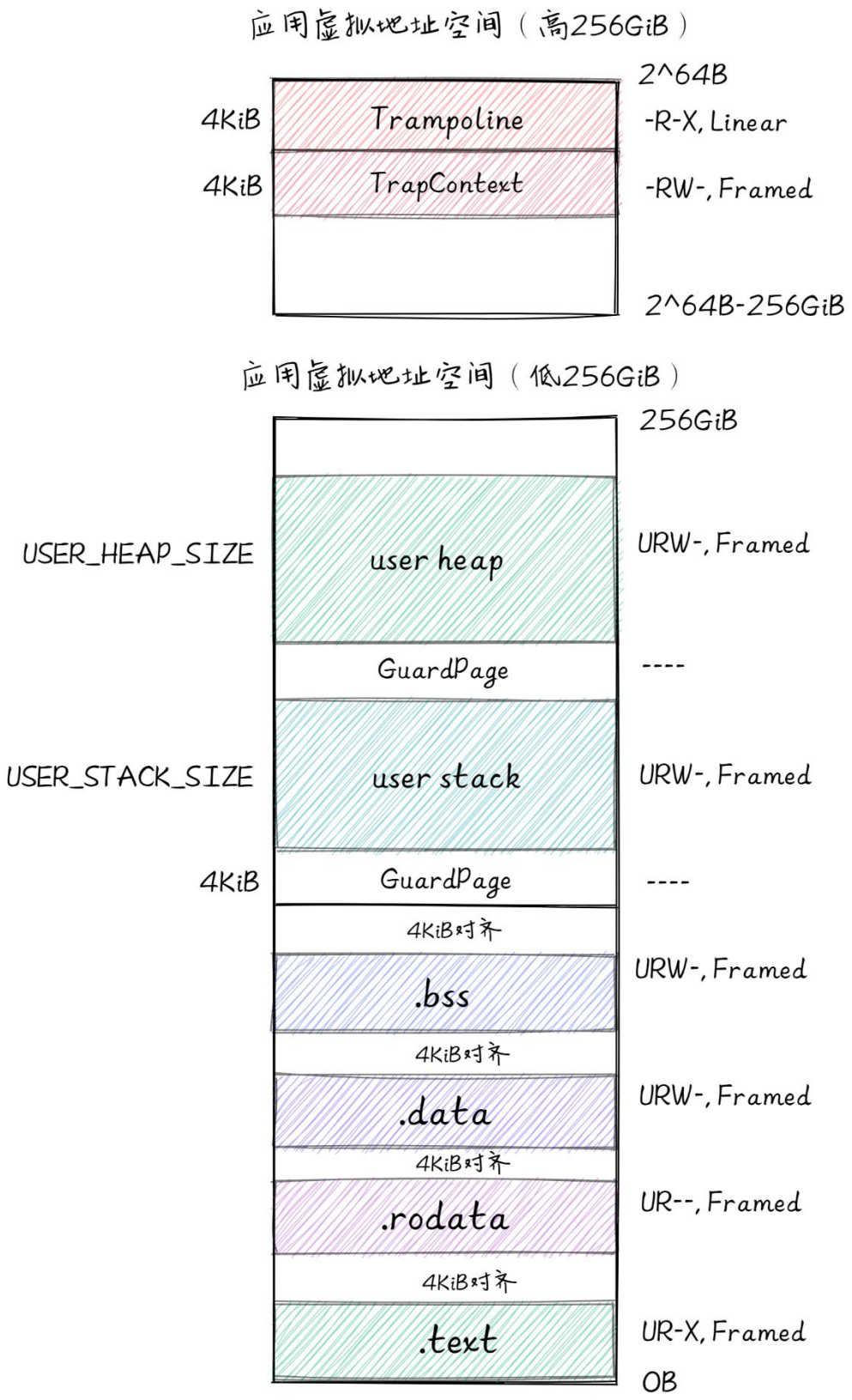

图 3-2 应用虚拟地址空间

3.1.3 地址相关的数据结构

下面分别给出了物理地址、虚拟地址、物理页号、虚拟页号的 Rust 类型声明，它们都是 Rust 的元组式结构体，可以看成 usize 的一种简单包装。我们刻意将它们各自抽象出不同的类型而不是都使用与 RISC-V 64 硬件直接对应的 usize 基本类型，就是为了在 Rust 编译器的帮助下，通过多种方便且安全的类型转换(Type Conversion)来构建页表。

```
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysAddr(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtAddr(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysPageNum(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtPageNum(pub usize);
```

3.1.4 物理页帧

我们将物理页帧抽象为 FrameTracker 结构体，这样就可以利用 Rust 语言特性实现在 FrameTracker 结构体创建时初始化内存空间，在 FrameTracker 结构体生命周期结束后回收物理页帧的功能。

```
pub struct FrameTracker {
    pub ppn: PhysPageNum,
}
```

物理页帧管理器向内核提供 new、alloc 和 dealloc 接口，分别用于初始化物理页帧管理器、分配和回收物理页帧；为了支持多种物理页帧管理方法，我们将接口抽象为 trait，再由具体的物理页帧管理器实现这写接口。

```
trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}
```

目前 RongOS 采用栈式物理页帧管理方案：利用 current，end 字段限定物理页帧管理器管理的内存空间范围，内核初始化时会将除内核代码所占空间之外的

剩余物理内存交给物理页帧管理器管理；利用 recycled 字段以栈式结构存放被回收的物理页帧，在下次分配时如果该字段非空，则优先使用该字段栈顶的物理页帧。实现代码如下：

```
pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}
impl StackFrameAllocator {
    pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
        self.current = l.0;
        self.end = r.0;
    }
}
impl FrameAllocator for StackFrameAllocator {
    fn new() -> Self {...}
    fn alloc(&mut self) -> Option<PhysPageNum> {
        if let Some(ppn) = self.recycled.pop() {
            Some(ppn.into())
        }
        else if self.current == self.end {
            None
        }
        else {
            self.current += 1;
            Some((self.current - 1).into())
        }
    }
    fn dealloc(&mut self, ppn: PhysPageNum) {
        let ppn = ppn.0;
        if ppn >= self.current || self.recycled.iter().any(|&v| v ==
ppn) {
            panic!("Frame ppn={:#x} has not been allocated!", ppn);
        }
        self.recycled.push(ppn);
    }
}
```

在结构体设计中我们利用 Rust 语言的生命周期和所有权机制来实现结构体的自动回收功能，Rust 中每个变量都有且只有一个所有者，当所有者离开作用域，变量会被自动释放/析构。因此我们只需定义变量析构时的具体动作，而不需要

手动进行变量析构。

如下代码为物理页帧的析构过程，当物理页帧生命周期结束后，内核会自动调用 FrameTracker::drop()函数开始对物理页帧的析构，经过两次函数调用后由栈式物理页帧管理器在验证物理页帧有效性后，将此物理页帧的物理页号压入回收物栈中供下次分配使用。

```
impl Drop for FrameTracker {
    fn drop(&mut self) {
        frame_dealloc(self.ppn);
    }
}
pub fn frame_dealloc(ppn: PhysPageNum) {
    FRAME_ALLOCATOR.exclusive_access().dealloc(ppn);
}
```

3.1.5 页表项

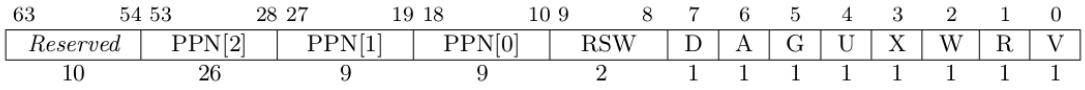

图 3-3 SV39 页表项

图3-3为SV39分页模式下的页表项，其中 [63:54] 这10位保留不用，[53:10] 这 44 位是物理页号，最低的 8 位 [7:0] 则是标志位，它们的含义如下：

- V(Valid)：仅当位 V 为 1 时，页表项才是合法的；
- R(Read)/W(Write)/X(eXecute)：分别控制索引到这个页表项的对应虚拟页面是否允许读/写/执行；
- U(User)：控制索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下是否被允许访问；
- A(Accessed)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被访问过；
- D(Dirty)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被修改过。

如下代码为页表项数据结构的具体实现，我们使用名为 PageTableEntry 的结构体包含了一个 usize 类型，用以存储页表项的 64 位数据；成员函数 PageTableEntry::new()提供了将物理页号和页表项标志位按比特写入物理空间的

功能。通过分析 PageTableEntry 结构体和页表大小可知，每个 PageTableEntry 结构体占用 64bit 空间，页大小在内核中定义为 4KiB，每个物理页可以存放 512 个 PageTableEntry 结构体。

```
pub struct PageTableEntry {
    pub bits: usize,
}
impl PageTableEntry {
    pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
        PageTableEntry {
            bits: ppn.0 << 10 | flags.bits as usize,
        }
    }
}
```

3.1.6 地址空间具体实现

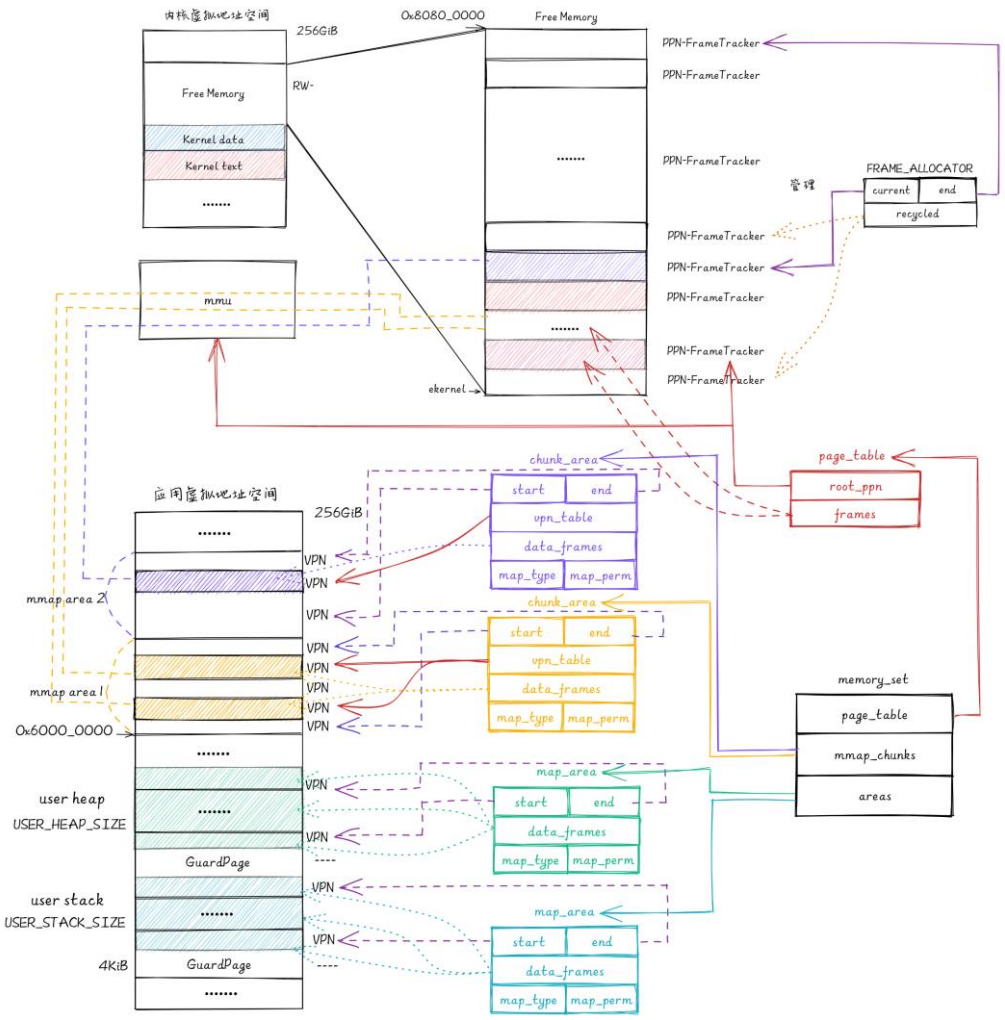

图 3-4 地址空间实现细节-数据结构

如图 3-5 所示，地址空间部分由四个结构体组成，具体内容如表 3-1 所示。

表 3-1 地址空间结构体定义

结构体	名称	逻辑功能
MemorySet	地址空间	数据结构如图右下角所示，由页表、连续逻辑段和离散逻辑段组成，用于整体描述上图左下角所示的应用虚拟地址空间
MapArea	(虚拟地址)连续逻辑段	数据结构如图中下部所示， start,end 字段用于描述连续逻辑段虚拟页号范围； data_frame 字段用于存放连续逻辑段中物理页帧对应的 Frame_tracker； map_type,map_perm 分别表示内存空间映射方法和内存空间权限；
ChunkArea	(虚拟地址)离散逻辑段	数据结构如图中部所示， start,end 字段用于描述离散逻辑段虚拟页表范围； vpn_table 字段用于存放离散逻辑段中已被映射的物理页帧对应的 Frame_tracker； data_frame 字段用于存放离散逻辑段中物理页帧对应的 Frame_tracker; map_type,map_perm 分别表示内存空间映射方法和内存空间权限；
PageTable	SV39 多级页表	数据结构如图右侧中部所示， root_ppn 字段表示多级页表更页面的物理页号； frames 字段用于保存页表项所在页对应的 Frame_tracker

（3） 页表

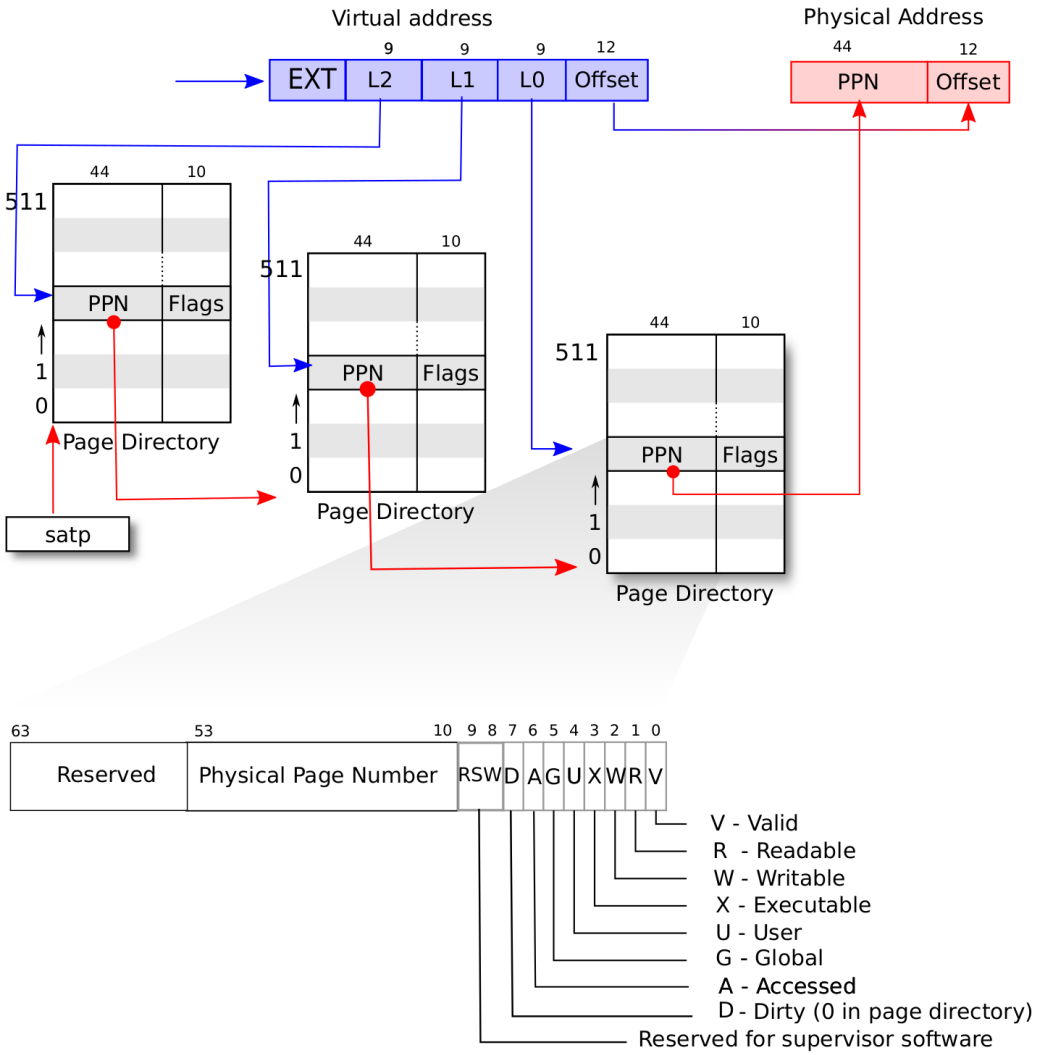

图 3-6 SV39 地址转换

图 3-6 为 SV39 地址转换的全过程图示，页表的功能就是管理存有页表项的物理页帧，通过对页表项的修改实现地址映射的修改。

对页表结构体的定义如下，其中 root_ppn 字段是页表根页面对应的物理页号，只有将其格式化并写入 satp 寄存器后 SV39 多级页表才会被正式启用，使用不同的 root_ppn 格式化并写入 satp 寄存器便可完成页表的切换，从而实现地址空间的切换。

```
pub struct PageTable {
    root_ppn: PhysPageNum,
    frames: Vec<FrameTracker>,
}
```

（4） 逻辑段

逻辑段根据其虚拟地址空间中页面是否会被一次性连续映射分为连续逻辑段和离散逻辑段；类似于进程代码空间和堆栈空间这种在进程创建时逻辑段会被全部映射到虚拟地址空间的属于连续逻辑段，类似于 mmap area 这样的发生缺页异常才尝试进行加载的属于离散逻辑段。

在逻辑段结构设计中连续逻辑段因页面会被连续映射，在连续逻辑段结构体中只需保存连续逻辑段起止虚拟页号便可确定逻辑段范围；而离散逻辑段中页面只有在发生缺页异常后才会被加载，虚拟地址空间中页面不是连续的，故在离散逻辑段结构体中还需保存已加载的页面对应的虚拟页号。

当连续逻辑段建立映射时，MapArea::map()循环调用 MapArea::map_one()映射一页，实现连续地址的映射。MapArea::map_one()首先判断逻辑段的映射方式，如果为恒等映射就令物理页号和虚拟页号相同，如果为随机映射则函数为每一个虚拟页向物理页帧管理器申请一个物理页帧，然后在页表中建立映射。

代码实现如下：

```
impl MapArea {
    fn map_one(&mut self,page_table: &mut PageTable, vpn: VirtPageNum) {
        let ppn: PhysPageNum;
        match self.map_type {
            MapType::Identical => {
                ppn = PhysPageNum(vpn.0);
            }
            MapType::Framed => {
                let frame = frame_alloc().unwrap();
                ppn = frame.ppn;
                self.data_frames.insert(vpn, frame);
            }
        }
        let pte_flags =
PTEFlags::from_bits(self.map_perm.bits).unwrap();
        page_table.map(vpn, ppn, pte_flags);
    }
    pub fn map(&mut self, page_table: &mut PageTable) {
        for vpn in self.vpn_range {
            self.map_one(page_table, vpn);
        }
    }
}
```

（5） 地址空间

地址空间是页表、连续逻辑段和离散逻辑段的综合，以下为地址空间的定义：

```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
    mmap_chunks: Vec<ChunkArea>,
}
```

如下代码所示，地址空间有三种生成的方式，分别为内核生成内核地址空间、从 ELF 文件生成地址空间和从已有用户程序复制地址空间，这三种地址空间生成方式分别用在内存管理初始化、exec 和 fork 中。

```
impl MemorySet {
    pub fn new_kernel() -> Self {...}
    pub fn from_elf(elf_data: &[u8]) -> (Self, usize, usize, usize){...}
    pub fn from_existed_user(user_space: &MemorySet) -> MemorySet{...}
}
```

3.2 进程管理

3.2.1 进程标识符

进程标识符是一个互不相同的整数，用于唯一标识一个进程，如下所示，我们使用 RAII（Resource Acquire Is Initialization）的思想，将其抽象为一个 PidHandle 类型，当它的生命周期结束后对应的整数会被编译器自动回收。

```
pub struct PidHandle(pub usize);
```

为管理进程标识符的分配，保证进程标识符的唯一性，我们用类似之前的物理页帧分配器 FrameAllocator 的设计思想，实现了一个同样使用简单栈式分配策略的进程标识符分配器 PidAllocator，并将其全局实例化为 PID_ALLOCATOR。

```
struct PidAllocator {
    current: usize,
    recycled: Vec<usize>,
}

impl PidAllocator {
    pub fn new() -> Self {
        PidAllocator {
            current: 0,
            recycled: Vec::new(),
        }
```

```
    }
    pub fn alloc(&mut self) -> PidHandle {
        if let Some(pid) = self.recycled.pop() {
            PidHandle(pid)
        } else {
            self.current += 1;
            PidHandle(self.current - 1)
        }
    }
    pub fn dealloc(&mut self, pid: usize) {
        assert!(pid < self.current);
        assert!(
            !self.recycled.iter().any(|ppid| *ppid == pid),
            "pid {} has been deallocated!", pid
        );
        self.recycled.push(pid);
    }
}
impl Drop for PidHandle {
    fn drop(&mut self) {
        PID_ALLOCATOR.exclusive_access().dealloc(self.0);
    }
}
```

PidAllocator::alloc 将会分配出去一个将 usize 包装之后的 PidHandle。我们将其包装为一个全局分配进程标识符的接口 pid_alloc 提供给内核的其他子模块.同时我们也为 PidHandle 实现 Drop Trait 来允许编译器进行自动的资源回收。

在内存管理中的内核地址空间设计概览中我们介绍过内核地址空间布局，我们将每个应用的内核栈按照进程标识符从小到大的顺序将它们作为逻辑段从高地址到低地址放在内核地址空间中，且两两之间保留一个守护页面使得我们能够尽可能早的发现内核栈溢出问题。以下代码为内核栈的实现：

```
pub struct KernelStack {
    pid: usize,
}
impl KernelStack {
    #[allow(unused)]
    pub fn push_on_top<T>(&self, value: T) -> *mut T
    where
        T: Sized,
    {
        let kernel_stack_top = self.get_top();
```

```
        let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as
*mut T;
        unsafe {
            *ptr_mut = value;
        }
        ptr_mut
    }
    pub fn get_top(&self) -> usize {
        let (_, kernel_stack_top) = kernel_stack_position(self.0);
        kernel_stack_top
    }
    impl Drop for KernelStack {
        fn drop(&mut self) {
            let (kernel_stack_bottom, _) =
kernel_stack_position(self.pid);
            let kernel_stack_bottom_va: VirtAddr =
kernel_stack_bottom.into();
            KERNEL_SPACE
                .exclusive_access()
                .remove_area_with_start_vpn(kernel_stack_bottom_va.into(
));
        }
    }
}
```

KernelStack::push_on_top 函数可以将一个类型为 T 的变量压入内核栈顶并返回其裸指针，这也是一个泛型函数。它在实现的时候用到了 KernelStack::get_top 方法来获取当前内核栈顶在内核地址空间中的地址。内核栈 KernelStack 也用到了 RAII 的思想，具体来说，实际保存它的物理页帧的生命周期与它绑定在一起，当 KernelStack 生命周期结束后，这些物理页帧也将会被编译器自动回收。这仅需要为 KernelStack 实现 Drop Trait，一旦它的生命周期结束则在内核地址空间中将对应的逻辑段删除。

3.2.2 进程控制块

在内核中，每个进程的执行状态、资源控制等元数据均保存在进程控制块结构中，它是内核对进程进行管理的单位，在内核看来，它就等价于一个进程。

任务控制块中包含两部分：

- 在初始化之后就不再变化的元数据：直接放在任务控制块中。这里将 pid、tgid 和内核栈 KernelStack 放在其中；

- 在运行过程中可能发生变化的元数据：则放在 TaskControlBlockInner 中，将它再包裹上一层 UPSafeCell<T>放在任务控制块中。这是因为在我们的设计中外层只能获取任务控制块的不可变引用，若想修改里面的部分内容的话这需要 UPSafeCell<T>所提供的内部可变性。

TaskControlBlockInner 中则包含下面这些内容：

- trap_cx_ppn：指出了应用地址空间中的 Trap 上下文，被放在的物理页帧的物理页号。
- task_cx：将暂停的任务的任务上下文保存在任务控制块中。
- task_status：维护当前进程的执行状态。
- parent：指向当前进程的父进程（如果存在的话）。注意我们使用 Weak 而非 Arc 来包裹另一个任务控制块，因此这个智能指针将不会影响父进程的引用计数。
- children：将当前进程的所有子进程的任务控制块以 Arc 智能指针的形式保存在一个向量中，这样才能够更方便的找到它们。
- exit_code：当进程调用 exit 系统调用主动退出或者执行出错由内核终止的时候，它的退出码 exit_code 会被内核保存在它的任务控制块中，并等待它的父进程通过 waitpid 回收它的资源的同时也收集它的 PID 以及退出码。
- base_size：应用数据仅有可能出现在应用地址空间低于 base_size：字节的区域中。借助它我们可以清楚的知道应用有多少数据驻留在内存中。
- memory_set：表示应用地址空间。
- mmap_area：表示 mmap 内存空间信息的结构体
- heap_start：表示堆空间起始虚拟地址
- heap_pt：表示堆空间当前堆顶虚拟地址

任务控制块 TaskControlBlock 目前提供以下方法：

- inner_exclusive_access()：获取 TaskControlBlockInner 的可变引用
- getpid()：以 usize 的形式返回当前进程的进程标识符
- new()：创建一个新的进程，目前仅用于内核中手动创建唯一一个初始进程 initproc
- exec()：实现 exec 系统调用，即当前进程加载并执行另一个 ELF 格式可

执行文件。

- fork()：用来实现 fork 系统调用，即当前进程 fork 出来一个与之几乎相同的子进程。
- mmap()：用来在进程虚拟地址空间中分配创建一片虚拟内存地址映射
- munmap()：用来在进程虚拟地址空间中删除一片虚拟内存地址映射
- lazy_mmap()：用来在 mmap 缺页时使用 lzay 加载机制加载一片虚拟内存地址映射

3.2.3 任务管理器

TaskManager 将所有的任务控制块用引用计数 Arc 智能指针包裹后放在一个双端队列 VecDeque 中。代码实现如下：

```
pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}
```

正如之前介绍的那样，我们并不直接将任务控制块放到 TaskManager 里面，而是将它们放在内核堆上，在任务管理器中仅存放他们的引用计数智能指针，这也是任务管理器的操作单位。这样做的原因在于，任务控制块经常需要被放入/取出，如果直接移动任务控制块自身将会带来大量的数据拷贝开销，而对于智能指针进行移动则没有多少开销。其次，允许任务控制块的共享引用在某些情况下能够让我们的实现更加方便。

TaskManager 提供 add/fetch 两个操作，前者表示将一个任务加入队尾，后者则表示从队头中取出一个任务来执行。从调度算法来看，这里用到的就是最简单的 RR 算法。代码实现如下：

```
impl TaskManager {
    pub fn new() -> Self {
        Self { ready_queue: VecDeque::new() }
    }
    pub fn add(&mut self, task: Arc<TaskControlBlock>) {
        self.ready_queue.push_back(task);
    }
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.ready_queue.pop_front()
    }
}
```

全局实例 TASK_MANAGER 则提供给内核的其他子模块 add_task/fetch_task

两个函数。代码实现如下：

```
lazy_static! {
    pub static ref TASK_MANAGER: UPSafeCell<TaskManager> =
        unsafe { UPSafeCell::new(TaskManager::new()) };
}
pub fn add_task(task: Arc<TaskControlBlock>) {
    PID2TCB
        .exclusive_access()
        .insert(task.getpid(), Arc::clone(&task));
    TASK_MANAGER.exclusive_access().add(task);
}
pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
    TASK_MANAGER.exclusive_access().fetch()
}
```

3.2.4 处理器管理结构

处理器管理结构用于维护 CPU 状态信息，我们为每一个核心创建了一个全局实例，以下为处理器管理结构体代码：

```
pub struct Processor {
    current: Option<Arc<TaskControlBlock>>,
    idle_task_cx: TaskContext,
}
impl Processor {
    pub fn take_current(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.current.take()
    }
    pub fn current(&self) -> Option<Arc<TaskControlBlock>> {
        self.current.as_ref().map(Arc::clone)
    }
}
```

在抢占式调度模型中，在一个处理器上执行的任务常常被换入或换出，因此我们提供了相关的接口用于获取处理器正在执行的任务信息。

```
pub fn take_current_task() -> Option<Arc<TaskControlBlock>> {...}
pub fn current_task() -> Option<Arc<TaskControlBlock>> {...}
pub fn current_user_token() -> usize {...}
pub fn current_trap_cx() -> &'static mut TrapContext {...}
```

3.2.5 初始进程的创建

内核初始化完毕之后即会调用 task 子模块提供的 add_initproc 函数来将初始进程 initproc 加入任务管理器，但在这之前我们需要初始化初始进程的进程控制

块 INITPROC，这个过程基于 lazy_static 在运行时完成。

```
lazy_static! {
    pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new({
        extern "C" {
            fn _num_app();
        }
        let num_app_ptr = _num_app as usize as *const usize;
        let num_app = unsafe { num_app_ptr.read_volatile() };
        let app_start = unsafe
{ core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1) };
        TaskControlBlock::new( unsafe{
            core::slice::from_raw_parts(
                app_start[0] as *const u8,
                app_start[1] - app_start[0]
            ) }
        )
    });
}

pub fn add_initproc() {
    add_task(INITPROC.clone());
}
```

我们调用 TaskControlBlock::new 来创建一个进程控制块，它需要传入 ELF 可执行文件的数据切片作为参数，我们已经将初始进程代码通过内联汇编固定在内核数据内，这个初始程序会依次运行全部测试程序，只需通过全局符号 _num_app 确定数据区起始地址后按数据切片的格式读取即可。在初始化 INITPROC 之后，就可以在 add_initproc 中调用 task 的任务管理器 manager 子模块提供的 add_task 接口，将其加入到任务管理器。

3.2.6 进程调度机制

通过调用 task 子模块提供的 suspend_current_and_run_next 函数可以暂停当前任务并切换到下一个任务，当应用调用 sys_yield 主动交出使用权、本轮时间片用尽或者由于某些原因内核中的处理无法继续的时候，就会在内核中调用此函数触发调度机制并进行任务切换。以下代码为这两种情况的代码实现：

```
pub fn sys_yield() -> isize {
    suspend_current_and_run_next();
    0
}
```

```
pub fn trap_handler() -> ! {
    set_kernel_trap_entry();
    let scause = scause::read();
    let stval = stval::read();
    match scause.cause() {
        Trap::Interrupt(Interrupt::SupervisorTimer) => {
            set_next_trigger();
            suspend_current_and_run_next();
        }
        ...
    }
    trap_return();
}
```

以下代码为 suspend_current_and_run_next 函数的具体实现：

```
pub fn suspend_current_and_run_next() {
    let task = take_current_task().unwrap();
    let mut task_inner = task.inner_exclusive_access();
    let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
    task_inner.task_status = TaskStatus::Ready;
    drop(task_inner);
    add_task(task);
    schedule(task_cx_ptr);
}
```

首先通过 take_current_task 来取出当前正在执行的任务，修改其进程控制块内的状态，随后将这个任务放入任务管理器的队尾。接着调用 schedule 函数来触发调度并切换任务。

3.2.7 进程生成机制

fork()首先调用 MemorySet::from_existed_user 函数将父进程的地址空间拷贝一份作为子进程的用户地址空间，然后调用 pid_alloc 函数获取一个 PID 根据 PID 创建应用内核栈，紧接着复制父进程的文件描述符表、创建进程控制块，最后把新生成的进程加入到子进程向量中函数返回子进程的进程控制块。

```
pub fn fork(self: &Arc<TaskControlBlock>, is_create_thread: bool) ->
Arc<TaskControlBlock> {
    let mut parent_inner = self.inner_exclusive_access();
    // 拷贝用户地址空间
    let memory_set =
MemorySet::from_existed_user(&parent_inner.memory_set);
    let trap_cx_ppn = memory_set
        .translate(VirtAddr::from(TRAP_CONTEXT).into())
```

```
        .unwrap()
        .ppn();
    let pid_handle = pid_alloc();   // 分配一个 PID
    let mut tgid = 0;
    if is_create_thread{
        tgid = self.pid.0;
    }
    else{
        tgid = pid_handle.0;
    }
    // 根据 PID 创建一个应用内核栈
    let kernel_stack = KernelStack::new(&pid_handle);
    let kernel_stack_top = kernel_stack.get_top();

    let mut new_fd_table: Vec<Option<Arc<dyn File + Send + Sync>>> =
Vec::new();
    for fd in parent_inner.fd_table.iter() {
        if let Some(file) = fd {
            new_fd_table.push(Some(file.clone()));
        } else {
            new_fd_table.push(None);
        }
    }

    let task_control_block = Arc::new(TaskControlBlock {
        pid: pid_handle,
        tgid,
        kernel_stack,
        inner: unsafe {
            UPSafeCell::new(TaskControlBlockInner {
                trap_cx_ppn,
                base_size: parent_inner.base_size,
                heap_start: parent_inner.heap_start,
                heap_pt: parent_inner.heap_pt,
                task_cx:
TaskContext::goto_trap_return(kernel_stack_top),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: Some(Arc::downgrade(self)),
                children: Vec::new(),
                exit_code: 0,
                fd_table: new_fd_table,
                signals: SignalFlags::empty(),
                current_path: parent_inner.current_path.clone(),
```

```
                mmap_area: MmapArea::new(VirtAddr::from(MMAP_BASE),
VirtAddr::from(MMAP_BASE)),
            })
        },
    });
    // 把新生成的进程加入到子进程向量中
    parent_inner.children.push(task_control_block.clone());
    // 更新子进程 trap 上下文中的栈顶指针
    let trap_cx =
task_control_block.inner_exclusive_access().get_trap_cx();
    trap_cx.kernel_sp = kernel_stack_top;

    task_control_block
}
```

exec()使得一个进程能够加载一个新应用的 ELF 可执行文件中的代码和数据替换原有的应用地址空间中的内容，并开始执行。系统调用分发函数会根据系统调用号将 exec 系统调用交由 TaskControlBlock::exec 函数执行。该函数首先从 ELF 文件生成一个全新的地址空间并直接替换进来，这将导致原有的地址空间生命周期结束，里面包含的全部物理页帧都会被回收；然后修改新的地址空间中的 Trap 上下文，将解析得到的应用入口点、用户栈位置以及一些内核的信息进行初始化，这样才能正常实现 Trap 机制。

以下代码为 exec() 的实现：

```
pub fn exec(&self, elf_data: &[u8], args: Vec<String>) {
    // 从 ELF 文件生成一个全新的地址空间并直接替换
    let (memory_set, mut user_sp,user_heap, entry_point) =
MemorySet::from_elf(elf_data);
    let trap_cx_ppn = memory_set
        .translate(VirtAddr::from(TRAP_CONTEXT).into())
        .unwrap()
        .ppn();

    user_sp -= (args.len() + 1) * core::mem::size_of::<usize>();
    let argv_base = user_sp;
    let mut argv: Vec<_> = (0..=args.len()) // 获取参数字符串首地址数组在
用户栈中的可变引用
        .map(|arg| {
            translated_refmut(
                memory_set.token(),
                (argv_base + arg * core::mem::size_of::<usize>()) as
*mut usize,
```

```
            )
        })
        .collect();

    // 参数字符串
    *argv[args.len()] = 0;  // 标记参数尾
    for i in 0..args.len() {
        user_sp -= args[i].len() + 1;   // 在用户栈中分配参数i 的空间
        *argv[i] = user_sp;             // 在参数字符串首地址数组中记录参数
i 首地址
        let mut p = user_sp;
        for c in args[i].as_bytes() {   // 将参数写入到用户栈
            *translated_refmut(memory_set.token(), p as *mut u8) = *c;
            p += 1;
        }                               // 写入字符串结束标记
        *translated_refmut(memory_set.token(), p as *mut u8) = 0;
    }

    user_sp -= user_sp % core::mem::size_of::<usize>();

    let mut inner = self.inner_exclusive_access();
    inner.memory_set = memory_set;  // 这将导致原有的地址空间生命周期结束,
里面包含的全部物理页帧都会被回收
    inner.heap_start = user_heap;
    inner.heap_pt = user_heap;
    inner.trap_cx_ppn = trap_cx_ppn;
    let trap_cx = inner.get_trap_cx();
    // 修改新的地址空间中的 Trap 上下文，将解析得到的应用入口点、用户栈位置以
及一些内核的信息进行初始化
    *trap_cx = TrapContext::app_init_context(
        entry_point,
        user_sp,
        KERNEL_SPACE.exclusive_access().token(),
        self.kernel_stack.get_top(),
        trap_handler as usize,
    );
    // 修改 Trap 上下文中的 a0/a1 寄存器
    trap_cx.x[10] = args.len(); // a0 表示命令行参数的个数
    trap_cx.x[11] = argv_base;  // a1 则表示 参数字符串首地址数组 的起始地
址
}
```

3.2.8 进程退出与资源回收机制

当应用调用 sys_exit 系统调用主动退出或者出错由内核终止之后，会在内核中调用 exit_current_and_run_next 函数退出当前进程并切换到下一个进程。

```
pub fn exit_current_and_run_next(exit_code: i32) {
    // 获取访问权限，修改进程状态
    let task = take_current_task().unwrap();
    remove_from_pid2task(task.getpid());
    let mut inner = task.inner_exclusive_access();
    inner.task_status = TaskStatus::Zombie; // 后续才能被父进程在 waitpid
系统调用的时候回收
    // 记录退出码，后续父进程在 waitpid 的时候可以收集
    inner.exit_code = exit_code;
    {   // 将这个进程的子进程转移到 initproc 进程的子进程中
        let mut initproc_inner = INITPROC.inner_exclusive_access();
        for child in inner.children.iter() {
            child.inner_exclusive_access().parent =
Some(Arc::downgrade(&INITPROC));
            initproc_inner.children.push(child.clone()); // 引用计数 -1
        }
    }
    inner.children.clear(); // 引用计数 +1
    // 对于当前进程占用的资源进行早期回收
    inner.memory_set.recycle_data_pages();
    drop(inner);
    drop(task);
    // 使用全 0 的上下文填充换出上下文，开启新一轮进程调度
    let mut _unused = TaskContext::zero_init();
    schedule(&mut _unused as *mut _);
}
```

我们调用 take_current_task 来将当前进程控制块从处理器监控 PROCESSOR 中取出一份拷贝，这是为了正确维护进程控制块的引用计数；我们将进程控制块中的状态修改为 TaskStatus::Zombie 即僵尸进程，这样它后续才能被父进程在 waitpid 系统调用的时候回收；我们将传入的退出码 exit_code 写入进程控制块中，后续父进程在 waitpid 的时候可以收集；MemorySet::recycle_data_pages 只是将地址空间中的逻辑段列表 areas 清空（即执行 Vec 向量清空），这将导致应用地址空间被回收（即进程的数据和代码对应的物理页帧都被回收），但用来存放页表的那些物理页帧此时还不会被回收（会由父进程最后回收子进程剩余的占用资源）。最后我们调用 schedule 触发调度及任务切换。

父进程通过 sys_waitpid 系统调用来回收子进程的资源并收集它的一些信息：

```
pub fn sys_waitpid(pid: isize, exit_code_ptr: *mut i32) -> isize {
    let task = current_task().unwrap();
    let inner = task.inner_exclusive_access();

    // 根据 pid 参数查找有没有符合要求的进程
    if !inner.children.iter()
        .any(|p| pid == -1 || pid as usize == p.getpid()) {
        return -1;
    }
    drop(inner);
    loop{
        let mut inner = task.inner_exclusive_access();
        // 查找所有符合 PID 要求的处于僵尸状态的进程，如果有的话还需要同时找出
它在当前进程控制块子进程向量中的下标
        let pair = inner.children.iter().enumerate().find(|(_, p)| {
            p.inner_exclusive_access().is_zombie() && (pid == -1 || pid
as usize == p.getpid())
        });
        if let Some((idx, _)) = pair {
            // 将子进程从向量中移除并置于当前上下文中
            let child = inner.children.remove(idx);
            assert_eq!(Arc::strong_count(&child), 1);
            // 收集的子进程信息返回
            let found_pid = child.getpid();
            let exit_code = child.inner_exclusive_access().exit_code;
            // 将子进程的退出码写入到当前进程的应用地址空间中
            if exit_code_ptr as usize != 0 {
                *translated_refmut(inner.memory_set.token(),
exit_code_ptr) = exit_code << 8;
            }
            return found_pid as isize;
        } else {
            // 如果找不到的话则放权等待
            drop(inner);
            suspend_current_and_run_next();
        }
    }
}
```

3.3 文件系统

3.3.1 内核：虚拟文件系统

如图 3-7 所示，虚拟文件系统（Virtual File System，VFS）是一个内核软件层，在具体的文件系统之上抽象的一层，用来处理与文件系统相关的所有系统调用，表现为能够给各种文件系统提供一个通用的接口，使上层的应用程序能够使用通用的接口访问不同文件系统，同时也为不同文件系统的通信提供了媒介。

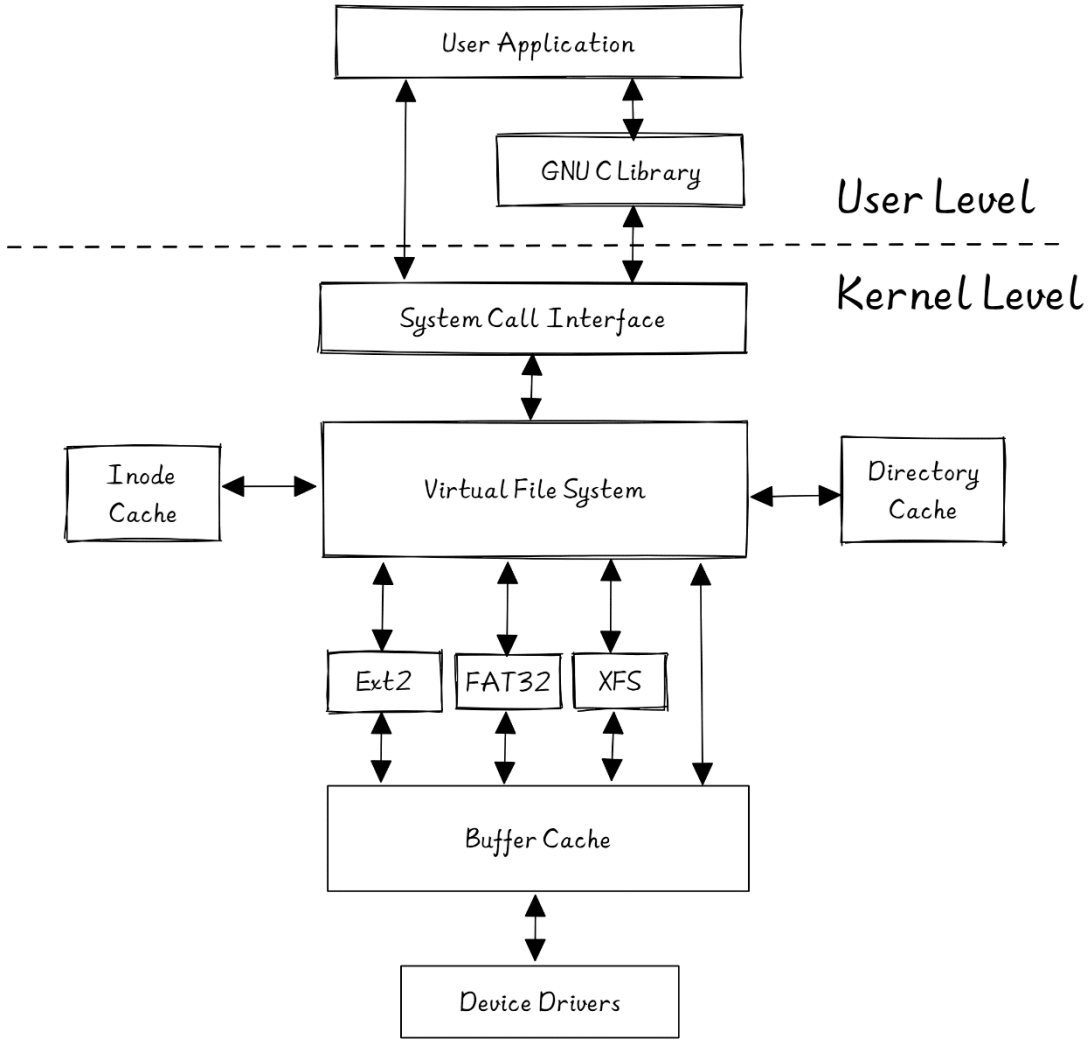

图 3-7 虚拟文件系统架构

VFS 为了提供对不同底层文件系统的统一接口，需要有一个高度的抽象和建模，这就是 VFS 的核心设计——统一文件模型。目前的 Linux 系统的 VFS 都是源于 Unix 家族，因此这里所说的 VFS 对所有 Unix 家族的系统都适用。Unix 家族的 VFS 的文件模型定义了四种对象，这四种对象构建起了统一文件模型。

- Superblock：存储文件系统基本的元数据。如文件系统类型、大小、状态，以及其他元数据相关的信息（元元数据）
- Index node（inode）：保存一个文件相关的元数据。包括文件的所有者（用户、组）、访问时间、文件类型等，但不包括这个文件的名称。文件和目录均有具体的 inode 对应。
- Directory entry（dentry）：保存了文件（目录）名称和具体的 inode 的对应关系，用来粘合二者，同时可以实现目录与其包含的文件之间的映射关系。另外也作为缓存的对象，缓存最近最常访问的文件或目录，提升系统性能。
- File：一组逻辑上相关联的数据，被一个进程打开并关联使用。

但相较于 Linux，RongOS 可以说是非常简单，因此我们的 VFS 层并没有完全按照 Unix 的文件模型，而是自己设计了一个 OSInode 结构用于进程访问。

```
pub struct OSInode {
    readable: bool,
    writable: bool,
    inner: Mutex<OSInodeInner>,
}

pub struct OSInodeInner {
    offset: usize,
    inode: Arc<VFile>,
}
```

我们将读写属性放在外面，将文件实际内容包括文件当前偏移量放在了 OSInodeInner 中，这样做的好处是无需取得互斥锁即可先进行权限判断，可以提高系统整体性能。注意，这里的 VFile 类型是由下一级文件系统提供（即后续会提到的 Simple-FAT32）。

为了实现文件系统相关的系统调用，我们为 OSInode 提供了以下接口：

- new：新建一个 OSInode 对象，其中 inner.inode（即 VFile 对象）由形参传入。
- delete：删除文件，在文件系统中删除目录项且清空文件数据。
- read：读取文件内容保存到缓冲区中
- write：将缓冲区中的数据写入到文件中
- get_fstat：获取文件状态信息

- get_dirent：获取目录文件的目录项信息

在 inode.rs 文件中，还包含以下两个重要接口：

- open：根据工作目录、文件路径、打开方式三个参数打开（或创建）一个文件，返回 OSInode 对象的引用
- chdir：修改进程当前的工作目录

3.3.2 内核：文件系统初始化

在 inode.rs 中，有如下代码：

```
lazy_static! {
    pub static ref ROOT_INODE: Arc<VFile> = {
        let fat32_manager = FAT32Manager::open(BLOCK_DEVICE.clone());
        Arc::new(create_root_vfile(&fat32_manager)) // 返回根目录
    };
}
```

这里使用到了 Rust 的 lazy_static 宏定义，它的作用是对全局变量初始化进行延迟，即首次使用时才进行初始化。

对 ROOT_INODE 的初始化分两步：

1. 打开文件系统，获得文件系统管理器对象，其中会初始化 FAT32 文件系统。（详见 3-FAT32 ）
2. 创建根目录的 VFile 对象，也即 ROOT_INODE，接下来内核所有的文件系统操作都是以它为基础。

3.3.3 内核：一切皆是文件

在 UNIX 操作系统中，“一切皆文件” (Everything is a file) 是一种重要的设计思想，这种设计思想继承于 Multics 操作系统的通用性文件的设计理念，并进行了进一步的简化。它将键盘、显示器、以磁盘为代表的存储介质、以串口为代表的通信设备等都抽象成了文件这一概念。

RongOS 中体现此思想的是位于 mod.rs 中的 File trait：

```
pub trait File: Send + Sync {
    fn readable(&self) -> bool;
    fn writable(&self) -> bool;
    fn read(&self, buf: UserBuffer) -> usize;
    fn write(&self, buf: UserBuffer) -> usize;
    fn get_fstat(&self, kstat: &mut Kstat);
```

```
    fn get_dirent(&self, dirent: &mut Dirent) -> isize;
    fn get_name(&self) -> String;
}
```

可以认为它定义了一个名叫 File 接口规范，只要是实现 File 这一接口的，都可以被拿来当做文件处理。因此，在我们的进程控制块中，对文件描述符表有如下定义：

```
pub struct TaskControlBlockInner {
    ......
    pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>,
    ......
}
```

即文件描述符表中存储的表项为实现了 File、Send、Sync 这三个接口的类型，而不局限于我们之前定义的 OSInode 。

目前同样实现 File 接口的有标准输入/输出与管道，可以查看同目录下 pipe.rs 与 stdio.rs 中的代码。

3.3.4 内核：部分系统调用说明

（1） 获取文件状态与目录项信息

sys_fstat 与 sys_getdents64 两个系统调用都是获取一些信息，两个所需的结构体与方法分别定义在 stat.rs 与 dirent.rs 中，其中使用了#[repr(C)]把结构体按照 C 语言的标准进行调整顺序、大小和对齐，以支持 C 语言程序的使用。

另外，位于内存模块中的 UserBuffer 也为这两个系统调用提供了支持，它是用户地址空间中一段缓冲区的抽象，通过它提供的 write 方法，内核可以方便地在用户地址空间进行写入数据。

（2） 挂载

由于目前系统中对挂载的需求不是很大，因此我们只设计了一个挂载表来记录挂载设备，它的定义如下：

```
lazy_static! {
    pub static ref MNT_TABLE: Arc<Mutex<MountTable>> = {
        let mnt_table = MountTable { mnt_list: Vec::new() };
        Arc::new(Mutex::new(mnt_table))
    };
}
```

3.3.5 FAT32：概述

由于 rCore-Tutorial-v3 中实现了一个简易的文件系统 easy-fs，其分层较为合理，因此 RongOS 所使用的 Simple-FAT32 文件系统在此基础上进行修改，使其符合 FAT32 文件系统的规格要求。

其次，Simple-FAT32 也借鉴了上一届 UltraOS 的 FAT32 实现，我们与 UltraOS 的开发者进行了积极的沟通与交流，对其一些不完善的地方进行改进，一些我们认为欠缺的地方进行重新设计。鉴于此，我们沿袭了其 FAT32 文件系统的名字，将我们的文件系统也命名为 Simple-FAT32。

图 3-8 描述了 Simple-FAT32 的分层结构。

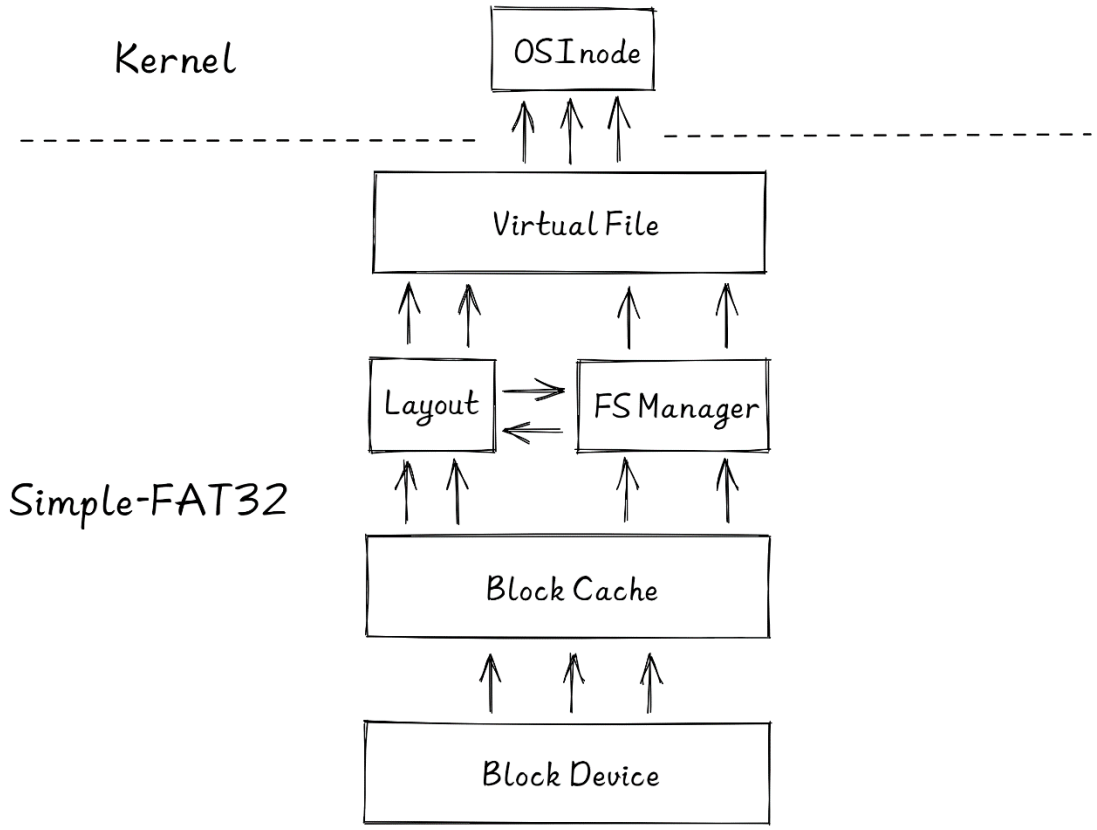

图 3-8 Simple-FAT32 分层结构

3.3.6 FAT32：磁盘块设备接口层

定义了磁盘块的读写接口。

该 trait 仅需求两个函数 read_block 和 write_block，分别代表将数据从块设备读到内存缓冲区中，或者将数据从内存缓冲区写回到块设备中，数据需要以块为单位进行读写。

```rust
pub trait BlockDevice: Send + Sync + Any {
  fn read_block(&self, block_id: usize, buf: &mut [u8]);
  fn write_block(&self, block_id: usize, buf: &[u8]);
  fn handle_irq(&self);
}
```

在本次比赛中，实现 BlockDevice trait 的有两种类型：

- SDCardWrapper：K210 硬件平台的 MircoSD 抽象，用于在开发板上测试。
- VirtIOBlock：VirtIO 总线架构下的块设备抽象，用于在 Qemu 模拟器调试使用。

3.3.7 FAT32：块缓存层

当操作系统频繁读写磁盘块时，由于硬盘读写速度相较于内存是数量级上的差距，因此系统整体性能会受硬盘读写速度的牵制而极大降低。解决该问题最常见的手段就是引入缓存，将使用的磁盘块读到内存的一个缓冲区中，这个缓冲区中的内容是可以直接读写的，那么后续对这个数据块的大部分访问就可以在内存中完成了，这样就可以大幅提高 I/O 速度。当这个磁盘块不需要再被使用时（通常是由于长时间未使用而被替换算法选中），可以根据其内容是否有更改（或写回标志位）来决定是否将这片缓冲区中的内容写回磁盘。

基于这种想法，Simple-FAT32 中引入了块缓存层，其主要由两个结构体组成：

- BlockCache：块缓存结构体，是物理存储介质中一个块的映射。
  - cache：长度为 BLOCK_SIZE 的字符数组，磁盘块内容的实际存储位置。
  - block_id：保存当前块缓存在物理存储介质的实际块号，用于判断重复和数据写回。
  - block_device：当前块缓存所属的块设备引用，用于以后处理多个块设备的情况。
  - modified：更改标记，若当前块缓存的内容有更改需要写回磁盘，则值为 True。
- BlockCacheManager：块缓存管理器，用于实现块缓存的分配与回收等功能。
  - start_sec：扇区偏移量，当磁盘有分区时，对块设备的读写需要加上这个值。

注：扇区是磁盘物理层面的概念，而磁盘块是文件系统逻辑层面的概念，但由于目前磁盘块大小和扇区大小都是 512 字节，因此我们简单地将扇区概念等同于磁盘块的概念，下文将不再强调二者区别

- limit：块缓存上限，当块缓存数量达到上限时，需要使用替换算法进行淘汰。
- queue：保存块缓存的双端队列，其中保存物理块号和对应块缓存的映射关系。

块缓存层对外主要提供 get_block_cache 接口，上层模块可以通过这个函数来获取对应物理块号的块缓存，随后调用块缓存的 read/modify 方法即可对存储设备的实际数据进行读取/修改，最后当块缓存的生命周期结束时，其会自动调用 sync 方法进行判断是否将内容写回磁盘。（此处采用了 RAII 的想法）

```
pub fn get_block_cache(block_id: usize, block_device: Arc<dyn
BlockDevice>) -> Arc<RwLock<BlockCache>> {...}

pub fn read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V
{...}

pub fn modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&mut T) ->
V) -> V {...}
```

3.3.8 FAT32：磁盘布局层

磁盘布局层是体现 FAT32 规范的关键，在这一层中会严格按照 FAT32 的标准进行数据结构的设计。此层我们使用的参考资料为：Microsoft Extensible Firmware Initiative FAT32 File System Specification(Version 1.03, December 6, 2000)

一个 FAT 文件系统卷由四个基本区域组成，它们按此顺序排列在卷上：

- 保留区域
- FAT 区域
- 根目录区域（注意：这不存在于 FAT32 卷中）
- 文件和目录数据区域

因此 Simple-FAT32 中 FAT32 文件系统的磁盘布局如图 3-9 所示：

图 3-9 FAT32 文件系统磁盘布局

（1） 保留区域

保留区域的第一个扇区是 BPB(BIOS Parameter Block)，有时又称引导扇区或第 0 扇区。

该扇区前 36 个字节是 FAT12、FAT16、FAT32 共有的 BPB 字段，从第 36 个字节起 FAT32 的扩展 BPB 字段将与 FAT12、FAT16 不同，直至第 90 个字节。而后便是引导代码，最后以 0xAA55 作为结束标记。

BPB 字段中较为重要的有：

- BPB_BytsPerSec：每个扇区所包含的字节数量。
- BPB_SecPerClus：每个簇所包含的扇区数量。
- BPB_RsvdSecCnt：保留扇区的数目，通过它能获得第一个 FAT 所在的扇区，在 FAT32 中为 32。
- BPB_NumFATs：FAT 文件系统中 FAT 的数量，通常为 2。

扩展 BPB 字段中较为重要的有：

- BPB_FATSz32：一个 FAT 所占的扇区数
- BPB_FSInfo：FsInfo 结构在保留区域中的扇区号，通常为 1。

对于 BPB 字段，Simple-FAT32 为其设计了完整的结构体，在读入时使用结构体映射的方式进行一一对应，由于这些变量只在最初读入 FAT32 文件系统信息时使用到，用完即释放，因此不会对内存造成非常大的负担。

由于在 FAT32 卷上 FAT 可以是一个非常大的数据结构，因此 FAT32 设计了一个用来存储最后已知空闲簇号的字段，这样在 FAT32 API 调用时就无需重复计算有多少空闲的空间，加快了 FAT32 的处理效率。这一字段最后演变为 FsInfo 结构体，通常被存储在保留区域的 1 号扇区内（从 0 计数）。

FsInfo 结构体中较为重要的有：

- FSI_Free_Count：FAT32 卷上最近已知的空闲簇计数。
- FSI_Nxt_Free：最后被分配的簇号。

以上两个字段会在 FAT32 管理器进行簇的分配与回收过程中使用。

（2） 文件分配表

文件分配表 (File Allocation Table，FAT) 是存储在存储介质上的表，它指示磁盘上所有数据簇的状态和位置。 它可以被认为是磁盘的“目录”，由于 FAT 的存在，文件的簇就可以不必在磁盘上彼此相邻。

Simple-FAT32 中对 FAT 结构体的设计十分简单，仅两个整型用于表明 FAT 的起始扇区。

```
pub struct FAT {
    fat1_sector: u32,
    fat2_sector: u32,
}
```

但我们为其实现了非常多的提供给上层的接口：

- get_free_cluster：搜索下一个可用簇
- get_next_cluster：查询当前簇的下一个簇
- set_next_cluster：设置当前簇的下一个簇
- get_cluster_at：获取某个簇链的第 i 个簇
- final_cluster：获取某个簇链的最后一个簇
- get_all_cluster_of：获取某个簇链从指定簇开始的所有簇
- count_claster_num：获取某个簇链从指定簇开始到结尾的簇的数量

（3） 目录与文件

FAT 目录是一个由 32 字节结构的线性列表组成的文件。根目录是唯一一个必须始终存在的特殊目录。对于 FAT12 和 FAT16，根目录位于磁盘上紧跟着最后一个 FAT 的固定位置，并且根据 BPB_RootEntCnt 的值具有固定的大小。对于 FAT32，根目录是可变大小的，并且是一个簇链，就像其他普通的目录一样。FAT32 卷上根目录的第一个簇存储在 BPB_RootClus 中。与其他普通的目录不同，任何 FAT 类型上的根目录本身没有任何日期或时间戳，没有文件名(除了隐含的文件名)，也不包含 . 和 .. 这两个特殊的应位于目录前两条的目录。

在 FAT 文件系统中，目录与在一个目录下的文件都称为常规文件，它们的区别仅仅体现在目录项的属性字段中，因此下文可能会将目录与文件这两个概念进行混用，读者只需明白它们只在一个字段上有差异即可。

短文件名目录项的定义如下：

```
pub struct ShortDirEntry {
    dir_name: [u8; 8],      // 短文件名
    dir_extension: [u8; 3], // 扩展名
    dir_attr: u8,           // 文件属性
    dir_ntres: u8,          // 保留给 Windows NT 使用
    dir_crt_time_tenth: u8, // 文件创建的时间戳
    dir_crt_time: u16,      // 文件创建的时间
```

```
    dir_crt_date: u16,      // 文件创建的日期
    dir_lst_acc_date: u16,  // 上一次访问日期
    dir_fst_clus_hi: u16,   // 文件起始簇号的高 16 位
    dir_wrt_time: u16,      // 上一次写入的时间
    dir_wrt_date: u16,      // 上一次写入的日期
    dir_fst_clus_lo: u16,   // 文件起始簇号的低 16 位
    dir_file_size: u32,     // 文件大小（以字节为单位）
}
```

由于内核目前对时间支持得不完善，因此 Simple-FAT32 中没有对文件的各种时间做处理。除了私有成员变量的访问与修改，短文件名目录项中较为重要的接口有（代码中有详细的过程注释）：

- get_pos：获取文件偏移量所在的簇、扇区和偏移
- read_at：以偏移量读取文件
- write_at：以偏移量写文件

长文件名目录项的定义如下：

```
pub struct LongDirEntry {
    ldir_ord: u8,              // 长文件名目录项的序列号
    ldir_name1: [u8; 10],      // 5 个字符
    ldir_attr: u8,             // 长文件名目录项标志
    ldir_type: u8,             // 如果为零，则表示目录项是长文件名的一部分
    ldir_chksum: u8,           // 根据对应短文件名计算出的校验值
    ldir_name2: [u8; 12],      // 6 个字符
    ldir_fst_clus_lo: [u8; 2], // 文件起始簇号
    ldir_name3: [u8; 4],       // 2 个字符
}
```

长文件名目录项与短文件名目录项大小一致，也是 32 字节。FAT32 对长文件名的定义是：文件名超出 8 个字节或扩展名超出 3 个字节。

长文件名目录项的特点：

- 为了低版本的 OS 或程序能正确读取长文件名文件，系统自动为所有长文件名文件创建了一个对应的短文件名，使对应数据既可以用长文件名寻址，也可以用短文件名寻址。不支持长文件名的 OS 或程序会忽略它认为不合法的长文件名字段，而支持长文件名的 OS 或程序则会以长文件名为显式项来记录和编辑，并隐藏短文件名。
- 长文件名的实现有赖于目录项偏移为 0xB 的属性字节（即 LDIR_Attr），当此字节的属性为：只读、隐藏、系统、卷标，即其值为 0xFH 时，DOS 和 WIN32 会认为其不合法而忽略其存在。这正是长文件名存在的依据。

- 系统将长文件名以 13 个字符为单位进行切割，每一组占据一个目录项。所以可能一个文件需要多个目录项，这时长文件名的各个目录项按倒序排列在目录表中，以防与其他文件名混淆。长文件名的第一部分距离短文件名目录项是最近的。
- 系统在存储长文件名时，总是先按倒序填充长文件名目录项，然后紧跟其对应的短文件名。长文件名中并不存储对应文件的文件开始簇、文件大小、各种时间和日期属性。文件的这些属性还是存放在短文件名目录项中，一个长文件名总是和其相应的短文件名一一对应，短文件名没有了长文件名还可以读，但长文件名如果没有对应的短文件名，不管什么系统都将忽略其存在，所以短文件名是至关重要的。

对于长短文件名的配对，长文件名和短文件名之间的联系光靠他们之间的位置关系维系是远远不够的。此时需要用到长文件名的 0xD 字节（即 LDIR_Chksum），此校验和是用短文件名的 11 个字符通过一种运算方式来得到的。系统根据相应的算法来确定相应的长文件名和短文件名是否匹配。如果通过短文件名计算出来的校验和与长文件名中的 0xD 偏移处数据不相等，那么系统不会将它们进行配对。

举一个简单的例子，假设一个文件的文件名为 The quick brown.fox，则其在 FAT32 中的布局如图 3-10 所示：

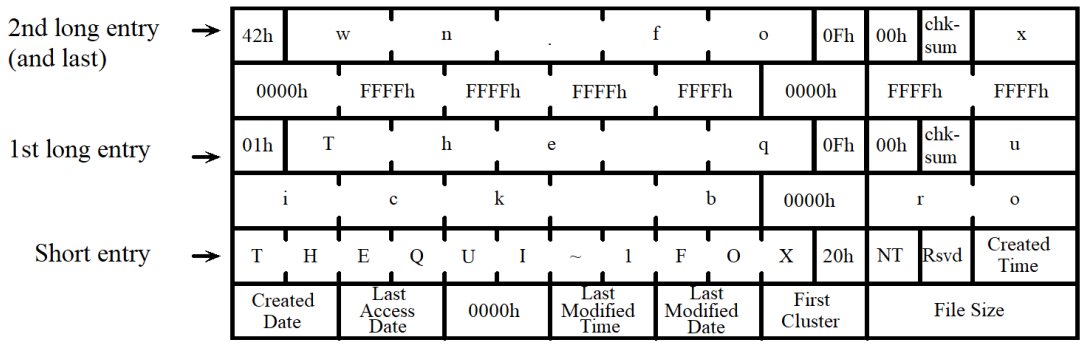

图 3-10 举例：The quick brown.fox 磁盘布局

3.3.9 FAT32：文件系统管理层

文件系统管理层的主要作用：

- 启动文件系统：读取 FAT 保留区域的数据，并进行校验，获取 FsInfo 与 FAT 的引用供其他层使用。

- 管理磁盘上的簇：负责簇的分配、回收与清空
- 提供工具函数：如长文件名拆分、根据给定的字节数量求出需要的簇数等

```rust
pub struct FAT32Manager {
    block_device: Arc<dyn BlockDevice>,       // 块设备的引用
    fsinfo: Arc<RwLock<FSInfo>>,              // 文件系统信息扇区的引用
    sectors_per_cluster: u32,                 // 每个簇的扇区数
    bytes_per_sector: u32,                    // 每个扇区的字节数
    bytes_per_cluster: u32,                   // 每个簇的字节数
    fat: Arc<RwLock<FAT>>,                    // FAT 的引用
    root_sec: u32,                            // 根目录所在簇号
    vroot_dirent: Arc<RwLock<ShortDirEntry>>, // 虚拟根目录项
}
```

FAT32Manager 提供的重要接口有：

- open：打开现有的 FAT32 文件系统
- alloc_cluster：分配指定数量的簇
- dealloc_cluster：回收给定的簇

另外计算相关的工具类函数有：

- first_sector_of_cluster：获取某个簇的第一个扇区
- size_to_clusters：根据给定的字节数量求出需要的簇数
- cluster_num_needed：计算扩大至 new_size 需要多少个簇

文件名相关的工具类函数有：

- split_name_ext：拆分文件名和后缀
- short_name_format：将短文件名格式化为目录项存储的内容
- generate_short_name：由长文件名生成短文件名
- long_name_split：将长文件名拆分，返回字符串数组

3.3.10 FAT32：虚拟文件层

可以不必过多关注此层的名称，原 easy-fs 中称为索引节点层，UltraOS 中称虚拟文件系统层。不论叫什么，这层的设计思想是为了让那些不关心磁盘布局具体实现的文件系统的使用者（操作系统内核）能便捷地对文件和目录进行操作，即屏蔽了文件系统的内部细节，抽象出一个虚拟文件提供给内核使用。

虚拟文件层主要设计了一个文件抽象，实现了文件创建、清空、删除、查找、

读取、写入等直接提供给操作系统内核调用的接口。

```
pub struct VFile {
    name: String,                       // 文件名
    short_sector: usize,                // 文件短目录项所在扇区
    short_offset: usize,                // 文件短目录项所在扇区的偏移
    long_pos_vec: Vec<(usize, usize)>,  // 长目录项的位置<sector, offset>
    attribute: u8,                      // 文件属性
    fs: Arc<RwLock<FAT32Manager>>,      // 文件系统引用
    block_device: Arc<dyn BlockDevice>, // 块设备引用
}
```

VFile 中有一些对下层接口的封装（都是私有函数，仅结构体内部使用）：

- read_short_dirent：获取短文件名目录项内容的不可变引用
- modify_short_dirent：获取短文件名目录项内容的可变引用
- modify_long_dirent：获得长文件名目录项内容的可变引用
- get_pos：获取文件偏移量所在的扇区和偏移
- first_cluster：获取文件的第一个簇
- increase_size：对文件进行扩容，new_size 是文件当前偏移量加 buf 长度

VFile 最终提供给文件系统使用者的接口（即 Simple-FAT32 目前能提供的所有功能）：

- name：获取文件名
- file_size：获取文件大小
- is_dir：判断文件是否为目录文件
- is_short：判断文件是否为短文件名文件
- read_at：以偏移量读取文件内容保存到缓冲区中
- write_at：以偏移量将缓冲区中的数据写入到文件中
- create：在当前目录下创建文件
- find_vfile_bypath：根据给定路径在当前目录下搜索文件
- clear：清空文件内容
- remove：删除文件
- stat：获取文件状态信息
- dirent_info：获取目录文件的目录项信息

最后，在内核的 inode 文件中，接入 Simple-FAT32 的类型、常量、函数如下，可以看出五级分层设计能很大程度上简化使用者的编程难度，几乎只需要接

入 VFile 这个类型。

```
use simple_fat32::{FAT32Manager, VFile, ATTR_ARCHIVE,
ATTR_DIRECTORY,create_root_vfile};
```

4 项目总结

4.1 比赛过程中的重要进展

- 完成 rCore-Tutorial-Book-v3 所有章节内容
- 搭建测试环境，在操作内核中运行 C 语言程序
- 实现进程管理三大基础系统调用（fork、exec、waitpid）
- 完成 FAT32 文件系统
- 加入自动测试程序，开始提交测试
- 操作系统内核成功对接 FAT32 文件系统
- 完成初赛所要求的全部系统调用

4.2 遇到的主要问题和解决方法

4.2.1 内存管理：关于系统调用数据拷贝时虚拟地址连续但物理地址不连续的问题

我们在测试系统调用时，发现用户程序通过调用 sys_uname 和 sys_gettime 系统调用时只回传了部分数据。

我们首先考虑编译器对结构体空间是否有不同的布局，因为我们的内核使用 Rust 编译但用户程序使用 C 编译且我们在测试基本数据类型时没有发现这一错误。我们通过 sizeof()函数测试了内核中的数据占用空间和用户程序中数据占用空间，发现结构体大小是相同的便排除了结构体布局的问题。后来我们意识到内核传递数据的方式是直接写入内核虚拟地址空间，内核虚拟地址空间采用的是恒等映射，即等同于直接写入物理地址，但是由于应用内存地址连续经地址映射后可能不连续，因此我们断定这是跨页访问的问题。

由于存在跨页访问数据传递问题，我们认为原有的内核向用户传递数据的接口已经过时，我们便对其进行改进，不再直接向物理地址写入数据，而是引入了

一个结构体 UserBuffer 用字节数组的引用表示用户空间一定长度的内存空间，在创建 UserBuffer 时我们判断了地址是否跨页，如果跨页需要从新软件模拟地址映射获取到物理地址，然后再获取对物理地址的引用。当我们需要向用户空间传递数据时，我们先通过 UserBuffer::new()创建一块字节数组引用，然后向这一引用写入数据即完成向用户空间传递数据。

4.2.2 文件系统：关于 dyn 类型不能转换为具体类型的问题

这是一个 Rust 语言特性的问题，由于 trait 的设计想法就是以一种抽象的方式定义共享的行为，因此从一个具体类型变为该 dyn 后就无法再使用自己”独特“的方法，这个问题困扰了我们很久。

在早期开发中，我们从文件描述符表中取出一个普通文件，我们理所当然地把它当成 OSInode 来使用，但是 Rust 认为它是 dyn File + Send + Sync 类型，因此不允许我们使用定义在 OSInode 中的方法，这会造成非常大的不便。

个人猜测 UltraOS 也遇见过这种问题，他们的做法是设计了一个文件枚举类进行区分处理：

```
// UltraOS's codes
pub enum FileClass {
    File (Arc<OSInode>),
    Abstr (Arc<dyn File + Send + Sync>),
}
pub struct FileDescripter{
    cloexec: bool,
    pub fclass: FileClass,
}
pub type FdTable =  Vec<Option<FileDescripter>>;
pub struct TaskControlBlockInner {
    ...
    pub fd_table: FdTable,
    ...
}
```

可以看出他们为了解决这个问题，相对于 rCore-Tutorial-v3 新增了很多类型。

我们最开始的想法是使用 unsafe 语句，用最原始的裸指针指向那片数据的地址，然后进行强制类型转换，但实施起来并没有那么简单（可能是因为我们对 Rust 的熟练度不够）最后我们放弃了这个方法。（其实这种不安全的做法也是 Rust 不推崇的）

目前 RongOS 是做出了一些“妥协”，我们把为了实现系统调用的文件描述符表中的 OSInode 所需的部分方法（即 get_fstat 与 get_dirent）也定义在了 File trait 中，这会显得它臃肿，但我们觉得也还算合理，毕竟 pipe 文件也可以使用有状态信息。

至于 get_name 的设计，是因为我们发现在系统调用中其实只是为了使用 OSInode 所对应的 VFile 对象，而 VFile 提供的 find_vfile_bypath 可以根据文件路径返回 VFile 对象，所以我们先在 OSInode 中实现了 name 方法，然后在 File trait 添加了 get_name，以此完成 dyn File + Send + Sync 到 VFile 的转换。

这一做法即保留了“一切皆是文件”的思想，也大幅减少了相关系统调用的代码行数。

4.2.3 文件系统：关于文件名大小写的问题

我们在实际测试中发现，Linux 对于短文件名文件也会创建一个长文件名目录项，以此来区分文件名的大小写（因为短文件名目录项中只存放大写，就无法判断字符的大小写情况，而微软使用了 DIR_NTRes 这一字段来标明文件名的大小写情况）。Simple-FAT32 中没有对文件名大小写进行检查，因此也不会为短文件名文件自动创建长文件名目录项，这样的做法使得 Simple-FAT32“兼容”了 Linux 和微软的两种标准。

4.3 项目特征描述

- 使用 Rust 语言编写
- 支持 79 条系统调用
- 独立的应用地址空间和内核地址空间
- 适配 SV39 多级页表、使用 Lazy、COW 机制
- 支持虚拟文件系统，适配自制的 FAT32 文件系统
- 支持 ELF 格式的静态/动态链接程序
- 通过 lmbench 基准测试，支持 busybox、lua 等程序
- 支持信号机制的基于时间片轮转调度策略的进程管理机制
- 丰富的调试方案：不同等级的调试信息输出、使用编译参数方便调试
- 详细的项目文档和 doc comment 格式代码注释

5 决赛第一阶段

5.1 内容简述

决赛第一阶段使用 musl 和 libc-test 作为赛题。与初赛相同，编译好的 elf 文件存放在 SD 卡根目录中，需要内核初始化完成后主动依次运行。我们的系统需要支持 musl 库，且要支持动态链接，并通过 libc-test 的测试。

5.2 遇到的主要问题和解决方法

5.2.1 用户栈初始化需要符合 systemv 标准

runtest.exe 测试程序进不了 main 函数，错误被 trap_handle 捕捉到后为访存错误。使用 gdb 调试时发现在用户程序发起 exec 系统调用后正常返回内核，当 syscall 结束后进行 trap 返回，在 trap 返回的最后一句 sret 后程序出错，故怀疑是 sret 执行的相关环境问题（如 sepc 寄存器等）。但想了很久并没有找出错误，后来调试中发现，当程序控制流从 rust 的内核转向 c 语言的用户程序时，gdb 控制不了程序的单步进行，因此有可能是 sret 后面的语句出错，其表现为执行 sret 后程序出错，因此这里会有一个思维误区。所幸我们发现了这一点，后续分析应该是 libc 初始化阶段（即 init_libc 函数）出错，继而寻找有关标准与规范，看到了用户栈的初始化内容（如图 5-1 所示），发现与我们现有的并不相符，因此加以修改解决了问题。

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 eightbyte
Auxiliary vector entries ...		2 eightbytes each
0		eightbyte
Environment pointers ...		1 eightbyte each
0	`8+8*argc+%rsp`	eightbyte
Argument pointers	`8+%rsp`	argc eightbytes
Argument count	`%rsp`	eightbyte
Undefined	Low Addresses	

图 5-1 初始化程序栈

5.2.2 程序加载时数据拷贝对齐问题

当用户程序跑到__libc_start_init 函数时，在一个循环中执行出错，其涉及__init_array_start 字段，经过调试后猜测是程序段加载出错，严格实践与思考后确定问题为程序第二段加载时会有数据的拷贝，但没有考虑对齐的问题，即我们默认程序段和地址是按页对齐的，然而 runtest.exe 并没有（如图 5-2 所示）。因此我们手动添加了一个修正量 offset 进行页对齐，解决了这个问题。

$$offset = start\_virtual\_address \ \% \ PAGE\_SIZE$$

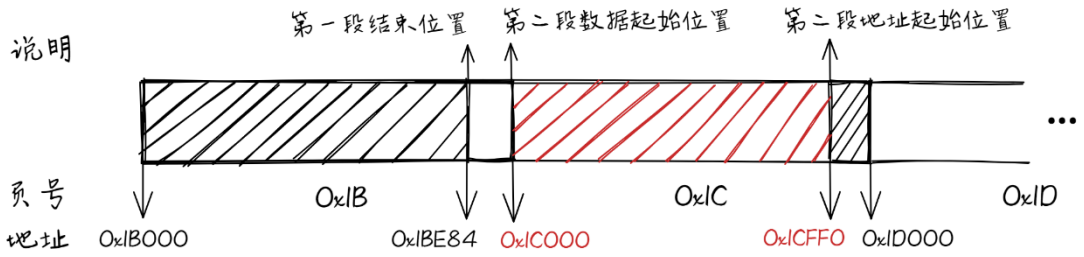

图 5-2 runtest.exe 加载位置

5.2.3 动态内存分配出错

成功运行 runtest.exe 后，在加载 entry-static.exe 时内核抛出异常：堆分配错误，分配大小为 1048576（1M）。使用 gdb 和输出调试定位了很久，终于发现是在读取 elf 文件的时候产生的堆分配，rust 中的 Vec 数据结构扩容的策略是乘 2 扩大，因此当容量为 512K 仍不够时，它会申请 1M 的空间。RongOS 目前的内核堆空间设定为 2M，可能由于其他函数中还有没有释放掉的变量在堆空间中，因此导致堆空间不足的情况。由于内存资源紧缺，我们只将堆空间扩大至 3M，目前运行中不会产生内核堆空间不足问题。但这种暴力扩大堆空间的做法并不值得提倡，我们还在尝试一种可以充分利用堆空间的分段加载方式。（但由于我们对 entry-static.exe 这个大文件做了缓存，使得运行下一个测试用例时无需重复读取文件，其缓存数据保留在堆中，因此很难取舍。）

若只谈论解决方法，则这个问题不值得写在文档中，但其涉及了很多其他方面的点，如堆分配器（分配策略）、一个数据类型的扩容策略等完全可以自定义的内容，因此做一下记录。

5.2.4 动态加载

有关动态加载的知识，我们主要参照了《程序员的自我修养--链接、装载与库》这本书。由决赛第一阶段内容简述可以得知，libc.so 是一个动态链接器，因此我们只需要加载并运行 libc.so 即可，它会进行自举、装载共享对象、重定位和初始化等工作。

内核在加载 ELF 程序时，会检查 ELF 程序段中是否有 INTERP 类型，如果有则还需额外地加载其指定的动态链接器（根据比赛的说明，这里我们直接加载 libc.so）。将动态链接器加载完成后，我们需要传入其需要的辅助参数（AT_BASE、AT_PHENT、AT_ENTRY 等）以及在环境变量中添加动态库的存放路径。与加载静态程序不同的是，加载动态程序后，用户程序的入口地址应该是动态链接器的地址而不是动态程序，即先从内核跳转到动态链接器，再从动态链接器跳转到应用程序。

6 决赛第二阶段

6.1 内容简述

决赛第二阶段要求我们的系统需要尽力支持 busybox, lua, lmbench 三个程序的多种执行方式形成的测试用例。在功能完整的情况下，尽可能优化系统性能，以在 lmbench 测试程序中取得更高的分数。

本届比赛还有一个可选的扩展内容，即支持 Alpine Linux 中的一些应用程序。

6.2 遇到的主要问题和解决方法

6.2.1 在命令行中使用管道出现内存不足问题

进入用户程序后，我们的内核最初有 1044 页供用户程序分配使用，大小略多于 4M，加载完 busybox sh 后仍有 732 页空闲，大小约为 2.85M。但在运行 busybox echo "1" | busybox cat 命令后，会出现内存不足现象，且更加奇怪的是这个现象是具有随机性的。

我们主观上觉得 2.85M 的空间足够两个 busybox 运行，但实际上还是失败了。通过不断地打印内核运行过程和内存使用情况，我们找到了问题的根源：运行程序时为先 fork 再 exec，在 fork 过程中内核会复制父进程的内存空间，我们在这里是全部分配物理页的，然后在 exec 正常加载后，原来那份父进程的内存副本才会被释放掉。

用一个测试数据可以更清楚地说明这一问题：

（为了便捷，我们使用 A 代表 busybox echo “1”，B 代表 busybox cat。简单认为 fork(A)是指父进程 fork 后准备加载 A 程序）

运行阶段	物理页帧使用情况
busybox sh 运行成功，等待输入	356
输入 busybox echo "1" | busybox cat	
fork(A)	668
exec(A)	969
原 fork(A)的内存副本释放后	661

fork(B)	975
exec(B)	出现内存不足情况

我们计划使用 lazy 加载和 cow 来解决这一问题。（实际上在很早的时候就应该使用这两种机制，但由于内存的充足一直拖到了现在）

再说说随机性的问题，调试后发现这是由于程序调度导致的，主要有以下几种情况（可能不全面）

1、fork(A)-fork(B) -exec(A)- exec(B)，这在 exec(A)时就出现内存不足情况

2、fork(A)-exec(A)-fork(B)-exec (B)，这在最后的 exec(B)时会出现内存不足情况

3、fork(A)- exec(A)-exit(A)-fork(B)-exec(B) ，由于 A 的 exit 导致其内存资源被回收，因此这种情况下 exec(B)会有足够的内存加载 B 程序，从而可以正常运行 A | B 命令。

6.2.2 程序在调用 mmap 后出现访存错误的问题

我们发现程序在调用 mmap 时指定了 mmap 的起始地址，且这一段地址不在内核规定的默认 mmap 范围内，通过打印系统调用时传入的参数，我们发现程序指定的虚拟地址包含于现有的逻辑段中，且 mmap flag 中包含有 MAP_FIXED 标记，在重新查询了 Linux man 中关于 mmap 标记的定义后，我们发现我们的 sys_mmap 函数没有实现对 MAX_FIX 的支持。为实现此功能，我们为 MemorySet 结构体新增了 check_va_range() 和 reduce_chunk_range() 函数分别用于实现检查虚拟地址范围是否包含在已有逻辑段中和根据需要插入的虚拟地址范围缩减已有的逻辑段地址范围，修改了 sys_mmap() 函数逻辑：若发现 flag 中包含 MAP_FIXED 则缩减相关逻辑段范围为待映射空间腾出空间，其余流程不变。

6.2.3 程序“跑飞”：pc 值超出代码段范围

在调试过程中时常会遇到内核 Trap 到非法（Illegal Instruction）、指令页加载错误（InstructionPageFault）；还有可能 pc 值正常，但访问地址不在用户空间逻辑段范围内而被内核 Trap 到 PageFault 的问题。

在排除用户程序本身的逻辑问题以及由 PageFault 触发的 Lazy 加载或 COW 加载的情况后我们发现 pc 值或者访存地址均由内存中的值增加偏移量后得出，

由此我们怀疑是内存中的数据未得到正确加载，在通过 debug_show_data()函数按页打印内存中的数据并与 ELF 文件相对比，发现内存中只有部分数据被正确加载，而其余部分数据均为 0。检查 Lazy 加载函数发现只有发生缺页异常的地址之后到页尾的数据会被加载，于是便将 Lazy 加载的起始地址从发生异常的地址改为发生异常地址对应的页的首地址，以实现整页加载。

在定位问题的过程中，我们还意识到在解决 6.2.2 处的问题时单纯通过缩小逻辑段范围实现逻辑段缩小存在不足。即缩减逻辑段范围并不会修改页表，使被缩减段内存空间失效，若被缩减段已经被加载，那么在 mmap 之后再去访问该地址不会触发缺页异常，数据也还是上一逻辑段中的数据未被修改。要修复此问题可以在缩减地址范围的同时对已分配物理页帧且虚拟地址不在缩减后地址范围内的物理页进行 unmap()，使被缩减的空间重新变为无效状态，在新逻辑段被映射后程序访问改区域便会触发缺页异常，数据便可得到正确加载。

7 尾声

7.1 分工和协作

林杰克：负责 FAT32 文件系统与内核文件系统相关模块，编写测试环境

莫佳洋：负责进程管理、内存管理模块，以及比赛其他的系统调用

7.2 提交仓库目录与文件描述

仓库地址：https://gitlab.eduxiji.net/19061120/oskernel2022-segmentfault

目录描述：

- bootloader：运行于 QEMU 与 k210 两个平台的 SBI 二进制文件
- doc：项目文档
- os：内核代码
  - boards：定义了一些与运行平台相关的参数
  - drivers：一些底层驱动代码
  - fs：虚拟文件系统代码
  - mm：内存管理相关代码
  - sync：同步与互斥访问代码
  - syscall：系统调用相关代码
  - task：进程管理相关代码
  - trap：RISCV 自陷相关代码
- simple-fat32：FAT32 文件系统代码
- test_program：比赛所使用的测试程序与评判程序源码
- user_c：自定义的使用 C 语言编写的测试程序

7.3 比赛收获

林杰克：这次比赛给予了我极大的促进力，充实了课程非常少的大三下生活。通过学习、独立编写一个操作系统内核，我加深了对操作系统各种概念的理解以及懂得了它们从何而来，它们出现是为了解决怎么样的问题。另外，对 FAT32 规

范的学习不仅让我掌握了 FAT32 文件系统，还在另一个角度教会了我应该如何去搜寻一份规范文档，如何去系统地学习一个现有标准。还有两个意外之喜：一是学习了 Rust 语言，它严格的编译检查让我非常喜欢它，现在它已经成为了我的主要开发语言。二是分工协作能力的提升，在比赛期间频繁地使用 git 命令，让我对版本控制、代码合并等操作越发得熟练。

莫佳洋：通过这次内核实现赛，我在前人的基础上开发一个全新的操作系统内核实在是收获颇丰，这一过程中我了解了一台计算机启动的细节，以及特权级转换的过程。在内核代码的编写过程中，我不断在摸索中学习了一门新的语言 Rust，完成了从读懂到会写的过程。在这次比赛中我主要开发内存管理模块和进程管理模块，期间我第一次把在大学期间学习到的知识在代码编写中运用。在编写期间也遇到了不少问题和 bug，期间少不了队友、其他队伍同学以及知道老师的帮助和启发，对此我深表谢意。这一经历让我不仅将已有知识学以致用还让我学习到了新的知识，也提高了我合作开发的能力，我收益匪浅。