哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

# 基于 RISC-V64 的多核操作系统

# jkxsOS

健康向上好青年项目组
项目成员：丁浩卓 郑启洋 陈林锟
指导老师：夏文 仇洁婷

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

2022 年 8 月

# 目录

# 1 概述

## 1.1 项目整体情况

jkxs-OS 致力于开发一个能在 RISCV-64 处理器上运行的宏内核操作系统。我们以清华大学吴一凡同学的教学项目 rCore-Tutorial 为基础，在其代码框架上进行迭代开发。

之所以选择 rCore-Tutorial，一是它的开发语言——Rust 在系统编程领域具有得天独厚的优势，它天然地保证了程序的内存安全和线程安全，能够帮助我们规避内核开发中的诸多潜在问题；二是 rCore-Tutorial 是一个较为完整的系统，具有良好的可拓展性，基于 rCore-Tutorial 开发能够避免“重复造轮子”的麻烦，节省了在线程上下文切换、页表访问等大量细节的实现上需要花费的时间，使我们能将更多的精力投入到内核性能的优化、健壮性的增强、用户体验的提升等环节上，做出更有意义、更具有创新性的工作。

对于基于 riscv-64 体系结构的操作系统内核实现，上一届代表哈尔滨工业大学（深圳）参赛的 UltraOS 队伍已经进行了相当多的探索。作为本科生小组独立开发完成的项目，UltraOS 无疑是杰出的。它不仅实现了大赛的全部功能要求，还对系统性能进行了针对性的优化，且具有诸多亮点（创造性的 Monitor 调试模块、初始进程和 shell 的回收、kmem 设计、相当完善的文档等）。UltraOS 的精妙设计给我们提供了许多灵感，有了 UltraOS 的重要探索，可以说我们已经“站在了巨人的肩膀上”。不过，我们不希望 jkxs-OS 仅仅成为一个“UltraOS 的翻版”；我们希望 jkxs-OS 能够比“巨人”站得更高、看的更远——具体来说，就是拥有比 UltraOS 更完善的硬件支持、更好的性能、更优雅的设计、更完善的文档、更好的用户体验。

截止比赛结束，我们具体工作如下：

1. 内核运行的必要模块

进程调度。我们使用进程控制块(PCB)作为用户进程的抽象，在其上记录了进程运行所需要的一系列关键信息；为每个 CPU 核抽象出一个任务处理器 Processor，记

录当前在其上执行的进程。我们还创建了全局任务管理器 Task Manager，用于记录正在排队等待运行的进程，便于 Processor 进行任务的调度。

内存管理。我们对内核和用户拥有的内存空间统一抽象成一个类 MemorySet，用于方便地实现对内存的管理。它对外提供了完善的接口，避免了外界对页表的直接操作；我们还扩展了 MemorySet 的接口，使其能支持 mmap 和 heap 区域的管理；最后，我们实现了 Lazy Allocation 机制，避免一次分配过大的内存，从而减少内存的浪费。

2. 多核运行支持

我们对内核中所用的全局变量(PCB, Frame Allocator, Task Manager, Console 等)添加了读写锁(RwLock)，确保了线程安全。互斥锁(Mutex Lock)相比，读写锁支持多核并发的读操作，具有更好的并行度。

文件系统模块也对 FAT、文件系统管理器，块缓存等数据结构加锁，保证了多核并行条件下程序的正确性。

3. FAT32 文件系统

我们采用五层的分层结构来设计 FAT32 文件系统，从下到上分别为磁盘块设备接口层、块缓存层、磁盘布局层、文件系统管理层、虚拟文件系统层。

我们沿袭 rCore-Tutorial 的松耦合模块化设计思路，将 FAT32 文件系统从内核中分离出来，形成一个独立的 Cargo crate。这样，我们就可以单独对文件系统进行用户态测试。在用户态测试完毕后，可直接放到内核中，形成有文件系统支持的新内核。

4. 现实程序运行支持

rCore-Tutorial 已经提供了 27 条系统调用，但它们有些是冗余的（如线程、信号量相关的系统调用，我们的内核实现暂时不需要这些功能），且大多数不符合 POSIX 标准。这使得原生的 rCore-Tutorial 不能通过 Online Judge 平台的评测，更不可能支持 busybox 这类复杂的应用程序。因此，我们修改了并扩展了系统调用的接口，实现了 syscall 的规范化。

为更好地为现实程序的运行提供支持，我们将系统调用的数量扩充至 68 个，也

相应地实现了一些机制来支持这些 syscall 的执行（如 mmap 机制、信号机制等）

部分复杂的应用程序（如 busybox）需要实现更完善的内核机制才能运行。为此我们也做了很多工作，如扩展 exec 系统调用，将程序运行的必要参数提前压入用户栈；又如开启浮点运算机制、实现进程上下文切换时 tp 寄存器的保存与恢复等。

目前，我们能够支持 busybox、lua、lmbench、libc-test 等大赛要求的现实程序。不仅如此，我们也移植了一个精简版的 rootfs，并在此基础上支持 sh、vi 等 Linux 原生用户程序。更进一步的，我们移植了 gcc，从而支持 C 语言程序的编译与执行，打造了一套贴近现实的用户编辑、工作环境。

5. SBI 支持与多核启动

在 qemu 上， 可使用 Rust-SBI 或 Open-SBI 支持内核运行, 并在此基础上实现多核启动；

在 K210 上, 可使用 Rust-SBI 支持内核运行；

在 FU740 上, 可使用 Open-SBI 支持内核运行, 并在此基础上实现多核启动。

6. 多硬件平台支持

我们在实践中发现，Qemu 提供的虚拟硬件环境和真实开发板有诸多细微的差别，这使得能够在 Qemu 上稳定运行的程序在真实开发板上经常出现各种各样的问题（例如，lazy_static!宏无法在 Hifive Unmatched 开发板上正常工作，需要更换为 Lazy Cell）。因此，对具体开发板的硬件环境进行适配，是我们必须完成的工作。

目前，我们的内核已经完成了对 K210 和 Hifive Unmatched 的适配，并且均在在线评测平台上获得满分。对于 Hifive Unmatched，我们完成了更多的工作，包括 MMIO 的调整、SDCard 驱动的开发与测试等。

7. 更便捷的调试功能

我们编写了 Monitor 模块，帮助我们对内核进行更方便的调试。它的基本原理是预留专用内存区域，以其内部数据作为内核调试输出开关。这样，通过 GDB 修改对应内存的值，就可以达到控制调试目标、输出粒度、调试信息输出是否开启等各类参数的效果。

系统调用追踪是一类极为常用的调试手段，我们的内核也实现了这一功能。内核可为用户反馈系统调用 id、名称、输入参数、返回值等信息，用户也可以在 user_shell 中通过 trace 命令控制 syscall 调试信息的输出开关。

8. 更好的用户体验

虽然大赛需要的测试程序不需要 user_shell 也可以直接执行，但是如果缺少了 shell，操作系统就失去了与用户交互的能力，而且也为调试带来了困难。rCore-Tutorial 已有一个简单的 shell 实现，但这个 shell 使用起来有相当多的不便之处，比如使用左右键命令会出错、使用上下键光标会跑飞，使得命令的输入相当不人性化。

基于此，我们实现了一个功能更加强大、且用户体验更好的 shell。它不仅解决了上下左右键按下时命令出错、光标跑飞等问题，还支持 tab 命令补全、命令历史回溯等功能，为调试带来极大便利。

我们实现了一套较为完整的日志系统，用户可选择最小日志输出等级，并通过 debug!，info!，warning!，error!等宏打印不同输出等级的日志信息。日志系统能为不同等级的输出信息设置不同的颜色，且能显示输出语句所在文件的名称和所在行数，使内核输出更加清晰、直观，更利于调试。

# 2 系统设计

## 2.1 系统整体架构设计

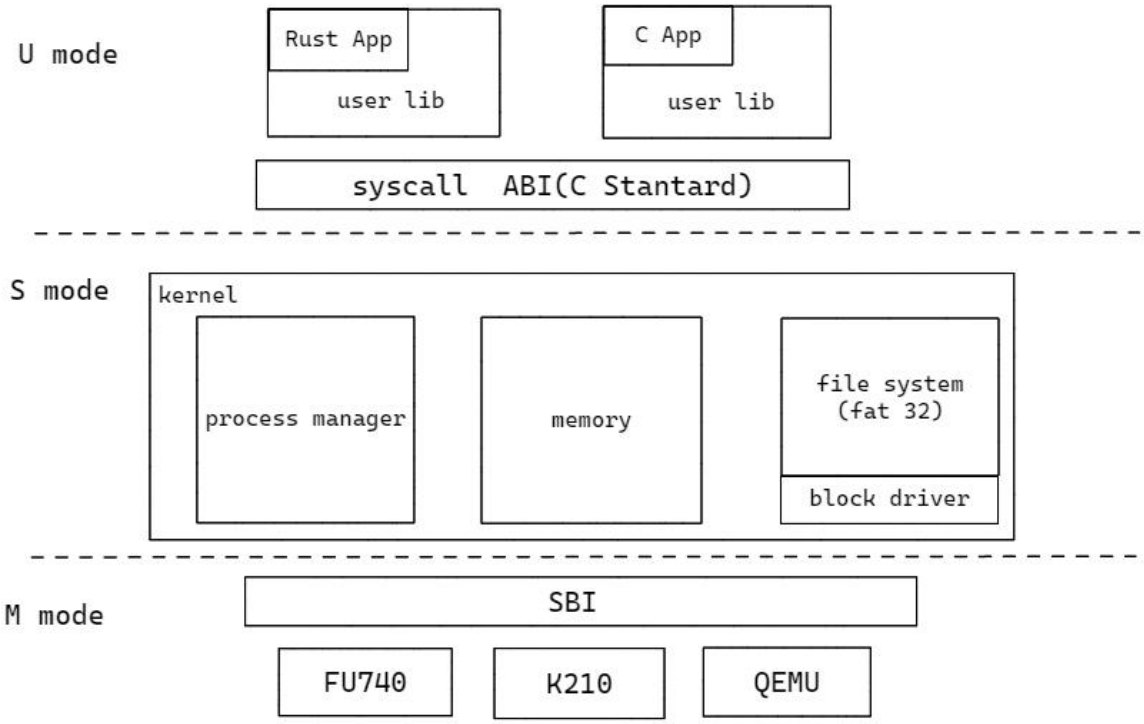


图 1　jkxs-OS 系统简要架构图

本系统设计与开发采用模块化方法，各模块之间并行开发。基于 RISC-V 架构特点，我们开发的重点为基于 SBI 在 S mode 下运行的内核模块，同时需要兼顾 U mode 下用户程序编写体验以及硬件平台相关的驱动实现。接下来就以 SBI 与硬件驱动、内核模块、用户程序支持的顺序简单介绍 jkxs-OS 系统。

SBI(Supervisor Binary Interface)是运行在 S 态的程序与 S 态运行时环境的接口，可以类比于 Linux 中的系统调用。简单而言就是提供了一个 M 态为 S 态提供服务的标准接口，为我们实现了部分硬件抽象。使得我们在开发 S 态程序(kernel)可以少考虑一些硬件平台之间的差异。在本系统中，我们支持使用 Rust-SBI 或者 Open-SBI 去为我们的内核提供运行时服务。

对于硬件驱动并不在 SBI 的范畴，各个硬件厂商对于相同协议的硬件实现可能存在一定的差异。所以我们需要在内核中完成对 fu740/k210 的硬件驱动模块，这其中

包括 SDcard 驱动、时钟控制驱动等模块。

Kernel，作为我们系统中最核心的部分主要包含了三个部分进程管理、内存管理、文件系统。进程管理主要实现了进程的状态控制、切换与调度。内存管理部分主要实现了页表管理、内存隔离和动态内存分配。文件系统基于块设备实现了 FAT 32 文件系统，支持对持久化存储设备进行修改与读取。同时在内核模块我们还基于 SBI 实现多核启动，并支持任务的多核调度。

支持用户态程序是内核的主要目的，所以我们对原始的系统调用进行封装，为用户态程序提供更加方便好用的用户函数库(user lib)。再此基础上，为进一步提高用户体验，我们实现了功能较为完善的命令行程序(shell)。

# 3 系统设计与实现

## 3.1 进程管理

### 3.1.1 进程相关的基本数据结构

(1) 进程控制块 PCB

进程控制块（Process Control Block, PCB）作为用户进程的抽象，其上记录了进程运行所需要的一些列关键信息。

PCB 分为两部分：一部分是进程号 pid，其在进程创建之后就不再改变了，不需要对其加锁。其余部分是可变的进程信息，包括进程的页表信息、父子关系、退出码、当前工作路径、文件描述符表、信号信息、堆和 mmap 信息、所拥有的线程等，它们需要用互斥锁保护起来。

PCB 是进程存在的唯一标识，每当进程创建时，PCB 也随之创建；而每当进程结束时，PCB 也随之销毁。

(2) 线程控制块 TCB

线程控制块(Task Control Block, TCB)记录线程运行的一些信息，是线程存在的唯一标识。线程是 CPU 进行调度的基本单位（在调度的语境下，本文档中“线程”和“任

务”可以等同），而进程可以被理解为线程的“容器”，不能被直接调度。线程拥有独立的内核栈、用户栈及 Trap 上下文、任务调度上下文，除此以外没有独立的内存空间。

同样地，TCB 也可以被分为不变和可变的两部分。不变的部分为线程的父进程指针、内核栈（这部分不需加锁），可变部分为 TCB 资源结构体、TrapContext 所在物理页帧号、任务上下文、线程状态、退出码。其中，TCB 资源结构体（TaskUserRes）与线程所拥有的的资源（用户栈内存、TrapContext 内存）相绑定：TaskUserRes 创建时，页帧分配器分配内存，PCB 将线程资源内存注册到用户页表中；而 TaskUserRes 销毁时，页帧分配器回收内存，用户页表自动删除资源内存的对应表项。

在 jkxs-OS 的实现中，虽然每个进程都有且只有一个线程（这意味着线程和进程的概念可以等同起来，理论上不引入线程概念，即只有 PCB，没有 TCB 也可以），但我们还是选择将 PCB 的一部分内容移入 TCB 中。这种做法使得 PCB 的结构不至于太过臃肿，也有利于提高内核代码的可扩展性，方便日后引入多线程。

(3) 任务管理器 TaskManager

任务管理器（Task Manager）是专门用于管理就绪线程的数据结构。其内部实现了一个队列，每当一个线程转变成就绪态时，就调用 TaskManager 的 add 方法将其加入就绪队列尾部；每当一个任务处理器 Processor 空闲时，就可以从就绪队列首部取出一个线程，进行调度，线程变成运行态。这种设计方式非常适合应用于时间片轮转调度算法（Round Robin，RR）和先来先服务调度算法（First Come First Serve，FCFS）。

(4) 任务处理器 Processor

任务处理器（Processor）可被认为是 CPU 核的抽象，每一个 CPU 核心对应一个 Processor，内核可以以 tp 寄存器为索引，获取当前核心对应的 Processor。Processor 保存了当前在其上运行的线程的一个引用，以及当前 idle 执行流的上下文。通过 Processor，内核可以方便地获取 CPU 核执行任务的状态。

### 3.1.2 进程的创建与结束

(1) 进程的创建

我们已经知道，PCB 是进程存在的唯一标识，因此进程的创建实际上就是 PCB 的创建。PCB 有三种创建方法：

- 简单创建方法。该方法中，我们只需给定用户程序的全部 ELF 数据，即可新建一个 PCB。这一过程中，内核创建了进程专属的内存空间结构体 MemorySet，为进程分配了内存，并在用户页表中完成了必要的地址映射；内核也对 PCB 的其他成员进行了简单的初始化，如分配 PID、设置初始工作路径、设置初始文件描述符表等。但是，这一方法不能维护新进程与已有进程间的父子关系，因此只适用于初始进程 initproc 的创建。
- fork 创建方法。该方法在已有的 PCB 上执行，用于复制出一个几乎一模一样的 PCB，新的 PCB 拥有自己的内核栈、用户栈、PID 等资源。这个过程中，内核需要为新 PCB 分配相应内存空间，并将原 PCB 中用户空间的数据复制到新 PCB 的用户空间中，还需维护新 PCB 与原 PCB 的父子关系（原 PCB 为父，新 PCB 为子）。
- exec 方法。这一方法常与 fork 结合使用，它实际上并未创建出新的 PCB，而是对原 PCB 的内容进行全面的替换。PCB 在执行 exec 后，获得了全新的 MemorySet、全新的用户栈、全新的文件描述符表等。为对现实程序提供更好的支持，我们对 exec 进行了扩展，使其能将程序运行所需的自定义参数提前压入用户栈中。

进程创建时，还会创建一个线程。内核会为此线程分配用户栈、内核栈、TrapContext、TaskContext 等内容。内核将 TaskContext 中的 ra 置为 trap_return 地址，sp 置为内核栈底的地址，这样，下次线程调度时就能将跳转到中断返回处执行，且能切换为新线程的内核栈。内核还会将 TrapContext 的 x0 设定为指定值，表示 fork 或 exec 的返回值；将 TrapContext 的 sepc 设定为指定值（fork 中为父进程的主线程 sepc，exec 中为 elf 头部设定的程序执行起始地址），使得 Trap 处理结束后 cpu 能回到正确的地方执行程序。

(2) 进程的结束

一个进程结束时，内核并不直接销毁其 PCB，而是设置其状态为 zombie，并设置其退出码，然后释放进程拥有的大部分资源。此时，还要将该进程的所有子进程挂在 initproc 之下，等待 initproc 的回收。如果内核直接销毁 PCB，那么该进程的父进程就无法获得其退出码，进程退出的原因也就无从知晓了。

内核可以真正销毁 PCB 的时机在 waitpid 系统调用中。waitpid 中，父进程找到指定的已退出子进程，获取其退出码后，就可以彻底释放子进程的 PCB 了。

### 3.1.3 任务（线程）调度的基本思路

我们在 jkxs-OS 中，采取时间片轮转调度算法（Round Robin，RR）对线程进行调度。由于目前线程和进程是一一对应的，故线程的切换也会引起进程的切换。

内核结合 idle 控制流和任务上下文实现线程的切换。在一个 Processor 开始调度时，就进入了一个等待就绪任务的无限循环中，这个无限循环的执行流就是 idle 控制流。每当 Processor 发现一个就绪的线程，就保存当前的 idle 控制流上下文，然后恢复目标任务的任务上下文，并切换到对应任务的内核栈中；如果没有找到就绪任务，则在 idle 控制流中继续循环。而当一个任务执行结束之后（可能是任务已经退出，或者被暂停），则 Processor 会保存任务的上下文，转而切换到 idle 控制流和 idle 栈中。

引入 idle 控制流虽然引入了一定的开销，但是它让各执行流的分工更加明确了，同时也使最初的引导栈（Boot Stack）空间得到充分的利用。

## 3.2 内存管理

### 3.2.1 内存管理相关数据结构

(1) 内存空间结构体 MemorySet

内存空间结构体（MemorySet）可被看作内核或用户拥有的页表、内存空间及内存信息的统一抽象。它对外提供了完善的接口，使外界能够对内核或进程的内存空

间进行方便的管理。

MemorySet 的成员有：页表 page_table、所有数据页帧区域 areas、所有堆内存页帧 heap_frames、所有 mmap 内存页帧 mmap_areas。MemorySet 在这些成员上定义了一系列操作，包括页表的映射和取消映射、内存区域（areas 或 heap_frames 或 mmap_areas）的创建和删除、获取页表 SATP 值、用指定页表对虚拟地址进行翻译等。

MemorySet 的设计利用了 RAII 的思想，即“资源获取就是初始化”。当 MemorySet 创建时，页帧分配器分配内存，页表进行相应的映射；而 MemorySet 销毁时，其拥有的所有页帧将返还给页帧分配器，页表会取消相应的映射。

MemorySet 可分为内核 MemorySet 和用户 MemorySet。内核 MemorySet 以全局变量的形式存在，需要用读写锁保护起来，以确保线程安全；而用户 MemorySet 被封装在 PCB 中，被 PCB 的内部 Mutex 锁保护起来。

(2) 页帧分配器 FrameAllocator

jkxs-OS 定义了一个全局的页帧分配器（FrameAllocator），同样由一个读写锁保护起来。FrameAllocator 为内核提供了动态申请和释放内存的能力，这样就可以加强内核对各种以内存为基础的资源分配与管理。它内部有三个成员，分别是 current、end 和 recycled 数组，共同记录了当前尚未被分配出去的物理页帧号 ppn 的集合。其中[current, end)区间的物理页帧从未被分配出去，而 recycled 记录了所有已被回收的物理页帧号，它们在将来还可以再次被分配。

(3) 页帧标识 FrameTracker

FrameTracker 是伴随着页帧分配器出现的，它是已经被分配出去的物理页帧的唯一标识。jkxs-OS 规定，页帧的分配只能通过 frame_alloc()接口实现，该函数中 FrameAllocator 会分配一个物理页号 ppn，绑定在页帧标识 FrameTracker 上。该 FrameTracker 一般会被保存在特定的结构体中，如 MemorySet 的 areas 中，这样 FrameTracker 的生命周期就被延长了。当 FrameTracker 的生命周期结束时（代表该进程已经不再拥有该页帧了），会自动调用其 Drop 方法，将其 ppn 重新加入 FrameTracker 中。

FrameTracker 的设计也应用了 RAII 的思想，将开发者从手动释放页帧的繁琐工作中解放出来，且有效避免了内存泄漏问题的发生。

### 3.2.2 内核地址空间设计与实现

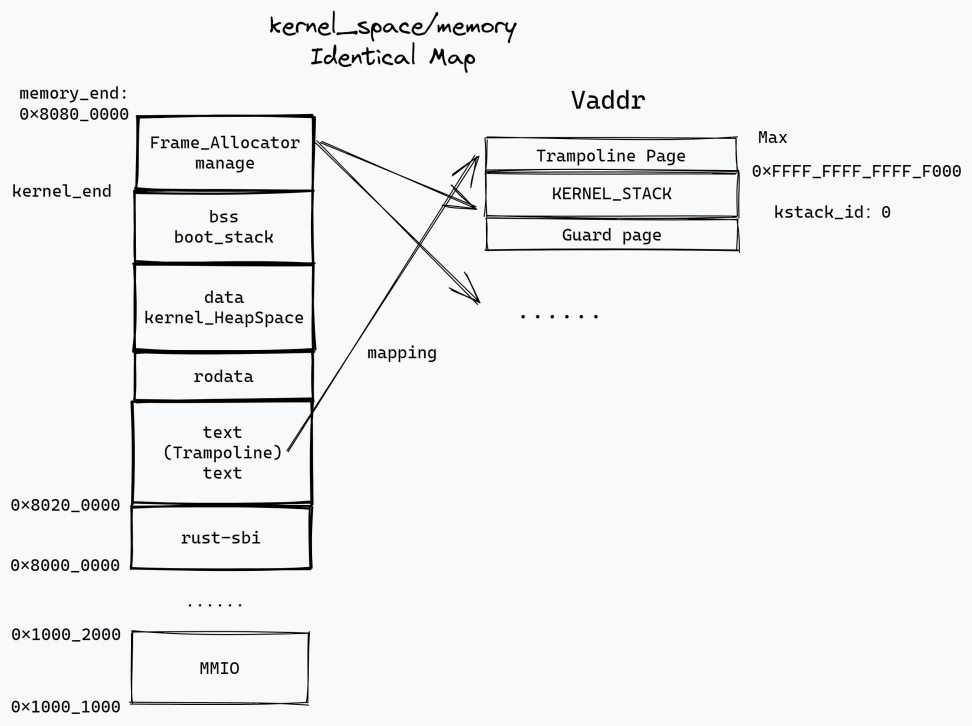


图 2　内核地址空间分布

内核地址空间的分布如图 2 所示。

- [0x1000_1000, 0x1000_2000) 为 MMIO 映射区域（对 Qemu 而言）。CPU 可以通过访问 MMIO 区域内的地址来对外设进行访问，串口读写、SDCard 读写都是基于 MMIO 实现的。不同硬件平台上 MMIO 映射的地址会有所差异。
- [0x8000_0000, 0x8020_0000) 为 SBI（Supervisor Binary Interface）区域。SBI 专门为 riscv 体系结构设计，可看作是 Machine Mode 提供给 Supervisor Mode 的功能接口。它力图屏蔽硬件层的不一致性，使上层软件不必过多关心硬件的细节。在本项目中，SBI 既可以作为 Bootloader，完成相应的准备后引导 CPU 进入 S 态运行内核；也可以为 S 态提供各种 M 态的服务，如串口输入输出、关机等。

- [0x8000_0000, kernel_end) 为内核区域。和一个标准的 elf 类似，内核区域也分为代码段（.text），只读数据段（.rodata），可读写数据段（.data 和.bss）。
- [kernel_end, memory_end) 为空闲的内存区域，可被 FrameAllocator 分配给页表或用户进程。memory_end 标识可用物理内存的结束地址，其值由物理内存的容量决定。

内核地址空间采用恒等映射，即内核页表中每一对映射中，虚拟地址和物理地址都是相等的。

### 3.2.3 用户地址空间设计与实现

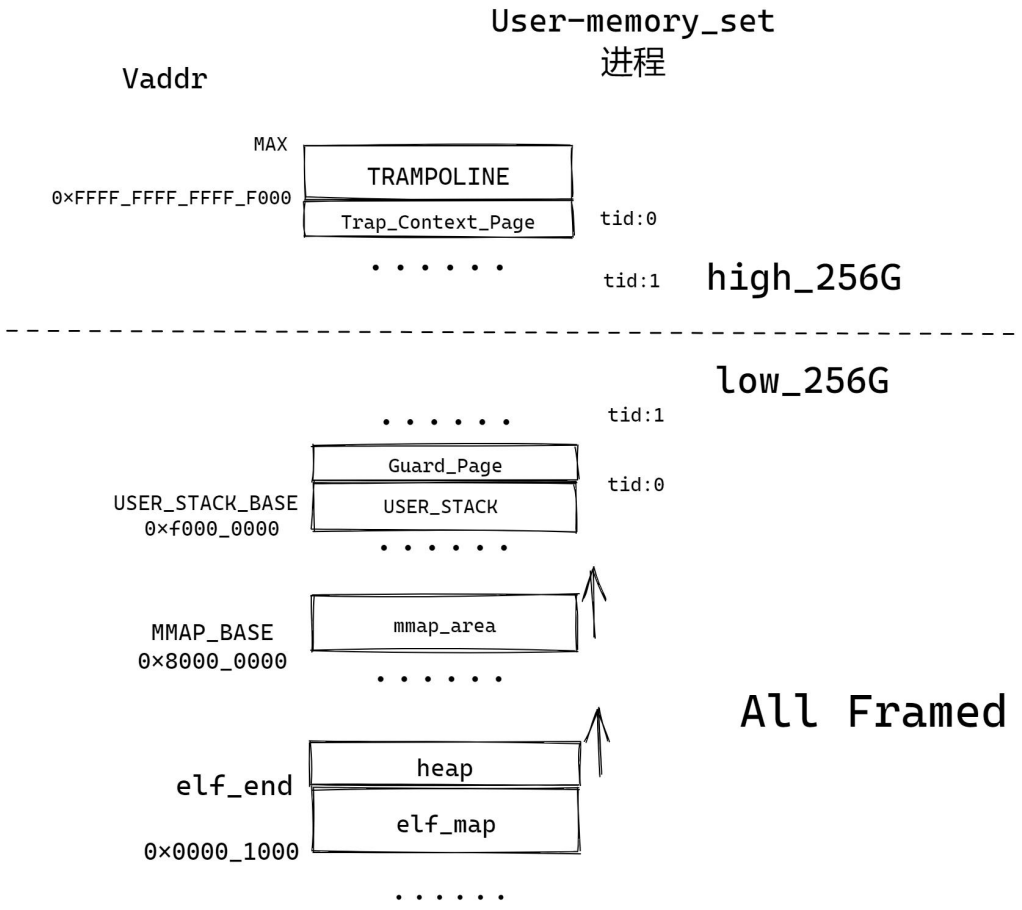


图 3　用户进程地址空间分布

用户进程地址空间的分布如图 3 所示。

- [0x1000, elf_end) 为 elf 区域。这部分区域分为代码段（.text），只读数据段

（.rodata），可读写数据段（.data 和.bss）。

- [end_end, heap_top) 为用户堆内存区域。进程可以通过 brk 或 sbrk 系统调用实现堆内存的动态增长或减少；堆内存是从低地址向高地址增长的。
- [0xF000_0000, 0xF000_0000 + n * (Page_size + Stack_size) ) 为用户栈区域，其中 n 为进程拥有的线程数量，本项目中 n 总为 1。每个用户栈顶部都用一个 Guard Page 保护起来，每当访问到 Guard Page 就会触发缺页异常，防止栈溢出。栈内存都是从高地址向低地址增长的。
- [0xFFFF_FFFF_FFFF_F000 - n * Page_size, 0xFFFF_FFFF_FFFF_F000) 为 TrapContext 区域，其中 n 为进程拥有的线程数量，本项目中 n 总为 1。实际上，在 TCB 中也有 TrapContext 的存在，用户空间中的 TrapContext 和 TCB 中的 TrapContext 指向的其实是同一个内存区域。用户页表也进行 TrapContext 的映射，可以使 Trap 的处理逻辑更容易实现。
- [0xFFFF_FFFF_FFFF_F000, 0xFFFF_FFFF_FFFF_FFFF] 为跳板映射区域，其实际指向的是内核.text 段中的一个名为 Trampoline 的区域。将 stvec 统一设置为 Trampoline，可以使线程响应中断时自动跳转到保存上下文、Trap 处理逻辑中。

## 3.3 SDcard 驱动

此模块为块设备抽象接口在 SDcard 存储设备上的具体实现。基于块设备的抽象接口，在文件系统看来块设备访问时完全透明的，文件系统与块设备驱动解耦。由于同时支持了 K210 和 FU740，所以 SDcard 驱动其实有两个具体的实现，且二者都为 SPI 模式下的 SDcard 驱动。在 FU740 上，我们使用了我们自主开发的 SDcard 驱动。在 K210 上，我们使用开源项目 rCore-Tutorial 的 SD 卡驱动块设备驱动。以下主要介绍在 FU740 上 SDcard 驱动的实现。

整体设计框图如图 4 所示：

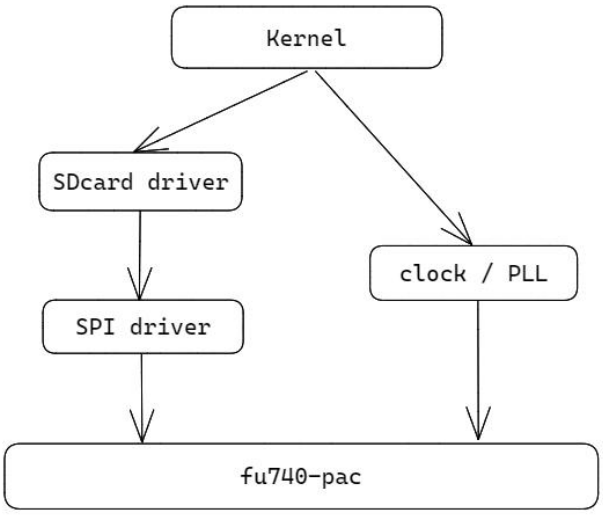


图 4　SDCard 驱动整体设计框图

### 3.3.1 外设寄存器抽象层

该层主要实现对 FU740 SOC 上外设寄存器访问的抽象，为上层提供访问 FU740 外设寄存器的方法。本着不重复造轮子的思想，同时提高开发效率，我们使用 riscv-rust 组织的开源项目 fu740-pac(Peripheral access API for FU740 SoC)为外设寄存器抽象。

### 3.3.2 SPI 驱动层

在 SPI 驱动层，我们主要基于 FU740 上的 SPI 控制模块[Serial Peripheral Interface (SPI) controller]实现一个 Rust 版本的 SPI 协议。显然, 对于不同的 SOC 在 SPI 控制模块的硬件实现是不同的。在这一层,我们屏蔽了 FU740 实现 SPI 控制和传输的硬件实现差异, 为上层使用提供统一的接口。

```
pub trait SPI {
    fn init(&self);     //初始化/复位SPI相关寄存器
    fn configure(       //配置SPI相关寄存器
        &self,
        protocol: u8,
        endianness: bool,
        cs_active_high: u32,
        csid: u32,
    );
    fn set_clk_rate(&self, div: u32) -> u32;                    // 设置SPI时钟频率
    fn recv_data(&self, chip_select: u32, rx: &mut [u8]);       //接受数据
    fn send_data(&self, chip_select: u32, tx: &[u8]);           //发送数据
    fn fill_data(&self, chip_select: u32, value: u32, tx_len: usize);//未实现
    fn switch_cs(&self, enable: bool, csid: u32);               //片选与SPI-csmode设置
}
```

图 5　SPI 驱动接口设计

简单来说,初始化,SPI 状态设置和 SPI 时钟设置这三个抽象接口的具体实现都是基于硬件手册和 SPI 协议对相关 SPI 外设寄存器相应值的写入.

对于 SPI 数据发送和接收大概可以分为以下几个步骤:

- 设置 fmt::direction 传输方向 ,csid 片选 ID, 片选 ID 相应的模式 csmod .
- 将数据写入 SPI 传输队列(硬件实现,长度为 8)
- 设置相应的 wartmark
- 循环读取 ip(等待 wartmark 反馈)
- 若为接收数据则从 rxdata 中读取并保存数据
- 释放 csmod

在使用 Rust 实现 SPI 驱动时,我们参考了基于 C 语言的 hifive SPI 驱动实现: Linux hifive SPI 驱动与官方 SDK SPI 驱动.

### 3.3.3 SDcard 驱动层

在 SDcard 驱动层,由于有了 SPI 驱动层的抽象.　我们只需要基于 SPI 提供的接口,实现 SPI 模式下的 SDcard 驱动在 SDcard 驱动层,由于有了 SPI 驱动层的抽象。我们只需要基于 SPI 提供的接口，实现 SPI 模式下的 SDcard 驱动。作为参考，SD Association 给出的文档_Part1_Physical_Layer_Simplified_Specification_Ver8.00.pdf_的第 7 章_SPI Mode_中对此有着详细的描述。

简单来说我们只需要实现三个目标,一是 SDcard 的初始化，二是 SDcard 的写操作,

三是 SDcard 的读操作.

- SDcard 初始化

  对于初始化,大致可以分为如下几个步骤

  - 让 SD 卡进入 SPI 模式　CMD0
  - 确认 SD 卡的接口条件　CMD8(不一定会收到回复)
  - 确定 SD 卡的工作电压区间　CMD58
  - 使 SD 卡脱离空闲状态(IDLE)　CMD55 + ACMD41
  - 确定 SDcard 的 Capacity 类型　CMD58

- SDcard 的读操作

  SD 卡的读操作主要是通过 CMD17（读取单个 Block）和 CMD18（读取多个连续 Block）完成。在本模块中,二者都已实现,但目前系统中仅使用到 CMD17. 下面简要介绍单个 block 的读取.

  - 主机发送 CMD17,等待并读取 SDcard response 确定 SDcard 已经成功接收命令
  - 不断读取 SDcard 发送到的数据,直到出现长度为 1Byte 的'Start Token'（其值为 0xfe）.接下来的数据就是主机所需数据.

- SDcard 的写操作

  SD 卡的写操作与读操作类似。写操作可以通过 CMD24（写入单个 Block）和 CMD25（写入多个 Block）完成。同样,目前暂时仅使用到 CMD24

  - 主机发送 CMD24,等待并读取 SDcard response 确定 SDcard 已经成功接收命令
  - 发送带'Start Block Tocken'的数据块
  - 等待 SDcard data-response
  - 循环读取 SDcard response,直到 SDcard 返回 0xff(表明 SDcard 完成本次写数据的处理)

值得注意的是,正常 SDcard 的写操作处理时,SDcard 在发送 data-response 后仍需要一定的时间进行数据处理, 一般而言需要发送 CMD13 去确定 SDcard 的处理结果. 但在这里我们通过不断读取 response 来实现了相同的效果.

## 3.4 FAT32 文件系统

文件系统的实现采用了五层的分层结构，从下到上分别为磁盘块设备接口层、块缓存层、磁盘布局层、文件系统管理层、虚拟文件系统层。

此外，由于 Unmatched 赛道将 SD 卡镜像放置到内存 0x90000000 位置，因此访问 SD 卡内容无需通过 SD 卡驱动，而是直接访问对应位置的内存即可。我们重新设计了内存镜像访问层用于替换低两层的磁盘块设备接口层和块缓存层，并向上暴露出原有的接口以供访问。

### 3.4.1 磁盘块设备接口层

磁盘块设备接口层提供了 BlockDevice trait，该 trait 要求实现两个方法 read_block 和 write_block 以分别用于对块设备的读写，这相当于提供了块设备的抽象接口，无论是哪种块设备，只要实现了这个 trait，都可以接入文件系统以供使用。

### 3.4.2 块缓存层

块缓存层用于在内存中缓存磁盘块的数据，其中实现了块缓存 BlockCache 和块缓存管理器 BlockCacheManager。

块缓存 BlockCache 向外提供了 read 和 modify 接口，用于对块缓存的读写。当创建一个 BlockCache 时，会通过磁盘块设备接口层所提供的接口，将一个磁盘块读取到缓冲区中。

我们设定缓存的大小为 2MB，即可以同时驻留 4096 个磁盘块。考虑到 FAT32 文件系统对文件进行读写的特殊性，我们设置了两个缓存，分别为信息缓存和数据缓存。其中信息缓存用于缓存 FAT 表、引导扇区、FSInfo 扇区、目录等用于信息检索的块，而数据缓存则用于缓存文件内容。为此，我们也提供了两个全局块缓存管

理器 INFO_BLOCK_CACHE_MANAGER 和 DATA_BLOCK_CACHE_MANAGER。

### 3.4.3 内存镜像访问层

对于对应块号的块的访问，内存镜像访问层可以帮我们计算出该块所在的内存位置。同时，相较于原来块缓存层的 BlockCacheManager 和 BlockCache 的双锁设计，内存镜像访问层去除掉了 BlockCacheManager 的锁，但是由于对对应内存区域的访问必须是互斥的，因此 BlockCache 的锁需要保留。

### 3.4.4 磁盘布局层

- DBR(DOS Boot Record)

DOS 引导记录占据一个扇区，且通常位于 0 号扇区(逻辑 0 扇区)，下文中的 BPB 和 EBR 二者共同组成 DBR 的 512 个字节。DBR 用于解释文件系统，一般仅在进入文件系统时读入。我们为 BPB 实现了成员字段与磁盘数据一一对应的结构体，而 EBR 由于冗余数据过多，仅实现到 FSInfo(文件系统信息扇区)字段为止。在进入文件系统读入 DBR 扇区后，就可以将块缓存的引用转换为 BPB 和 EBR 结构体的引用，从而通过对应字段来获取文件系统的相关信息。

  - BPB(BIOS Parameter Block)

| 字节偏移(十进制) | 字节偏移(十六进制) | 字节数 | 意义 |
|---|---|---|---|
| 0 | 0x00 | 3 | 跳转指令 |
| 3 | 0x03 | 8 | 文件系统标志和版本号 |
| 11 | 0x0B | 2 | 每个扇区的字节数 |
| 13 | 0x0D | 1 | 每个簇的扇区数 |
| 14 | 0x0E | 2 | 保留区的扇区数(包含引导记录扇区) |
| 16 | 0x10 | 1 | FAT 表个数(通常为 2) |

| | | | |
|---|---|---|---|
| 17 | 0x11 | 2 | 0 |
| 19 | 0x13 | 2 | 0 |
| 21 | 0x15 | 1 | 存储介质 |
| 22 | 0x16 | 2 | 0 |
| 24 | 0x18 | 2 | 每磁道扇区数 |
| 26 | 0x1A | 2 | 磁头数 |
| 28 | 0x1C | 4 | EBR 分区之前所隐藏的扇区数 |
| 32 | 0x20 | 4 | 文件系统总扇区数 |

表 1　BPB 内字段

- EBR(Extended Boot Record)

EBR 紧跟在 BPB 的后面，二者共占一个扇区。

| 字节偏移(十进制) | 字节偏移(十六进制) | 字节数 | 意义 |
|---|---|---|---|
| 36 | 0x24 | 4 | 每个 FAT 表占用扇区数 |
| 40 | 0x28 | 2 | 标志 |
| 42 | 0x2A | 2 | FAT 版本号 |
| 44 | 0x2C | 4 | 根目录所在簇号，通常为 2 |
| 48 | 0x30 | 2 | FSInfo 扇区号 |
| 50 | 0x32 | 2 | 备份引导扇区的扇区号 |
| 52 | 0x34 | 12 | 保留 |
| 64 | 0x40 | 1 | 驱动号 |
| 65 | 0x41 | 1 | 保留 |
| 66 | 0x42 | 1 | 0x29 |
| 67 | 0x43 | 4 | 卷序列号 |
| 71 | 0x47 | 11 | 卷标 |
| 82 | 0x52 | 8 | “FAT32” |
| 90 | 0x5A | 420 | 引导代码 |
| 510 | 0x1FE | 2 | 0xAA55 |

表 2　EBR 内字段

- FSInfo Structure

FSInfo 信息扇区一般位于文件系统的 1 号扇区，用于记录文件系统中空闲簇的数量和最近一次分配的簇号，帮助空闲簇的检索。FSInfo 扇区中的有效字段非常少，但是需要频繁地读写。考虑到这个因素，我们仍然为其实现了完整的结构体 FSInfoInner，作为结构体 FSInfo 的成员之一。其常驻内存，且不通过块缓存来进行读写，对 FSInfo 扇区进行读写只需要修改 FSInfoInner 结构体相应字段的值即可。当然，我们也为 FSInfo 实现了同步机制，我们维护了 FSInfo 扇区所在的物理扇区号 fsinfo_sector，当 FSInfo 的生命周期结束或者需要同步到磁盘时，就会将 FSInfoInner 即 FSInfo 扇区的内容写入到磁盘中的对应位置。

| 字节偏移(十进制) | 字节偏移(十六进制) | 字节数 | 意义 |
|---|---|---|---|
| 0 | 0x0 | 4 | 0x41615252 |
| 4 | 0x4 | 480 | 保留，0 |
| 484 | 0x1E4 | 4 | 0x61417272 |
| 488 | 0x1E8 | 4 | 文件系统的空簇数 |
| 492 | 0x1EC | 4 | 最近一次分配的簇号 |
| 496 | 0x1F0 | 12 | 保留 |
| 508 | 0x1FC | 4 | 0xAA550000 |

表 3　FSInfo 扇区内字段

- File Allocation Table(文件分配表 FAT)

FAT32 文件系统中分配磁盘空间按簇来分配，即文件在占用磁盘空间时的基本单位是簇。即使是一个只有一个字节的小文件，操作系统也会给它分配一个簇来存储。而对于大文件则可能需要多个簇来存储，这多个簇在磁盘中不一定是连续存放的。因此需要 FAT 表来描述簇的分配状态以及表明文件或目录的下一个簇的簇号。

FAT32 中每个簇的簇地址为 32bits，其中只使用低 28 位来寻址磁盘上的簇，保留最高的 4 位。FAT 表中的所有字节位置以 4 字节为单位进行划分，并对所有划分后的位置由 0 进行地址编号。0 号地址与 1 号地址被系统保留并存储特殊标志内容。从 2 号地址开始，每个地址对应于数据区的簇号，FAT 表中的地址编号与数据区中的簇号相同。

当文件系统格式化时，分配给 FAT 表的区域会被清空，由于簇号起始于 2 号，FAT 表的 0 号表项与 1 号表项不与任何簇对应，因此需要在 FAT 表的 0 号表项和 1 号表项写入对应的特定值，其中 0 号表项值总为 0x0FFFFFF8，1 号表项值总为 0x0FFFFFFF。最后在 2 号 FAT 表项写入一个结束标志表示根目录。

在 FAT 表中获取簇链中当前簇 cluster 的下一个簇的簇号的方法如图 6 所示：

```
// first_fat_sector为FAT表的第一个扇区
// FAT_ENTRY_PER_SEC为每个扇区的FAT表项个数
let fat_sector = first_fat_sector + cluster / FAT_ENTRY_PER_SEC;
let fat_offset = 4 * (cluster % FAT_ENTRY_PER_SEC);

// 读取磁盘中逻辑扇区号为fat_sector的扇区读到块缓存中
// 并读取该扇区中偏移为fat_offset的4个字节得到next_cluster

return next_cluster & 0x0FFFFFFF
```

图 6　FAT 表中获取下一簇簇号的伪代码

如果 next_cluster 大于等于 0x0FFFFFF8，则表示该簇链中不再有簇了，整个文件已经被读取完。如果 next_cluster 等于 0x0FFFFFF7，则表示这个簇为坏簇。如果 next_cluster 为 0，说明对应簇未被分配使用。

基于上述方法，我们为 FAT 表实现了表 4 中的主要接口：

| 接口名称 | 描述 |
| --- | --- |
| next_free_cluster | 查询某个簇开始的下一个空闲簇号 |
| get_next_cluster | 查询某个簇的下一个簇的簇号 |
| set_next_cluster | 设置某个簇的下一个簇的簇号 |
| get_cluster_at | 获取某个簇所在簇链的，从该簇开始计数的第 i 个簇的簇号 |
| get_final_cluster | 获取某个簇所在簇链的最后一个簇的簇号 |
| get_all_clusters | 获取某个簇所在簇链的，从该簇开始的所有簇号 |
| cluster_count | 获取某个簇所在簇链的，从该簇开始计数的簇的个数 |

表 4　FAT 表主要接口

上述接口上述接口均需要通过信息缓存对块设备进行读写，因此我们对 FAT 表的内存结构仅保存了 FAT1、FAT2 的起始扇区号和 FAT 表所允许的最大簇数。

- 簇链 Chain

簇链的优化主要有两个原因：一是如果将整个 FAT 表当作一个整体对象并利用读写锁进行互斥访问，当对两个文件分别进行读和写时，虽然这两个文件所拥有的簇链是不同的，但读操作仍然会被写操作阻塞。二是 FAT 表是链式存储结构，读取某个簇链上的第 n 个簇的簇号只能 O(n)顺序查找，如果能够做到随机访问，那便能提升很大的性能。

为此，我们将文件所拥有的簇链进行了缓存，对外暴露出与 FAT 表基本一致的接口。当试图通过 Chain 对簇链进行查询时，会先判断该簇链的第一个簇 first_cluster 是否在 Chain 中，如果在的话则可以进行查询，否则需要通过 fat 表进行查询并更新 Chain。

- 目录

FAT 文件系统中有两种类型的目录，分别是 Standard 8.3 目录项（出现在所有 FAT 文件系统上）和长文件名目录项（可选地出现以允许更长的文件名）

- Standard 8.3(短文件名)

每个文件或子目录都一定会被分配一个短文件名目录项，短文件名目录项包含了文件的相关信息，包括文件名、拓展名、属性等，其格式如表 5 所示：

| 字节偏移 | 字节数 | 含义 |
|---|---|---|
| 0 | 11 | 文件名，前 8 个字符是名称(不足 8 个则用 0x20 填充)<br>后 3 个字符是扩展名(如果是子目录则用 0x20 填充) |
| 11 | 1 | 文件属性，包括 READ_ONLY=0x01、HIDDEN=0x02、SYSTEM=0x04、VOLUME_ID=0x08、DIRECTORY=0x10、ARCHIVE=0x20、LFN=READ_ONLY |
| 12 | 1 | 默认为 0，表示短文件名全大写表示（包括扩展名） |
| 13 | 1 | 以十分之一秒为单位的文件创建时间 |
| 14 | 2 | 文件创建时间，其中小时占 5bits，分钟占 6bits， |

| | | |
|---|---|---|
| | | 秒占 5bits，秒数需要乘 2 |
| 16 | 2 | 文件创建日期，其中年份占 7bits（相对于 1980 年），月份占 4bits，日期占 5bits |
| 18 | 2 | 文件最近访问日期 |
| 20 | 2 | 文件起始簇号的高 16 位 |
| 22 | 2 | 文件最近修改时间 |
| 24 | 2 | 文件最近修改日期 |
| 26 | 2 | 文件起始簇号的低 16 位 |
| 28 | 4 | 以字节为单位的文件大小（如果是子目录则全置为 0） |

表 5　短文件名目录项字段

需要注意的是，对于一个短文件名目录项的第一个字符，如果该目录项正在使用中则 0x0 位置的值为文件名或子目录名的第一个字符；如果该目录项未被使用则 0x0 位置的值为 0x00；如果该目录项曾经被使用过但是现在已经被删除则 0x0 位置的值为 0xE5。

对于短文件名目录项，我们实现了完整的结构体 ShortDirEntry 与其磁盘数据一一对应。除了对其字段进行读写和判断外，我们还对短文件名目录项实现了表 6 中的接口：

| 接口名称 | 描述 |
|---|---|
| get_pos | 获取文件偏移量所在的簇、扇区和扇区内偏移 |
| read_at | 以偏移量读文件 |
| write_at | 以偏移量写文件 |
| checksum | 计算短文件名的校验和 |
| find_short_name | 在该目录下查询指定名字的短文件名目录项 |
| find_free_dirent | 在该目录下查询空白的目录项 |

表 6　短文件名目录项实现接口

事实上，以偏移量读写文件涉及到 FAT 表的读写、块缓存层的调用等等。通过上述接口的实现，上层模块在进行相关操作时，就不必关心其内部的复杂实现，

只需要传入所需的参数即可。

- Long File Names

长文件名目录项的格式如表 7 所示：

| 字节偏移 | 字节数 | 含义 |
|---|---|---|
| 0 | 1 | 长文件名目录项的序列号 |
| 1 | 10 | 长文件名的 1~5 个字符（Unicode 编码，每个字符两个字节） |
| 11 | 1 | 0x0F |
| 12 | 1 | 0 |
| 13 | 1 | 短文件名的校验和<br>(一个文件的不同长文件名的目录项的短文件名校验和相同) |
| 14 | 12 | 长文件名的 6~11 个字符 |
| 26 | 2 | 0 |
| 28 | 4 | 长文件名的 12~13 个字符 |

表 7　长文件名目录项字段

长文件名目录项的注意点：

- 长文件名目录项总是有个紧随其后的 Standard 8.3 目录项。
- 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。所以一个文件可能需要多个长文件名目录项，这时长文件名的各个目录项按倒序排列在目录表中，其第一部分距离短文件名目录项是最近的。
- 长文件名目录项的第一个字节为序列号。一个文件的第一个目录项序列号为 1，然后依次递增。如果是该文件的最后一个长文件名目录项，则将该目录项的序号与 0x40 进行或运算的结果写入该位置。如果该长文件名目录项对应的文件或子目录被删除，则将该字节设置成删除标志 0xE5。
- 长文件名如果结束了但还有未使用的字节，则会在在文件名后先填充两个的 0x00，然后开始使用 0xFF 填充。

当创建一个长文件名文件时，其短文件名的命名原则为：

- 取长文件名的前 6 个字符加上”~1”形成短文件名，扩展名不变。
- 如果已存在这个文件名，则符号”~”后的数字递增，直到 5。
- 如果文件名中"~"后面的数字达到 5，则短文件名只使用长文件名的前两个字母。通过数学操纵长文件名的剩余字母生成短文件名的后四个字母，然后加后缀"~1"直到最后(如果有必要，可以是其他数字以避免重复的文件名)。

对于长文件名目录项，我们也实现了完整的结构体 LongDirEntry 与其磁盘数据一一对应。

■ "."目录项和".."目录项

一个子目录的起始簇中的前两个目录项为"."目录项和".."目录项，其中"."目录项中记录的起始簇号也就是该子目录目前所处的位置。

### 3.4.5 文件系统管理层

我们在文件系统管理层实现了一个文件系统管理器 FAT32Manager，它用于统筹并联系起下层模块，并基于它们完成文件系统的打开、簇的分配与去分配、时间的转换等操作。

为此，FAT32Manager 的结构体定义如表 8 所示：

| 成员 | 描述 |
| --- | --- |
| bytes_per_sector | 每个扇区的字节数 |
| bytes_per_cluster | 每个簇的字节数 |
| sectors_per_cluster | 每个簇的扇区数 |
| root_sector | 根目录所在扇区号 |
| block_device | 块设备的引用 |
| fsinfo | FSInfo 结构体的引用 |
| fat | FAT 表内存结构的引用 |
| root_dirent | 虚拟根目录项 |

表 8　FAT32Manager 结构体定义

其中，我们知道 FAT32 文件系统中文件或目录的相关信息都存储在其短文件名目录项中，然后通过短文件名目录项 ShortDirEntry 提供的相关接口就可以实现对文件或目录的读写等操作。然而，在 FAT32 文件系统中根目录并没有对应的目录项，因此我们创建了一个虚拟的根目录项，保存在 FAT32Manager 的 root_dirent 字段中，通过其来对根目录进行相关操作。

我们为 FAT32Manager 实现了表 9 中的主要接口：

| 接口名称 | 描述 |
| --- | --- |
| open | 打开文件系统 |
| clear_cluster | 清除指定簇中的所有扇区 |
| alloc_cluster | 在 FAT 表上分配指定数量个簇 |
| dealloc_cluster | 在 FAT 表上去分配指定的簇 |
| long_name_split | 拆分长文件名 |

表 9　FAT32Manager 主要接口

FAT32 文件系统的 SD 卡整体布局如图 7 所示：

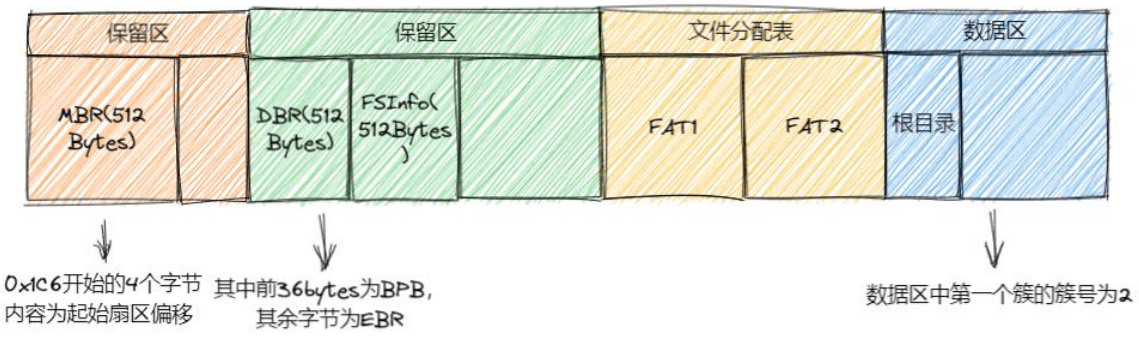


图 7　FAT32 文件系统的 SD 卡整体布局

由此我们可以得到 open 即打开文件系统的流程如下：

1. 读入 MBR 所在扇区(物理 0 扇区)，读取其中 0x1C6 开始的 4 个字节的内容，即相对扇区数。设置块缓存的起始扇区偏移为该相对扇区数。此后通过块缓存层对磁盘进行访问仅需通过逻辑扇区号即可。(需要注意的是，读入 MBR 所在扇区时也是通过块缓存层完成，只不过块缓存层初始的起始扇区偏移为 0)。

2. 读入逻辑 0 扇区，读取其中的 BPB 和 EBR 获取文件系统的相关信息。

3. 通过 EBR 中的 FSInfo 扇区号，读取 FSInfo，校验签名并构造其内存结构。

4. 通过 BPB 的保留区的扇区数、EBR 的 FAT 表占用扇区数等信息，构造 FAT 表的内存结构。

5. 构造虚拟根目录项。

6. 构造文件系统管理器并返回。

### 3.4.6 虚拟文件系统层

虚拟文件系统层实现了 FAT32 文件系统的虚拟文件对象 VFile。为了方便对其访问，我们记录了其短文件名目录项所在的扇区 short_sector、扇区偏移 short_offset 以及其所有长文件名目录项的位置 long_pos_vec。同时因为对文件的操作需要底层文件系统和块设备的支持，所以我们也保存了文件系统和块设备的引用 fs 和 block_device。为方便内核使用，我们也保存了文件名 name 和属性 attribute。

我们为 VFile 实现的主要接口如表 10 所示：

| 接口名称 | 描述 |
| --- | --- |
| create | 在当前目录下创建目录项 |
| find_vfile_name | 通过名称查找当前目录下的目录项 |
| find_vfile_path | 通过相对路径查找目录项 |
| read_at | 对短文件名目录项的 read_at 的封装，以偏移量读取文件 |
| write_at | 对短文件名目录项的 write_at 的封装，以偏移量写文件 |
| ls | 获取当前目录下的所有文件名及其属性 |
| dirent_info | 获取当前目录下指定目录项的信息(包括名称、短文件名目录项偏移、起始簇号、属性) |
| clear | 清除文件内容 |
| remove | 删除文件 |

表 10　VFile 主要接口

# 4 系统测试

## 4.1 测试准备

对于 FAT32 文件系统在用户态下的简单测试，我们通过 Rust 标准库中的 std::file::File 以访问 Linux 上的一个文件，在为其加锁后并为其实现 BlockDevice trait 后，就相当于用 Linux 上的一个文件模拟一个块设备。使用 File::create 在 Linux 上创建一个固定大小的文件 fat32.img，并对其进行 FAT32 格式化得到文件系统镜像，然后就可以利用这个镜像进行测试了。

对于 SDcard 驱动的测试，分为两个部分。一是使用 qemu 6.2.0 模拟真实 sifive-u 硬件平台(fu540), 在其上进行 SPI-SDcard 的仿真测试。这部分测试框架主要参考徐文浩同学的 SDcard 驱动仿真测试架构。二是在真实 fu740 硬件平台上编写测试函数进行驱动测试。

## 4.2 测试方法

| 测试 | 测试方法 |
| --- | --- |
| 文件系统功能测试 | |
| 文件系统启动测试 | 使用文件系统打开 FAT32 镜像，测试是否能够正确读取文件系统基本信息、校验签名是否有效 |
| 文件创建测试 | 创建短文件名文件、长文件名文件、短文件名同名的长文件名文件 |
| 目录创建测试 | 创建目录与多级目录 |
| ls 测试 | 根目录和其子目录下调用 ls 方法 |
| 文件查找测试 | 通过文件名查找当前目录下的文件或目录，通过相对路径查找多级目录下的文件或目录，测试是否能够查找到对应文件或目录 |
| 文件读写测试 | 进行了单簇简单读写测试和多簇规定大小读写测试，通过写入再读出来判断内容是否一致 |

| | |
|---|---|
| 文件删除测试 | 进行文件的删除，测试删除后能否 ls 或查找到已删除文件 |
| 文件系统写入磁盘测试 | 使用文件系统再次打开 FAT32 镜像，检查上一次写入磁盘的值是否正确。或者通过 Linux 下的 mount 命令将 FAT32 镜像挂载在目录下，检查相关的值 |

表 11　文件系统功能测试

| 测试 | 测试方法 |
|---|---|
| SDcard 驱动功能仿真测试(Standard Capacity) | |
| 简单读写测试 | 以 4 block 为单位进行先写后读，验证正确性 |
| SDcard 驱动功能测试(SDHC/SDXC) | |
| 大数据块读写测试 | 一次性写入 512 block 大小的数据，然后读出验证正确性 |
| 小数据块连续读写测试 | 对一个 block 进行先写后读，重复 512 次，循环过程中验证正确性 |

表 12　SDcard 驱动功能仿真测试

## 4.3 测试结果

| 测试 | 测试结果 |
|---|---|
| 文件系统功能测试：均通过 | |
| 文件系统启动测试 | 文件系统能够正确启动，能够正确读取出文件系统基本信息、校验签名有效 |
| 文件创建测试 | 短文件名文件、长文件名文件、短文件名同名的长文件名文件均能够正确创建 |
| 目录创建测试 | 目录与多级目录均能够正确创建 |
| ls 测试 | 目录下的所有目录项及其属性就能正确打印出来 |
| 文件查找测试 | 通过文件名和相对路径均能够查找多级目录下的文件或目录 |
| 文件读写测试 | 进行了单簇简单读写测试和多簇规定大小读写测试，读出的内容与写入的内容一致 |

| | |
|---|---|
| 文件删除测试 | 进行文件的删除，删除后不能 ls 或查找到已删除的文件 |
| 文件系统写入磁盘测试 | 使用文件系统再次打开 FAT32 镜像，上一次写入磁盘的值正确。通过 Linux 下的 mount 命令将 FAT32 镜像挂载在目录下，相关的值均正确。 |

表 13　文件系统功能测试结果

| 测试 | 测试结果 |
|---|---|
| SDcard 驱动功能仿真测试通过 | |
| 简单读写测试 | 每次写入数据与读出数据都相同，仿真环境中读写功能正常 |
| SDcard 驱动功能测试通过 | |
| 大数据块读写测试 | 写入数据与读出数据相同，大文件读写功能正确 |
| 小数据块连续读写测试 | 写入数据与读出数据相同，大量小数据块连续读写的功能正确 |

表 14　SDcard 驱动功能仿真测试结果

# 5 总结与展望

## 5.1 工作总结

（1）通过模块化开发实现了较为完整的操作系统内核：支持 Linux 基本标准的系统调用；

（2）多硬件平台适配：在 K210 和 FU740 上完成真实硬件平台的适配；

（3）实现多核支持：在 FU740/QEMU 上多核运行成功；

（4）支持多个用户程序：移植了精简版的 rootfs，支持 sh、vi 等 Linux 原生用户程序，移植 gcc，打造了一套贴近现实的用户编辑、工作环境。

## 5.2 创新点

（1）Rust 编写: 100% Rust 的操作系统内核；

（2）多核支持：jkxs-OS 支持在 fu740 及 qemu 上多核并行运行；

（3）多平台支持：jkxs-OS 同时适配了 k210 与 fu740 两个硬件平台；

（4）良好的用户体验：功能强大的 shell，支持 tab 命令补全、命令历史回溯等功能；

（5）便捷的调试功能：自实现系统调用追踪与较为完整的日志系统。

## 5.3 未来展望

（1）支持更多真实世界中的应用程序；

（2）基于硬件平台拓展内核的功能如网卡等；

（3）提升多核运行的稳定性；

（4）SDcard 驱动的稳定性。