

# 概述

本项目旨在用 rust 开发一个基于 RISC-V 架构的操作系统内核 (s0s)，支持启动初始化、中断、I/O、进程管理、内存管理、执行文件解析、fat32 文件系统功能等，同时能够在真实硬件平台 k210 上运行。

本项目移植于 rCore，使用了洛佳等开发的 rustsbi。使用 GPL v3 协议。

Rust 是一门同时追求安全、并发和性能的现代系统级编程语言，其丰富的类型系统和所有权模型保证了内存安全和线程安全，在编译期就能够消除各种各样的错误，而其性能还能与 C/C++ 相媲美。得益于 rust 的诸多优秀特性，本项目使用 rust 进行开发。

RISC-V 是一个开源的精简指令集架构，相比于 x86 架构，其设计简单，且没有历史的包袱，因此我们更能专注于操作系统的设计。

rCore 是清华开发的一个开源操作系统，使用了模块化设计，虽然功能简单但五脏俱全，且拥有丰富的文档。我们的项目移植与此，添加使用了 fat32 文件系统，并更改和添加了许多系统调用，支持部分 busybox 和 lua 功能。

# 环境

本项目使用的环境与 rcore 一样，可参考 rcore 相应的[环境配置](#)。[gdb插件配置](#)

# 启动

注：由于 s0s 需在评测机上进行评测，且该评测机不能联网，因此在编译的时候添加了 --offline 选项，而在本地启动运行的时候需要删除 os/Makefile 以及 user/Makefile 中 cargo build 后面的 --offline 选项，以便下载所需依赖。

另：s0s 启动时默认执行一系列系统调用测试后关机，因此若需要进入 shell 进行交互，需注释掉 user/src/bin/user\_shell.rs 中的 test()。

```
559  #[no_mangle]
    ▶ Run | Debug
560  pub fn main() -> i32 {
561      println!("Rust user shell");
562      let mut line: String = String::new();
563      test();
564      print!("{}", LINE_START);
565      loop {
566          let c: u8 = getchar();
567          match c {
568              LF: u8 | CR: u8 => {
```

# qemu

进入 os 目录，执行 make run 即可在 qemu 上运行

# k210

先准备 sdcard，之后进入 os 目录执行 make sdcard 初始化 sd 卡，烧写文件系统镜像。

之后将 sd 卡插入 k210，并连接 k210 开发板，执行 `make run BOARD=k210` 烧写操作系统镜像，即可启动运行。

## 操作系统镜像

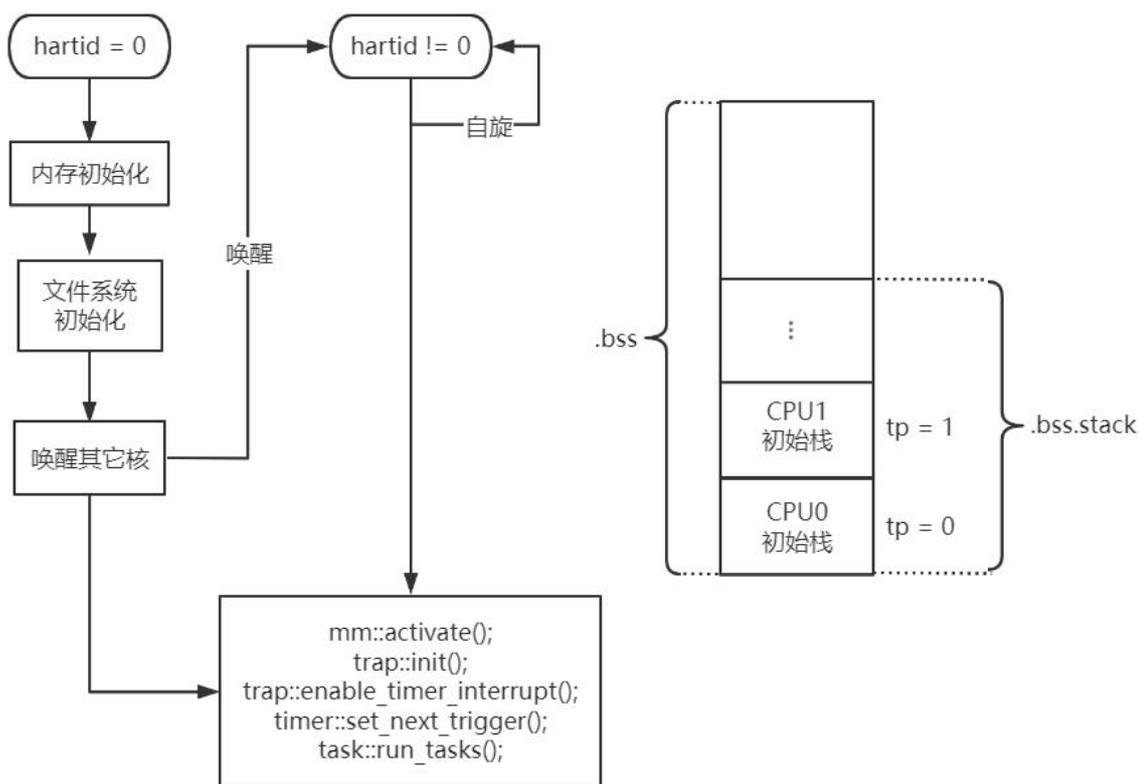
在项目根目录下执行 `make all` 可在根目录下获得操作系统镜像 `os.bin`

# 系统设计与实现

## 操作系统启动

经过 `rustsbi`（相当于 `bootloader`）启动后，CPU 控制权移交到我们手中，目前处于 `S` 态。

`rustsbi` 为操作系统的运行提供了环境（`SEE`）和一些接口（`SBI`），我们在 `os/src/sbi.rs` 中进行了封装。首先对操作系统进行初始化，刚得到控制权时，`a0` 寄存器保存着 CPU 核 id（`hartid`），我们将 `hartid` 保存在 `tp` 寄存器中，在 `trap` 上下文中对 `tp` 寄存器进行保存，一旦进入内核，`tp` 寄存器即存储着 `hartid`，而切换到用户态，恢复 `tp` 寄存器的值后，其值可能被使用更改。之后为每一个核分配了初始栈（该栈后续用作 `idle` 控制进程的内核栈），然后进行初始化：首先让 `hartid` 不为 0 的其它核自旋，而在 `hartid = 0` 的核中对内存、文件系统进行了初始化，然后运行 `initproc.rs` 初始化程序，该程序启动 `user_shell.rs` 程序完成 `shell` 的启动。之后我们就可以唤醒其它核，每个核分别对 `trap`、`timer` 等进行设置之后开始运行。



注：这里使用一个全局的原子变量来在 `hart0` 初始化完成之前卡住其他 `hart`，但是它是放在 `.bss` 段的，在裸机环境下，其他 `hart` 会在 `hart0` 将 `.bss` 段清零之前就访问这个原子变量，可能产生错误的结果，可能需要重新考虑多核的启动流程。

## 进程管理

## 进程控制块

SOS 使用 `TCB` 表示一个进程，其结构如下：

```
pub struct TaskControlBlock {
    // immutable
    pub pid: PidHandle,
    pub ppid: usize,
    pub kernel_stack: KernelStack,
    // mutable
    inner: Mutex<TaskControlBlockInner>,
}

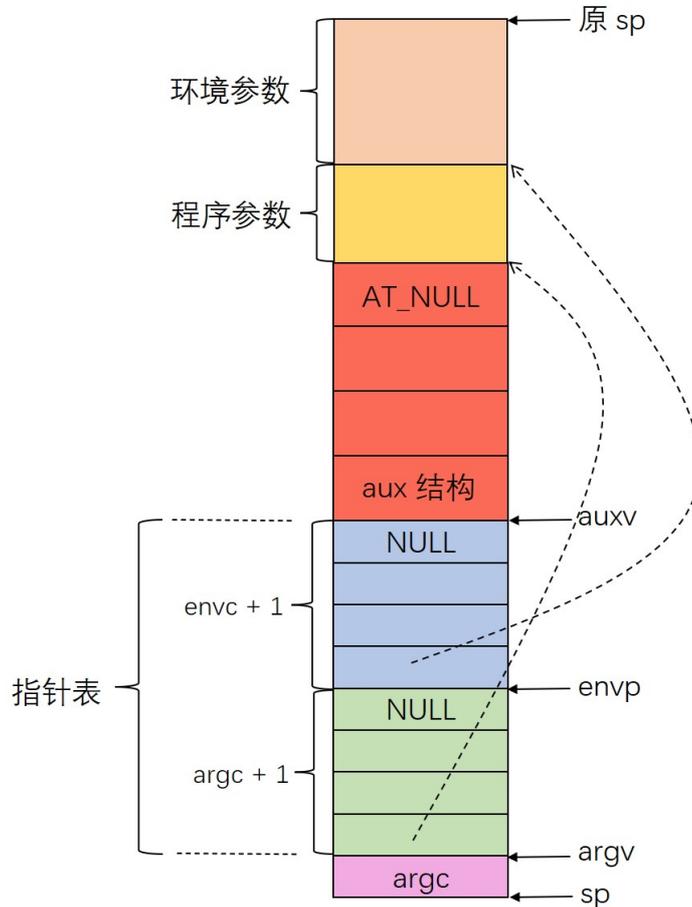
pub struct TaskControlBlockInner {
    pub trap_cx_ppn: PhysPageNum,
    pub base_size: usize,
    pub task_cx: TaskContext,
    pub task_status: TaskStatus,
    pub memory_set: MemorySet,
    pub brk: usize,
    pub mmap_top: usize,
    pub parent: Option<weak<TaskControlBlock>>,
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
    pub fd_table: Vec<Option<FileClass>>,
    pub cwd: Arc<OSInode>,
    pub signals: SignalFlags,
    pub signal_mask: SignalFlags,
    // the signal which is being handling
    pub handling_sig: isize,
    // Signal actions
    pub signal_actions: SignalActions,
    // if the task is killed
    pub killed: bool,
    // if the task is frozen by a signal
    pub frozen: bool,
    pub trap_ctx_backup: Option<TrapContext>,
}
```

`TCB` 使用了内部可变性的设计，将不可变部分与可变部分分离开来，但对外则显现不可变性。以下仅介绍几个需注意的成员变量：

- `pid` 字段使用资源获取即初始化（RAII）进行设计，这样将进程号作为一种资源绑定到该变量上，当其生命周期结束时，则自动回收该资源。相反，父进程号 `ppid` 则单纯的使用整型变量进行表示。
- `brk` 变量表示用户数据区的最高位地址。
- `mmap_top` 表示 `mmap` 映射空间的最高位地址。
- `children` 和 `parent` 变量分别表示子进程和父进程，其中对于子进程进行共享引用，对于父进程则采用弱引用，这样可以防止形成引用循环导致内存泄漏。
- `fd_table` 是文件描述符表，在原来 `rcore` 的设计中，`TCB` 的文件描述符表保存的类型为动态类型 `dyn File + Send + Sync`，而有的系统调用需针对具体的文件类型，但 `rust` 中不支持对动态类型进行强制转换，无法使用具体的文件类型所特有的函数方法，因此将文件描述符表的类型进行细化分离，设置为一个枚举，之后针对具体的类型实现相应的系统调用。

```
pub enum FileClass {
    OSInode(Arc<OSInode>),
    PIPE(Arc<Pipe>),
    Other(Arc<dyn File + Send + Sync>),
}
```

在加载可执行文件中，我们使用如下的栈格式对用户栈进行初始化：



## 任务管理器

我们将任务/进程交给 `TaskManager` 和 `Processor` 管理，其中 `TaskManager`（任务管理器）管理着待执行的进程，而 `Processor` 表示当前运行的进程，一个核就有一个 `Processor` 实例。因此，对于每个尚未结束的进程的任务控制块都只能被引用一次，要么在任务管理器中，要么则是在代表 CPU 处理器的 `Processor`。其结构如下：

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

pub struct Processor {
    current: Option<Arc<TaskControlBlock>>,
    idle_task_cx: TaskContext,
}
```

我们为每一个核实例化一个 `TaskManager` 和 `Processor` 结构，在进程调度上，我们使用多队列调度算法，每个核维护自己的进程队列，针对自己管理的进程使用 `FIFO` 算法进行调度，不同核之间对于调度上没有联系，这样虽然可能出现负载均衡的问题，但防止了某些问题的发生。

在每个核的调度上，我们将等待就绪的进程放入队列中，之后调度的时候从队列中取出一个进程进行切换。而在进程切换时，并不是直接将当前进程切到待执行的进程中，而是增加了一个中间结点 `idle` (`idle` 不断循环查找下一个进程，其使用的是操作系统的启动栈)，先将当前进程切换到 `idle`，之后再从 `idle` 切换到待执行的进程。这样其它正常进程调度换出时只需直接调用 `schedule`，而不用去关心调度的事情。使得调度相关的数据不会出现在进程内核栈上，也使得调度机制对于换出进程的 `Trap` 执行流是不可见的。

我们使用以下接口对进程进行操作：

```
// 添加任务
pub fn add_task(task: Arc<TaskControlBlock>)

// 获取任务
pub fn fetch_task() -> Option<Arc<TaskControlBlock>>

// 获取当前运行的任务
pub fn current_task() -> Option<Arc<TaskControlBlock>>

// 调度
pub fn schedule(switched_task_cx_ptr: *mut TaskContext)
```

## 多核管理

在多核的实现上，并没有增加过多的结构，最主要的是解决因多核而出现的问题，现总结一些麻烦的问题：

1. 进程切换的时候将当前进程添加到空闲队列中等待执行，但此时还没完成进程的切换操作（先 `add_task`，之后再 `switch`）。于是将进程加入队列的那一刻另一个核马上选择该进程执行，则两个核同时进入了该进程的内核栈，导致 `panic`。这只会出现在单队列调度中，将调度换为多队列调度即可。
2. 进程回收资源时出现复杂的死锁。在处理进程的子进程时，将当前进程的子进程挂到 `initproc` 进程中，此时会获得当前进程和 `initproc` 进程的锁，之后会依次获得子进程的锁。而在多核中，当父子进程同时处于 `exit` 中时，父进程的 `exit` 处理需要获得本身和 `initproc` 进程的锁，子进程也需要获得本身和 `initproc` 进程的锁；因此若父进程先获得本身和 `initproc` 进程的锁，子进程获得本身的锁时：父进程由于等待将子进程挂到 `initproc` 进程上，需要获得子进程的锁；而子进程等待 `initproc` 进程的锁，此时产生了死锁。解决方法是先将共有资源即 `initproc` 进程的锁先获得，再进行其它的锁的处理。
3. `sys_wait4` 回收子进程时，发现回收的子进程的引用计数不为1，说明除了本身还有其它地方引用了该子进程。而这有时并不是错误，还是多核的时序问题。在多核条件下，由于待回收的子进程先改变状态再 `drop`，但在改变状态后另一个运行父进程的核立马进行了对子进程的回收，此时子进程的引用计数还是2。而同样在 `sys_wait4` 加入 `println` 语句时由于隐含的睡眠延迟，使程序正确执行。因此当出现引用计数不正确时，我们将当前进程挂起等待一段时间后再次进行处理。

## 内存管理

`s0s` 刚启动的时候并没有开启分页机制，此时直接访问物理地址，而当开启分页机制后，所访问的均为虚拟地址。为了在分页机制开启的前后时刻，能正常访问相应的内容，将内核地址空间的一部分进行恒等映射，也即此时的虚拟地址等于物理地址。同时，我们在数据段上分配了一段空间用作内核堆，并交给伙伴分配器管理，这样我们即可在内核中使用动态存储结构，方便内核的实现。

我们对物理页进行如下抽象，同样使用 `RAII` 进行设计，一旦 `ppn` 生命周期结束，则回收物理页资源。

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

我们将除内核使用以外的内存进行管理，使用栈式页帧管理策略对内存页进行分配回收，提供了如下接口：

```
// 分配内存页
pub fn frame_alloc() -> Option<FrameTracker>

// 回收内存页
pub fn frame_dealloc(ppn: PhysPageNum)
```

对于逻辑段我们抽象成以下结构

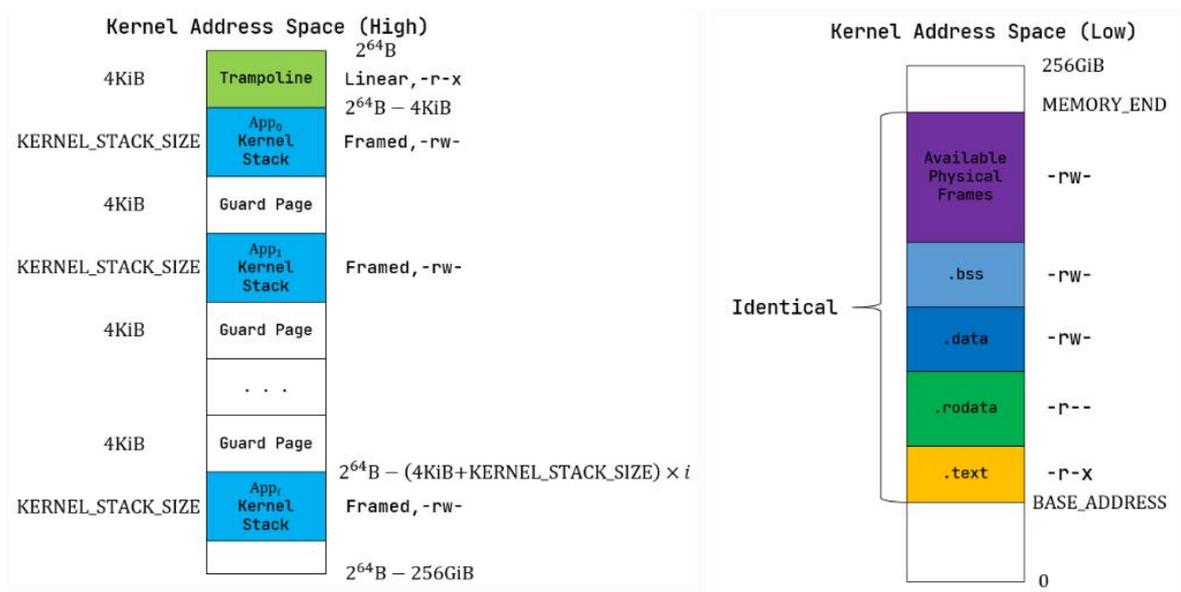
```
pub struct MapArea {
    vpn_range: VPNRange,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    map_type: MapType,
    map_perm: MapPermission,
}
```

之后再进一步封装成地址空间结构

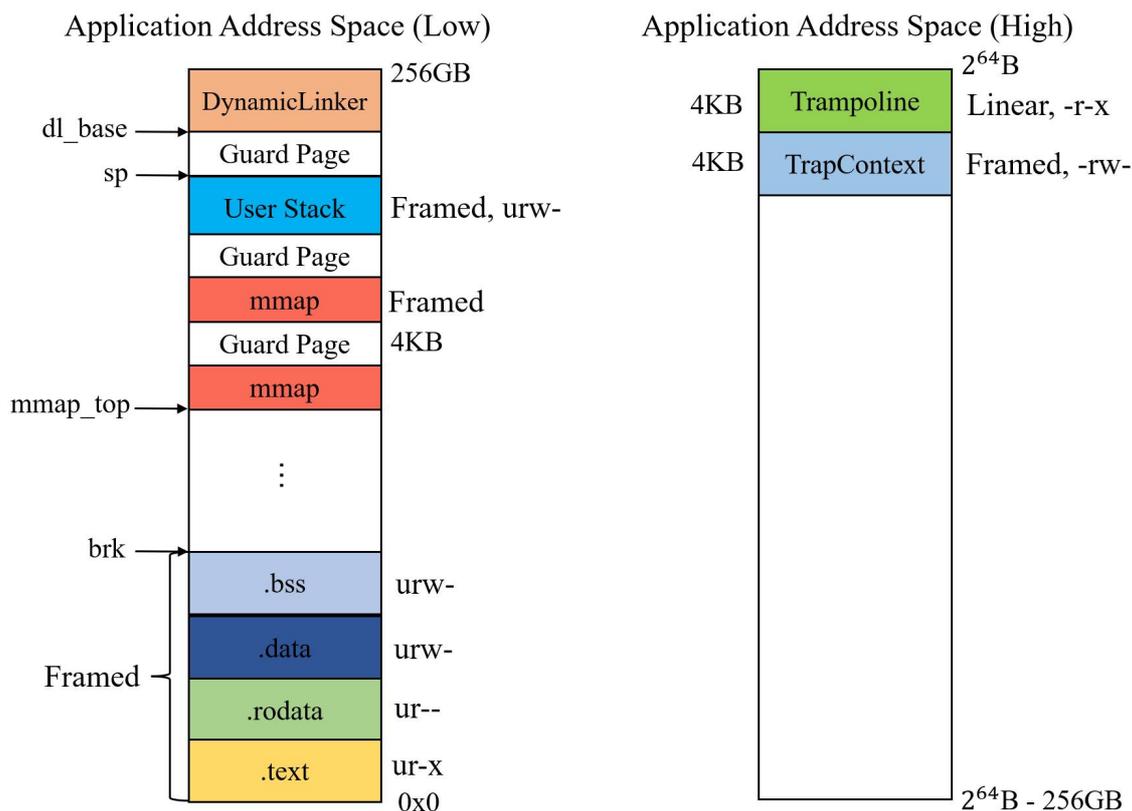
```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}
```

其中 `page_table` 维护页表节点，而 `areas` 则维护具体的页数据，这样一个进程所使用的页都进行了统一的管理，并进一步作为成员变量存储在进程控制块中。

s0s 将内核与用户地址空间分离，其分布如图。我们在地址空间的最高位设置了一个跳板，用于进程切换的平滑过度。用户的次高页面保存的是 trap 的上下文，并不是保存在内核栈中。如果要保存 trap 到内核栈中，需要内核地址空间的 `token` 写入到 `satp` 寄存器中，同时还要跳转到内核栈，因此需要更改 `sp`。而这就需要保存用户的这两个寄存器到临时寄存器中，但只有 `sscratch` 一个寄存器进行周转，所以只能将用户 trap 上下文保存到用户地址空间中。



## 内核地址空间分布



## 用户地址空间分布

在 `rCore` 原来的设计中，所有的页数据默认进行了页面对齐，而这可能造成数据的缺失错误，因此我们对页表的数据复制中添加了 `offset` 偏移量，从而正确处理页表的数据。

```
pub fn copy_data(&mut self, page_table: &mut PageTable, data: &[u8], offset: usize) {
    assert_eq!(self.map_type, MapType::Framed);
    let mut start: usize = 0;
    let mut current_vpn: VirtPageNum = self.vpn_range.get_start();
    let len: usize = data.len();

    // 页表数据不对齐，先处理不对齐的部分
    let src: &[u8] = &data[..len.min(PAGE_SIZE - offset)];
    let dst: &mut [u8] = &mut page_table &mut PageTable
        .translate(current_vpn) Option<PageTableEntry>
        .unwrap() PageTableEntry
        .ppn() PhysPageNum
        .get_bytes_array()[offset..offset + src.len()];
    dst.copy_from_slice(src);
    start += PAGE_SIZE - offset;
    current_vpn.step();
}
```

## 文件系统

`s0s` 实现了 `fat32` 文件系统，其中有些结构参考了 `ultraos`。类似于 `rCore` 的简单文件系统，向外声明了块设备抽象接口，这样只要内核实现了相应的接口，就可以使用该文件系统的功能。于是文件系统与内核进行了解耦，这样可以在更方便的用户态的情况下对文件系统进行测试，之后再接入内核。同时将文件系统大致分为了五层结构，可方便进行设计与实现。

`fat32` 文件系统自下而上可分为如下五层结构：

- 磁盘块设备接口层 (`fat32/src/block_dev.rs`): 读写磁盘块设备的 `trait` 接口。

- 块缓存层 (fat32/src/block\_cache.rs): 位于内存的磁盘块数据缓存
- 磁盘数据结构层 (fat32/src/dir\_entry.rs, fat32/src/layout.rs): 表示磁盘文件系统的数据结构
- 磁盘块管理器层 (fat32/src/fat32\_manager.rs): 实现对磁盘文件系统的管理
- 索引节点层 (fat32/src/vfs.rs): 实现文件创建/文件打开/文件读写等操作

## 磁盘块设备接口层

该层定义了如下 trait 接口，所有块设备实现了该接口，即可调用文件系统相关的功能。

```
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]);
}
```

## 块缓存层

在该层缓存了磁盘块，其中每个缓存块与磁盘块大小一样，都为 512B，总缓存大小为  $16 * 512B = 8K$ 。缓存块使用队列进行管理，在替换缓存块的时候使用类似 LRU 的算法，从队列头开始查找未被使用或已经使用完毕的缓存块进行替换，对于脏块则进行写回操作。同时对缓存块管理器设置了 `start_sec` 变量表示起始扇区号，这样在后续操作时则可以使用逻辑号进行磁盘块的查找。该层对外开放了如下接口：

```
// 获取缓存块
pub fn get_block_cache(block_id: usize, block_device: Arc<dyn BlockDevice>,) ->
Arc<RwLock<BlockCache>>

// 同步所有缓存块
pub fn block_cache_sync_all()

// 设置缓存块管理器起始扇区号
pub fn set_start_sec(start_sec: usize)
```

## 磁盘数据结构层

该层用来实现 fat32 文件系统所需的结构，包括目录项、BPB、FSInfo 扇区以及 FAT 表。

### 目录项

fat32 文件系统使用目录项表示一个文件，因此对目录项的结构设计尤其重要。在目录项中，分为短文件目录项和长文件目录项，我们在目录项结构中保存整个目录项内容 (0x20 字节)，这样可以方便读取和写回文件；同时提供接口读取目录项的内容，对于目录项的操作，我们将其放置在 `inode` 节点中。

```
pub struct ShortDirEntry {
    name: [u8; 8], // 删除时第0位为0xE5，未使用时为0x00。有多余可以用0x20填充
    extension: [u8; 3],
    pub attribute: u8,
    winnt_reserved: u8,
    _creation_tenths: u8, // 创建时间的 10 毫秒位
    creation_time: u16,
    creation_date: u16,
    last_acc_date: u16,
    cluster_high: u16,
    modification_time: u16,
    modification_date: u16,
```

```

cluster_low: u16,
pub size: u32,
}

pub struct LongDirEntry {
    // use Unicode !!!
    // 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 (OR) 运算”的结果写入该位置。
    // 长文件名以 \0 结尾, 其它位置填充 0xff
    pub order: u8, // 删除时为0xE5
    name1: [u8; 10], // 5characters
    pub attribute: u8, // should be 0x0F
    _reserved: u8,
    pub checksum: u8,
    name2: [u8; 12], // 6characters
    _zero: [u8; 2],
    name3: [u8; 4], // 2characters
}

```

对于长文件目录项, 可以说其只是附着于短文件目录项, 因此对于长文件目录项的文件名解析尤为重要, 我们在相应的接口中实现了该功能; 而对于短文件目录项, 则提供接口读写相应的字段。其中需注意文件首簇号的读取, 文档规定有些代表根目录的目录项 (., ..) 其首簇号设置为 0, 虽然我们的实现并没有按这些要求, 但通过挂载得到的文件系统镜像就不一定了, 为了保证鲁棒性, 我们对其进行了转换, 如下。

```

pub fn first_cluster(&self) -> u32 {
    match ((self.cluster_high as u32) << 16) + (self.cluster_low as u32) {
        0 => 2,
        first_cluster => first_cluster,
    }
}

```

## BPB

BPB 结构如下, 由于构建文件系统时, 我们只需读取一次 BPB 结构, 因此不同于 `ultraos`, 将 BPB 分开来, 并存储整个块; 我们一次读入完整的 DBR 块, 并从中提取所需的信息, 之后保存为 BPB 结构, 在构建磁盘块管理器的时候, 将信息保存到磁盘块管理器结构体之后, 即可释放掉 BPB 结构。

```

pub struct BPB {
    pub bytes_per_sector: u16, // 每扇区字节数
    pub sectors_per_cluster: u8, // 每簇扇区数
    pub reserved_sectors: u16, // 保留扇区数
    pub fats: u8, // FAT 表数目
    pub total_sectors: u32, // 总扇区数
    pub sectors_per_fat: u32, // 一个 FAT 表扇区数
    pub root_clusters: u32, // 第一个目录簇号
    pub fsinfo_sector: u16, // fsinfo扇区的扇区号
}

```

## FSInfo

由于 `FSInfo` 扇区功能较少，在 `FSInfo` 结构上我们不对其进行相关的映射，而是仅保留其所在扇区的位置信息。在需要的时候读取相关的块即可。需注意的是在该扇区偏移 492 的字段中并不保存下一个可用簇号，而是设置为最后一个分配的簇号。提供如下接口：

```
// 读取空闲簇数
pub fn read_free_clusters(&self, block_device: Arc<dyn BlockDevice>) -> u32

// 写空闲簇数
pub fn write_free_clusters(&self, free_clusters: u32, block_device: Arc<dyn
BlockDevice>)

// 读取最后一个分配的簇号
pub fn read_first_free_cluster(&self, block_device: Arc<dyn BlockDevice>) -> u32

// 更新最后一个分配的簇号
pub fn write_first_free_cluster(&self, start_cluster: u32, block_device: Arc<dyn
BlockDevice>)
```

## FAT

涉及到 `FAT` 表时，需要注意相关簇号的设置，如下所示，我们以 `FREE_CLUSTER` 表示空闲簇号，`BAD_CLUSTER` 表示坏簇。而由于不同系统下结尾簇号的表示不一致，在我们的实现中，以 `END_CLUSTER` 表示簇的结尾，而为了统一不同系统下结尾簇的差异，保证鲁棒性，使用 `BOUND_CLUSTER` 表示一个边界，任何大于或等于该簇号的簇号将被认为是结尾簇。

```
pub const FREE_CLUSTER: u32 = 0x00000000;
pub const BOUND_CLUSTER: u32 = 0x0FFFFFF8;
pub const END_CLUSTER: u32 = 0x0FFFFFFF;
pub const BAD_CLUSTER: u32 = 0x0FFFFFF7;
```

同时为了实现的方便，我们并没有实现对备用 `FAT` 表进行管理，也即只使用第一个 `FAT` 表。`FAT` 表结构如下：

```
pub struct FAT {
    fat1_sector: u32, // FAT1起始扇区
    sectors: u32,    // 一个 FAT 表的大小
    entries: u32,   // 每扇区有多少个 FAT 表项
}
```

对于 `FAT` 表，其功能是对文件数据区的搜索以及簇的分配，我们对此提供了丰富的接口。

```
// 搜索下一可用簇
pub fn next_free_cluster(&self, current_cluster: u32, block_device: Arc<dyn
BlockDevice>) -> u32

// 查询当前簇的下一个簇
pub fn get_next_cluster(&self, cluster: u32, block_device: Arc<dyn BlockDevice>)
-> u32

// 设置当前簇的下一个簇
pub fn set_next_cluster(&self, cluster: u32, next_cluster: u32, block_device:
Arc<dyn BlockDevice>)
```

```

// 从 start_cluster 开始的第 index 个簇号
pub fn get_cluster_at(&self, start_cluster: u32, index: u32, block_device: Arc<dyn
BlockDevice>) -> u32

// 获得从 start_cluster (含) 开始的所有簇链
pub fn get_all_clusters(&self, start_cluster: u32, block_device: Arc<dyn
BlockDevice>) -> Vec<u32>

// 获取文件最后一个簇号
pub fn get_final_cluster(&self, start_cluster: u32, block_device: Arc<dyn
BlockDevice>) -> u32

// 计算文件所用簇的大小
pub fn count_cluster_num(&self, start_cluster: u32, block_device: Arc<dyn
BlockDevice>) -> u32

```

## 磁盘块管理层

该层实现对磁盘文件系统的管理，具体实现 `FAT32Manager` 文件管理器结构，包括簇的分配回收以及针对 fat32 文件系统特有的长短文件名转换等功能。文件管理器结构如下：

```

pub struct FAT32Manager {
    block_device: Arc<dyn BlockDevice>,
    fsinfo: Arc<FSInfo>,
    fat: Arc<RwLock<FAT>>,
    pub sectors_per_cluster: u32,
    pub bytes_per_sector: u32,
    pub bytes_per_cluster: u32,
    root_sec: u32, // 数据区的起始扇区
}

```

这里需注意 `fat` `fsinfo` 这两个成员变量的实现，一个进行了加锁，而另一个没有：由于 `FSInfo` 结构只保存 `fsinfo` 扇区地址，所有的操作都通过查找相应的块完成，因此可以不对 `fsinfo` 添加锁（因为相应的缓存块进行了加锁操作）。而 `FAT` 结构在操作时还经过一些其它处理计算，当 `fat` 没有加锁，只对块缓冲加锁并且其操作完成时，可能转到其它线程执行，此时会破坏已有的结构（如当前请求的块缓冲被使用而导致继续使用该块的结果进行计算而产生错误），因此对 `fat` 进行了加锁。

我们提供了 `pub fn open(block_device: Arc<dyn BlockDevice>) -> Arc<RwLock<Self>>` 接口用来打开一个文件系统，并返回文件系统管理器，在该接口实现中，正如前面所说，只读取了一次 `BPB` 结构，获取相应信息之后即释放，此后不再读取。而在开发板调试过程中，发现内核一开始因对齐异常产生错误，无法启动，而在 `qemu` 上能正常运行。后来发现 `k210` 对于内存对齐有严格的限制，访存相应的类型其地址应相对应对齐，如 `u32` 应对齐 32 位。而在 `MBR/DBR` 结构中，有很多字段并不满足相应的对齐要求，因此将相应的字段看作一个字节数组，之后手动从该数组中还原出相应的数据。

```

let start_sector: u32 =
    get_block_cache(block_id: 0, block_device: Arc::clone(self: &block_device))
        .read() RwLockReadGuard<BlockCache>
        .read(offset: 0x1c6, f: |sector: &[u8; 4]| {
            let mut start_sec: u32 = 0;
            for i: usize in 0..4 {
                let tmp: u32 = sector[i] as u32;
                start_sec = start_sec + (tmp << (8 * i));
            }
            start_sec
        });

```

我们提供如下接口对簇进行分配和回收：

```

pub fn alloc_cluster(&self, num: u32) -> u32
pub fn dealloc_cluster(&self, clusters: Vec<u32>)

```

另外提供如下接口对长短文件名进行相应的操作：

```

// 拆分文件名和后缀
pub fn split_name_ext<'a>(&self, name: &'a str) -> (&'a str, &'a str)

// 判断一个文件名是否符合短文件名
pub fn is_short_name(&self, name: &str) -> bool

// 将短文件名格式化为目录项存储的内容
pub fn short_name_format(&self, name: &str) -> ([u8; 8], [u8; 3])

// 将长文件名拆分格式化，末尾补0，其余填补0xff
pub fn long_name_format(&self, name: &str) -> Vec<[u8; 13]>

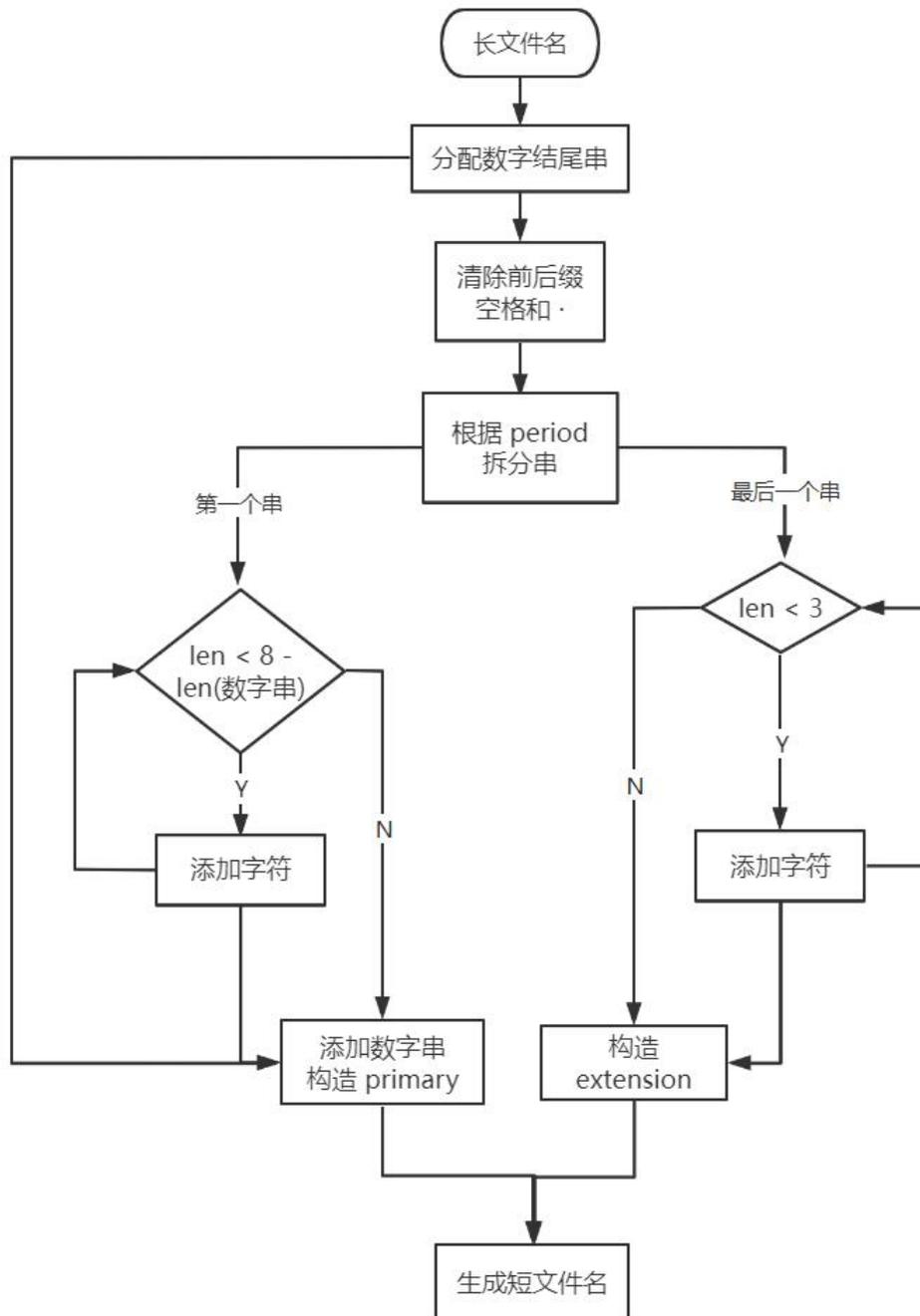
// 由长文件名生成短文件名
pub fn generate_short_name(&self, long_name: &str) -> String

```

对于由长文件名生成短文件名的算法，文档并没有具体的规定实现，只有大概的流程，我们的算法只支持 ASCII 码，同时为了简化步骤并保证在文件名空间内不发生冲突，采用如下算法：

1. 分配数字结尾串（从 ~1 逐渐递增，该步骤保证生成的文件名不发生冲突）
2. 去除长文件名所有的前后缀空格及 .（称为 period）
3. 将长文件名根据 period 进行拆分
  - 对于第一个拆分串按如下操作进行 primary name 的生成：遍历第一个串的字符，若还有字符且 primary name 的长度小于 8 - 数字结尾串的长度，则添加到 primary 中，否则结束并在 primary 末尾添加数字结尾串
  - 对于最后一个拆分串（不同于第一个，若有），按如下操作进行 extension 的生成：遍历最后一个串的字符，若还有字符且 extension 的长度小于 3，则添加到 extension 中。
  - 需注意除数字结尾串之外所有字符均大写

通过该算法则生成符合短文件名格式且与文件名空间不冲突的短文件名，流程图如下。



## 索引节点层

索引节点层实现文件创建/文件打开/文件读写等操作，该层对文件进行抽象，形成 Inode 节点。该层对上层提供了 Inode 的抽象，并提供相应的操作接口，所有的文件操作均是通过其完成，而为了区分不同文件，我们需要找到识别文件的标志。不同于 Linux0.11 的 minix 文件系统，文件本身就通过 Inode 节点号进行区分；fat32 文件系统并没有相应的设计，其一个文件被分割成目录项和数据区两部分，且对于目录文件来说，其可能对应几个目录项（目录文件本身目录项以及 ., .. 目录项），因此不能通过目录项来区分文件，同样由确定目录项的扇区和偏移量也不行。而一个文件可以由簇号识别，对应文件的数据区，所以我们采用通过首簇号区分文件。而对于新创建的文件来说，其可能未被分配存储空间，这时我们采用绝对路径对文件进行区分。

对于普通文件来说，可能需要更改文件大小、时间等信息，而这存储在目录项中。因此 Inode 存储文件对应的目录项，同时保存父目录的首簇号以及该目录项在父目录的偏移，从而确定该目录项的位置，这样设计也是为了便于获得文件名以及文件路径。而由于目录项所需更改的字段较少（目前只有 size 这一字段）且所占空间不大，将其复制到内存，并设置 dirt 位，只在必要的时候进行更新。Inode 节点结构如下：

```

pub struct Inode {
    dir_first_cluster: u32,           // 文件父目录数据的首簇号
    dir_offset: usize,                // 文件短目录项在文件父目录数据区的偏移
    short_dir_entry: ShortDirEntry,  // 文件短目录项
    dirt: bool,                       // 短目录项的脏位（判断短目录项是否有改变，而不是文件
    // 数据）
    fs: Arc<RwLock<FAT32Manager>>,
    fat: Arc<RwLock<FAT>>,
    block_device: Arc<dyn BlockDevice>,
}

```

对于根目录，其 `dir_first_cluster = 0`，`dir_offset` 没有意义。虽然成员 `fs` 包含 `fat` 以及 `block_device`，但将其分离出来方便处理。

我们为 `Inode` 结构实现了 `Drop trait`，这样当节点生命周期结束时，可以自动回收资源，对更改过的目录项则进行写回操作。对于相应的接口实现，我们需注意：在查找的时候不能直接根据文件名判断它是以长文件还是短文件形式存储，然后分类查找。因为有的系统即使为短文件也以长文件形式存储（如 `linux` 挂载 `fat32` 文件系统），这时根据长文件形成的短文件名不一定与本文件名相同。同样也不能提前根据文件名来对文件名进行相应的格式转换。

在该层中，我们提供了丰富的接口供调用，并提供了详细的文档，具体可以参见代码的实现。

## 测试

`fat32` 架构设计的一个优点在于它可以在 `Rust` 应用开发环境（Windows/macOS/Ubuntu）中，按照应用程序库的开发方式来进行测试，不必过早的放到内核中测试运行。众所周知，内核运行在裸机环境上，对其进行调试很困难。而面向应用的开发环境对于调试的支持更为完善，从基于命令行的 `GDB` 到 `IDE` 提供的图形化调试界面都能给文件系统的开发带来很大帮助。另外一点是，由于 `fat32` 需要放到在裸机上运行的内核中，使得 `fat32` 只能使用 `no_std` 模式，不能在 `fat32` 中调用标准库 `std`。但是在把 `fat32` 作为一个应用的库运行的时候，可以暂时让使用它的应用程序调用标准库 `std`，这也会在开发调试上带来一些方便。

`fat32` 的测试放在另一个名为 `fat32-fuse` 的应用程序中，不同于 `fat32`，它是一个可以调用标准库 `std` 的应用程序，能够在 `Rust` 应用开发环境上运行并很容易调试。在 `fat32-fuse/main.rs` 中，我们对文件系统进行了相应的测试，测试的内容包括根目录、创建文件、目录文件、复杂路径、随机读写等。将文件系统镜像保存为 `fat32.img` 并存放在 `fat32-fuse` 目录下，使用命令 `cargo test` 即可进行文件系统功能的测试。

## 接入内核

我们将文件系统提供的 `Inode` 节点进一步封装为 `OSInode` 节点接入内核

```

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

pub struct OSInodeInner {
    offset: usize,
    pub inode: Arc<RwLock<Inode>>,
}

```

对于 `osInode`，我们使用了内部可变性的设计，这样我们可以在对外不可变或仅在不可变借用的情况下对内部的变量进行修改。这样对于共享的引用不用在外面再套上一层锁，可以对锁的粒度最小化。不过对于 `Inode` 节点，我们并没有这样处理，于是有的地方的实现较为复杂，后面可以针对此进行改进。

我们向外提供如下接口对文件进行打开和创建：

```
/// 获取根结点
pub fn root_inode() -> Arc<OSInode>

/// 打开指定目录下的文件
pub fn open_file(dir: Arc<OSInode>, name: &str, flags: OpenFlags) ->
Option<Arc<OSInode>>

/// 根据路径打开文件
pub fn open_file_at(fd: isize, path: &str, flags: OpenFlags) ->
Option<Arc<OSInode>>

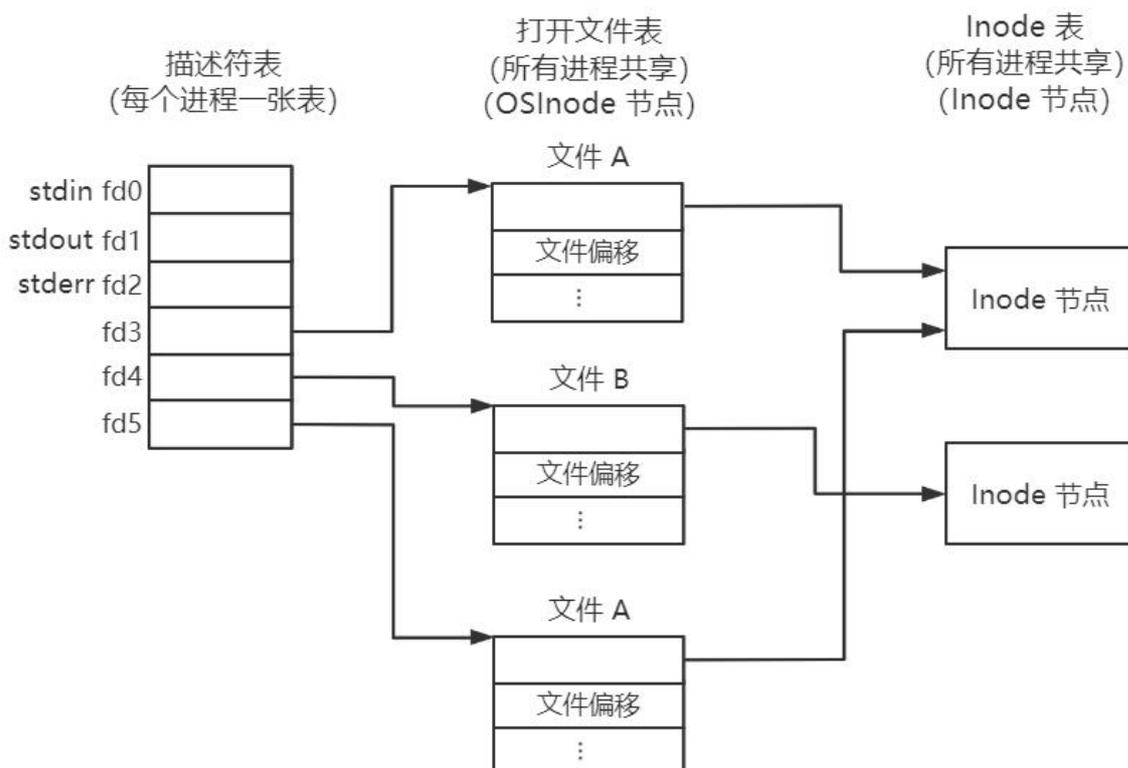
/// 创建目录
pub fn mkdir_at(fd: isize, path: &str) -> isize

/// 移除指定文件的链接(可用于删除文件)
pub fn unlinkat(fd: isize, path: &str) -> isize
```

在内核中，我们引入了两个表：`inode` 表和打开文件表。

- `inode` 表：整个系统只有一份，存储文件的 `Inode` 结点，且每个节点只有一份。如果该表在 `vfs.rs` 中进行管理，则什么时候进行创建或删除无法确定，因此可以在需要打开文件的时候进行处理，这时能够保证所获得的文件都是需要的，且应该加进表中。
- 打开文件表：整个系统只有一份，存储打开的文件的 `osInode` 节点。可以打开相同的文件，这时会在打开文件表存储两个 `osInode` 节点，但这两个结点指向的 `Inode` 节点相同，只是文件偏移量不同。根节点存在 0 号位置。

两个表关系如图所示，



我们提供了如下接口对两个表进行相应的操作：

```
/// 向 inode 表插入结点
pub fn insert_inode_table(inode: Inode) -> Arc<RwLock<Inode>>

/// 向打开文件表插入结点
pub fn insert_file_table(readable: bool, writable: bool, inode: Inode) ->
Arc<OSInode>

/// 移除 inode 表不被使用的结点
pub fn remove_from_inode_table()

/// 移除打开文件表不被使用的结点
pub fn remove_from_file_table()
```

## 总结

---

本项目使用 `rust` 开发了一个基于 `RISC-V` 架构的操作系统内核，支持启动初始化、中断、I/O、进程管理、内存管理、执行文件解析、`fat32` 文件系统功能等，同时能够在真实硬件平台 `K210` 上运行。在开发过程中，学习了 `rust` 语言及其特性，了解了 `RISC-V` 架构，对于操作系统内核也有了进一步的理解。在构建文件系统的过程中，更加深入的了解了相关锁的机制，对锁的运用有了进一步的体会；而在多核的实现中，也逐步加深了对死锁的了解和解决办法；在加载可执行文件中，明白了文档规范的重要性，对系统调用的实现也更加规范化。不过，本项目有很多地方做了简化，也有些无法解决的问题，仍需要不断地进行改进优化。

## 改进

---

- 添加 `cow` 功能
- 完善多核功能，使其能够在开发板上运行
- 对于 `fat32` 文件系统，可能需要重新考虑锁，并改进 `inode` 节点，进行内部可变性设计
- 添加交换内存功能

## 致谢

---

非常感谢刘国军老师的支持和指导以及陈天宇同学给予的帮助。