

实验七

实验七

题目7-1：文件系统——文件访问

实验目的

实验内容

实验设计原理

open () 函数

close()函数

read()函数

write()函数

lseek()函数

实验步骤

实验结果及分析

程序代码

题目7-2：设计文件系统

实验目的

实验内容：

实验设计原理

块设备接口层

块缓存层

数据结构

缓冲块

缓冲块管理器

磁盘数据结构层

磁盘布局

位图数据结构

实际布局的代码

磁盘块管理器

索引节点/VFS

在用户态测试文件系统

实验结果及分析

题目7-1：文件系统——文件访问

实验目的

学习使用文件访问系统调用函数，了解各调用具体功能和底层原理。

实验内容

利用create、open、close、read、write等文件访问系统调用创建文件，并源文件中内容拷贝至目标文件。

实验设计原理

open () 函数

功能描述：用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。
所需头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

函数原型：

```
int open(const char *pathname, int flags, int perms)
```

参数：

`pathname` :被打开的文件名（可包括路径名如"dev/ttyS0"）

`flags` :文件打开方式,

`O_RDONLY` :以只读方式打开文件

`O_WRONLY` :以只写方式打开文件

`O_RDWR` :以读写方式打开文件

`O_CREAT` :如果改文件不存在，就创建一个新的文件，并用第三个参数为其设置权限

`O_EXCL` :如果使用`O_CREAT`时文件存在，则返回错误消息。这一参数可测试文件是否存在。此时`open`是原子操作，防止多个进程同时创建同一个文件

`O_NOCTTY` :使用本参数时，若文件为终端，那么该终端不会成为调用`open()`的那个进程的控制终端

`O_TRUNC` :若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为0

`O_APPEND` :以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾，即将写入的数据添加到文件的末尾

`O_NONBLOCK` : 如果`pathname`指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的I/O操作设置非阻塞方式。

`O_SYNC` :使每次`write`都等到物理I/O操作完成。

`O_RSYN` :`read` 等待所有写入同一区域的写操作完成后再进行

在`open()`函数中，`flags`参数可以通过"|"组合构成，但前3个标准常量（`O_RDONLY`，`O_WRONLY`，和`O_RDWR`）不能互相组合。

`perms` :被打开文件的存取权限，可以用两种方法表示，可以用一组宏定义：`S_I(R/W/X)`

(`USR/GRP/OTH`),其中`R/W/X`表示读写执行权限，

`USR/GRP/OTH`分别表示文件的所有者/文件所属组/其他用户,如`S_IRUUR|S_IWUUR|S_IXUUR`, (`-rex-----`),也可用八进制`800`表示同样的权限

返回值：

成功：返回文件描述符

失败：返回-1

close()函数

功能描述：用于关闭一个被打开的文件

所需头文件：

```
#include <unistd.h>
```

函数原型：

```
int close(int fd)
```

参数: fd文件描述符

函数返回值: 0成功, -1出错

read()函数

功能描述: 从文件读取数据。

所需头文件:

```
#include <unistd.h>
```

函数原型:

```
ssize_t read(int fd, void *buf, size_t count);
```

参数:

fd: 将要读取数据的文件描述词。

buf: 指缓冲区, 即读取的数据会被放到这个缓冲区中去。

count: 表示调用一次**read**操作, 应该读多少数量的字符。

返回值: 返回所读取的字节数;

0 (读到EOF) ; -1 (出错) 。

以下几种情况会导致读取到的字节数小于 **count**:

1. 读取普通文件时, 读到文件末尾还不够 **count** 字节。
例如: 如果文件只有 30 字节, 而我们想读取 100字节, 那么实际读到的只有 30 字节, **read** 函数返回 30 。此时再使用 **read** 函数作用于这个文件会导致 **read** 返回 0 。
2. 从终端设备 (terminal device) 读取时, 一般情况下每次只能读取一行。
3. 从网络读取时, 网络缓存可能导致读取的字节数小于 **count**字节。
4. 读取 **pipe** 或者 **FIFO** 时, **pipe** 或 **FIFO** 里的字节数可能小于 **count** 。
5. 从面向记录 (record-oriented) 的设备读取时, 某些面向记录的设备 (如磁带) 每次最多只能返回一个记录。
6. 在读取了部分数据时被信号中断。

读操作始于 cfo 。在成功返回之前, cfo 增加, 增量为实际读取到的字节数。

write()函数

功能描述: 向文件写入数据。

所需头文件:

```
#include <unistd.h>
```

函数原型:

```
ssize_t write(int fd, void *buf, size_t count);
```

返回值:

写入文件的字节数 (成功) ; -1 (出错)

功能: write 函数向 filesdes 中写入 count 字节数据, 数据来源为 buf 。返回值一般总是等于 count, 否则就是出错了。常见的出错原因是磁盘空间满了或者超过了文件大小限制。

对于普通文件, 写操作始于 cfo 。如果打开文件时使用了 O_APPEND, 则每次写操作都将数据写入文件末尾。成功写入后, cfo 增加, 增量为实际写入的字节数。

lseek()函数

功能描述： 用于在指定的文件描述符中将文件指针定位到相应位置。

所需头文件：

```
#include <unistd.h>, #include <sys/types.h>
```

函数原型：

```
off_t lseek(int fd, off_t offset, int whence);
```

参数：

`fd`:文件描述符

`offset`:偏移量，每一个读写操作所需要移动的距离，单位是字节，可正可负（向前移，向后移）

`whence`：

- SEEK_SET:当前位置为文件的开头，新位置为偏移量的大小
- SEEK_CUR:当前位置为指针的位置，新位置为当前位置加上偏移量
- SEEK_END:当前位置为文件的结尾，新位置为文件大小加上偏移量的大小

返回值：

成功：返回当前位移

失败：返回-1

实验步骤

使用open函数创建源文件后用write函数写数据。使用open函数创建目标文件后调用lseek函数将源文件的读写指针移到起始点，接上read读取源文件内容的while循环。最终调用close关闭源文件和目标文件流。

实验结果及分析

实验五 多核多线程编程 详情

剩余体验时间: 02:33:55

结束体验

体验手册 云产品资源 实验报告

体验云账号, 创建资源后生成

收起

子用户名: u-7k6b05d@1915471854401972

子用户密码: Cs8tZ7Pq8Nd2Vn4E

AK ID: LTAI5tGg2DxMr6qpQSUZcQv

AK Secret: 1K5VepsSgyYDOWVMZx87fCDEQ...

注意: 若登录子账号, 需打开RAM控制台进行登录。

一键复制子账号登录链接

ECS服务器

磁盘ID: d-uf66fp06fjk3y0e5iwn

ECS公网地址: 47.100.102.6

ECS登录名: root

登录密码: En8Gy0Vg6A

ECS实例ID: i-uf66xvzy226tqbhtsf

IP白名单: 0.0.0.0/0

地域: 华东2(上海)

2. root@iZuf66xvzy226tqbhtsf: ~

2. root@iZuf66xvzy226tqbhtsf: ~

```

#define BUFFER_SIZE 128
#define SRC_FILE_NAME "src_file.txt"
#define DEST_FILE_NAME "dest_file.txt"
#define OFFSET 0

int main()
{
    int src_file,dest_file;
    unsigned char src_buff[BUFFER_SIZE];
    unsigned char dest_buff[BUFFER_SIZE];
    int real_read_len = 0;
    char str[BUFFER_SIZE] = "this is a testabout\nopen()\nclose()\nwrite()\nread()\nlseek()\nend of the file\n";
    src_file=open(SRC_FILE_NAME, O_RDWR|O_CREAT,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
    if(src_file<0)
    {
        printf("open file error!!!\n");
        exit(1);
    }
    write(src_file, str, sizeof(str));
    dest_file=open(DEST_FILE_NAME, O_RDWR|O_CREAT,
    S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
    if(dest_file<0)
    {
        printf("open file error!!!\n");
        exit(1);
    }
    lseek(src_file, OFFSET, SEEK_SET);
    while((real_read_len=read(src_file, src_buff, sizeof(src_buff)))>0)
    {
        printf("src_file:%s", src_buff);
        write(dest_file,src_buff,real_read_len);
    }
    lseek(dest_file, OFFSET, SEEK_SET);
    while((real_read_len=read(dest_file,dest_buff,sizeof(dest_buff)))>0);
    printf("dest_file:%s", dest_buff);
    close(src_file);
    close(dest_file);
    return 0;
}

```

46,1 Bot

实验五 多核多线程编程 详情

剩余体验时间: 02:58:33

结束体验

体验手册 云产品资源 实验报告

体验云账号, 创建资源后生成

收起

子用户名: u-yhfutr2j@1915471854401972

子用户密码: Dz2Er9Bt8Zz8S8t8Y

AK ID: LTAI5tAcMDcV85yXz2J9b9aCp

AK Secret: Sa7Vhd6OQJRomiTSRzYDzdCsyyl...

注意: 若登录子账号, 需打开RAM控制台进行登录。

一键复制子账号登录链接

ECS服务器

磁盘ID: d-uf60617c7nq1kt9p82m3

ECS公网地址: 139.196.83.59

ECS登录名: root

登录密码: Vq9Vt3PjON

ECS实例ID: i-uf61htgm781j7gquwyd

IP白名单: 0.0.0.0/0

地域: 华东2(上海)

2. root@iZuf61htgm781j7gquwyd: ~

2. root@iZuf61htgm781j7gquwyd: ~

Last login: Mon Mar 14 15:10:39 2022 from 118.31.243.98

Welcome to Alibaba Cloud Elastic Compute Service !

[root@iZuf61htgm781j7gquwyd ~]# vim a.c

[root@iZuf61htgm781j7gquwyd ~]# gcc a.c -o a

[root@iZuf61htgm781j7gquwyd ~]# ls

a a.c

[root@iZuf61htgm781j7gquwyd ~]# ./a

src_file:this is a testabout

open()

close()

write()

read()

lseek()

end of the file

dest_file:this is a testabout

open()

close()

write()

read()

lseek()

end of the file

[root@iZuf61htgm781j7gquwyd ~]#

源文件及目标文件在所有操作后内容相同。

程序代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include <errno.h>
#define BUFFER_SIZE 128
#define SRC_FILE_NAME "src_file.txt"
#define DEST_FILE_NAME "dest_file.txt"
#define OFFSET 0
int main()
{
    int src_file,dest_file;
    unsigned char src_buff[BUFFER_SIZE];
```

```

    unsigned char dest_buff[BUFFER_SIZE];
    int real_read_len = 0;
    char str[BUFFER_SIZE] = "this is a
testabout\nopen()\nclose()\nwrite()\nread()\nlseek()\nend of the file\n";
    src_file=open(SRC_FILE_NAME,
O_RDWR|O_CREAT,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
    if(src_file<0)
    {
        printf("open file error!!!\n");
        exit(1);
    }
    write(src_file, str, sizeof(str));
    dest_file=open(DEST_FILE_NAME,
O_RDWR|O_CREAT,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
    if(dest_file<0)
    {
        printf("open file error!!!\n");
        exit(1);
    }
    lseek(src_file, OFFSET, SEEK_SET);
    while((real_read_len=read(src_file, src_buff, sizeof(src_buff)))>0)
    {
        printf("src_file:%s", src_buff);
        write(dest_file,src_buff,real_read_len);
    }
    lseek(dest_file, OFFSET, SEEK_SET);
    while((real_read_len=read(dest_file,dest_buff,sizeof(dest_buff)))>0);
    printf("dest_file:%s", dest_buff);
    close(src_file);
    close(dest_file);
    return 0;
}

```

题目7-2：设计文件系统

实验目的

学习文件系统的构成

实验内容：

使用Rust语言实现一个文件系统，并且将一个文件抽象成块设备，在文件上做测试。

在文件系统的设计上，采用松耦合，模块化的设计思想，与底层设备驱动通过抽象接口BlockDevice连接，然后使用Rust提供的alloc crate 隔离了操作系统内核的内存管理，避免了直接调用内存管理的内核函数。避免了直接调用内存管理的内核函数。在底层的驱动上，使用轮询方式来访问virtio_blk的虚拟磁盘设备，从而避免了访问外设中断的内核函数，同时在设计中避免了直接访问进程相关数据和函数，从而隔离了操作系统内核的进程管理。

文件系统本身也划分成不同的层次，形成层次化和模块化的设计架构，从下到上分成五个不同的层次。

1. 磁盘块设备接口层：定义的以块大小为单位对磁盘块设备进行读写的接口
2. 块缓存层：从内存中缓存磁盘块的数据，避免频繁读写磁盘
3. 磁盘数据结构层：磁盘上的超级块，位图，索引节点，数据块，目录项等核心数据结构和相关处理。

4. 磁盘块管理器层：合并了上述核心数据结构和磁盘布局所形成的磁盘文件系统数据结构，以及基于这些结构创建/打卡文件系统的相关处理和磁盘块的分配和回收。
5. 索引节点层：管理索引节点 文件控制块 数据结构，并实现文件创建/文件打开/文件读写等成员函数，来向上支持文件操作相关的系统调用。

实验设计原理

块设备接口层

这对应的是block_dev.rs 文件

块设备接口层实际上是为了统一不同设备，在设备之上的一层抽象，所以代码内容中，只有一个对外的trait(接口)。

```
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]);  
}
```

read_block 方法是为了将编号为 block_id 的块从磁盘读入内存的缓冲区 buf 中。

write_block 方法是为了将内存中的缓冲区 buf 中的数据写入磁盘编号为 block_id 的块。

这个接口属于文件系统的最底层，所以在文件系统中实际上没有一个实现了BlockDevice的具体类型，而是操作系统，或者文件系统的使用者提供这么一个具体类型。

从这里也就可以看出文件系统的泛用性，只要是实现了BlockDevice的块设备驱动程序，不论是单纯的一个文件，还是实际上的操作系统，都可以使用这个文件系统。

实际上的文件系统中，有块和扇区两个概念，在随机读写数据的时候，用的是扇区，在存储文件的时候，用的是块。

通常一个扇区是512字节，而块则是一个或多个扇区，Ext4的文件系统一个块默认是4096字节

在我们自己实现的问题系统中，一个块的大小等于一个扇区等于512字节。

块缓存层

这个层次对应的文件是block_cache.rs

之所以存在缓冲区，是因为操作系统频繁读写速度缓慢的磁盘块会导致系统性能极大的降低。解决这个问题的常见手法就是通过 read_block 方法把数据从磁盘读到内存的缓冲区，后续对这个数据块的访问就可以在内存完成。

考虑到同步性的问题和实现的简易性，在这个文件系统中，我将缓冲区统一管理起来，当要读写一个块的时候，首先去统一缓冲区中，去看这个块是否已经被缓存了。同时，对于实际文件块的读写时机，也可以通过这个全局管理器处理，从而实现尽可能少的 read_block 和 write_block 的调用。

数据结构

缓冲块

```
pub struct BlockCache {
    cache: [u8; BLOCK_SZ], //BLOCK_SZ=512 是块大小，这里是实际内存的缓冲区
    block_id: usize, //记录这个块缓存来自于磁盘的那一块
    block_device: Arc<dyn BlockDevice>, //此处是BlockDevice 是底层块设备的引用，用来最后写到内存上，也就是块的读写。
    modified: bool, //记录这个块从磁盘载入内存缓冲后，是否被修改过
}
```

这个块的构造函数是

```
pub fn new(block_id: usize, block_device: Arc<dyn BlockDevice>) -> Self {
    let mut cache = [0u8; BLOCK_SZ];
    block_device.read_block(block_id, &mut cache);
    Self {
        cache,
        block_id,
        block_device,
        modified: false,
    }
}
```

在构造函数中，首先创建了一个可变的数组做cache，然后从block_device中，通过 block_id 读一个块的内容到这个cache中，最后返回这个BlockCache

为了能够正常的使用缓冲区，还有如下的代码

```
//得到一个BlockCache内部的缓冲区中指定偏移量的offset的字节地址
fn addr_of_offset(&self, offset: usize) -> usize {
    &self.cache[offset] as *const _ as usize
}
//一个泛型方法，可以获取缓冲区中位于偏移量offset的一个类型为T的磁盘的数据结构的不可变引用，通过Sized的泛型约束，保证了T类型是一个可以在编译时获取大小的类型
pub fn get_ref<T>(&self, offset: usize) -> &T
where
    T: Sized,
{
    let type_size = core::mem::size_of::<T>(); //此处首先取出这个类型的大小
    assert!(offset + type_size <= BLOCK_SZ); //断言当前的缓冲区确实包含了整个数据结构
    let addr = self.addr_of_offset(offset); //断言成功后，从这个偏移量取出对应的缓冲区字节地址
    unsafe { &*(addr as *const T) } //返回这个不可变引用
}
//同上，但是这个返回的是一个可变引用，也就是调用这个方法是为了修改内部数据
pub fn get_mut<T>(&mut self, offset: usize) -> &mut T
where
    T: Sized,
{
    let type_size = core::mem::size_of::<T>();
    assert!(offset + type_size <= BLOCK_SZ);
    self.modified = true; //因为是个可变引用，所以此处直接将modified修改为true，表明这个缓冲区之后需要同步到磁盘
    let addr = self.addr_of_offset(offset);
    unsafe { &mut *(addr as *mut T) }
}
```



```

//以下进一步封装的调用，作用是在offset处获取一个类型为T的磁盘上的数据结构的不可变/可变引用，然后在这个引用上执行传入的闭包f执行的操作，FnOnce移交了调用者的所有权，由编译器保证只执行一次，此处的impl可以被理解为泛型类型参数的简单语法，但是实际上采用这种impl语法会导致不能在调用的时候使用turbo-fish语法指定T的具体类型
pub fn read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V {
    f(self.get_ref(offset))
}
pub fn modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&mut T) -> V) -> V {
    f(self.get_mut(offset))
}

//用于同步数据，将modified修改为false，然后写会磁盘
pub fn sync(&mut self) {
    if self.modified {
        self.modified = false;
        self.block_device.write_block(self.block_id, &self.cache);
    }
}

//Drop类似于C++的析构函数，这里意味着当BlockCache被释放的时候，调用sync，处理缓冲块中的数据。
impl Drop for BlockCache {
    fn drop(&mut self) {
        self.sync()
    }
}

```

整个BlockCache体现了RAII的思想，获取即初始化，释放就处理好所有后续，使用局部对象管理声明周期。

缓冲块管理器

```

pub struct BlockCacheManager {
    queue: VecDeque<(usize, Arc<Mutex<BlockCache>>>>),
}

```

//这个实际的类型是 (usize, Arc<Mutex<BlockCache>>>) 的元组，是块编号和块缓存。通过 <Arc<Mutex>> 同时提供了共享引用和互斥访问，保证了多线程的安全，之所以使用共享引用，是因为块缓存需要在Manager中保留一个引用，同时还需要返回给使用者来做读写。

//Arc是一个原子引用计数，Rc不是原子的，所以不能用于多线程
//为了避免性能惩罚，实际上只需要给需要修改的成员变量加上<Mutex>

通过一个双端队列管理缓冲区。

此处使用FIFO实现这个缓冲区管理。

```

//构造函数如下
pub fn new() -> Self {
    Self {
        queue: VecDeque::new(),
    }
}

```

下面是其他的方法

```

//从Manager中获取一个块缓存
pub fn get_block_cache(
    &mut self,
    block_id: usize,
    block_device: Arc<dyn BlockDevice>,
) -> Arc<Mutex<BlockCache>> {

    if let Some(pair) = self.queue.iter().find(|pair| pair.0 == block_id) {
        //首先迭代queue并查找目前所需的block是否已经在管理器中，如果存在就直接返回，此处的1是指返回的是Arc的引用
        Arc::clone(&pair.1)
    } else {
        //不存在于Manager中
        //下面是此时Manager队列已满了，实际上这个双端队列是可以无限增长的，但是为了保证内存空间不被过度占用，同时保证Manager的查找效率，所以限制了BLOCK_CACHE_SIZE
        if self.queue.len() == BLOCK_CACHE_SIZE {
            // from front to tail
            if let Some((idx, _)) = self
                .queue
                .iter()
                .enumerate()//下面这一行寻找一个只有一个引用的磁盘缓存
                .find(|(_, pair)| Arc::strong_count(&pair.1) == 1)
            {
                //丢弃找到的缓存块，此处 ..= 运算符的作用 x..=y 从x到y，并且包括y，此处实际上就是idx，之所以这么写是因为drain的参数类型为range
                self.queue.drain(idx..=idx);
            } else {
                //所有的缓冲块都有不止一个引用，没有可以丢弃的
                panic!("Run out of BlockCache!");
            }
        }
        // 上面的代码已经保证了现在一定有一个位置提供给新的block_cache，所以直接push_back到队列中

        let block_cache = Arc::new(Mutex::new(BlockCache::new(
            block_id,
            Arc::clone(&block_device),
        )));
        self.queue.push_back((block_id, Arc::clone(&block_cache)));
        block_cache
    }
}

```

之后是暴露在外，实际给用来操作 block_manager的代码

```

//懒加载宏，直到使用Manager的时候才构造这个Manager此处不使用Arc是因为全局只有一个Manager，不存在多个所有权的问题。
lazy_static! {
    pub static ref BLOCK_CACHE_MANAGER: Mutex<BlockCacheManager> =
        Mutex::new(BlockCacheManager::new());
}

//此处的dyn 指的是对响应BlockDevice的调用是动态分配的，此处是一个接口而不是泛型，实际上此处已经BlockDevice的实际类型信息，并且包含了两个指针项，一个指向数据，一个指向此Trait对应的方法。同时保证了BlockDevice是一个动态安全的Trait，最易懂但不全面的动态安全，理解就是这个BlockDevice保证 所有的这个对象内部所有数据不受Sized约束，可以被简单的分成两个指针。

//首先对Block_CACHE上锁，然后代理这个get_block_cache，通过块id获取数据到block_device中，同时由于返回的是一个Mutex，所以保证了使用者调用 lock 方法后再调用 read,modify来访问缓冲区。
pub fn get_block_cache(

```

```

        block_id: usize,
        block_device: Arc<dyn BlockDevice>,
    ) -> Arc<Mutex<BlockCache>> {
        BLOCK_CACHE_MANAGER
            .lock()
            .get_block_cache(block_id, block_device)
    }
    //此处用于同步所有的块
    pub fn block_cache_sync_all() {
        let manager = BLOCK_CACHE_MANAGER.lock();
        for (_, cache) in manager.queue.iter() {
            cache.lock().sync();
        }
    }
}

```

磁盘数据结构层

对应的代码是 layout.rs 和 bitmap.rs

此处的功能是将一个逻辑上的文件目录树结构映射到磁盘上，并且决定磁盘上每个块应该存储文件哪些数据，这也是文件系统最重要的功能。

磁盘布局

首先是一个Super-Block，通过Magic Number提供文件系统合法性检查，同时用于定位其他连续区域的位置。

第二个区域是一个索引节点的位图，长度为若干个块，记录后面索引节点区域哪些索引节点已经被分配，哪些没有被分配。

第三个区域是索引节点区域，每个块都储存了若干索引节点。

第四个区域是一个数据块位图，长度是若干个块，记录哪些数据块已经被分配了，哪些没有被分配。

第五个区域是数据块区域，是实际上保存的数据。

索引节点实际上就是 inode，也是上文代码中写到的 id，这个 inode 包含了stat命令能看到的信
息，也保存了实际的文件/目录的数据块的索引信息。

位图数据结构

```

pub struct Bitmap {
    start_block_id: usize,
    blocks: usize,
}

```

位图实际上的数据结构如上图

```

pub fn new(start_block_id: usize, blocks: usize) -> Self {
    Self {
        start_block_id,
        blocks,
    }
}

```

在生成一个位图的时候，根据初始磁盘id和块的个数来生成这片区域。

为了快速的访问位图，Bitmap数据结构实际上是在内存中驻留的，实际上写在磁盘上的位图则是 `type BitmapBlock = [u64; 64]`；一个磁盘块 被解释成一个长度为64的 64位数的数组， $64 \times 64 = 4096$ bit 是一组数据

下面是分配一个bit的代码

```
pub fn alloc(&self, block_device: &Arc<dyn BlockDevice>) -> Option<usize> {
    //首先遍历自身包含的区域
    for block_id in 0..self.blocks {
        let pos = get_block_cache(
            block_id + self.start_block_id as usize, //注意此处加上了
start_block_id
            Arc::clone(block_device),
        )
        .lock()
        .modify(0, |bitmap_block: &mut BitmapBlock| { //此处modify的偏移量之所以是
0, 是因为目前在操作的整片区域上只有一个文件块就是BitmapBlock。传递进去的闭包显式声明了
BitmapBlock类型，否则modify内部调用的get_mut不知道应该用那个类型来解析数据。

            //bitmap_block
            if let Some((bits64_pos, inner_pos)) = bitmap_block
                .iter()
                .enumerate() //此处是在遍历每个u64，当这个u64 !=u64::MAX ,也就是不是所有
位都被置1时
                .find(|(&, bits64)| **bits64 != u64::MAX)//下面的map是对所有找到的
有0 的u64,通过 trailing_ones置为一。
                .map(|(bits64_pos, bits64)| (bits64_pos, bits64.trailing_ones()
as usize))
            {
                // 在完成了上述操作后，
                bitmap_block[bits64_pos] |= 1u64 << inner_pos;
                Some(block_id * BLOCK_BITS + bits64_pos * 64 + inner_pos as
usize) //此处对应的是函数的返回值Option<usize>，返回了第一个非1的位置
            } else {
                //没有找到，返回None
                None
                //上面 Some和None的值实际上是 let pos = 的返回值。
            }
        });
        //注意这个if 还在 前面 for block_id 的遍历里，放在这里的作用是在上面找到了一个pos
后，就直接返回，避免了更多的时间消耗
        if pos.is_some() {
            return pos;
        }
    }
    //完全没有找到，返回None
    None
}
```

回收一个bit

```
fn decomposition(mut bit: usize) -> (usize, usize, usize) {
    let block_pos = bit / BLOCK_BITS;
    bit %= BLOCK_BITS; //在一块中的绝对位置
    (block_pos, bit / 64, bit % 64) //块号，组号，组内编号
}
```

```

}
pub fn dealloc(&self, block_device: &Arc<dyn BlockDevice>, bit: usize) {
    let (block_pos, bits64_pos, inner_pos) = decomposition(bit); //计算出当前的bit对应的block ID, 以及在这块Block的哪一个u64组, 最后是组内编号。
    get_block_cache(block_pos + self.start_block_id, Arc::clone(block_device))
        .lock()
        .modify(0, |bitmap_block: &mut BitmapBlock| {
            //通过位运算清零, 下面是断言这个位置确实有数据
            assert!(bitmap_block[bits64_pos] & (1u64 << inner_pos) > 0);
            bitmap_block[bits64_pos] -= 1u64 << inner_pos;
        });
}

```

实际布局的代码

首先是SuperBlock

```

pub struct SuperBlock {
    magic: u32, //用于做合法性校验
    pub total_blocks: u32, //此处是文件系统所占的块总数, 并不一定是设备的块总数, 一个设备上实际上可以有多个文件系统
    //下面的四个则是后面的四个区域所占的块数
    pub inode_bitmap_blocks: u32,
    pub inode_area_blocks: u32,
    pub data_bitmap_blocks: u32,
    pub data_area_blocks: u32,
}

```

下面是Super Block的方法

```

//此处初始化的时候, 每个块的大小都是从外部传入的, 因为这里的划分是上层磁盘块管理器需要做的操作
pub fn initialize(
    &mut self,
    total_blocks: u32,
    inode_bitmap_blocks: u32,
    inode_area_blocks: u32,
    data_bitmap_blocks: u32,
    data_area_blocks: u32,
) {
    *self = Self {
        magic: EFS_MAGIC,
        total_blocks,
        inode_bitmap_blocks,
        inode_area_blocks,
        data_bitmap_blocks,
        data_area_blocks,
    }
}
//此处用来判断完整性
pub fn is_valid(&self) -> bool {
    self.magic == EFS_MAGIC
}

```

磁盘上的索引节点

每个块的编号用一个u32存储

```
#[derive(PartialEq)]
pub enum DiskInodeType {
    File,
    Directory,
}
#[repr(C)] //表明按照C 的内存布局
pub struct DiskInode {
    pub size: u32, //文件, 目录内容的字节数
    pub direct: [u32; INODE_DIRECT_COUNT], //文件很小的时候, 所用的直接索引。当
    INODE_DIRECT_COUNT=28时, 这里可以获取到 28块, 每块是512字节, 所以总共是14Kib
    pub indirect1: u32, //指向是一个一级索引块 *注意, 此处是一个512字节的整块,
    512*8/32=128个数据块=64Kib*
    pub indirect2: u32, //通过二级索引, 最多可以索引8MiB的内容 *也是一个512字节,
    512*8/32=128个一级, 128*64Kib=8Mib*
    type_: DiskInodeType,
}
//32+32*28+32*2+32=1024=128*8
//但是这里的 DiskInodeType类型, 其实sizeof之后是u8只有八个字节, 但是因为前面使用的repr(C)使用
了C的内存布局, 所以DiskInode是128字节。
```

DiskInode的对应实现的方法

首先是比较简单的初始化方法, 通过传入的type_值, 将一个DiskInode初始化位一个文件或目录。

```
pub fn initialize(&mut self, type_: DiskInodeType) {
    self.size = 0;
    self.direct.iter_mut().for_each(|v| *v = 0);
    self.indirect1 = 0;
    self.indirect2 = 0;
    self.type_ = type_;
}
```

然后还有极其简单的 is_file 和 is_dir 两个方法, 如下是计算为了容纳自身 self.size 的内容, 所需的块数。实际上就是文件大小/块大小, 然后向上取整。

```
pub fn data_blocks(&self) -> u32 {
    self::_data_blocks(self.size)
}
fn _data_blocks(size: u32) -> u32 {
    (size + BLOCK_SZ as u32 - 1) / BLOCK_SZ as u32
}
```

total_blocks 用来计算包含索引块, 实际上需要的块数, 然后 blocks_num_needed 用来计算将一个DiskInode扩容, 需要的新的块数。

```
pub fn total_blocks(size: u32) -> u32 {
    let data_blocks = self::_data_blocks(size) as usize;
    let mut total = data_blocks as usize;
    // indirect1
    if data_blocks > INODE_DIRECT_COUNT {
```

```

        total += 1;
    }
    // indirect2
    if data_blocks > INDIRECT1_BOUND {
        total += 1;
        // sub indirect1
        total +=
            (data_blocks - INDIRECT1_BOUND + INODE_INDIRECT1_COUNT - 1) /
            INODE_INDIRECT1_COUNT;
    }
    total as u32
}
pub fn blocks_num_needed(&self, new_size: u32) -> u32 {
    assert!(new_size >= self.size);
    self::total_blocks(new_size) - self::total_blocks(self.size)
}

```

索引块最重要的是索引功能，索引函数如下

```

pub fn get_block_id(&self, inner_id: u32, block_device: &Arc<dyn BlockDevice>) -
> u32 {
    let inner_id = inner_id as usize;
    //根据Block的id, 来判断利用哪一级的索引
    if inner_id < INODE_DIRECT_COUNT {
        self.direct[inner_id]
    } else if inner_id < INDIRECT1_BOUND {
        get_block_cache(self.indirect1 as usize, Arc::clone(block_device))
            .lock()//此处获取了一个512字节的整块，然后在read传入的闭包中，将这个整块解析为
            一个u32的数组，然后截断并获取id 512/4*32/8=512字节
        //type IndirectBlock = [u32; BLOCK_SZ=512 / 4];
        .read(0, |indirect_block: &IndirectBlock| {
            indirect_block[inner_id - INODE_DIRECT_COUNT]
        })
    } else {
        let last = inner_id - INDIRECT1_BOUND;
        let indirect1 = get_block_cache(self.indirect2 as usize,
        Arc::clone(block_device))
            .lock()
            .read(0, |indirect2: &IndirectBlock| {
                indirect2[last / INODE_INDIRECT1_COUNT]
            });
        //先获取一级索引，然后索引到数据块
        get_block_cache(indirect1 as usize, Arc::clone(block_device))
            .lock()
            .read(0, |indirect1: &IndirectBlock| {
                indirect1[last % INODE_INDIRECT1_COUNT]
            })
    }
}

```

为了让索引块可以索引到数据块，我们需要添加一个 `increase_size` 方法

```

pub fn increase_size(
    &mut self,
    new_size: u32,
    new_blocks: Vec<u32>, //保存了当前块所需块的编号的向量

```

```

        block_device: &Arc<dyn BlockDevice>,
    ) {
        let mut current_blocks = self.data_blocks(); //当前数据块大小
        self.size = new_size;
        let mut total_blocks = self.data_blocks(); //在上面更新了new_size, 此处计算出的是
        新的总数据块大小
        let mut new_blocks = new_blocks.into_iter(); //new_blocks是传入的新的部分
        // 所用的最大块数小于直接索引的块, 直接填满直接索引 *填充, 直到填满/达到了total的要求*
        while current_blocks < total_blocks.min(INODE_DIRECT_COUNT as u32) {
            self.direct[current_blocks as usize] = new_blocks.next().unwrap();
            current_blocks += 1;
        }
        // 只填满直接索引还不够
        if total_blocks > INODE_DIRECT_COUNT as u32 {
            //确认直接索引已经填满
            if current_blocks == INODE_DIRECT_COUNT as u32 {
                //分配出一个间接索引块
                self.indirect1 = new_blocks.next().unwrap();
            }
            current_blocks -= INODE_DIRECT_COUNT as u32;
            total_blocks -= INODE_DIRECT_COUNT as u32;
            //减去直接索引的块数。
        } else {
            return;
        }
        // 通过indirect1获取一个块, 然后将其解释为一个u32的数组, 将其填充。
        get_block_cache(self.indirect1 as usize, Arc::clone(block_device))
            .lock()
            .modify(0, |indirect1: &mut IndirectBlock| {
                while current_blocks < total_blocks.min(INODE_INDIRECT1_COUNT as
u32) {
                    indirect1[current_blocks as usize] = new_blocks.next().unwrap();
                    current_blocks += 1;
                }
            });
        // 二级索引
        if total_blocks > INODE_INDIRECT1_COUNT as u32 {
            if current_blocks == INODE_INDIRECT1_COUNT as u32 {
                self.indirect2 = new_blocks.next().unwrap();
            }
            current_blocks -= INODE_INDIRECT1_COUNT as u32;
            total_blocks -= INODE_INDIRECT1_COUNT as u32;
        } else {
            return;
        }
        // 将 (a0, b0) 转换为 (a1, b1) b是二级索引中的一级索引块号
        let mut a0 = current_blocks as usize / INODE_INDIRECT1_COUNT;
        let mut b0 = current_blocks as usize % INODE_INDIRECT1_COUNT;
        let a1 = total_blocks as usize / INODE_INDIRECT1_COUNT;
        let b1 = total_blocks as usize % INODE_INDIRECT1_COUNT;
        // 二级索引的分配
        get_block_cache(self.indirect2 as usize, Arc::clone(block_device))
            .lock()
            .modify(0, |indirect2: &mut IndirectBlock| {
                while (a0 < a1) || (a0 == a1 && b0 < b1) {
                    if b0 == 0 {
                        indirect2[a0] = new_blocks.next().unwrap();
                    }
                }
            });
    }
}

```



```

        // 填充当前的一级索引块
        get_block_cache(indirect2[a0] as usize,
Arc::clone(block_device))
            .lock()
            .modify(0, |indirect1: &mut IndirectBlock| {
                indirect1[b0] = new_blocks.next().unwrap();
            });
        // move to next
        b0 += 1;
        if b0 == INODE_INDIRECT1_COUNT {
            b0 = 0;
            a0 += 1;
        }
    }
});
}

```

下面这个函数用来清空文件内容，并且回收数据和索引块，回收到的块编号保存在 `Vec<u32>` 这个向量里。实际实现类似于上面的分配空间-----> 只是清空了size，需要后续对返回的向量继续处理，来实际上的清空数据。或许可以视为懒删除？

```

pub fn clear_size(&mut self, block_device: &Arc<dyn BlockDevice>) -> Vec<u32>;

```

下面则是最激动人心的读写函数的部分。

```

pub fn read_at(
    &self,
    offset: usize,
    buf: &mut [u8], // 在读写的时候，将被读写的数据视为一个字节序列。每次都选取其中的一段区间做操作
    block_device: &Arc<dyn BlockDevice>,
) -> usize {
    let mut start = offset;
    let end = (offset + buf.len()).min(self.size as usize); // 通过当前的这块，知道文件的实际上的大小，同时如果文件剩下的内容还足够多，那么缓冲区会被填满；否则文件剩下的全部内容都会被读到缓冲区中。
    if start >= end {
        return 0;
    }
    let mut start_block = start / BLOCK_SZ; // 目前是文件内部的多少块
    let mut read_size = 0;
    loop {
        // 计算当前的已有数据的最终块
        let mut end_current_block = (start / BLOCK_SZ + 1) * BLOCK_SZ;
        end_current_block = end_current_block.min(end);
        // 读数据
        let block_read_size = end_current_block - start;
        let dst = &mut buf[read_size..read_size + block_read_size];
        get_block_cache(
            self.get_block_id(start_block as u32, block_device) as usize,
            Arc::clone(block_device),
        )
        .lock() // 实际的读操作
        .read(0, |data_block: &DataBlock| {
            let src = &data_block[start % BLOCK_SZ..start % BLOCK_SZ + block_read_size];

```

```

        dst.copy_from_slice(src);
    });
    //移动的下一个block
    read_size += block_read_size;
    if end_current_block == end {
        break;
    }
    start_block += 1;
    start = end_current_block;
}
//返回实际读了多少数据
read_size
}
//写文件保证了 buf中的内容全部被写入块，但是当offset开始的区间小于文件的范围时，需要由调用者提前扩充空间。
pub fn write_at(
    &mut self,
    offset: usize,
    buf: &[u8],
    block_device: &Arc<dyn BlockDevice>,
) -> usize {
    let mut start = offset;
    let end = (offset + buf.len()).min(self.size as usize);
    assert!(start <= end);
    let mut start_block = start / BLOCK_SZ;
    let mut write_size = 0usize;
    loop {
        let mut end_current_block = (start / BLOCK_SZ + 1) * BLOCK_SZ;
        end_current_block = end_current_block.min(end);

        let block_write_size = end_current_block - start;
        get_block_cache(
            self.get_block_id(start_block as u32, block_device) as usize,
            Arc::clone(block_device),
        )
        .lock()
        .modify(0, |data_block: &mut DataBlock| {
            let src = &buf[write_size..write_size + block_write_size];
            let dst = &mut data_block[start % BLOCK_SZ..start % BLOCK_SZ +
block_write_size];
            dst.copy_from_slice(src);
        });
        write_size += block_write_size;

        if end_current_block == end {
            break;
        }
        start_block += 1;
        start = end_current_block;
    }
    write_size
}

```

在索引中，文件和目录是不同的，因为文件只要可以读出来就行了，不用管内部数据的结构，但是目录是需要内部结构的，所以目录块需要特定的格式，将目录块看作目录项的一个序列。

```

//目录项二元组，首个元素是目录下面一个文件/目录的名，另一个项是其对应的节点编号。
#[repr(C)]//同C对齐 一个数据块 512字节，一个目录项32字节 8*28+32=256 256/8=32 所以一个数据块可以存储16个目录项。
pub struct DirEntry {
    name: [u8; NAME_LENGTH_LIMIT + 1],
    inode_number: u32,
}

pub const DIRENT_SZ: usize = 32;

impl DirEntry {
    //空目录项，在VFS中使用
    pub fn empty() -> Self {
        Self {
            name: [0u8; NAME_LENGTH_LIMIT + 1],//文件名的限制，
            inode_number: 0,
        }
    }
    //通过文件名和节点号生成的实际有用目录项。
    pub fn new(name: &str, inode_number: u32) -> Self {
        let mut bytes = [0u8; NAME_LENGTH_LIMIT + 1];
        bytes[..name.len()].copy_from_slice(name.as_bytes());
        Self {
            name: bytes,
            inode_number,
        }
    }
    //由于Inode的read_at/write_at函数需要传入的参数是u8数组，所以此处将其转为字节序列
    pub fn as_bytes(&self) -> &[u8] {
        unsafe { core::slice::from_raw_parts(self as *const _ as usize as *const u8, DIRENT_SZ) }
    }
    pub fn as_bytes_mut(&mut self) -> &mut [u8] {
        unsafe { core::slice::from_raw_parts_mut(self as *mut _ as usize as *mut u8, DIRENT_SZ) }
    }
    pub fn name(&self) -> &str {
        let len = (0usize..).find(|i| self.name[*i] == 0).unwrap();
        core::str::from_utf8(&self.name[..len]).unwrap()
    }
    pub fn inode_number(&self) -> u32 {
        self.inode_number
    }
}

```

磁盘块管理器

对应的文件是efs.rs

这里实现的磁盘块管理器的作用，是为了实现文件系统的磁盘布局，在磁盘数据结构层，虽然提到了磁盘布局的方式，但是代码中其实并没有显示出这个布局。这个布局将在磁盘块管理器实现。

从这层次开始，所有的数据结构都是存放在内存中了。

磁盘管理器的数据结构如下所示

```
pub struct EasyFileSystem {
    pub block_device: Arc<dyn BlockDevice>,
    pub inode_bitmap: Bitmap,
    pub data_bitmap: Bitmap,
    inode_area_start_block: u32,
    data_area_start_block: u32,
}
```

包含了索引节点和数据块两个位图，同时记录了索引节点区域和数据块区域的起始编号，从而可以确定在磁盘上，索引节点和数据块的具体位置。

如下是实际的文件系统创建

```
pub fn create(
    block_device: Arc<dyn BlockDevice>,
    total_blocks: u32,
    inode_bitmap_blocks: u32,
) -> Arc<Mutex<Self>> {
    // 下面是在计算位图合适的大小，计算的目的是求出inode需要多少块才能是得位图中每一个bit都有
    // 对应的实际的inode，剩下的块则分配给数据位图和数据区
    let inode_bitmap = Bitmap::new(1, inode_bitmap_blocks as usize);
    let inode_num = inode_bitmap.maximum();
    let inode_area_blocks =
        ((inode_num * core::mem::size_of::<DiskInode>() + BLOCK_SZ - 1) /
        BLOCK_SZ) as u32;
    let inode_total_blocks = inode_bitmap_blocks + inode_area_blocks;
    let data_total_blocks = total_blocks - 1 - inode_total_blocks; //总的节点减去分
    // 配给inode的节点
    let data_bitmap_blocks = (data_total_blocks + 4096) / 4097;
    let data_area_blocks = data_total_blocks - data_bitmap_blocks;
    let data_bitmap = Bitmap::new(
        (1 + inode_bitmap_blocks + inode_area_blocks) as usize,
        data_bitmap_blocks as usize,
    );
    //此处实际的创建了管理器实例
    let mut efs = Self {
        block_device: Arc::clone(&block_device),
        inode_bitmap,
        data_bitmap,
        inode_area_start_block: 1 + inode_bitmap_blocks,
        data_area_start_block: 1 + inode_total_blocks + data_bitmap_blocks,
    };
    // 先清空所有的节点
    for i in 0..total_blocks {
        get_block_cache(i as usize, Arc::clone(&block_device))
            .lock()
            .modify(0, |data_block: &mut DataBlock| {
                for byte in data_block.iter_mut() {
                    *byte = 0;
                }
            });
    }
    // 初始化超级块
    get_block_cache(0, Arc::clone(&block_device)).lock().modify(
        0,
        |super_block: &mut SuperBlock| {
            super_block.initialize(
```

```

        total_blocks,
        inode_bitmap_blocks,
        inode_area_blocks,
        data_bitmap_blocks,
        data_area_blocks,
    );
},
);

// 创建根目录
assert_eq!(efs.alloc_inode(), 0);
let (root_inode_block_id, root_inode_offset) = efs.get_disk_inode_pos(0); // 因为
是首次分配，所以编号为0
get_block_cache(root_inode_block_id as usize, Arc::clone(&block_device))
    .lock()
    .modify(root_inode_offset, |disk_inode: &mut DiskInode| {
        disk_inode.initialize(DiskInodeType::Directory);
    });
// 将初始化的内容立刻写到磁盘上。
block_cache_sync_all();
Arc::new(Mutex::new(efs))
}

```

如下是文件系统的使用，打开一个块设备，从上面读到文件系统的元数据。同样是返回一个文件系统，create和open的区别是，create通过传入的块设备，总block数，inode位图大小，来在块设备上创建文件系统，一个副作用是在块设备上写操作，而open则是在块设备上读操作，通过super块的数据构造文件系统。

```

pub fn open(block_device: Arc<dyn BlockDevice>) -> Arc<Mutex<Self>> {
    // 读入第一个块，也就是超级块
    get_block_cache(0, Arc::clone(&block_device))
        .lock()
        .read(0, |super_block: &SuperBlock| {
            assert!(super_block.is_valid(), "Error loading EFS!");
            let inode_total_blocks =
                super_block.inode_bitmap_blocks + super_block.inode_area_blocks;
            // 根据读入的超级块，初始化文件系统
            let efs = Self {
                block_device,
                inode_bitmap: Bitmap::new(1, super_block.inode_bitmap_blocks as
                usize),
                data_bitmap: Bitmap::new(
                    (1 + inode_total_blocks) as usize,
                    super_block.data_bitmap_blocks as usize,
                ),
                inode_area_start_block: 1 + super_block.inode_bitmap_blocks,
                data_area_start_block: 1 + inode_total_blocks +
                super_block.data_bitmap_blocks,
            };
            Arc::new(Mutex::new(efs))
        })
}

```

此处的管理器拥有对磁盘数据最完整的了解，所以可以通过位图上的bit编号计算出各个存储inode的块和存储data的块在磁盘上的实际位置。也就是 `get_disk_inode_pos`，`get_data_block_id` 两个函数。

下面是三个管理磁盘数据块的函数，此处没有`dealloc_inode`，是因为目前还不支持文件删除。

```
pub fn alloc_inode(&mut self) -> u32 {
    self.inode_bitmap.alloc(&self.block_device).unwrap() as u32
}

//注意此处返回的是block ID，也就是在块设备上的编号，而不是在数据块区域的ID
pub fn alloc_data(&mut self) -> u32 {
    self.data_bitmap.alloc(&self.block_device).unwrap() as u32 +
    self.data_area_start_block
}

pub fn dealloc_data(&mut self, block_id: u32) {
    get_block_cache(
        block_id as usize,
        Arc::clone(&self.block_device)
    )
    .lock()
    .modify(0, |data_block: &mut DataBlock| {
        data_block.iter_mut().for_each(|p| { *p = 0; })
    });
    self.data_bitmap.dealloc(
        &self.block_device,
        (block_id - self.data_area_start_block) as usize
    )
}
```

由于目前只支持绝对路径，所以这一层还提供了一个函数来简单的访问根目录所在的位置，此处的Inode定义在`vfs`中，初始化的过程是调用`Inode::new`的时候，将`block_id`设置为0，

```
pub fn root_inode(efs: &Arc<Mutex<Self>>) -> Inode {
    let block_device = Arc::clone(&efs.lock().block_device);

    let (block_id, block_offset) = efs.lock().get_disk_inode_pos(0);

    Inode::new(block_id, block_offset, Arc::clone(efs), block_device)
}
```

索引节点/VFS

在Linux世界中，之所以能存在如此多的文件系统，是因为通过`vfs`做了一层抽象，实际上，文件系统的使用者也不关心磁盘布局怎么实现的，更关心的是逻辑上的文件和目录，所以需要Inode之前实现的叫做`DiskInode`，`DiskInode`是写在磁盘块中的，Inode则是放在内存中记录文件索引节点信息的。

这一部分的代码在`vfs.rs`中。

```
pub struct Inode {
    block_id: usize,
    block_offset: usize,
    //上面的两项用于记录Inode对应的DiskInode在磁盘上的位置。
    fs: Arc<Mutex<EasyFileSystem>>, //用于接受文件系统的用户的操作的fs
    block_device: Arc<dyn BlockDevice>,
}
```

以下是两个为了简化对`DiskInode`访问的方法

```

fn read_disk_inode<V>(&self, f: impl FnOnce(&DiskInode) -> V) -> V {
    get_block_cache(self.block_id, Arc::clone(&self.block_device))
        .lock()
        .read(self.block_offset, f)
}

fn modify_disk_inode<V>(&self, f: impl FnOnce(&mut DiskInode) -> V) -> V {
    get_block_cache(self.block_id, Arc::clone(&self.block_device))
        .lock()
        .modify(self.block_offset, f)
}

```

在文件索引方面

此文件系统目前还只有一个目录，所以查找的逻辑较为简单，只是在根目录的目录项中，根据文件名找到文件的inode编号即可。

```

pub fn find(&self, name: &str) -> Option<Arc<Inode>> { //不一定能找到所以是Option,
默认
    let fs = self.fs.lock(); //提前获取了锁，而不是在创建inode块的时候再获取整个的锁
    self.read_disk_inode(|disk_inode| {
        self.find_inode_id(name, disk_inode) //在下面，同时此处的返回值有了None
            .map(|inode_id| { //根据找到的inode_id来构造Inode
                let (block_id, block_offset) = fs.get_disk_inode_pos(inode_id);
                Arc::new(Self::new(
                    block_id,
                    block_offset,
                    self.fs.clone(),
                    self.block_device.clone(),
                ))
            })
    })
}

fn find_inode_id(
    &self,
    name: &str,
    disk_inode: &DiskInode,
) -> Option<u32> {
    //在一个目录节点中寻找文件 *只会在根目录节点上调用*
    assert!(disk_inode.is_dir());
    let file_count = (disk_inode.size as usize) / DIRENT_SZ; //记录了这块里一共
由多少个DirEntry
    let mut dirent = DirEntry::empty(); //这个是目录项的二元组，不是目录节点的意思
    for i in 0..file_count { //遍历这块中的所有 DirEntry
        assert_eq!(
            disk_inode.read_at(
                DIRENT_SZ * i,
                dirent.as_bytes_mut(),
                &self.block_device,
            ),
            DIRENT_SZ,
        );
        if dirent.name() == name { //找到了
            return Some(dirent.inode_number() as u32);
        }
    }
}

```

```

    None
}

```

所有暴露给文件系统使用者的文件系统操作，全程都需要持有文件系统的互斥锁，而文件系统的内部操作，都假设调用自己的线程已经有了锁，所以不会再尝试获取锁。从而保证多核情况下，也只有一个核做文件系统相关操作。

其他给文件系统使用者方法---ls

```

pub fn ls(&self) -> Vec<String> {
    let _fs = self.fs.lock(); //使用 _fs 从而达到获取锁，但是不使用的目的，但是如果用
    _ 的话，则会被编译器给优化掉，从而不能获取锁， .lock(), 获取到的实例时MutexGuard类型，会自己
    解锁

    self.read_disk_inode(|disk_inode| {
        let file_count = (disk_inode.size as usize) / DIRENT_SZ;
        let mut v: Vec<String> = Vec::new();
        for i in 0..file_count {
            let mut dirent = DirEntry::empty();
            assert_eq!(
                disk_inode.read_at(
                    i * DIRENT_SZ,
                    dirent.as_bytes_mut(),
                    &self.block_device,
                ),
                DIRENT_SZ,
            );
            v.push(String::from(dirent.name()));
        }
        v
    })
}

```

给外部使用的方法，create

```

pub fn create(&self, name: &str) -> Option<Arc<Inode>> {
    let mut fs = self.fs.lock();
    let op = |root_inode: &DiskInode| {
        //断言是在 根目录 创建文件
        assert!(root_inode.is_dir());
        //判断文件是否已经被创建了。 *这个方法调用的时候，就已经获取了锁，所以内部没有获取锁*
        self.find_inode_id(name, root_inode)
    };
    //文件已经存在，所以Return None
    if self.read_disk_inode(op).is_some() { //注意这个self 实际上是根目录的Inode，op
    里也断言了这个操作。
        return None;
    }
    //分配一个 inode节点，并初始化
    let new_inode_id = fs.alloc_inode();
    let (new_inode_block_id, new_inode_block_offset) =
    fs.get_disk_inode_pos(new_inode_id);
    get_block_cache(new_inode_block_id as usize, Arc::clone(&self.block_device))
    .lock()
    .modify(new_inode_block_offset, |new_inode: &mut DiskInode| {
        new_inode.initialize(DiskInodeType::File);
    });
}

```



```

    });
    self.modify_disk_inode(|root_inode| {
        // 添加到 根目录节点的 文件名列表中。
        let file_count = (root_inode.size as usize) / DIRENT_SZ;
        let new_size = (file_count + 1) * DIRENT_SZ;
        // 扩大节点大小
        self.increase_size(new_size as u32, root_inode, &mut fs);
        // 写到目录节点上
        let dirent = DirEntry::new(name, new_inode_id);
        root_inode.write_at(
            file_count * DIRENT_SZ,
            dirent.as_bytes(),
            &self.block_device,
        );
    });

    let (block_id, block_offset) = fs.get_disk_inode_pos(new_inode_id);
    block_cache_sync_all();
    // 返回 inode
    Some(Arc::new(Self::new(
        block_id,
        block_offset,
        self.fs.clone(),
        self.block_device.clone(),
    )))
}

```

清空文件内容

```

pub fn clear(&self) {
    let mut fs = self.fs.lock();
    self.modify_disk_inode(|disk_inode| {
        let size = disk_inode.size;
        let data_blocks_dealloc = disk_inode.clear_size(&self.block_device); // 此处是将size置零，但是数据其实还没有被清空，只是返回了一个要清空的数据的向量
        // 断言要清空的数据向量和之前计算出的向量相同。
        assert!(data_blocks_dealloc.len() == DiskInode::total_blocks(size) as
        usize);
        for data_block in data_blocks_dealloc.into_iter() {
            fs.dealloc_data(data_block); // 真实的情况数据
        }
    });
    block_cache_sync_all();
}

```

读写文件

```

// 同样是作用在字节序列的一段区间上。
pub fn read_at(&self, offset: usize, buf: &mut [u8]) -> usize {
    let _fs = self.fs.lock();
    self.read_disk_inode(|disk_inode| disk_inode.read_at(offset, buf,
    &self.block_device))
}

pub fn write_at(&self, offset: usize, buf: &[u8]) -> usize {
    let mut fs = self.fs.lock();

```

```

        let size = self.modify_disk_inode(|disk_inode| {
            self.increase_size((offset + buf.len()) as u32, disk_inode, &mut fs);//注意此处首先进行了扩容，让文件确实可以全部写进去。
            disk_inode.write_at(offset, buf, &self.block_device)
        });
        block_cache_sync_all();
        size
    }
}

```

在用户态测试文件系统

由于在实现文件系统的时候，最底层只是一个实现了BlockDevice Trait的类型，所以可以把任何实现了这个Trait的东西作为最底层，最简单也是最容易想到的实现就是给一个文件添加上这两个Trait，也就是以下的代码代表的东西。

```

struct BlockFile(Mutex<File>);
//通过 Seek实现随机读写功能
impl BlockDevice for BlockFile {
    fn read_block(&self, block_id: usize, buf: &mut [u8]) {
        let mut file = self.0.lock().unwrap();
        file.seek(SeekFrom::Start((block_id * BLOCK_SZ) as u64))
            .expect("Error when seeking!");
        assert_eq!(file.read(buf).unwrap(), BLOCK_SZ, "Not a complete block!");
    }

    fn write_block(&self, block_id: usize, buf: &[u8]) {
        let mut file = self.0.lock().unwrap();
        file.seek(SeekFrom::Start((block_id * BLOCK_SZ) as u64))
            .expect("Error when seeking!");
        assert_eq!(file.write(buf).unwrap(), BLOCK_SZ, "Not a complete block!");
    }
}

```

测试这个文件系统的方法如下

```

fn efs_test() -> std::io::Result<()> {
    let block_file = Arc::new(BlockFile(Mutex::new({
        let f = OpenOptions::new()//设置打开的标志位
            .read(true)
            .write(true)
            .create(true)
            .open("target/fs.img")?;
        f.set_len(8192 * 512).unwrap();
        f
    })));
    EasyFileSystem::create(block_file.clone(), 4096, 1);//先在文件上写入文件系统
    let efs = EasyFileSystem::open(block_file.clone());//然后通过文件系统打开块设备
    let root_inode = EasyFileSystem::root_inode(&efs);//获取根目录
    root_inode.create("filea");//在根目录上创建文件
    root_inode.create("fileb");
    for name in root_inode.ls() {
        println!("{}", name);
    }//打印创建的文件
    let filea = root_inode.find("filea").unwrap();//查找文件，并获取这个文件
    let greet_str = "Hello, world!";
    filea.write_at(0, greet_str.as_bytes());//在filea文件上写数据
}

```

```

//let mut buffer = [0u8; 512];
let mut buffer = [0u8; 233];
let len = filea.read_at(0, &mut buffer); //从filea文件上取数据
assert_eq!(greet_str, core::str::from_utf8(&buffer[..len]).unwrap(),); //断言是否相同

//随机数据的测试的闭包
let mut random_str_test = |len: usize| {
    filea.clear();
    assert_eq!(filea.read_at(0, &mut buffer), 0,);
    let mut str = String::new();
    use rand;
    for _ in 0..len {
        str.push(char::from('0' as u8 + rand::random::<u8>() % 10));
    }
    filea.write_at(0, str.as_bytes());
    let mut read_buffer = [0u8; 127];
    let mut offset = 0usize;
    let mut read_str = String::new();
    loop {
        let len = filea.read_at(offset, &mut read_buffer);
        if len == 0 {
            break;
        }
        offset += len;
        read_str.push_str(core::str::from_utf8(&read_buffer[..len]).unwrap());
    }
    assert_eq!(str, read_str);
};

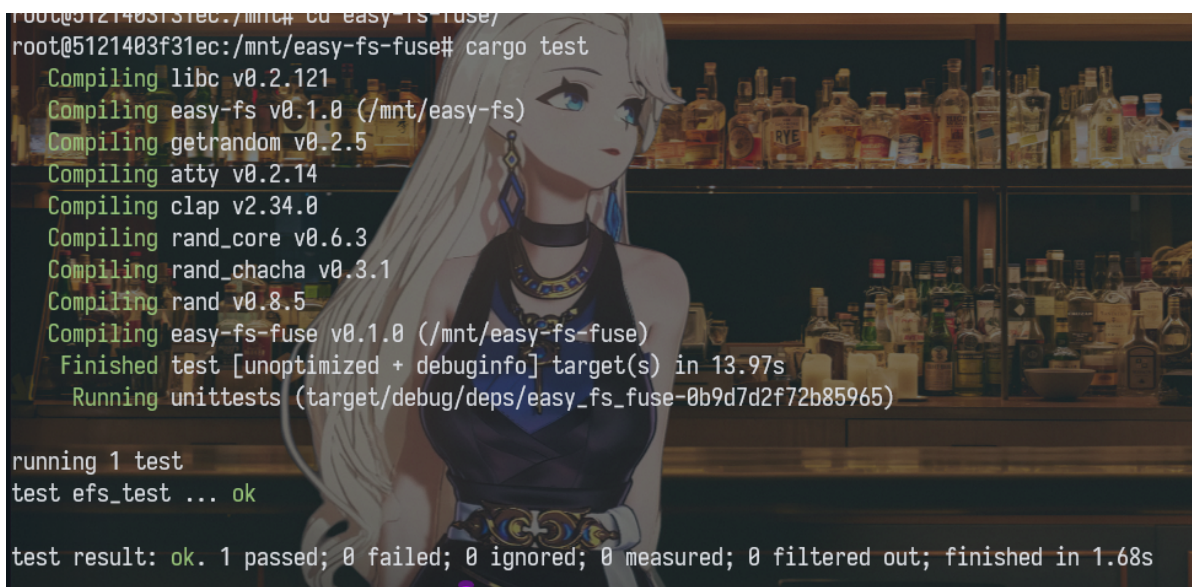
//调用随机测试的闭包
random_str_test(4 * BLOCK_SZ);
random_str_test(8 * BLOCK_SZ + BLOCK_SZ / 2);
random_str_test(100 * BLOCK_SZ);
random_str_test(70 * BLOCK_SZ + BLOCK_SZ / 7);
random_str_test((12 + 128) * BLOCK_SZ);
random_str_test(400 * BLOCK_SZ);
random_str_test(1000 * BLOCK_SZ);
random_str_test(2000 * BLOCK_SZ);
ok()
}

```

而在一个实际的裸机内核的中，写在文件系统的是可执行程序更能体现处文件系统的作用 虽然都是一个字节一个字节的写入

实现这个步骤的方法，就是首先创建一个文件系统的 img，然后把可执行文件同样读入内存，接着在文件系统中创建同名文件，然后将可执行文件写入到这个文件，整个过程相当于在Linux的两个文件系统移动文件。

实验结果及分析



```
root@5121403f31ec:/mnt# cd easy-fs-fuse/
root@5121403f31ec:/mnt/easy-fs-fuse# cargo test
  Compiling libc v0.2.121
  Compiling easy-fs v0.1.0 (/mnt/easy-fs)
  Compiling getrandom v0.2.5
  Compiling atty v0.2.14
  Compiling clap v2.34.0
  Compiling rand_core v0.6.3
  Compiling rand_chacha v0.3.1
  Compiling rand v0.8.5
  Compiling easy-fs-fuse v0.1.0 (/mnt/easy-fs-fuse)
  Finished test [unoptimized + debuginfo] target(s) in 13.97s
  Running unittests (target/debug/deps/easy_fs_fuse-0b9d7d2f72b85965)

running 1 test
test efs_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 1.68s
```

文件读写的测试成功通过。