lab6

编程题

1.扩展easy-fs文件系统功能，扩大单个文件的大小，支持三级间接inode。

在修改之前，先看看原始inode的结构：

```
/// The max number of direct inodes
const INODE_DIRECT_COUNT: usize = 28;

#[repr(C)]
pub struct DiskInode {
    pub size: u32,
    pub direct: [u32; INODE_DIRECT_COUNT],
    pub indirect1: u32,
    pub indirect2: u32,
    type_: DiskInodeType,
}

#[derive(PartialEq)]
pub enum DiskInodeType {
    File,
    Directory,
}
```

一个 `DiskInode` 在磁盘上占据128字节的空间。我们考虑加入 `indirect3` 字段并缩减 `INODE_DIRECT_COUNT` 为27以保持 `DiskInode` 的大小不变。此时直接索引可索引13.5KiB的内容，一级间接索引和二级间接索引仍然能索引64KiB和8MiB的内容，而三级间接索引能索引128 * 8MiB = 1GiB的内容。当文件大小大于13.5KiB + 64KiB + 8MiB时，需要用到三级间接索引。

下面的改动都集中在 `easy-fs/src/layout.rs` 中。首先修改 `DiskInode` 和相关的常量定义。

```
pub struct DiskInode {
    pub size: u32,
    pub direct: [u32; INODE_DIRECT_COUNT],
    pub indirect1: u32,
    pub indirect2: u32,
    pub indirect3: u32, //增加
    type_: DiskInodeType,
}
```

在计算给定文件大小对应的块总数时，需要新增对三级间接索引的处理。三级间接索引的存在使得二级间接索引所需的块数不再计入所有的剩余数据块。

取 `level2_extra`（需要额外的间接级别2块数）和 `INODE_INDIRECT1_COUNT`（每个间接级别1块可以容纳的数据块数量）中较小的值，将较小的值加到 `total` 变量上，以累计所需的总块数。

```
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
        let level2_extra =
            (data_blocks - INDIRECT1_BOUND + INODE_INDIRECT1_COUNT - 1) /
INODE_INDIRECT1_COUNT;
        total += level2_extra.min(INODE_INDIRECT1_COUNT); //增加
    }
    // indirect3
    if data_blocks > INDIRECT2_BOUND {
        let remaining = data_blocks - INDIRECT2_BOUND;
        let level2_extra = (remaining + INODE_INDIRECT2_COUNT - 1) /
INODE_INDIRECT2_COUNT;
        let level3_extra = (remaining + INODE_INDIRECT1_COUNT - 1) /
INODE_INDIRECT1_COUNT;
        total += 1 + level2_extra + level3_extra;
    }
    total as u32
}
```

`DiskInode` 的 `get_block_id` 方法中遇到三级间接索引要额外读取三次块缓存。

```
pub fn get_block_id(&self, inner_id: u32, block_device: &Arc<dyn BlockDevice>) -
> u32 {
    let inner_id = inner_id as usize;  // 将 inner_id 转换为 usize 类型
    if inner_id < INODE_DIRECT_COUNT {
        // 如果 inner_id 小于 INODE_DIRECT_COUNT
        // ...
    } else if inner_id < INDIRECT1_BOUND {
        // 如果 inner_id 小于 INDIRECT1_BOUND
        // ...
    } else if inner_id < INDIRECT2_BOUND {
        // 如果 inner_id 小于 INDIRECT2_BOUND
        // ...
    } else { // 对三级间接索引的处理
        let last = inner_id - INDIRECT2_BOUND;  // 计算 last 的值，用于访问三级间接索
引
        let indirect1 = get_block_cache(self.indirect3 as usize,
Arc::clone(block_device))
            .lock()
            .read(0, |indirect3: &IndirectBlock| {
                indirect3[last / INODE_INDIRECT2_COUNT]  // 访问间接索引数组中的元素
            });
        let indirect2 = get_block_cache(indirect1 as usize,
Arc::clone(block_device))
            .lock()
            .read(0, |indirect2: &IndirectBlock| {
```

```
                indirect2[(last % INODE_INDIRECT2_COUNT) /
INODE_INDIRECT1_COUNT]  // 访问间接索引数组中的元素
            });
        get_block_cache(indirect2 as usize, Arc::clone(block_device))
            .lock()
            .read(0, |indirect1: &IndirectBlock| {
                indirect1[(last % INODE_INDIRECT2_COUNT) %
INODE_INDIRECT1_COUNT]  // 访问间接索引数组中的元素
            })
    }
}
```

方法 `increase_size` 的实现本身比较繁琐，如果按照原有的一级和二级间接索引的方式实现对三级间接索引的处理，代码会比较丑陋。实际上多重间接索引是树结构，变量 `current_blocks` 和 `total_blocks` 对应着当前树的叶子数量和目标叶子数量，我们可以用递归函数来实现树的生长。先实现以下的辅助方法：

```
/// Helper to build tree recursively
/// 从 `src_leaf` 扩展到 `dst_leaf` 的叶子节点数量的辅助函数
fn build_tree(
    &self,
    blocks: &mut alloc::vec::IntoIter<u32>,
    block_id: u32,
    mut cur_leaf: usize,
    src_leaf: usize,
    dst_leaf: usize,
    cur_depth: usize,
    dst_depth: usize,
    block_device: &Arc<dyn BlockDevice>,
) -> usize {
    if cur_depth == dst_depth {
        return cur_leaf + 1;  // 如果当前深度等于目标深度，则返回当前叶子节点数加1
    }
    get_block_cache(block_id as usize, Arc::clone(block_device))
        .lock()
        .modify(0, |indirect_block: &mut IndirectBlock| {
            let mut i = 0;
            while i < INODE_INDIRECT1_COUNT && cur_leaf < dst_leaf {
                if cur_leaf >= src_leaf {
                    indirect_block[i] = blocks.next().unwrap();  // 将 blocks 迭代
器中的下一个块ID赋值给间接块数组
                }
                cur_leaf = self.build_tree(
                    blocks,
                    indirect_block[i],  // 递归调用 build_tree 函数构建子树
                    cur_leaf,
                    src_leaf,
                    dst_leaf,
                    cur_depth + 1,
                    dst_depth,
                    block_device,
                );
                i += 1;
            }
```

```
        });
    cur_leaf  // 返回当前叶子节点数
}
```

然后修改方法 `increase_size`。不要忘记在填充二级间接索引时维护 `current_blocks` 的变化，并限制目标索引 `(a1, b1)` 的范围。

```rust
/// Increase the size of current disk inode
pub fn increase_size(
    &mut self,
    new_size: u32,
    new_blocks: Vec<u32>,
    block_device: &Arc<dyn BlockDevice>,
) {
    // ...
    // alloc indirect2
    // ...
    // fill indirect2 from (a0, b0) -> (a1, b1)
    // 不要忘记限制 (a1, b1) 的范围
    // ...
    // alloc indirect3
    if total_blocks > INODE_INDIRECT2_COUNT as u32 {
        if current_blocks == INODE_INDIRECT2_COUNT as u32 {
            self.indirect3 = new_blocks.next().unwrap();
        }
        current_blocks -= INODE_INDIRECT2_COUNT as u32;
        total_blocks -= INODE_INDIRECT2_COUNT as u32;
    } else {
        return;
    }
    // fill indirect3
    self.build_tree(
        &mut new_blocks,
        self.indirect3,
        0,
        current_blocks as usize,
        total_blocks as usize,
        0,
        3,
        block_device,
    );
```

对方法 `clear_size` 的修改与 `increase_size` 类似。先实现辅助方法 `collect_tree_blocks`：

```rust
/// Helper to recycle blocks recursively
/// 递归回收块的辅助函数
fn collect_tree_blocks(
    &self,
    collected: &mut Vec<u32>, // 存储回收的块 ID 的向量
    block_id: u32, // 当前块的 ID
    mut cur_leaf: usize, // 当前叶子节点的索引
    max_leaf: usize, // 最大叶子节点的索引
    cur_depth: usize, // 当前深度
    dst_depth: usize, // 目标深度
```

```
    block_device: &Arc<dyn BlockDevice>, // 块设备的引用
) -> usize {
    if cur_depth == dst_depth {
        return cur_leaf + 1; // 达到目标深度，返回更新后的叶子节点索引
    }
    get_block_cache(block_id as usize, Arc::clone(block_device)) // 获取块缓存
        .lock() // 获取块缓存的互斥锁
        .read(0, |indirect_block: &IndirectBlock| {
            // 在互斥锁的作用下读取间接块
            let mut i = 0;
            while i < INODE_INDIRECT1_COUNT && cur_leaf < max_leaf {
                collected.push(indirect_block[i]); // 将块 ID 添加到回收的块向量中
                cur_leaf = self.collect_tree_blocks(
                    collected,
                    indirect_block[i], // 递归回收子块
                    cur_leaf,
                    max_leaf,
                    cur_depth + 1,
                    dst_depth,
                    block_device,
                );
                i += 1;
            }
        });
    cur_leaf // 返回更新后的叶子节点索引
}
```

然后修改方法 `clear_size`。

```
/// Clear size to zero and return blocks that should be deallocated.
/// 清除大小为零并返回应该被释放的块。
/// 我们稍后会将块内容清零。

pub fn clear_size(&mut self, block_device: &Arc<dyn BlockDevice>) -> Vec<u32> {
    // ...
    // indirect2 block
    // ...

    // 将 indirect2 清零
    self.indirect2 = 0;

    // indirect3 block
    assert!(data_blocks <= INODE_INDIRECT3_COUNT);
    if data_blocks > INODE_INDIRECT2_COUNT {
        v.push(self.indirect3);
        data_blocks -= INODE_INDIRECT2_COUNT;
    } else {
        return v;
    }

    // indirect3
    self.collect_tree_blocks(&mut v, self.indirect3, 0, data_blocks, 0, 3,
block_device);

    // 将 indirect3 清零
```

```
        self.indirect3 = 0;

        // 返回需要释放的块
        v
    }
```

接下来你可以在 `easy-fs-fuse/src/main.rs` 中测试easy-fs文件系统的修改，比如读写大小超过10MiB的文件。

2.扩展easy-fs文件系统功能，支持二级目录结构。可扩展：支持N级目录结构。

实际上easy-fs现有的代码支持目录的存在，只不过整个文件系统只有根目录一个目录，我们考虑放宽现有代码的一些限制。

原本的 `easy-fs/src/vfs.rs` 中有一个用于在当前目录下创建常规文件的 `create` 方法，我们给它加个参数并包装一下：

```
easy-fs/src/vfs.rs
impl Inode {
    /// Create inode under current inode by name
    /// 根据名称在当前 inode 下创建 inode
    fn create_inode(&self, name: &str, inode_type: DiskInodeType) ->
Option<Arc<Inode>> {
        let mut fs = self.fs.lock();
        let op = |root_inode: &DiskInode| {
            // 断言它是一个目录
            assert!(root_inode.is_dir());
            // 文件是否已经创建?
            self.find_inode_id(name, root_inode)
        };
        if self.read_disk_inode(op).is_some() {
            return None;
        }
        // 创建一个新文件
        // 分配一个具有间接块的 inode
        let new_inode_id = fs.alloc_inode();
        // 初始化 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(inode_type); //调用了 initialize 方法来初始化新
的 inode。
            });
        self.modify_disk_inode(|root_inode| {
            // 在目录项中添加文件
            let file_count = (root_inode.size as usize) / DIRENT_SZ;
            let new_size = (file_count + 1) * DIRENT_SZ;
            // 增加文件大小
            self.increase_size(new_size as u32, root_inode, &mut fs);
            // 写入目录项
```

```
            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(),
        )))
        // 编译器会自动释放 efs 锁
    }

    /// Create regular file under current inode
    /// 在当前 inode 下创建普通文件
    pub fn create(&self, name: &str) -> Option<Arc<Inode>> {
        self.create_inode(name, DiskInodeType::File)
    }

    /// Create directory under current inode
    /// 在当前 inode 下创建目录
    pub fn create_dir(&self, name: &str) -> Option<Arc<Inode>> {
        self.create_inode(name, DiskInodeType::Directory)
    }
}
```

这样我们就可以在一个目录底下调用 `create_dir` 创建新目录了（笑）。本质上我们什么也没改，我们再改改其它方法装装样子：

```
easy-fs/src/vfs.rs
impl Inode {
    /// List inodes under current inode
    /// 列出当前 inode 下的所有 inode
    pub fn ls(&self) -> Vec<String> {
        let _fs = self.fs.lock();
        self.read_disk_inode(|disk_inode| {
            let mut v: Vec<String> = Vec::new();
            if disk_inode.is_file() { //如果 disk_inode 是一个文件（而不是目录）
                return v; //返回空的结果向量v
            }

            let file_count = (disk_inode.size as usize) / DIRENT_SZ;
            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
        })
    }

    /// Write data to current inode
    /// 向当前 inode 写入数据
    pub fn write_at(&self, offset: usize, buf: &[u8]) -> usize {
        let mut fs = self.fs.lock();
        let size = self.modify_disk_inode(|disk_inode| {
            assert!(disk_inode.is_file()); //验证 disk_inode 是否表示一个文件而不是一
个目录

            self.increase_size((offset + buf.len()) as u32, disk_inode, &mut
fs);
            disk_inode.write_at(offset, buf, &self.block_device)
        });
        block_cache_sync_all();
        size
    }

    /// Clear the data in current inode
    /// 清除当前 inode 中的数据
    pub fn clear(&self) {
        let mut fs = self.fs.lock();
        self.modify_disk_inode(|disk_inode| {
            assert!(disk_inode.is_file());

            let size = disk_inode.size;
            let data_blocks_dealloc = disk_inode.clear_size(&self.block_device);
            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);
            }
        });
        block_cache_sync_all();
    }
}
```

对一个普通文件的inode调用 `ls` 方法毫无意义，但为了保持接口不变，我们返回一个空 `Vec`。随意地清空或写入目录文件都会损坏目录结构，这里直接在 `write_at` 和 `clear` 方法中断言，你也可以改成其它的错误处理方式。

接下来是实际一点的修改（有，但不多）：我们让 `find` 方法支持简单的相对路径（不含“.”和“..”）。

```
easy-fs/src/vfs.rs
impl Inode {
    /// Find inode under current inode by **path**
    /// 根据路径在当前 inode 下查找 inode
    pub fn find(&self, path: &str) -> Option<Arc<Inode>> {
        let fs = self.fs.lock(); // 获取文件系统锁
        let mut block_id = self.block_id as u32; // 当前块的 ID
```

```
        let mut block_offset = self.block_offset; // 当前块的偏移量
        for name in path.split('/').filter(|s| !s.is_empty()) {
            // 遍历路径中的每个组件
            let inode_id = get_block_cache(block_id as usize,
self.block_device.clone()) // 通过块 ID 获取块缓存
                .lock() // 获取块缓存的互斥锁
                .read(block_offset, |disk_inode: &DiskInode| {
                    // 在互斥锁的作用下读取磁盘 inode
                    if disk_inode.is_file() {
                        return None; // 如果是文件，则返回 None
                    }
                    self.find_inode_id(name, disk_inode) // 在当前目录下查找指定名称
的 inode
                });
            if inode_id.is_none() {
                return None; // 如果找不到指定名称的 inode，则返回 None
            }
            (block_id, block_offset) = fs.get_disk_inode_pos(inode_id.unwrap());
// 获取找到的 inode 的块 ID 和偏移量
        }
        Some(Arc::new(Self::new(
            block_id,
            block_offset,
            self.fs.clone(),
            self.block_device.clone(),
        ))) // 返回找到的 inode
    }
}
```

最后在 `easy-fs-fuse/src/main.rs` 里试试我们添加的新特性：

```
easy-fs-fuse/src/main.rs
fn read_string(file: &Arc<Inode>) -> String {
    let mut read_buffer = [0u8; 512]; // 读取缓冲区
    let mut offset = 0usize; // 读取偏移量
    let mut read_str = String::new(); // 读取的字符串
    loop {
        let len = file.read_at(offset, &mut read_buffer); // 在指定偏移量处读取文件内
容到缓冲区
        if len == 0 {
            break; // 如果读取长度为0，则退出循环
        }
        offset += len; // 更新读取偏移量
        read_str.push_str(core::str::from_utf8(&read_buffer[..len]).unwrap());
// 将缓冲区中的内容转换为字符串并追加到读取的字符串中
    }
    read_str // 返回读取的字符串
}

fn tree(inode: &Arc<Inode>, name: &str, depth: usize) {
    for _ in 0..depth {
        print!("  "); // 打印缩进
    }
    println!("{}", name); // 打印当前目录项的名称
    for name in inode.ls() { // 遍历当前目录项下的所有子项名称
```

```
        let child = inode.find(&name).unwrap(); // 根据子项名称查找对应的子项 inode
        tree(&child, &name, depth + 1); // 递归遍历子项的子项
    }
}

#[test]
fn efs_dir_test() -> std::io::Result<()> {
    let block_file = Arc::new(BlockFile(Mutex::new({
        let f = OpenOptions::new()
            .read(true)
            .write(true)
            .create(true)
            .open("target/fs.img")?;
        f.set_len(8192 * 512).unwrap();
        f
    })));
    EasyFileSystem::create(block_file.clone(), 4096, 1); // 创建文件系统
    let efs = EasyFileSystem::open(block_file.clone()); // 打开文件系统
    let root = Arc::new(EasyFileSystem::root_inode(&efs)); // 获取根目录的 inode
    root.create("f1"); // 在根目录下创建文件 f1
    root.create("f2"); // 在根目录下创建文件 f2

    let d1 = root.create_dir("d1").unwrap(); // 在根目录下创建目录 d1

    let f3 = d1.create("f3").unwrap(); // 在目录 d1 下创建文件 f3
    let d2 = d1.create_dir("d2").unwrap(); // 在目录 d1 下创建目录 d2

    let f4 = d2.create("f4").unwrap(); // 在目录 d2 下创建文件 f4
    tree(&root, "/", 0); // 以树状结构输出文件系统目录结构

    let f3_content = "3333333";
    let f4_content = "44444444444444444444";
    f3.write_at(0, f3_content.as_bytes()); // 将内容写入文件 f3
    f4.write_at(0, f4_content.as_bytes()); // 将内容写入文件 f4

    assert_eq!(read_string(&d1.find("f3").unwrap()), f3_content); // 断言读取文件
f3 的内容与预期相等
    assert_eq!(read_string(&root.find("/d1/f3").unwrap()), f3_content); // 断言读
取文件 /d1/f3 的内容与预期相等
    assert_eq!(read_string(&d2.find("f4").unwrap()), f4_content); // 断言读取文件
f4 的内容与预期相等
    assert_eq!(read_string(&d1.find("d2/f4").unwrap()), f4_content); // 断言读取文
件 d2/f4 的内容与预期相等
    assert_eq!(read_string(&root.find("/d1/d2/f4").unwrap()), f4_content); // 断
言读取文件 /d1/d2/f4 的内容与预期相等
    assert!(f3.find("whatever").is_none()); // 断言查找文件 f3 下的子项 "whatever"
返回 None
    Ok(())
}
```

问答题

1.文件系统的功能是什么?

将数据以文件的形式持久化保存在存储设备上。

2.目前的文件系统只有单级目录，假设想要支持多级文件目录，请描述你设想的实现方式，描述合理即可。

允许在目录项中存在目录（原本只能存在普通文件）即可。

3.软链接和硬链接是干什么的？有什么区别？当删除一个软链接或硬链接时分别会发生什么?

软硬链接的作用都是给一个文件以”别名”，使得不同的多个路径可以指向同一个文件。当删除软链接时候，对文件没有任何影响，当删除硬链接时，文件的引用计数会被减一，若引用计数为0，则该文件所占据的磁盘空间将会被回收。

4.在有了多级目录之后，我们就也可以为一个目录增加硬链接了。在这种情况下，文件树中是否可能出现环路(软硬链接都可以，鼓励多尝试)？你认为应该如何解决？请在你喜欢的系统上实现一个环路，描述你的实现方式以及系统提示、实际测试结果。

是可以出现环路的，一种可能的解决方式是在访问文件的时候检查自己遍历的路径中是否有重复的inode，并在发现环路时返回错误。

5.目录是一类特殊的文件，存放的是什么内容？用户可以自己修改目录内容吗?

存放的是目录中的文件列表以及他们对应的inode，通常而言用户不能自己修改目录的内容，但是可以通过操作目录（如mv里面的文件）的方式间接修改。

6.在实际操作系统中，如Linux，为什么会存在大量的文件系统类型?

因为不同的文件系统有着不同的特性，比如对于特定种类的存储设备的优化，或是快照和多设备管理等高级特性，适用于不同的使用场景。

7.可以把文件控制块放到目录项中吗？这样做有什么优缺点?

可以，是对于小目录可以减少一次磁盘访问，提升性能，但是对大目录而言会使得在目录中查找文件的性能降低。

8.为什么要同时维护进程的打开文件表和操作系统的打开文件表？这两个打开文件表有什么区别和联系?

多个进程可能会同时打开同一个文件，操作系统级的打开文件表可以加快后续的打开操作，但同时由于每个进程打开文件时使用的访问模式或是偏移量不同，所以还需要进程的打开文件表另外记录。

9.文件分配的三种方式是如何组织文件数据块的？各有什么特征（存储、文件读写、可靠性）？

连续分配：实现简单、存取速度快，但是难以动态增加文件大小，长期使用后会产生大量无法使用（过小而无法放入大文件）碎片空间。

链接分配：可以处理文件大小的动态增长，也不会出现碎片，但是只能按顺序访问文件中的块，同时一旦有一个块损坏，后面的其他块也无法读取，可靠性差。

索引分配：可以随机访问文件中的偏移量，但是对于大文件需要实现多级索引，实现较为复杂。

10.如果一个程序打开了一个文件，写入了一些数据，但是没有及时关闭，可能会有什么后果？如果打开文件后，又进一步发出了读文件的系统调用，操作系统中各个组件是如何相互协作完成整个读文件的系统调用的？

> (若也没有flush的话）假如此时操作系统崩溃，尚处于内存缓冲区中未写入磁盘的数据将会丢失，同时也会占用文件描述符，造成资源的浪费。首先是系统调用处理的部分，将这一请求转发给文件系统子系统，文件系统子系统再将其转发给块设备子系统，最后再由块设备子系统转发给实际的磁盘驱动程序读取数据，最终返回给程序。

11.文件系统是一个操作系统必要的组件吗？是否可以将文件系统放到用户态？这样做有什么好处？操作系统需要提供哪些基本支持？

不是，如在本章之前的rCore就没有文件系统。可以，如在Linux下就有FUSE这样的框架可以实现这一点。这样可以使得文件系统的实现更为灵活，开发与调试更为简便。操作系统需要提供一个注册用户态文件系统实现的机制，以及将收到的文件系统相关系统调用转发给注册的用户态进程的支持。