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。

答：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 {//修改DiskInode 和相关的常量定义
    pub size: u32,//文件当前的字节长度
    pub direct: [u32; INODE_DIRECT_COUNT],//单级间接块指针，指向一个 “间接块” 的块号。
    pub indirect1: u32,//双级间接块指针，指向一个包含一级间接块号的块。
    pub indirect2: u32,//三级间接块指针，指向一个包含二级间接块号的块。
    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。

```
pub fn total_blocks(size: u32) -> u32 {//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。
    let data_blocks = Self::_data_blocks(size) as usize;
    let mut total = data_blocks as usize;//计算数据块数
    // indirect1
    if data_blocks > INODE_DIRECT_COUNT {//一级间接块 (indirect1)
        total += 1;
    }
    // indirect2
    if data_blocks > INDIRECT1_BOUND {//二级间接块 (indirect2)
        total += 1;
        // sub indirect1
        total +=
            (data_blocks - INDIRECT1_BOUND + INODE_INDIRECT1_COUNT - 1) / INODE_INDIRECT1_COUNT;
    }//三级间接块 (indirect3)
    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。

```rust
pub fn get_block_id(&self, inner_id: u32, block_device: &Arc<dyn BlockDevice>) -> 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。
    let inner_id = inner_id as usize;
    if inner_id < INODE_DIRECT_COUNT {//直接块分支
        self.direct[inner_id]
    } else if inner_id < INDIRECT1_BOUND { //一级间接块分支
        get_block_cache(self.indirect1 as usize, Arc::clone(block_device))
            .lock()
            .read(0, |indirect_block: &IndirectBlock| {
                indirect_block[inner_id - INODE_DIRECT_COUNT]
            })
    } else {//二级间接块分支
        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]
            });
        get_block_cache(indirect1 as usize, Arc::clone(block_device))
            .lock()
            .read(0, |indirect1: &IndirectBlock| {
                indirect1[last % INODE_INDIRECT1_COUNT]
            })
    }
}
```

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
pub fn increase_size(//修改方法 increase_size
    &mut self,
    new_size: u32,
    new_blocks: Vec<u32>,
    block_device: &Arc<dyn BlockDevice>,
) {
    let mut current_blocks = self.data_blocks();
    self.size = new_size;
    let mut total_blocks = self.data_blocks();
    let mut new_blocks = new_blocks.into_iter();
    // fill direct
    while current_blocks < total_blocks.min(INODE_DIRECT_COUNT as u32) {
        self.direct[current_blocks as usize] = new_blocks.next().unwrap();
        current_blocks += 1;
    }
    // alloc indirect1
    if total_blocks > INODE_DIRECT_COUNT as u32 {
        if current_blocks == INODE_DIRECT_COUNT as u32 {
            self.indirect1 = new_blocks.next().unwrap();
        }
        current_blocks -= INODE_DIRECT_COUNT as u32;
        total_blocks -= INODE_DIRECT_COUNT as u32;
    } else {
        return;
    }
    // fill indirect1
    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;
            }
        });
    // alloc indirect2
    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;
    }
```

对方法 clear_size 的修改与 increase_size 类似：

```rust
pub fn clear_size(&mut self, block_device: &Arc<dyn BlockDevice>) -> Vec<u32> {
    let mut v: Vec<u32> = Vec::new();
    let mut data_blocks = self.data_blocks() as usize;
    self.size = 0;
    let mut current_blocks = 0usize;
    // direct
    while current_blocks < data_blocks.min(INODE_DIRECT_COUNT) {
        v.push(self.direct[current_blocks]);
        self.direct[current_blocks] = 0;
        current_blocks += 1;
    }
    // indirect1 block
    if data_blocks > INODE_DIRECT_COUNT {
        v.push(self.indirect1);
        data_blocks -= INODE_DIRECT_COUNT;
        current_blocks = 0;
    } else {
        return v;
    }
    // indirect1
    get_block_cache(self.indirect1 as usize, Arc::clone(block_device))
        .lock()
        .modify(0, |indirect1: &mut IndirectBlock| {
            while current_blocks < data_blocks.min(INODE_INDIRECT1_COUNT) {
                v.push(indirect1[current_blocks]);
                //indirect1[current_blocks] = 0;
                current_blocks += 1;
            }
        });
    self.indirect1 = 0;
    // indirect2 block
    if data_blocks > INODE_INDIRECT1_COUNT {
        v.push(self.indirect2);
        data_blocks -= INODE_INDIRECT1_COUNT;
    } else {
        return v;
    }
    // indirect2
    assert!(data_blocks <= INODE_INDIRECT2_COUNT);
    let a1 = data_blocks / INODE_INDIRECT1_COUNT;
    let b1 = data_blocks % INODE_INDIRECT1_COUNT;
    get_block_cache(self.indirect2 as usize, Arc::clone(block_device))
        .lock()
        .modify(0, |indirect2: &mut IndirectBlock| {
            // full indirect1 blocks
            for entry in indirect2.iter_mut().take(a1) {
                v.push(*entry);
                get_block_cache(*entry as usize, Arc::clone(block_device))
                    .lock()
                    .modify(0, |indirect1: &mut IndirectBlock| {
                        for entry in indirect1.iter() {
```

接下来我们可以在 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。

```
    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(())
}
```

随机生成的测试文件

使用指令“cargo test”来进行测试，得到结果：

```
   Compiling easy-fs v0.1.0 (/home/oslab/lab6/rCore-sp23/easy-fs)
   Compiling getrandom v0.2.16
   Compiling atty v0.2.14
   Compiling clap v2.34.0
   Compiling rand_core v0.6.4
   Compiling ppv-lite86 v0.2.21
   Compiling rand_chacha v0.3.1
   Compiling rand v0.8.5
   Compiling easy-fs-fuse v0.1.0 (/home/oslab/lab6/rCore-sp23/easy-fs-fuse)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 3.86s
     Running unittests src/main.rs (target/debug/deps/easy_fs_fuse-7dc8bf078ed1b
16a)

running 1 test
test efs_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; fini
shed in 0.46s
```

测试通过。

2. * 扩展内核功能，支持 stat 系统调用，能显示文件的 inode 元数据信息。

答：想要实现该功能，我们可以使用 stat 的变体，fstat 系统调用。

我们先在 mod.rs 中添加声明：

```rust
pub const SYSCALL_READ: usize = 63;
pub const SYSCALL_WRITE: usize = 64;
pub const SYSCALL_UNLINKAT: usize = 35;
pub const SYSCALL_LINKAT: usize = 37;
pub const SYSCALL_FSTAT: usize = 80;
pub const SYSCALL_EXIT: usize = 93;
pub const SYSCALL_YIELD: usize = 124;
pub const SYSCALL_FORK: usize = 220;
pub const SYSCALL_EXEC: usize = 221;
pub const SYSCALL_WAITPID: usize = 260;
pub const SYSCALL_GETPID: usize = 172;
pub const SYSCALL_GET_TIME: usize = 169;
pub const SYSCALL_TASK_INFO: usize = 410;
pub const SYSCALL_MUNMAP: usize = 215;
pub const SYSCALL_MMAP: usize = 222;
pub const SYSCALL_SBRK: usize = 214;
pub const SYSCALL_SPAWN: usize = 400;
pub const SYSCALL_SET_PRIORITY: usize = 140;
pub const SYSCALL_OPEN: usize = 56;
pub const SYSCALL_CLOSE: usize = 57;
```

然后在系统调用分发器中添加 fstat 的声明，用于处理用户态程序发起的 fstat 系统调用。

```rust
    SYSCALL_FSTAT => sys_fstat(args[0], args[1] as *mut Stat),
```

接着在 fs.rs 中添加 fstat 的实现。我们现在开头导入 fs 模块中与文件操作、元数据访问相关的一系列重要元素，使得后续代码更简洁、更容易书写。

```rust
use crate::fs::{open_file, OpenFlags, Stat, ROOT_INODE, OSInode, StatMode};
```

接着后面进行具体实现。实现代码如下：

```rust
/// YOUR JOB: Implement fstat.
pub fn sys_fstat(fd: usize, st: *mut Stat) -> isize {
    let task = current_task().unwrap();
    let inner = task.inner_exclusive_access();//获取当前任务和文件表

    // check legality
    if fd >= inner.fd_table.len() {
        return -1;
    }
    if inner.fd_table[fd].is_none() {
        return -1;
    }//校验文件描述符合法性

    let ino: u64;
    let nlink: u32;
    if let Some(file_node) = &inner.fd_table[fd] {
        let any: &dyn Any = file_node.as_any();
        let os_node = any.downcast_ref::<OSInode>().unwrap();
        ino = os_node.get_inode_id();
        let (block_id, block_offset) = os_node.get_inode_pos();
        nlink = ROOT_INODE.get_link_num(block_id, block_offset);
    } else {
        return -1;
    }//从文件描述符获取 inode 信息

    let stat = &Stat {
        dev: 0,
        ino: ino,
        mode: StatMode::FILE,
        nlink: nlink,
        pad: [0;7],
    };// 构造返回给用户的 Stat 结构
```

之后，我们在 syscall.rs 中完成整个调用的实现：

```rust
pub fn sys_fstat(fd: usize, st: &Stat) -> isize {
    syscall(SYSCALL_FSTAT, [fd, st as *const _ as usize, 0])
}
```

然后，我们要完成 stat 结构体，该部分的实现在 lib.rs 中：

```rust
pub struct Stat {
    /// ID of device containing file
    pub dev: u64,
    /// inode number
    pub ino: u64,
    /// file type and mode
    pub mode: StatMode,
    /// number of hard links
    pub nlink: u32,
    /// unused pad
    pad: [u64; 7],
}

impl Stat {
    pub fn new() -> Self {
        Stat {
            dev: 0,
            ino: 0,
            mode: StatMode::NULL,
            nlink: 0,
            pad: [0; 7],
        }
    }
}

impl Default for Stat {
    fn default() -> Self {
        Self::new()
    }
}

bitflags! {
    pub struct StatMode: u32 {
        const NULL  = 0;
        /// directory
        const DIR   = 0o040000;
        /// ordinary regular file
        const FILE  = 0o100000;
    }
}
```

接着我们设计 stat 显示文件的 inode 元数据信息的测试程序:

```rust
#![no_std]
#![no_main]

extern crate user_lib;
use user_lib::{open, write, close, fstat, println, OpenFlags, Stat, StatMode};

#[no_mangle]
fn main() -> i32 {
    // 1. 创建并写入测试文件
    let fname = "testfile\0";
    // O_CREAT | O_WRONLY
    let fd = open(fname, OpenFlags::WRONLY | OpenFlags::CREATE);
    if fd < 0 {
        println!("open failed");
        return -1;
    }
    // 写入一点数据，不关心内容
    let _ = write(fd as usize, b"x");
    close(fd as usize);

    // 2. 重新以只读方式打开，用于 fstat
    let fd = open(fname, OpenFlags::RDONLY);
    if fd < 0 {
        println!("open for fstat failed");
        return -1;
    }

    // 3. 调用 fstat
    let mut st = Stat::default();
    let ret = fstat(fd as usize, &st);
    if ret < 0 {
        println!("fstat failed");
        return -1;
    }

    // 4. 打印并验证原有字段
    println!("fstat returned:");
    println!(" dev   = {}", st.dev);
    println!(" ino   = {}", st.ino);
    println!(" mode  = {:?}", st.mode);
    println!(" nlink = {}", st.nlink);

    // 测试条件：inode 非 0、类型是普通文件、链接数为 1
    if st.ino != 0
        && st.mode == StatMode::FILE
        && st.nlink == 1
    {
```

修改完之后我们重新编译

```
oslab@oslab-virtual-machine:~/lab6/rCore-sp23/user$ make clean && make build
     Removed 242 files, 127.8MiB total
     Removed 0 files
target/riscv64gc-unknown-none-elf/release/ch6b_filetest_simple target/riscv64gc-
unknown-none-elf/release/ch6b_initproc target/riscv64gc-unknown-none-elf/release
/ch6_file0 target/riscv64gc-unknown-none-elf/release/ch6_file1 target/riscv64gc-
unknown-none-elf/release/ch6_file2 target/riscv64gc-unknown-none-elf/release/ch6
_file3 target/riscv64gc-unknown-none-elf/release/ch6_stat target/riscv64gc-unkno
wn-none-elf/release/symbol
   Compiling memchr v2.7.4
   Compiling semver-parser v0.7.0
   Compiling regex-syntax v0.8.5
   Compiling lazy_static v1.5.0
```

编译完成之后运行“make run“

```
[initproc] launching ch6_stat
fstat returned:
 dev    = 0
 ino    = 7
 mode   = FILE
 nlink = 1
Test fstat OK!
```

可以看到结果中成果显示出了 stat 打印 node 的元数据信息，测试成果。

问答题:

1 * 文件系统的功能是什么?

答：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。

答 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？

答：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。

答：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？

答：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？

答：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？

答：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？

答：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）？

答：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？

答：(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?

答：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。

实验练习：

硬链接

硬链接要求两个不同的目录项指向同一个文件，在我们的文件系统中也就是两个不同名称目录项指向同一个磁盘块。

本节要求实现三个系统调用 sys_linkat、sys_unlinkat、sys_stat 。根据实验文档中的要求，我们对上述系统调用进行声明：

```rust
pub const SYSCALL_READ: usize = 63;
pub const SYSCALL_WRITE: usize = 64;
pub const SYSCALL_UNLINKAT: usize = 35;
pub const SYSCALL_LINKAT: usize = 37;
pub const SYSCALL_FSTAT: usize = 80;
pub const SYSCALL_EXIT: usize = 93;
pub const SYSCALL_YIELD: usize = 124;
pub const SYSCALL_FORK: usize = 220;
pub const SYSCALL_EXEC: usize = 221;
pub const SYSCALL_WAITPID: usize = 260;
pub const SYSCALL_GETPID: usize = 172;
pub const SYSCALL_GET_TIME: usize = 169;
pub const SYSCALL_TASK_INFO: usize = 410;
pub const SYSCALL_MUNMAP: usize = 215;
pub const SYSCALL_MMAP: usize = 222;
pub const SYSCALL_SBRK: usize = 214;
pub const SYSCALL_SPAWN: usize = 400;
pub const SYSCALL_SET_PRIORITY: usize = 140;
pub const SYSCALL_OPEN: usize = 56;
pub const SYSCALL_CLOSE: usize = 57;
```

之后我们开始其具体的实现。

根据系统调用号（SYSCALL_LINKAT、SYSCALL_UNLINKAT、SYSCALL_FSTAT），通过 match 分支选择相应的处理函数。

```rust
        SYSCALL_LINKAT => sys_linkat(args[1] as *const u8, args[3] as *const u8),
        SYSCALL_UNLINKAT => sys_unlinkat(args[1] as *const u8),
        SYSCALL_FSTAT => sys_fstat(args[0], args[1] as *mut Stat),
```

其中，对于 linkat：

args[1] as *const u8：转换为指向旧路径字符串（oldpath）的只读指针。args[3] as *const u8：转换为指向新路径字符串（newpath）的只读指针。

这里需要注意的事情是，linkat() 除了路径，还会从 args 中获取目录文件描述符和标志位，但这些参数在这里被我省略，在 sys_linkat 内部一并处理。

对于 unlinkat：

对应用户态的 unlinkat() 系统调用，用于删除（解除链接）指定路径名。args[1] as *const u8：转换为指向要删除的路径字符串

的只读指针。

Fatst：

对应用户态的 fstat() 系统调用，用于获取打开文件描述符的文件状态信息。args[0]：文件描述符（fd），一个整数。args[1] as *mut Stat：转换为指向 Stat 结构体的可变指针，用于在用户空间写回文件元数据（如文件大小、权限、时间戳等）。

声明之后我们来对上面系统调用进行具体实现：

LINKAT：

```
/// YOUR JOB: Implement linkat.
pub fn sys_linkat(old_name: *const u8, new_name: *const u8) -> isize {
    let token = current_user_token();
    let old = translated_str(token, old_name);
    let new = translated_str(token, new_name);
    println!("link {} to {}", new , old);
    if old.as_str() != new.as_str() {
        if let Some(_) = ROOT_INODE.link(old.as_str(), new.as_str()) {
            return 0;
        }
    }
    -1
}
```

大体逻辑为：获取当前进程/线程的用户地址空间标识符 token；将用户态的指针转换并拷贝为 Rust 字符串；打印调试信息；如果旧路径与新路径不同，则尝试在根目录（ROOT_INODE）下创建硬链接；成功返回 0，否则返回 -1。

UNLINKAT：

```
pub fn sys_unlinkat(name: *const u8) -> isize {
    let token = current_user_token();
    let name = translated_str(token, name);
    if let Some(inode) = ROOT_INODE.find(name.as_str()) {
        if ROOT_INODE.get_link_num(inode.block_id, inode.block_offset) == 1 {
            // clear data if only one link exists
            inode.clear();
        }
        return ROOT_INODE.unlink(name.as_str());
    }
    -1
}
```

这部分，首先获取用户地址空间的翻译令牌，将用户传入的 C 字符串指针转换成内核可操作的 Rust 字符串，然后在根目录的

inode 上查找对应文件，检查硬链接计数并在必要时清理数据，最后调用底层文件系统的 unlink 接口完成删除操作，成功返回文件系统接口的返回值（一般为 0），失败返回 -1

Fstat：

```rust
/// YOUR JOB: Implement fstat.
pub fn sys_fstat(fd: usize, st: *mut Stat) -> isize {
    let task = current_task().unwrap();
    let inner = task.inner_exclusive_access();//获取当前任务和文件表

    // check legality
    if fd >= inner.fd_table.len() {
        return -1;
    }
    if inner.fd_table[fd].is_none() {
        return -1;
    }//校验文件描述符合法性

    let ino: u64;
    let nlink: u32;
    if let Some(file_node) = &inner.fd_table[fd] {
        let any: &dyn Any = file_node.as_any();
        let os_node = any.downcast_ref::<OSInode>().unwrap();
        ino = os_node.get_inode_id();
        let (block_id, block_offset) = os_node.get_inode_pos();
        nlink = ROOT_INODE.get_link_num(block_id, block_offset);
    } else {
        return -1;
    }//从文件描述符获取 inode 信息

    let stat = &Stat {
        dev: 0,
        ino: ino,
        mode: StatMode::FILE,
        nlink: nlink,
        pad: [0;7],
    };// 构造返回给用户的 Stat 结构

    // copy data from kernel space to user space
    let token = inner.get_user_token();
    let st = translated_byte_buffer(token, st as *const u8, core::mem::size_of::<Stat>());
    let stat_ptr = stat as *const _ as *const u8;
    for (idx, byte) in st.into_iter().enumerate() {
        unsafe {
            byte.copy_from_slice(core::slice::from_raw_parts(stat_ptr.wrapping_byte_add(idx), byte.len()));
        }
    }
    0
}
```

首先，我们获取当前任务的上下文与文件描述符表；之后验证给定的文件描述符是否合法；再从文件描述符表中取出对应的 inode 信息；接着，构造要返回给用户的 Stat 结构；然后通过安全拷贝机制，将内核态的 Stat 数据写回到用户态提供的缓冲区；最后如果调用成功则返回 0，失败返回 -1。

之后我本来打算做测试程序的，但是我做完 fstat 的测试程序之后才发现，文件里面有作者已经写好的测试程序，只需要直接 test

就行了。

```rust
#![no_std]
#![no_main]

extern crate user_lib;
use user_lib::{open, write, close, fstat, println, OpenFlags, Stat, StatMode};

#[no_mangle]
fn main() -> i32 {
    // 1. 创建并写入测试文件
    let fname = "testfile\0";
    // O_CREAT | O_WRONLY
    let fd = open(fname, OpenFlags::WRONLY | OpenFlags::CREATE);
    if fd < 0 {
        println!("open failed");
        return -1;
    }
    // 写入一点数据，不关心内容
    let _ = write(fd as usize, b"x");
    close(fd as usize);

    // 2. 重新以只读方式打开，用于 fstat
    let fd = open(fname, OpenFlags::RDONLY);
    if fd < 0 {
        println!("open for fstat failed");
        return -1;
    }

    // 3. 调用 fstat
    let mut st = Stat::default();
    let ret = fstat(fd as usize, &st);
    if ret < 0 {
        println!("fstat failed");
        return -1;
    }

    // 4. 打印并验证原有字段
    println!("fstat returned:");
    println!(" dev   = {}", st.dev);
    println!(" ino   = {}", st.ino);
    println!(" mode  = {:?}", st.mode);
    println!(" nlink = {}", st.nlink);

    // 测试条件：inode 非 0、类型是普通文件、链接数为 1
    if st.ino != 0
        && st.mode == StatMode::FILE
        && st.nlink == 1
    {
```

自己写的测试程序

```rust
#![no_std]
#![no_main]
#![reexport_test_harness_main = "test_main"]
#![feature(custom_test_frameworks)]
#![test_runner(test_runner)]

#[macro_use]
extern crate user_lib;
use user_lib::{close, fstat, open, OpenFlags, Stat, StatMode};

/// 测试fstat，输出 Test fstat OK! 就算正确。

#[no_mangle]
pub fn main() -> i32 {
    let fname = "fname1\0";
    let fd = open(fname, OpenFlags::CREATE | OpenFlags::WRONLY);
    assert!(fd > 0);
    let fd = fd as usize;
    let stat: Stat = Stat::new();
    let ret = fstat(fd, &stat);
    assert_eq!(ret, 0);
    assert_eq!(stat.mode, StatMode::FILE);
    assert_eq!(stat.nlink, 1);
    close(fd);
    // unlink(fname);
    // It's recommended to rebuild the disk image. This program will not clean the file "fname1".
    println!("Test fstat OK!");
    0
}

pub fn test_runner(_test: &[&dyn Fn()]) {
    loop {}
}
```

文件里自带的测试程序，但是有个问题是该测试程序里面没有打印出具体 inode 信息而是只在测试完成后打印“test fstat ok”表示测试通过。

全部完成之后我们运行程序来进行测试：

```
[initproc] launching ch6_file3
test iteration 0
test iteration 1
test iteration 2
test iteration 3
test iteration 4
test iteration 5
test iteration 6
test iteration 7
test iteration 8
test iteration 9
Test mass open/unlink OK!
```

```
[initproc] launching ch6_file1
Test fstat OK!
[kernel] Application exited with code 0
[initproc] ch6_file1 exited with code 0
[initproc] launching ch6b_filetest_simple
file_test passed!
```

```
link linkname0 to fname2
link linkname1 to fname2
link linkname2 to fname2
Test link OK!
[kernel] Application exited with code 0
[initproc] ch6_file2 exited with code 0
[initproc] launching ch6_file0
Test file0 OK!
[kernel] Application exited with code 0
[initproc] ch6_file0 exited with code 0
[initproc] launching ch6_stat
fstat returned:
 dev   = 0
 ino   = 7
 mode  = FILE
 nlink = 1
Test fstat OK!
[kernel] Application exited with code 0
[initproc] ch6_stat exited with code 0
[initproc] launching symbol
EMPTY MAIN
[kernel] Application exited with code 0
[initproc] symbol exited with code 0
[initproc] all tests done!
```

测试全部通过。