

Linux 环境高级编程作业

Date: 2025-12-03

Contents

1. 整体设计	3
1.1. Disk 划分情况	3
1.2. API 层次设计	4
2. 代码结构	5
2.1. 项目文件结构	5
2.2. 主要 API 列表	6
3. 核心数据结构	9
3.1. disk	9
3.2. block	10
3.3. bitmap	11
3.4. filesystem	12
3.5. inode	13
3.6. direntry	14
3.7. path	15
3.8. cwd	15
3.9. file_handle	16
4. 运行测试	17
4.1. init	17
4.2. format	18
4.3. diskinfo and fsinfo	20
4.4. mkdir and ls	21
4.5. touch, write, cat and cp	22
4.6. stat, unlink and rmdir	23
4.7. hexdump analyze	24
4.8. stress	24

1. 整体设计

程序用 C 语言编写, 部分设计参考 Youtube Dr Jonas Birch 关于文件系统的项目: Filesystem from scratch. 实现了对磁盘文件的管理并暴露了一些上层文件系统接口.

1.1. Disk 划分情况

Disk 的空间划分如下:

Super block

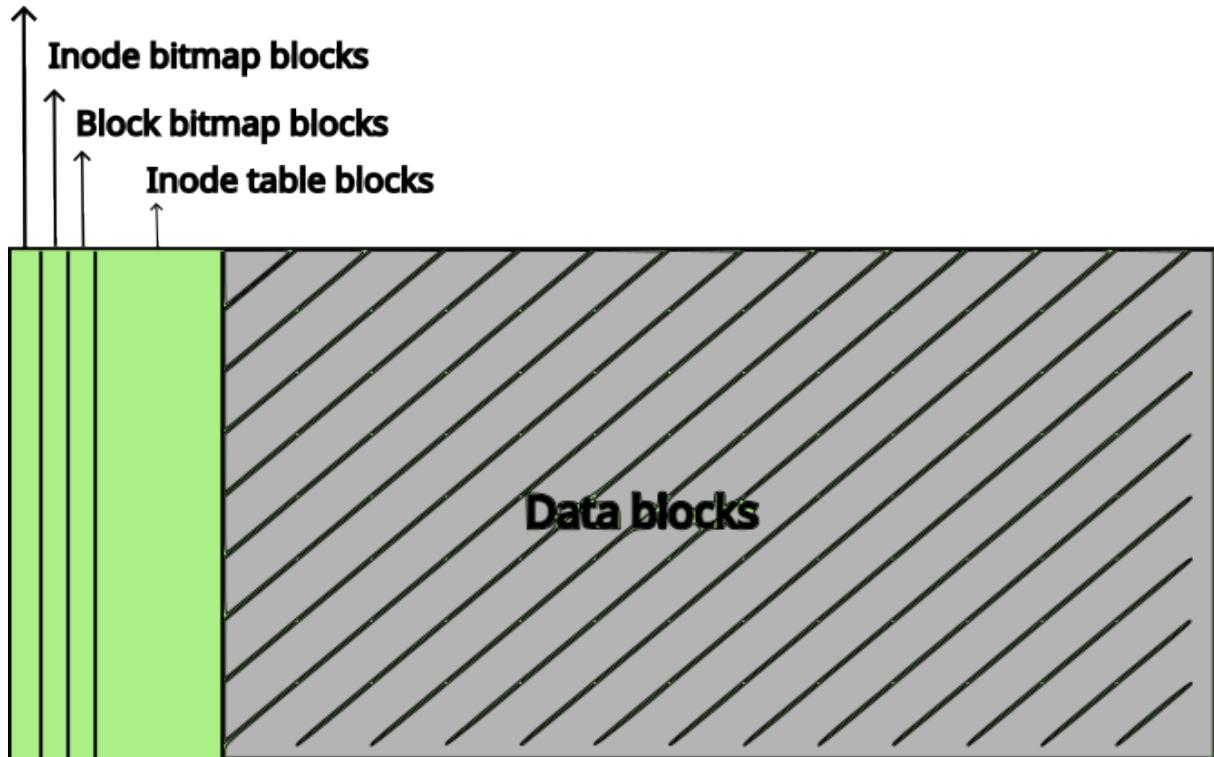


Figure 1: Disk structure

- Block size 是程序运行时动态确认的, 一般用 4096 bytes 也就是 4 KB
- Super block 占用一个 block
- Inode bitmap 和 Block bitmap 占用的 block 数量需要动态计算
- Inode table blocks 占用整个 disk 10% 的空间
- Data blocks 占用剩下 90% (减去 super block 和 bitmap blocks) 的空间

1.2. API 层次设计

结构如下:

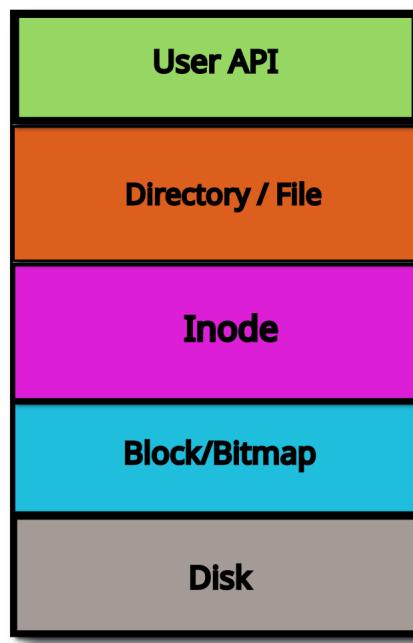


Figure 2: API structure

这里从下往上解释各层次的作用:

- **Disk** 层, 提供对 disk 文件的操作接口, 如获取 disk 文件描述符, 实际读/写一个 block
- **Block/Bitmap** 层, 提供 block 分配/释放的管理, 根据 bitmap 来分配空闲的块
- **Inode** 层, 提供 Inode 分配/释放, 以及读取/写入操作, 管理文件元信息
- **Directory/File** 层, 抽象出目录/文件操作的接口, 如读写目录/文件
- **User API** 层, 提供直接可用的文件操作接口, 类似 touch, mkdir 等 shell 命令

2. 代码结构

2.1. 项目文件结构

整体结构如下:

```
.  
├── Makefile          # Make 构建脚本  
├── README.md         # 说明文档和进度记录  
└── src               # 源代码目录  
    ├── lib              # 底层实现库  
    │   ├── bitmap.c      # Bitmap 相关, 位操作操作  
    │   ├── bitmap.h      #  
    │   ├── block.c       # Block 相关, block 分配/释放  
    │   ├── block.h       #  
    │   ├── cwd.c          # Current working directory 相关  
    │   ├── cwd.h          # 为了实现绝对路径/相对路径处理  
    │   ├── directory.c    # Inode type 是目录时, 对 dirent 的  
    │   ├── directory.h    # 增加/移除等操作  
    │   ├── dirent.c       # dirent 相关, 管理文件名和 inode number  
    │   ├── dirent.h       # 的映射关系  
    │   ├── disk.c          # 管理磁盘文件, 获取底层描述符, 保存磁盘  
    │   ├── disk.h          # 元信息  
    │   ├── error.h         # 错误返回码定义  
    │   ├── file.c          # File handle 相关, 将磁盘中的 inode 信息  
    │   ├── file.h          # 转换为内存中的 fh 信息, 便于文件信息共享  
    │   ├── fs_api.c        # 最上层 api 封装, 模拟日常 shell 操作  
    │   ├── fs_api.h        #  
    │   ├── fs.c             # 管理整个文件系统运行时信息, 并更新 bitmap  
    │   ├── fs.h             # 到磁盘  
    │   ├── inode.c          # Inode 管理, inode number 的分配/释放, 以及  
    │   ├── inode.h          # 对应 block 的管理  
    │   ├── path.c           # 实现对路径的解析和查找  
    │   └── path.h           #  
    └── main.c            # 上层应用程序  
    └── utils              #  
        ├── cat.c            # 模拟 cat 命令, 查看文件内容  
        ├── cp.c              # 模拟 cp 命令, 复制文件和目录  
        ├── diskinfo.c        # 模拟 fdisk -l 命令, 查看磁盘信息  
        ├── format.c          # 格式化一个 disk, 创建 super block 以及根目录  
        ├── fsinfo.c           # 查看文件系统信息, 比如 block 使用情况  
        ├── init.c             # 用 dd 创建一个 disk 文件  
        ├── ls.c               # 模拟 ls 命令, 查看目录内容  
        ├── mkdir.c            # 模拟 mkdir -p 命令, 递归创建目录  
        ├── rmdir.c            # 模拟 rm -r 命令, 递归删除目录  
        ├── stat.c             # 模拟 stat 命令, 查看文件对应的 inode 元信息  
        ├── stress.c           # 压力测试  
        ├── touch.c            # 模拟 touch 命令, 创建一个文件  
        ├── unlink.c           # 模拟 rm 命令, 删除一个文件  
        └── write.c            # 向文件写入内容
```

2.2. 主要 API 列表

这里仅列出关键结构的大致作用, 不包含参数和实现思路:

- ✓ dattach, 初始化 disk 结构体, 获取 low level 文件描述符
- ✓ ddetach, 释放 disk 资源
- ✓ dshow, 打印 disk 信息
- ✓ dread, 读取一个 block
- ✓ dwrite, 写入一个 block
- ✓ dreads, 读取一组 blocks
- ✓ dwrites, 写入一组 blocks
- ✓ bm_getbit, 获取某位
- ✓ bm_setbit, 设置某位
- ✓ bm_unsetbit, 取消某位
- ✓ bm_clearmap, 清理整个 bitmap
- ✓ get_inode_per_block, 获取每个 block 能容纳的 inode 节点数量
- ✓ bl_create, 创建一个 block
- ✓ bl_set_data, 设置 block 的 data 字段
- ✓ bl_get_data, 获取 block 的 data 字段
- ✓ bl_alloc, 查阅并更新 block bitmap, 分配一个可用的 block number (置为 1 表示已占用)
- ✓ bl_free, 查阅并更新 block bitmap, 释放一个 block number (置为 0 表示未占用)
- ✓ bl_clean, 初始化一个全 0 block
- ✓ fs_format, 用 fs 中定义的一些常量初始化 disk (操作磁盘文件)
- ✓ fs_mount, 读取 disk 文件信息, 初始化 filesystem 结构体
- ✓ fs_unmount, 释放 filesystem 结构体
- ✓ fs_show, 打印 filesystem 结构体信息
- ✓ ino_init, 清零一个 inode
- ✓ ino_alloc, 查阅并更新 inode bitmap, 分配一个可用的 inode number (置为 1 表示已占用)
- ✓ ino_free, 查阅并更新 inode bitmap, 释放一个可用的 inode number (置为 0 表示未占用)
- ✓ ino_read, 用 inode number 从磁盘读取一个 inode 信息
- ✓ ino_write, 向磁盘写入一个 inode 信息到指定 inode number
- ✓ ino_alloc_block_at, 向 direct_blocks 或 single_indirect 中分配可用的 block number
- ✓ ino_get_block_at, 从 direct_blocks 或 single_indirect 中读取一个 block number
- ✓ ino_free_block_at, 从 direct_blocks 或 single_indirect 中释放 block number
- ✓ ino_free_all_blocks, 从 direct_blocks 或 single_indirect 中释放所有 block number
- ✓ ino_show, 打印 inode 信息
- ✓ ino_is_valid, 检查是否保存有 inode number, 类型是否正确
- ✓ ino_get_block_count, 查看 inode 已分配多少 blocks
- ✓ ino_get_max_block_offset, 获取一个 inode 能管理的 blocks 的最大数目
- ✓ dirent_check_valid_name, 检查文件名是否符合要求, 这里是 [A-Za-z0-9._-]
- ✓ get_dirent_per_block, 获取一个 block 能存储的 dirent 数量
- ✓ dir_lookup, 从一个 directory inode 通过 name 查找对应的 inode number

- ✓ `dir_lookup_by_id`, 从一个 directory inode 通过 inode number 查找对应的文件的名字
- ✓ `dir_add`, 向一个 directory inode 中添加一个 directory entry
- ✓ `dir_remove`, 从一个 directory inode 中移除一个 directory entry
- ✓ `dir_list`, 列出一个 directory inode 中的所有 directory entries
- ✓ `dir_is_empty`, directory 是否为空 (可以有 . 和 ..)
- ✓ `dir_valid_name`, 目录名是否包含错误字符
- ✓ `dir_create_root`, 创建根目录, 添加两个 direntry: . 和 .. 都指向自身
- ✓ `dir_create`, 创建一个普通目录, 添加两个 direntry: . 指向自身, .. 指向父目录
- ✓ `dir_delete_empty`, 删除一个空目录 (只有 . 和 .. 的目录称空目录)
- ✓ `dir_show`, 打印 directory 信息
- ✓ `path_parse`, 将字符串转换为 path 数据结构, 按 / 拆分为 components, 并记录数量, 是否是绝对路径
- ✓ `path_to_string`, 将 path 根据是否是绝对路径拼接回路径字符串
- ✓ `path_show`, 打印一个 path 结构体的信息
- ✓ `path_is_valid`, 检查 path 是否合法
- ✓ `path_lookup`, 从一个 inode 查找制定 path 的 inode number
- ✓ `file_table_init`, 初始化一个全局的 file table
- ✓ `file_table_show`, 打印全局的 file table 信息
- ✓ `file_table_count`, 返回已打开的文件句柄数
- ✓ `file_open`, 按指定 flag 将一个 inode 转为内存中的 file handle
- ✓ `file_close`, 关闭一个 file handle
- ✓ `file_read`, 读取指定长度的文件内容
- ✓ `file_write`, 向文件写入指定长度的内容
- ✓ `file_seek`, 设置 file handle 的 offset
- ✓ `file_tell`, 返回当前的 offset
- ✓ `file_size`, 返回文件大小
- ✓ `file_show`, 打印文件信息
- ✓ `file_check_flags`, 检查文件的打开 flags 是否有效
- ✓ `file_chack_whence`, 检查文件的 offset 是否有效
- ✓ `cwd_init`, 初始化全局的 g_cwd 对象, 所有进程都能访问
- ✓ `cwd_get_inode`, 获取当前工作目录的 inode number
- ✓ `cwd_get_path`, 获取当前工作目录的 path 信息
- ✓ `cwd_chdir_inode`, 用 inode number 来变更当前工作目录
- ✓ `cwd_chdir_path`, 用 path 来变更当前工作目录
- ✓ `cwd_show`, 打印当前工作目录
- ✓ `fs_touch`, 创建一个文件, 类似 touch 命令
- ✓ `fs_unlink`, 删除一个文件, 类似 rm 命令
- ✓ `fs_cp`, 复制一个文件/目录, 类似 cp 命令
- ✓ `fs_mkdir`, 递归创建一个目录, 类似 mkdir -R
- ✓ `fs_rmdir`, 递归删除一个目录, 类似 rm -r

- ✓ `fs_ls`, 列出一个目录下的所有文件/目录
- ✓ `fs_cat`, 查看文件内容
- ✓ `fs_exists`, 查看文件是否存在
- ✓ `fs_stat`, 获取文件元信息

很多这里实现的接口暂时没有在 `fs_api` 上层中使用, 可以后续完善. 部分实现思路会在数据结构操作的章节描述.

3. 核心数据结构

下面的说明都以 `block_size` 为 4096 来说明. 在该项目中都使用 `stdint` 标准库中定义的类型, 比如 `uint8_t`, `uint16_t` 来替换 `char`, `int` 等类别, 以更好计算数据结构的长度信息.

3.1. disk

`disk` 是一个内存中的数据结构, 不会写入磁盘, 用来加载/卸载一个磁盘文件 (这里是普通的 Linux 大文件):

```
extern char *disk_paths[MAX_DISKS];

typedef uint8_t diskno; // Using to find disk file

// s_disk means: disk structure
struct s_disk {
    uint32_t fd;          // File descriptor, 4 bytes
    uint32_t blocks;      // Total block number, 4 bytes
    uint32_t block_size; // Block size, 4 bytes
    diskno id;
};

typedef struct s_disk disk;
```

- `fd` 是用 `open` 系统调用返回的文件描述符, 用来操作磁盘文件
- `blocks` 和 `block_size` 都是磁盘的元信息, 运行时通过命令行参数获取, 这样才能正确分割大文件为块
- `id` 用来定位磁盘文件的位置, 用作索引从 `disk_paths` 中获取文件路径:

```
char *disk_paths[MAX_DISKS] = {
    "/tmp/disk0.img",
    "/tmp/disk1.img",
    "/tmp/disk2.img",
    "/tmp/disk3.img",
    "/tmp/disk4.img",
    "/tmp/disk5.img",
    "/tmp/disk6.img",
    "/tmp/disk7.img",
    "/tmp/disk8.img",
    "/tmp/disk9.img",
};
```

这里说明 `dread/dwrite` 的思路, 每一个 block 有一个编号 `block number` (从 1 开始计数), `super block` 编号是 1, 后面的依次是 2, 3, 4 ...

读取 `super block`, 意味着读取:

- 0~4095 这 4096 个 bytes 的内容

可以得知读取的起始地址计算为: `start = (1-1) * 4096`, 即:

$$\text{start} = (\text{block_number} - 1) * \text{block_size}$$

3.2. block

block 的设计这里有些冗余, 后续发现不需要区分类别, 其就是一个 block_size 大小的字节数组,之所以不定义为 `uint8_t data[block_size]` 这样的预分配数组, 是因为 block_size 需要运行时通过参数获取, 并不能再编译期得到:

```
typedef enum {
    BlockTypeSuper,
    BlockTypeInode,
    BlockTypeData
} blocktype;

// 12 byte?
struct s_block {
    blocktype type;
    uint32_t block_size;
    uint8_t *data;
};

typedef struct s_block block;
```

由于 super block 是格式固定, 以便于写回磁盘后能够重新读取到内存中并区分出各个字段的含义. 因此单独创建一个结构体来管理, 只需要满足 `sizeof(super_block) < block_size` 即可:

```
struct s_superblock_data {
    uint16_t magic1; // 2 bytes
    uint16_t magic2; // 2 bytes
    uint32_t blocks; // 4 bytes
    uint32_t inodeblocks; // 4 bytes
    uint32_t inodes; // 4 bytes

    uint32_t inode_bitmap_start; // inode bitmap 起始块号
    uint32_t inode_bitmap_bl_count;

    uint32_t block_bitmap_start; // data block bitmap 起始块号
    uint32_t block_bitmap_bl_count;

    uint32_t inode_table_start; // inode table 起始块号

    uint32_t datablock_start; // 数据块起始位置
    uint32_t datablock_bl_count;

    uint32_t free_blocks; // 空闲块数 (用于快速检查)
    uint32_t free_inodes; // 空闲 inode 数
};

typedef struct s_superblock_data superblock_data;
```

- `magic number` 信息用来标记文件系统类别, 比如 ext2/ext3/ext4 的 `magic number` 为 `0xEF53`
- `superblock` 中存储的信息会在文件系统挂载时加载到内存中, 比如 `blocks` 会用来检查 `block number` 是否越界, 同理 `inodes`. 起始块号和块的数量可以用来定位一个连续的块区域, 比如 `inode bitmap` 存储在 2-3 block 上 (也具体范围动态计算)
- `free_blocks` 和 `free_inodes` 信息在后续程序中暂时没有用到 (一开始设计有, 现在也可以删除了), 大多数情况通过 `bitmap` 直接查询

3.3. bitmap

bitmap 本质上是一个大的字节数组, 用每一个 bit 来表示状态, 适用于需要大量布尔值的情况, 比如这里需要掌握所有 inode 和 block 的分配情况, 如果分配了, 则置为 1, 未分配则置为 0.

```
struct s_bitmap {
    uint8_t *bytes;
    uint32_t byte_len;
    uint32_t len; // max bit number
};

typedef struct s_bitmap bitmap;
```

- 同样由于 inode 的个数和 block 的总数是在运行时获取, 这里 bytes 通过后续堆上分配获取
- 之所以另外存储 byte_len (字节数量) 和 len (位数量):
 - byte_len 的使用场景: 创建字节数组时, 需要提供 byte 为单位的长度:
`(uint8_t*)malloc(byte_len)`. 以及清空整个字节数组时, 也需要提供 byte 为单位:
`memset(bytes, 0, byte_len)`
 - len 的使用场景: 用来检查 inode number 以及 block number 是否越界. 始终有 `index < len` 才行 (不然超出的部分没法用 bitmap 管理)

这里说明 bm_setbit 的思路, 其他的 bm_unsetbit 和 bm_getbit 都是类似的实现:

```
uint8_t bm_setbit(bitmap *bm, uint32_t idx);
```

首先要获取具体修改哪一个 byte, 这样才能通过 `bytes[index]` 这样的方式定位到:

- 比如 `idx = 18`, 那么应该修改 `idx / 8 = 2` 处的字节

```
byte_pos = idx / 8;
```

```
// byte index:
// 0      1      2
// bit index:
// 76543210 76543210 76543210
//
// 00000000 00000000 00000000
//                                ^
```

(需要注意, 虽然是 `index = 18`, 但实际上定位的是第 19 个 bit, 因为 `index = 0` 对应第 1 个 bit, `index = 1` 对应第 2 个 bit. 由于 inode 和 block number 最开始的设想是没有 0 这个 index, 因此实际传入的是 `inode number - 1`, 这样才能让 `inode number = 1` 对应传入 `index = 0`, 设置第 1 个 bit)

之后需要定位到具体 byte 中的某一位:

- 比如 `idx = 18`, 有 `idx % 8 = 2`, 通过构造一个 mask 如 `00000100` 然后进行 | 运算, 就能将对应位置为 1:

```
bit_pos = idx % 8;
```

```
bm->bytes[byte_pos] = bm->bytes[byte_pos] | (1 << bit_pos);
```

3.4. filesystem

`filesystem` 是内存中的数据结构, 用于从 `superblock` 中读取元信息并在程序运行期间维护整个文件系统的信息, 也要负责刷新 `bitmap` 到磁盘, 便于下次运行时获取分配信息.

```
struct s_filesystem {
    disk *dd;                                // Low level disk simulator

    // superblock metadata
    uint32_t blocks;
    uint32_t inodeblocks;
    uint32_t inodes;
    uint32_t inode_bitmap_start; // Dynamic calc
    uint32_t inode_bitmap_bl_count;
    uint32_t block_bitmap_start; // Dynamic calc
    uint32_t block_bitmap_bl_count;
    uint32_t inode_table_start; // Dynamic calc
    uint32_t datablock_start; // Dynamic calc
    uint32_t datablock_bl_count;

    // Bitmap objs
    bitmap *inode_bitmap; // inode alloc
    bitmap *block_bitmap; // block alloc

    // Thread synchronization
    pthread_mutex_t dir_lock; // Protects directory operations (dir_add,
    dir_remove, dir_lookup)
};

typedef struct s_filesystem filesystem;
```

- 这里大部分字段与 `superblock` 对应
- `dir_lock` 用于对 `inode` 类型为 `directory` 的节点进行操作时加锁. 这里主要是因为简化设计没有添加 `opendir`, `closedir` 等接口, 没有单独的 `directory` 结构体, 因此放在这里了. 对于 `inode` 类型为 `file` 的节点而言, 在内存中会加载为 `file_handle` 结构, 其中单独包含了锁

挂载一个文件系统, 就是指初始化一个 `filesystem` 结构体. 具体会读取 `superblock` 的内容并获取一个指向 `dd` 的指针 (获取 `block_size` 等信息).

卸载文件系统时进行 `bitmap` 的刷新操作, 并且要释放对应内存.

3.5. inode

inode 是最重要的一个数据结构, 需要存储在磁盘中, 负责存储文件的元信息, 包括: inode number (定位这个 inode 的位置), 文件类型, 文件大小, 包含文件存储内容的 block 位置等. 这里是简化的实现, 没有包含 refcount 引用计数, 因此没有实现软硬链接等, 也没有包含权限和用户/所属组等信息.

```
// 64 bytes
struct s_inode {
    uint32_t inode_number;
    filetype file_type;
    uint32_t file_size;

    uint32_t direct_blocks[DIRECT_POINTERS]; // 12 * 4 = 48 bytes

    uint32_t single_indirect;
};

typedef struct s_inode inode;
```

- 这里尽量让 inode 的大小是 16 的倍数, 这样不需要跨 block 存储同一个 inode 信息, 比如 block_size = 512, 就能存储 $512 / 64 = 8$ 个; block_size = 4096, 就能存储 $4096 / 64 = 64$ 个
- direct_blocks 数组中包含 block_number, 直接对应存储数据的 block 的位置
- single_indirect 是一级间接指针, 也是一个 block_number, 只不过其指向的 block 中存储的是 block_number 而不是具体的数据. 比如 block_size = 4096, block_number 长度是 32 位, 因此可以路由 $4096 / 32 = 128$ 个 block_number

这里的 inode 设计总共能存储 $(128 + 12) * 4096 = 573440$ 字节的内容, 也就是文件的最大大小, 如果想要增大, 则需要设计二级间接指针, 三级间接指针, 与之对应的, 会导致后续一些遍历的程序编写更复杂, 这里保持简单因此只用一级.

具体接口实现中需要注意 ino_alloc_block_at 这类实现, 主要是为了支持稀疏文件 (偶然看到的觉得有用就加上了), 比如这里可以路由的 block 数量为 $128+12=140$ 个, 因此 offset 可以是 0~139:

- offset=0 对应 direct_blocks[0]
- offset=1 对应 direct_blocks[1]
- offset=2 对应 direct_blocks[2]
- ...
- offset=12 对应 single_indirect 指向的 block 中的第一个 block_number 对应的 block

初始化时所有 offset 位置上都是 0, 这个接口的目的是为指定 offset 分配一个 block_number, 配合 ino_get_block_at 来进行随机读取, 而非只能顺序读取.

ino_read 和 ino_write 的思路一致, 需要计算偏移量以读取/写入磁盘的 inode table 区域: 从 filesystem 结构体中获取 inode table 的起始块号, 先定位 inode 存储在哪一个 block 中:

```
inodes_per_block = block_size / sizeof(inode);
block_offset      = inode_number / inodes_per_block;
```

然后再获取块内的偏移位置:

```
inode_offset = inode_number % inodes_per_block;
```

对应 inode 结构体在块内的起始地址就可以计算为:

```
begin_pos = inode_offset * sizeof(inode)
```

从这里开始读取 sizeof(inode) 长度即可.

3.6. direntry

direntry 是需要存储在磁盘中的数据结构, 是目录实际存储内容, 负责文件名和 inode number 之间的映射. 访问一个文件的流程就是找到文件名对应的 direntry, 然后找到 inoder number, 进而找到对应 block number 以读取内容.

```
// 256 bytes now
struct s_dirent {
    uint32_t inode_num;           // 4 bytes
    uint8_t name[MAX_FILENAME_LEN]; // 252 bytes
};
typedef struct s_dirent dirent;
```

- 这里设计同样保证是 16 的倍数, 避免一个 direntry 存储在不同的 block 中 (具体还是看 block_size 的设计)

direntry 结构体本身是非常简单的, 但其操作比较复杂, 具体对应 directory.h 中的接口, 需要实现增/删/查以及相应的调试接口.

这里说明 dir_add 的实现思路 (dir_remove 类似), 这里需要将更新刷写到磁盘, 因此需要加锁处理避免竞争条件.

由于这里没有引用计数以及软硬链接的实现, 因此判断一个 direntry 是否已经添加时, 既可以对比 file name, 也能对比 inode number (这里用 file name, 方便后续扩展). 比如 Linux 上的硬链接, 就是添加一个新的 direntry, 用不同的 file name 对应同一个 inode number.

dir_add 添加 direntry 时, 需要考虑当前已分配空间是否足够:

```
uint32_t block_used_space = dir_ino->file_size % fs->dd->block_size;
```

如果 block_used_space == 0 则说明没有剩余空间了. 因为对于一个目录而言, 创建时默认会添加两个 direntries, 此时 file_size = 64, 并分配有 1 个 block. 一个 direntry 是不会存储在两个 blocks 中的, 因此一个如果有某个 block 没有被填满, 那么 file_size % block_size 就一定不为 0.

由于可能是稀疏文件, 因此需要从第一个 block 开始一直遍历到最后一个 block, 这里 block_number == 0 表明该 offset 处没有分配 block_number, 对应空洞 (全零填充的部分):

```
block_number = ino_get_block_at(fs, dir_ino, n);
if (block_number == 0) {
    continue;
```

如果一个 block 是新分配的, 那么新 direntry 直接从 in_block_offset=0 位置开始存储. 如果是在已使用的 block 的剩余位置放置 direntry, 则需要遍历这个 block, 找到空余位置并插入.

3.7. path

path 用于解析字符串, 将其按照 / 拆分为多个 components, 便于顺序进行查找, 比如 /home/jie/Documents/hello.txt 会拆分为: ["home", "jie", "Documents", "hello.txt"], 然后查找顺序是: "home" 是 directory, 在其 direntries 查找 "jie"; 如果能找到, 且 "jie" 也是 directory, 则继续从其 direntries 中查找 "Documents"; 这样迭代查找直到最后一项.

```
struct s_path {
    char components[MAX_PATH_DEPTH][MAX_FILENAME_LEN];
    uint32_t count;           // Number of components
    uint8_t is_absolute;      // 1 if starts with '/', 0 otherwise
};

typedef struct s_path path;
```

- count 用于 for 循环遍历所有 components 时能提前返回并快速找到最后一项的 index

3.8. cwd

cwd (Current working directory) 结构体在内存中进行操作, 其添加主要是为了实现相对路径. 一般情况下, 需要配套 process 相关概念的实现, 但这里为了简化, 直接构建一个全局的 cwd, 所有线程/子进程可见.

比如需要创建的文件的路径是 hello/world.txt:

- 如果没有 cwd, 那么只能从根目录创建, 最终创建 /hello/world.txt 文件
- 如果有 cwd, 假如是 /home/jie, 那么此时就能创建 /home/jie/hello/world.txt 文件

```
struct s_cwd_context {
    filesystem *fs;

    uint32_t cwd_inode_num;
    inode cached_cwd_inode;
    uint8_t cache_valid;

    path p;
    pthread_rwlock_t rwlock;
};

typedef struct s_cwd_context cwd_context_t;

extern cwd_context_t g_cwd;
```

- 创建文件, 就是向类型为 directory 的 inode 添加 dirent, 因此需要 cwd_inode_num 来定位
- path p 是后续讲解的一个数据结构, 用于解析字符串路径, 便于利用文件名查找对应的 inode number
- pthread_rwlock_t rwlock 用于避免多线程修改时的竞争

3.9. file_handle

file_handle 是在内存中操作的数据结构. 其就是 inode 信息在内存中的表示, 再加上一些多线程/进程能够共享的信息, 比如 offset. 假如一个线程写入了 10bytes 的内容:

- 如果有 offset 共享, 此时 offset 则更新为 10; 另一个线程写入时就能从 offset 处接着写入
- 如果没有 offset 共享, 另一个线程写入时也从文件开头写入, 就会覆盖之前写入的内容

file_handle 的 open, close, read, write 等接口都仿照 C 标准库来实现.

```
struct s_file_handle {
    filesystem *fs;

    uint32_t inode_number;
    inode cached_inode;
    uint8_t cache_valid;

    uint32_t refcount;
    uint32_t offset;
    uint32_t flags;

    pthread_rwlock_t rwlock;
};

typedef struct s_file_handle file_handle;

struct s_global_file_table {
    file_handle *handles[MAX_OPEN_FILES];
    pthread_mutex_t lock;
    uint32_t count;
};

typedef struct s_global_file_table global_file_table;

extern global_file_table g_file_table;
```

- inode_number 指向该 file_handle 打开的文件, 用于获取文件内容, 文件大小等信息
- cached_inode 和 cache_valid 用于快速读取 inode 信息, 而非每次都用 inode_number 从磁盘查找并读取
- refcount 用于 file_handle 的共享, 但在这里还没有实现. 在 UNIX 接口中, fork 时子进程会复制父进程的 fd table, 在过程中会增加该引用计数, 在 close 一个 file_handle 时, 如果 refcount 为 0, 则释放改 file_handle, 否则只是让计数减 1
- flags 用于指定打开方式, 比如 Read only, Write only
- rwlock 用与避免多个线程对同一个文件进行操作时产生的竞争关系

s_global_file_table 用来限制当前文件打开数量, 全局管理整个进程开启的 file_handle, 以进行清理工作. 在多线程场景下也通过 lock 提供保护.

对于 file_read 和 file_write 的实现要注意一点, 这里读取/写入的时候并没有跳过空洞, 如果对应 offset 没有分配 block_number, 就会进行分配操作.

4. 运行测试

```
git clone https://github.com/zKurisu/SimpleFS.git  
cd SimpleFS  
make build
```

在 src/utils 目录下的文件会构建出下面工具:

- my_init, 创建一个磁盘文件
- my_format, 格式化一个磁盘文件, 并创建根目录
- my_diskinfo, 打印磁盘信息
- my_fsinfo, 打印文件系统信息
- my_mkdir, 递归创建目录
- my_rmdir, 递归删除目录
- my_ls, 列出目录下的内容(不递归)
- my_touch, 创建文件
- my_cp, 复制文件
- my_unlink, 删除文件(没有实现软/硬链接, 最初的 inode 结构体中就没有保存 ref 信息)
- my_write, 向文件写入内容
- my_cat, 打印文件内容
- my_stat, 查看文件/目录元信息
- my_stress, 多线程下的压力测试

分别进行测试并验证 disk 文件的内容.

4.1. init



```
homework git:(master) ✘ ./build/my_init 0 10000 4096  
10000+0 records in  
10000+0 records out  
40960000 bytes (41 MB, 39 MiB) copied, 0.0250976 s, 1.6 GB/s  
init: success to run [dd if=/dev/zero of=/tmp/disk0.img count=10000 bs=4096]  
init: created disk0.img with 10000 blocks of 4096 bytes  
homework git:(master) ✘ ls -l /tmp | grep disk  
.rw-r--r-- 41M jie 4 Dec 20:54 disk0.img  
homework git:(master) ✘ hexdump /tmp/disk0.img  
00000000 0000 0000 0000 0000 0000 0000 0000  
*  
2710000  
homework git:(master) ✘
```

Figure 3: SimpleFS init result

成功构建一个全 0 的文件.

4.2. format

```
⚡ homework git:(master) ✘ ./build/my_format 0 4096
Write inode bitmap block at block number: 2
Write inode bitmap block at block number: 3
Start block number for block bitmap is 4
format: success to run fs_format
format: format disk0.img with 4096 blocks of (null) bytes
format: success to init root directory with inode number [1]
⚡ homework git:(master) ✘ hexdump /tmp/disk0.img
00000000 0004 0017 2710 0000 03e8 0000 fa00 0000
00000010 0002 0000 0002 0000 0004 0000 0001 0000
00000020 0005 0000 03ed 0000 2324 0000 2324 0000
00000030 fa00 0000 0000 0000 0000 0000 0000 0000
00000040 0000 0000 0000 0000 0000 0000 0000 0000
*
00010000 0001 0000 0000 0000 0000 0000 0000 0000
00010100 0000 0000 0000 0000 0000 0000 0000 0000
*
00030000 ffff ffff ffff ffff ffff ffff ffff ffff
*
00030700 ffff ffff ffff ffff ffff ffff 3fff 0000
00030800 0000 0000 0000 0000 0000 0000 0000 0000
*
00040000 0001 0000 0002 0000 0200 0000 03ee 0000
00040100 0000 0000 0000 0000 0000 0000 0000 0000
*
00040300 0000 0000 0000 0000 0000 0000 03ed 0000
00040400 0000 0000 0000 0000 0000 0000 0000 0000
*
03ed0000 0001 0000 002e 0000 0000 0000 0000 0000
03ed0100 0000 0000 0000 0000 0000 0000 0000 0000
*
03ed1000 0001 0000 2e2e 0000 0000 0000 0000 0000
03ed1100 0000 0000 0000 0000 0000 0000 0000 0000
*
2710000
⚡ homework git:(master) ✘
```

Figure 4: SimpleFS format result

0000000 地址是第一个 block, 该部分内容对照如下:

- 0004 是 magic1, 0017 是 magic2
- 2710 0000 (32 位) 换算为 10 进制是 10000, 恰好是 init 程序中创建的 block 数量, 对应 blocks
- 03e8 0000 (32 位) 换算成 10 进制是 1000, 恰好 blocks 的 10%, 对应 inodeblocks
- fa00 0000 (32 位) 换算成 10 进制是 64000, 每个 block 能存 $4096/64=64$ 个 inode, 所有 inodeblocks 共存放 $64*1000=64000$ 个, 对应 inodes

- 0002 0000 对应 inode bitmap 的起始块号
- 0002 0000 对应 inode bitmap 所需的 block 数量, 计算: $64000/(4096*8)=1$, 且 $64000\%(4096 * 8) \neq 0$, 因此需要 2 个 blocks
- 0004 0000 对应 block bitmap 的起始块号
- 0001 0000 对应 block bitmap 所需的 block 数量, 计算: $10000/(4096*8)=0$, 且 $\frac{10000}{4096*8} \neq 0$, 因此需要 1 个 block
- 0005 0000 是 inode table 的起始块号
- 03ed 0000 是 data block 的起始块号, 对应十进制是 1005, 恰好是 1000 个 inode blocks 加上 1 个 superblock 加上 2 个 inode bitmap block 加上 1 个 block bitmap block
- 2324 0000 是数据块的数量, 换算为十进制是 8996, 恰好是 $10000 - 1000 - 1 - 2 - 1$
- 2324 0000 是 free_blocks, 计算的就是数据块的空闲数目
- fa00 0000 是 free_inodes 换算成十进制是 64000 (这里应该减 1, 但这个字段未使用因此暂时可以忽略)

0001000 换算成十进制是 4096, 恰好是第二个块的开始, 这里是 inode bitmap:

- 0001 0000 是由于分配了根目录消耗了一个 inode

0003000 换算成十进制是 12288, 恰好是第四个块的开始, 也就是 block bitmap:

- 这些分配考虑了一开始分配好的 1000 个 inode block

0004000 是第五个块的开始地址, 是 inode table:

- 0001 0000 是 inode number, 对应根目录的 number
- 0002 0000 是 file type, 对应 directory
- 0000 0200 是 file size, 换算成十进制是 512, 恰好是两个 dirent, 即 . 和 .. 相加的大小
- 03ee 0000 是 direct_blocks 的第一个元素, 由于有了两个 dirent, 因此需要分配一个块来存储, 块号是 1006, 即格式化后的第一个空闲块
- 03ed 0000 是 single_indirect 指向的块号, 是 1007

03ed000 换算成十进制是 4116480, 处以 4096 发现是 1005, 即第 1006 个块的起始位置, 对应的块号是 1006, 即根目录第一个直接块的位置:

- 0001 0000 是 inode number
- 002e 0000 是 ., 表示文件名映射

同理后面 002e 002e 是 .., 也是文件名映射.

4.3. diskinfo and fsinfo

```
✗ homework git:(master) ✘ ./build/my_diskinfo 0 4096
diskinfo: success to run fs_diskinfo
Disk 0:
blocks 10000
block size 4096
total size 40960000
✗ homework git:(master) ✘ ./build/my_fsinfo 0 4096
=====
Filesystem Information
=====
Disk:
Disk ID:          0
Block size:       4096 bytes
Total blocks:    10000
Total size:      40960000 bytes (40000.00 KB)

Filesystem Layout:
Total blocks:    10000
Inode blocks:    1000 (10.0%)
Total inodes:    64000
Data blocks:     9000 (90.0%)

Block Allocation Map:
[Block  1]:   Super Block
[Block  2]:   Inode Bitmap (2 blocks)
[Block  4]:   Block Bitmap (1 block)
[Block 5-1004]: Inode Table (1000 blocks)
[Block 1005-10000]: Data Blocks (9000 blocks)

Bitmap Statistics:
Inodes:
  Total:        64000
  Used:         1 (0.0%)
  Free:        63999 (100.0%)

Blocks:
  Total:        10000
  Used:        1006 (10.1%)
  Free:        8994 (89.9%)
  Free (data): 8994 blocks (36839424 bytes, 35976.00 KB)
=====

✗ homework git:(master) ✘
```

Figure 5: SimpleFS diskinfo and fsinfo result

4.4. mkdir and ls

```
homework git:(master) ✘ ./build/my_ls 0 4096 "/"
=====
Type Size(bytes)
d. 512 .
d. 512 ..
=====

ls: success to run [fs_ls]
homework git:(master) ✘ ./build/my_mkdir 0 4096 "/home/jie/homework"
input path is: /home/jie/homework
input path is: /home/
input path is: /home/jie/
mkdir: success to run [fs_mkdir]
mkdir: success to create dir [/home/jie/homework]
homework git:(master) ✘ ./build/my_ls 0 4096 "/"
=====
Type Size(bytes)
d. 768 .
d. 768 ..
d. 768 home
=====
ls: success to run [fs_ls]
homework git:(master) ✘ ./build/my_ls 0 4096 "/home"
=====
Type Size(bytes)
d. 768 ..
d. 768 .
d. 768 jie
=====
ls: success to run [fs_ls]
homework git:(master) ✘ ./build/my_ls 0 4096 "/home/jie"
=====
Type Size(bytes)
d. 768 ..
d. 768 .
d. 512 homework
=====
ls: success to run [fs_ls]
homework git:(master) ✘
```

Figure 6: SimpleFS mkdir and ls result

4.5. touch, write, cat and cp



```
✗ homework git:(master) ✗ ./build/my_touch 0 4096 "/home/jie/hello.txt"
touch: success to run [fs_touch]
touch: success to create /home/jie/hello.txt
✗ homework git:(master) ✗ ./build/my_ls 0 4096 "/home/jie"
=====
Type Size(bytes)
d. 768
d. 1280
d. 512
f. 0
=====
ls: success to run [fs_ls]
✗ homework git:(master) ✗ ./build/my_write 0 4096 "/home/jie/hello.txt" 0 "hello world"
write: success to run [fs_write]
write: success to write 11 bytes to [/home/jie/hello.txt]
✗ homework git:(master) ✗ ./build/my_cat 0 4096 "/home/jie/hello.txt"
hello world
=====
cat: success to run [fs_cat]
✗ homework git:(master) ✗ ./build/my_cp 0 4096 "/home/jie/hello.txt" "/home/jie/new_hello.txt"
cp: success to run [fs_cp]
cp: success to cp file [/home/jie/hello.txt] to [/home/jie/new_hello.txt]
✗ homework git:(master) ✗ ./build/my_ls 0 4096 "/home/jie"
=====
Type Size(bytes)
d. 768
d. 1536
d. 512
f. 11
f. 11
=====
ls: success to run [fs_ls]
✗ homework git:(master) ✗
```

Figure 7: SimpleFS touch, write, cat and cp result

4.6. stat, unlink and rmdir

```
✗ homework git:(master) ✘ ./build/my_stat 0 4096 "/home/jie/hello.txt"
stat: success to run [fs_stat]
stat info:
  path   : /home/jie/hello.txt
  type   : file
  size   : 11
  blocks : 1
✗ homework git:(master) ✘ ./build/my_stat 0 4096 "/home/jie/new_hello.txt"
stat: success to run [fs_stat]
stat info:
  path   : /home/jie/new_hello.txt
  type   : file
  size   : 11
  blocks : 1
✗ homework git:(master) ✘ ./build/my_unlink 0 4096 "/home/jie/new_hello.txt"
unlink: success to run [fs_unlink]
unlink: success to remove /home/jie/new_hello.txt
✗ homework git:(master) ✘ ./build/my_ls 0 4096 "/home/jie"
=====
Type Size(bytes)
d.    768
d.   1536
d.    512
f.     11
=====
ls: success to run [fs_ls]
✗ homework git:(master) ✘ ./build/my_rmdir 0 4096 "/home/jie"
dir_remove error: entry 'homework' not found
rmdir: success to run [fs_rmdir]
rmdir: success to remove dir [/home/jie]
✗ homework git:(master) ✘ ./build/my_ls 0 4096 "/home"
=====
Type Size(bytes)
d.    768
d.    768
=====
ls: success to run [fs_ls]
✗ homework git:(master) ✘ []
```

Figure 8: SimpleFS stat, unlink and rmdir result

4.7. hexdump analyze

```
homework git:(master) ✘ hexdump -C /tmp/disk0.img
00000000  04 00 17 00 10 27 00 00 e8 03 00 00 00 00 fa 00 00 | .....!
00000010  02 00 00 00 02 00 00 00 04 00 00 00 01 00 00 00 | .....
00000020  05 00 00 00 ed 03 00 00 28 23 00 00 24 23 00 00 | .....(#..$..
00000030  00 fa 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00001000  03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00003000  ff | .....
*
00003070  ff 00 00 | .....
00003080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00004000  01 00 00 00 02 00 00 00 00 03 00 00 ee 03 00 00 | .....
00004010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00004030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 ed 03 00 00 | .....
00004040  02 00 00 00 02 00 00 00 00 03 00 00 fo 03 00 00 | .....
00004050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00004070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 ef 03 00 00 | .....
00004080  03 00 00 00 02 00 00 00 00 06 00 00 f2 03 00 00 | .....
00004090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
000040b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 f1 03 00 00 | .....
000040c0  04 00 00 00 02 00 00 00 00 02 00 00 f4 03 00 00 | .....
000040d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
000040f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 f3 03 00 00 | .....
00004100  05 00 00 00 01 00 00 00 0b 00 00 00 f6 03 00 00 | .....
00004110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00004130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 f5 03 00 00 | .....
00004140  06 00 00 00 01 00 00 00 0b 00 00 00 f8 03 00 00 | .....
00004150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
00004170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 f7 03 00 00 | .....
00004180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
003ed000  01 00 00 00 2e 00 00 00 00 00 00 00 00 00 00 00 | .....
003ed010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
003ed100  01 00 00 00 2e 2e 00 00 00 00 00 00 00 00 00 00 | .....
003ed110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
003ed200  02 00 00 00 68 6f 6d 65 00 00 00 00 00 00 00 00 | ....home.....
003ed210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
*
003ef000  01 00 00 00 2e 2e 00 00 00 00 00 00 00 00 00 00 | .....
```

Figure 9: SimpleFS hexdump disk

4.8. stress

该部分代码让 AI 编写的, 运行有些问题, 准备后续修改更新.