实验报告

一、结构设置

1. 总体结构

总体结构采用 VSFS 结构，即 SuperBlock + Inode Bitmap + Data Bitmap + InodeBlock + DataBlock 的结构。

其中，SuperBlock 占据第一个块。一个 block 共有 4*1024*8=32768 bit，可以表示 32768 个文件或数据块的占用情况，由于一共 65536 个块且需要支持 32768 个文件，所以 inode bitmap 需要占一个数据块，data bitmap 需要占两个数据块。

自定义的 inode 的结构大小为 92B，所以一个 block 能放 4*1024/92 = 44 个 inode。一共需要 32768 个 inode，所以需要 32768/44 = 745 个 inode block。

剩余的空间为 datablock，去除前面的部分还剩余 64787 个 block，大小为 64787*4KB=259148KB，大于 250MB=256000KB，故满足“至少 250MiB 的真实可用空间”的要求。

总体结构大致如下表所示：

SuperBlock	Inode Bitmap	Data Bitmap1	Data Bitmap2	Inode Block	Data Block
0	1	2	3	4-748	749-65535

2. SuperBlock 结构

Superblock 结构未修改，表明整个文件系统的一些基本信息。

```
typedef struct SuperBlock{
    unsigned long f_bsize; //块大小
    fsblkcnt_t f_blocks;//块数量
    fsblkcnt_t f_bfree; //空闲块数量
    fsblkcnt_t f_bavail;//可用块数量
    fsfilcnt_t f_files; //文件节点数
    fsfilcnt_t f_ffree; //空闲节点数
    fsfilcnt_t f_favail;//可用节点数
    unsigned long  f_namemax;//文件名长度上限
}SuperBlock;
```

3. Bitmap 结构

Bitmap 的结构用 int 来实现，每个 int 的 32 位能代表 32 个文件或子目录的使用情况（1 为被占用，0 为空闲可用）。具体操作时需要使用位操作提取对应位置的使用情况。

```
typedef struct Bitmap{//int为4B，1个数据块4KB，等于1024个int
//每个int的每bit都表示一个block是否被占用
    int used[1024];
}Bitmap;
```

4. Inode 结构

Inode 中，有一些存储基本信息的变量，还有 block_num 记录使用的数据块的数量。为

了节省空间，没有使用 8B 的指针结构，而是用 4B 的 int 直接存储指向的数据块的实际编号，因为总共 65536 个块，只需要 16bit 便足够完全表示，所以可以用 int 来存储。

在 indirect 间接指针块中，为了防止将无意义内容识别为指针（数据块编号），将该指针块的第一个 int 用来记录该指针块的指针数。因此一个 indirect block 能存储 4KB/4B – 1 = 1023 个指向数据块的指针。间接指针块结构如下：

Indirect Block			
num	Pointer1	Pointer2	……

由此可以计算一个文件最大可能的大小。12 个直接指针共 12*4KB = 48KB。两个间接指针共 2*1023*4KB = 8184KB，所以单个文件可能的最大大小=48+8184 = 8232KB > 8MB，符合要求。

```
typedef struct Inode{  //inode-size: 92B
    mode_t mode; //表明是文件还是目录
    off_t size; //文件大小（字节数）
    time_t atime; //被访问的时间
    time_t ctime; //状态改变的时间
    time_t mtime; //被修改的时间
    int block_num; //一共使用了几个数据块
    int direct_pointer[12]; //直接指针，记录数据块的编号
    int indirect_pointer[2]; //间接指针
}Inode;
```

5. 目录数据块结构

目录数据块的目录项为文件或子目录，需要存储对应的名字和 inode 编号以供查找。所以定义 Dir_entry 结构表示目录项，由存储名字的 char 数组和 inode 编号组成。Directory_Block 结构表示目录数据块，由其目录项项数和目录项数组组成。

```
typedef struct Dir_entry{
    char filename[MAX_NAME_LENGTH + 1];
    int inode_id;
}Dir_entry;

typedef struct Directory_Block{
    int nums; //该目录下文件/子目录的数量
    Dir_entry dir_entry[DIR_ENTRY_NUM];
}Directory_Block;
```

二、辅助函数

使用的辅助函数数量较多，在此不一一讲解，以免篇幅过长，仅介绍它们的作用，具体的实现请见 "fs.c"

Read_SuperBlock() 读取 super block 块，返回一个 SuperBlock 结构。

Write_SuperBlock(SuperBlock superblock) 将传入的参数写进 super block 块。

有关 Bitmap 的操作主要通过位运算来实现。

Read_InodeBitmap() 读取 inode bitmap，并查找空闲的 inode，返回空闲可用的 inode 编号。

Read_DataBitmap(int blockid) 读取参数块号对应的 data bitmap，并查找空闲的 datablock，返回空闲可用的 data block 编号

Write_Bitmap(int blockid, int changeid, int isdelete) 修改 bitmap 块中参数对应的位置的空闲状态。如果 isdelete 为 0，表示将空闲块修改为占用状态，即 0 变 1；否则表示释放一个占用块，状态由占用变空闲，即 1 变 0。

Read_Inode(int inode_id) 读取编号对应的 inode 并将其以 Inode 结构返回

Write_Inode(int inode_id, Inode inode) 将参数传入的 inode 结构写入编号 inode_id 对应的 inode。

Read_Dir_DB(int blockid, void* buffer, fuse_fill_dir_t filler) 读取一个数据块中所有文件或子目录

有关目录的操作:

New_Indirect_Block(int indirect_block) 在参数对应的 block 处创建并初始化间接指针块

Open_New_DirBlock(int blockid) 将参数对应的 block 设置为新的目录数据块并初始化

Insert_In_Dir(int dir_inode, Dir_entry file_or_dir) 在编号为 dir_inode 的目录中插入目录项 file_or_dir。其实现情况比较复杂，因为需要找到该目录的最后一个目录块，并根据其是否已满、以及属于直接指针还是间接指针等不同情况做不同处理。

Find_From_DirDB(int blockid, char* name) 从编号为 blockid 的目录块中按名字 name 找文件或子目录，找到则返回对应的 inode 编号，否则返回-1

Find_InWhich_Block(int dir_inode, char* name) 从编号为 dir_inode 的目录中，查找名为 name 的文件或子目录在哪个数据块，若找到返回对应的数据块编号，否则返回-1

有关文件路径的操作:

Get_Inodeid_From_Path(char* path) 从路径 path 得到对应文件的 inode 编号,找不到返回-1

Get_FileName(char* path) 从路径 path 中提取文件名

Get_Father_Path(char* path) 从路径 path 中提取父目录路径

Find_DB_From_Order(Inode inode, int order) 找到文件的第 order 个数据块的块号

三、具体实现

需要完成的函数数量较多，所以不在此展示具体实现代码，仅仅简单讲解实现思路，具体代码请见“fs.c”

1. **mkfs** 初始化文件系统

   需要初始化 SuperBlock 块、Bitmap 块，并为根目录录入信息。

2. **fs_getattr** 查询一个目录文件或常规文件的信息

用所给路径读出对应的 inode 结构，并存入所给的结构体 stat 中即可

3. **fs_readdir** 查询一个目录文件下的所有文件

   首先用所给路径读出该目录的 inode 结构，然后根据 block_num 依次遍历其直接指针、间接指针 1 和间接指针 2，利用 Read_Dir_DB 函数打印每个目录数据块的信息，并最终在修改 atime 后重新写回该目录的 inode 即可

4. **fs_read** 对一个常规文件进行读操作

   因为要从 offset 位置开始读 size 大小，但是由于先定位 offset 再向后找 size 大小的情况比较复杂，所以直接读取到 offset+size 的位置再删除前 offset 个字节，操作更简单。读取时依次遍历其直接指针、间接指针 1 和间接指针 2，过程中只要达到了 offset+size 处就可以停止。需要注意的是，可能文件总大小小于 offset+size，那便只需要读到文件结束处即可。最终返回实际读取的大小。

5. **fs_mkdir** 创建一个目录文件

   这里操作比较简单，主要调用辅助函数。对待创建的子目录，首先通过 bitmap 找到空闲的 inode 并修改为占用状态，然后初始化子目录的 inode 并写入对应位置。对其父目录，则要将该子目录对应的目录项利用 Insert_In_Dir 函数，插入到父目录的目录块中即可。

6. **fs_rmdir** 删除一个目录文件

   删除子目录，首先要删除该目录本身的数据，即修改其所有指针指向的数据块在 bitmap 中的状态为空闲可用，以及修改其 inode 在 bitmap 中的状态为空闲可用。然后针对父目录，需要将该子目录所在的目录块中，位于其后的所有目录项依次前移一位，以达到在父目录中删除的效果。

7. **fs_unlink** 删除一个常规文件

   删除文件的操作与 fs_rmdir 中删除子目录的操作相同。

8. **fs_rename** 更改一个目录文件或常规文件的名称（或路径）

   实际可以拆分为两个操作，即在旧路径中删除，在新路径中插入。不过删除操作有所不同的是，不需要修改该文件自身的 inode 和 datablock，只需要将父目录中其后的目录项依次前移一位即可。插入操作则直接调用辅助函数 Insert_In_Dir。

9. **fs_truncate** 修改一个常规文件的大小信息

   将更改大小分为增大和缩减两种情况。增大时需要分配新的数据块，要依次尝试直接指针、间接指针 1 和间接指针 2，同时还要注意是否需要分配新的间接指针块等细节处理。缩减时，从末尾开始向前删除，但不需要真的删除数据块中的内容，只需要修改对应的 bitmap 即可，其中的数据便可自然作废。

10. **fs_utime** 修改一个目录文件或常规文件的时间信息

    只需要读出对应 inode 并修改时间即可

11. **fs_mknod** 创建一个常规文件

与 fs_mkdir 类似，主要调用辅助函数。对待创建的文件，首先通过 bitmap 找到空闲的 inode 并修改为占用状态，然后初始化文件的 inode 并写入对应位置。对其父目录，则要将该文件对应的目录项利用 Insert_In_Dir 函数，插入到父目录的目录块中即可。

12. **fs_write** 对一个常规文件进行写操作

    根据提示，write 前先检查大小是否足够，所以先调用 fs_truncate 分配足够大小。然后确定需要写入的数据块范围，遍历它们，依次写入 buffer 中的内容即可。

13. **fs_statfs** 查询文件系统整体的统计信息

    读出 superblock 块的内容存入所给结构体即可。

根据 PPT 提示，fs_open、fs_release、fs_opendir、fs_releasedir 无需修改

四、效果展示

用 traces 中的样例测试，除了测试点 13 拷贝 test.in 有问题外，均能正常运行且输出与相应 ans 一致。

例如测试 10 的效果：

```
[2020202279@work122 fslab-handout]$ cd mnt
[2020202279@work122 mnt]$ echo 12345 > file1
[2020202279@work122 mnt]$ more file1
12345
[2020202279@work122 mnt]$ echo 23333333333 > file1
[2020202279@work122 mnt]$ more file1
23333333333
[2020202279@work122 mnt]$ echo abcdefg >> file1
[2020202279@work122 mnt]$ more file1
23333333333
abcdefg
[2020202279@work122 mnt]$
```