FSLab 实验报告

涂奕腾 2020201018

一、文件系统空间结构设计

在空间分配上，我主要借鉴了 menci 的思想，针对本实验实际上并不需要 SuperBlock，所以只保留 2 种 bitmap，inode，datablock 三个部分。

实验要求支持 32768 个文件/目录(即 32768 个 inode)且包含了 65536 个 4kb 大小的数据块，而一个 4kb 的数据块正好包含了 4k * 8 = 32768 位，因此恰好可以用 1 个 block 作 inode bitmap，2 个 block 作 datablock bitmap，分别占用数据块编号为 0,1,2。就 bitmap 的实现而言，我使用 unsigned long long 的数组进行本地存储，并通过位运算来维护(见 二)，当然如果可以使用 C++语言的话直接用 STL 的 bitset 会更方便。

```
typedef unsigned long long bitmap_item;
typedef bitmap_item inode_bitmap[512];
typedef bitmap_item datablock_bitmap[1024];
```

对于指针，我们实际上并不需要用到指针，可以使用块的编号代替指针，由于编号在 0-65535 之间，恰好可以用一个 unsigned short 来表示。

针对每个 inode，我们也可以只保留本实验中所需的关键信息，特别是针对每个文档至多只有 8MB，我们可以在每个 inode 中只使用一个 indirect pointer，使用 unsigned short 代替指针后，一个块中可以包含 4kb / 2 = 2kb 个指针，恰好可以表示 8MB 的文件。精简后 inode 的大小可以压缩到 32B：

```
typedef struct{//32 bytes
    time_t atime, ctime, mtime; //3 * 8 bytes
    unsigned int size; // 4 bytes
    unsigned short indirect_ptr;// 2 bytes
    bool directory;// 1 bytes => 2 bytes
}inode;
```

则在一个块中包含了 4kb / 32b = 128 个 inode，所有的 inode 应占用 32768 / 128 = 256 个块，因此将编号为 3-258 的块分配给 inode，剩余的 259-65535 共 65277 个块为真正的数据块，能够满足 250MB 的要求。

对于目录项，只需要存储文件名和 inode 编号即可，为了对齐的要求加上一个 4 字节的 padding。目录项设计如下：

```
typedef struct{//目录项 32bytes
    char filename[26];
    unsigned short inode_num;
    int padding;
}dir_item;
```

读取一个数据块上的内容时，使用一个 char 数组存储单个数据块：

```
typedef char block_item[BLOCK_SIZE];
```

二、结构辅助函数设计

为了便于后续的调用，我针对不同结构分别设计了相应的辅助函数，同时针对运行过程中的错误处理，我借鉴了 menci 的方式使用 setjmp 和 longjmp 进行错误处理，出错时直接跳转到 setjmp 的下一条语句，这样就避免了对许多特殊返回值进行特判的操作。具体的函数如下：

1. bitmap

D_disk_read：读 datablock bitmap，由于读它时需要连续读两个数据块，所以单独写成一个函数

D_disk_write：写 datablock bitmap，同样是连续写两个块

bitmap_get_bit：查询 bitmap 上某位的值

bitmap_set_bit：将某位的值设置为 1

bitmap_reset_bit：将某位的值设置为 0

inode_bitmap_cnt：查询 inode bitmap 上有多少位设置为 1

datablock_bitmap_cnt：查询 datablock bitmap 上有多少位设置为 1，注意这里应该忽略 bitmap 和 inode 数据块(0-258)，从第 259 个块开始算

```
void D_disk_read(block_item ptr){//read datablock bitmap
    disk_read(1, ptr);
    disk_read(2, ptr + BLOCK_SIZE);
}
void D_disk_write(block_item ptr){//write datablock bitmap
    disk_write(1, ptr);
    disk_write(2, ptr + BLOCK_SIZE);
}
bool bitmap_get_bit(bitmap_item* bp, int pos){return (bp[pos / 64] >> (pos % 64)) & 1;}
void bitmap_set_bit(bitmap_item* bp, int pos){bp[pos / 64] |= 1ULL << (pos % 64);}
void bitmap_reset_bit(bitmap_item* bp, int pos){bp[pos / 64] &= ~(1ULL << (pos % 64));}
int inode_bitmap_cnt(bitmap_item *bp){
    int res = 0;
    for(int i = 0; i < 32768; ++i)
        if(bitmap_get_bit(bp, i)) ++res;
    return res;
}
int datablock_bitmap_cnt(bitmap_item *bp){
    int res = 0;
    for(int i = 259; i <= 65535; ++i)
        if(bitmap_get_bit(bp, i)) ++res;
    return res;
}
```

2. block

new_block：分配一个空数据块

alloc_blocks：分配多个空数据块

del_block：删除一个数据块
del_blocks：删除多个数据块
find_enough_block：查找是否有 t 个可用的数据块
get_num_of_block：计算 size 对应的数据块数目

```
unsigned short new_block(){
    datablock_bitmap bp;
    D_disk_read((void *)bp);
    for(int i = 259; i <= 65535; ++i)
        if(!bitmap_get_bit(bp, i)){
            bitmap_set_bit(bp, i);
            D_disk_write((void *)bp);
            return i;
        }
    longjmp(myerror, -ENOSPC);
}
void alloc_blocks(int t, unsigned short res[t]){
    if(!t) return;
    int cnt = 0;
    datablock_bitmap bp;
    D_disk_read((void *)bp);
    for(int i = 259; i <= 65535; ++i)
        if(!bitmap_get_bit(bp, i)){
            res[cnt++] = i;
            if(cnt == t) break;
        }
    if(cnt < t) longjmp(myerror, -ENOSPC);
    for(int i = 0; i < t; ++i) bitmap_set_bit(bp, res[i]);
    D_disk_write((void *)bp);
}
void del_block(unsigned short block_num){
    datablock_bitmap bp;
    D_disk_read((void *)bp);
    if(!bitmap_get_bit(bp, block_num)) return;
    bitmap_reset_bit(bp, block_num);
    char empty[4096];
    memset(empty, 0, sizeof(empty));
    disk_write(block_num, empty);
    D_disk_write((void *)bp);
}
void del_blocks(unsigned short *block_num, int t){
    if(!t) return;
    datablock_bitmap bp;
    D_disk_read((void *)bp);
    char empty[4096];
    memset(empty, 0, sizeof(empty));
    for(int i = 0; i < t; ++i){
        if(!bitmap_get_bit(bp, block_num[i])) continue;
        bitmap_reset_bit(bp, block_num[i]);
        disk_write(block_num[i], empty);
    }
    D_disk_write((void *)bp);
}
bool find_enough_block(int t){
    if(!t) return 1;
    datablock_bitmap bp;
    D_disk_read((void *)bp);
    for(int i = 259; i <= 65535; ++i)
        if(!bitmap_get_bit(bp, i) && !(--t)) return true;
    return false;
}
int get_num_of_block(size_t size){return size / BLOCK_SIZE + (size % BLOCK_SIZE > 0);}
```

3. inode

get_inode：读 inode
set_inode：写 inode
new_inode：分配一个 inode
del_inode：删除一个 inode
find_enough_inode：查找是否有 t 个可用的 inode
init_inode：初始化 inode
destory_inode：销毁 inode

```
inode get_inode(unsigned short inode_num){
    unsigned short blk_num = 3 + inode_num / 128;//块编号
    size_t offset = (inode_num % 128) * 32;//块中偏移量
    block_item blk;
    disk_read(blk_num, blk);
    return *(inode *)(blk + offset);
}
void set_inode(inode cur, unsigned short inode_num){
    unsigned short blk_num = 3 + inode_num / 128;
    size_t offset = (inode_num % 128) * 32;
    block_item blk;
    disk_read(blk_num, blk);
    *(inode *)(blk + offset) = cur;
    disk_write(blk_num, blk);
}
unsigned short new_inode(){
    inode_bitmap bp;
    disk_read(0, bp);
    for(int i = 1; i < 32768; ++i)
        if(!bitmap_get_bit(bp, i)){
            bitmap_set_bit(bp, i);
            disk_write(0, bp);
            return i;
        }
    longjmp(myerror, -ENOSPC);
}
void del_inode(unsigned short inode_num){
    inode_bitmap bp;
    disk_read(0, bp);
    bitmap_reset_bit(bp, inode_num);
    disk_write(0, bp);
}
bool find_enough_inode(int t){
    if(!t) return 1;
    inode_bitmap bp;
    disk_read(0, bp);
    for(int i = 1; i < 32768; ++i)
        if(!bitmap_get_bit(bp, i) && !(--t))
            return 1;
    return 0;
}
void init_inode(unsigned short inode_num, bool directory){
    inode tmp;
    tmp.atime = tmp.ctime = tmp.mtime = time(NULL);
    tmp.size = 0;
    tmp.indirect_ptr = new_block();
    tmp.directory = directory;
    set_inode(tmp, inode_num);
}
void destory_inode(unsigned short inode_num){
    inode tmp = get_inode(inode_num);
    block_item blk;
    disk_read(tmp.indirect_ptr, blk);
    del_blocks((unsigned short *)blk, get_num_of_block(tmp.size));
    del_block(tmp.indirect_ptr);
    del_inode(inode_num);
}
```

4. directory

get_dir：读目录
set_dir：写目录
expand_dir：新增一个目录项时检查是否要在 indirect pointer 中新分配一个数据块
get_num_of_dir_item：计算目录项数目
find_dir：按文件/目录名查找其 inode 编号

```
void get_dir(inode dir_inode, dir_item *res) {
    block_item dir_blk;
    disk_read(dir_inode.indirect_ptr, dir_blk);
    unsigned short *blk_num = (unsigned short *)dir_blk;
    char *ptr = (char *)res;
    int num_of_blk = get_num_of_block(dir_inode.size);
    for (int i = 0; i < num_of_blk; ++i, ptr += BLOCK_SIZE)
        disk_read(blk_num[i], ptr);
}
void set_dir(inode dir_inode, dir_item *dir){
    block_item dir_blk;
    disk_read(dir_inode.indirect_ptr, dir_blk);
    unsigned short *blk_num = (unsigned short *) dir_blk;
    char *ptr = (char *) dir;
    int num_of_blk = get_num_of_block(dir_inode.size);
    for(int i = 0; i < num_of_blk; ++i, ptr += BLOCK_SIZE) disk_write(blk_num[i], ptr);
    int cnt = 0;
    for(int j = num_of_blk; j < 2048 &&  blk_num[j]; ++j) ++cnt;
    del_blocks(&blk_num[num_of_blk], cnt);
    disk_write(dir_inode.indirect_ptr, dir_blk);
}
void expand_dir(inode dir_inode){
    int num_of_block = get_num_of_block(dir_inode.size + 1);
    if(get_num_of_block(dir_inode.size) == num_of_block) return;
    block_item dir_blk;
    disk_read(dir_inode.indirect_ptr, dir_blk);
    unsigned short *blk_num = (unsigned short *) dir_blk;
    blk_num[num_of_block - 1] = new_block();
    disk_write(dir_inode.indirect_ptr, dir_blk);
}
int get_num_of_dir_item(inode dir_inode){return dir_inode.size / sizeof(dir_item);}
dir_item* find_dir(dir_item* dir, int cnt, const char* filename, int len){
    char name[24];
    memset(name, 0, 24);
    memcpy(name, filename, len);
    for(dir_item *i = dir; i != dir + cnt; ++i)
        if(!memcmp(i->filename, name, 24)) return i;
    return NULL;
}
```

5. path

path2inode：根据路径找到目标文件的 inode

path2pinode：根据路径解析出目标文件父目录的 inode

在实现这两个函数的时候，我一开始是使用两个指针对路径字符串的每一级进行解析，但在参考 menci 的代码后发现使用 strchrnul 函数能够更方便的处理，于是最终选

择参考了 menci 的方法，我自己的字符串模拟以注释保留：

```
inode path2inode(const char *path, unsigned short *inode_num){
    inode cur = get_inode(0);
    if(inode_num) *inode_num = 0;
    while(1){
        if(*path == '/') ++path;
        if(!*path) break;
        int len = strchrnul(path, '/') - path;
        if(!cur.directory) longjmp(myerror, -ENOTDIR);
        static dir_item *ptr, res[32768];
        get_dir(cur, res);
        ptr = find_dir(res, get_num_of_dir_item(cur), path, len);
        if(ptr){
            cur = get_inode(ptr->inode_num);
            if(inode_num) *inode_num = ptr->inode_num;
            path += len;
        }
        else longjmp(myerror, -ENOENT);
    }
    return cur;
}

/*
inode path2inode(const char *path, unsigned short *inode_num){
    inode cur = get_inode(0);
    if(inode_num) *inode_num = 0;
    int len = strlen(path), head = 0, tail = 0, k;
    char mypath[len + 10], filename[25];
    strncpy(mypath, path, len + 10);
    if(len == 1 && path[0] == '/') return cur;
    if(path[len - 1] == '/') --len;
    mypath[len] = '\0';
    for(int i = head + 1; i < len; ++i){
        if(mypath[i] != '/') continue;
        tail = i, k = 0;
        for(int j = head + 1; j < tail; ++j, ++k) filename[k] = mypath[j];
        filename[k++] = '\0';
        static dir_item *ptr, res[32768];
        get_dir(cur, res);
        ptr = find_dir(res, get_num_of_dir_item(cur), filename, k);
        if(ptr){
            cur = get_inode(ptr->inode_num);
            if(inode_num) *inode_num = ptr->inode_num;
        }
        else longjmp(myerror, -ENOENT);
        head = tail , i = head;
    }
    k = 0;
    for(int j = head + 1; j < len; ++j, ++k) filename[k] = mypath[j];
    filename[k++] = '\0';
    static dir_item *ptr, res[32768];
    get_dir(cur, res);
    ptr = find_dir(res, get_num_of_dir_item(cur), filename, k);
    if(ptr){
        cur = get_inode(ptr->inode_num);
        if(inode_num) *inode_num = ptr->inode_num;
    }
    else longjmp(myerror, -ENOENT);
    return cur;
}
*/

inode path2pinode(const char* path, unsigned short *res_num, char name[25], bool *use){
    inode cur = get_inode(0), res;
    unsigned short inode_num = 0;
    if(use) *use = true;
    bool no_use = false;
    while(1){
        if(*path == '/') ++path;
        if(!*path) break;
        if(no_use) longjmp(myerror, -ENOENT);
        int len = strchrnul(path, '/') - path;
        if(!cur.directory) longjmp(myerror, -ENOTDIR);
        static dir_item *ptr, tmp[32768];
        res = cur;
        get_dir(cur, tmp);
        ptr = find_dir(tmp, get_num_of_dir_item(cur), path, len);
        if(res_num) *res_num = inode_num;
        if(ptr){
            cur = get_inode(ptr->inode_num);
            if(res_num)inode_num = ptr->inode_num;
        }
        else no_use = true;
        memset(name, 0, 25);
        strncpy(name, path, len);
        name[len] = '\0';
        path += len;
    }
    return res;
}
```

三、 各函数实现

1. mkfs

由于不需要 superblock(有 superblock 时则还需初始化 superblock)，bitmap 的初值也为 0，在这个函数中只需初始化根目录的 inode 即可：

```
int mkfs(){
    init_inode(0, 1);
    return 0;
}
```

2. fs_getattr

用 path2inode 通过路径找到对应的 inode，读取其信息即可

```
int fs_getattr (const char *path, struct stat *attr){
    int error = setjmp(myerror);
    if(error) return error;
    inode tmp  = path2inode(path, NULL);
    attr->st_mode = tmp.directory ? DIRMODE : REGMODE;
    attr->st_nlink = 1;
    attr->st_uid = getuid();
    attr->st_gid = getgid();
    attr->st_size = tmp.size;
    attr->st_atime = tmp.atime;
    attr->st_mtime = tmp.mtime;
    attr->st_ctime = tmp.ctime;
    return 0;
}
```

3. fs_readattr

用 path2inode 通过路径找到对应的 inode，再用 get_dir 直接读取目录即可，注意此时应该更新目录文件的 access time：

```
int fs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset,
struct fuse_file_info *fi){
    int error = setjmp(myerror);
    if(error) return error;
    unsigned short inode_num;
    inode tmp = path2inode(path, &inode_num);
    if(!tmp.directory) return -ENOTDIR;
    static dir_item dir[32768];
    get_dir(tmp, dir);
    for(dir_item* it = dir; it != dir + get_num_of_dir_item(tmp); ++it){
        char name[25];
        memset(name, 0, sizeof(name));
        memcpy(name, it->filename, 24);
        filler(buffer, name, NULL, 0);
    }
    tmp.atime = time(NULL), set_inode(tmp, inode_num);
    return 0;
}
```

4. fs_read

首先还是用 path2inode 通过路径找到目标文件的 inode，然后根据偏移量和读取长度进行读取。这里我也使用了 menci 的一个枚举的技巧，使用一个 for 循环记录第几个块，块中偏移量，块中读取的长度，剩余长度以避免针对偏移量和长度的分类讨论。在之后的 fs_write 中也使用了该方法。

```
for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size = size;
        (blk_size = min(4096 - blk_offset, remain_size)) != 0;
        ++i, blk_offset = 0, remain_size -= blk_size);
```

将读取到的内容写入 buffer，并更新 inode 的 access time：

```
int fs_read(const char *path, char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi){
    int error = setjmp(myerror);
    if(error) return error;
    unsigned short inode_num;
    inode tmp = path2inode(path, &inode_num);
    if(tmp.directory) return -EISDIR;
    block_item ptr;
    disk_read(tmp.indirect_ptr, ptr);
    unsigned short *blk_num = (unsigned short*) ptr;
    for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size =
size; (blk_size = min(4096 - blk_offset, remain_size)) != 0; ++i, blk_offset = 0,
remain_size -= blk_size){
        block_item blk;
        disk_read(blk_num[i], blk);
        memcpy(buffer, &blk[blk_offset], blk_size);
        buffer += blk_size;
    }
    tmp.atime = time(NULL), set_inode(tmp, inode_num);
    return size;
}
```

5. fs_mknod/fs_mkdir

这两个函数本质上是一样的，共用一个 mk 实现即可。

首先应当检查是否有足够的空间，即 1 个 inode 和 2 个 datablock(indirect pointer 和

一个数据块)，若无则应当直接报错。随后使用 path2pinode 解析路径得到父目录的 inode，更新父目录的目录项，注意此处应使用 expand_dir 检查新增一个目录项时父目录是否需要新分配一个数据块，随后为这个新分配的目录项(即新文件)分配一个 inode 并初始化。最后更新父目录的 atime，mtime，ctime。

```
int mk(const char *path, bool directory){
    int error = setjmp(myerror);
    if(error) return error;
    if(!find_enough_inode(1) || !find_enough_block(2)) return -ENOSPC;
    char name[25];  bool use; unsigned short inode_num;
    inode fa = path2pinode(path, &inode_num, name, &use);
    int cnt = get_num_of_dir_item(fa);
    static dir_item ptr[32768];
    get_dir(fa, ptr);
    expand_dir(fa), fa.size += sizeof(dir_item);
    ptr[cnt].inode_num = new_inode();
    memcpy(ptr[cnt].filename, name, 24);
    init_inode(ptr[cnt].inode_num, directory);
    fa.atime = fa.mtime = fa.ctime = time(NULL);
    set_dir(fa, ptr), set_inode(fa, inode_num);
    return 0;
}
int fs_mknod (const char *path, mode_t mode, dev_t dev){
    return mk(path, 0);
}
int fs_mkdir (const char *path, mode_t mode){
    return mk(path, 1);
}
```

6. fs_unlink/fs_rmdir

类似于前两个函数，共用一个 rm 实现，先用 path2pinode 解析路径得到父目录的 inode，删除待删除文件的目录项即可。更新父目录的 ctime 和 mtime。

```
int rm(const char *path){
    int error = setjmp(myerror);
    if(error) return error;
    char name[25]; unsigned short inode_num;
    inode fa = path2pinode(path, &inode_num, name, NULL);
    static dir_item dir[32768];
    get_dir(fa, dir);
    dir_item* ptr = find_dir(dir, get_num_of_dir_item(fa), name, strlen(name));
    if(!ptr) return -ENOENT;
    destory_inode(ptr->inode_num);
    memcpy(ptr, ptr + 1, sizeof(dir_item) * ((dir + get_num_of_dir_item(fa)) - (ptr + 
1)));
    fa.ctime = fa.mtime = time(NULL), fa.size -= sizeof(dir_item);
    set_dir(fa, dir), set_inode(fa, inode_num);
    return 0;
}
int fs_rmdir (const char *path){
    return rm(path);
}
int fs_unlink (const char *path){
    return rm(path);
}
```

7. fs_rename

首先用 path2pinode 找到文件原来位置和目标位置的父目录 inode，查找该文件的目录项。若二者的目录项相同则表示该文件只需要在原位置的目录项中改名即可，否则在原位置的父目录中删去一个目录项，在新位置的父目录中新增一个目录项：

```
int fs_rename (const char *oldpath, const char *newpath){
    if(!strcmp(oldpath, newpath)) return 0;
    fs_unlink(newpath);
    int error = setjmp(myerror);
    if(error) return error;
    char from_name[25], to_name[25]; unsigned short from_num ,to_num;
    inode from = path2pinode(oldpath, &from_num, from_name, NULL), to =
path2pinode(newpath, &to_num, to_name, NULL);
    static dir_item from_dir[32768];
    get_dir(from, from_dir);
    dir_item *from_ptr = find_dir(from_dir, get_num_of_dir_item(from), from_name,
strlen(from_name));
    from.atime = from.mtime = time(NULL);
    if(!from_ptr) return -ENOENT;
    if(from_num == to_num) memcpy(from_ptr->filename, to_name, 24);
    else{
        static dir_item to_dir[32768];
        get_dir(to, to_dir);
        dir_item *to_ptr = find_dir(to_dir, get_num_of_dir_item(to), to_name,
strlen(to_name));
        if(to_ptr) return -EEXIST;
        expand_dir(to);
        int cnt = get_num_of_dir_item(to);
        memcpy(to_dir[cnt].filename, to_name, 24);
        to_dir[cnt].inode_num = from_ptr->inode_num;
        to.size += sizeof(dir_item), to.atime = to.mtime = time(NULL);
        set_dir(to, to_dir), set_inode(to, to_num);
        memcpy(from_ptr, from_ptr + 1, sizeof(dir_item) * ((from_dir +
get_num_of_dir_item(from)) - (from_ptr + 1)));
        from.size -= sizeof(dir_item);
    }
    set_dir(from, from_dir), set_inode(from, from_num);
    return 0;
}
```

8. fs_write

首先用 path2inode 解析路径得该文件 inode，如果是目录则报错。然后用和 fs_read 中相同的枚举方法得到需要新分配的块数，然后逐块写即可。最后修改 inode 的 mtime 和 ctime

```
int fs_write (const char *path, const char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi){
    int error = setjmp(myerror);
    if(error) return 0;
    unsigned short inode_num;
    inode tmp = path2inode(path, &inode_num);
    if(tmp.directory) return -EISDIR;
    block_item ptr;
    disk_read(tmp.indirect_ptr, ptr);
    unsigned short * blk_num = (unsigned short*) ptr;
    int cnt = 0;
    for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size =
size; (blk_size = min(4096 - blk_offset, remain_size)) != 0; ++i, blk_offset = 0,
remain_size -= blk_size)
        if(!blk_num[i]) ++cnt;
    unsigned short new_blk[cnt];
    alloc_blocks(cnt, new_blk);
    for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size =
size; (blk_size = min(4096 - blk_offset, remain_size)) != 0; ++i, blk_offset = 0,
remain_size -= blk_size){
        if(!blk_num[i]) blk_num[i] = new_blk[--cnt];
        block_item blk;
        disk_read(blk_num[i], blk);
        memcpy(&blk[blk_offset], buffer, blk_size);
        buffer += blk_size;
        disk_write(blk_num[i], blk);
    }
    disk_write(tmp.indirect_ptr, ptr);
    tmp.mtime = tmp.ctime = time(NULL), tmp.size = max(tmp.size, size + offset);
    set_inode(tmp, inode_num);
    return size;
}
```

9. fs_truncate

首先用 path2inode 解析路径得该文件 inode，然后计算修改前后所需的数据块数目进行分类讨论。如果是块数变多，则新分配相应数量的块；否则则删除相应数量的块，注意要讨论大小不是 4kb 的整数倍时最后一个块的情况。

```
int fs_write (const char *path, const char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi){
    int error = setjmp(myerror);
    if(error) return error;
    unsigned short inode_num;
    inode tmp = path2inode(path, &inode_num);
    if(tmp.directory) return -EISDIR;
    block_item ptr;
    disk_read(tmp.indirect_ptr, ptr);
    unsigned short * blk_num = (unsigned short*) ptr;
    int cnt = 0;
    for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size =
size; (blk_size = min(4096 - blk_offset, remain_size)) != 0; ++i, blk_offset = 0,
remain_size -= blk_size)
        if(!blk_num[i]) ++cnt;
    unsigned short new_blk[cnt];
    alloc_blocks(cnt, new_blk);
    for(int i = offset / 4096, blk_offset = (offset & 0xFFF), blk_size, remain_size =
size; (blk_size = min(4096 - blk_offset, remain_size)) != 0; ++i, blk_offset = 0,
remain_size -= blk_size){
        if(!blk_num[i]) blk_num[i] = new_blk[--cnt];
        block_item blk;
        disk_read(blk_num[i], blk);
        memcpy(&blk[blk_offset], buffer, blk_size);
        buffer += blk_size;
        disk_write(blk_num[i], blk);
    }
    disk_write(tmp.indirect_ptr, ptr);
    tmp.mtime = tmp.ctime = time(NULL), tmp.size = max(tmp.size, size + offset);
    set_inode(tmp, inode_num);
    return size;
}
```

10. fs_utime/fs_statfs

直接解析路径、修改或记录相应内容即可。

```
int fs_utime (const char *path, struct utimbuf *buffer){
    int error = setjmp(myerror);
    if(error) return error;
    unsigned short inode_num;
```

```c
    inode tmp  = path2inode(path, &inode_num);
    tmp.atime = buffer->actime;
    tmp.mtime = buffer->modtime;
    tmp.ctime = time(NULL);
    set_inode(tmp, inode_num);
    return 0;
}

int fs_statfs (const char *path, struct statvfs *stat){
    inode_bitmap i;
    datablock_bitmap d;
    disk_read(0, i);
    D_disk_read((void *)d);
    stat->f_bsize = BLOCK_SIZE;
    stat->f_blocks = 65277;
    stat->f_bavail = stat->f_bfree = 65277 - datablock_bitmap_cnt(d);
    stat->f_files = 32768;
    stat->f_ffree = stat->f_favail = 32767 - inode_bitmap_cnt(i);
    stat->f_namemax = 24;
    return 0;
}
```

四、 一些思考

1. 关于块的擦除

在之前的 del_block 和 del_blocks 函数中，我实际上清空了数据块(写为全 0)，但实际上在删除尾部的连续块时直接修改 size 和 bitmap 以限制对删除块的访问即可，可以省去一些空间清理的时间。

2. 关于空间局部性

在本文件系统的实现中，数据块的分配是比较 naive 地优先选择编号小的块，这样并不能很好的利用数据的局部性。可以考虑在组织 bitmap 的时候使用平衡树、B 树等结构分配连续的块，以提高空间局部性，也便于后续 Cache 的缓存。