Project 6 File System 设计文档

中国科学院大学
陈灿宇
2019.1.24

1　文件系统初始化设计

(1) 请阐述你设计的文件系统对磁盘的布局（可以使用图例表示），包括从磁盘哪个位置开始，superblock，inode map，block/sector map，inode table 以及数据区各自占用的磁盘空间大小

```
/*
* SD card file system for OS seminar
* This filesystem looks like this:
* FS size : 1GB
* 1 Block : 4KB, total Blocks : 256K
* 1 Inode : 128B
* Inode Bitmap size : 8K / 8 * 1B= 1KB
* Block Bitmap size : 256K / 8 * 1B= 32KB
* FS_START_SD_OFFSET = 0x20000000, //512MB
* FS_MAGIC_NUMBER = 0x2e575159,
* -------------------------------------------------------------------------------
* | Superblock    | Block Bitmap   | Inode Bitmap   | Inode Table     |  Blocks     |
* | 1 Block 4KB   | 8 Blocks 32KB  | 1 Block   4KB  | 256 Blocks 1MB  |  Others     |
* -------------------------------------------------------------------------------
*/
```

(2) 请列出你设计的 superblock 和 inode 数据结构，并阐明各项含义。请说明你设计的文件系统能支持的最大文件大小，最多文件数目，以及单个目录下能支持的最多文件/子目录数目。

- superblock

```
typedef struct superblock {
    uint32_t s_disk_size;                //磁盘总容量
    uint32_t s_block_size;               //block大小
    uint32_t s_magic;                    //魔数(0x2e575159)

    uint32_t s_total_inodes_cnt;         //磁盘总 inode 数
    uint32_t s_total_blocks_cnt;         //磁盘总 block 数
    uint32_t s_free_inode_cnt;           //磁盘空闲 inode 数
    uint32_t s_free_blocks_cnt;          //磁盘空闲 block 数

    uint32_t s_blockbmp_block_index;     //Block Bitmap起始 block
    uint32_t s_inodebmp_block_index;     //Inode Bitmap起始 block
    uint32_t s_inodetable_block_index;   //Inode Table起始 block
    uint32_t s_data_block_index;         //data Blocks起始 block

```

```
        uint32_t s_checknum;                //校验和
        uint32_t s_inode_size;              //inode 大小
        uint32_t s_dentry_size;             //dentry大小

        uint32_t padding[114];
    } superblock_t; //size: 128*sizeof(int) -> 512Byte
```

- inode

```
    typedef struct inode {
        uint16_t i_fmode;                     //文件类型和权限信息
        uint16_t i_links_cnt;                 //硬链接数量

        uint32_t i_fsize;                     //文件大小
        uint32_t i_fnum;                      //目录内文件数(不含.和..)

        uint32_t i_atime;                     //最后访问时间
        uint32_t i_ctime;                     //元数据最后修改时间
        uint32_t i_mtime;                     //文件最后修改时间

        uint32_t i_direct_table[MAX_DIRECT_NUM]; //直接block指针
        uint32_t i_indirect_block_1_ptr;      //1级指针block指针
        uint32_t i_indirect_block_2_ptr;      //2级指针block指针
        uint32_t i_indirect_block_3_ptr;      //3级指针block指针

        uint32_t i_num;                       //inode number

        uint32_t padding[10];
    } inode_t;  //size: 32*sizeof(int) -> 128Byte
```

- 支持信息

  block map 采用 12 个直接指针、1 个 1 级指针、1 个 2 级指针和 1 个 3 级指针, 支持的单个最大文件大小为 4402345721856 B(约 4100 GB); 最多支持 65536 个文件 (包含目录); 单个目录下能支持的最多文件 (包含目录) 为 17196662976 个。

- 块分配策略

  将块的分配情况以 bitmap 的形式存于内存中, 同时同步到持久化介质中, 查找空闲数据块时从头开始搜索 (后续版本可以随机化) ; 新建目录时分配 1 个空闲数据块 (存放. 和..) , 新建文件时不分配数据块, 实际写入时按需求分配数据块。

(3) 设计或实现过程中遇到的问题和得到的经验 (如果有的话可以写下来, 不是必需项)

2　文件操作设计

（1）请说明创建一个文件所涉及的元数据新增和修改操作，例如需要新增哪些元数据，需要修改哪些元数据

需要找到一个空的 inode 存放文件的相关信息，需要添加的信息包括文件类型和权限信息、硬链接数量、文件大小、目录内文件数 (不含. 和..) 、最后访问时间、元数据最后修改时间、文件最后修改时间、直接 block 指针、1 级指针 block 指针、2 级指针 block 指针、3 级指针 block 指针、inode number。

```
int do_touch(char *name, mode_t mode)
{
...
    free_inum = find_free_inode();
    set_inode_bmp(free_inum);
    sync_to_disk_inode_bmp();

    new_inode.i_fmode = S_IFREG | mode;
    new_inode.i_links_cnt = 1;
    new_inode.i_fsize = 0;
    new_inode.i_fnum = 0;
    new_inode.i_atime = get_ticks();
    new_inode.i_ctime = get_ticks();
    new_inode.i_mtime = get_ticks();
    bzero(new_inode.i_direct_table, MAX_DIRECT_NUM*sizeof(uint32_t));
    new_inode.i_indirect_block_1_ptr = NULL;
    new_inode.i_indirect_block_2_ptr = NULL;
    new_inode.i_indirect_block_3_ptr = NULL;
    new_inode.i_num = free_inum;
    bzero(new_inode.padding, 10*sizeof(uint32_t));
    sync_to_disk_inode(&new_inode);
...
}
```

然后需要将该文件的文件名和 inum 写入该文件的父目录, 同时修改父目录的 inode 的文件大小 i_fsize，目录内文件数 (不含. 和..)i_fnum 信息。

```
static int write_dentry(inode_t* inode_ptr, uint32_t dnum, dentry_t* dentry_ptr)
{
...
    if(get_block_index_in_dir(inode_ptr, major_index) == 0){
        uint32_t free_block_index = find_free_block();
        set_block_bmp(free_block_index);
        sync_to_disk_block_bmp();
        superblock_ptr->s_free_blocks_cnt--;
        sync_to_disk_superblock();

        write_block_index_in_dir(inode_ptr, major_index, free_block_index);
        inode_ptr->i_fsize += BLOCK_SIZE;
        inode_ptr->i_fnum++;
        sync_to_disk_inode(inode_ptr);
    }
    else{
        inode_ptr->i_fnum++;
        sync_to_disk_inode(inode_ptr);
        read_block(get_block_index_in_dir(inode_ptr, major_index), dentry_block_buffer);
    }
    memcpy((uint8_t *)(&(dentry_table[minor_index])), (uint8_t *)dentry_ptr, DENTRY_SIZE)
        ;
    write_block(get_block_index_in_dir(inode_ptr, major_index), dentry_block_buffer);
...
}
```

还要修改 superblock 中的磁盘空闲 inode 数 s_free_inode_cnt, inode bitmap 的 inode 分配信息。

（2）如果完成了 bonus，请说明硬链接、软链接和 rename 涉及的操作流程

- 硬链接

  do_link() 实现了硬链接操作。其流程如下: 首先, 检查待创建的链接来源是否为目录, 如果为目录, 则报异常, 因为硬链接的源不能是一个目录; 否则, 在目标链接的父目录当中为其分配 dentry , 并将 src_inode 中的链接数加一。

```
void do_link(char *src_path, char *new_path)
{
    bzero(parent_buffer, MAX_PATH_LENGTH);
    bzero(path_buffer, MAX_PATH_LENGTH);
    bzero(name_buffer, MAX_NAME_LENGTH);

    uint32_t src_inum = parse_path(src_path, current_dir_ptr);
    inode_t src_inode;
    sync_from_disk_inode(src_inum, &src_inode);

    if(S_ISDIR(src_inode.i_fmode)){
        vt100_move_cursor(1, 45);
        printk("[FS ERROR] ERROR_LINK_CANNOT_BE_DIR\n");
        return ;
    }

    strcpy(path_buffer, new_path);
    separate_path(path_buffer, parent_buffer, name_buffer);
    uint32_t parent_inum = parse_path(parent_buffer, current_dir_ptr);

    inode_t parent_inode;
    sync_from_disk_inode(parent_inum, &parent_inode);
    src_inode.i_links_cnt++;
    sync_to_disk_inode(&src_inode);

    dentry_t parent_den;
    parent_den.d_inum = src_inum;
    strcpy(parent_den.d_name, name_buffer);
    write_dentry(&parent_inode, parent_inode.i_fnum+2, &parent_den);

    return;
}
```

- 软链接

  do_symlink() 实现了符号链接操作。创建 (make) 系列的操作基本都是比较相似的 (包括 mkdir 在内) 。首先, 检查文件是否已存在; 然后，分配目录项和 i-node 并更新文件系统统计信息。符号链接的关键是创建一个特殊的 S_IFLNK 类型的文件, 除了以上类似的操作外，还要在分配的直接块中应填入指向目标的完整路径。

```
void do_symlink(char *src_path, char *new_path)
{
    bzero(parent_buffer, MAX_PATH_LENGTH);
    bzero(path_buffer, MAX_PATH_LENGTH);
    bzero(name_buffer, MAX_NAME_LENGTH);
    bzero(data_block_buffer, BLOCK_SIZE);

    char *_p = ".";
    strcpy(path_buffer, _p);
    strcpy(path_buffer+1, src_path);
    separate_path(path_buffer, parent_buffer, name_buffer);

    uint32_t parent_inum = 0, free_inum, free_block_index;
    parent_inum = parse_path(new_path, current_dir_ptr);

    sync_from_disk_block_bmp();
    sync_from_disk_inode_bmp();

    inode_t parent_inode, new_inode;
    sync_from_disk_inode(parent_inum, &parent_inode);

    free_inum = find_free_inode();
    set_inode_bmp(free_inum);
    sync_to_disk_inode_bmp();

    superblock_ptr->s_free_inode_cnt--;
    sync_to_disk_superblock();
    free_block_index = find_free_block();
    set_block_bmp(free_block_index);
    sync_to_disk_block_bmp();
    superblock_ptr->s_free_blocks_cnt--;
    sync_to_disk_superblock();

    new_inode.i_fmode = S_IFLNK;
    new_inode.i_links_cnt = 1;
    new_inode.i_fsize = BLOCK_SIZE;
    new_inode.i_fnum = 0;
    new_inode.i_atime = get_ticks();
    new_inode.i_ctime = get_ticks();
    new_inode.i_mtime = get_ticks();
    bzero(new_inode.i_direct_table, MAX_DIRECT_NUM*sizeof(uint32_t));
    new_inode.i_direct_table[0] = free_block_index;
    new_inode.i_indirect_block_1_ptr = NULL;
    new_inode.i_indirect_block_2_ptr = NULL;
    new_inode.i_indirect_block_3_ptr = NULL;
    new_inode.i_num = free_inum;
    bzero(new_inode.padding, 10*sizeof(uint32_t));
    sync_to_disk_inode(&new_inode);

    bzero(dentry_block_buffer, BLOCK_SIZE);
    dentry_t *new_dentry_table = (dentry_t *)dentry_block_buffer;
    new_dentry_table[0].d_inum = free_inum;
    strcpy(new_dentry_table[0].d_name, ".");
    new_dentry_table[1].d_inum = parent_inum;
    strcpy(new_dentry_table[1].d_name, "..");
    sync_to_disk_dentry(free_block_index);

    sync_from_disk_file_data(free_block_index);
    memcpy(data_block_buffer + sizeof(dentry_t)*2, (uint8_t *)src_path, strlen(
        src_path));
    data_block_buffer[sizeof(dentry_t)*2 + strlen(src_path)] = '\0';
    sync_to_disk_file_data(free_block_index);

    dentry_t parent_dentry;
    parent_dentry.d_inum = free_inum;
    strcpy(parent_dentry.d_name, name_buffer);
    write_dentry(&parent_inode, parent_inode.i_fnum+2, &parent_dentry);
    return;
}
```

- rename

  do_rename() 实现了重命名操作。重命名操作流程如下: 首先, 确认新文件名不存在相应的文件 (文件或目录存在), 且新路径不是旧路径的子串 (非法操作) ; 然后遍历父目录查找其 dentry, 并修改文件名。

- find

  do_find() 实现了查询操作。查询操作比较简单, 首先通过类似于 cd 的操作进入到指定目录下, 然后检查当前目录下是否有指定文件或目录。

3 目录操作设计

（1）请说明文件系统执行 ls 命令查看一个绝对路径时的操作流程

绝对路径的解析函数见下，这是我设计的一个比较精巧的函数，从指定的 inode（一般就是 current_inode）开始往下查询，返回以‘/’作为分隔符最下层的一个文件名的 inum。在解析完路径之后，就可以通过 inum 找到对应的 inode，然后再通过这个 inode 查询该目录下的所有文件（包括目录），保存到 ls_buffer 中，从 shell 中输出。

```c
uint32_t parse_path(const char *path, inode_t *inode_ptr)
{
    bzero(parse_file_buffer, MAX_PATH_LENGTH);

    char *p_ = "/";
    strcpy(parse_file_buffer, (char *)path);
    strcpy(parse_file_buffer+strlen(parse_file_buffer), p_);

    int i = 0;
    char *_p = &parse_file_buffer[0];

    inode_t _inode;
    uint32_t inum;
    memcpy((uint8_t *)&_inode, (uint8_t *)inode_ptr, INODE_SIZE);

    uint32_t l = strlen(parse_file_buffer);

    for(; i < l; i++){
        if(parse_file_buffer[i] == '/'){
            parse_file_buffer[i] = '\0';

            inum = find_file(&_inode, _p);
            sync_from_disk_inode(inum, &_inode);

            _p = &parse_file_buffer[i+1];
        }
    }
    return inum;
}
```

（2）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

4 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

因为我在最初的设计中就考虑到了文件系统对于单个大文件的支持，所以我的设计中采用了 12 个直接指针、1 个 1 级指针、1 个 2 级指针和 1 个 3 级指针进行索引，能支持的单个最大文件大小为 4402345721856 B(约 4100 GB)，而将找到的空闲的 block 加入到 inode 的索引中和将 block 从 inode 中释放的过程其实是比较复杂的，关键的两个函数见下，write_block_index_in_inode() 是将找到的空闲 block 写入到 inode 的 idx 索引位置。release_inode_block() 是将 inode 的所有索引的 block 释放掉。

```c
void write_block_index_in_inode(inode_t *inode_ptr, uint32_t idx, uint32_t block_index)
{
    bzero(buffer1, POINTER_PER_BLOCK*sizeof(uint32_t));
    bzero(buffer2, POINTER_PER_BLOCK*sizeof(uint32_t));
    bzero(buffer3, POINTER_PER_BLOCK*sizeof(uint32_t));

    if(idx < FIRST_POINTER){
        inode_ptr->i_direct_table[idx] = block_index;
        sync_to_disk_inode(inode_ptr);
        return;
    }

    uint32_t free_index_1;
    if(idx < SECOND_POINTER){
        if(inode_ptr->i_indirect_block_1_ptr == 0){
            free_index_1 = find_free_block();

            set_block_bmp(free_index_1);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_1);
            inode_ptr->i_indirect_block_1_ptr = free_index_1;
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);
        }
        read_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
        buffer1[idx - FIRST_POINTER] = block_index;
        write_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
        return;
    }

    uint32_t free_index_2;
    if(idx < THIRD_POINTER){
        if(inode_ptr->i_indirect_block_2_ptr == 0){
            free_index_1 = find_free_block();

            set_block_bmp(free_index_1);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_1);
            inode_ptr->i_indirect_block_2_ptr = free_index_1;
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);
        }
        read_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
        if(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK] == 0){
            free_index_2 = find_free_block();

            set_block_bmp(free_index_2);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_2);
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);

            buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK] = free_index_2;
            write_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
        }
        read_block(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK], (uint8_t *)buffer2);
        buffer2[(idx - SECOND_POINTER) % POINTER_PER_BLOCK] = block_index;
        write_block(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK], (uint8_t *)buffer2);
    }

    uint32_t free_index_3;
    if(idx < MAX_BLOCK_INDEX){
        if(inode_ptr->i_indirect_block_3_ptr == 0){
            free_index_1 = find_free_block();

            set_block_bmp(free_index_1);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_1);
            inode_ptr->i_indirect_block_3_ptr = free_index_1;
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);
        }
        read_block(inode_ptr->i_indirect_block_3_ptr, (uint8_t *)buffer1);
        if(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK)] == 0)
            {
            free_index_2 = find_free_block();

            set_block_bmp(free_index_2);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_2);
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);

            buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK)] =
                free_index_2;
            write_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
        }
        read_block(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK
            )], (uint8_t *)buffer2);
        if(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK * POINTER_PER_BLOCK)) /
            POINTER_PER_BLOCK] == 0){
            free_index_3 = find_free_block();

            set_block_bmp(free_index_3);
            sync_to_disk_block_bmp();

            superblock_ptr->s_free_blocks_cnt--;
            sync_to_disk_superblock();

            clear_block_index(free_index_3);
            inode_ptr->i_fsize += BLOCK_SIZE;
            sync_to_disk_inode(inode_ptr);

            buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK * POINTER_PER_BLOCK)) /
                POINTER_PER_BLOCK] = free_index_3;
            write_block(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK *
                POINTER_PER_BLOCK)], (uint8_t *)buffer2);
        }
        read_block(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK *
            POINTER_PER_BLOCK)) / POINTER_PER_BLOCK], (uint8_t *)buffer3);
        buffer3[(idx - THIRD_POINTER) % POINTER_PER_BLOCK] = block_index;
        write_block(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK *
            POINTER_PER_BLOCK)) / POINTER_PER_BLOCK], (uint8_t *)buffer3);
        return;
```

```c
    }
    return;
}
```

```c
void release_inode_block(inode_t *inode_ptr)
{
    uint32_t i, j, k;
    bzero(buffer1, POINTER_PER_BLOCK*sizeof(uint32_t));
    bzero(buffer2, POINTER_PER_BLOCK*sizeof(uint32_t));
    bzero(buffer3, POINTER_PER_BLOCK*sizeof(uint32_t));

    for(i = 0; i < FIRST_POINTER; i++){
        if(inode_ptr->i_direct_table[i] == 0){
            return;
        }
        unset_block_bmp(inode_ptr->i_direct_table[i]);
    }

    if(inode_ptr->i_indirect_block_1_ptr == 0){
        return;
    }
    read_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
    for(i = 0; i < POINTER_PER_BLOCK; i++){
        if(buffer1[i] == 0){
            unset_block_bmp(inode_ptr->i_indirect_block_1_ptr);
            return;
        }
        unset_block_bmp(buffer1[i]);
    }
    unset_block_bmp(inode_ptr->i_indirect_block_1_ptr);

    if(inode_ptr->i_indirect_block_2_ptr == 0){
        return;
    }
    read_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
    for(i = 0; i < POINTER_PER_BLOCK; i++){
        if(buffer1[i] == 0){
            unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);
            return;
        }
        read_block(buffer1[i], (uint8_t *)buffer2);
        for(j = 0; j < POINTER_PER_BLOCK; j++){
            if(buffer2[j] == 0){
                unset_block_bmp(buffer1[i]);
                unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);
                return;
            }
            unset_block_bmp(buffer2[j]);
        }
        unset_block_bmp(buffer1[i]);
    }
    unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);

    if(inode_ptr->i_indirect_block_3_ptr == 0){
        return;
    }
    read_block(inode_ptr->i_indirect_block_3_ptr, (uint8_t *)buffer1);
    for(i = 0; i < POINTER_PER_BLOCK; i++){
        if(buffer1[i] == 0){
            unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
            return;
        }
        read_block(buffer1[i], (uint8_t *)buffer2);
        for(j = 0; j < POINTER_PER_BLOCK; j++){
            if(buffer2[j] == 0){
                unset_block_bmp(buffer1[i]);
                unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
                return;
            }
            read_block(buffer2[j], (uint8_t *)buffer3);
            for(k = 0; k < POINTER_PER_BLOCK; k++){
                if(buffer3[k] == 0){
                    unset_block_bmp(buffer2[j]);
                    unset_block_bmp(buffer1[i]);
                    unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
                    return;
                }
                unset_block_bmp(buffer3[k]);
            }
            unset_block_bmp(buffer2[j]);
        }
        unset_block_bmp(buffer1[i]);
    }
    unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
    return;
}
```