

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

```

1      /*
2      * SD card file system for OS seminar
3      * This filesystem looks like this:
4      * FS size : 1GB
5      * 1 Block : 4KB, total Blocks : 256K
6      * 1 Inode : 128B
7      * Inode Bitmap size : 8K / 8 * 1B= 1KB
8      * Block Bitmap size : 256K / 8 * 1B= 32KB
9      * FS_START_SD_OFFSET = 0x20000000, //512MB
10     * FS_MAGIC_NUMBER = 0x2e575159,
11     * -----
12     * | Superblock   | Block Bitmap | Inode Bitmap | Inode Table   | Blocks   |
13     * | 1 Block 4KB | 8 Blocks 32KB | 1 Block 4KB  | 256 Blocks 1MB | Others   |
14     * -----
15     */

```

(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

```

1      typedef struct superblock {
2          uint32_t s_disk_size;           // 磁盘总容量
3          uint32_t s_block_size;         // block大小
4          uint32_t s_magic;               // 魔数(0x2e575159)
5
6          uint32_t s_total_inodes_cnt;    // 磁盘总 inode 数
7          uint32_t s_total_blocks_cnt;    // 磁盘总 block 数
8          uint32_t s_free_inode_cnt;      // 磁盘空闲 inode 数
9          uint32_t s_free_blocks_cnt;     // 磁盘空闲 block 数
10
11         uint32_t s_blockbmp_block_index; // Block Bitmap起始 block
12         uint32_t s_inodebmp_block_index; // Inode Bitmap起始 block
13         uint32_t s_inodetable_block_index; // Inode Table起始 block
14         uint32_t s_data_block_index;     // data Blocks起始 block
15     }

```

```

16      uint32_t s_checknum;           //校验和
17      uint32_t s_inode_size;         //inode 大小
18      uint32_t s_dentry_size;        //dentry大小
19
20      uint32_t padding[114];
21  } superblock_t; //size: 128*sizeof(int) -> 512Byte

```

- inode

```

1      typedef struct inode {
2          uint16_t i_fmode;           //文件类型和权限信息
3          uint16_t i_links_cnt;       //硬链接数量
4
5          uint32_t i_fsize;           //文件大小
6          uint32_t i_fnum;           //目录内文件数(不含.和..)
7
8          uint32_t i_atime;           //最后访问时间
9          uint32_t i_ctime;           //元数据最后修改时间
10         uint32_t i_mtime;           //文件最后修改时间
11
12         uint32_t i_direct_table[MAX_DIRECT_NUM]; //直接block指针
13         uint32_t i_indirect_block_1_ptr;         //1级指针block指针
14         uint32_t i_indirect_block_2_ptr;         //2级指针block指针
15         uint32_t i_indirect_block_3_ptr;         //3级指针block指针
16
17         uint32_t i_num;                         //inode number
18
19         uint32_t padding[10];
20     } 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。

```

1  int do_touch(char *name, mode_t mode)
2  {
3  ...
4      free_inum = find_free_inode();
5      set_inode_bmp(free_inum);
6      sync_to_disk_inode_bmp();
7
8      new_inode.i_fmode = S_IFREG | mode;
9      new_inode.i_links_cnt = 1;
10     new_inode.i_fsize = 0;
11     new_inode.i_fnum = 0;
12     new_inode.i_atime = get_ticks();
13     new_inode.i_ctime = get_ticks();
14     new_inode.i_mtime = get_ticks();
15     bzero(new_inode.i_direct_table, MAX_DIRECT_NUM*sizeof(uint32_t));
16     new_inode.i_indirect_block_1_ptr = NULL;
17     new_inode.i_indirect_block_2_ptr = NULL;
18     new_inode.i_indirect_block_3_ptr = NULL;
19     new_inode.i_num = free_inum;
20     bzero(new_inode.padding, 10*sizeof(uint32_t));
21     sync_to_disk_inode(&new_inode);
22 ...
23 }
```

然后将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。

```

1  static int write_dentry(inode_t* inode_ptr, uint32_t dnum, dentry_t* dentry_ptr)
2  {
3  ...
4      if(get_block_index_in_dir(inode_ptr, major_index) == 0){
5          uint32_t free_block_index = find_free_block();
6          set_block_bmp(free_block_index);
7          sync_to_disk_block_bmp();
8          superblock_ptr->s_free_blocks_cnt--;
9          sync_to_disk_superblock();
10
11         write_block_index_in_dir(inode_ptr, major_index, free_block_index);
12         inode_ptr->i_fsize += BLOCK_SIZE;
13         inode_ptr->i_fnum++;

```

```

14     sync_to_disk_inode(inode_ptr);
15 }
16 else{
17     inode_ptr->i_fnum++;
18     sync_to_disk_inode(inode_ptr);
19     read_block(get_block_index_in_dir(inode_ptr, major_index), dentry_block_buffer);
20 }
21 memcpy((uint8_t *)&(dentry_table[minor_index]), (uint8_t *)dentry_ptr, DENTRY_SIZE)
    ;
22 write_block(get_block_index_in_dir(inode_ptr, major_index), dentry_block_buffer);
23 ...
24 }

```

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。

```

1 void do_link(char *src_path, char *new_path)
2 {
3     bzero(parent_buffer, MAX_PATH_LENGTH);
4     bzero(path_buffer, MAX_PATH_LENGTH);
5     bzero(name_buffer, MAX_NAME_LENGTH);
6
7     uint32_t src_inum = parse_path(src_path, current_dir_ptr);
8     inode_t src_inode;
9     sync_from_disk_inode(src_inum, &src_inode);
10
11     if(S_ISDIR(src_inode.i_fmode)){
12         vt100_move_cursor(1, 45);
13         printk("[FS ERROR] ERROR_LINK_CANNOT_BE_DIR\n");
14         return ;
15     }
16
17     strcpy(path_buffer, new_path);
18     separate_path(path_buffer, parent_buffer, name_buffer);
19     uint32_t parent_inum = parse_path(parent_buffer, current_dir_ptr);
20
21     inode_t parent_inode;
22     sync_from_disk_inode(parent_inum, &parent_inode);
23     src_inode.i_links_cnt++;
24     sync_to_disk_inode(&src_inode);
25

```

```

26     dentry_t parent_den;
27     parent_den.d_inum = src_inum;
28     strcpy(parent_den.d_name, name_buffer);
29     write_dentry(&parent_inode, parent_inode.i_fnum+2, &parent_den);
30
31     return;
32 }

```

- 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。

```

1 void do_symlink(char *src_path, char *new_path)
2 {
3     bzero(parent_buffer, MAX_PATH_LENGTH);
4     bzero(path_buffer, MAX_PATH_LENGTH);
5     bzero(name_buffer, MAX_NAME_LENGTH);
6     bzero(data_block_buffer, BLOCK_SIZE);
7
8     char *_p = ".";
9     strcpy(path_buffer, _p);
10    strcpy(path_buffer+1, src_path);
11    separate_path(path_buffer, parent_buffer, name_buffer);
12
13    uint32_t parent_inum = 0, free_inum, free_block_index;
14    parent_inum = parse_path(new_path, current_dir_ptr);
15
16    sync_from_disk_block_bmp();
17    sync_from_disk_inode_bmp();
18
19    inode_t parent_inode, new_inode;
20    sync_from_disk_inode(parent_inum, &parent_inode);
21
22    free_inum = find_free_inode();
23    set_inode_bmp(free_inum);
24    sync_to_disk_inode_bmp();
25
26    superblock_ptr->s_free_inode_cnt--;
27    sync_to_disk_superblock();
28    free_block_index = find_free_block();
29    set_block_bmp(free_block_index);
30    sync_to_disk_block_bmp();
31    superblock_ptr->s_free_blocks_cnt--;
32    sync_to_disk_superblock();
33
34    new_inode.i_fmode = S_IFLNK;

```

```

35     new_inode.i_links_cnt = 1;
36     new_inode.i_fsize = BLOCK_SIZE;
37     new_inode.i_fnum = 0;
38     new_inode.i_atime = get_ticks();
39     new_inode.i_ctime = get_ticks();
40     new_inode.i_mtime = get_ticks();
41     bzero(new_inode.i_direct_table, MAX_DIRECT_NUM*sizeof(uint32_t));
42     new_inode.i_direct_table[0] = free_block_index;
43     new_inode.i_indirect_block_1_ptr = NULL;
44     new_inode.i_indirect_block_2_ptr = NULL;
45     new_inode.i_indirect_block_3_ptr = NULL;
46     new_inode.i_num = free_inum;
47     bzero(new_inode.padding, 10*sizeof(uint32_t));
48     sync_to_disk_inode(&new_inode);
49
50     bzero(dentry_block_buffer, BLOCK_SIZE);
51     dentry_t *new_dentry_table = (dentry_t *)dentry_block_buffer;
52     new_dentry_table[0].d_inum = free_inum;
53     strcpy(new_dentry_table[0].d_name, ".");
54     new_dentry_table[1].d_inum = parent_inum;
55     strcpy(new_dentry_table[1].d_name, "..");
56     sync_to_disk_dentry(free_block_index);
57
58     sync_from_disk_file_data(free_block_index);
59     memcpy(data_block_buffer + sizeof(dentry_t)*2, (uint8_t *)src_path, strlen(
        src_path));
60     data_block_buffer[sizeof(dentry_t)*2 + strlen(src_path)] = '\0';
61     sync_to_disk_file_data(free_block_index);
62
63     dentry_t parent_dentry;
64     parent_dentry.d_inum = free_inum;
65     strcpy(parent_dentry.d_name, name_buffer);
66     write_dentry(&parent_inode, parent_inode.i_fnum+2, &parent_dentry);
67     return;
68 }

```

- 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。

```
1 uint32_t parse_path(const char *path, inode_t *inode_ptr)
2 {
3     bzero(parse_file_buffer, MAX_PATH_LENGTH);
4
5     char *p_ = "/";
6     strcpy(parse_file_buffer, (char *)path);
7     strcpy(parse_file_buffer+strlen(parse_file_buffer), p_);
8
9     int i = 0;
10    char *_p = &parse_file_buffer[0];
11
12    inode_t _inode;
13    uint32_t inum;
14    memcpy((uint8_t *)&_inode, (uint8_t *)inode_ptr, INODE_SIZE);
15
16    uint32_t l = strlen(parse_file_buffer);
17
18    for(; i < l; i++){
19        if(parse_file_buffer[i] == '/'){
20            parse_file_buffer[i] = '\\0';
21
22            inum = find_file(&_inode, _p);
23            sync_from_disk_inode(inum, &_inode);
24
25            _p = &parse_file_buffer[i+1];
26        }
27    }
28    return inum;
29 }
```

(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。

```

1 void write_block_index_in_inode(inode_t *inode_ptr, uint32_t idx, uint32_t block_index)
2 {
3     bzero(buffer1, POINTER_PER_BLOCK*sizeof(uint32_t));
4     bzero(buffer2, POINTER_PER_BLOCK*sizeof(uint32_t));
5     bzero(buffer3, POINTER_PER_BLOCK*sizeof(uint32_t));
6
7     if(idx < FIRST_POINTER){
8         inode_ptr->i_direct_table[idx] = block_index;
9         sync_to_disk_inode(inode_ptr);
10        return;
11    }
12
13    uint32_t free_index_1;
14    if(idx < SECOND_POINTER){
15        if(inode_ptr->i_indirect_block_1_ptr == 0){
16            free_index_1 = find_free_block();
17
18            set_block_bmp(free_index_1);
19            sync_to_disk_block_bmp();
20
21            superblock_ptr->s_free_blocks_cnt--;
22            sync_to_disk_superblock();
23
24            clear_block_index(free_index_1);
25            inode_ptr->i_indirect_block_1_ptr = free_index_1;
26            inode_ptr->i_fsize += BLOCK_SIZE;
27            sync_to_disk_inode(inode_ptr);
28        }
29        read_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
30        buffer1[idx - FIRST_POINTER] = block_index;
31        write_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
32        return;
33    }
34
35    uint32_t free_index_2;
36    if(idx < THIRD_POINTER){
37        if(inode_ptr->i_indirect_block_2_ptr == 0){
38            free_index_1 = find_free_block();

```



```
39
40     set_block_bmp(free_index_1);
41     sync_to_disk_block_bmp();
42
43     superblock_ptr->s_free_blocks_cnt--;
44     sync_to_disk_superblock();
45
46     clear_block_index(free_index_1);
47     inode_ptr->i_indirect_block_2_ptr = free_index_1;
48     inode_ptr->i_fsize += BLOCK_SIZE;
49     sync_to_disk_inode(inode_ptr);
50 }
51 read_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
52 if(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK] == 0){
53     free_index_2 = find_free_block();
54
55     set_block_bmp(free_index_2);
56     sync_to_disk_block_bmp();
57
58     superblock_ptr->s_free_blocks_cnt--;
59     sync_to_disk_superblock();
60
61     clear_block_index(free_index_2);
62     inode_ptr->i_fsize += BLOCK_SIZE;
63     sync_to_disk_inode(inode_ptr);
64
65     buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK] = free_index_2;
66     write_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
67 }
68 read_block(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK], (uint8_t *)
        buffer2);
69 buffer2[(idx - SECOND_POINTER) % POINTER_PER_BLOCK] = block_index;
70 write_block(buffer1[(idx - SECOND_POINTER) / POINTER_PER_BLOCK], (uint8_t *)
        buffer2);
71 }
72
73 uint32_t free_index_3;
74 if(idx < MAX_BLOCK_INDEX){
75     if(inode_ptr->i_indirect_block_3_ptr == 0){
76         free_index_1 = find_free_block();
77
78         set_block_bmp(free_index_1);
79         sync_to_disk_block_bmp();
80
81         superblock_ptr->s_free_blocks_cnt--;
82         sync_to_disk_superblock();
83
84         clear_block_index(free_index_1);
85         inode_ptr->i_indirect_block_3_ptr = free_index_1;
```

```

86         inode_ptr->i_fsize += BLOCK_SIZE;
87         sync_to_disk_inode(inode_ptr);
88     }
89     read_block(inode_ptr->i_indirect_block_3_ptr, (uint8_t *)buffer1);
90     if(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK)] == 0)
91     {
92         free_index_2 = find_free_block();
93         set_block_bmp(free_index_2);
94         sync_to_disk_block_bmp();
95
96         superblock_ptr->s_free_blocks_cnt--;
97         sync_to_disk_superblock();
98
99         clear_block_index(free_index_2);
100        inode_ptr->i_fsize += BLOCK_SIZE;
101        sync_to_disk_inode(inode_ptr);
102
103        buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK)] =
            free_index_2;
104        write_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
105    }
106    read_block(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK * POINTER_PER_BLOCK
        )], (uint8_t *)buffer2);
107    if(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK * POINTER_PER_BLOCK)) /
        POINTER_PER_BLOCK] == 0){
108        free_index_3 = find_free_block();
109
110        set_block_bmp(free_index_3);
111        sync_to_disk_block_bmp();
112
113        superblock_ptr->s_free_blocks_cnt--;
114        sync_to_disk_superblock();
115
116        clear_block_index(free_index_3);
117        inode_ptr->i_fsize += BLOCK_SIZE;
118        sync_to_disk_inode(inode_ptr);
119
120        buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK * POINTER_PER_BLOCK)) /
            POINTER_PER_BLOCK] = free_index_3;
121        write_block(buffer1[(idx - THIRD_POINTER) / (POINTER_PER_BLOCK *
            POINTER_PER_BLOCK)], (uint8_t *)buffer2);
122    }
123    read_block(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK *
        POINTER_PER_BLOCK)) / POINTER_PER_BLOCK], (uint8_t *)buffer3);
124    buffer3[(idx - THIRD_POINTER) % POINTER_PER_BLOCK] = block_index;
125    write_block(buffer2[((idx - THIRD_POINTER) % (POINTER_PER_BLOCK *
        POINTER_PER_BLOCK)) / POINTER_PER_BLOCK], (uint8_t *)buffer3);
126    return;

```

```
127     }
128     return;
129 }

1 void release_inode_block(inode_t *inode_ptr)
2 {
3     uint32_t i, j, k;
4     bzero(buffer1, POINTER_PER_BLOCK*sizeof(uint32_t));
5     bzero(buffer2, POINTER_PER_BLOCK*sizeof(uint32_t));
6     bzero(buffer3, POINTER_PER_BLOCK*sizeof(uint32_t));
7
8     for(i = 0; i < FIRST_POINTER; i++){
9         if(inode_ptr->i_direct_table[i] == 0){
10             return;
11         }
12         unset_block_bmp(inode_ptr->i_direct_table[i]);
13     }
14
15     if(inode_ptr->i_indirect_block_1_ptr == 0){
16         return;
17     }
18     read_block(inode_ptr->i_indirect_block_1_ptr, (uint8_t *)buffer1);
19     for(i = 0; i < POINTER_PER_BLOCK; i++){
20         if(buffer1[i] == 0){
21             unset_block_bmp(inode_ptr->i_indirect_block_1_ptr);
22             return;
23         }
24         unset_block_bmp(buffer1[i]);
25     }
26     unset_block_bmp(inode_ptr->i_indirect_block_1_ptr);
27
28     if(inode_ptr->i_indirect_block_2_ptr == 0){
29         return;
30     }
31     read_block(inode_ptr->i_indirect_block_2_ptr, (uint8_t *)buffer1);
32     for(i = 0; i < POINTER_PER_BLOCK; i++){
33         if(buffer1[i] == 0){
34             unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);
35             return;
36         }
37         read_block(buffer1[i], (uint8_t *)buffer2);
38         for(j = 0; j < POINTER_PER_BLOCK; j++){
39             if(buffer2[j] == 0){
40                 unset_block_bmp(buffer1[i]);
41                 unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);
42                 return;
43             }
44             unset_block_bmp(buffer2[j]);
45         }
46     }
```

```
46     unset_block_bmp(buffer1[i]);
47 }
48 unset_block_bmp(inode_ptr->i_indirect_block_2_ptr);
49
50 if(inode_ptr->i_indirect_block_3_ptr == 0){
51     return;
52 }
53 read_block(inode_ptr->i_indirect_block_3_ptr, (uint8_t *)buffer1);
54 for(i = 0; i < POINTER_PER_BLOCK; i++){
55     if(buffer1[i] == 0){
56         unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
57         return;
58     }
59     read_block(buffer1[i], (uint8_t *)buffer2);
60     for(j = 0; j < POINTER_PER_BLOCK; j++){
61         if(buffer2[j] == 0){
62             unset_block_bmp(buffer1[i]);
63             unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
64             return;
65         }
66         read_block(buffer2[j], (uint8_t *)buffer3);
67         for(k = 0; k < POINTER_PER_BLOCK; k++){
68             if(buffer3[k] == 0){
69                 unset_block_bmp(buffer2[j]);
70                 unset_block_bmp(buffer1[i]);
71                 unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
72                 return;
73             }
74             unset_block_bmp(buffer3[k]);
75         }
76         unset_block_bmp(buffer2[j]);
77     }
78     unset_block_bmp(buffer1[i]);
79 }
80 unset_block_bmp(inode_ptr->i_indirect_block_3_ptr);
81 return;
82 }
```
