Project 6 File System 设计文档

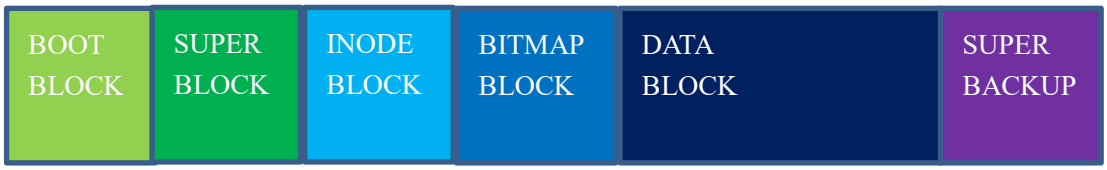
中国科学院大学

[姓名] 孔静

[日期] 2017/01/09

1. 文件系统初始化设计

（1）



总共有 256 块逻辑块。

```
#define BLOCK_COUNT 8
#define EXTENT_SIZE (FS_SIZE / BLOCK_COUNT)
```

BOOT BLOCK 在第 0 个逻辑块，SUPER BLOCK 在第 1 个逻辑块，它的备份在最后一个逻辑块，第（EXTENT_SIZE – 1 = 255）块，**各占一块。**

```
#define BOOT_BLOCK 0
#define SUPER_BLOCK 1
#define SUPER_BLOCK_BACKUP (EXTENT_SIZE - 1)
```

最先计算的是 BITMAP BLOCK 的逻辑块数量，简单的假设每个逻辑块都对应 1 位 BITMAP，每个逻辑块都对应一个 INODE，然后每个 INODE 又对应 1 位 BITMAP，所以就是每个逻辑块对应 2 位 BITMAP，剩下空间就是 INODE 和 DATA 的，每 1 个逻辑块的 INODE 可以有 128 个 INODE 信息记录，对应最多 128 个 DATA BLOCK，用剩余数量做除法的结果。减 1 除，最后加 1，目的是向上取整。

最终 BITMAP BLOCK 有 1 块，INODE BLOCK 有 2 块，DATA BLOCK 有 250 块。

```
#define BIT_MAP_BLOCKS ((2 * EXTENT_SIZE - 1) / BLOCK_BIT + 1)
//1
#define INODE_BLOCKS ((EXTENT_SIZE - 3 - BIT_MAP_BLOCKS - 1) / (1 + INODE_COUNT) + 1)
//2
#define DATA_BLOCKS (EXTENT_SIZE - 3 - BIT_MAP_BLOCKS - INODE_BLOCKS)
//250
```

（2）

在文件系统的最后一个逻辑块处，备份 superblock，每次启动时，检查 superblock 的 magic number，如果不是设定的数字，那么读取备份，并检查备份的 magic number，如果符合，将备份覆盖 superblock 即恢复被损坏的 superblock，如果不符合则执行 mkfs 初始化文件系统。

（3）

```
typedef struct {
    int magic_num; // 格式化与否
    int extent_size; // 文件系统的大小
    int inode_start; // Inode开始的地方
    int inode_number; // Inode数量
    int inode_blocks; // Inode所占block数
    int bitmap_start; // Block Allocation Map开始的地方
    int bitmap_blocks; // Block Allocation Map所占block数
    int data_start; // Data Block开始的地方
    int data_blocks; // Data Block所占block数
} super_block_t;
```

```
typedef struct {
    int size; // 所占字节大小
    short type; // 类型，目录 或 文件
    short number; //如果是目录，文件数量；如果是文件，打开次数
    short used_blocks; // 再使用的Block数
    short blocks[DIRECT_BLOCK_NUM]; // 直接指向Block
    short indirect_block; // 间接指向Block
    short links; // link数量
    char padding;//对齐
} inode_t;
```

一个 inode 支持 8 个直接块，间接的可以指向 4096 / 2 = 2048 个逻辑块，理论上最多 2056 个逻辑块，即**理论最大 8224KB**，8M 多一点，然而该文件系统给 DATA BLOCK 的数量仅 250，且根目录需要占 1 个 DATA BLOCK，间接索引也要占 1 个 DATA BLOCK，实

际上至多 248 个，即**实际最大文件大小 992KB**。最多文件数量，取决于 INODE 的数量，INODE 占 2 个逻辑块，2*128=256，**最多 256 个文件**。单个目录，一个目录对应一个 INODE，等同于一个文件，一个文件理论最大 8224KB，那么**理论最多文件/子目录数目是 131584-2 个，实际最多 15872-2=15870 个**（-2 是减去“.”目录本身信息以及“..”父目录信息。)

（4）

文件系统 BLOCK 分配策略，按需分配，需要用了，就 add_block 去申请一个可用的逻辑块。

（5）

没有写 bouns。

（6）

debug 太多了，各种 mistake，写的过程中没有遇到问题，怎么说，debug 的也都是一些逻辑上写错的东西。

2. 文件操作设计

（1）

link 先是检查一些东西，比如旧名字是否存在，即 path_lookup 函数返回有效 INODE，而新名字要不存在对应文件，以及该文件不能是目录类型，最后进行插入 link 操作，如果插入不成功（可能是空间不足)，则返回失败，成功，返回 0。

unlink 类似 link，检查需要 unlink 的东西是否存在，然后进行 unlink 操作，其中如果

unlink 后 link=0 并且没有打开它的话即 number=0 的话，会进行删除操作。

（2）

在 file_table 的 cursor 记录 lseek 位置。

```
typedef struct{
    bool_t is_open; // 是否打开
    int cursor; // 当前读/写位置
    short inode; // 文件Inode
    short mode; // 读写模式
} descriptor_t;
```

fs_lseek 支持超出文件大小的位置移动，会进行增加文件大小，并对增加的部分进行填 0 操作。

（3）

```
int fs_close(int fd)
{
    if(fd >= 0 && fd < MAX_FD_TABLE)
    {
        if(!file_table[fd].is_open)
            return -1;
        fd_free(fd);
        return 0;
    }
    return -1;
}
```

如图，先检查 fd 是否有效，范围有效，和是打开的文件的 fd，然后关闭即可。

其中 fd_free 操作，对打开的文件的 number（目前打开个数）和 link（连接数）判断，如果都是 0，会删除该文件。

（4）

比如一开始 link 和 unlink 没有判断是不是目录，然后进行 rmdir 删除目录后，目录 link

仍然保留着，就是一个 BUG，后来对 link、unlink 进行判断，如果是目录，无法 link 和 unlink。

其他记不太清楚了。

3. 目录操作设计

（1）

当前目录是个全局变量，最开始 current_dir = ROOT_DIRECTORY，只有 cd 的时候才会改变它。current_dir 记录当前目录的 INODE NUMBER。

（2）

写了一个路径解析函数，首先判断第一个字符是不是‘/’，如果是就将解析所选择的第一个目录切到 ROOT_DIRECTORY，不是就选择 current_dir，然后写了个循环，以‘/’为分割，不断循环去线性查找切换目录。

cd 和 ls，以及其他涉及到 fileName 之类的函数，均调用了这个路径解析函数，所以如果没有遗漏，所有的都是支持相对目录绝对目录的。

（3）

大部分是粗心写错，一旦定位到出 bug 的地方，一看就知道哪里错了。就是定位麻烦了点，我只会 printf，2333。

4. 关键函数功能

以下三个函数，以及 EXTENT 开头的宏，将读写清零从 512K 改为 4096K。

```
static void extent_read(short index, char *data_buf);
```

```
static void extent_write(short index, char *data_buf);

static void extent_zero(char *data_buf);
```

关于 DATA BLOCK 区域的读写。

```
static void data_read(short index, char *data_buf);

static void data_write(short index, char *data_buf);
```

关于 inode 的初始化、读写。

```
static inode_t *inode_read(short index, char *inode_buf);

static void inode_write(short index, char *inode_buf);

static short inode_init(int type);//type 为目录或文件
```

关于 inode 和 data block 空间的申请，释放。

```
static short space_alloc(int type);//type 为目录或文件

static void space_free(short index, int type, int number);
```

//不仅进行释放空间，对于 inode 类型，还会释放 inode 所占用的 datablock，对于目录类型，还会进行目录下内容的 unlink。

关于目录删除插入查找等操作。

```
static void dir_delete(file_entry_t *file);//link 减少，并判断是否应该删除文件。

static int dir_delete_swap(char *name, int dir_inode);//删除目录中的一项，并将最后一项与其替换位置，可能还会释放 data block 空间。
```

```
static int dir_insert(char *name, short dir_inode, short entry_inode);//插入
static short dir_find(char *name, short dir_inode);//查找
```

空间不够了，需要申请 data block 时。

```
static short add_block(inode_t *inode);
```

路径解析，线性查找，循环中调用 dir_find 函数。

```
static short path_lookup(char *name);
```

文件描述符申请，释放

```
static int fd_alloc(short index, int flags);//查找是否有可用的文件描述符
static void fd_free(int fd);//关闭文件，并判断是否删除文件。
```

int fs_init(void);启动

int fs_mkfs(void);初始化

int fs_open(char *fileName, int flags);打开文件，支持路径解析；如果文件不存在但可写模式，创建文件

int fs_close(int fd);关闭文件。

int fs_read(int fd, char *buf, int count);读文件。

int fs_write(int fd, char *buf, int count);写文件，支持超出文件大小部分写，当然前提是可以申请到足够空间。

int fs_lseek(int fd, int offset);改变文件描述符中 cursor 位置。

int fs_mkdir(char *fileName);创建目录，支持路径解析

int fs_rmdir(char *fileName);删除目录，支持路径解析

int fs_cd(char *dirName);切换目录，支持路径解析

int fs_link(char *old_fileName, char *new_fileName);创建链接，支持路径解析

int fs_unlink(char *fileName);删除链接，支持路径解析

int fs_stat(char *fileName, fileStat *buf);显示文件/目录信息，支持路径解析

int fs_ls(short index, char *name);显示当前目录信息，不支持路径解析

参考文献

[单击此处键入参考文献内容]