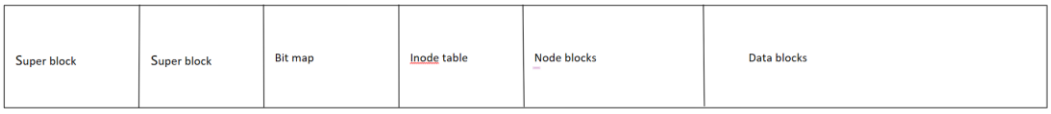
Project 6 File System 设计文档

中国科学院大学
张旭
2018/1/21

1. 文件系统初始化设计

请至少包含以下内容

（1） 请用图表示你设计的文件系统对磁盘的布局



一个 inode 结构实际大小 284B，对齐 196B，一个块有 8 个 inode，inode 占有 15136 个块。Inode 块对应的 map 有 15136B，每个字节对应一个块内的 8 个 inode。

data 有 1025792 个块，对应的 map 有 128224B。

map 有 34 块。

Dentry 结构实际大小 268。一个块里面有 15 个 dentry 结构。

硬盘总共有 1048576 块，未利用 7612 块

（2） 你如何实现 superblock 的备份？如何判断 superblock 损坏，以及当有一个 superblock 损坏时你的文件系统如何正常启动？

创建文件系统，以及每次在文件系统内创建文件时，我会备份 superblock，备份函数如下：

```
void write_super(){
    char buf[4096];
    memset(buf,0,4096);
    struct superblock_t *sptr;
    sptr = (struct superblock_t *)buf;
    sptr->magic = MAGIC;
    sptr->f_blocks = DATA_BLOCK_NUM;
    sptr->f_files = 8 * INODE_BLOCK_NUM;
    sptr->f_ffree = sblock.f_files;
    sptr->f_bfree = sblock.f_bfree;
    sptr->Sblock2_offset = SECTOR_SIZE;
    sptr->Sblock_len = 64;
    sptr->Bmap_offset = 2 * SECTOR_SIZE;
    sptr->Bmap_len = DATA_MAP_SIZE;
    sptr->imap_offset = sptr->Bmap_offset + DATA_MAP_SIZE;
    sptr->imap_len = INODE_MAP_SIZE;
    sptr->Itable_offset = 36 * SECTOR_SIZE;
    sptr->Itable_len = sblock.Itable_ptr - (struct inode_t*)(sptr->Itable_offset);
    sptr->datablock_offset = sblock.datablock_offset;
    sptr->datablock_len = (sblock.data_ptr - sblock.datablock_offset)/SECTOR_SIZE;
    device_write_sector(buf, 0);
    device_write_sector(buf, 1);
}
```

对于判断 superblock 损坏，我仅仅认为当 magic 不对时，该超块被破坏，实际上被破坏的情况有很多。当主超块被破坏时，我会读出备用超块进行初始化，同时修复主超块。

（3） 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。

内存中的超块结构如下：

```
struct superblock{
    unsigned int f_blocks;
    unsigned int f_files;
    unsigned int f_ffree;
    unsigned int f_bfree;
    unsigned int imap_offset;
    unsigned int sb_offset;
    unsigned int Itable_offset;
    struct inode_t *Itable_ptr;
    unsigned int datablock_offset;
    unsigned int data_ptr;
    // Add what you need, Like locks
};
```

磁盘中的超块结构如下：

```
struct superblock_t{
    unsigned int magic;
    unsigned int fs_mode;
    unsigned int Sblock2_offset;
    unsigned int Sblock_len;
    unsigned int Bmap_offset;
    unsigned int Bmap_len;
    unsigned int imap_offset;
    unsigned int imap_len;
    unsigned int Itable_offset;
    unsigned int Itable_len;
    unsigned int datablock_offset;
    unsigned int datablock_len;
    unsigned int f_blocks;
    unsigned int f_files;
    unsigned int f_ffree;
    unsigned int f_bfree;
    // complete it
};
```

各项含义：

- f_blocks：系统总共的块数
- f_files：系统最大 inode 数
- f_ffree：剩余可用 inode 数
- f_bfree：剩余可分配的块数
- imap_offset：inode 对应的 map 偏移量
- imap_len：inode 对应的 map 大小
- Bmap_offset：data blocks 对应的 map 偏移量
- Bmap_len：data blocks 对应的 map 大小
- Fs_mode：文件系统的权限
- Magic：文件系统的魔数
- Sblock_len：超块的大小
- Sblock2_offset：备用超块的起始地址
- datablock_offset：数据块的起始地址
- datablock_len：数据块的大小

内存中 inode 结构如下：

```
struct inode{
    char fname[256];
    unsigned int file_type;
    unsigned int file_size;
    unsigned int link_count;
    unsigned int creat_time;
    unsigned int mode;
    //unsigned int imm_point[10];
    unsigned int current_inode;
    unsigned int f_inode;//第一次创建时的父节点
    unsigned int first_point;
    unsigned int slink_node;
    // Add what you need, Like locks
};
```

磁盘中 inode 结构如下：

```
struct inode_t{
    // complete it
    char fname[256];
    unsigned int file_type;
    mode_t mode;
    unsigned int file_size;
    unsigned int link_count;
    unsigned int creat_time;
    //unsigned int imm_point[10];       //10个块
    unsigned int first_point;       //1024个块（4M）
    unsigned int f_inode;
    unsigned int current_inode;
    unsigned int slink_node;
};
```

各项的含义：

- fname：文件名称
- file_type：文件的类型
- mode：文件权限
- file_size：文件的大小
- link_count：硬链接的个数
- create_time：创建时间
- first_point：指向数据块的一级指针
- f_inode：父节点的 inode 编号
- current_inode：当前节点的 inode 编号
- slink_node：符号连接的 inode 编号

在我的设计中总共可以有 121088 个文件，每个文件通过一个一级指针指向它的数据，故一个文件最大 4M。

一个目录下最多有 1024 个项。

（4） 请说明你设计的文件系统的块分配策略，按需分配还是有设计其他分配策略？

创建一个文件时，首先分配一个页面，由一级指针指向，然后再按需分配块。

（5） 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）

由于对 fuse 一些结构和固有返回值不太熟悉，碰了很多坑，建议老师以后给与更多的信息。

2. 文件操作设计

（1） 请说明 link 和 unlink 的操作流程

Link 函数如下：

```
int p6fs_link(const char *path, const char *newpath)
{
    char *name = path;
    char *Lname = newpath;
    char buf[4096];
    unsigned int name_len, Lname_len, haxi, iindex, Lindex, Lhaxi;
    struct inode inode_temp, link_itemp;
    struct dlocation dloc;
    struct dentry *dptr, dentry_temp;
    memset(buf,0,4096);
    iindex = fpath2inode(&name, &inode_temp);
    Lindex = fpath2inode(&Lname, &link_itemp);
    Lhaxi = path_parse(Lname);
    haxi = path_parse(name);
    name_len = strlen(name);
    Lname_len = strlen(Lname);
    Lindex = find_hash(&dloc, link_itemp.first_point, haxi, Lname, Lname_len);
    iindex = find_hash(&dloc, inode_temp.first_point, haxi, name, name_len);
    read_inode(&link_itemp, Lindex);
    device_read_sector(buf, dloc.phoffset / SECTOR_SIZE);
    dptr = (struct dentry*)buf + dloc.index;
    //memcpy(&dentry_temp, dptr, DENTRY_SIZE);
    dptr->ftype = link_itemp.file_type;
    dptr->finode = link_itemp.f_inode;
    link_itemp.link_count += 1;
    inode_temp.file_size += (link_itemp.file_size - SECTOR_SIZE);
    device_write_sector(buf, dloc.phoffset / SECTOR_SIZE);
    write_inode(&link_itemp);
    write_inode(&inode_temp);
    return 0;
    //memcpy(dptr)
}
```

流程：

1） 找到两个路径对应文件的 inode 以及父节点的 inode

2） 修改 path 父节点的目录项，将 path 对应的 inode 编号替换为 newpath 的 inode 编号

3） Newpath 文件的 link_count 加一

4） 写回磁盘

Unlink 函数如下：

```
int p6fs_unlink(const char *path)
{
    char *name = path;
    unsigned int name_len,haxi, iindex;
    struct inode inode_temp, link_itemp;
    struct dlocation dloc;
    iindex = fpath2inode(&name, &inode_temp);
    haxi = path_parse(name);
    name_len = strlen(name);
    iindex = find_hash(&dloc, inode_temp.first_point, haxi, name, name_len);
    read_inode(&link_itemp, iindex);
    link_itemp.link_count--;
    if (link_itemp.link_count == 0) {
        free_inode(iindex);
    }
    else{
        write_inode(&link_itemp);
    }
    inode_temp.file_size -= link_itemp.file_size;
    write_inode(&inode_temp);
    return 0;
}
```

流程：

1） 找到 path 连接的 inode

2） Link_count 减一

3） 若 link_count 为 0，则调用 free_inode 函数，删除 inode。

4） 改变 Path 的父节点的文件大小

5） 写回磁盘

（2） 请说明 rename 涉及的操作流程

Rename 函数如下：

```
int p6fs_rename(const char *path, const char *newpath)
{
    char *name = path;
    char *Nname = newpath;
    int i, j, flag;
    unsigned int name_len, Nname_len, haxi, iindex, Nindex, Nhaxi;
    unsigned int *ptr;
    struct inode inode_temp, New_itemp, iinode;
    struct dlocation dloc,Ndloc;
    struct dentry *dptr, dentry_temp,pre_dentry,next_dentry;
    iindex = fpath2inode(&name, &inode_temp);
    Nindex = fpath2inode(&Nname, &New_itemp);
    Nhaxi = path_parse(Nname);
    haxi = path_parse(name);
    name_len = strlen(name);
    Nname_len = strlen(Nname);
    iindex = find_hash(&dloc, inode_temp.first_point, haxi, name, name_len);
    Nindex = find_hash(&Ndloc, New_itemp.first_point, haxi, Nname, Nname_len);
    read_inode(&iinode, iindex);
    inode_temp.file_size -= iinode.file_size;
    iinode.f_inode = New_itemp.current_inode;
    New_itemp.file_size += iinode.file_size;
    write_inode(&iinode);
    write_inode(&New_itemp);
    write_inode(&inode_temp);
    read_dentry(&dentry_temp, dloc.index, dloc.phoffset);
    read_dentry(&pre_dentry, dloc.pre_index, dloc.phoffset);
    read_dentry(&next_dentry, dentry_temp.next_index, dloc.phoffset);
    pre_dentry.next_index = dentry_temp.next_index;
    next_dentry.pre_index = dentry_temp.pre_index;
    dentry_temp.next_index = -1;
    dentry_temp.pre_index = Ndloc.pre_index;
    write_dentry(&pre_dentry, dloc.pre_index, dloc.phoffset);
    write_dentry(&next_dentry, pre_dentry.next_index, dloc.phoffset);
    write_dentry(&dentry_temp, Ndloc.index, Ndloc.phoffset);
    return 0;
}
```

流程：

1） 找到两个路径对应文件的 inode 以及父节点的 inode

2） 将 newpath 父节点的目录项根据 path 的 inode 进行修改（维护链表）

3） 清除 path 父节点的目录项

4） 写回磁盘

3. 目录操作设计

（1） 请说明 rmdir 的操作流程？

Rmdir 函数如下：

```
int p6fs_rmdir(const char *path)
{
    char *name = path + 1;
    rmdir(name,&root_inode);
    return 0;
}
```

我递归调用 rmdir 函数，以实现递归修改文件大小。函数截图如下：

```
int rmdir(char *name, struct inode *root) {          //返回文件夹大小
    char *next_name;
    unsigned int name_len, haxi, iindex,fsize;
    struct inode inode_temp, *temp;
    struct dlocation dloc;
    next_name = strchr(name, '/');
    haxi = path_parse(name);
    if (next_name != NULL) {
        name_len = next_name - name;
        iindex = find_hash(&dloc, root->first_point, haxi, name, name_len);
        read_inode(&inode_temp, iindex);
        next_name++;
        fsize = rmdir(next_name, &inode_temp);
        root->file_size -= fsize;
        write_inode(root);
        return fsize;
    }
    else {
        name_len = strlen(name);
        iindex = find_hash(&dloc, root->first_point, haxi, name, name_len);
        read_inode(&inode_temp, iindex);
        char    buf[4096];
        memset(buf,0,4096);
        struct dentry *dptr;
        unsigned int read_sector = dloc.phoffset / SECTOR_SIZE;
        device_read_sector(buf, read_sector);
        dptr = (struct dentry *)buf + dloc.index;
        memset((void*)dptr, 0, sizeof(struct dentry));
        device_write_sector(buf, read_sector);
        write_imap(iindex,0);
        root->file_size -= inode_temp.file_size;
        write_inode(root);
        return inode_temp.file_size;
    }
}
```

对于递归的每一层，流程：

1） 找到对应的 inode

2） 修改文件大小。并写回磁盘

3） 返回底层传来的文件大小

最底层：

1） 从父节点的目录中删除对应的目录项

2） 删除 inode 节点

3） 修改 map，写回磁盘

4） 返回文件大小

4. 关键函数功能

目录项布局设计：

一个目录一共 63 个组（630 个块，总共 1034 个块），每组（由开头字母区分）可以有 150 个目录项，文件的命名允许 52 个字母（大小写），10 个数字和‘_’。

组内前 126 个目录项由第二个字符区分，后 15 个为溢出区

第三个字符决定放在高目录项或低目录项。

若前 63 个组内没查到，则到后 63 个备用组内查找（每组六个块，378 个块），备用组有 90 个目录项，前 63 个目录项由第二个字符区分，后 15 个为溢出区。

dentey 的 fname[0]=‘\0’时，说明该目录项没有被使用。

所以本系统最多允许前两个字母相同的目录项 54 个。

Hash 函数查找流程：

1） 若一级间指块未分配所在的地址，分配块，并返回。

2） 根据 hash 值找到链表起始位置，沿着链表往下查找，找到则返回地址。

3） 若未找到，在溢出区，备用区，备用溢出区依次寻找一个空位，维护链表并返回地址。

此外，为了方便，我完成了很多功能的函数封装，详情见代码。