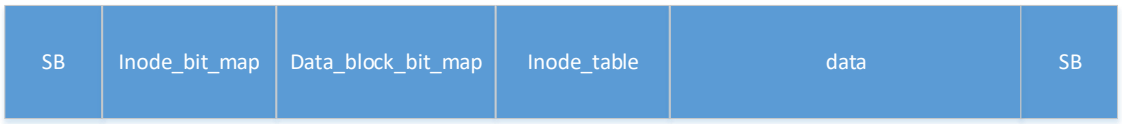
Project 6 File System 设计文档

中国科学院大学
[姓名] 袁峥
[日期] 2018.1.19

1. 文件系统初始化设计

（1）请用图表示你设计的文件系统对磁盘的布局（布局上可以不考虑 boot block 的大小，直接从逻辑地址第 0 块开始），并说明各部分占用的磁盘空间大小，例如 superblock，inode 的元数据等



设计每个 block 为 4KB，对于 4GB 的磁盘一共有 1024*1024=1048576 个 block。
superblock 及其备份各占一个 block，inode_bit_map 占 16 个 block，datablock_bit_map 占 32 个 block，inode_table 占 16384 个 block，剩余 1032142 个 block 为数据区。

（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？

理论上，在整个 4G 磁盘的第一个 block 和最后一个 block 为 superblock，其中第一份为主 superblock，后一份为备份，两份同时修改，当第一份被破坏时，读取第二块中的内容。在实现时，由于为了便于测试两块 superblock 全部被破坏时的情况，若将备份块放在磁盘最后一块，数据覆盖耗时太长，因此将备份块放在了第二块。

判断 superblock 是否损坏可以采用将 superblock 所在块中的所有字节进行异或，并将结果记录在 superblock 中，读取时判断异或结果是否正确，如果不正确则表示已经被破坏。在实现时，采用了另一种较为简易的方面，对 p6fs 文件系统设置了一个 magic number，并记录在 superblock 块的开头，读取时判断该数是否正确，如果错误则表示已经被破坏。

当有一个 superblock 损坏时，初始化函数会自动读取备份块中的数据，判断备份块中的数据是否也被破坏，如果被破坏则重新创建文件系统，否则根据备份块中的数据组织磁盘结构。

（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。
superblock 数据结构：

```
struct superblock_t{
    // complete it
    uint32_t magic_number;
    uint32_t size;
    uint32_t num_of_inode;
    uint32_t num_of_datablock;
    uint32_t size_of_inode;
    uint32_t size_of_datablock;
    uint32_t start_add_inode;
    uint32_t start_add_datablock;
    uint32_t start_add_inode_bit;
    uint32_t start_add_datablock_bit;
}sb_t;
```

magic_number 为自定义的 p6fs 的魔数，用于判断 superblock 是否损坏。
size 表示该 p6fs 文件系统的大小，在本实验中为 4GB。
num_of_inode 为文件系统中最大支持的 inode 数，为 512*1024。
num_of_datablock 为文件系统中最大支持的数据块数，为 1032142。
size_of_inode 为每个 inode 元数据的大小，为 128B。
size_of_datablock 为每个数据块的大小，为 4096B。
start_add_inode、start_add_datablock、start_add_inode_bit、start_add_datablock_bit 分别为 p6fs 文件系统中 inode 块、数据块、inode 的 bitmap、datablock 的 bitmap 的起始地址。

inode 数据结构：

```
struct inode_t{
    // complete it
    uint32_t mode;
    uint32_t count;
    uint32_t size;
    uint32_t access_time;
    uint32_t modify_time;
    uint32_t create_time;
    uint32_t direct_ptr[10];
    uint32_t first_layer_ptr;
    uint32_t second_layer_ptr;
};
```

mode 为 inode 节点所表示的文件类型：文件、目录、软链接以及 9 位操作权限。
count 为该 inode 的引用计数。
size 为该 inode 所代表的文件的大小。
access_time、modify_time、create_time 分别为该 inode 的上次访问时间、上次修改时间和创建时间。
direct_ptr、first_layer_ptr、second_layer_ptr 分别为该 inode 所代表的文件内容的 10 个直接指针、1 个一级间址指针和 1 个二级间址指针。

最大文件大小：10*4KB+1024*4KB+1024*1024*4KB=4GB+4MB+40KB。
最多文件数目：即 superblock 中的 num_of_inode，为 512*1024。
单个目录下能支持的最多文件/子目录数目：在进行目录解析时，考虑到 p6fs 的实际情况，至采用了一级间址指针的解析，没有进行二级间址指针的解析。每个文件在目录中占用 64B，因此单个目录下最多支持的文件数为$\frac{4MB+40KB}{64B}$=67109504。

（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？

块分配策略为按需分配，在 p6fs_init、p6fs_write、p6fs_symlink、p6fs_mkdir 等函数中需要新块时进行分配，同时在 p6fs_truncate、p6fs_rmdir、p6fs_unlink、p6fs_rename 等函数

释放数据块时进行回收。

（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）

在设计上 superblock 的备份块应该和主 superblock 块的距离离的相对远一点比较好，因此在设计上本来将 superblock 分别放在 4G 磁盘的第一块和最后一块，但是在实现后的测试时发现，如果想要将两个块都破坏到需要往磁盘内写入 4G 内容，速度较慢，因此为了测试两个 superblock 都被破坏的情况，将备份 superblock 放在离主 superblock 较近的地方。

2. 文件操作设计

（1）请说明 mknod 涉及的操作流程

1、解析路径是否存在，如果存在则返回 EEXIST。

2、解析父路径是否存在，如果不存在则返回 ENOENT。

3、分配 inode，并在 inode 中写入相关信息。

4、在父目录页中写入新文件的信息，并更新父目录的 inode。

（2）请说明 link 和 unlink 涉及的操作流程

link：

1、解析路径 path 和 newpath，如果 newpaht 存在则返回 EEXIST，如果 path 不存在则返回 ENOENT。

2、解析 path 的父路径是否存在，如果不存在则返回 ENOENT。

3、更新 inode 中的 count（引用计数）及 modify_time。

4、在 newpath 的父目录中加入 newpath 的信息，并更新其父目录的 inode。

unlink：

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、如果 inode 的 count 大于 1，将 count 减 1，并更新 modify_time。

3、如果 inode 的 count 等于 1，删除 inode 及其所包含的 datablock，并在父目录中删除其路径，更新 inode。

（3）请说明 open 涉及的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、分配文件描述符，并填入 ino 及 flag 信息。

3、将 fileInfo->fh 指向分配的文件描述符。

（4）请说明 read 涉及的操作流程

1、根据 fileInfo->fh 中的 ino 获取 inode 信息。

2、查看 fileInfo->fh 中的 flag，判断是否读权限。

3、先定位到 offset 位置，然后从该位置开始读取 size 大小到 buf。

4、更新 inode 的 access_time。

（5）请说明 write 涉及的操作流程

1、根据 fileInfo->fh 中的 ino 获取 inode 信息。

2、查看 fileInfo->fh 中的 flag，判断是否写权限。

3、先判断当前文件大小是否小于 offset+size，若小于则分配满足 offset+size 的 datablock。

4、先定位到 offset 位置，然后从该位置开始写 size 大小的 buf 中的内容。

5、更新 inode 的 access_time、modify_time、size。

（6）请说明 truncate 涉及的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、如果 newsize 大于原文件的 size，则在原文件后面补 newsize-size 大小的 0。

3、如果 newsize 小于原文件的 size，则将原文件大于 newsize 的部分删除，并回收相应的 datablock。

4、更新 inode 中的 size、modify_time、access_time 信息。

（7）请说明 release 涉及的操作流程

释放文件描述符，并将其 fileInfo->fh 赋为 NULL。

（8）请说明 rename 涉及的操作流程

1、解析路径 path 和 newpath，如果 newpaht 存在则返回 EEXIST，如果 path 不存在则返回 ENOENT。

2、在路径 path 的父目录中删除 path 的信息，并更新其 inode。

3、在路径 newpath 的父目录中增加 newpath 的信息，并更新其 inode。

（9）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）

在文件操作中比较复杂的是 read 和 write 中对文件读写时，需要根据 offset 和 size 来对文件进行定位，这部分需要根据具体大小来解析到其在哪一级文件块指针，在解析的过程中要分类讨论的情况较多，而且较容易出错。

3. 目录操作设计

（1）请说明 mkdir 的操作流程

1、解析需要创建的目录路径是否存在，如果已经存在则返回 EEXIST。

2、解析创建目录的父目录是否已经存在，如果不存在则返回 ENOENT。

3、分配 inode 和 datablock，分别存放新目录的元数据和目录页。

4、在新分配的 datablock 中放入“.”和“..”的信息。

5、在创建目录的父目录的目录页中放入新创建目录的信息，并更新父目录的 inode 信息。

（2）请说明 rmdir 的操作流程

1、解析需要删除的目录路径是否存在，如果不存在则返回 ENOENT。

2、查看需要删除的目录是否为空目录，如果不是空目录则返回 ENOTEMPTY。

3、删除目录的目录页及 inode 并更新 bitmap 进行回收。

4、在删除目录的父目录中删除新删除的目录信息，并更新其 inode。

（3）请说明 readdir 的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、根据目录页大小逐项读取目录页中的信息，并通过 filler 函数放入 buf。

（4）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）

这个实验的代码量比较大，在一开始的设计上做的不是很好，后来发现在目录和文件部分有许多代码其实可以复用，在代码复用这一块做的不是特别好，导致写的代码比较长。

4. 其他操作设计

（1）请说明 symlink 涉及的操作流程

1、解析路径 link，如果已经存在则返回 EEXIST。

2、解析 link 的父路径，如果不存在则返回 ENOENT。

3、分配新的 inode 及 datablock，在 datablock 中写入 path，并更新 inode 信息（mode 设为软链接）。

4、更新 link 的父目录页，加入 link 的信息，并更新其父目录的 inode。

（2）请说明 readlink 涉及的操作流程

1、解析路径 path，如果不存在则返回 ENOENT。

2、判断其 inode 的 mode 是否为 S_IFLNK（软链接），如果不是则返回 EINVAL。

3、从 inode 所指向的 datablock 中读出路径并放入 link 中。

4、更新 inode 中的 access_time 信息。

（3）请说明 getattr 涉及的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、将 inode 中的 mode、count、size、modify_time、access_time、create_time 存入 statbuf 中的相应项中。

3、通过 fuse_get_context 函数获得 uid 和 gid。

（4）请说明 utime 涉及的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、将 inode 中的 modify_time 及 access_time 赋为 ubuf 中的相应项数据。

（5）请说明 chmod 涉及的操作流程

1、解析路径是否存在，如果不存在则返回 ENOENT。

2、更新 inode 中的 mode 信息。

（6）请说明 statfs 涉及的操作流程

填写 statInfo 中的 f_bsize、f_frsize、f_blocks、f_bfree、f_bavail、f_files、f_ffree、f_favail 的信息。

（7）请说明 init 涉及的操作流程

读取磁盘中的第一个 block，查看其中的 magic_number 是否正确，若不正确则读取备份 superblock 中的信息，查看 magic_number 是否正确，若两个块中的 magic_number 都不正确则进入 mkfs，否则进入 mount。

mkfs：

1、将 superblock 块中应保存的值同时存入主 superblock 及备份 superblock。

2、同时清空磁盘和内核中的 inode_bitmap 及 block_bitmap。

3、分配 inode 及 datablock，存放根目录的信息。

mount：

1、将 superblock 中相关数据存入内核。

2、将磁盘中的 inode 信息存入内核。

3、扫描 inode_bitmap 及 block_bitmap，计算剩余数，并将 bitmap 存入内核。

（8）请说明 destroy 涉及的操作流程

释放内核中的 inode，并关闭磁盘及 log_file。

5. 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

1、p6fs_init（common.c）

```
void* p6fs_init(struct fuse_conn_info *conn)
{
    /*init fs
     think what mkfs() and mount() should do.
     create or rebuild memory structures.

     e.g
     S1 Read the magic number from disk
     S2 Compare with YOUR Magic
     S3 if(exist)
        then
            mount();
        else
            mkfs();
     */

     int i,j,k;
     struct superblock_t* sb_ptr;
     int flag = 1;
     unsigned char buf[SECTOR_SIZE];

     // initial file descriptor table and inode table
     for (i = 0; i < MAX_OPEN_FILE; i++)
        fd_table[i].used = FALSE;
     for (i = 0; i < MAX_INODE; i++)
        pthread_mutex_init(&(inode_table[i].mutex), NULL);

     free_block = 1032142;
     free_inode = 512*1024;

     device_read_sector(buf,0);
     sb_ptr = (struct superblock_t *)buf;
     if (sb_ptr->magic_number != MAGIC_NUM){
        device_read_sector(buf,1);
        sb_ptr = (struct superblock_t *)buf;
        if (sb_ptr->magic_number != MAGIC_NUM)

     }

     if (flag == 1){   //mount
        sb_t.magic_number = MAGIC_NUM;
        sb_t.num_of_datablock = sb_ptr->num_of_datablock;
        sb_t.num_of_inode = sb_ptr->num_of_inode;
        sb_t.size = sb_ptr->size;
        sb_t.size_of_datablock = sb_ptr->size_of_datablock;
        sb_t.size_of_inode = sb_ptr->size_of_inode;
        sb_t.start_add_datablock = sb_ptr->start_add_datablock;
        sb_t.start_add_datablock_bit = sb_ptr->start_add_datablock_bit;
        sb_t.start_add_inode = sb_ptr->start_add_inode;
        sb_t.start_add_inode_bit = sb_ptr->start_add_inode_bit;

        sb.sb = &sb_t;
        pthread_mutex_init(&(sb.mutex), NULL);
        for (i = 0; i < 16384; i++)
        {
            device_read_sector(buf,ino_block+i);
            for (j = 0; j < 32; j++){
                inode_table[i*32+j].inode = (struct inode_t *) malloc(sizeof(struct inode_t));
                memcpy(inode_table[i*32+j].inode,&buf[j*128],sizeof(struct inode_t));
            }
        }

        for (i = ino_bit; i < ino_bit+16; i++)
        {
            device_read_sector(buf,i);
            memcpy(inode_bit_map+(i-ino_bit)*SECTOR_SIZE,buf,SECTOR_SIZE);
            for (j = 0; j < SECTOR_SIZE; j++)
            {
                for (k = 0; k < 8; k++)
                {
                    if (buf[j] & 0x1) free_inode--;
                    buf[j] = buf[j] >> 1;
                }
```

```
        }
    }

    for (i = data_bit; i < data_bit+32; i++)
    {
        device_read_sector(buf,i);
        memcpy(block_bit_map+(i-data_bit)*SECTOR_SIZE,buf,SECTOR_SIZE);
        for (j = 0; j < SECTOR_SIZE; j++)
        {
            for (k = 0; k < 8; k++)
            {
                if (buf[j] & 0x1) free_block--;
                buf[j] = buf[j] >> 1;
            }
        }
    }
    DEBUG("mount finished!!!");
} « end if flag==1 »
else{           //mkfs
    memset(buf,0,SECTOR_SIZE);
    sb_ptr = (struct superblock_t *)buf;
    sb_ptr->magic_number = MAGIC_NUM;
    sb_ptr->num_of_datablock = 1032142;
    sb_ptr->num_of_inode = 512*1024;
    sb_ptr->size = 4*1024*1024;   //KB
    sb_ptr->size_of_datablock = 4096;
    sb_ptr->size_of_inode = 128;
    sb_ptr->start_add_inode_bit = SECTOR_SIZE*ino_bit;
    sb_ptr->start_add_datablock_bit = SECTOR_SIZE*data_bit;
    sb_ptr->start_add_inode = SECTOR_SIZE*ino_block;
    sb_ptr->start_add_datablock = SECTOR_SIZE*data_block;
    memcpy(&sb_t,buf,sizeof(sb_t));
    sb.sb = &sb_t;
    pthread_mutex_init(&sb.mutex, NULL);
    device_write_sector(buf,0);
    device_write_sector(buf,1);

    memset(buf,0,SECTOR_SIZE);
    for (i = ino_bit; i <ino_block; i++)
        device_write_sector(buf,i);
    for (i = 0; i < free_inode; i++)
        inode_table[i].inode = (struct inode_t *) malloc(sizeof(struct inode_t));

    memset(inode_bit_map,0,16*SECTOR_SIZE);
    memset(block_bit_map,0,32*SECTOR_SIZE);

    //write root_dir datablock
    int block_id = create_datablock();
    struct dentry* den_ptr;
    den_ptr = (struct dentry*)buf;
    strcpy(den_ptr->name,".");
    den_ptr->ino = block_id;
    den_ptr++;
    strcpy(den_ptr->name,"..");
    den_ptr->ino = block_id;
    device_write_sector(buf,data_block);

    //write root_dir inode
    int inode_id = create_inode();
    memset(buf,0,SECTOR_SIZE);
    struct inode_t* inode_ptr;
    inode_ptr = (struct inode_t*)buf;
    inode_ptr->size = 2*sizeof(struct dentry);
    inode_ptr->access_time = time(NULL);
    inode_ptr->modify_time = inode_ptr->access_time;
    inode_ptr->mode = S_IFDIR | 0755;
    inode_ptr->count = 1;
    inode_ptr->direct_ptr[inode_id] = data_block;
    device_write_sector(buf,ino_block);

    memcpy(inode_table[inode_id].inode,buf,sizeof(struct inode_t));

    DEBUG("mkfs finished!!!");
} « end else »

root_dir = (struct entry_m *) malloc(sizeof(struct entry_m));
strcpy(root_dir->name,"/");
root_dir->ino = 0;
root_dir->sibling = NULL;
root_dir->first_child = NULL;

inode_table[0].tree_node = root_dir;
/*HOWTO use @return
struct fuse_context *fuse_con = fuse_get_context();
fuse_con->private_data = (void *)xxx;
return fuse_con->private_data;

the fuse_context is a global variable, you can use it in
all file operation, and you could also get uid,gid and pid
from it.

*/
return NULL;
} « end p6fs_init »
```

首先根据主 superblock 及备份 superblock 中的 magic_number 判断进入 mount 还是 mkfs。然后分别进行挂载或者创建文件系统。

2、p6fs_read（common.c）

```
int p6fs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fileInfo)
{
    /* get inode from file handle and do operation*/
    int ino = ((struct file_info*)(fileInfo->fh))->ino;

    if ((((struct file_info*)(fileInfo->fh))->flag & 0x3)== O_WRONLY)
        return -EACCES;

    int i,j;
    int have_read = 0 ,actual_size = min(inode_table[ino].inode->size-offset,size);
    unsigned char temp_buf[SECTOR_SIZE];
    unsigned char first_layer_buf[SECTOR_SIZE];
    unsigned char second_layer_buf[SECTOR_SIZE];
    int first_layer_num,second_layer_num;
    uint32_t *first_layer_ptr,*second_layer_ptr;

    int flag = 0;
    if (offset < 10*SECTOR_SIZE){
        for (i = offset/SECTOR_SIZE; i < 10; i++){
            device_read_sector(temp_buf,inode_table[ino].inode->direct_ptr[i]);
            if (have_read == 0){
                memcpy(buf, temp_buf + offset%SECTOR_SIZE,min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE));
                have_read += min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE);
            }
            else if (actual_size-have_read < SECTOR_SIZE){
                memcpy(buf + have_read, temp_buf,actual_size-have_read);
                have_read += actual_size-have_read;
            }
            else{
                memcpy(buf + have_read, temp_buf,SECTOR_SIZE);
                have_read += SECTOR_SIZE;
            }
            if (have_read == actual_size){
                flag = 1;
                break;
            }
        }
    } « end if offset<10*SECTOR_SIZE »

    if (offset < (10+1024)*SECTOR_SIZE && flag == 0){
        if (offset < 10*SECTOR_SIZE)
            first_layer_num = 0;
        else
            first_layer_num = offset/SECTOR_SIZE-10;
        device_read_sector(first_layer_buf, inode_table[ino].inode->first_layer_ptr);
        first_layer_ptr = (uint32_t *)first_layer_buf;
        for (i = first_layer_num; i < 1024; i++){
            device_read_sector(temp_buf,*(first_layer_ptr+i));
            if (have_read == 0){
                memcpy(buf, temp_buf + offset%SECTOR_SIZE,min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE));
                have_read += min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE);
            }
            else if (actual_size-have_read < SECTOR_SIZE){
                memcpy(buf + have_read, temp_buf,actual_size-have_read);
                have_read += actual_size-have_read;
            }
            else{
                memcpy(buf + have_read, temp_buf,SECTOR_SIZE);
                have_read += SECTOR_SIZE;
            }
            if (have_read == actual_size){
                flag = 1;
                break;
            }
        }
    } « end if offset<(10+1024)*SECT... »
    if (flag == 0){
        if (offset < (10+1024)*SECTOR_SIZE)
            second_layer_num = 0;
        else
            second_layer_num = (offset/SECTOR_SIZE-(10+1024))/1024;
        device_read_sector(second_layer_buf, inode_table[ino].inode->second_layer_ptr);
        second_layer_ptr = (uint32_t *)second_layer_buf;
        for (i = second_layer_num; i < 1024; i++){
            device_read_sector(first_layer_buf,*(second_layer_ptr+i));
            first_layer_ptr = (uint32_t *)first_layer_buf;
            if (have_read == 0)
                first_layer_num = ((offset-(10+1024)*SECTOR_SIZE)%(1024*SECTOR_SIZE))/SECTOR_SIZE;
            else

                first_layer_num = 0;
            for (j = first_layer_num; j < 1024; j++){
                device_read_sector(temp_buf,*(first_layer_ptr+j));
                if (have_read == 0){
                    memcpy(buf, temp_buf + offset%SECTOR_SIZE,min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE));
                    have_read += min(actual_size,SECTOR_SIZE-offset%SECTOR_SIZE);
                }
                else if (actual_size-have_read < SECTOR_SIZE){
                    memcpy(buf + have_read, temp_buf,actual_size-have_read);
                    have_read += actual_size-have_read;
                }
                else{
                    memcpy(buf + have_read, temp_buf,SECTOR_SIZE);
                    have_read += SECTOR_SIZE;
                }
                if (have_read == actual_size){
                    flag = 1;
                    break;
                }
            }
            if (flag == 1)
                break;
        } « end for i=second_layer_num;i<... »
    } « end if flag==0 »
    //memset(buf+have_read,0,size-actual_size);
    device_read_sector(temp_buf,ino_block+ino/32);
    inode_table[ino].inode->access_time = time(NULL);
    memcpy(temp_buf+(ino%32)*128,inode_table[ino].inode,sizeof(struct inode_t));
    device_write_sector(temp_buf,ino_block+ino/32);
    return have_read;
} « end p6fs_read »
```