Project 6 File System 设计文档

中国科学院大学
张远航
2018 年 1 月 21 日

1. 文件系统初始化设计

(1) 文件系统磁盘布局

设计的磁盘布局与 Design Review 时展示的相同，且在老师的建议下，将备份的超级块移动到了磁盘尾端，如表 1 所示，

超级块 Superblock	数据块位图 Datablock Bitmap	i-node 位图 i-node Bitmap	i-node 表 i-node Table	数据块 Datablocks	备份超级块 Superblock Backup

表 1: 磁盘布局

其中各部分空间分配如下（扇区大小与数据块大小相同，均为 4KB）：两个超级块各 4KB；磁盘共 4GB，大约有 2^20 个可用的空闲块，每个块需要 1 位来记录是否被占用，这样数据块位图共需要 64 个扇区；i-node 位图同理，我们遵循 EXT 文件系统的传统，bytes-per-inode 比选取 16384，这样共有 2^18 个可用的 i-node ，如果每个结构体分配 128B 则需要 32MB 的存储空间，对应于 2^13 个扇区。接下来的部分为数据块；整个磁盘的最后一个扇区为备份的超级块。

(2) 超级块与 i-node 结构

超级块。 超级块数据结构定义如下，主要包含的是文件系统的相关统计信息：

```
struct superblock_t{
    int magic_number;               // 文件系统的魔数
    int size;                       // 文件系统的总大小

    int inode_table;                // i-node表所在扇区
    int inode_map;                  // i-node位图所在扇区
    int total_inode_cnt;            // i-node总数
    int free_inode_cnt;             // 可用i-node数

    int block_map;                  // 数据块位图所在扇区
    int total_block_cnt;            // 数据块总数
```

```
    int free_block_cnt;             // 可用数据块总数
};
```

P6FS 挂载时会首先读取主超级块，如果魔数符合 P6FS 定义，则认为存在文件系统，然后读取备份超级块分别求校验码（这里采用的是较为流行的 MD5 校验，`digestMem` 函数会将内存中指定起始位置和大小的单元进行 MD5 校验），然后进行对比。如果二者不一致，则认为某个超级块有损坏，需要从 i-node 和数据块位图重建超级块。

如果读出主超级块的魔数不符合 P6FS 定义，且备份超级块的魔数正常，则认为备份超级块是正常的，使用备份超级块即可。

i-node。 i-node 数据结构定义如下：

```
struct inode_t{
    int size;                       // 文件大小（以字节计）
    mode_t mode;                    // 文件模式（包含权限位、类型等）
    unsigned int link_count;        // 链接次数

    int block[MAX_DIRECT_NUM];      // 直接块的扇区号
    int indirect_table;             // 一级间址块的扇区号

    time_t ctime;                   // 创建时间
    time_t atime;                   // 访问时间
    time_t mtime;                   // 修改时间

    uid_t uid;                      // 所有人ID
    gid_t gid;                      // 所有组ID
};
```

这些属性的选取与 `<sys/stat.h>` 文件中的相关字段密切相关。最初单独为文件类型设定了一个字段 `type`，后来发现 `stat.h` 中有 `S_ISLNK(mode)`、`ALLPERMS` 之类的宏可以直接借用，就把整个文件系统的设计统一起来了。

由于只实现了一级间址块，该文件系统能支持的最大单个文件大小为 $12 \times 4\,\mathrm{K} + 1024 \times (4\,\mathrm{KB} / 4\,\mathrm{B}) \times 4\,\mathrm{K} \approx 4.05\mathrm{M}$，因此这一 4GB 的文件系统平均大概能容纳 1000 个文件左右。假设每个文件都只占用 i-node 而不实际分配数据块，那么该文件系统所能支持的最大文件数目为 2^{18}；参考 Linux 文件系统的实际情况，我们只给目录文件分配了一个直接块（大小为 4096B）。目录中，`dentry` 的数据结构设计如下，每条记录为一个文件的文件名和其 i-node 号：

```
struct dentry{
    char filename[MAX_FILENAME_LEN];
    int ino;
};
```

其中文件名长度最大值 `MAX_FILENAME_LEN` 设定为 64，因此可计算出一个目录下能够存放的文件/子目录数最多为 60 个。

注意，内存内存储的数据结构副本需要互斥锁以保证并发访问的正确性。

(3) 块分配策略

文件系统采用按需分配块策略，块分配和块回收通过 `alloc_blocks` 和 `recycle_blocks` 两个函数完成。当文件已分配的数据块不足时，`alloc_blocks` 会查询数据块位图为 i-node 分配新的直接/间接块；`recycle_blocks` 则会在文件变小时释放出文件不再需要的块（包括间址块）。

(4) 实现过程中问题和得到的经验

调试是一个非常漫长的过程，直接看一眼我的 commit 记录好了……[1]

想说的其实有很多，但是一一写出来实在需要很久，而提交时间快到了……总之，出现问题最多的地方基本上都是指针，还有数据类型之间的 implicit cast。以及，QNX 那个手册页面上非常清楚地指明了每一个函数的操作流程，对我后期完善异常处理帮助很大。得到的经验是，要多看手册，利用好系统头文件。

一步步地实现了基本功能、Bonus 功能、异常处理、权限检查，感觉还是很有成就感的。

2. 文件操作设计

(1) link 与 unlink 操作

`link` 操作实现了硬链接操作。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 EISDIR 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`，然后将其 ino 设为来源的 ino 并将 i-node 中的链接数加一。

`unlink` 操作可以删除一个给定的符号链接、硬链接或一般文件，其流程如下：首先，确认删除目标不是一个文件夹，否则报 EISDIR 异常；否则，按其文件名在父目录中找到相应的 `dentry` 并删除，然后将其对应的 i-node 的链接次数减一。当链接次数减小到 0 时，文件彻底不复存在，此时应释放其 i-node，更新超级块中的文件系统统计信息并回收其所拥有的全部数据块。

[1] https://github.com/sailordiary/B62010Y-B62011Y-COS318-UCAS/commits/master

(2) 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。

(3) mknod 和 symlink 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。如果是目录和符号链接的话，还要分配一个直接块。对于符号链接，其直接块中应填入指向目标的完整路径。新创建文件和目录的权限通过 mode 参数的权限位判定，符号链接的默认权限为 0777（Linux 的传统）。

(4) Bonus: utime 操作

touch 命令会修改文件的时间戳，这是通过 utime 实现的。按照 QNX 手册中的提示，utime 操作实现如下：首先确认当前用户是否有权限修改文件信息，然后根据 ubuf 是否为空完成下列操作：如果是无权限（通过 fuse_get_context 取得的访问 UID 与文件的 UID 不一致），则返回 EACCES 异常（ubuf 为空）或 EPERM 异常（ubuf 不为空）；有权限的情况下，如果 ubuf 为空指针，更新 i-node 的访问和修改时间戳为当前时间；否则；将时间戳修改为 ubuf 结构体中指定的时间即可。

(5) Bonus: chmod 操作

chmod 操作会修改文件的权限位，主要是以下三步实现：

```
inode->mode &= ~ALLPERMS;
inode->mode |= (mode & ALLPERMS);
inode->ctime = time(NULL);
```

其中， ALLPERMS 是 sys/stat.h 中给出的权限位掩码。

3. 目录操作设计

(1) 目录删除操作

rmdir 这一操作步骤如下：首先判断路径是否合法（“.” 和 “..” 不能删除；但是 FUSE 似乎会主动判断）；检查目录是否为空（否则报 ENOTEMPTY 异常）；接着，释放为目录分配的直接块，并在父目录中删除该目录对应的 dentry。

这里有一个小细节：最初我没有想好怎么在初始化一个目录块时快速地将除. 和.. 外的目录项无效掉，后来想了一个很简单的方法，直接把整个扇区先用 `memset` 填成 -1（0xFF），然后再写入. 和.. 的目录项，这样把整个扇区当做 `dentry` 数组读出时其他项的 ino 就都为 -1 了。

4. 关键函数功能

- `inode_from_path()`：解析路径名，更迭查询 dentry 所在的数据块号和目录的 i-node 号；中间需要检查访问权限、目录是否存在以及中间层是否为目录。
- `recycle_blocks()` 和 `alloc_blocks()`：块分配和块回收，先分配/回收直接块，再分配/回收间接块。注意应该先判断完是否有足够空间再对相关的数据结构做实际的修改操作，否则会破坏文件系统的一致性。
- `digestMem()`：一个简单的 MD5 校验函数，原本由 RSA 官方用 C++ 实现，修改的 `md5.h` 去掉了类的定义，使得其可用于 C 语言。