Fslab 实验报告

李梓童 2017202121

I. 块设备的组织结构

superblock	bitmap	bitmap	Root directory	……	

采用 linked-based approach，每一个磁盘块中有一个 NextBlock 可以存储下一个关联块的顺序编号。因此第一个 root inode 作为根目录，其后可以连接 datablock 或 inodeblock。一共有 256MB/4KB=65536 blocks=4K*2*8，故用两个 bitmap 块来存储块的使用情况。

磁盘块结构：

```
struct disk_block {
    int size; //  当前块已用空间
    long NextBlock; //下一个块号
    char data[MAX_DATA_IN_BLOCK];
};
```

II. 文件信息节点的结构

```
struct inode_file_dir { //  约  90 bytes
    int uid;
    int gid;
    int link_count; //有几个链接指向这个 inode
    char fname[MAX_FILENAME + 1]; //文件名称
    char fext[MAX_EXTENSION + 1]; //拓展名
    long fsize;    //文件大小
    long StartBlock; //初始块号
    time_t atime;
    time_t mtime;
    time_t ctime;
    int flag;    //0:未使用; 1:常规文件; 2:目录文件
};
```

III. 实现函数的重要细节

- int fs_getattr (const char *path, struct stat *attr)

  解析路径要点：

  前期首先从 superblock 获取根目录信息，若路径为“/”，则直接返回根目录文件 inode，

并处理路径为空的情况。

中期把形如“/path1/path2/path3/path4”的路径通过循环拆解成若干“parent/child”的目录形式，在循环中，使用 strchr(p, '/')以及*p=’\0’截断来分割读取父子文件的名称，找到父目录后，循环遍历父目录中各 inode 的文件名，若找到所需要的文件名读取改文件名指向的新的数据块。遍历父目录下文件名的循环停止情况有：找到所需文件节点、没有找到但是已经偏移量已达到父目录文件大小。外层总路径循环停止情况为 strchr(q, '/')==NULL，即路径中已经不存在“/”，达到了最次级子文件。

后期将文件信息读到 inode 结构体中，进行结构体内容的复制。

- int fs_mknod (const char *path, mode_t mode, dev_t dev)
  int fs_mkdir (const char *path, mode_t mode)

  以上两个函数由 int create_inode(const char* path, int flag)一并实现。

  前期需解析 path，获取父目录信息，这一过程由函数 anal_path 实现。anal_path 大体与 get_attr_from_path 相似，也是大循环套小循环、把全路径拆解成若干父子文件的形式。与 get_attr_from_path 不同之处在于，在遍历父目录下文件的内层循环中，anal_path 另外取了一个 file_dir_2 和 start_blk 来分别存储父文件对应的 inode 结构体和起始数据块编号，并将起始数据块编号返回。找到父目录后，将指针指向父目录下内容的尾部，准备新建 inode。

  把新生成的 inode 写入其父目录下、并为新 inode 分配 datablock 时需考虑可用空间不够大的情况。这里主要用 find_free_blocks 和 enlarge_block 来处理。Find_free_blocks 中用位运算，循环处理读取的 bitmap 中的每个字节，找到在不大于用户需要的 num 个数下最大的连续空闲块个数 max，这些连续块中的第一个块编号以长整型指针返回。Enlarge_block 中，首先若 find_free_blocks 返回的 max 为 0，则说明没有空闲块了，返回-ENOSPC，否则将新找到的块编号接到父目录块的 NextBlock 上，更新父目录块以及初始化新的数据块。

- int fs_rmdir (const char *path)
  int fs_unlink (const char *path)

  以上两个函数由 int rm_file_dir(const char *path, int flag) 一并实现。

  从路径 path 读取相应的 inode 信息之后，如果是常规文件，则使用 NextClear 将其对应的数据块清空。NextClear 内部是一个循环，用类似于处理链表的方式读取从起始块开始的下一个数据块，将其在 bitmap 上置零，终止条件是 NextBlock 值为-1，表示已经到了链表尾端。如果是目录文件，则先判断它是否非空。如果是空目录，则将它的 flag 置为 0，表示为空 inode。

- int fs_rename (const char *oldpath, const char *newname)

  前期：如果新旧路径相同，则正常返回。对于路径 newname 对应的 inode，若不存在，则先用 create_inode 造出一个和 oldpath 指向的 inode 类型相同的 inode，并解析出 newname 中的最次级父目录。

  然后处理新旧路径对应的 inode 类型不一致和新路径(若为目录文件)非空的情况。

将旧路径指向的 inode 信息赋到新路径 inode 上，即链接转移，并更新时间，将修改后的 inode 写入父目录下。

最后把旧路径中的链接删除。

- int fs_write (const char *path, const char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)

  前期对传入的参数进行处理。读取 path 指向的 inode，如果 offset 超过 inode 的文件大小，则返回错误。如果 inode 数据块不止一个，则需计算出 offset 对应到相关数据块中的偏移量,也就要减掉前面多出来的部分。对于写入的 size 也有多种情况：如果 size+修改后的 offset 仍未超过数据块最大容量，则可以直接写入；如果超过数据块最大容量，则要 enlarge_block 添加新块；如果不够写，则要返回-ENOSPC。实际写入时,还存在写失败时把状态恢复到写之前的问题,处理方法简单概括为“先模拟再实操”，将在第四部分详细说明。

  完成数据块写入之后，更新 inode 的相关信息。

- int fs_truncate (const char *path, off_t size)

  分三种情况讨论：

  一，size 小于 path 指向 inode 的文件大小，也就是要缩减数据块。先算出 size 跨越了几个数据块，到交界的数据块处把 size 以后的数据置空，把其后的数据块用 nextClear 清空。

  二,size 大于 path 指向 inode 的文件大小但不需要添加新的数据块。这个比较方便，修改 inode 的 size 即可。

  三，size 大于 path 指向 inode 的文件大小并且需要添加新的数据块。这个在处理的时候也需要考虑空间不够的问题,所以用先模拟一遍再实操的方法,和 write 相似。

- int fs_utime (const char *path, struct utimbuf *buffer)

  从 path 读 inode，复制 buffer 中的信息，写回 path。

- int fs_statfs (const char *path, struct statvfs *stat)

  用 getbitmapused()计算 bitmap 中使用了的数据块个数。

  关于可用节点数：由于使用的是链表连接磁盘块的方法，并没有很严格的 inode 数据区，用户可以把所有空间都拿来放 inode，也可以正常一点使用；所以在计算可用节点数的时候，采用的计算方法是（数据块大小/节点大小）*剩余可用数据块数量,实际上做的是最乐观的估计,假设用户把余下来的所有数据块都拿来放节点了。

- int fs_open (const char *path, struct fuse_file_info *fi)

  int fs_opendir (const char *path, struct fuse_file_info *fi)

  int fs_release (const char *path, struct fuse_file_info *fi)

  int fs_releasedir (const char *path, struct fuse_file_info *fi)

  在 fs_open 函数中判断了一下 O_APPEND，写入到 fi->fh 区。想着也许在 write 中会用到，但是实际测试的时候发现不用也可以，所以就没有再做改动。

- int fs_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)

read 中考虑的也主要是 size、offset、文件大小三者之间的关系。

如果 offset+size>文件大小，则读取文件内的部分。如果 offset>文件大小，则不读取。读取时跳过 offset 覆盖的数据块，比较在交界块处的 real_offset、size 的大小。如果 size>数据块大小，则要读的不止一个数据块，采用循环方式把数据块内容填到 buffer 中去。

最后更新 inode。

IV. 遇到的问题及解决方案

- **进行 Write 操作时，若写失败，则如何恢复写之前的数据?**

  考虑写失败的原因为空间不足，如果先把能写的块写了、然后才发现找不到可用空间块了，之前写入的块里就等于是不完整数据。为了避免这个情况，采用先模拟写一遍的方法，有点像 log。第一遍 get_free_block 时，仅仅在 towrite 和 size 上操作，不涉及实际的磁盘读写，如果顺利跑完整个循环，则说明有足够的空间进行读写，之后再进行第二遍循环，此时会将数据实际写入磁盘。

  同样的操作在 truncate 中也有出现。

- **什么情况下，rename 函数返回-ENOSPC?**

  当把一个常规文件移动到新的目录下时，如果该目录没有足够的空间来放这个常规文件（没有足够多的指针来指向常规文件的数据块），则会返回-ENOSPC。

  但是在 linked-approach 情境中，只要 oldpath 下常规文件存在、newname 指向的 inode 成功创造，理论上就不会有-ENOSPC 报错。核心原因是 inode 中的“指针”只有一个 Startblock，只要将其替换为 oldpath-inode 的 Startblock 即可，不存在 Multi-index 中数量有限指针的问题。而 oldpath-inode 在 create 的时候已经考虑过空间是否足够的情况，如果空间不够则在 create 时便会返回-ENOSPC，oldpath-inode 的存在性便保证了常规文件有足够的存储空间。

V. 反思总结

- 链表连接数据块和指针指示两种数据块组织方式的比较：

  用指针域存放数据块地址，最大可存储数据由指针个数决定。链表连接数据块，则最大可用数据块个数由已使用的数据块和磁盘总大小决定。且链表方式中的 inode 指示数据块位置内容的大小小于指针方式中的 inode，可看成链表把后者 inode 里指针占用的存储空间分散到了各个数据块上。

  在读取方式上，指针索引式文件系统可以随机读取，而链表连接式文件系统则需要读完前面几个数据块才能到达目标块，在随机读取时效率很差。

  在碎片整理上，如果文件写入的块过于离散，文件读取的性能就会变得很差。这时就需要通过碎片整理将同一个文件所属的区块集合在一起，这样数据的读取会比较容易。链表式的文件系统需要碎片处理，而指针索引式文件系统则基本上不太需要。

VI. 参考资料

- Github 上 ext2 的实现，虽然和链表操作不太一样，但在函数的策略（如什么情况下报什么错误 ）上有所参考：https://github.com/alperakcan/fuse-ext2
- Ext2 inode 设计：https://elixir.bootlin.com/linux/latest/source/fs/ext2/ext2.h