用户态文件系统实验报告

郑凯文

2021 年 6 月 18 日

1 简介

本项目中，我实现了一个简易的文件系统`easy-fs`（在模块划分、磁盘布局上，很大程度上参考了操作系统课程Rust版本的easy-fs），并将其接入到FUSE完成用户态文件系统。在原先`easy-fs`的基础上，支持了多级目录、硬链接、一定程度的并发，并添加了简单的多用户权限控制与面向整盘的加密。

2 代码说明与运行方式

`/src`文件夹下为文件系统、FUSE接口、一个简单测例与两个面向文件系统的性能测例源码，在`/fs`、`/test`、`/bench1`、`/bench2`下分别make得到执行文件。`bench_mount1.cpp`和`bench_mount2.cpp`则是在FUSE挂载后，使用Linux文件io的性能测例，直接使用g++命令编译。

默认设置下，模拟硬盘的文件位于`/tmp/disk*`（在测试并发时，添加序号后缀以模拟多个设备），挂载点位于`/disk`。设备的总容量为1G，若出现文件损坏或密码错误，会出现打开失败的问题，可使用`rm /tmp/disk*`删除文件并重新挂载，这时程序会自动新建合法文件以模拟硬盘。

本项目使用了FUSE的C++ wrapper Fusepp，FUSE版本为最新。这样便于利用C++的一些面向对象特性，也是我广泛使用以使代码简洁的方法：

- 使用`shared_ptr`管理对象指针，当引用计数为0时，自动析构
- 使用模板、函数对象以及lambda表达式实现的匿名函数，方便地将某块数据作为对象访问并调用其成员函数

使用FUSE包装的文件系统有三种调用方式：

- `./easyfs /disk 0 -f`
- `./easyfs /disk 2 uid gid -f`
- `./easyfs /disk 3 uid gid password -f`

其中可以指定装载硬盘的uid、gid和密码，若不指定则默认uid、gid为0，密码为空。这里采取的是在挂载时给定uid、gid的方式，而非使用`getuid()`、`getgid()`系统调用获取，原因是这样方便测试，否则需要在真实系统中创建和切换用户和组。

命令中加上`-f`参数使得挂载位于前台，可以方便地用`Ctrl+C`结束挂载。使用`-d`参数可以输出调试信息，在性能测试时一般不使用。

3 文件系统设计

文件系统自下而上可以分为若干层级：

- 块设备
- 块缓存
- 磁盘数据结构
- Inode
- 文件系统接口

3.1 块设备

将实际操作的硬盘抽象为块设备，在模拟中，单个设备为单个文件。硬盘按照一定的块大小被分割为许多块，这里采用的块大小为512字节。块设备实现的接口为，给定块id，将数据从硬盘读入内存，或从内存写入硬盘，也就是提供了设备和内存的界面。

实现时，使用`fread()`、`fwrite()`、`fseek()`进行文件的读写。每个块设备都有一个读写锁，读写都在锁的保护下进行，因此单个设备的读写是原子的。

若只使用单个设备，则不能做到真正的并发，因此我模拟了多个设备的情况，在`utils.h`中可以更改设备数量。所有设备的容量总和为1G，对于某个块id，将其均匀映射到某个设备中，每个设备有单独的锁，这样就可以进行分布式并发读写。

3.2 块缓存与块缓存管理器

对磁盘的频繁读写（用文件读写模拟）性能较低，因此在内存中设置一定大小的缓存。使用系统结构的语言，我使用的是4096组的16路组相联，采用写回、写分配，因此缓存的总容量为32MB。每个块id根据低12位映射到4096组中的一组，组内则简单地采用FIFO替换策略。

块缓存管理器使用`vector`和数组保存着所有块缓存，且使用`shared_ptr`管理，每个块缓存提供了读、写函数，且有dirty位。当某块块缓存的引用计数为0时（生命周期结束），将自动析构，析构时会检查dirty位来判断是否写回。这样，当别处用不到时，块缓存管理器只需从`vector`中将块缓存移除，便可自动完成写回。

每个块都有着自己的锁，因此对同个块的操作是互斥的。块缓存管理器则在查询、替换、调入时，对涉及到的组的FIFO队列加锁。需要注意的是，为了防止不同线程持有对应同一块id的不同块缓存对象（这样对于块的操作就不互斥了），块缓存管理器在替换时，会选择从前到后第一个引用计数为1（在别处没有使用）的块缓存进行替换，这需要保证线程同时持有的块缓存数乘以

线程数不大于路数，否则可能找不到块可以替换。在整个系统中，线程同时持有的块缓存数不超过4，因此16路的缓存大约可以支持4个线程。

再往上的层次对磁盘的访问都通过一个全局的块缓存管理器进行。只需提供块id，块缓存管理器便可提供对应的块缓存，调入、替换的操作都对上层透明。

3.3 磁盘数据结构

3.3.1 磁盘布局

SuperBlock	Inode Bitmap	Inode Area	Data Bitmap	Data Area

在磁盘上，按照块id从小到大，分为5个连续区域：

- 超级块，长度为1个块
- Inode位图，长度若干个块，记录后面Inode区域的分配情况
- Inode区域，长度许多个块，每个块都放了若干磁盘Inode
- 数据位图，长度若干个块，记录数据块的分配情况
- 数据区域，长度许多许多个块，每个块存放文件数据/多级索引的次级索引/目录的目录项

3.3.2 超级块

```
struct SuperBlock
{
    u32 magic;
    u32 total_blocks;
    u32 inode_bitmap_blocks;
    u32 inode_area_blocks;
    u32 data_bitmap_blocks;
    u32 data_area_blocks;
};
```

超级块虽然占用第0号块，但实际数据大小远小于512字节。其第一个字段是magic number，必须为一个特定值，它用来判断文件系统是否损坏/密码是否错误。后面若干字段记录了磁盘布局情况，用它就可以定位各个区域。

3.3.3 位图

位图区域为若干个块，每个块的512字节也就是4096比特，可以用来表示4096个位置的占用情况，0表示未占用，1表示占用。

设置了Bitmap对象来对某段位图进行管理，它可以进行分配或回收。为了加速查找，将每个块解释为u64数组，首先找到第一个非全1的u64的位置，再使用__builtin_ctzll()找到第一个0的位置。

每个位图具有一个锁，分配、回收都需要上锁。每次分配、回收修改位图区域的某个块后，都需要立即同步（实际上，在我的实现中，所有的元数据都需要修改后立即同步以保证崩溃一致性，见之后的说明）。

3.3.4 磁盘Inode与数据块

```
struct DiskInode
{
    u32 size;
    u32 direct[inode_direct_count];
    u32 indirect1;
    u32 indirect2;
    u32 nlink;
    u32 uid;
    u32 gid;
    u32 mode;
    DiskInodeType type;
    u32 dirent_num;
    i64 atime;
    i64 ctime;
};
```

磁盘Inode保存在Inode区域，其长度为128字节，因此一个块可以保存4个磁盘Inode。其记录了文件系统中文件/目录的各种元数据，包括大小，多级索引，硬链接数，文件所有者的uid、gid，权限，访问时间、修改时间等。

读取文件数据时，通过磁盘Inode中的索引进行。其包括若干个一级索引（直接指向包含文件数据的数据块），一个一级索引（指向包含128个直接索引的数据块）和一个二级索引（指向包含128个一级索引的数据块）。块大小为512字节，而一个索引（块id）为4字节，因此一个间接索引块可以包含128个索引。在文件较小时，只需要使用直接索引，速度块，多级索引需要多次访问磁盘，但有效增加了文件的最大长度。对于二级索引，最多支持索引128x128个数据块，共128x128x512字节也就是8MB。

在文件大小不足时，提供函数increase_size()，通过位图分配数据块（位于数据区域），它们用作间接索引块或保存文件数据的块，使得文件大小足够。同样有函数clear_size()可以来回收用作间接索引块、文件数据块的所有数据块。

读写文件时，磁盘Inode提供了read_at()、write_at()函数来读写文件数据。它会自动查询多级索引，定位到某个位置的文件数据所在的数据块。

对于多级索引的正确性，我写了测例test，向一个文件写入8MB的数据并读取和对比内容。8MB的大小用上了直接索引、一级索引和二级索引，可以证明它们都是正确的。

3.3.5 目录项

对于目录而言，可以将其视为普通的文件，只是文件内容为顺序排布的目录项。当然，对于用户而言，修改目录的文件内容是被禁止的，在处理文件读写等调用时，文件和目录会被区别对待。

```
struct DirEntry
{
    u8 name[name_length_limit + 1];
    u32 inode_number;
};
```

目录项的大小为32字节，记录着目录中文件名称及其Inode编号。通过目录项，可以定位到下一级文件或目录。

3.3.6 Inode

Inode为在内存中方便管理磁盘Inode的简单对象，它记录着Inode编号、磁盘Inode所在块id与偏移量。它提供了一系列对于文件/目录操作的包装，本质上是利用块缓存管理器得到磁盘Inode所在的块，将其解释为DiskInode对象，并调用其提供的一系列函数。

事实上，通过Inode来进行文件操作时（无论是修改文件元数据、文件扩容或修改文件数据），同一个文件，甚至磁盘Inode位于同一个块上的文件都是互斥的，这是因为读取/修改磁盘Inode，需要获取其所在的磁盘块的锁。即使是不同的Inode对象，若指向同一文件，都会通过块缓存管理器获取同一个磁盘块，而对磁盘块的操作是上锁的。因此，我的文件系统只支持非常有限的并发：同时读/写磁盘Inode位于不同磁盘块的文件。对于读写相同 件的不同块、相同块，都是串行的。

3.3.7 文件系统接口

EasyFileSystem将Inode进行了简单包装，内部存储着磁盘布局、根目录Inode、Inode位图和数据位图，向用户提供了文件系统的必要功能。

```
class EasyFileSystem
{
    static shared_ptr<EasyFileSystem> open(shared_ptr<BlockDevice> _block_device);
    static shared_ptr<EasyFileSystem> create(shared_ptr<BlockDevice> _block_device);
    shared_ptr<Inode> find(string path, i32 &err);
    shared_ptr<Inode> create(string path, DiskInodeType type, i32 &err, u32 mode);
    i64 unlink(string path);
```

```
    i64 rename(string from, string to);
    i64 link(string from, string to);
    shared_ptr<Inode> get_inode(u32 inode_id);
};
```

包括创建、打开文件系统，查找文件/目录，创建文件/目录，删除硬链接，文件移动，创建硬链接等。不使用FUSE接口，它已经可以作为一个独立的文件系统使用。在并发性能测试时，一种测例便是绕过FUSE，直接使用文件系统的测例。

4 权限与加密

4.1 多用户与权限

仿照Linux的用户权限系统，我添加了朴素的多用户支持和权限检查。

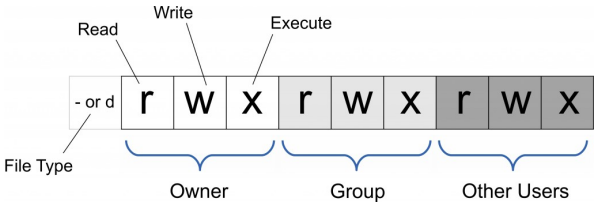

图 1: 权限模式

用户的属性有用户id和组id，在创建文件或目录时，在磁盘Inode中记录Owner。同时，文件或目录具有权限模式字段，它可以用一个整数表示，其中的若干标志位决定着访问权限（图1），即同个uid/同个gid/其它用户是否可以读/写/执行。

如之前所属，为了方便测试，用户的uid和gid从运行时的命令行参数得到。在处理文件系统调用时，对权限进行的检查有：

- 访问某个文件（包括读、写文件，删除文件等任何涉及了文件有关信息的操作）时，当前用户必须具有文件树上经过的目录的读权限和执行权限
- 创建文件/目录/删除硬链接时，当前用户必须具有文件树上经过的目录的读权限和执行权限，同时具有最后一级目录的写权限
- 创建硬链接时，当前用户必须具有相对于来源的读权限，相对于目标所在最后一级目录的写权限
- 读/写文件/读目录/删除硬链接时，当前用户必须具有目标的相应读/写权限
- 移动文件时，当前用户必须具有源、目标所在的最后一级目录的写权限

在这里，假设uid=0的用户为root，拥有所有权限。

4.2 加密

我实现的是对于整盘的加密，包括超级块、位图、磁盘Inode、数据在内的所有磁盘块都被加密。

由于使用了块设备的抽象，这种加密可以容易地实现，我们只需要把底层的块设备改为“加密块设备”：在读取磁盘块时进行解密，写入磁盘块时进行加密。

密码从命令行参数获取，程序会首先对密码进行SHA3-256得到256位摘要，将前128位作为密钥，后128位作为初始向量，使用AES-128进行加解密。密码算法采用的是我在现代密码学课程上的实现代码，其中AES-128参照了OpenSSL的写法，使用T表和循环展开加速，可以达到超过OpenSSL的吞吐率。SHA3-256则比OpenSSL慢不少，但其只是在开始得到密钥和初始向量时使用一次，不构成性能瓶颈。

文件系统在打开时，会检查magic number，若密码错误，magic number有误，那么文件系统打开失败。

5 FUSE接口实现

我实现了`getattr`、`opendir`、`readdir`、`releasedir`、`open`、`read`、`write`、`fsync`、`release`、`create`、`mkdir`、`unlink`、`rmdir`、`rename`、`link`、`chmod`、`chown`共17种接口。

在并发问题上，我在整个FUSE系统中设置了1个锁，采取的策略是：

- 对于涉及到文件树查询或修改的操作，如创建目录/打开文件，为整个调用过程加FUSE锁，不同调用互斥
- 对于文件读写（在FUSE的约定中，`read`、`write`时已经进行了`open`，此时获取到了文件Inode编号，可以直接利用Inode进行文件读写，无需访问文件树），不加FUSE锁。此时，操作是否互斥，取决于是否受到块缓存互斥的限制，即磁盘Inode位于不同磁盘块的文件的读写是可以并发的

在各种调用中，我处理的错误类型包括：

- ENOTDIR：路径某一级（非最后一级）非目录，或打开目录时最后一级非目录
- ENOENT：文件/目录不存在
- EISDIR：打开文件时最后一级是目录
- ENAMETOOLONG：文件/目录名过长（创建文件目录，移动文件，硬链接）
- EEXIST：文件/目录已存在（创建文件目录，移动文件，硬链接）
- ENOTEMPTY：删除目录时，目录非空
- EPERM：操作非法，如对目录创建硬链接
- EACCES：权限不足

5.1 getattr

给定文件/目录路径，获取`struct stat`格式的文件信息。

5.2 opendir readdir releasedir

这是一套连续的调用，一般用在ls等命令行指令中。opendir时根据fi->flags & O_ACCMODE决定打开模式（读/写/读写），判断权限，将Inode编号存入fi->fh，在读取时使用。readdir时，遍历目录内容，传递所有子目录/子文件的struct stat格式的文件信息。releasedir时，将fi->fh置为-1。

5.3 open read write fsync release

这也是一套连续的调用，FUSE约定read、write、fsync在open之后。open同opendir，此外需要判断fi->flags中是否具有O_TRUNC标志，若具有需要先将文件内容清空（如echo "abc" > test）。fsync时，将块缓存管理器中所有属于当前文件的文件数据的数据块写回（我在BlockCache中设置了inode_id字段以标识属于的文件，便于查找）。注意这里只是同步数据，因为为了保证崩溃一致性，所有的元数据（间接索引块、磁盘Inode）都是修改时立即同步的。

5.4 create mkdir

创建文件/目录。除了创建的文件/目录的磁盘Inode的DiskInodeType不同，其它都是相同的。这里要在磁盘Inode中保存调用时的参数mode_t mode作为访问权限。

5.5 link

进行硬链接。在Linux中，硬链接不能对目录进行，这里也一样。硬链接可以使得两个目录项指向同一Inode编号。

5.6 unlink

删除硬链接，当硬链接数减少至0时，删除文件。在磁盘Inode中维护了硬链接数，初始为1，link时加1，unlink时减1。

5.7 rmdir

删除文件夹。只有当文件夹为空时，才能删除。当命令行中为rm添加-r参数递归删除时，FUSE会自动使用其它的调用获取子文件/子目录信息并先将它们删除，因此这样的实现是符合约定的。

5.8 rename

移动文件。有RENAME_EXCHANGE和RENAME_NOREPLACE两种选项，我这里没有判断flags，而是直接实现为RENAME_NOREPLACE，即目标路径存在时不进行覆盖，而是返回错误。

5.9 chmod chown

chmod更改文件/目录访问权限，chown更改文件/目录Owner的uid、gid。

6 效果演示

6.1 文件/目录读写

先不考虑权限问题，以root登录，运行./easyfs /disk 2 0 0 -f。

```
zhengkw@LAPTOP-K80Q9D95:/mnt/c/Users/zhengkw/Desktop$ cd /disk
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir a
zhengkw@LAPTOP-K80Q9D95:/disk$ touch b
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir a/a
zhengkw@LAPTOP-K80Q9D95:/disk$ touch a/b
zhengkw@LAPTOP-K80Q9D95:/disk$ ls
a  b
zhengkw@LAPTOP-K80Q9D95:/disk$ ls a
a  b
zhengkw@LAPTOP-K80Q9D95:/disk$ cd a
zhengkw@LAPTOP-K80Q9D95:/disk/a$ echo "abc" > b
zhengkw@LAPTOP-K80Q9D95:/disk/a$ echo "abc" >> b
zhengkw@LAPTOP-K80Q9D95:/disk/a$ cat b
abc
abc
zhengkw@LAPTOP-K80Q9D95:/disk/a$ echo "a" > b
zhengkw@LAPTOP-K80Q9D95:/disk/a$ cat b
a
zhengkw@LAPTOP-K80Q9D95:/disk/a$ rmdir a
zhengkw@LAPTOP-K80Q9D95:/disk/a$ ls
b
zhengkw@LAPTOP-K80Q9D95:/disk/a$ cd ..
zhengkw@LAPTOP-K80Q9D95:/disk$ rmdir a
rmdir: failed to remove 'a': Directory not empty
zhengkw@LAPTOP-K80Q9D95:/disk$ rm -r a
```

```
zhengkw@LAPTOP-K80Q9D95:/disk$ ls
b
zhengkw@LAPTOP-K80Q9D95:/disk$ stat b
  File: b
  Size: 0               Blocks: 0          IO Block: 512    regular empty file
Device: 2dh/45d Inode: 3           Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-18 22:32:16.000000000 +0800
Modify: 2021-06-18 22:32:16.-1409282771 +0800
Change: 2021-06-18 22:32:16.-1409279424 +0800
 Birth: -
zhengkw@LAPTOP-K80Q9D95:/disk$ echo "abcde" > b
zhengkw@LAPTOP-K80Q9D95:/disk$ stat b
  File: b
  Size: 6               Blocks: 1          IO Block: 512    regular file
Device: 2dh/45d Inode: 3           Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-18 22:32:16.000000000 +0800
Modify: 2021-06-18 22:36:20.-1409282771 +0800
Change: 2021-06-18 22:36:20.-1409283024 +0800
 Birth: -
```

上述演示了多级目录的创建、ls、echo、cat、rm -r、stat等。对于时间的维护，我并没有遵循Linux的Access、Modify、Change Time的定义，只是记录了两个时间，读/写文件分别更改访问/修改时间。

6.2 硬链接

```
zhengkw@LAPTOP-K80Q9D95:/disk$ rm -r *
zhengkw@LAPTOP-K80Q9D95:/disk$ touch a
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir b
zhengkw@LAPTOP-K80Q9D95:/disk$ link a b/a
zhengkw@LAPTOP-K80Q9D95:/disk$ ls b
a
zhengkw@LAPTOP-K80Q9D95:/disk$ echo "abc" > a
zhengkw@LAPTOP-K80Q9D95:/disk$ cat b/a
abc
zhengkw@LAPTOP-K80Q9D95:/disk$ stat a
  File: a
  Size: 4               Blocks: 1          IO Block: 512    regular file
Device: 2dh/45d Inode: 8           Links: 2
```

```
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-18 22:43:14.000000000 +0800
Modify: 2021-06-18 22:43:09.-1275060451 +0800
Change: 2021-06-18 22:43:09.000000001 +0800
 Birth: -
zhengkw@LAPTOP-K80Q9D95:/disk$ unlink b/a
zhengkw@LAPTOP-K80Q9D95:/disk$ stat a
  File: a
  Size: 4               Blocks: 1          IO Block: 512    regular file
Device: 2dh/45d Inode: 8           Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-18 22:43:14.000000000 +0800
Modify: 2021-06-18 22:43:09.-1275060451 +0800
Change: 2021-06-18 22:43:09.-1275061936 +0800
 Birth: -
zhengkw@LAPTOP-K80Q9D95:/disk$ unlink a
zhengkw@LAPTOP-K80Q9D95:/disk$ ls
b
```

删除硬链接时，元数据中记录的硬链接数减1，减为0时删除文件。

6.3 文件移动

```
zhengkw@LAPTOP-K80Q9D95:/disk$ rm -r *
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir a
zhengkw@LAPTOP-K80Q9D95:/disk$ touch a/a
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir a/b
zhengkw@LAPTOP-K80Q9D95:/disk$ echo "aaa" > a/a
zhengkw@LAPTOP-K80Q9D95:/disk$ mv a/a a/b/a
zhengkw@LAPTOP-K80Q9D95:/disk$ ls a
b
zhengkw@LAPTOP-K80Q9D95:/disk$ ls a/b
a
zhengkw@LAPTOP-K80Q9D95:/disk$ cat a/b/a
aaa
```

6.4 多用户权限

首先以uid=0，gid=0登录

```
zhengkw@LAPTOP-K80Q9D95:/disk$ mkdir a
```

```
zhengkw@LAPTOP-K80Q9D95:/disk$ touch a/a
zhengkw@LAPTOP-K80Q9D95:/disk$ chmod 750 a
zhengkw@LAPTOP-K80Q9D95:/disk$ stat a
  File: a
  Size: 32              Blocks: 1          IO Block: 512    directory
Device: 2dh/45d Inode: 5           Links: 1
Access: (0750/drwxr-x---)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-18 22:54:35.000000000 +0800
Modify: 2021-06-18 22:54:35.000006189 +0800
Change: 2021-06-18 22:54:35.000005792 +0800
 Birth: -
```

将文件权限改为允许同gid读/执行，uid、gid都不同则无任何权限。以uid=1，gid=0登录

```
zhengkw@LAPTOP-K80Q9D95:/disk$ ls a
a
zhengkw@LAPTOP-K80Q9D95:/disk$ touch a/b
touch: cannot touch 'a/b': Permission denied
```

可以查看文件夹a，但不能在其中创建文件。

以uid=2，gid=1登录

```
zhengkw@LAPTOP-K80Q9D95:/disk$ ls a
ls: cannot access 'a': Permission denied
```

查看的权限也没有了。我们换回uid=0，gid=0，使用chown更改所有者为uid=2，gid=1

```
zhengkw@LAPTOP-K80Q9D95:/disk$ chown 2:1 a
zhengkw@LAPTOP-K80Q9D95:/disk$ stat a
  File: a
  Size: 32              Blocks: 1          IO Block: 512    directory
Device: 2eh/46d Inode: 2           Links: 1
Access: (0750/drwxr-x---)  Uid: (    2/     bin)   Gid: (    1/  daemon)
Access: 2021-06-18 22:57:54.000000000 +0800
Modify: 2021-06-18 22:54:35.-134214355 +0800
Change: 2021-06-18 22:54:35.-134213152 +0800
 Birth: -
```

可以看到，所有者发生了变化。此时再以uid=2，gid=1登录，权限限制已经被解除了。

7 并发性能

为测试并发读写的性能，我尝试了不同的块设备数量、线程数、读写规模，并将直接面向文件

系统的吞吐率和使用FUSE挂载后的吞吐率进行对比。如先前所述，这里的并发读写指的是对磁盘Inode位于不同块的文件进行，对同一个文件无法做到并发。

7.1 低读写规模+频繁读写

这个测试中，进行100000次写操作，但每次只写512字节。

表 1: 面向文件系统的吞吐率

吞吐率（MB/s） 线程数 \ 块设备数量	1	4
1	77	73
2	29	147
3	27	202
4	25	257

表 2: FUSE挂载后的吞吐率

吞吐率（MB/s） 线程数 \ 块设备数量	1	4
1	6	7
2	14	15
3	18	19
4	19	24

在这种参数设置下，面向文件系统时，多个块设备能明显提高并发性，使得吞吐率随线程数近似线性增长。这是由于操作的数据块很少，块缓存命中率很高，块设备的读写成为性能瓶颈，多个块设备能被有效利用。单个块设备时，由于锁的开销，多线程吞吐率远不如单线程。

FUSE挂载后，即使在4个块设备时，吞吐率也低的惊人。由于写操作很多、每次写的数据很少，每进行一次写操作，FUSE都需要用户态-内核态的切换，占据了主要开销。

7.2 高读写规模

这个测试中，进行1000次写操作，每次写8MB。

表 3: 面向文件系统的吞吐率

吞吐率（MB/s） 线程数 \ 块设备数量	1	4
1	598	633
2	425	418
3	495	487

表 4: FUSE挂载后的吞吐率

吞吐率（MB/s） 线程数 \ 块设备数量	1	4
1	301	361
2	369	364
3	409	410

在这种参数设置下，块设备的数量对多线程的吞吐率反而影响不大了。这应该是由于每次写的数据很多，多线程会导致缓存出现频繁的冲突缺失，块设备的锁反而不是瓶颈了。

同时，FUSE挂载后吞吐率降低并不多，因为写操作次数较少，与每次8MB的读写量相比，用户态-内核态切换开销占比降低。

8 崩溃一致性

这里的崩溃一致性是指元数据的崩溃 致性，也就是元数据是自洽的，如磁盘Inode指向的数据块在位图中对应位必然为1、目录项中指向的Inode编号必然有效。无法保证崩溃时写操作的数据写入完毕等，也不可避免地会出现孤儿数据块。

在我的实现中，要保证崩溃一致性，那么所有元数据必然在修改后立即同步。否则，由于无法控制块缓存换出的时机，可能在崩溃时，一部分元数据写回了，一部分没有写回，这完全是无法预料和操控的。只有修改后立即同步，设计元数据修改的顺序才有意义。

下面以两个例子来说明崩溃一致性的保证。

8.1 创建文件/目录

1. 分配、初始化新文件的Inode，写回
2. 在目录的Inode中添加目录项

3. 将目录的Inode写回

由于新文件分配在目录Inode写回之前，可能出现孤儿Inode，但目录的目录项必然指向合法Inode。

8.2 文件写时扩容

1. 计算扩容量，在位图中分配数据块，填充间接索引

2. 修改磁盘Inode中的‘size‘字段、直接/间接索引字段等

由于`size`是最后改变的，若在第二步前崩溃，那么会出现很多孤儿数据块，但读Inode时由于文件大小没有增加，即使间接索引只填充了一半，也不会被用到。

9 小组分工与情况说明

我独自完成了所有工作。

非常遗憾，由于网络学堂DDL进行过变动，我对非毕业生提交作业的截止时间产生了误解和记忆偏差，因此时间安排上错过了DDL。我本来是要和队友2人组队完成，但DDL时我几乎没有做任何工作，队友则先写了一个baseline并以个人的名义提交了，于是商量后决定取消组队，我个人进行补交。对此给队友和助教带来的不便，我感到十分自责、愧疚和抱歉!