HDD 和 SSD 混合文件系统

欧阳浩岚 22214379

一、概述

本次大作业我实现的是一个基于 fuse 的 HDD 和 SSD 混合文件系统，总体的框架基于 ext2 文件系统，并在此基础上增加了对用户透明的 HDD 与 SSD 混合存储架构。

以下是本文件系统里支持的操作：

```
static struct fuse_operations fs_ops = {
  .getattr = fs_getattr,
  .mknod = fs_mknod,
  .mkdir = fs_mkdir,
  .unlink = fs_unlink,
  .rmdir = fs_rmdir,
  .open = fs_open,
  .read = fs_read,
  .write = fs_write,
  .readdir = fs_readdir,
  .init = fs_init,
};
```

根据这些操作函数的要求，我实现了以下模块：

- 磁盘管理
- 文件系统元数据管理
- Inode 管理
- 缓存管理

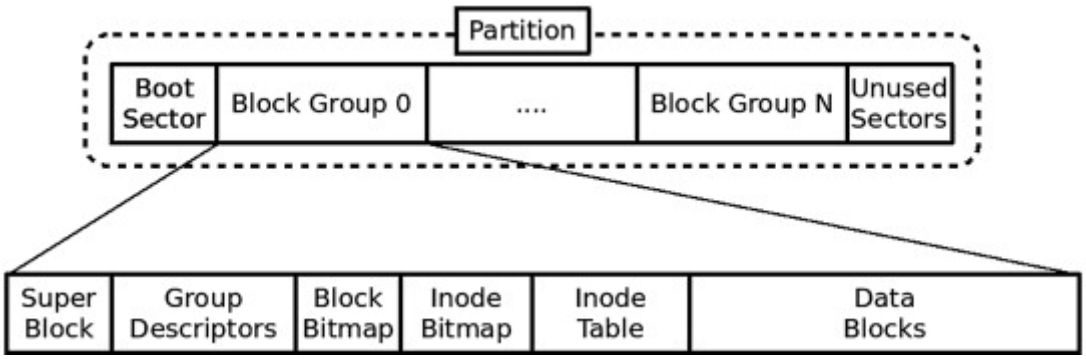
在具体代码的实现中，我参考了 ext4fuse 的部分代码，这是一个利用 C 语言实现的基于 fuse 框架的 ext4 文件系统。然而，它的代码只实现了一个基于单磁盘的只读文件系统。本文件系统利用 C++ 语言实现了一个支持文件增删查改的混合磁盘 ext2 文件系统。

二、文件系统总体框架

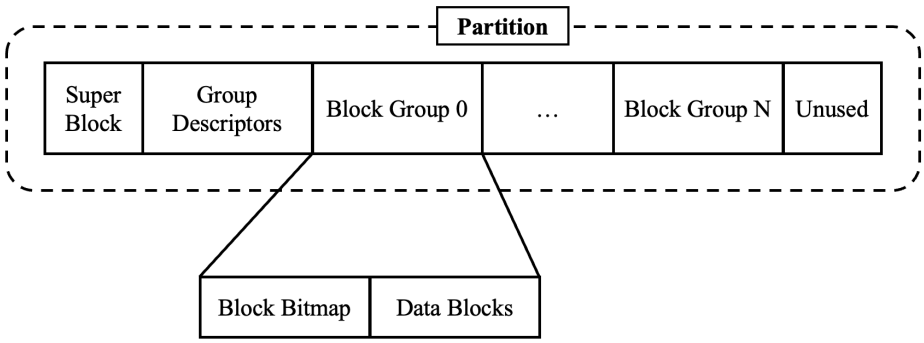
文件系统在磁盘上的布局基本与 ext2 文件系统相同。考虑到磁盘里的元数据例如 Super Block、GDT、Inode 表都是文件系统里频繁访问的，我们把这部分元数据全部放入 SSD 以提高文件系统的速

度。HDD 主要是放置数据块，并利用位图 （bitmap） 来查找空闲块。

SSD 上的布局：



HDD 上的布局：

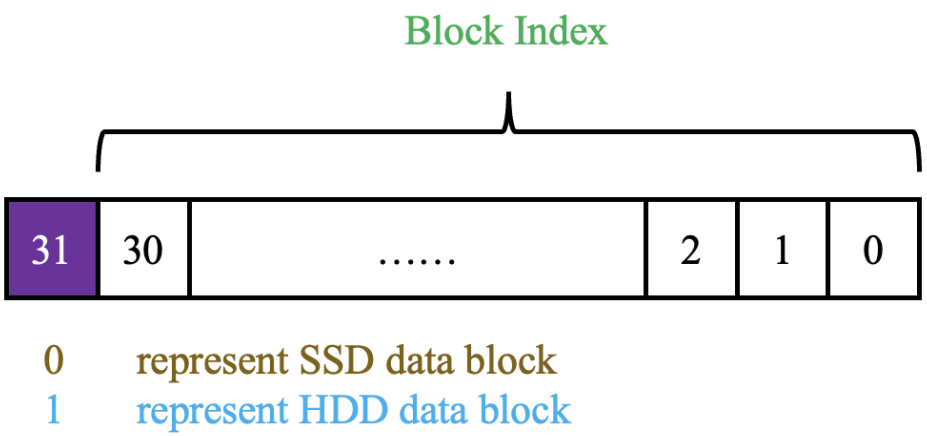


其中，Super Block 和 GDT 也被算到 Group 0 里，被 Block Bitmap 记录。

1.磁盘管理

1.1 统一的块地址

为了统一 SSD 和 HDD 的块寻址，本文件系统使用统一的块地址编码方式：



在寻址时，只需通过最高位的掩码即可得到寻址块所在设备，即：

```
#define HDD_MASK ((uint32_t)1 << 31)
#define IN_HDD(__blk_idx) ((__blk_idx & HDD_MASK) != 0)
#define HDD_BLOCK_IDX(__blk_idx) (__blk_idx | HDD_MASK)
#define UN_HDD_MASK(__blk_idx) (__blk_idx &(~HDD_MASK))

ssize_t DiskManager::disk_block_read(void *buf, size_t nbyte, uint32_t pblock)
{
  if (IN_HDD(pblock)) {
    pblock = UN_HDD_MASK(pblock);
    return hdd_disk_read(buf, pblock);
  } else {
    return ssd_disk_read(buf, pblock);
  }
}
```

这里，我们把所有的路径文件以及用于间接寻址的数据块都放在 SSD 里，普通文件**超过阈值的数据块**放在 **HDD** 里。

1.2 磁盘读写

由于磁盘内部也是一个线性的地址，Block n 的开始地址计算方式为：

$$\mathbf{block\_n\_offset} = \mathbf{n} * \mathbf{block\_size}$$

这里以 SSD 的块读取的代码作为说明：

```
#define BLOCKS2BYTES(__blks) ((uint64_t)(__blks)*block_size_)

ssize_t DiskManager::ssd_disk_block_read(void *buf, uint64_t block_idx) {
  assert(block_size_ > 0);

  off_t offset = BLOCKS2BYTES(block_idx);
  return ssd_disk_read(buf, block_size_, offset);
}
```

2.文件系统元数据管理

文件系统里所有的元数据都是参考 ext4 wiki ，Super Block 里的结构：

```
struct ext4_super_block {
  ...
```

```
    __le32  s_blocks_count_lo;  /* Blocks count */
    __le32  s_log_block_size; /* Block size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le16  s_inode_size;    /* size of inode structure */
    ...
};
```

Group Descriptor 里的结构：

```
struct ext4_group_desc
{
    __le32  bg_block_bitmap_lo; /* Blocks bitmap block */
    __le32  bg_inode_bitmap_lo; /* Inodes bitmap block */
    __le32  bg_inode_table_lo;  /* Inodes table block */
    __le16  bg_free_blocks_count_lo;/* Free blocks count */
    __le16  bg_free_inodes_count_lo;/* Free inodes count */
    ...
};
```

3.Inode 管理

3.1 数据结构

由于文件系统暂时还没有实现时间记录模块和权限管理，`uid`、`gid`、`atime`、`ctime`、`mtime` 等都被忽略了，Inode 的结构如下：

```
struct ext4_inode {
    ...
    __le16  i_mode;    /* File mode */
    __le32  i_size_lo;  /* Size in bytes */
    __le32  i_blocks_lo;  /* Blocks count */
    __le32  i_flags;  /* File flags */
    __le32  i_block[EXT4_N_BLOCKS];/* Pointers to blocks */
    ...
};
```

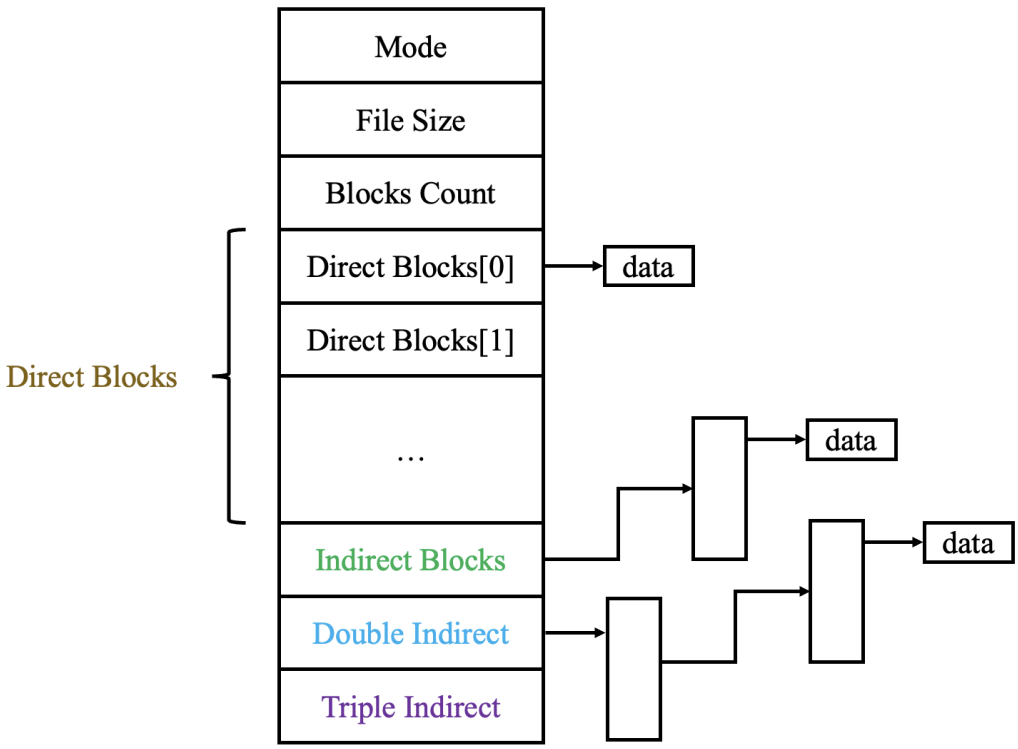
目录文件的结构：

```
struct ext4_dir_entry_2 {
    __le32  inode;          /* Inode number */
    __le16  rec_len;        /* Directory entry length */
    __u8    name_len;       /* Name length */
    __u8    file_type;
    char    name[EXT4_NAME_LEN];    /* File name */
};
```

3.2 数据块

3.2.1 寻址方式

下图是 Inode 结构：



目前本文件系统还未实现 extent 特性，故数据块的寻址还是采用间接寻址。具体的实现方式是：给定一个文件的逻辑块地址 lblock ，取值范围从 0 到 文件最后一个目录块，文件数据块里存储着文件块的实际地址 pblock。

```
uint32_t InodeManager::get_data_pblock(const ext4_inode &inode,
                                       uint32_t lblock) {
  if (lblock < EXT4_NDIR_BLOCKS) { // direct data block
    return inode.i_block[lblock];
  } else if (lblock < MAX_IND_BLOCK) {
    uint32_t index_block = inode.i_block[EXT4_IND_BLOCK];
    return get_data_pblock_ind(lblock - EXT4_NDIR_BLOCKS, index_block);
```

```
  } else if (lblock < MAX_DIND_BLOCK) {
    uint32_t dindex_block = inode.i_block[EXT4_DIND_BLOCK];
    return get_data_pblock_dind(lblock - MAX_IND_BLOCK, dindex_block);
  } else if (lblock < MAX_TIND_BLOCK) {
    uint32_t tindex_block = inode.i_block[EXT4_TIND_BLOCK];
    return get_data_pblock_tind(lblock - MAX_DIND_BLOCK, tindex_block);
  } else {
    LOG(FATAL) << "lblock exceed max data block size";
    return 0;
  }
}
```

间接寻址方式就是一个递归，这里以 Double Indirect 作为示例代码：

```
uint32_t InodeManager::get_data_pblock_dind(uint32_t lblock,
                                            uint32_t dindex_pblock) {
  if (dindex_pblock == 0)
    return 0;

  uint32_t index_pblock;
  uint32_t index_block_in_dind_offset =
      (lblock / MAX_IND_BLOCK) * sizeof(uint32_t);
  GET_INSTANCE(DiskManager)
      .disk_read(&index_pblock, sizeof(uint32_t), dindex_pblock,
                 index_block_in_dind_offset);

  lblock %= MAX_IND_BLOCK; // calculate index in index block
  return get_data_pblock_ind(lblock, index_pblock);
}
```

3.2.2 申请新的数据块

为了将大的数据块放到 HDD 里，普通文件在申请

3.3 目录项

目录结构如下：

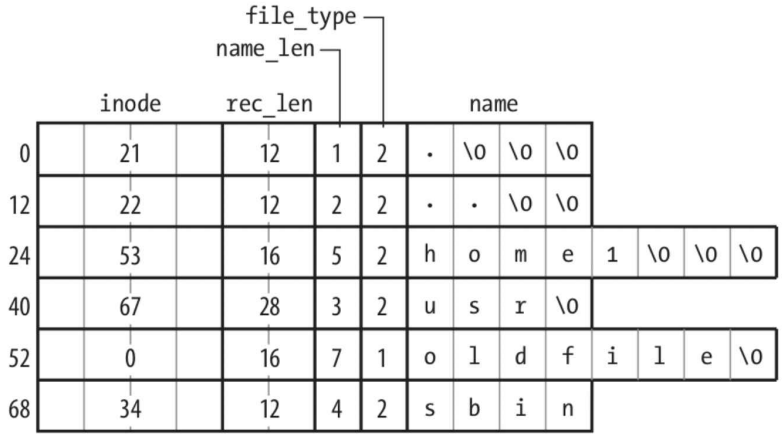


***Figure 3** Example of an Ext2 directory*

3.3.1 遍历方式

由于目录项不是等长的，需要通过上一项的 rec_len 来计算下一个目录项位置：

```
while ((dentry = get_dentry(prefix_inode, offset, dir_ctx)) != nullptr) {
  offset += dentry->rec_len; // get next entry

  // ignore deleted file
  if (dentry->inode == 0)
    continue;

  std::string cur_filename(dentry->name, (size_t)dentry->name_len);

  // deal with "." and ".."
  if (cur_filename == "." || cur_filename == "..") {
    continue;
  }

  // Record each valid entry
  LOG(INFO) << dentry_str(*dentry);
}
```

3.3.2 删除目录项

这里用一个例子来说明具体实现方式，假设当前目录文件如前面的目录图示，目录文件里的内容是

```
{inode = 21, rec_len = 12, name_len = 1, file type = 2, name "."}
{inode = 22, rec_len = 12, name_len = 2, file type = 2, name ".."}
{inode = 53, rec_len = 16, name_len = 5, file type = 2, name "homei"}
{inode = 67, rec_len = 28, name_len = 3, file type = 2, name "usr"}
{inode = 34, rec_len = 12, name_len = 4, file type = 2, name "sbin"}
```

图中偏移量为 52 的 oldfile 被删除了，它原来占的 16 个字节都被添加到 usr 项中，故 usr 的 rec_len 是 28，同时 oldfile 的 inode 也被置为 0。

3.3.3 添加目录项

还是上面的例子，这里添加一个 newdirectory 的目录项到上面的目录里，得到：

```
{inode = 21, rec_len = 12, name_len = 1,  file_type = 2, name = "."}
{inode = 22, rec_len = 12, name_len = 2,  file_type = 2, name = ".."}
{inode = 53, rec_len = 16, name_len = 5,  file_type = 2, name = "homei"}
{inode = 67, rec_len = 28, name_len = 3,  file_type = 2, name = "usr"}
{inode = 34, rec_len = 12, name_len = 4,  file_type = 2, name = "sbin"}
{inode = 68, rec_len = 80, name_len = 12, file_type = 2, name = "newdirectory"}
```

这里原因是，该项需要 20 个字节，而上面所有的 rec_len 都不足以给出足够的空间，当前目录只能申请一个新的 80 字节的数据块来进行存放。（虽然 usr 的 rec_len 有 28，但是存放 usr 和 newdirectory 两个目录项所需的最小字节数为 32）

这里再添加一个 newfile 目录项，得到

```
{inode = 21, rec_len = 12, name_len = 1,  file_type = 2, name = "."}
{inode = 22, rec_len = 12, name_len = 2,  file_type = 2, name = ".."}
{inode = 53, rec_len = 16, name_len = 5,  file_type = 2, name = "homei"}
{inode = 67, rec_len = 12, name_len = 3,  file_type = 2, name = "usr"}
{inode = 69, rec_len = 16, name_len = 7,  file_type = 1, name = "newfile"}
{inode = 34, rec_len = 12, name_len = 4,  file_type = 2, name = "sbin"}
{inode = 68, rec_len = 80, name_len = 12, file_type = 2, name = "newdirectory"}
```

4. 缓存管理

目前只实现了**文件路径缓存**，这里实现的是一个基于内存的哈希大表。由于文件名可能会重复，本文件系统利用父目录的 Inode 作为标识。具体的索引计算是：

$$\texttt{file\_index} = \texttt{\#\$\{parent\_id\}file\_name}$$

假如有个路径 "/dir1/dir2/dir3/filename"，假设 "/dir1/dir2/dir3" 的 Inode 号是 15，那么这个路径的哈希索引是 `#15filename`。

5.管理类实现

文件系统里每个模块都使用类实现，然而每个模块都只需要维护一个类实体，故这些管理类都采用设计模式里的单例模式进行实现。单例模式的实现方式如下：

```
#define GET_INSTANCE(X) X::get_instance()

class Singleton {
public:
    static Singleton& get_instance() {
        static Singleton instance;
        return instance;
    }

    void do_something();

private:
    Singleton() {}
};
```

在调用这个类时，使用代码

```
GET_INSTANCE(Singleton).do_something();
```

三、文件系统操作具体实现

1.初始化

文件系统初始化主要要把磁盘里的元数据读到相应模块的内存里，这里分别是：

- 文件系统的 Super Block
- SSD 里的 Group Descriptor 表
- HDD 里的元数据
- 初始化 Inode 模块

```
void *fs_init(fuse_conn_info *conn, fuse_config *cfg) {
  (void)cfg;

  // fill in super block
  GET_INSTANCE(MetaDataManager).super_block_fill();
```

```
6
7   // fill in gdt
8   GET_INSTANCE(MetaDataManager).gdt_fill();
9
10  // initialize hdd disk
11  GET_INSTANCE(MetaDataManager).hdd_disk_init();
12
13  // Initialize root inode
14  GET_INSTANCE(InodeManager).init();
15  return NULL;
16 }
```

2.打开文件

`open` 函数主要是在 `read` 和 `write` 函数调用前使用，主要作用：

- 判断文件是否存在
- 缓存相应的 Inode 号。

```
1  int fs_open(const char *path, fuse_file_info *fi) {
2    LOG(INFO) << "Open file: " << path;
3
4    uint32_t inode_num = GET_INSTANCE(InodeManager).get_idx_by_path(path);
5    if (inode_num == 0)
6      return -ENOENT;
7    fi->fh = inode_num;
8    LOG(INFO) << "Open " << path << " in inode: #" << fi->fh;
9
10   return 0;
11 }
```

3.创建文件

创建文件的流程：

- 将给定的路径 "/parent_dir/filename" 拆分成父目录 /parent_dir 和文件名 filename
- 为 filename 申请一个新的 Inode 项
- 在父目录文件 /parent_dir 里添加 filename 目录项

目录文件的创建相似，会增加以下两步：

- 申请一个新的数据块
- 在其中填写 `.` 和 `..` 两项

```
int fs_mknod(const char *path_cstr, mode_t mode, dev_t rdev) {
  std::string parent_path, filename;

  // get path's parent directory
  // parent directory does exist ensure by the callee
  get_parent_dir(path_cstr, parent_path, filename);
  LOG(INFO) << "parent directory: " << parent_path << " dirname: " << filename;

  if (filename.size() > EXT4_NAME_LEN)
    return -ENAMETOOLONG;

  int ret;
  uint32_t parent_inode_idx, cur_inode_idx;
  ext4_inode prefix_inode, cur_inode;
  parent_inode_idx = GET_INSTANCE(InodeManager).get_idx_by_path(parent_path);
  GET_INSTANCE(InodeManager).get_inode_by_idx(parent_inode_idx, prefix_inode);

  // Create new file
  // Allocate inode for new file
  cur_inode_idx = GET_INSTANCE(MetaDataManager).get_new_inode_idx();

  // Initialize new file inode
  memset(&cur_inode, 0, sizeof(ext4_inode));
  uint16_t i_mode = mode;
  cur_inode = {
      .i_mode = i_mode,
  };

  // Update on-disk parent_inode file content
  ext4_dir_entry_2 cur_dentry;
  set_dir_dentry(cur_dentry, cur_inode_idx, filename);
  GET_INSTANCE(InodeManager).add_dentry(prefix_inode, cur_dentry);

  // Update on-disk Inode table
  GET_INSTANCE(InodeManager).update_disk_inode(cur_inode_idx, cur_inode);
  GET_INSTANCE(InodeManager).update_disk_inode(parent_inode_idx, prefix_inode);

  return 0;
}
```

4.普通文件读取

这部分要注意两点：

- 文件按块存放的，读取的地址要与 block_size 对齐
- 文件要一块一块的读，因为文件的数据块可能不是连续存放

```
int fs_read(const char *path, char *buf, size_t size, off_t offset,
          fuse_file_info *fi) {
  assert(offset >= 0);

  // Avoid access WRONLY file
  if (((fi->flags & O_ACCMODE) == O_WRONLY))
      return -EACCES;

  size_t bytes;
  size_t ret = 0;
  size_t un_offset = (size_t)offset;
  uint32_t block_size = GET_INSTANCE(MetaDataManager).block_size();
  ext4_inode inode;

  int get_inode_ret =
      GET_INSTANCE(InodeManager).get_inode_by_idx(fi->fh, inode);
  if (get_inode_ret < 0) {
    return get_inode_ret;
  }

  // truncate size
  size = truncate_size(inode, size, offset);

  // read the first block and doing the alignment
  bytes = first_read(inode, buf, size, offset);

  ret = bytes;
  buf += bytes;
  un_offset += bytes;
  for (uint32_t lblock = un_offset / block_size; size > ret; lblock++) {
    bytes = (size - ret) > block_size ? block_size : size - ret;

    uint64_t pblock = GET_INSTANCE(InodeManager).get_data_pblock(inode, lblock);
    if (pblock) {
      GET_INSTANCE(DiskManager).disk_read(buf, bytes, pblock, 0);
    } else {  // deal with sparse file
      memset(buf, 0, bytes);
      LOG(INFO) << "Sparse file, skipping " << bytes << " bytes";
    }
```

```
    ret += bytes;
    buf += bytes;
  }
  assert(size == ret);
  return ret;
}
```

5.目录文件读取

目录文件的思路与目录项遍历一模一样，这里与普通文件读取的区别是：读取目录文件需要将目录项一个个加入 `filler` 函数。

```
int fs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset
            fuse_file_info *fi, fuse_readdir_flags flags) {

  (void)fi;
  (void)offset;
  (void)flags;

  struct stat st;
  ext4_inode inode;
  fuse_fill_dir_flags fill_flags = FUSE_FILL_DIR_PLUS;  // fix Input/output errc

  uint32_t block_size = GET_INSTANCE(MetaDataManager).block_size();
  DirCtx dir_ctx(block_size);

  int ret = GET_INSTANCE(InodeManager).get_inode_by_path(path, inode);
  if (ret < 0)
    return ret;

  off_t dentry_off = 0;
  ext4_dir_entry_2 *dentry = nullptr;
  while ((dentry =
              GET_INSTANCE(InodeManager).get_dentry(inode, dentry_off, dir_ctx))
         nullptr) {
    dentry_off += dentry->rec_len;

    if (dentry->inode == 0)
      continue;

    // since cfg->use_ino is not set, no need to set this
    // st.st_ino = dentry->inode;
    st.st_mode = inode.i_mode;
```

```
    std::string filename(dentry->name, (size_t)dentry->name_len);

    if (filler(buf, filename.c_str(), &st, 0, fill_flags))
      break;
  }

  return 0;
}
```

6.文件写入

写入的流程与读取差不多，但在查找相应数据块时，可能出现查找错误——该数据块不存在，需要从磁盘里申请一个新的数据块。

```
int fs_write(const char *path, const char *buf, size_t size, off_t offset,
             struct fuse_file_info *fi) {
  assert(offset >= 0);

  uint32_t inode_idx;
  ext4_inode inode;
  if (fi) {
    if (((fi->flags & O_ACCMODE) == O_RDONLY))
      return -EACCES;
    inode_idx = fi->fh;
  } else {
    inode_idx= GET_INSTANCE(InodeManager).get_idx_by_path(path);
  }

  int get_inode_ret = GET_INSTANCE(InodeManager).get_inode_by_idx(inode_idx, ino
  if (get_inode_ret < 0) {
    return get_inode_ret;
  }

  size_t bytes;
  size_t ret = 0;
  size_t un_offset = (size_t)offset;
  uint32_t block_size = GET_INSTANCE(MetaDataManager).block_size();

  // write the first block and doing the alignment
  bytes = first_write(inode, buf, size, offset);

  ret = bytes;
  buf += bytes;
```

```
  un_offset += bytes;
  for (uint32_t lblock = un_offset / block_size; size > ret; lblock++) {
    uint64_t pblock = GET_INSTANCE(InodeManager).get_data_pblock(inode, lblock);
    if (pblock == 0) {  // fill in empty block
      pblock = GET_INSTANCE(MetaDataManager).alloc_new_pblock(lblock);
      GET_INSTANCE(InodeManager).set_data_pblock(inode, lblock, pblock);
    }

    bytes = (size - ret) > block_size ? block_size : size - ret;
    GET_INSTANCE(DiskManager).disk_write(buf, bytes, pblock, 0);
    LOG(INFO) << "Write " << bytes << " to block #" << pblock;

    ret += bytes;
    buf += bytes;
  }

  assert(size == ret);

  // Set new file stat to inode
  uint64_t file_size = GET_INSTANCE(InodeManager).get_file_size(inode);
  if ((uint64_t)offset + size > file_size) {
    GET_INSTANCE(InodeManager).set_file_size(inode, (size_t)offset + size);
  }
  GET_INSTANCE(InodeManager).update_disk_inode(inode_idx, inode);

  return ret;
}
```

7.目录文件删除

普通文件删除和目录文件删除相似，但目录文件删除更复杂，这里直接介绍目录文件的删除。目录文件的删除包含：

- 删除当前目录在父目录的目录项
- 删除目录下所有文件
- 当前目录文件

删除当前目录文件下的所有文件可以通过递归实现：

```
void InodeManager::rm_dir(ext4_inode &cur_inode, uint32_t cur_inode_idx) {
  off_t dentry_off = 0;
  uint32_t block_size = GET_INSTANCE(MetaDataManager).block_size();
  DirCtx dir_ctx(block_size);
```

```
5
6   // ignore . and ..
7   ext4_dir_entry_2 *dentry = get_dentry(cur_inode, dentry_off, dir_ctx);
8   dentry_off += dentry->rec_len;
9   dentry = GET_INSTANCE(InodeManager).get_dentry(cur_inode, dentry_off, dir_ctx)
10  dentry_off += dentry->rec_len;
11
12  // iter the whole directory file
13  while ((dentry =
14              GET_INSTANCE(InodeManager).get_dentry(cur_inode, dentry_off, dir_c
15          nullptr) {
16    dentry_off += dentry->rec_len;
17
18    if (dentry->inode == 0)
19      continue;
20
21    ext4_inode iter_inode;
22    get_inode_by_idx(dentry->inode, iter_inode);
23
24    // recursively delete file
25    if ((dentry->file_type & 0x2) != 0) {
26      rm_dir(iter_inode, dentry->inode);
27    } else {
28      rm_file(iter_inode, dentry->inode);
29    }
30  }
31 }
```

完整的实现如下：

```
1  int fs_rmdir(const char *path) {
2    LOG(INFO) << "Rmdir begin:";
3    LOG(INFO) << "rmdir( " << path  << " )";
4    std::string parent_path, dirname;
5
6    // get path's parent directory
7    // parent directory does exist ensure by the callee
8    get_parent_dir(path, parent_path, dirname);
9    LOG(INFO) << "parent directory: " << parent_path << " dirname: " << dirname;
10
11   uint32_t parent_inode_idx, cur_inode_idx;
12   ext4_inode prefix_inode, cur_inode;
13   cur_inode_idx = GET_INSTANCE(InodeManager).get_idx_by_path(path);
14   GET_INSTANCE(InodeManager).get_inode_by_idx(cur_inode_idx, cur_inode);
15   GET_INSTANCE(InodeManager).get_inode_by_path(parent_path, prefix_inode);
```

```
16
17    // delete dentry
18    GET_INSTANCE(InodeManager).rm_dentry(prefix_inode, cur_inode_idx);
19
20    // delete all the file under this directory (include itself)
21    GET_INSTANCE(InodeManager).rm_dir(cur_inode, cur_inode_idx);
22    return 0;
23  }
```

四、实验

1.安装 fuse

1. 编译命令：

```
1 git clone https://github.com/libfuse/libfuse.git; cd libfuse
2 mkdir build; cd build
3 meson setup ..
```

2. 安装命令：

```
1 ninja
2 sudo ninja install
```

3. 查看命令：

```
1 pkg-config --list-all | grep fuse
```

```
# ouyhlan @ lsmkv in ~ [9:32:22]
$ pkg-config --list-all | grep fuse
fuse3                   fuse3 - Filesystem in Userspace
```

4. 将 fuse 以库的形式引入 CMakeList.txt

```
1 # pkg-config
```

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(fuse REQUIRED IMPORTED_TARGET fuse3)

# add the executable
add_executable(Hybrid-Fs ${SOURCES})
target_link_libraries(Hybrid-Fs PRIVATE PkgConfig::fuse)
```

2.文件系统运行结果

1. 创建磁盘文件

这里使用 `dd` 命令在 HDD 和 SSD 两个磁盘中创建两个空的 128 MB 大的文件用作对文件系统盘。这里我在 HDD 创建一个文件叫 `fs.W4aX` ，在 SSD 创建文件 `fs.0Kgt`：

```
dd if=/dev/zero of=/home/ouyhlan/fs/fs.W4aX bs=$((1024*1024)) count=128 &> /dev/
dd if=/dev/zero of=/mnt/nvme/fs/fs.0Kgt bs=$((1024*1024)) count=128 &> /dev/null
```

2. 创建文件系统

使用 `mke2fs` 命令在 SSD 的磁盘文件 `fs.0Kgt` 上创建 ext2 文件系统：

```
mke2fs -F -t ext2 /mnt/nvme/fs/fs.0Kgt &> /dev/null
```

3. 初始化文件系统

使用 `debugfs` 命令在 SSD 的磁盘文件 `fs.0Kgt` 上提前创建几个目录文件和文本文件用来测试文件系统的读取功能：

```
debugfs fs/fs.0Kgt -w
mkdir dir1
mkdir dir1/dir2
write /mnt/nvme/fs/1.txt dir1/1.txt
```

```
# ouyhlan @ lsmkv in /mnt/nvme [11:39:47]
$ debugfs fs/fs.0Kgt -w
debugfs 1.46.5 (30-Dec-2021)
debugfs:  mkdir dir1
debugfs:  mkdir dir1/dir2
debugfs:  write /mnt/nvme/fs/1.txt dir1/1.txt
Allocated inode: 14
```

4. 运行程序

把 `/mnt/nvme/test` 作为 fuse 挂载点，使用命令：

```
build/Hybrid-Fs -f /mnt/nvme/test --hdd_filename=/home/ouyhlan/fs/fs.W4aX \
--ssd_filename=/mnt/nvme/fs/fs.0Kgt
```

5. 测试 `readdir` 函数：

```
cd test
tree .
```

```
# ouyhlan @ lsmkv in /mnt/nvme [11:45:00] C:1
$ cd test

# ouyhlan @ lsmkv in /mnt/nvme/test [11:45:17]
$ tree .
.
├── dir1
│   ├── 1.txt
│   └── dir2
└── lost+found

3 directories, 1 file
```

刚刚设置的三个文件都在文件系统里

6. 测试 `read` 函数：

使用 `cat` 命令读取 `1.txt`：

```
cat 1.txt
```

```
# ouyhlan @ lsmkv in /mnt/nvme/test [11:45:21]
$ cd dir1

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:48:20]
$ cat 1.txt
hello world
```

7. 测试 `mknod` 和 `write` 函数：

```
1 echo "filesystem" > 2.txt
2 echo "apple pie" >> 1.txt
3 cat 2.txt
4 cat 1.txt
```

```
# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:48:23]
$ echo "filesystem" > 2.txt

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:49:52]
$ echo "apple pie" >> 1.txt

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:50:03]
$ cat 2.txt
filesystem

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:50:07]
$ cat 1.txt
hello world
apple pie
```

8. 测试 `mkdir` 函数：

```
1 mkdir dir3
2 mkdir ../dir4
```

```
# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:50:10]
$ mkdir dir3

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:53:02]
$ mkdir ../dir4

# ouyhlan @ lsmkv in /mnt/nvme/test/dir1 [11:53:06]
$ cd ..

# ouyhlan @ lsmkv in /mnt/nvme/test [11:53:09]
$ tree
.
├── dir1
│   ├── 1.txt
│   ├── 2.txt
│   ├── dir2
│   └── dir3
├── dir4
└── lost+found
```

9. 测试 `unlink` 和 `rmdir` 函数：

```
rm dir1/1.txt
rmdir dir1
```

```
# ouyhlan @ lsmkv in /mnt/nvme/test [12:27:11]
$ rm dir1/1.txt

# ouyhlan @ lsmkv in /mnt/nvme/test [12:27:16]
$ tree
.
├── dir1
│   ├── 2.txt
│   ├── dir2
│   └── dir3
├── dir4
└── lost+found

5 directories, 1 file

# ouyhlan @ lsmkv in /mnt/nvme/test [12:27:17]
$ rmdir dir1

# ouyhlan @ lsmkv in /mnt/nvme/test [12:27:23]
$ tree
.
├── dir4
└── lost+found

2 directories, 0 files
```

10. 测试申请 HDD 数据块资源

由于本文件系统实现的是用户透明的混合文件系统，我们采用打日志的方式去记录文件写入前后 HDD 空闲数据块数量的情况。同时，我们设置 1MB 作为使用 HDD 数据块的阈值，利用 `dd` 命令申请 2MB 和 4 MB 的文件：

```
dd if=/dev/zero of=/mnt/nvme/test/dir4/big1 bs=$((1024*1024)) count=2 &> /dev/nu
dd if=/dev/zero of=/mnt/nvme/test/dir4/big2 bs=$((1024*1024)) count=4 &> /dev/nu
```

```
# ouyhlan @ lsmkv in /mnt/nvme/test [12:50:39]
$ dd if=/dev/zero of=/mnt/nvme/test/dir4/big1 bs=$((1024*1024)) count=2 &> /dev/null

# ouyhlan @ lsmkv in /mnt/nvme/test [12:50:51]
$ dd if=/dev/zero of=/mnt/nvme/test/dir4/big2 bs=$((1024*1024)) count=4 &> /dev/null

# ouyhlan @ lsmkv in /mnt/nvme/test [12:50:58]
$ ll
total 20K
drwxr-xr-x 2 root root 4.0K Jun 30 12:50 dir4
drwx------ 2 root root  16K Jun 30 12:50 lost+found

# ouyhlan @ lsmkv in /mnt/nvme/test [12:51:01]
$ cd dir4

# ouyhlan @ lsmkv in /mnt/nvme/test/dir4 [12:51:06]
$ ll
total 6.0M
-rw-rw-r-- 0 root root 2.0M Jan  1  1970 big1
-rw-rw-r-- 0 root root 4.0M Jan  1  1970 big2
```

日志：

```
I20230630 12:50:51.505344 439991 fs_write.cc:41] Write begin:
I20230630 12:50:51.505379 439991 fs_write.cc:42] write( /dir4/big1, buf, 1048576, 1048576, fi->fh=13)
I20230630 12:50:51.505390 439991 MetaData.cc:461] HDD occupy 2 blocks
I20230630 12:50:51.505400 439991 MetaData.cc:89] Inode idx: #12 's Inode table offset: 11
I20230630 12:50:51.505425 439991 inode.cc:73] Read Inode #13 from offset: 48128
I20230630 12:50:51.508365 439991 MetaData.cc:89] Inode idx: #12 's Inode table offset: 11
I20230630 12:50:51.508389 439991 inode.cc:83] Write Inode #13 from offset: 48128 : Inode{ i_mode=33204 i_size_lo=2097152 i_blocks_lo=4096 i_block[0]=2066 }
I20230630 12:50:51.508404 439991 MetaData.cc:461] HDD occupy 258 blocks
I20230630 12:50:51.508412 439991 fs_write.cc:96] Write done
```

```
I20230630 12:50:57.985399 439991 fs_write.cc:42] write( /dir4/big2, buf, 1048576, 1048576, fi->fh=14)
I20230630 12:50:57.985410 439991 MetaData.cc:461] HDD occupy 258 blocks
I20230630 12:50:57.985420 439991 MetaData.cc:89] Inode idx: #13 's Inode table offset: 11
I20230630 12:50:57.985435 439991 inode.cc:73] Read Inode #14 from offset: 48384
I20230630 12:50:57.988495 439991 MetaData.cc:89] Inode idx: #13 's Inode table offset: 11
I20230630 12:50:57.988516 439991 inode.cc:83] Write Inode #14 from offset: 48384 : Inode{ i_mode=33204 i_size_lo=2097152 i_blocks_lo=4096 i_block[0]=2323 }
I20230630 12:50:57.988530 439991 MetaData.cc:461] HDD occupy 514 blocks
I20230630 12:50:57.988539 439991 fs_write.cc:96] Write done
```

`dd` 这个命令是一次 1 MB 写入的，第一个日志从 HDD 占用数据块从 2 个 变成 258 个，即在 HDD 成功写入 1 MB。第二个日志 HDD 占用数据块从 258 个变成 514 个，也是成功写入了 1 MB。