哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2021 秋季

课程名称：操作系统

实验名称：基于 FUSE 的青春版 EXT2 文件系统

学生班级：2019 级 4 班

学生学号：190110419

学生姓名：李怡凯

评阅教师：

报告成绩：

实验与创新实践教育中心制

2020 年 9 月

一、实验详细设计

1、 总体设计方案

本文件系统分为 3 个主要层次，分别是应用接口、文件系统内部实现与底层交互与数据结构，文件系统层次结构图如图 1 所示。

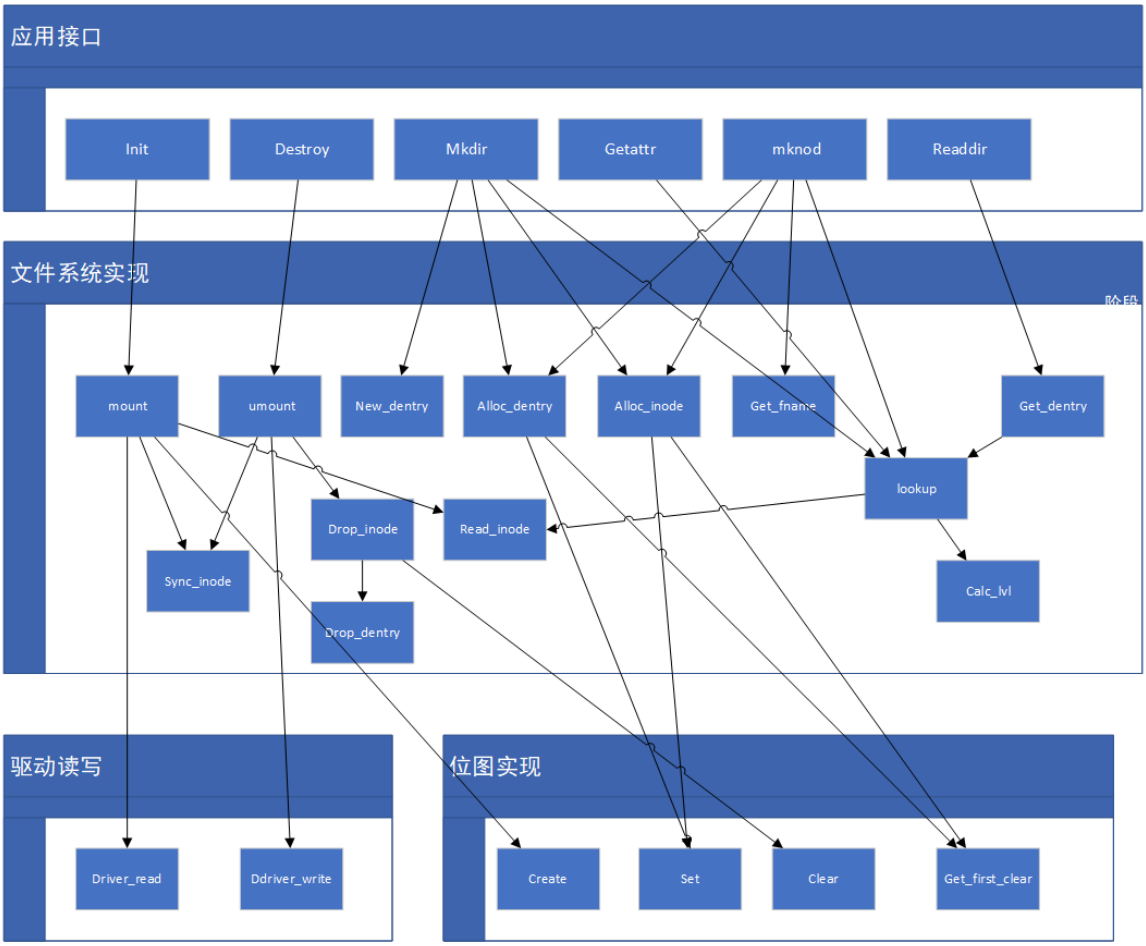

图 1 文件系统层次结构图

2、 功能详细说明

2.1 应用接口

- Init

  实现文件系统的挂载，调用下层文件系统的 mount 函数。

- Destroy

  实现文件系统的卸载，调用下层文件系统的 umount 函数。

- Mkdir

  实现创建文件夹的功能，通过调用下层 new_dentry 与 alloc_dentry 功能实现。

- Getattr

  获取文件信息，与 readdir 共同实现 ls 等命令的实现，通过调用 lookup 函数实现

- Mknod
  创建文件，调用下层 alloc_inode，alloc_dentry 等函数实现。
- Readdir
  依次读取文件夹中的所有文件，获取它们的信息，与 get_attr 函数共同实现 ls 等命令。

2.2 文件系统数据结构

磁盘上的数据结构如下：

- 超级块

```
struct myfs_super_d {
    uint32_t magic;
    int      fd;
    int      is_mounted;                  // 确认是否挂载

    int      sz_io;                      // 磁盘 IO 块大小
    int      sz_disk;                    // 磁盘容量
    // 文件系统参数
    int      max_ino;                     // 最多支持的文件数量 1K
    int      sz_used;                    // 已用磁盘空间
    // 位图
    int      map_inode_blks;              // inode 位图占用的块数
    int      map_inode_offset;            // inode 位图在磁盘上的偏移

    int      map_data_blks;               // data 位图占用的块数
    int      map_data_offset;             // data 位图在磁盘上的偏移
    // 索引节点
    int      inode_blks;                 // 磁盘上索引节点数量
    int      inode_offset;               // 索引节点起始偏移地址
    // 数据块
    int      data_blks;                  // 磁盘上数据块数量
    int      data_offset;                // 数据块偏移地址
};
```

- 索引节点

```
struct myfs_inode_d {
    uint32_t          ino;                // 自己的 inode 块号,对应文件系统,每块为 1KB
    /* TODO: Define yourself */
    int               size;               // 文件已占用空间
    int               link;               // 链接数
    // FILE_TYPE          ftype;              // 文件类型（目录类型、普通文件类型）
    int               cnt;                // 如果是目录类型文件，下面有几个目录项；如果是文件型文件，对应使用了多少个 block_pointer
```

```
    uint32_t           block_pointer[6];   // 数据块指针，保存指向的数据块在数据块区的块号，如果是目录则只有第一个元素有值，目录项之间通过 next_dno 连接
};
```

- 目录项

```
struct myfs_dentry_d {
    char        name[MAX_NAME_LEN];         // 指向的 inode 的文件名
    uint32_t    ino;                        // 指向的 inode 的 inode 块号
    FILE_TYPE   ftype;                      // 目录项指向的是文件还是文件夹
    int         valid;                      // 当前 dentry 是否有效
    uint32_t    dno;                        // 本 dentry 存放在数据块区的块号
    uint32_t    next_dno;                   // 下个目录项的块号
    /* TODO: Define yourself */
};
```

内存上缓存的数据结构如下：

- 超级块

```
struct myfs_super {
    int      fd;
    int      sz_io;                         // 从 super_d 继承
    int      sz_disk;                       // 从 super_d 继承
    // 文件系统参数
    int      max_ino;                       // 从 super_d 继承
    int      sz_used;                       // 从 super_d 继承

    // 位图
    int      map_inode_blks;                // 从 super_d 继承
    int      map_inode_offset;              // 从 super_d 继承
    int      map_data_blks;                 // 从 super_d 继承
    int      map_data_offset;               // 从 super_d 继承

    uint8_t*  map_inode;                    // inode 块位图
    uint8_t*  map_data;                     // data 块位图

    int      inode_offset;                  // 从 super_d 继承
    int      inode_blks;                    // 从 super_d 继承
    int      data_offset;                   // 从 super_d 继承
    int      data_blks;                     // 从 super_d 继承

    int     is_mounted;                     // 从 super_d 继承

    struct myfs_dentry* root_dentry;        // 根目录 dentry 方便快速访问
};
```

- 索引节点

```
struct myfs_inode {
```

```
    uint32_t                    ino;                // 从 inode_d 继承
    /* TODO: Define yourself */
    int                         size;               // 从 inode_d 继承
    int                         link;               // 从 inode_d 继承
    // FILE_TYPE          ftype;                                  // 文件类型
（目录类型、普通文件类型）
    int                         cnt;                // 从 inode_d 继承
    uint32_t                    block_pointer[6];   // 从 inode_d 继承
    struct myfs_dentry*         father_dentry;      // 指向该 inode 的
dentry，在读入时建立
    struct myfs_dentry*         son_dentrys;        // 如果是目录则为目录项
链表指针，在读入 inode 的时候顺便读如并建立
    struct myfs_dentry*         son_dentrys_end;    // 指向 son_dentrys 的最
后一个 dentry，方便插入
    uint8_t*                    data;               // inode 指向的数据，在
read_inode 的时候读入
};
```

- 目录项

```
struct myfs_dentry {
    char                name[MAX_NAME_LEN];         // 从 dentry_d 继承
    uint32_t            ino;                        // 从 dentry_d 继承
    struct myfs_inode*  inode;                      // 指向的 inode 指针，在
建立 dentry 到 inode 的映射时建立
    FILE_TYPE           ftype;                      // 从 dentry_d 继承
    int                 valid;                      // 从 dentry_d 继承
    struct myfs_dentry* sibling;                    // 指向同一目录下的其他
目录项，保存在内存中时用这个指针进行记录，在 read_inode 读入 dentrys 的时候建立
    uint32_t            dno;                        // 从 dentry_d 继承
    uint32_t            next_dno;                   // 从 dentry_d 继承
    /* TODO: Define yourself */
};
```

2.3 文件系统实现

- Mount

  实现文件系统挂载。首先将超级块对应的磁盘内容读取出来，通过比对 magic_num 来判断当前文件系统是否挂载过。如果是首次挂载则需要进行初始化，计算各部分的空间分配以及根节点、位图的初始化；如果之前挂载过则直接读取出来即可。

- Umount

  实现文件系统挂载。将内存数据结构中的信息写回磁盘，然后释放内存上的占用。

- New_dentry

  在内存当中分配一个目录项 dentry。根据文件名与文件类型生成一个目录项 dentry，但并不在磁盘上占用，而是等待其被分配到某个索引

节点上时才在磁盘上分配其磁盘块号。

- Alloc_dentry

  在内存中将一个 dentry 挂到一个目录索引节点 inode 上，并在数据块区位图中分配一个位置用于定位其在磁盘上的存放位置。采用尾插法的方式将目录项插入索引节点，为了提高插入速度在索引节点的内存数据结构中添加指向最后一个目录项的指针。

- Alloc_inode

  为一个目录项 dentry 分配 1 个指向的索引节点 inode。首先创建一个新的索引节点，然后在位图中寻找一个空闲位置赋值给该节点，然后将相关信息填入节点，最后将 dentry 的 inode 指针指向该节点。

- Get_fname

  解析路径获取路径当中的最后一级文件名。通过使用 strrchr 函数从后往前搜索最后一次路径分隔符’/’出现的位置，从而定位到最后一级的路径，即文件名。

- Get_dentry

  根据序号读取目录索引节点 inode 下的目录项 dentry。通过遍历索引节点的目录项链表，寻找对应序号的目录项，然后返回该目录项的指针。

- Lookup

  根据路径寻找该路径是否存在。首先用 calc_lvl 函数计算出路径的级数，然后按级寻找每一级的文件，并且在最后一级前的所有文件都应该是文件夹类型的文件，如果不是则返回错误。

- Drop_inode

  在内存上消除一个节点及其内容或其下的所有目录项。如果索引节点指向文件则首先将索引节点指向的数据块释放并清除数据块位图中的对应位置，然后再释放索引节点并清除索引节点位图的对应位置；如果索引节点指向目录则要进一步递归地将所有目录项也进行释放，然后再释放该节点。

- Read_inode

  读取某个索引节点，将其从磁盘读入内存，并将其下的文件内容或目录项一起读入内存。

- Sync_inode

  将某个索引节点从内存同步回磁盘，如果该索引节点指向一个文件则把文件内容写回磁盘，如果是一个文件夹则递归把其下的所有目录项指向的索引节点也写回磁盘。

- Drop_dentry

  在内存中消除某个目录项，并将其指向的索引节点一并消除。

- Calc_lvl

  根据路径计算该路径的层级数，其中根目录为第 0 级，根目录下的文件或文件夹为第 1 级。

2.4 驱动读写

- Driver_read

  根据给定的磁盘偏移位置与读取的字节数，在磁盘上读取相应的内容。因为磁盘的读写必须按照给定的读写字节对齐，并且每次需要读出

一整个磁盘块，所有在函数中对传入的偏移位置进行对齐，然后读出能够覆盖所需内容的所有磁盘块，最后将磁盘块上对应的内容复制到提供的指针指向的内存当中。

- Driver_write

  根据给定的磁盘偏移位置和读取的字节数，在磁盘上写相应的内容。同上，磁盘写需要首先将磁盘内容读出，经过修改后再写回磁盘，同样需要首先将偏移字节数与读取字节数按照磁盘块大小对齐，然后再进行读取与修改，最后写回。

2.5 位图实现

- Create

  根据记录的块数量生成位图，并初始化所有位置均为 0。使用一个表来记录每个位置的使用情况，固定使用 1 个字节（8 位）作为一行，所以根据给定的块数量可以计算出所需要的字节数，进而分配相应的空间并初始化为 0.

- Set

  将位图给定位置设置为 1.通过给定的位置计算出其行列号，然后读出该行对应的字节，通过将原数据与（1<<bit）进行或操作将对应位置赋值为 1.

- Clear

  将位图给定位置设置为 0。通过给定的位置计算出其行列号，然后读出该行对应的字节，通过将原数据与~（1<<bit）进行与操作将对应位置赋值为 0.

- Get_first_clear

  在位图上寻找第 1 个为 0 的位置。通过遍历整个位图来寻找第一个为 0 的位置。

3、 实验特色

在本文件系统中，在实现基础功能的基础上还实现了内存的缓存结构，即将超级块、索引节点、位图等磁盘上的数据结构在内存中进行缓存，等到文件系统卸载的时候再写回，提高程序的性能。此外还通过层级封装的思想进行设计，将基础操作封装成接口供上层调用。同时大量使用宏定义与宏函数来简化寻址等相关操作，使代码可读性与可维护性更好。

二、用户手册

1. 挂载

   挂载命令如下：

   ./myfs –device=[YOUR DEVICE] -f -s [YOUR MOUNT POINT]

2. 卸载

   卸载命令如下：

   fusermount -u [YOUR MOUNT POINT]

注意卸载时需要首先离开挂载的文件夹，如果遇到提示文件系统忙需要等待文件系统，如果长时间等待仍未结束可以通过 fusermount -zu [YOUR MOUNT POINT]强制卸载

3. 创建文件夹
   创建文件夹命令同 Linux 使用 mkdir
4. 创建文件
   创建文件同 Linux 使用 touch
5. 列出文件夹信息
   同 Linux 使用 ls

三、实验收获和建议

本次实验通过使用 fuse 与 ddriver 实现一个基础的 EXT2 文件系统，并且能够挂载到 Linux 操作系统上运行，进一步加深对文件系统的概念的理解，以及对 Linux 操作系统一切皆文件的思想的理解。

四、参考资料

1. 哈尔滨工业大学（深圳）操作系统实验指导书：https://hitsz-lab.gitee.io/os-labs-2021/lab5/part1/