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：计算机 7 班

学生学号：190110722

学生姓名：邹啟正

评阅教师：

报告成绩：

实验与创新实践教育中心制

2020 年 9 月

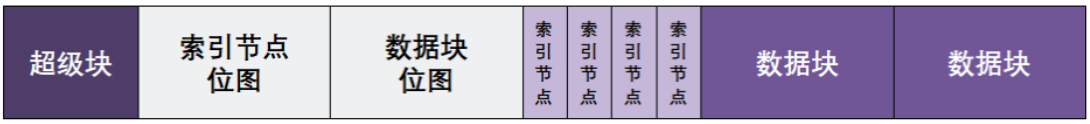
一、实验详细设计

1. 总体设计方案

整个文件系统需要我们自己独立实现的部分没有我想象中那么多，因为善良的助教给我们写了原理十分详尽的指导书以及一个类 EXT2 文件系统 simplefs 做参考。

我们通过 ddriver（disk driver） 驱动来访问 ddriver 设备，从而简化磁盘访问操作。Ddriver 驱动操作很容易掌握，所以主要研究如何在 ddriver 设备上组织文件，基于 FUSE 在用户态设计青春版 EXT2 的介质数据结构、内存数据结构以及一些文件操作的钩子函数，然后通过 FUSE 挂载到 linux 上运行。

本文件系统基本布局：



假定每个文件都占 6 个数据块，整个虚拟磁盘大小通过 ddriver_ioctl 获得为 4194304bit 即 4M，通过 ddriver_ioctl 获得 io_size 为 512b，本文件系统设定数据块大小为 1k。估算各部分占用块数：

1. 超级块我定义的结构体有 10 个 int 型成员，占 40b,一个数据块是 1kb，故占一个块。
2. 索引节点我定义的结构体有 5 个 int 型成员加一个 uint16_t 型的长度为 6 的数组，故 inode 大小为 32b，1 个数据块大致容纳 32 个 inode,设 inode 数为 n,则 n/32+n*6<4093（预计位图大小基本只占 1、2 块），感觉 n 太大也不好，这里我取 n 为 512,即 inode 为 16 块。
3. Inode 为 16 块，索引位图 512bit，64byte,占一块。
4. 数据位图约 4Kbit 即 512byte，占一块。

本文件系统实现的功能（只做了必做部分）：挂载文件系统；卸载文件系统；创建文件/文件夹；查看文件夹下的文件（读取文件夹内容）；

2. 功能详细说明

2.1 文件系统介质、内存数据结构设计

内存数据结构设计：

```
//设备
struct custom_options {
     char*        device;
};
//超级块
struct newfs_super {
    uint32_t magic;
```

```
    int      fd;
    /* TODO: Define yourself */
    int                     driver_fd;
    int                     sz_io;
    int                     sz_blk;
    int                     sz_disk;
    int                     sz_usage;

    int                     max_ino;                // 最多支持的文件数
    int                     max_data;               // 最多数据块
    uint8_t*                map_inode;
    int                     map_inode_blks;
    int                     map_inode_offset;       // inode 位图偏移
    uint8_t*                map_data;
    int                     map_data_blks;          // data 位图占用的块数
    int                     map_data_offset;
    int                     inode_offset;
    int                     data_offset;
    boolean                 is_mounted;
    struct newfs_dentry*    root_dentry;
};
//索引节点
struct newfs_inode {
    // uint32_t ino1;
    /* TODO: Define yourself */
    int                         ino;                    /* 在 inode 位图中的下标 */
    int                         size;                   /* 文件已占用空间 */
    int                         dir_cnt;
    struct newfs_dentry*        dentry;                 /* 指向该 inode 的 dentry */
    struct newfs_dentry*        dentrys;                /* 所有目录项 */
    uint8_t*                    data;
    int                         data_blk[6];            // 数据块指针
};
//目录项
struct newfs_dentry {
```

```
    char                        name[MAX_NAME_LEN];
    /* TODO: Define yourself */
    char                        fname[NEWFS_MAX_FILE_NAME];
    struct newfs_dentry*        parent;                     /* 父亲 Inode 的 dentry */
    struct newfs_dentry*        brother;                    /* 兄弟 */
    int                         ino;
    struct newfs_inode*         inode;                      /* 指向 inode */
    NEWFS_FILE_TYPE             ftype;
};
```

介质数据结构设计：

```
struct newfs_super_d
{
    uint32_t            magic_num;
    int                 sz_usage;

    int                 max_ino;
    int                 max_data;                   // 最多数据块
    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
    int                 inode_offset;               // inode 在磁盘上的偏移
    int                 data_offset;
};
// 32B
struct newfs_inode_d
{
    int                 ino;                        /* 在 inode 位图中的下标 */
    int                 size;                       /* 文件已占用空间 */
    int                 dir_cnt;
    NEWFS_FILE_TYPE     ftype;
    int                 link;               // 链接数
    uint16_t            data_blk[NEWFS_DATA_PER_FILE];
};
```

```
25.
26. struct newfs_dentry_d
27. {
28.     char                fname[NEWFS_MAX_FILE_NAME];
29.     NEWFS_FILE_TYPE     ftype;
30.     int                 ino;                        /* 指向的 ino
    号 */
31. };
```

2.2 封装对 ddriver 的访问代码

这里直接用了 simplefs 的封装 ddriver 读写代码，simplefs 的 io 块 512b、数据块 512b，但青春版的数据块是 1kb，io 块是 512b。之所以可以直接使用是因为封装的 ddriver 读写代码是先将偏移和 size 与 io 块对齐然后一块一块（io 块）读写，青春版不过是多读几块。不需要改动代码。

2.3 挂载

当挂载 FUSE 文件系统时，会执行的钩子是.init，在.init 函数里调用.mount 函数:读取超级块后估算各部分的大小，newfs_supe_d 初始化后读位图，然后全局变量 newfs_super 初始化完成，青春版文件系统初始化完成。

这里用了测试函数 newfs_dump_map()进行调试，也是在 simplefs 文件系统的测试函数基础上添加了打印数据位图。

挂载不成功退出 fuse 框架。

2.4 卸载

卸载执行的钩子是.destroy 函数，在该函数里调用.umount 函数，先调用 newfs_sync_inode()函数将内存中 inode 和数据块写回虚拟磁盘。再将超级块、两个位图写回最后关闭驱动。

在 newfs_sync_inode()函数里先循环将 inode 里的数据块指针数组写回虚拟磁盘，然后是重点：

```
1. /* Cycle 1: 写 INODE */
2.                                                  /* Cycle 2: 写 数
    据 */
3. if (NEWFS_IS_DIR(inode)) {
4.     // 目录文件
5.     dentry_cursor = inode->dentrys;
6.     // 不再使用固定数据块
7.     // 改为使用 inode 指示数据块
8.     offset        = NEWFS_DATA_OFS(inode->data_blk[0]);
9.     // offset        = inode(ino);
10.     while (dentry_cursor != NULL)
11.     {
12.         memcpy(dentry_d.fname, dentry_cursor->fname, NEWFS_MAX_FILE_N
    AME);
13.         dentry_d.ftype = dentry_cursor->ftype;
14.         dentry_d.ino = dentry_cursor->ino;
15.         if (newfs_driver_write(offset, (uint8_t *)&dentry_d,
```

```
                                sizeof(struct newfs_dentry_d)) != NEWFS_
    ERROR_NONE) {
                NEWFS_DBG("[%s] io error\n", __func__);
                return -NEWFS_ERROR_IO;
            }
            // 递归写回各个子目录项的 inode
            if (dentry_cursor->inode != NULL) {
                newfs_sync_inode(dentry_cursor->inode);
            }

            dentry_cursor = dentry_cursor->brother;
            offset += sizeof(struct newfs_dentry_d);
        }
    }
    else if (NEWFS_IS_REG(inode)) {
        // 数据文件
        if (newfs_driver_write(offset, inode->data,
                               NEWFS_BLKS_SZ(1)) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return -NEWFS_ERROR_IO;
        }
    }
```

Inode 指向的是目录文件时，递归写回各个子目录项的 inode。

写回的偏移地址与 simplefs 有区别。由于整体布局的不同，simplefs 数据块是被严格限制的即 1 个索引节点固定分配 16 个连续的数据块，青春版不同，因此需要修改偏移地址相关的宏定义：

```
#define NEWFS_INO_SZ()                  (sizeof(struct newfs_inode_d))

#define NEWFS_INO_OFS(ino)              (newfs_super.inode_offset + ino
    * NEWFS_INO_SZ())
#define NEWFS_DATA_OFS(blk)             (newfs_super.data_offset +  NEW
   FS_BLKS_SZ((blk)))
```

2.5 工具函数：

1) char* newfs_get_fname(const char* path){}//获取文件名
2) int newfs_calc_lvl(const char * path){}//计算路径层级
3) int newfs_alloc_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry){}// 为一个 inode 分配 dentry，采用头插法(这个函数参考了 simplefs 的相应函数)
4) int newfs_alloc_data_blk(){}//分配一个数据块
   每次分配 inode 需要为其初始化一个数据块，那么就需要通过对数据块位图的访问和操作来分配数据块，对操作与对索引节点操作相似，都是通过线性查找，返回一个空闲的数据块号。

```
/**
```

```
 * @brief 分配一个数据块
 *
 * @return 返回块号
 */
int
newfs_alloc_data_blk(){

    int byte_cursor = 0;
     int bit_cursor  = 0;
     int data_blk_cursor  = 0;
     boolean is_find_free_blk = FALSE;
     for (byte_cursor = 0; byte_cursor < NEWFS_MAX_DATA()/UINT8_BITS;
          byte_cursor++)
     {
         for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {

             if((newfs_super.map_data[byte_cursor] & (0x1 << bit_curso
r)) == 0) {
                                                  /* 当前
data_blk_cursor 位置空闲 */
                 newfs_super.map_data[byte_cursor] |= (0x1 << bit_curs
or);
                 is_find_free_blk = TRUE;
                 break;
             }
             data_blk_cursor++;
         }
         if (is_find_free_blk) {
             return data_blk_cursor;
         }
     }
     return -NEWFS_ERROR_NOSPACE;
}
```

5) struct newfs_inode* newfs_alloc_inode(struct newfs_dentry * dentry){}// 为 dentry 分配一个 inode，占用位图(参考 simplefs)
通过对索引节点位图的线性查找，找到一个空闲的 inode 进行分配，并完成初始化，前面也说过 simplefs 与青春版布局的不同，这里 simplefs 在分配一个 inode 后初始化数据块分配了 16 个块，而青春版因为每个 inode 指向的 6 个数据块不是连续的，只需要初始化一个数据块即可。

```
/**
 * @brief 为 dentry 分配一个 inode，占用位图
 *
```

```
 * @param dentry 该 dentry 指向分配的 inode
 * @return newfs_inode
 */
struct newfs_inode* newfs_alloc_inode(struct newfs_dentry * dentry) {
    struct newfs_inode* inode;
    int byte_cursor = 0;
     int bit_cursor  = 0;
     int ino_cursor  = 0;
     boolean is_find_free_entry = FALSE;

     // for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_inode_blks);          byte_cursor++)
     for (byte_cursor = 0; byte_cursor < NEWFS_MAX_INO()/UINT8_BITS; byte_cursor++)
     {
         for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {

             if((newfs_super.map_inode[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                     /* 当前 ino_cursor 位置空闲 */
                 newfs_super.map_inode[byte_cursor] |= (0x1 << bit_cursor);
                 is_find_free_entry = TRUE;
                 break;
             }
             ino_cursor++;
         }
         if (is_find_free_entry) {
             break;
         }
     }

     // if (!is_find_free_entry || ino_cursor == newfs_super.max_ino)

     //     return -NEWFS_ERROR_NOSPACE;

     inode = (struct newfs_inode*)malloc(sizeof(struct newfs_inode));

     inode->ino  = ino_cursor;
     inode->size = 0;
                                                     /* dentry 指向 inode */
     dentry->inode = inode;
```

```
    dentry->ino   = inode->ino;
                                                        /* inode 指回 dentry */
    inode->dentry = dentry;

    inode->dir_cnt = 0;
    inode->dentrys = NULL;
    inode->data_blk[0] = newfs_alloc_data_blk();

    if (NEWFS_IS_REG(inode)) {
        inode->data = (uint8_t *)malloc(NEWFS_BLKS_SZ(1));
    }

    return inode;
}
```

6) int newfs_sync_inode(struct newfs_inode * inode){}// 将内存 inode 及其下方结构全部刷回磁盘，主要步骤分为写回该 inode 和写回该 inode 对应的文件内容，如果是目录，则递归写回目录内的内容，如果是文件，则将文件写回。该函数在讲解卸载时有详细介绍。
7) struct newfs_inode* newfs_read_inode(struct newfs_dentry * dentry, int ino){}//读取 dentry 指向的 inode
8) struct newfs_dentry* newfs_get_dentry(struct newfs_inode * inode, int dir){}//找到 inode 的第 dir 条目录项
9) struct newfs_dentry* newfs_lookup(const char * path, boolean* is_find, boolean* is_root)//根据 path 找到目录项

2.6 其他钩子函数

1) .get_attr 钩子参考 simplefs 文件系统实现：用 lookup 解析路径，获取 Inode，填充 newfs_stat
   Code:

```
/**
 * @brief 获取文件或目录的属性，该函数非常重要
 *
 * @param path 相对于挂载点的路径
 * @param newfs_stat 返回状态
 * @return int 0 成功，否则失败
 */
int newfs_getattr(const char* path, struct stat * newfs_stat) {
    /* TODO: 解析路径，获取 Inode，填充 newfs_stat，和/fs/simplefs/sfs.c 的 sfs_getattr()函数实现基本一致 */
    boolean is_find, is_root;
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    if (is_find == FALSE) {
```

```
        return -NEWFS_ERROR_NOTFOUND;
    }

    if (NEWFS_IS_DIR(dentry->inode)) {
        newfs_stat->st_mode = S_IFDIR | NEWFS_DEFAULT_PERM;
        newfs_stat->st_size = dentry->inode->dir_cnt * sizeof(struct newfs_dentry_d);
    }
    else if (NEWFS_IS_REG(dentry->inode)) {
        newfs_stat->st_mode = S_IFREG | NEWFS_DEFAULT_PERM;
        newfs_stat->st_size = dentry->inode->size;
    }

    newfs_stat->st_nlink = 1;
    newfs_stat->st_uid     = getuid();
    newfs_stat->st_gid     = getgid();
    newfs_stat->st_atime   = time(NULL);
    newfs_stat->st_mtime   = time(NULL);
    newfs_stat->st_blksize = NEWFS_BLK_SZ();

    if (is_root) {
        newfs_stat->st_size  = newfs_super.sz_usage;
        newfs_stat->st_blocks = NEWFS_DISK_SZ() / NEWFS_BLK_SZ();
        newfs_stat->st_nlink  = 2;        /* !特殊,根目录 link 数为 2 */
    }
    return NEWFS_ERROR_NONE;
}
```

2) .mknod 钩子函数：解析路径，并创建相应的文件

Code:

```
/**
 * @brief 创建文件
 *
 * @param path 相对于挂载点的路径
 * @param mode 创建文件的模式，可忽略
 * @param dev 设备类型，可忽略
 * @return int 0 成功，否则失败
 */
int newfs_mknod(const char* path, mode_t mode, dev_t dev) {
    /* TODO: 解析路径，并创建相应的文件 */
    boolean is_find, is_root;

    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);
```

```
    struct newfs_dentry* dentry;
    struct newfs_inode* inode;
    char* fname;

    if (is_find == TRUE) {
        return -NEWFS_ERROR_EXISTS;
    }

    fname = newfs_get_fname(path);

    if (S_ISREG(mode)) {// 文件
        dentry = new_dentry(fname, NEWFS_REG_FILE);
    } else if (S_ISDIR(mode)) {// 文件夹
        dentry = new_dentry(fname, NEWFS_DIR);
    }
    dentry->parent = last_dentry;
    inode = newfs_alloc_inode(dentry);
    newfs_alloc_dentry(last_dentry->inode, dentry);

    return NEWFS_ERROR_NONE;
}
```

3) .mkdir 钩子函数： 和 mkmod 逻辑相似，先解析给定的路径，判断文件/目录是否已经存在了，如果不存在，再为它分配目录项、inode 和数据块。

Code:

```
/**
 * @brief 创建目录
 *
 * @param path 相对于挂载点的路径
 * @param mode 创建模式（只读？只写？），可忽略
 * @return int 0 成功，否则失败
 */
int newfs_mkdir(const char* path, mode_t mode) {
    /* TODO: 解析路径，创建目录 */
    boolean is_find, is_root;
    char* fname;
    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_dentry* dentry;
    struct newfs_inode*  inode;

    if (is_find) {
        return -NEWFS_ERROR_EXISTS;
    }
```

```
19.
20.     if (NEWFS_IS_REG(last_dentry->inode)) {
21.         return -NEWFS_ERROR_UNSUPPORTED;
22.     }
23.
24.     fname  = newfs_get_fname(path);
25.     dentry = new_dentry(fname, NEWFS_DIR);
26.     dentry->parent = last_dentry;
27.     inode  = newfs_alloc_inode(dentry);
28.     newfs_alloc_dentry(last_dentry->inode, dentry);
29.     //newfs_dump_map(0);
30.     //newfs_dump_map(1);
31.     return NEWFS_ERROR_NONE;
32. }
```

4) .readdir 钩子函数：用 lookup 解析路径，获取目录的 Inode，并读取目录项。

Code:

```
1. /**
2.  * @brief 遍历目录项，填充至 buf，并交给 FUSE 输出
3.  *
4.  * @param path 相对于挂载点的路径
5.  * @param buf 输出 buffer
6.  * @param filler 参数讲解:
7.  *
8.  * typedef int (*fuse_fill_dir_t) (void *buf, const char *name,
9.  *                 const struct stat *stbuf, off_t off)
10.  * buf: name 会被复制到 buf 中
11.  * name: dentry 名字
12.  * stbuf: 文件状态，可忽略
13.  * off: 下一次 offset 从哪里开始，这里可以理解为第几个 dentry
14.  *
15.  * @param offset 第几个目录项？
16.  * @param fi 可忽略
17.  * @return int 0 成功，否则失败
18.  */
19. int newfs_readdir(const char * path, void * buf, fuse_fill_dir_t fill
    er, off_t offset,
20.                          struct fuse_file_info * fi) {
21.     /* TODO: 解析路径，获取目录的 Inode，并读取目录项，利用 filler 填充到 buf，
    可参考/fs/simplefs/sfs.c 的 sfs_readdir()函数实现 */
22.     boolean is_find, is_root;
23.     int     cur_dir = offset;
24.
```

```
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_dentry* sub_dentry;
    struct newfs_inode* inode;
    if (is_find) {
        inode = dentry->inode;
        sub_dentry = newfs_get_dentry(inode, cur_dir);
        if (sub_dentry) {
            filler(buf, sub_dentry->fname, NULL, ++offset);
        }
        return NEWFS_ERROR_NONE;
    }
    return -NEWFS_ERROR_NOTFOUND;
}
```

3. 实验结果

```
190110722@OSLabExecNode0:~/user-land-filesystem/fs/newfs$ ./tests/fs_test.sh
目标设备 /home/guests/190110722/ddriver
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0455084 s, 92.2 MB/s
>>>>>>>>>>>>>>>>>>> TEST_MOUNT
pass: [all-the-mount-test]
<<<<<<<<<<<<<<<<<<<

>>>>>>>>>>>>>>>>>>> TEST_MKDIR
TEST: mkdir ./mnt/dir0
pass: -> mkdir ./mnt/dir0
TEST: mkdir ./mnt/dir0/dir0
pass: -> mkdir ./mnt/dir0/dir0
TEST: mkdir ./mnt/dir0/dir0/dir0
pass: -> mkdir ./mnt/dir0/dir0/dir0
TEST: mkdir ./mnt/dir1
pass: -> mkdir ./mnt/dir1
<<<<<<<<<<<<<<<<<<<

>>>>>>>>>>>>>>>>>>> TEST_TOUCH
TEST: touch ./mnt/file0
pass: -> touch ./mnt/file0
TEST: touch ./mnt/dir0/file0
pass: -> touch ./mnt/dir0/file0
TEST: touch ./mnt/dir0/dir0/file0
pass: -> touch ./mnt/dir0/dir0/file0
TEST: touch ./mnt/dir0/dir0/dir0/file0
pass: -> touch ./mnt/dir0/dir0/dir0/file0
TEST: touch ./mnt/dir1/file0
pass: -> touch ./mnt/dir1/file0
<<<<<<<<<<<<<<<<<<<

>>>>>>>>>>>>>>>>>>> TEST_LS
TEST: ls ./mnt/
dir0  dir1  file0
pass: -> ls ./mnt/
TEST: ls ./mnt/dir0
dir0  file0
pass: -> ls ./mnt/dir0
TEST: ls ./mnt/dir0/dir0
dir0  file0
pass: -> ls ./mnt/dir0/dir0
TEST: ls ./mnt/dir0/dir0/dir0
file0
pass: -> ls ./mnt/dir0/dir0/dir0
TEST: ls ./mnt/dir1
file0
```

```
>>>>>>>>>>>>>>>>>>>> TEST_REMOUNT
pass: -> fusermount -u ./mnt
pass: -> ../build/newfs --device=/home/guests/190110722/ddriver ./mnt
TEST: ls ./mnt/
dir0  dir1  file0
pass: -> ls ./mnt/
TEST: ls ./mnt/dir0
dir0  file0
pass: -> ls ./mnt/dir0
TEST: ls ./mnt/dir0/dir0
dir0  file0
pass: -> ls ./mnt/dir0/dir0
TEST: ls ./mnt/dir0/dir0/dir0
file0
pass: -> ls ./mnt/dir0/dir0/dir0
TEST: ls ./mnt/dir1
file0
pass: -> ls ./mnt/dir1
pass: -> fusermount -u ./mnt
<<<<<<<<<<<<<<<<<<<<

pass: 恭喜你，通过所有测试 (23/23)
```

二、用户手册

实现的文件系统中的所有命令使用方式

1. fusermount -u
   用法：fusermount -u mountpoint 将挂载点卸载
2. mkdir：建立目录
   用法：mkdir [OPTION]... DIRECTORY... 建立目录 DIRECTORY
3. cd：目录切换
   用法：cd [-L|[-P [-e]] [-@]] [dir] 切换目录到 dir。
4. ls：目录查看
   用法：ls [OPTION]... [FILE]... 部分选项： -l，列表查看，显示更多信息 -i，打印文件的 inode 号和文件号 -R，递归列出目录下的所有子目录 -r，反序列出目录项
5. touch： 创建文件
   用法：touch [OPTION]... FILE... 修改文件或者目录的时间属性，若不存在，系统会建立一个新的文件。

三、实验收获和建议

操作系统整个实验课程过程中的收获、感受、问题、建议等。

1. 操作系统整个实验课程中的收获：
   1.1 最重要的一点收获：对 xv6 操作系统理解更加深入了，这虽然是操作系统课程，但我们无论是同学还是助教老师可没时间精力在 linux 操作系统上进行实验，在 xv6

上做实验简直再合适不过了，xv6，麻雀虽小五脏俱全！

1.2 除了理解操作系统各部分原理，我写 c 语言的功底也有所长进，这是在做实验的时候锻炼到的，同时对程序规范的理解也更进一步了，比如康康助教写的文件系统样例，你真的能看出很多好的编码习惯。

1.3 看得见的收获：主要是一个自己写的能在 linux 上跑的文件系统吧（虽然很多东西助教已经给了，但我脸皮够厚）

2. 操作系统整个实验课程中的感受：

2.1 感觉不愧于咱的校训“规格严格、功夫到家”

2.2 感觉挺幸运的，因为有这么和善耐心强大的老师和助教。

2.3 自己挺菜的，计算机基础还有点欠缺（编码、操作系统知识等）

3. 操作系统整个实验课程中的建议：

3.1 建议老师和助教们注意休息，苏婷老师真的挺辛苦，我看好多实验课都是她教……

3.2 我觉得可以在 MIT 的实验上多变动一点（毕竟网上代码太多）

四、参考资料

Simplefs

助教 YYDS