

2022春操作系统实验四实验报告

PB21111681 朱炜荣

实验目标

1. 实验第一部分：实现FAT文件系统读操作

- 能够运行 `tree` , `ls` 命令查看文件目录结构
- 能够正确读取小于一个簇的文件, 通过短文件读取测试
- 能够正确读取长文件, 通过长文件读取测试

2. 实验第二部分：实现FAT文件系统创建/删除文件、目录操作

- 能够运行 `touch` 、 `mkdir` 命令创建新文件、目录, 要保证文件属性的正确填写
- 能够运行 `rm` 、 `rmdir` 、 `rm -r` 命令删除已有文件、目录, 要保证簇的正确释放

3. 实验第三部分：实现FAT文件系统写操作

- 能够正确写入文件, 通过文件写入、截断测试

4. 实验第四部分：FAT 文件系统性能优化

- 在模拟磁盘环境下, 性能达到基准测试 50%
- 在模拟磁盘环境下, 性能达到基准测试的 200%

实验环境

- OS: Ubuntu 20.04.4 LTS
- Linux内核版本: 5.15.0
- 虚拟机: VirtualBox
- 虚拟机CPU: 6核
- 虚拟机内存大小: 4096MB

实验过程

1. 实验第一部分

- 实验的最开始我先通读了一遍助教给出的实验文档, 大致了解了实验具体是希望我们做什么。然后下载解压好文件夹后, 阅读了 `simple_fat16.c` 中前部分的一些函数定义, 同时在阅读的过程中思考了如何正确合理地使用这些函数。
- 然后是一个一个填写 `TODO1.x` 系列。

■ TODO1.1:

感觉没有什么可以解释的, 主要是看懂结构体的定义之后给根目录的扇区号和扇区数赋值。

```
sector_t root_sec = meta.root_sec;
```

```
size_t nsec = meta.root_sectors;
```

■ TODO1.2:

与TODO1.1比较类似，赋值唯一不同的变量是扇区号通过簇号和簇号变化扇区号函数得到。

```
sector_t sec = cluster_first_sector(clus);  
  
size_t nsec = meta.sec_per_clus;  
  
state = find_entry_in_sectors(*remains, len, sec,  
nsec, slot);
```

■ TODO1.3:

主要是完成find_entry_in_sectors () 函数。函数的主要逻辑为两个for循环遍历，外层for循环主要是遍历所有的扇区号，因为读写的基本逻辑必须以整个扇区为颗粒度。内层for循环主要是以每个扇区内以目录项大小为颗粒度遍历目录项中的信息。并根据目录项信息不同，返回不同的数值。

■ TODO1.4:

主要为read_fat_entry () 函数的完成。

```
char sector_buffer[PHYSICAL_SECTOR_SIZE];  
  
sector_t sec = meta.fat_sec + clus * 2 / meta.sector_size;  
  
sector_read(sec, sector_buffer);  
  
off_t off = clus * 2 % meta.sector_size;  
  
cluster_t index;  
  
memcpy(&index, sector_buffer + off, sizeof(cluster_t));  
  
return index;
```

首先是计算簇号的偏移量，2是每个簇的字节大小。然后依靠所计算的扇区号进行读取，并在需要读取的簇号偏移处进行赋值读取。

■ TODO1.5:

填充代码如下：

```
for(size_t i=0; i < nsec; i++) {  
  
    sector_t sec = first_sec + i;  
  
    sector_read(sec, sector_buffer);  
  
    for(size_t off = 0; off < meta.sector_size; off +=  
DIR_ENTRY_SIZE) {  
  
        DIR_ENTRY* dir = (DIR_ENTRY*)(sector_buffer + off);  
  
        if(is_free(dir)){  
  
            break;  
  
        }  
  
        else if(is_deleted(dir) || is_lfn(dir->DIR_Attr)){  
  
            continue;  
  
        }  
  
        else{  
  
            to_longname(dir->DIR_Name, name, MAX_NAME_LEN);  
  
            filler(buf, name, NULL, 0, 0);  
  
        }  
}
```

```
    }
```

```
}
```

依旧是遍历扇区中的目录项，遍历时需要先检查该目录项是否为空，然后再判断是否为长文件或已经删除文件，最后如果之前都不是，那么我们便处理得到目录项的文件名，用to_longname将其转化为我们真正看到的文件名形式，利用filler函数传出。

■ TODO1.6:

主要代码如下：

```
size_t flag = 0;

while(flag != size){

    off_t off = offset % meta.cluster_size;

    size_t index_size = min(meta.cluster_size - off, size - flag);

    int ret = read_from_cluster_at_offset(clus, off, buffer+flag,
index_size);

    if(ret < 0){

        return ret;

    }

    offset += ret;

    flag += ret;

    if(offset % meta.cluster_size == 0){

        clus = read_fat_entry(clus);

    }

}
```

其逻辑为：首先定义一个读取字节多少的变量，然后进行遍历，先计算当前偏移在每个簇中的偏移，然后index_size实时刷新定义我们从簇中读取的大小，防止读取的并非一整个簇，然后以实时偏移，实时大小在簇中进行读取，offset和flag也随之更新，如果读取完成的话，我们便去寻找下一个簇。最后当读取字节数大小与size相同时便结束循环。

- 利用助教给出的自动化测试脚本，跑通前6项之后便结束对应函数填充。

2. 实验第二部分

- 从此部分开始，实验的逻辑开始变得难起来，除了本身的代码逻辑外，此处的代码还需要调用借鉴自己在TODO1.x系列中写过的函数，而当时可能编写的代码存在微小的逻辑错误，而脚本没有发现，等到第二部分检测时才暴露出来：（
- 开始处理TODO2.x系列：

■ TODO2.1:

```
char sector_buffer[PHYSICAL_SECTOR_SIZE];

// TODO2.1:

// 1. 读取 slot.dir 所在的扇区

sector_read(slot.sector, sector_buffer);

// 2. 将目录项写入buffer对应的位置（Hint：使用memcpy）

memcpy(sector_buffer+slot.offset, &slot.dir, sizeof(DIR_ENTRY));

// 3. 将整个扇区完整写回
```

```
sector_write(slot.sector, sector_buffer);
```

由于助教给出的注释已经十分详尽了，就按照逻辑提示补全代码。

■ TODO2.2:

```
// TODO2.2: 修改第 i 个 FAT 表中，clus_sec 扇区中，sec_off 偏移处的表项，使其  
值为 data
```

```
// 1. 计算第 i 个 FAT 表所在扇区，进一步计算clus应的FAT表项所在扇区
```

```
// 2. 读取该扇区并在对应位置写入数据
```

```
// 3. 将该扇区写回
```

```
sector_t sec = meta.fat_sec + clus_sec + i * meta.sec_per_fat;
```

```
sector_read(sec, sector_buffer);
```

```
memcpy(sector_buffer + sec_off, &data, sizeof(cluster_t));
```

```
sector_write(sec, sector_buffer);
```

类似地，这里的逻辑和提示也比较充足。这里的变量定义在write_fat_entry () 函数中给出，主要的逻辑与read_fat_entry()函数对称。

■ TODO2.3:

```
cluster_t index=CLUSTER_MIN;
```

```
while(allocated < n && index != CLUSTER_END){
```

```
    if(read_fat_entry(index++) == 0x00){
```

```
        clusters[allocated++] = index-1;
```

```
    }
```

```
}
```

按照可用合理的簇号递增进行一个个遍历，如果簇为空闲簇那么就将对应的簇号存入我们暂时定义的数组中。当全部遍历完或者数量达到预期便退出循环。

■ TODO2.4:

助教自己完成了（乐）。

■ TODO2.5:

```
for(size_t i=0; i < n; i++){
```

```
    write_fat_entry(clusters[i], clusters[i+1]);
```

```
}
```

由于暂时定义的数组多了一位用于定义结束簇，所以直接进行分配数量的遍历就好，逻辑与构建链表的过程有些类似。

■ TODO2.6 && TODO2.7:

对应的函数为创建目录，代码如下：

```
int fat16_mkdir(const char *path, mode_t mode) {
```

```
    printf("mkdir(path='%s', mode=%03o)\n", path, mode);
```

```
    // TODO2.6: 参考fat16_mknod实现，创建新目录
```

```
    // Hint: 注意设置的属性不同。
```

// Hint: 新目录最开始即有两个目录项, 分别是.和.., 所以需要给新目录分配一个簇。

// Hint: 你可以使用 alloc_clusters 来分配簇。

DirEntrySlot slot;

const char* filename = NULL;

int ret = find_empty_slot(path, &slot, &filename);

if(ret < 0) {

return ret;

}

cluster_t first;

ret = alloc_clusters(1,&first);

if(ret < 0){

return ret;

}

char shortname[11];

ret = to_shortcode(filename, MAX_NAME_LEN, shortname);

if(ret < 0) {

return ret;

}

ret = dir_entry_create(slot, shortname, ATTR_DIRECTORY, first, 0);

if(ret < 0) {

return ret;

}

const char DOT_NAME[] = ".";

const char DOTDOT_NAME[] = "..";

// TODO2.7: 使用 dir_entry_create 创建 . 和 .. 目录项

// Hint: 两个目录项分别在你刚刚分配的簇的前两项。

// Hint: 记得修改下面的返回值。

sector_t first_sec = cluster_first_sector(first);

slot.sector = first_sec;

slot.offset = 0;

ret = dir_entry_create(slot,DOT_NAME,ATTR_DIRECTORY, first, 0);

if(ret < 0){

```

        return ret;
    }

    slot.offset = DIR_ENTRY_SIZE;

    // cluster_t index=CLUSTER_MIN;

    // while(index != CLUSTER_END){
        // if(read_fat_entry(index) == *first){
            // *parent = index;

        // }

        // index++;

    // }

    ret = dir_entry_create(slot,DOTDOT_NAME,ATTR_DIRECTORY, first, 0);

    if(ret < 0){
        return ret;
    }

    return 0;
}

```

主要逻辑为先找到一个空槽，然后分配1个簇大小的空间，然后处理目录项的名字问题，最后借助参数和函数dir_entry_create创建目录项。与创建文件不同，目录项的定义以后自身便携带两个子目录，一个为当前目录，另一个上一级目录，与上面逻辑类似分别处理"/"和"/.."的目录项创建，由于dir_entry_create () 函数内部逻辑中又重新处理了slot的一些成员信息，所以在外面只处理簇号、偏移即可。

- 删除目录项函数：

```

int flag = 0;

char sector_buffer[MAX_LOGICAL_SECTOR_SIZE];

sector_read(cluster_first_sector(slot.dir.DIR_FstClusLO),sector_buffer);

for(size_t off = 0; off < meta.sector_size; off+= DIR_ENTRY_SIZE){
    DIR_ENTRY *dir_in = (DIR_ENTRY *) (sector_buffer + off);

    if(is_free(dir_in) || is_dot(dir_in) || is_deleted(dir_in)){
        continue;
    }

    if(is_valid(dir_in)){
        flag = 1;
        break;
    }
}

if(flag == 1){
    return -ENOTEMPTY;
}

```

```

    }

    ret = free_clusters(dir->DIR_FstClusLO);

    if(ret < 0){

        return ret;

    }

    dir->DIR_Name[0] = NAME_DELETED;

    ret = dir_entry_write(slot);

    if(ret < 0){

        return ret;

    }

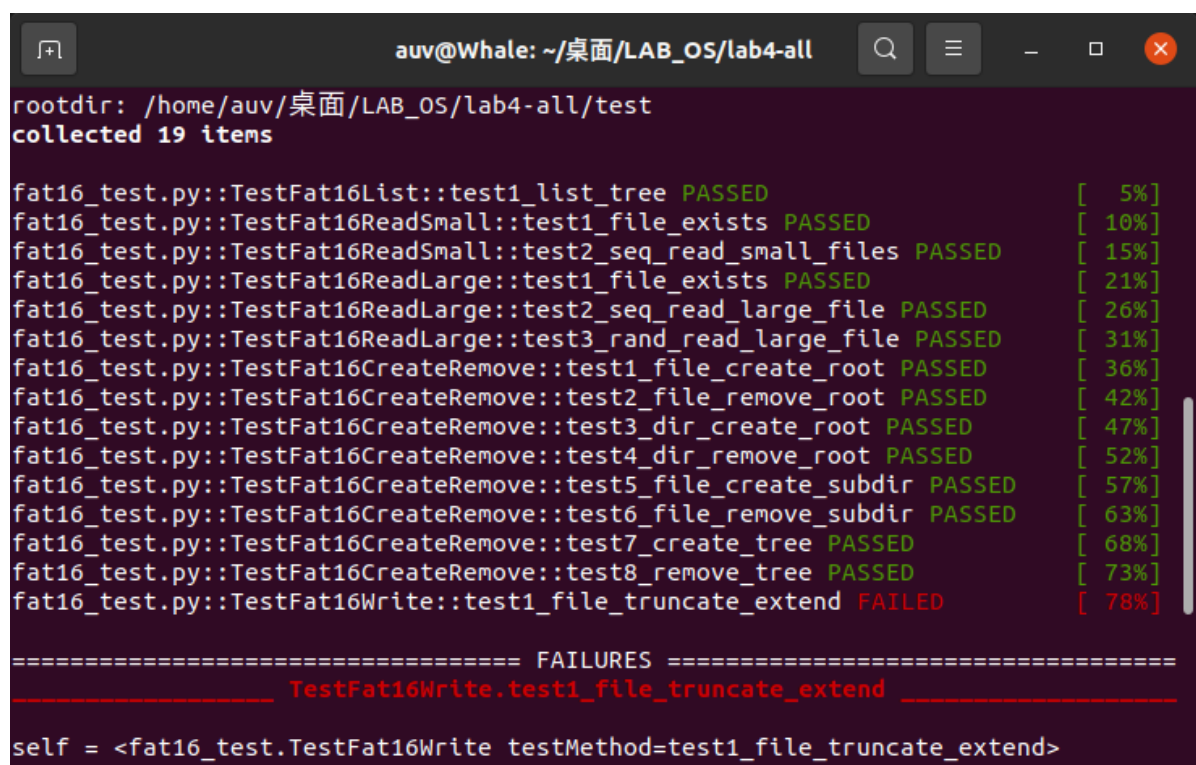
```

借助助教给出的提示，首先读取目录的扇区，然后遍历扇区中的目录项，然后处理目录项的特征，如果是空项、"/"、"./"、删除项便忽略，如果不合法那么直接跳出并返回-ENOTEMPTY 错误，如果均合法便——释放这些簇，最后修改该目录项为删除项并返回0。

3. 实验第三部分和实验第四部分在撰写实验报告时还没有进行，在后续的补查中会其写完。

实验结果

实验中助教给出的自动化脚本运行结果如下：



```

auv@Whale: ~/桌面/LAB_OS/lab4-all
rootdir: /home/auv/桌面/LAB_OS/lab4-all/test
collected 19 items

fat16_test.py::TestFat16List::test1_list_tree PASSED [ 5%]
fat16_test.py::TestFat16ReadSmall::test1_file_exists PASSED [ 10%]
fat16_test.py::TestFat16ReadSmall::test2_seq_read_small_files PASSED [ 15%]
fat16_test.py::TestFat16ReadLarge::test1_file_exists PASSED [ 21%]
fat16_test.py::TestFat16ReadLarge::test2_seq_read_large_file PASSED [ 26%]
fat16_test.py::TestFat16ReadLarge::test3_rand_read_large_file PASSED [ 31%]
fat16_test.py::TestFat16CreateRemove::test1_file_create_root PASSED [ 36%]
fat16_test.py::TestFat16CreateRemove::test2_file_remove_root PASSED [ 42%]
fat16_test.py::TestFat16CreateRemove::test3_dir_create_root PASSED [ 47%]
fat16_test.py::TestFat16CreateRemove::test4_dir_remove_root PASSED [ 52%]
fat16_test.py::TestFat16CreateRemove::test5_file_create_subdir PASSED [ 57%]
fat16_test.py::TestFat16CreateRemove::test6_file_remove_subdir PASSED [ 63%]
fat16_test.py::TestFat16CreateRemove::test7_create_tree PASSED [ 68%]
fat16_test.py::TestFat16CreateRemove::test8_remove_tree PASSED [ 73%]
fat16_test.py::TestFat16Write::test1_file_truncate_extend FAILED [ 78%]

===== FAILURES =====
_____ TestFat16Write.test1_file_truncate_extend _____

self = <fat16_test.TestFat16Write testMethod=test1_file_truncate_extend>

```

即目前完成了实验要求的7分部分。

总结

1. 加深了在操作系统课程中学到的有关FAT相关内容的理解。
2. 难以通过printf大法debug从而倒逼了自己的逻辑严谨。
3. 还有3分的内容没有进行，会在最后补查之前写完再检查。
4. 进一步了解了Linux，以及其一切皆文件的伟大之处，减少了不必要的类型定义减少了额外的开销。
5. 理解了文件系统的重要意义，如果仅有很大的内存却没有合理的逻辑来操作读写的话，时间代价和成本会非常高。

吐槽和建议：最后一次实验的难度分化合理，只不过时间安排大多与cod实验冲突、最后又遇上了期末考试周，导致大家写实验时心情也不是很好（。最后建议实验报告的提交能否与实验检查截止对应上，先写完报告再补实验总感觉有些怪怪的）