

lab4 report

实验目标

- 熟悉 FAT16 文件系统存储结构
 - FAT 表项
 - 簇, 扇区
 - 目录项
- 利用 FUSE 实现一个 FAT 文件系统
 - 文件或目录的读
 - 文件或目录的创建与删除
 - 文件或目录的写
- 优化文件系统性能 (选做)

实验环境

- 虚拟机:
 - APP: VMware Workstation Pro
 - OS: Ubuntu 20.04.4 LTS
 - 内存: 8GB
 - 处理器: 4
- 物理机:
 - CPU: i7-11800H @2.30GHz
 - 内存: 32G

实验过程

关于环境配置的问题实验文档已经讲的很清楚了, 不再赘述。这里主要是解释代码填空。

任务一：读文件或目录

read_fat_entry

读取给定簇号的 FAT 表项

先计算在哪个扇区（FAT 表的起始扇区 + 偏移量），再计算在扇区中的偏移量。先读取整个扇区，再取目标部分。

代码如下：

```
1  cluster_t read_fat_entry(cluster_t clus) {
2      char sector_buffer[PHYSICAL_SECTOR_SIZE];
3      // 计算表项扇区号
4      size_t offset = clus * sizeof(cluster_t);
5      sector_t sector = offset / meta.sector_size + meta.fat_sec;
6      // 读取扇区
7      sector_read(sector, sector_buffer);
8      // 读取表项值
9      offset %= meta.sector_size;
10     return *(cluster_t *) (sector_buffer + offset);
11 }
```

find_entry_in_sectors

在给定的连续扇区中查找给定名称的目录项

循环遍历每个待查找扇区的每个目录项，比较文件名。若找到了目标文件，返回 `FIND_EXIST`，若找到了空槽，返回 `FIND_EMPTY`，否则返回 `FIND_FULL`（表示没找到且这些扇区都满了）

注意：排除已被删除的目录项的干扰。

代码如下：

```
1  int find_entry_in_sectors(const char* name, size_t len, sector_t
   from_sector, size_t sectors_count, DirEntrySlot* slot) {
2      char buffer[PHYSICAL_SECTOR_SIZE];
3      // 对每一个待查找的扇区：
4      for(size_t i = 0; i < sectors_count; i++) {
5          // 读取第 i 个扇区
6          sector_read(from_sector + i, buffer);
7          for(size_t off = 0; off < meta.sector_size; off += DIR_ENTRY_SIZE)
           {
```

```

8         DIR_ENTRY* dir = (DIR_ENTRY*)(buffer + off);
9         // 检测目录项是否合法
10        if (is_deleted(dir)) {
11            continue;
12        }
13        // 比较文件名
14        if(check_name(name, len, dir)) {
15            // 找到目标文件
16            slot->dir = *dir;
17            slot->sector = from_sector + i;
18            slot->offset = off;
19            return FIND_EXIST;
20        }
21        else if (is_free(dir)) {
22            // 找到空槽
23            slot->dir = *dir;
24            slot->sector = from_sector + i;
25            slot->offset = off;
26            return FIND_EMPTY;
27        }
28    }
29    }
30    // printf("find_entry_in_sectors: no empty slot\n");
31    return FIND_FULL;
32 }

```

后面还有个函数 `find_entry_internal` 是基于此函数的，虽然函数本身很复杂，但要填的地方很简单，这里就不提及了。

fat16_readdir

读目录的主函数，读取给定路径的目录包含哪些文件

找路径对应目录的目录项，获取第一个簇的信息，然后遍历每一个簇的每一个扇区。

注意：碰到空项就可以停止了，提升性能；排除非法项的干扰，例如已删除的项

代码如下：

```

1  int fat16_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
    off_t offset,
2      struct fuse_file_info *fi, enum fuse_readdir_flags
    flags) {
3      bool root = path_is_root(path);
4      // DIR_ENTRY dir;
5      cluster_t clus = CLUSTER_END;

```

```

6     if(!root) {
7         DirEntrySlot slot;
8         DIR_ENTRY* dir = &(slot.dir);
9         int ret = find_entry(path, &slot);
10        if(ret < 0) {
11            return ret;
12        }
13        clus = dir->DIR_FstClusL0;    // 不是根目录
14        if(!is_directory(dir->DIR_Attr)) {
15            return -ENOTDIR;
16        }
17    }
18    // 要读的目录项的第一个簇位于 clus, 请你读取该簇中的所有目录项。
19    char sector_buffer[MAX_LOGICAL_SECTOR_SIZE];
20    char name[MAX_NAME_LEN];
21    // 遍历簇
22    while (root || is_cluster_inuse(clus)) {
23        sector_t first_sec;
24        size_t nsec;
25        if(root) {
26            first_sec = meta.root_sec;
27            nsec = meta.root_sectors;
28        } else {
29            first_sec = cluster_first_sector(clus);
30            nsec = meta.sec_per_clus;
31        }
32        // 遍历扇区
33        for(size_t i = 0; i < nsec; i++) {
34            sector_t sec = first_sec + i;
35            sector_read(sec, sector_buffer);
36            // 遍历目录项
37            for (size_t off = 0; off < meta.sector_size; off +=
DIR_ENTRY_SIZE) {
38                DIR_ENTRY* dir = (DIR_ENTRY*)(sector_buffer + off);
39                if (is_free(dir))
40                    break;
41                if (!is_valid(dir))
42                    continue;
43                // 获取长文件名
44                to_longname(dir->DIR_Name, name, sizeof(name));
45                // 填入 buf
46                filler(buf, name, NULL, 0, 0);
47            }
48        }
49        if(root)
50            break;

```

```

51         clus = read_fat_entry(clus);
52     }
53     return 0;
54 }

```

fat16_read

读文件的主函数，从path对应的文件的offset字节处开始读取size字节的数据

先找到要读的起始位置所在簇，然后计算在此簇中的偏移量，再调用 `read_from_cluster_at_offset` 函数即可。

代码如下：

```

1  int fat16_read(const char *path, char *buffer, size_t size, off_t offset,
2                struct fuse_file_info *fi) {
3      printf("read(path='%s', offset=%ld, size=%lu)\n", path, offset, size);
4      if(path_is_root(path)) {
5          return -EISDIR;
6      }
7      DirEntrySlot slot;
8      DIR_ENTRY* dir = &(slot.dir);
9      int ret = find_entry(path, &slot);
10     if(ret < 0) {
11         return ret;
12     }
13     if(is_directory(dir->DIR_Attr)) {
14         return -EISDIR;
15     }
16     if(offset > dir->DIR_FileSize) {
17         return -EINVAL;
18     }
19     size = min(size, dir->DIR_FileSize - offset);
20     cluster_t clus = dir->DIR_FstClusL0;
21     size_t p = 0;
22     while (offset >= meta.cluster_size) {
23         offset -= meta.cluster_size;
24         clus = read_fat_entry(clus);
25     }
26     while (p < size) {
27         size_t len = min(meta.cluster_size - offset, size - p);
28         read_from_cluster_at_offset(clus, offset, buffer + p, len);
29         p += len;
30         offset = 0;
31         clus = read_fat_entry(clus);

```

```

32     }
33     return p;
34 }
```

后面写文件的代码和这里逻辑上是一样的，计算起始簇和偏移量。

任务二：创建、删除文件或目录

dir_entry_write

写入给定的目录项

先整个读取目录项所在的扇区，再将目录项写入到恰当位置，最后再写回扇区。

代码如下：

```

1  int dir_entry_write(DirEntrySlot slot) {
2      char sector_buffer[PHYSICAL_SECTOR_SIZE];
3      sector_read(slot.sector, sector_buffer);
4      memcpy(sector_buffer + slot.offset, &(slot.dir), sizeof(DIR_ENTRY));
5      sector_write(slot.sector, sector_buffer);
6      return 0;
7  }
```

这里提一下 slot 的数据结构：

- `DIR_ENTRY dir`
- `sector_t sector`
- `size_t offset`

存储了一个目录项的具体内容，以及它所在扇区，和它在该扇区中的偏移量。

注意：后面创建目录项时也要用到这个，其中第二和第三个成员指代目标地址

write_fat_entry

修改 FAT 表项

和读 FAT 表项一样计算位置，读取整个扇区，修改目标位置，再写回。

代码如下：

```

1  int write_fat_entry(cluster_t clus, cluster_t data) {
2      char sector_buffer[MAX_LOGICAL_SECTOR_SIZE];
3      size_t clus_off = clus * sizeof(cluster_t);
4      size_t sec_off = clus_off % meta.sector_size;
5      for(size_t i = 0; i < meta.fats; i++) {
6          sector_t fat_sec = meta.fat_sec + i * meta.sec_per_fat;
7          sector_t clus_sec = clus_off / meta.sector_size + fat_sec;
8          sector_read(clus_sec, sector_buffer);
9          memcpy(sector_buffer + sec_off, &data, sizeof(cluster_t));
10         sector_write(clus_sec, sector_buffer);
11     }
12     return 0;
13 }

```

alloc_clusters

分配簇

扫描 FAT 表，找 n 个空闲的簇，将其链接起来并返回第一个簇，注意要清空找到的簇。

代码如下：

```

1  int alloc_clusters(size_t n, cluster_t* first_clus) {
2      if (n == 0)
3          return CLUSTER_END;
4      // 用于保存找到的n个空闲簇，另外在末尾加上CLUSTER_END，共n+1个簇号
5      cluster_t *clusters = malloc((n + 1) * sizeof(cluster_t));
6      size_t allocated = 0; // 已找到的空闲簇个数
7      // 使用 read_fat_entry 函数来读取FAT表项的值，根据该值判断簇是否空闲。
8      for (size_t i = 2; i < meta.clusters; i++) {
9          if (!read_fat_entry(i)) {
10             clusters[allocated++] = i;
11             if (allocated == n) {
12                 break;
13             }
14         }
15     }
16     if(allocated != n) { // 找不到n个簇，分配失败
17         free(clusters);
18         return -ENOSPC;
19     }
20     // 找到了n个空闲簇，将CLUSTER_END加至末尾。
21     clusters[n] = CLUSTER_END;
22     // 清零要分配的簇

```

```

23     for(size_t i = 0; i < n; i++) {
24         int ret = cluster_clear(clusters[i]);
25         if(ret < 0) {
26             free(clusters);
27             return ret;
28         }
29     }
30     // 将每个簇连接到下一个
31     for(size_t i = 0; i < n; i++) {
32         write_fat_entry(clusters[i], clusters[i + 1]);
33     }
34     *first_clus = clusters[0];
35     free(clusters);
36     return 0;
37 }

```

fat16_mkdir

创建文件夹的主函数，创建 path 对应的文件夹

在给定路径里找空槽，创建目录项。

注意：新目录不是空，而有两个目录项，分别是 `.` 和 `..`，所以需要分配一个簇并且再创建两个目录项

代码如下：

```

1  int fat16_mkdir(const char *path, mode_t mode) {
2      printf("mkdir(path='%s', mode=%03o)\n", path, mode);
3      DirEntrySlot slot;
4      const char* filename = NULL;
5      // 找空槽
6      int ret = find_empty_slot(path, &slot, &filename);
7      if(ret < 0) {
8          return ret;
9      }
10     char shortname[11];
11     ret = to_shortcode(filename, MAX_NAME_LEN, shortname);
12     if(ret < 0) {
13         return ret;
14     }
15     // 分配簇
16     ret = alloc_clusters(1, &(slot.dir.DIR_FstClusL0));
17     if(ret < 0) {
18         return ret;
19     }

```



```

20     // 创建目录项
21     ret = dir_entry_create(slot, shortname, ATTR_DIRECTORY,
22                             slot.dir.DIR_FstClusL0, 2 * sizeof(DIR_ENTRY));
23     if(ret < 0) {
24         return ret;
25     }
26     const char DOT_NAME[] = ".";
27     const char DOTDOT_NAME[] = "..";
28     // 创建.目录项
29     slot.sector = cluster_first_sector(slot.dir.DIR_FstClusL0);
30     slot.offset = 0;
31     ret = dir_entry_create(slot, DOT_NAME, ATTR_DIRECTORY,
32                             slot.dir.DIR_FstClusL0, 2 * sizeof(DIR_ENTRY));
33     if(ret < 0) {
34         return ret;
35     }
36     // 创建..目录项
37     slot.offset = sizeof(DIR_ENTRY);
38     ret = dir_entry_create(slot, DOTDOT_NAME, ATTR_DIRECTORY, 0, 3 *
39                             sizeof(DIR_ENTRY));
40     if(ret < 0) {
41         return ret;
42     }
43     return 0;
44 }

```

fat16_rmdir

删除文件夹的主函数

先判断目录是否为空（忽略 `.` 和 `..`），再释放目录所占的所有簇，然后标记删除目录项。

注意：排除已删除项的干扰

代码如下：

```

1  int fat16_rmdir(const char *path) {
2      printf("rmdir(path='%s')\n", path);
3      if(path_is_root(path)) {
4          return -EBUSY;
5      }
6      DirEntrySlot slot;
7      int ret = find_entry(path, &slot);
8      if(ret < 0) {
9          return ret;

```

```

10     }
11     DIR_ENTRY* dir = &(slot.dir);
12     if(!is_directory(dir->DIR_Attr)) {
13         return -ENOTDIR;
14     }
15     // 读取目录项, 判断目录是否为空(忽略.和..)
16     cluster_t clus = dir->DIR_FstClusLO;
17     char sector_buffer[MAX_LOGICAL_SECTOR_SIZE];
18     while(is_cluster_inuse(clus)) {
19         // 读取目录项
20         sector_t first_sec = cluster_first_sector(clus);
21         for(size_t i = 0; i < meta.sec_per_clus; i++) {
22             sector_t sec = first_sec + i;
23             sector_read(sec, sector_buffer);
24             for(size_t off = 0; off < meta.sector_size; off +=
DIR_ENTRY_SIZE) {
25                 DIR_ENTRY* entry = (DIR_ENTRY*)(sector_buffer + off);
26                 // 忽略.和..
27                 if(is_dot(entry)) {
28                     continue;
29                 }
30                 // 不为空
31                 if(!is_free(entry) && !is_deleted(entry)) {
32                     return -ENOTEMPTY;
33                 }
34             }
35         }
36         clus = read_fat_entry(clus);
37     }
38     // 释放目录所占的所有簇
39     ret = free_clusters(dir->DIR_FstClusLO);
40     if(ret < 0) {
41         return ret;
42     }
43     // 删除目录项
44     dir->DIR_Name[0] = NAME_DELETED;
45     ret = dir_entry_write(slot);
46     if(ret < 0) {
47         return ret;
48     }
49     return 0;
50 }

```

任务三：写文件、裁剪文件

write_to_cluster_at_offset

在给定簇的给定位置写入给定大小的数据

先计算数据所在扇区，和扇区内偏移量，然后循环写入每一个扇区，直到写入完全。

代码如下：

```

1  ssize_t write_to_cluster_at_offset(cluster_t clus, off_t offset, const
    char* data, size_t size) {
2      assert(offset + size <= meta.cluster_size); // offset + size 必须小于簇
    大小
3      char sector_buffer[PHYSICAL_SECTOR_SIZE];
4      size_t pos = 0;
5      // 计算数据所在扇区
6      sector_t first_sec = cluster_first_sector(clus);
7      sector_t sec = first_sec + offset / meta.sector_size;
8      // 计算数据在扇区中的偏移
9      offset %= meta.sector_size;
10     while (pos < size) {
11         // 读取扇区
12         sector_read(sec, sector_buffer);
13         // 写入数据
14         size_t write_size = min(size - pos, meta.sector_size - offset);
15         memcpy(sector_buffer + offset, data + pos, write_size);
16         // 写回扇区
17         sector_write(sec, sector_buffer);
18         // 更新状态
19         pos += write_size;
20         sec++;
21         offset = 0;
22     }
23     return pos;
24 }
```

file_reserve_clusters

为文件分配新的簇以满足大小要求

先计算需要多少簇，然后分两种情况，如果文件原本没有簇，那么直接分配即可，否则需要分配恰当数量的簇，再与原来的拼接。

代码如下：

```

1  int file_reserve_clusters(DIR_ENTRY* dir, size_t size) {
2      // 计算需要多少簇
3      size_t need_clus = size / meta.cluster_size + 1;
4      // 文件没有簇
5      if(!is_cluster_inuse(dir->DIR_FstClusL0)) {
6          int ret = alloc_clusters(need_clus, &(dir->DIR_FstClusL0));
7          if(ret < 0) {
8              return -1;
9          }
10     }
11     // 文件已有簇
12     else {
13         // 找最后一个簇
14         cluster_t clus = dir->DIR_FstClusL0;
15         need_clus--;
16         while(is_cluster_inuse(read_fat_entry(clus))) {
17             clus = read_fat_entry(clus);
18             need_clus--;
19         }
20         if (need_clus <= 0) {
21             return 0;
22         }
23         cluster_t temp;
24         int ret = alloc_clusters(need_clus, &temp);
25         if(ret < 0) {
26             return -1;
27         }
28         // 连接簇
29         write_fat_entry(clus, temp);
30     }
31     return 0;
32 }

```

fat16_write

写文件的主函数

四步走：

1. 找到目录项
2. 调整文件大小
3. 写入数据
4. 更新信息

各个步骤的实现方法前面都有过，这里就略过了。

代码如下：

```

1  int fat16_write(const char *path, const char *data, size_t size, off_t
    offset,
2
3      struct fuse_file_info *fi) {
4      printf("write(path='%s', offset=%ld, size=%lu)\n", path, offset,
    size);
5      // 找到文件目录项
6      DirEntrySlot slot;
7      int ret = find_entry(path, &slot);
8      if(ret < 0) {
9          return -1;
10     }
11     DIR_ENTRY* dir = &(slot.dir);
12     // 扩展文件大小
13     if(dir->DIR_FileSize < offset + size) {
14         // printf("nobuf there~");
15         ret = file_reserve_clusters(dir, offset + size);
16         if(ret < 0) {
17             return -1;
18         }
19     }
20     // 找到写入数据的起始簇
21     cluster_t clus = dir->DIR_FstClusL0;
22     int temp = offset / meta.cluster_size;
23     while(temp--) {
24         clus = read_fat_entry(clus);
25     }
26     off_t back_up = offset;
27     offset %= meta.cluster_size;
28     // 写入数据
29     int pos = 0;
30     while(pos < size) {
31         size_t write_size = min(size - pos, meta.cluster_size - offset);
32         // printf("nobuf there~");
33         ret = write_to_cluster_at_offset(clus, offset, data + pos,
    write_size);
34         if(ret < 0) {
35             return -1;
36         }
37         pos += write_size;
38         offset = 0;
39         clus = read_fat_entry(clus);
40     }

```

```

40     // 更改文件大小
41     dir->DIR_FileSize = max(dir->DIR_FileSize, back_up + size);
42     // 更新目录项
43     dir_entry_write(slot);
44     return pos;
45 }

```

fat16_truncate

裁剪文件大小

也是先找到目录项，然后得到文件原大小，再根据情况决定扩大或缩小。

代码如下：

```

1  int fat16_truncate(const char *path, off_t size, struct fuse_file_info*
    fi) {
2      printf("truncate(path='%s', size=%lu)\n", path, size);
3      // 找到文件目录项
4      DirEntrySlot slot;
5      int ret = find_entry(path, &slot);
6      if(ret < 0) {
7          return -1;
8      }
9      DIR_ENTRY* dir = &(slot.dir);
10     // 计算簇数
11     int n1 = dir->DIR_FileSize / meta.cluster_size + 1;
12     int n2 = size / meta.cluster_size + 1;
13     // 文件大小不变
14     if(n1 == n2) {}
15     // 文件变小
16     else if (n1 > n2) {
17         // 找到最后一个簇
18         cluster_t clus = dir->DIR_FstClusL0;
19         cluster_t temp; // 记录最后一个簇
20         while(n2--) {
21             if (n2 == 0)
22                 temp = clus;
23             clus = read_fat_entry(clus);
24         }
25         // 结束符提前
26         write_fat_entry(temp, CLUSTER_END);
27         // 释放簇
28         free_clusters(clus);
29     }
30     // 文件变大

```

```

31     else {
32         // 扩展文件大小
33         ret = file_reserve_clusters(dir, size);
34         if(ret < 0) {
35             return -1;
36         }
37     }
38     // 更新文件大小
39     dir->DIR_FileSize = size;
40     // 更新目录项
41     dir_entry_write(slot);
42     return 0;
43 }

```

任务四：性能优化

由于期末压力，这个选做实在没时间写，摆摆了。

实验结果

测试一：读、写、创建、删除测试，运行结果如下：

```

问题 2  输出  调试控制台  终端  端口
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 PASSED [ 78%]
fat16_test.py::TestFat16Write::test2_file_truncate_shrink PASSED [ 84%]
fat16_test.py::TestFat16Write::test2_seq_write_small_files PASSED [ 89%]
fat16_test.py::TestFat16Write::test3_seq_write_large_file PASSED [ 94%]
fat16_test.py::TestFat16Write::test4_rand_write_large_file PASSED [100%]

===== 还需要单独安装 fat16 包，即运行：===== 19 passed in 7.14s =====
> $ fusermount -zu ./fat16
ningli@liano:~/os_lab/lab4$

```

可以看出，测试样例全部通过。

测试二：性能测试，运行结果如下：

```
问题 2 输出 调试控制台 终端 端口
Read 1200/2000 times.
Read 1300/2000 times.
Read 1400/2000 times.
Read 1500/2000 times.
Read 1600/2000 times.
Read 1700/2000 times.
Read 1800/2000 times.
Read 1900/2000 times.
22.812475456000357
std time:
22.497754376000103
your time:需要单独安装 fuse3 包, 即运行:
22.812475456000357
ningli@liano:~/os_lab/lab4$
```

性能大概是基准测试的 98%，符合要求。

总结

实验难度有点大，主要是不知道怎么调试，但磨了两天还是磨出来了，写完之后思路还是比较清晰的。

建议：别把实验放在期末周！🤖