OS lab4 FAT文件系统的实现

PB21081601 张芷苒

1 实验环境

- VirtualBox
- OS: Ubuntu 20.04 LTS
- Linux内核版本: 5.9.0
- libfuse3

2 实验任务

1. 任务一：实现FAT文件系统读操作（满分5分）
   (a) 能够运行 tree ， ls 命令查看文件目录结构 （3分）
   (b) 能够正确读取小于一个簇的文件，通过短文件读取测试（1分）
   (c) 能够正确读取长文件，通过长文件读取测试（1分）
2. 任务二：实现FAT文件系统创建/删除文件、目录操作（满分2分）
   (a) 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（1分）
   (b) 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（1分）
3. 任务三：实现FAT文件系统写操作（满分2分）
   - 能够正确写入文件，通过文件写入、截断测试（2分）
4. 任务四：FAT 文件系统性能优化（满分1分）
   (a) 在模拟磁盘环境下，性能达到基准测试 50% （1分）
   (b) 在模拟磁盘环境下，性能达到基准测试的 200% （额外+1分）

3 实验过程

以下介绍`simple_fat16.c`中todo部分的代码填空实现过程解释。

```
cluster_t read_fat_entry(cluster_t clus)
{
    char sector_buffer[PHYSICAL_SECTOR_SIZE];
    // 1. 计算簇号 clus 对应的 FAT 表项的偏移量，并计算表项所在的扇区号
    uint32_t fat_offset = clus * 4; // 在 FAT32 中，每个表项占 4 字节
    uint32_t fat_sector = FIRST_FAT_SECTOR + (fat_offset / PHYSICAL_SECTOR_SIZE);
    uint32_t entry_offset = fat_offset % PHYSICAL_SECTOR_SIZE;

    // 2. 使用 sector_read 函数读取该扇区
    sector_read(fat_sector, sector_buffer);

    // 3. 计算簇号 clus 对应的 FAT 表项在该扇区中的偏移量
    // 在此，我们已经在第一步中计算了 entry_offset。

    // 4. 从该偏移量处读取对应表项的值，并返回
    return *(cluster_t*)&sector_buffer[entry_offset];
}
```

首先，根据给定的簇号，我们需要计算出这个簇号对应的 FAT 表项在 FAT 表中的偏移量。由于我们是在 FAT32 文件系统中，每一个 FAT 表项占用 4 字节，所以，这个偏移量就是簇号乘以 4。然后，我们需要计算出这个 FAT 表项在哪一个扇区，扇区号就是首个 FAT 扇区号加上偏移量除以每个扇区的大小。偏移量对每个扇区的大小取余，就可以得到这个 FAT 表项在扇区中的偏移量。

然后，我们使用 `sector_read` 函数读取这个扇区到缓冲区。

最后，我们从缓冲区中读取这个 FAT 表项的值。注意这里我们使用了一个类型转换，因为我们从缓冲区中读取的是一个字节序列，我们需要将其转换为我们想要的类型，即 `cluster_t`。

这样就成功地读取了一个 FAT 表项，并将它返回。

```
int find_entry_in_sectors(const char* name, size_t len,
            sector_t from_sector, size_t sectors_count,
            DirEntrySlot* slot) {
```

```
    char buffer[PHYSICAL_SECTOR_SIZE];
    // 对每一个待查找的扇区:
    for(size_t i = 0; i < sectors_count; i++) {
        // 1. 使用 sector_read 函数读取从扇区号 from_sector 开始的第 i 个扇区
        sector_read(from_sector + i, buffer);
        int empty_entry_found = 0;

        // 2. 对该扇区中的每一个目录项，检查是否是待查找的目录项（注意检查目录项是否合法）
        for(size_t off = 0; off < PHYSICAL_SECTOR_SIZE; off += DIR_ENTRY_SIZE) {
            DIR_ENTRY* dir = (DIR_ENTRY*)(buffer + off);
            // 3. 如果是待查找的目录项，将该目录项的信息填入 slot 中，并返回 FIND_EXIST
            if (memcmp(dir->DIR_Name, name, len) == 0) {
                slot->sector = from_sector + i;
                slot->offset = off;
                return FIND_EXIST;
            }
            // 4. 如果不是待查找的目录项，检查该目录项是否为空，如果为空，将该目录项的信息填入 slot 中，并返回 FIND_EMPTY
            if (!empty_entry_found && dir->DIR_Name[0] == 0xE5 || dir->DIR_Name[0] == 0x00) {
                empty_entry_found = 1;
                slot->sector = from_sector + i;
                slot->offset = off;
            }
        }
        // 5. 如果不是待查找的目录项，且该扇区中的所有目录项都不为空，返回 FIND_FULL
        if(empty_entry_found) {
            return FIND_EMPTY;
        }
    }
    return FIND_FULL;
}
```

首先循环遍历每个扇区，使用 `sector_read` 函数读取扇区内容到缓冲区。然后，对扇区中的每个目录项，我们检查它的名字是否是我们要查找的。如果是，我们填充 `slot` 结构并返回 `FIND_EXIST`。如果不是，我们检查它是否为空。如果是空的，我们记录第一个空的条目的位置。如果在一个扇区的遍历结束后，我们找到了空的条目，我们返回 `FIND_EMPTY`。如果所有的条目都已经被使用，我们就继续遍历下一个扇区。如果我们已经遍历了所有的扇区但仍然没有找到空的条目，我们返回 `FIND_FULL`。

```
int find_entry_internal(const char* path, DirEntrySlot* slot,
const char** remains) {
    // ....
    if(level == 0) {
        // TODO:1.1: 设置根目录的扇区号和扇区数
        sector_t root_sec = meta.root_dir_sectors;    // 获取根目
录的扇区号
        size_t nsec = meta.root_dir_sectors_count;    // 获取根目
录的扇区数
        // 使用 find_entry_in_sectors 寻找相应的目录项
        state = find_entry_in_sectors(*remains, len, root_sec,
nsec, slot);
        // ...
    } else {
        // ...
        while (is_cluster_inuse(clus)) {
            // TODO:1.2: 在 clus 对应的簇中查找每个目录项。
            sector_t clus_sector = cluster_to_sector(clus);    //
转换簇号为扇区号
            state = find_entry_in_sectors(*remains, len,
clus_sector, meta.sectors_per_cluster, slot);
            // ...
        }
    }
    // ...
    return state;
}
```

这样做的目的是为了实现对文件系统中指定路径的查找操作。在这个过程中，我们需要分别处理根目录和非根目录的情况。对于根目录，由于其位置是固定的，因此我们可以直接从元数据中获取其扇区号和扇区数。对于非根目录，我们需要首先将簇号转换为扇区号，然后在这些扇区中查找目录项。这样做的主要目的是为了将路径映射到实际的文件系统结构中，从而可以执行诸如打开文件、读写文件等操作.

```
int fat16_readdir(const char *path, void *buf, fuse_fill_dir_t
filler, off_t offset,
                  struct fuse_file_info *fi, enum
fuse_readdir_flags flags) {
    // ...

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

```
        sector_t sec = first_sec + i;
        sector_read(sec, sector_buffer);
        // TODO:1.5: 对扇区中每个目录项:
        for(size_t off=0; off < meta.sector_size; off += DIR_ENTRY_SIZE) {
            DIR_ENTRY* dir_entry = (DIR_ENTRY*)(sector_buffer + off);
            // 1. 确认其是否是表示文件或目录的项
            if(is_valid_dir_entry(dir_entry)) {
                // 2. 从 FAT 文件名中，获得长文件名
                to_longname(dir_entry->DIR_Name, name);
                // 3. 使用 filler 填入 buf
                if(filler(buf, name, NULL, 0) != 0) {
                    return -ENOMEM;
                }
            } else if(dir_entry->DIR_Name[0] == '\0') {
                // 4. 找到空项即可结束查找
                return 0;
            }
        }
    }

    // ...

    return 0;
}
```

这样做的目的是为了实现读取目录内容的功能。对于每个扇区中的每个目录项，我们首先检查是否是一个有效的目录项。如果是，我们将其文件名转换为长文件名，并使用`filler`函数将其写入缓冲区。如果遇到一个空的目录项（即其文件名的第一个字符为`\0`），我们就可以停止搜索，因为这意味着该扇区后面的所有目录项都是空的。如果在使用`filler`函数时返回非零值，我们将返回一个表示内存不足的错误码。

这样做的主要原因是，目录实际上是一种特殊的文件，其内容是一系列的目录项。在FAT16文件系统中，每个目录项包括文件名、属性、大小等信息。因此，通过读取目录项，我们就可以获取到目录中所有文件的信息，从而实现了读取目录内容的功能。

```
int fat16_read(const char *path, char *buffer, size_t size, off_t offset,
               struct fuse_file_info *fi) {
    // ...

```

```
    cluster_t clus = dir->DIR_FstClusLO;
    size_t p = 0;
    // TODO:1.6: clus 初始为该文件第一个簇，利用 read_from_cluster_at_offset 函数，从正确的簇中读取数据。
    while(p < size) {
        off_t clus_offset = offset % (meta.sec_per_clus * meta.sector_size);
        size_t clus_size = min(meta.sec_per_clus * meta.sector_size - clus_offset, size - p);
        size_t read_size = read_from_cluster_at_offset(clus, clus_offset, clus_size, buffer + p);
        p += read_size;
        offset += read_size;

        if(read_size < clus_size) {
            break;
        }

        clus = read_fat_entry(clus);
        if(!is_cluster_inuse(clus)) {
            break;
        }
    }

    return p;
}
```

这样做的原因是为了实现从文件中读取数据的功能。我们首先计算要从当前簇中读取的数据量，这取决于当前簇还剩下多少数据，以及我们还需要读取多少数据。然后，我们使用 `read_from_cluster_at_offset` 函数从当前簇的指定偏移量开始读取数据。

然后，我们更新已读取的数据量和偏移量，并检查是否已经读取了所有需要的数据。如果还有数据需要读取，我们就读取下一个簇的FAT项，并判断是否需要读取下一个簇的数据。

这样做的主要原因是，文件系统是以簇为单位进行数据存储的。因此，当我们需要读取一个文件的数据时，我们需要首先确定数据所在的簇，然后从这个簇中读取数据。由于一个文件的数据可能分布在多个簇中，因此我们可能需要从多个簇中读取数据。同时，我们还需要处理数据跨簇的情况，即一个数据可能开始于一个簇，结束于另一个簇.

```
int dir_entry_write(DirEntrySlot slot) {
    char sector_buffer[PHYSICAL_SECTOR_SIZE];
    // TODO:2.1:
```

```
    //  1. 读取 slot.dir 所在的扇区
    sector_t sec = slot.sec;
    sector_read(sec, 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(sec, sector_buffer);
    return 0;
}
```

这样做的原因是，目录项的写入操作在文件系统中是比较常见的操作，如创建新文件、删除文件、修改文件属性等。在 FAT16 文件系统中，目录项存储在扇区中，每个扇区可以存储多个目录项。但是，由于硬件的限制，我们无法直接修改扇区中的单个目录项，只能通过读取整个扇区的内容，修改扇区中的指定部分，然后将整个扇区写回的方式来修改单个目录项。

我们首先使用 `sector_read` 函数读取目录项所在的扇区的内容，然后使用 `memcpy` 函数将新的目录项的内容复制到扇区缓冲区的相应位置，最后再使用 `sector_write` 函数将扇区缓冲区的内容写回到扇区中，完成目录项的修改。

这样做的好处是，不仅可以修改单个目录项，还可以避免频繁地读写硬盘，提高了文件系统的性能。同时，由于目录项的大小是固定的，因此，我们可以直接使用 `memcpy` 函数进行复制，无需进行复杂的计算，简化了代码的编写。

```
int write_fat_entry(cluster_t clus, cluster_t data) {
    char sector_buffer[MAX_LOGICAL_SECTOR_SIZE];
    size_t clus_off = clus * sizeof(cluster_t);
    sector_t clus_sec = clus_off / meta.sector_size;
    size_t sec_off = clus_off % meta.sector_size;
    for(size_t i = 0; i < meta.fats; i++) {
        // TODO:2.2: 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
        sector_t sec = meta.reserved_sectors + i *
meta.fat_size_sectors + clus_sec;

        //   2. 读取该扇区并在对应位置写入数据
        sector_read(sec, sector_buffer);
```

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

        //   3. 将该扇区写回
        sector_write(sec, sector_buffer);
    }
    return 0;
}
```

这样做的原因是，FAT16文件系统中，每一个簇在FAT（File Allocation Table，文件分配表）中都有一个对应的表项，用于记录该簇的状态，如空闲、已使用、坏簇等，以及指向文件的下一个簇（如果该簇已被文件使用）。

函数 `write_fat_entry` 用于修改FAT表项的值。当我们需要创建新文件、扩展文件、删除文件或者修复坏簇等操作时，可能需要修改FAT表项的值。

我们首先计算需要修改的FAT表项所在的扇区以及在该扇区中的偏移量，然后读取该扇区的内容，修改相应的表项，最后将扇区的内容写回到扇区中。因为FAT16文件系统可能有多个FAT表，为了保持一致性，我们需要对所有的FAT表进行相同的修改。

由于硬件的限制，我们无法直接修改扇区中的单个表项，只能通过读取整个扇区的内容，修改扇区中的指定部分，然后将整个扇区写回的方式来修改表项。

```
int alloc_clusters(size_t n, cluster_t* first_clus) {
    if (n == 0)
        return CLUSTER_END;

    cluster_t *clusters = malloc((n + 1) * sizeof(cluster_t));
    size_t allocated = 0;

    for(cluster_t i = 2; i < meta.data_clusters + 2; i++) {
        cluster_t fat_entry;
        read_fat_entry(i, &fat_entry);
        if(fat_entry == CLUSTER_FREE) {
            clusters[allocated++] = i;
            if(allocated == n) {
                break;
            }
        }
    }

    if(allocated != n) {
        free(clusters);
```

```
        return -ENOSPC;
    }

    clusters[n] = CLUSTER_END;

    for(size_t i = 0; i < n; i++) {
        int ret = cluster_clear(clusters[i]);
        if(ret < 0) {
            free(clusters);
            return ret;
        }
    }

    for(size_t i = 0; i < n; i++) {
        write_fat_entry(clusters[i], clusters[i + 1]);
    }

    *first_clus = clusters[0];
    free(clusters);
    return 0;
}
```

当我们需要创建新文件或者扩展已有的文件时，我们需要在文件系统中找到一些空闲的簇并分配给文件。而文件系统中的簇是通过FAT表来组织的，每个簇在FAT表中有一个对应的表项，记录该簇的状态（空闲、已使用、坏簇等）以及文件的下一个簇（如果该簇已被文件使用）。

在分配簇时，我们首先扫描FAT表，找到一些空闲的簇（FAT表项的值为0），然后修改这些簇的FAT表项，使得这些簇通过FAT表项连接在一起，形成一个簇链，最后一个簇的FAT表项的值为0xFFFF，表示这是文件的最后一个簇。分配的过程中，我们还需要清空新分配的簇的内容，避免在文件中留下旧的数据。

在这个过程中，我们使用`cluster_clear`函数清空簇的内容，使用`write_fat_entry`函数修改FAT表项的值。

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

    if(path_is_root(path)) {
        return -EEXIST;
    }

    cluster_t clus;
```

```
    int ret = alloc_clusters(1, &clus);
    if(ret < 0) {
        return ret;
    }

    const char DOT_NAME[] =    ".          ";
    const char DOTDOT_NAME[] = "..         ";

    DIR_ENTRY dot;
    dir_entry_create(&dot, DOT_NAME, ATTR_DIRECTORY, clus, 2 * sizeof(DIR_ENTRY));
    cluster_write(clus, 0, (char*)&dot, sizeof(DIR_ENTRY));

    DIR_ENTRY dotdot;
    dir_entry_create(&dotdot, DOTDOT_NAME, ATTR_DIRECTORY, 0, 2 * sizeof(DIR_ENTRY));
    cluster_write(clus, sizeof(DIR_ENTRY), (char*)&dotdot, sizeof(DIR_ENTRY));

    ret = fat16_mknod(path, mode, clus, ATTR_DIRECTORY, 2 * sizeof(DIR_ENTRY));
    if(ret < 0) {
        return ret;
    }

    return 0;
}
```

在FAT文件系统中，创建新目录需要分配一个新簇并在其中创建两个特殊的目录项：`.`和`..`。`.`指向新目录自身，而`..`指向新目录的父目录。这两个目录项在新目录的簇中占据第一和第二个位置，紧随其后的就是新目录的子目录项。创建目录的最后一步是在父目录的簇中创建一个新的目录项，指向新目录的簇。

在这个过程中，我们使用了`alloc_clusters`函数来分配新簇，`dir_entry_create`函数来创建目录项，`cluster_write`函数来将目录项写入簇中，以及`fat16_mknod`函数在父目录的簇中创建新目录的目录项。需要注意的是，`.`和`..`目录项的文件大小字段应为新目录目录项的总大小，因为这两个目录项本身也算作新目录的内容。

```
int fat16_rmdir(const char *path) {
    printf("rmdir(path='%s')\n", path);
    if(path_is_root(path)) {
        return -EBUSY;
```

```
    }

    DirEntrySlot slot;
    DIR_ENTRY* dir = &(slot.dir);
    int ret = find_entry(path, &slot);
    if(ret < 0) {
        return ret;
    }
    if(!is_directory(dir->DIR_Attr)) {
        return -ENOTDIR;
    }

    DIR_ENTRY buf[16]; // suppose the cluster size is 512
    size_t read_bytes = cluster_read(dir->DIR_FstClusLO, 0,
(char*)buf, sizeof(buf));
    if (read_bytes < 2 * sizeof(DIR_ENTRY)) {
        return -EIO; // Error: cannot read the directory entries
    }

    // Check if directory is empty (exclude "." and "..")
    for (int i = 2; i < read_bytes / sizeof(DIR_ENTRY); ++i) {
        if (buf[i].DIR_Name[0] ≠ SLOT_EMPTY &&
buf[i].DIR_Name[0] ≠ SLOT_DELETED) {
            return -ENOTEMPTY;
        }
    }

    // Clear the cluster
    ret = cluster_clear(dir->DIR_FstClusLO);
    if (ret < 0) {
        return ret;
    }

    // Write CLUSTER_FREE to FAT
    ret = write_fat_entry(dir->DIR_FstClusLO, CLUSTER_FREE);
    if (ret < 0) {
        return ret;
    }

    // Delete the directory entry in parent directory
    dir->DIR_Name[0] = SLOT_DELETED;
    ret = dir_entry_write(slot);
    if (ret < 0) {
        return ret;
```

```
    }

    return 0;
}
```

在FAT文件系统中，删除一个目录涉及到的步骤包括：检查目录是否为空，释放目录所占用的簇，将簇的FAT表项设置为CLUSTER_FREE，以及在父目录的簇中删除目录的目录项。

我们首先检查目录是否为空。由于每个目录都包含两个特殊的目录项“.”和“..”，所以我们需要从第三个目录项开始检查，如果发现任何非空的目录项，则返回-ENOTEMPTY错误。注意，这里我们假设簇的大小是512字节，这意味着每个簇可以容纳16个目录项。

如果目录为空，我们就可以删除它。首先，我们使用cluster_clear函数来清空目录所在的簇。然后，我们使用write_fat_entry函数将簇的FAT表项设置为CLUSTER_FREE，表示该簇现在是空闲的。最后，我们在父目录的簇中删除目录的目录项。这通过将目录项的第一个字符设置为SLOT_DELETED，然后使用dir_entry_write函数将目录项写回簇来完成。

```
ssize_t write_to_cluster_at_offset(cluster_t clus, off_t offset, const char* data, size_t size) {
    assert(offset + size ≤ meta.cluster_size);  // offset + size 必须小于簇大小
    char sector_buffer[PHYSICAL_SECTOR_SIZE];

    size_t pos = 0;
    size_t sec_idx = offset / meta.sector_size;  // 计算第一个扇区索引
    size_t sec_off = offset % meta.sector_size;  // 计算第一个扇区偏移量

    // 遍历所有涉及的扇区
    for (; pos < size;) {
        sector_t sec = cluster_to_sector(clus) + sec_idx;
        int ret = sector_read(sec, sector_buffer);  // 读取扇区内容
        if (ret < 0) {
            return ret;  // 读取失败，返回错误代码
        }

        // 计算此次要写入的数据大小
        size_t write_size = meta.sector_size - sec_off;
        if (write_size > size - pos) {
            write_size = size - pos;
```

```
        }

        // 将数据写入扇区缓冲区
        memcpy(sector_buffer + sec_off, data + pos, write_size);
        ret = sector_write(sec, sector_buffer);  // 写入扇区
        if (ret < 0) {
            return ret;  // 写入失败，返回错误代码
        }

        // 更新变量
        pos += write_size;
        sec_idx++;
        sec_off = 0;
    }

    return pos;
}
```

我们要将数据写入编号为`clus`的簇的`offset`位置，这需要考虑多个扇区的情况，因为一个簇可能包含多个扇区。我们首先通过偏移量计算出第一个要写入的扇区的索引和该扇区内的偏移量。然后我们遍历所有涉及的扇区，对于每一个扇区，我们先读取它的内容到缓冲区中，然后将数据写入缓冲区的相应位置，最后将缓冲区的内容写回扇区。我们根据已写入的数据大小来更新写入位置，并继续处理下一个扇区，直到所有数据都被写入。

```
// 计算需要多少簇
    size_t needed_clusters = (size + meta.cluster_size - 1) / meta.cluster_size;

    // 如果文件没有簇，直接分配足够的簇
    if (dir->DIR_FstClusLO == 0) {
        cluster_t first_clus;
        int ret = alloc_clusters(needed_clusters, &first_clus);
        if (ret < 0) {
            return ret;  // 分配失败，返回错误代码
        }
        dir->DIR_FstClusLO = first_clus;
    }
    // 如果文件已有簇，找到最后一个簇（哪个簇是当前该文件的最后一个簇？），并计算需要额外分配多少个簇
    else {
        cluster_t cur_clus = dir->DIR_FstClusLO;
        cluster_t next_clus;
```

```
        size_t actual_clusters = 0;  // 文件已有的簇数

        while ((read_fat_entry(cur_clus, &next_clus) == 0) &&
(next_clus != CLUSTER_END)) {
            cur_clus = next_clus;
            actual_clusters++;
        }

        // 分配额外的簇，并将分配好的簇连在最后一个簇后
        if (needed_clusters > actual_clusters) {
            cluster_t first_clus;
            int ret = alloc_clusters(needed_clusters -
actual_clusters, &first_clus);
            if (ret < 0) {
                return ret;  // 分配失败，返回错误代码
            }
            write_fat_entry(cur_clus, first_clus);
        }
    }

    dir->DIR_FileSize = size;
    return 0;
```

首先，我们计算需要多少簇来容纳所需的大小。如果文件当前没有簇，我们直接分配所需数量的簇。如果文件已经有了一些簇，我们找到文件最后一个簇（通过遍历FAT表直到找到指向CLUSTER_END的表项），并计算需要额外分配多少个簇。然后，我们分配这些额外的簇，并将这些新分配的簇连接到文件的最后一个簇之后。最后，我们更新文件的大小。

```
int fat16_write(const char *path, const char *data, size_t size,
off_t offset,
                struct fuse_file_info *fi) {
    printf("write(path='%s', offset=%ld, size=%lu)\n", path,
offset, size);

    // 查找文件
    DirEntrySlot slot;
    int ret = find_entry(path, &slot);
    if(ret < 0) {
        return ret;  // 文件未找到，返回错误代码
    }

    // 检查是否是目录
    if(is_directory(slot.dir.DIR_Attr)) {
```

```
        return -EISDIR;  // 是目录，返回错误代码
    }

    // 确保文件足够大以包含所有写入的数据
    size_t final_size = offset + size;
    if(final_size > slot.dir.DIR_FileSize) {
        ret = file_reserve_clusters(&slot.dir, final_size);
        if(ret < 0) {
            return ret;  // 簇分配失败，返回错误代码
        }
    }

    // 写入数据
    cluster_t clus = slot.dir.DIR_FstClusLO;
    size_t clus_size = meta.cluster_size;
    size_t written_bytes = 0;
    while(written_bytes < size) {
        size_t clus_offset = offset % clus_size;
        size_t write_size = min(size - written_bytes, clus_size
- clus_offset);
        ret = write_to_cluster_at_offset(clus, clus_offset, data
+ written_bytes, write_size);
        if(ret < 0) {
            return ret;  // 写入失败，返回错误代码
        }

        written_bytes += write_size;
        offset += write_size;

        // 若写入过程跨越簇，获取下一个簇
        if(offset / clus_size > clus) {
            read_fat_entry(clus, &clus);
        }
    }

    // 更新目录项
    update_dir_entry(&slot);

    return written_bytes;
}
```

此函数首先找到给定路径的文件的目录项。然后，它检查找到的项是否是目录，如果是，则返回错误。接着，它检查文件的当前大小是否足够容纳从给定偏移开始的所有写入的数据。如果文件大小不够，它会尝试扩展文件到足够的大小。然后，它开始将数据写入文件，注意处理跨越簇边界的情况。最后，它会更新目录项来反映任何可能的更改，如文件大小。

```
int fat16_truncate(const char *path, off_t size, struct fuse_file_info* fi) {
    printf("truncate(path='%s', size=%lu)\n", path, size);

    // 查找文件
    DirEntrySlot slot;
    int ret = find_entry(path, &slot);
    if(ret < 0) {
        return ret;  // 文件未找到，返回错误代码
    }

    // 检查是否是目录
    if(is_directory(slot.dir.DIR_Attr)) {
        return -EISDIR;  // 是目录，返回错误代码
    }

    // 获取当前文件大小
    size_t cur_size = slot.dir.DIR_FileSize;

    if(size < cur_size) {
        // 如果需要的大小小于当前大小，释放多余的簇
        size_t required_clusters = (size + meta.cluster_size - 1) / meta.cluster_size;
        size_t current_clusters = (cur_size + meta.cluster_size - 1) / meta.cluster_size;

        // 获取要删除的第一个簇
        cluster_t clus_to_remove = slot.dir.DIR_FstClusLO;
        for(size_t i = 0; i < required_clusters; ++i) {
            read_fat_entry(clus_to_remove, &clus_to_remove);
        }

        // 释放多余的簇
        cluster_t next_clus;
        while(clus_to_remove != CLUSTER_END) {
            read_fat_entry(clus_to_remove, &next_clus);
            write_fat_entry(clus_to_remove, CLUSTER_FREE);
```

```
            clus_to_remove = next_clus;
        }

        // 将最后一个簇设置为结束簇
        if(required_clusters > 0) {
            cluster_t last_clus = slot.dir.DIR_FstClusLO;
            for(size_t i = 1; i < required_clusters; ++i) {
                read_fat_entry(last_clus, &last_clus);
            }
            write_fat_entry(last_clus, CLUSTER_END);
        }

        // 更新文件大小
        slot.dir.DIR_FileSize = size;
    } else if(size > cur_size) {
        // 如果需要的大小大于当前大小，分配更多的簇，并将新的部分置零
        ret = file_reserve_clusters(&slot.dir, size);
        if(ret < 0) {
            return ret;  // 簇分配失败，返回错误代码
        }

        // 将新的部分置零
        size_t bytes_to_zero = size - cur_size;
        char* zero_buffer = calloc(bytes_to_zero, 1);
        ret = write_to_cluster_at_offset(slot.dir.DIR_FstClusLO,
cur_size, zero_buffer, bytes_to_zero);
        free(zero_buffer);
        if(ret < 0) {
            return ret;  // 写入失败，返回错误代码
        }
    }

    // 更新目录项
    update_dir_entry(&slot);

    return 0;
}
```

此函数首先找到给定路径的文件的目录项。然后，它检查找到的项是否是目录，如果是，则返回错误。接着，它检查新的大小是否小于当前文件大小。如果新的大小更小，它会释放任何不再需要的簇，并将最后一个需要的簇设置为结束簇。如果新的大小更大，它会分配更多的簇，并将新的部分置零。最后，它更新目录项以反映任何可能的更改，如文件大小。