

操作系统专题实践 - 实验3

71123329 段嘉文

文件系统的实现

操作系统专题实践 - 实验3

文件系统的实现

实验目的

实验内容

具体要求

设计思路

主要数据结构及其说明

实现文件系统核心功能

编译与测试

实验体会

参考资料

实验目的

通过实验完整了解文件系统实现机制，掌握用户态文件系统的设计与实现方法，理解VFS（虚拟文件系统）的工作原理，熟悉FUSE（Filesystem in Userspace）框架的使用，深入理解文件系统的基本操作接口及其实现细节。

实验内容

实现具有设备创建、分区格式化、注册文件系统、文件夹操作、文件操作功能的完整文件系统。

具体要求

- 实现文件的创建和删除功能
- 实现文件的读写操作
- 实现目录的创建和删除功能
- 能够正确处理文件属性（大小、时间等）

5. 实现文件系统的挂载和卸载
6. 支持基本的ls、cat、echo等命令操作
7. 确保文件系统操作的正确性和稳定性

设计思路

本实验原本计划实现内核态文件系统，但考虑到以下因素，最终选择了基于FUSE的用户态文件系统方案：

内核态文件系统的挑战：

- 开发难度大，需要深入理解VFS接口和内核编程
- 调试困难，无法使用常规调试工具，只能通过printk和dmesg
- 风险高，代码错误可能导致系统崩溃
- 时间成本高，完整实现需要大量时间
- 内核版本差异大，不同版本的VFS接口存在较大变化

FUSE方案的优势：

- 开发简单，使用普通C语言编程，无需内核编程知识
- 调试方便，可以使用gdb、printf等常规调试手段
- 安全性高，程序崩溃不会影响系统稳定性
- 快速实现，1-2天即可完成基础版本
- 同样满足实验要求，涵盖文件系统的核心概念

本次实现的文件系统命名为SimpleFS，采用纯内存存储方式，所有数据保存在内存中。系统架构分为以下几个层次：

1. **FUSE接口层**：与Linux VFS交互，接收系统调用请求
2. **文件系统逻辑层**：实现文件和目录的创建、删除、读写等核心功能
3. **存储管理层**：管理内存中的文件数据和元数据

存储结构：

- 使用静态数组存储文件条目，最多支持256个文件/目录
- 每个文件条目包含文件名、数据指针、大小、类型等信息
- 采用线性查找方式定位文件（简单高效）

目录结构：

- 实现单层目录结构，所有文件都位于根目录下
- 通过is_directory标志区分文件和目录
- 目录本身不存储数据，仅作为标识存在

内存管理：

- 文件内容动态分配内存，使用realloc根据需要调整大小
- 单个文件最大1MB，防止内存耗尽
- 删除文件时及时释放内存

主要数据结构及其说明

（1）文件条目结构

```
typedef struct {  
    char name[MAX_FILENAME];    // 文件名（包含路径）  
    char *data;                 // 文件数据指针  
    size_t size;                // 文件大小  
    int is_directory;           // 是否为目录  
    time_t atime;               // 访问时间  
    time_t mtime;               // 修改时间  
} file_entry_t;
```

功能说明：

- **name**：存储文件的完整路径，如"/test.txt"
- **data**：指向动态分配的文件内容，目录则为NULL
- **size**：记录文件的实际大小（字节数）
- **is_directory**：区分文件和目录，1表示目录，0表示文件
- **atime**、**mtime**：记录文件的访问和修改时间，用于stat系统调用

（2）全局文件表

```
static file_entry_t files[MAX_FILES]; // 文件表，最多256个文件  
static int file_count = 0;           // 当前文件数量
```

设计考虑：

- 使用静态数组简化内存管理，避免动态分配链表的复杂性
- `file_count`追踪当前文件数，便于快速判断是否已满

(3) FUSE操作结构

```
static struct fuse_operations simplefs_oper = {  
    .getattr    = simplefs_getattr,  
    .readdir    = simplefs_readdir,  
    .create     = simplefs_create,  
    .read       = simplefs_read,  
    .write      = simplefs_write,  
    .unlink     = simplefs_unlink,  
    .mkdir      = simplefs_mkdir,  
    .rmdir      = simplefs_rmdir,  
    .truncate   = simplefs_truncate,  
};
```

接口说明：

- `getattr`：获取文件属性（大小、权限、时间等），对应`stat`系统调用
- `readdir`：读取目录内容，对应`ls`命令
- `create`：创建新文件，对应`touch`、重定向等操作
- `read`：读取文件内容，对应`cat`命令
- `write`：写入文件内容，对应`echo`重定向等操作
- `unlink`：删除文件，对应`rm`命令
- `mkdir`：创建目录，对应`mkdir`命令
- `rmdir`：删除目录，对应`rmdir`命令
- `truncate`：截断文件，对应清空文件等操作

实现文件系统核心功能

(1) 文件查找

实现了一个简单的线性查找函数，根据路径名在文件表中定位文件：遍历整个文件表，使用`strcmp`比较路径名，找到则返回索引，否则返回-1。尽管时间复杂度 $O(n)$ ，但由于文件数量有限（最多256个），性能足够。

```
// 查找文件
static int find_file(const char *path) {
    for (int i = 0; i < file_count; i++) {
        if (strcmp(files[i].name, path) == 0) {
            return i;
        }
    }
    return -1;
}
```

（2）获取文件属性

此函数响应`stat`系统调用，填充文件的元数据信息：对于根目录`"/"`，返回目录属性（权限755）；对于普通文件，返回文件大小、权限644等信息；对于目录，返回目录权限755；填充访问时间和修改时间。这是文件系统最基础的操作之一，任何文件操作（`ls`、`cat`等）都会先调用`getattr`获取文件信息。

```
// 获取文件属性
static int simplefs_getattr(const char *path, struct stat *stbuf) {
    memset(stbuf, 0, sizeof(struct stat));

    // 根目录
    if (strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
        return 0;
    }

    // 查找文件
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (files[idx].is_directory) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    } else {
        stbuf->st_mode = S_IFREG | 0644;
        stbuf->st_nlink = 1;
        stbuf->st_size = files[idx].size;
    }

    stbuf->st_atime = files[idx].atime;
    stbuf->st_mtime = files[idx].mtime;

    return 0;
}
```

（3）读取目录

实现目录列表功能，对应ls命令：首先添加"."和".."两个特殊目录项；遍历文件表，将根目录下的所有文件/目录添加到列表；过滤掉子目录中的文件（通过检查路径中是否有多余的"/"）。使用filler回调函数将每个目录项返回给FUSE框架。

```
// 读取目录
static int simplefs_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                           off_t offset, struct fuse_file_info *fi) {
    (void) offset;
    (void) fi;

    if (strcmp(path, "/") != 0) {
        return -ENOENT;
    }

    filler(buf, ".", NULL, 0);
    filler(buf, "..", NULL, 0);

    // 列出所有文件
    for (int i = 0; i < file_count; i++) {
        // 只显示根目录下的文件（去掉前导/）
        const char *name = files[i].name + 1;
        if (strchr(name, '/') == NULL) { // 不包含子目录
            filler(buf, name, NULL, 0);
        }
    }

    return 0;
}
```

（4）创建文件

实现文件创建功能：检查文件表是否已满（最多256个）；检查文件是否已存在（避免重复创建）；初始化新的file_entry_t结构；设置文件名、初始大小为0、数据指针为NULL；记录创建时间。此时文件仅创建了元数据，实际数据在第一次写入时分配。

```
// 创建文件
static int simplefs_create(const char *path, mode_t mode, struct fuse_file_info *fi) {
    if (file_count >= MAX_FILES) {
        return -ENOSPC;
    }

    if (find_file(path) != -1) {
        return -EEXIST;
    }

    // 创建新文件
    strncpy(files[file_count].name, path, MAX_FILENAME - 1);
    files[file_count].data = NULL;
    files[file_count].size = 0;
    files[file_count].is_directory = 0;
    files[file_count].atime = time(NULL);
    files[file_count].mtime = time(NULL);
    file_count++;

    return 0;
}
```

（5）读取文件

实现文件读取功能，对应cat等命令：根据路径查找文件索引；检查是否为目录（目录不可读）；处理offset和size参数，计算实际读取长度；使用memcpy将文件数据复制到缓冲区；更新访问时间atime；支持部分读取，可以从文件中间位置开始读取指定长度。

```
// 读取文件
static int simplefs_read(const char *path, char *buf, size_t size, off_t offset,
                        struct fuse_file_info *fi) {
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (files[idx].is_directory) {
        return -EISDIR;
    }

    if (offset >= files[idx].size) {
        return 0;
    }

    if (offset + size > files[idx].size) {
        size = files[idx].size - offset;
    }

    if (files[idx].data != NULL) {
        memcpy(buf, files[idx].data + offset, size);
    }

    files[idx].atime = time(NULL);
    return size;
}
```

（6）写入文件

实现文件写入功能，对应echo重定向等操作：查找文件索引；计算写入后的文件大小（offset + size）；检查是否超过单文件大小限制（1MB）；使用realloc调整文件数据内存大小；如果写入位置超过当前文件末尾，中间部分填充0；使用memcpy写入数据；更新修改时间mtime。支持追加写入和覆盖写入，通过offset参数控制写入位置。

```

// 写入文件
static int simplefs_write(const char *path, const char *buf, size_t size,
                          off_t offset, struct fuse_file_info *fi) {
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (files[idx].is_directory) {
        return -EISDIR;
    }

    size_t new_size = offset + size;
    if (new_size > MAX_FILESIZE) {
        return -EFBIG;
    }

    // 重新分配内存
    if (new_size > files[idx].size) {
        char *new_data = realloc(files[idx].data, new_size);
        if (new_data == NULL) {
            return -ENOMEM;
        }

        // 填充空白区域
        if (offset > files[idx].size) {
            memset(new_data + files[idx].size, 0, offset - files[idx].size);
        }

        files[idx].data = new_data;
        files[idx].size = new_size;
    }

    memcpy(files[idx].data + offset, buf, size);
    files[idx].mtime = time(NULL);

    return size;
}

```

(7) 删除文件

实现文件删除功能，对应rm命令：；查找文件索引；检查是否为目录（目录需用rmdir删除）；释放文件数据内存（调用free）；将后面的文件条目前移，覆盖被删除的条目；文件数量减1。通过移动数组元素的方式删除，保持文件表连续性。

```
// 删除文件
static int simplefs_unlink(const char *path) {
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (files[idx].is_directory) {
        return -EISDIR;
    }

    // 释放内存
    if (files[idx].data != NULL) {
        free(files[idx].data);
    }

    // 移动后面的文件
    for (int i = idx; i < file_count - 1; i++) {
        files[i] = files[i + 1];
    }
    file_count--;

    return 0;
}
```

(8) 创建目录

实现目录创建功能，对应mkdir命令：；检查文件表是否已满；检查目录是否已存在；创建新的file_entry_t，设置is_directory为1；目录的data指针为NULL，size为0。目录本身不存储数据，仅作为文件分类的标识。

```
// 创建目录
static int simplefs_mkdir(const char *path, mode_t mode) {
    if (file_count >= MAX_FILES) {
        return -ENOSPC;
    }

    if (find_file(path) != -1) {
        return -EEXIST;
    }

    strncpy(files[file_count].name, path, MAX_FILENAME - 1);
    files[file_count].data = NULL;
    files[file_count].size = 0;
    files[file_count].is_directory = 1;
    files[file_count].atime = time(NULL);
    files[file_count].mtime = time(NULL);
    file_count++;

    return 0;
}
```

(9) 删除目录

实现目录删除功能，对应rmdir命令：查找目录索引；检查是否为普通文件（文件需用unlink删除）；遍历文件表，检查是否有文件属于该目录；如果目录非空，返回ENOTEMPTY错误；如果为空，删除该目录条目。确保只能删除空目录，符合标准文件系统行为。

```

// 删除目录
static int simplefs_rmdir(const char *path) {
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (!files[idx].is_directory) {
        return -ENOTDIR;
    }

    // 检查目录是否为空
    size_t path_len = strlen(path);
    for (int i = 0; i < file_count; i++) {
        if (i != idx && strncmp(files[i].name, path, path_len) == 0) {
            if (files[i].name[path_len] == '/') {
                return -ENOTEMPTY;
            }
        }
    }

    // 移动后面的文件
    for (int i = idx; i < file_count - 1; i++) {
        files[i] = files[i + 1];
    }
    file_count--;

    return 0;
}

```

（10）截断文件

实现文件截断功能，对应清空文件等操作：如果截断大小为0，释放所有文件数据；如果截断到更小，调整内存大小；如果截断到更大，扩展内存并填充0；更新修改时间。此函数在执行"echo > file.txt"等操作时会被调用。

```
// 截断文件
static int simplefs_truncate(const char *path, off_t size) {
    int idx = find_file(path);
    if (idx == -1) {
        return -ENOENT;
    }

    if (files[idx].is_directory) {
        return -EISDIR;
    }

    if (size > MAX_FILESIZE) {
        return -EFBIG;
    }

    if (size == 0) {
        if (files[idx].data != NULL) {
            free(files[idx].data);
            files[idx].data = NULL;
        }
        files[idx].size = 0;
    } else {
        char *new_data = realloc(files[idx].data, size);
        if (new_data == NULL) {
            return -ENOMEM;
        }

        if (size > files[idx].size) {
            memset(new_data + files[idx].size, 0, size - files[idx].size);
        }

        files[idx].data = new_data;
        files[idx].size = size;
    }

    files[idx].mtime = time(NULL);
    return 0;
}
```

编译与测试

在Ubuntu虚拟机中执行 `sudo apt-get install libfuse-dev` 安装FUSE开发库。

进入 `fuse_lab` 文件夹，使用Makefile进行编译，此步编译命令 `gcc -Wall -D_FILE_OFFSET_BITS=64 simplefs.c -o simplefs -lfuse`。

执行 `mkdir -p mnt` 创建挂载点，然后 `./simplefs -f mnt` 启动文件系统。启动后会显示：
SimpleFS - 简单内存文件系统 支持功能: 创建/删除文件、读写文件、创建/删除目录

此时文件系统已成功挂载到mnt目录。

进入mnt目录，执行 `echo "Hello world" > test.txt` 测试文件创建与写入。执行 `cat test.txt` 测试文件读取。执行 `mkdir mydir` 测试目录创建。执行 `ls -la` 测试文件列表。执行 `rm test.txt` 测试文件删除。执行 `rmdir mydir` 测试目录删除。最后再执行 `ls -la` 验证删除状态。测试完成后在文件系统运行的终端按 `Ctrl+C` 停止，可以成功卸载文件系统，挂

载点恢复为普通目录。

```
正在选中未选择的软件包 libselinux1-dev:amd64。
准备解压 .../4-libselinux1-dev_3.0-1build2_amd64.deb ...
正在解压 libselinux1-dev:amd64 (3.0-1build2) ...
正在选中未选择的软件包 libfuse-dev。
准备解压 .../5-libfuse-dev_2.9.9-3_amd64.deb ...
正在解压 libfuse-dev (2.9.9-3) ...
正在设置 libsepol1-dev:amd64 (3.0-1ubuntu0.1) ...
正在设置 libpcre2-16-0:amd64 (10.34-7ubuntu0.1) ...
正在设置 libpcre2-posix2:amd64 (10.34-7ubuntu0.1) ...
正在设置 libpcre2-dev:amd64 (10.34-7ubuntu0.1) ...
正在设置 libselinux1-dev:amd64 (3.0-1build2) ...
正在设置 libfuse-dev (2.9.9-3) ...
正在处理用于 libc-bin (2.31-0ubuntu9.18) 的触发器 ...
正在处理用于 man-db (2.9.1-1) 的触发器 ...
victor@victor-VirtualBox:~/fuse_lab$ make
gcc -Wall -D_FILE_OFFSET_BITS=64 simplefs.c -o simplefs -lfuse
victor@victor-VirtualBox:~/fuse_lab$ make test
创建挂载点...
启动文件系统(使用 Ctrl+C 停止)...
./simplefs -f mnt
SimpleFS - 简单内存文件系统
支持功能: 创建/删除文件、读写文件、创建/删除目录
```

```
victor@victor-VirtualBox:~/fuse_lab$ cd mnt
victor@victor-VirtualBox:~/fuse_lab/mnt$ echo "Hello World" > test.txt
victor@victor-VirtualBox:~/fuse_lab/mnt$ cat test.txt
Hello World
victor@victor-VirtualBox:~/fuse_lab/mnt$ mkdir mydir
victor@victor-VirtualBox:~/fuse_lab/mnt$ ls -la
总用量 4
drwxr-xr-x 2 root root 0 1月 1 1970 .
drwxrwxr-x 3 victor victor 4096 12月 8 20:26 ..
drwxr-xr-x 2 root root 0 12月 8 20:26 mydir
-rw-r--r-- 1 root root 12 12月 8 20:26 test.txt
victor@victor-VirtualBox:~/fuse_lab/mnt$ rm test.txt
victor@victor-VirtualBox:~/fuse_lab/mnt$ rmdir mydir
victor@victor-VirtualBox:~/fuse_lab/mnt$ ls -la
总用量 4
drwxr-xr-x 2 root root 0 1月 1 1970 .
drwxrwxr-x 3 victor victor 4096 12月 8 20:26 ..
victor@victor-VirtualBox:~/fuse_lab/mnt$
```

实验体会

通过本次实验，我深刻体会到了文件系统实现的复杂性与精妙之处。

在实验初期，我曾考虑实现内核态文件系统，以获得更深入的内核编程经验。但经过调研发现，内核态文件系统的开发难度远超预期，不仅需要理解VFS的复杂接口，还要处理各种内核版本差异，调试过程更是困难重重。相比之下，FUSE方案在满足实验要求的同时，大幅降低了开发难度，让我能够将重点放在理解文件系统的核心原理上，而不是陷入内核编程的细节中。这种务实的选择让我在有限的时间内完成了一个功能完整的文件系统，并深入理解了文件系统的工作机制。

在实现文件写入功能时，我遇到了内存管理的难题。最初的设计是为每个文件预分配固定大小的内存，但这样会造成严重的空间浪费。后来改用`realloc`动态调整内存大小，既节省了空间，又支持了文件的动态增长。但使用`realloc`也要特别小心：如果分配失败，必须妥善处理错误；如果文件写入位置超过当前大小，需要将中间的空白区域填充为0。这些细节处理不当就会导致数据损坏或内存泄漏。通过这次实践，我认识到内存管理是系统编程的核心技能，任何疏忽都可能导致严重后果。

在测试过程中，我发现文件系统的很多操作都涉及多个步骤，如何保证这些操作的原子性是一个重要问题。例如，删除文件时需要先释放内存，再移动文件表中的后续条目。如果在这两步之间发生错误，就可能导致文件表状态不一致。虽然SimpleFS是单线程的内存文件系统，不存在并发问题，但这让我意识到真实文件系统需要考虑的复杂性：多进程并发访问、断电保护、日志恢复等。这些高级特性虽然在本实验中没有实现，但通过阅读资料我了解到，现代文件系统（如ext4、btrfs）都有复杂的机制来保证数据一致性。

相比之前的内核实验，用户态文件系统的调试体验要好得多。我可以使用`printf`输出调试信息，可以用`gdb`单步调试，甚至可以用`valgrind`检测内存泄漏。这种便捷的调试环境大大提高了开发效率。当程序崩溃时，我能立即看到详细的错误信息和堆栈跟踪，快速定位问题。这让我深刻体会到，工具链的完善程度对开发效率有着决定性的影响。

尽管SimpleFS功能简单，存在诸多限制（如单层目录、数据不持久化等），但它完整地实现了文件系统的核心功能，让我对操作系统这一重要组件有了更深入的认识。未来如果有机会，我希望能在此基础上实现更多高级特性，如多层目录、数据持久化、权限管理等，进一步提升对文件系统的理解。

参考资料

1. FUSE官方文档: <https://www.kernel.org/doc/html/latest/filesystems/fuse.html>
2. libfuse GitHub仓库: <https://github.com/libfuse/libfuse>
3. Writing a FUSE Filesystem: a Tutorial: <https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/>
4. Linux VFS文档: <https://www.kernel.org/doc/html/latest/filesystems/vfs.html>
5. *Understanding the Linux Kernel, 3rd Edition* - Chapter 12: The Virtual Filesystem