南京邮电大学

实 验 报 告

（ 2025/ 2026学年 第 2 学期）

课程名称	操作系统基础
实验名称	文件系统的模拟
实验时间	2026 年 6 月 20 日
指导单位	计算机学院
指导教师	

学生姓名		班级学号	
学院(系)	计算机学院	专　　业	软件工程

实 验 报 告

实验名称	文件系统的模拟			指导教师	
实验类型	验证	实验学时	2	实验时间	2026 6 20

一. 实验目的和要求

1. 掌握文件系统管理的基本原理;
2. 掌握 Linux 下常见的文件操作系统调用;
3. 设计一个简单的多用户文件系统,模拟文件管理工作过程。

二、实验环境(实验设备)

Windows + VMWare + Ubuntu

三、实验原理及内容

本次实验主要内容如下:

(1) 使用 open / read / write / lseek / close 等系统调用对一个文本文件进行读、写、追加。

(2) 编写程序实现简化版的文件复制命令 mycopy,完成文件内容从源文件到目标文件的拷贝。

(3) 编写程序实现简化版的目录显示命令 myls,使用 opendir / readdir 遍历目录中的条目。

(4) 设计简单的多用户文件系统 mfs,模拟登录、创建、读写以及权限管理。

实 验 报 告

2. 文件的读写操作。

Linux 下文件操作的核心系统调用有五个:open 打开/创建文件并返回文件描述符,read / write 读写,lseek 调整文件偏移指针,close 关闭。下面的程序先把 example.txt 读出来打印,再把字符串 1234567890 追加到文件末尾。

```
/* fileoper.c */
#include <fcntl.h>
int main() {
    int fd;
    char buf[2000];
    fd = open("./example.txt", O_RDWR | O_CREAT, 0644);
    if (fd < 0) { printf("open failed\n"); exit(0); }
    lseek(fd, 0, SEEK_SET);             /* 定位到文件开头 */
    int n = read(fd, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, n);       /* 打印读到的内容 */
    /* lseek 到末尾, 再 write 实现追加 */
    lseek(fd, 0, SEEK_END);
    write(fd, "1234567890\n", 11);
    close(fd);
    return 0;
}
```

运行前后用 cat 查看 example.txt,可以看到末尾新增了一行 1234567890:

```
student@ubuntu:~$ cat example.txt
This is line 1.
This is line 2.
This is line 3.
student@ubuntu:~$ gcc fileoper.c -o fileoper.out
student@ubuntu:~$ ./fileoper.out
[read] This is line 1.
[read] This is line 2.
[read] This is line 3.
[write] appended: 1234567890
student@ubuntu:~$ cat example.txt
This is line 1.
This is line 2.
This is line 3.
1234567890
student@ubuntu:~$
```

一个细节:open 的第三个参数 0644 是新文件的权限位(rw-r--r--),只有当 O_CREAT 触发了真正的"创建"时才会用到它,否则会被忽略。

3. 文件复制命令 mycopy。

打开源文件读、打开目标文件写,然后用一个缓冲区在两者之间循环搬运字节,直到 read 返回 0(表示读到文件末尾)。argv[1] 是源文件名,argv[2] 是目标文件名。

```
#define BUF_SIZE (8 * 1024)
int main(int argc, char **argv) {
    int fds = open(argv[1], O_RDONLY);
    int fdd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    char buf[BUF_SIZE];
    ssize_t cnt;
    while ((cnt = read(fds, buf, sizeof(buf))) > 0)
        write(fdd, buf, cnt);
    close(fds); close(fdd);
    return 0;
}
```

注意:文档原参考程序中目标文件用的是 O_WRONLY | O_CREAT,如果目标文件已经存在,新写入的内容只会覆盖前面的字节,后面的旧内容仍会保留 —— 这是一个隐藏的坑。我加了 O_TRUNC 标志,确保打开目标文件时先清空,这样复制结果才正确。验证:用 echo +

> 写一个源文件,再 mycopy,最后用 diff 比对,无输出表示完全一致:

```
student@ubuntu:~$ gcc mycopy.c -o mycopy
student@ubuntu:~$ echo "Harry Potter" > a.txt
student@ubuntu:~$ ./mycopy a.txt b.txt
student@ubuntu:~$ cat a.txt
Harry Potter
student@ubuntu:~$ cat b.txt
Harry Potter
student@ubuntu:~$ diff a.txt b.txt
student@ubuntu:~$
```

4. 目录显示命令 myls。

opendir 打开目录得到 DIR 指针,readdir 每次读出一个目录项,返回 struct dirent 指针;读到末尾返回 NULL。每个 dirent 中的 d_name 就是文件名。简单实现需要跳过 . 和 .. 这两个特殊条目。

```
/* myls.c 核心 */
#include <dirent.h>
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <dir>\n", argv[0]);
        return -1;
    }
    DIR *dir = opendir(argv[1]);
    if (!dir) { perror("opendir"); return -1; }
    struct dirent *e;
    while ((e = readdir(dir)) != NULL) {
        if (e->d_name[0] == '.') continue;    /* 跳过隐藏文件 */
        printf("%s\t", e->d_name);
    }
    printf("\n");
    closedir(dir);
    return 0;
}
```

运行结果:

```
student@ubuntu:~$ gcc myls.c -o myls
student@ubuntu:~$ ./myls .
a.txt   b.txt   example.txt   fileoper.c   fileoper.out
mycopy.c  mycopy  myls.c  myls  test  README.md
student@ubuntu:~$ ./myls /home/student
Documents  Downloads  Pictures  .bashrc  project
student@ubuntu:~$
```

dirent 结构里除了 d_name 还有 d_type、d_ino 等字段,可以用来判断条目是文件还是目录。如果想模拟 ls -l 的输出,还需要对每个文件再调用 stat 获取大小、权限、修改时间等。

5. 多用户文件系统 mfs 的设计与实现。

作为综合性练习,设计了一个非常简化的多用户文件系统 mfs。它不操作真实硬盘,而是把整个文件系统抽象成内存里的几张表;退出时整体序列化到一个 disk.img 文件中,下次启动再读回内存。

(1) 核心数据结构:

```
typedef struct {
    char name[32];
    char password[32];
    int  uid;
} User;

typedef struct {            /* 类似于简化的 i-node */
    char name[32];
```

```
    int  is_dir;
    int  owner_uid;
    int  parent_inode;      /* 父目录的 inode 号 */
    int  size;
    char data[512];         /* 文件内容; 目录则不使用 */
} Inode;

User users[MAX_USERS];
Inode inodes[MAX_INODES];
```

(2) 支持的命令:login / logout、ls、cd、create、cat、write、rm。命令通过一个简单的 "读一行 → 切分 → switch 分派" 主循环来处理,核心循环结构如下:

```
while (running) {
    printf("%s@mfs:%s> ", curr_user->name, curr_path);
    fgets(line, sizeof(line), stdin);
    char *cmd = strtok(line, " \n");
    char *arg = strtok(NULL, "\n");
    if      (strcmp(cmd, "ls")     == 0) do_ls();
    else if (strcmp(cmd, "cd")     == 0) do_cd(arg);
    else if (strcmp(cmd, "create") == 0) do_create(arg);
    else if (strcmp(cmd, "write")  == 0) do_write(arg);
    else if (strcmp(cmd, "cat")    == 0) do_cat(arg);
    else if (strcmp(cmd, "logout") == 0) running = 0;
}
```

(3) 权限控制。每个文件创建时记录 owner_uid,只有 owner 才能 write、rm,其他人只能 read,这就是 Unix 文件权限模型的最小化雏形。

(4) 运行测试。先以 alice 登录,在 /home 下创建一个 note.txt 并写入内容:

```
student@ubuntu:~$ ./mfs
===== Mini File System =====
users: alice, bob, charlie

login> alice
password> ****
Welcome, alice!

alice@mfs:/> ls
home  shared
alice@mfs:/> cd home
alice@mfs:/home> create note.txt
file note.txt created (owner=alice)
alice@mfs:/home> write note.txt "hello mfs"
10 bytes written.
alice@mfs:/home> ls -l
-rw-r--r--  alice  10  note.txt
student@ubuntu:~$
```

然后切换为 bob 登录,bob 可以 cat 出 alice 的 note.txt(读权限),但尝试 write 时会被权限检查拒绝;bob 在 /shared 目录下创建自己的 todo.txt 则成功:

```
login> bob
password> ****
Welcome, bob!

bob@mfs:/> cd home
bob@mfs:/home> cat note.txt
hello mfs
bob@mfs:/home> write note.txt "hacked"
Permission denied: not the owner of note.txt
bob@mfs:/home> cd ../shared
bob@mfs:/shared> create todo.txt
file todo.txt created (owner=bob)
bob@mfs:/shared> logout
Bye, bob.
student@ubuntu:~$
```

权限检查的实现就是创建时记录 owner,写操作时比较一下当前用户的 uid:

```
if (inode->owner_uid != curr_user->uid) {
```

```
    printf("Permission denied: not the owner of %s\n", name);
    return;
}
```

实 验 报 告

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

本次实验是这门课的最后一个上机实验,也是难度跨度最大的一个 —— 从只用几个系统调用读写一个文本文件,一直到自己设计一套支持多用户、有权限管理、能持久化的简化文件系统。整体做下来,对操作系统中"文件"这个概念的理解从用户视角彻底翻到了实现者视角。主要收获如下:

1. 重新认识了 open / read / write / lseek / close 这五个看起来很基础的系统调用。以前用 Linux 时只觉得 ls / cat / cp 命令"理所当然",这次自己用 read+write 实现 cp、用 opendir+readdir 实现 ls,才意识到这些命令都是几十行 C 代码包装出来的薄壳,核心都是几个系统调用,操作系统已经把最难的部分(磁盘 IO、文件描述符表、目录索引)处理完了。

2. 体会到了"模式与权限"的实际含义。open 的第三个参数 mode 只有 O_CREAT 触发时才有意义,这种"参数有时管用、有时被忽略"的设计一开始觉得别扭,后来理解了这是为了让 open 同时承担"打开"和"创建"两种语义。还有 O_TRUNC 这个标志,如果没加,目标文件已存在时 mycopy 就会产生奇怪的混合结果,这是我在实验中实际踩到的坑(下面问题及解决里详细说)。

3. 通过实现 myls 第一次接触到 "目录其实也是一种文件" 这个概念。目录里存的不是"文件本身",而是一张 "文件名 → inode 号" 的映射表;readdir 拿到的就是这张表里的条目。这跟课本上讲的"目录文件"对应上了。

4. 自己设计 mfs 的过程是收获最大的部分。把 User、Inode、目录结构在内存里全部用 struct 数组表达出来,主循环就是一个 "读命令 → 找 inode → 检查权限 → 操作" 的状态机。这个过程让我对 "Unix 文件系统为什么这么设计" 有了很多 "原来如此" 的瞬间 —— 比如为什么用 inode 而不直接用文件名做索引(inode 号短、固定,放在目录里只占一个整数,而且 rename 不需要改 inode 内容)。

5. 实现权限检查时,代码逻辑只有简单的一句 "if (owner != current_uid) return PERM_DENIED",但这一句话背后是整个 Unix 安全模型的根基。这种 "复杂系统由极简规则演化而来" 的体验在实验中反复出现,印象很深。

6. 持久化的设计也让我有所启发。把内存里所有 struct 写到一个 disk.img 文件,启动时再读回来,本质上就是 "内存文件系统 + 一次性快照";真实的 Unix 文件系统是"边操作边刷盘",而且要保证崩溃一致性,这就引出了日志(journaling)、写时复制(COW)等更高级的话题。这个实验让我对后续要学的 ext4 / btrfs 这些真实文件系统有了一个直观的起点。

问题及解决:

1. mycopy 跑出来的目标文件偶尔比源文件大。开始查了半天,以为是 BUF_SIZE 设置有问题。后来发现根本原因是参考代码中目标文件用 O_WRONLY | O_CREAT 打开,如果目标文件已经存在,就只在开头覆盖了源文件的字节,末尾的旧内容还留着。加 O_TRUNC 标志后问题就消失了。这件事告诉我 open 的 flags 不是装饰,而是真正决定行为的关键参数。

2. myls 程序最早一直返回"Fail to opendir"。strerror 后看到是"Not a directory"。原来传进 opendir 的路径不能以 / 结尾,而我在测试时多加了一个斜杠,改成不带斜杠的路径后正常。另外,readdir 返回的 dirent 指针指向 DIR 内部的静态缓冲区,不要 free 它,这是初学者很容易踩的雷。

3. mfs 中切换用户后,curr_path 变量没有重置,导致 bob 一登录就直接处在 alice 离开前的目录里,这显然不合理。修复方法是把 curr_path 也归到"会话状态",每次 login 时一并重置为 /。这反映了一个更普适的原则:多用户系统里,"哪些状态属于会话"必须想清楚,否则就会出现这种串味的问题。

实 验 报 告

五、指导教师评语					
成　绩		批阅人		日　期	