

# lab5 文件系统

---

191220138 杨飞洋

## 实验进度

---

已完成所有要求部分。

## 实验目的

---

通过代码模拟了解linux系统中文件系统的管理方法，并且实现几个系统调用，最后实现ls、cat命令。

## 数据结构和一些宏

---

文件描述符

```
struct File {  
    int state;  
    int inodeoffset; //xxx inodeoffset in filesystem, for syscall open  
    int offset; //xxx offset from SEEK_SET  
    int flags;  
};  
typedef struct File File;
```

flags权限

```
#define O_WRITE 0x01  
#define O_READ 0x02  
#define O_CREATE 0x04  
#define O_DIRECTORY 0x08
```

whence详解

```
#define SEEK_SET 0  
#define SEEK_CUR 1  
#define SEEK_END 2
```

inode

```

union Inode {
    uint8_t byte[INODE_SIZE];
    struct {
        int16_t type; // further implement privilege control, i.e., drwxrwxrwx,
        uid, gid, others
        int16_t linkCount;
        int32_t blockCount;
        int32_t size; // size of this file, byte as unit
        int32_t pointer[POINTER_NUM];
        int32_t singlyPointer;
        int32_t doublyPointer;
        int32_t triplyPointer;
    };
};

```

```

#define SUPER_BLOCK_SIZE 1024
#define GROUP_DESC_SIZE 32
#define INODE_BITMAP_SIZE BLOCK_SIZE
#define BLOCK_BITMAP_SIZE BLOCK_SIZE
#define INODE_SIZE 128
#define DIRENTRY_SIZE 128

#define UNKNOWN_TYPE 0
#define REGULAR_TYPE 1
#define DIRECTORY_TYPE 2
#define CHARACTER_TYPE 3
#define BLOCK_TYPE 4
#define FIFO_TYPE 5
#define SOCKET_TYPE 6
#define SYMBOLIC_TYPE 7

```

## 实验过程

### open

如果文件存在即 `ret == 0`

首先查看 `flag` 和文件类型是否匹配，如果不匹配直接返回-1

```

if ((sf->edx != 0x08 && destInode.type == DIRECTORY_TYPE) ||
    (sf->edx == 0x08 && destInode.type == REGULAR_TYPE)) {
    pcb[current].regs.eax = -1;
    return;
}

```

如果匹配，再查看一下是否是正在使用中的文件，即 `state == 1`，如果是，返回-1

如果是设备文件，直接返回其id

```

for (int i = 0; i < MAX_DEV_NUM; i++) {
    if (dev[i].inodeOffset == destInodeOffset && dev[i].state == 1) {
        pcb[current].regs.eax = i;
        return;
    }
}
for (int i = 0; i < MAX_FILE_NUM; i++) {
    if (file[i].inodeOffset == destInodeOffset && file[i].state == 1) {
        pcb[current].regs.eax = -1;
        return;
    }
}

```

如果不在使用中，为其分配一个文件描述符，配置以下属性，返回 `MAX_DEV_NUM + 文件描述符号`

```

for (int i = 0; i < MAX_FILE_NUM; i++) {
    if (file[i].state == 0) {
        file[i].state = 1;
        file[i].inodeOffset = destInodeOffset;
        file[i].offset = 0;
        file[i].flags = sf->edx;
        pcb[current].regs.eax = MAX_DEV_NUM + i;
        return;
    }
}

```

如果文件不存在，即 `ret == -1`

首先检查以下 `O_CREATE`，如果不允许被创建，返回-1.

```

if ((sf->edx >> 2) % 2 == 0) {
    //TODO: if O_CREATE not set
    pcb[current].regs.eax = -1;
    return;
}

```

如果是创建一个普通文件，首先利用 `stringChrR()` 函数可以查看路径中是否有 '/'，如果没有返回-1，否则这个函数返回最后一个 '/' 的元素下标，可以提取文件名和父目录。如果父目录不存在返回-1，否则利用 `allocInode()` 函数创建一个新的 `REGULAR_TYPE` 的文件。

```

if ((sf->edx >> 3) % 2 == 0) {
    //TODO: if O_DIRECTORY not set
    ret = stringChrR(str, '/', &size);
    if (ret == -1) { // no '/' in file path
        pcb[current].regs.eax = -1;
        return;
    }
    tmp = *(str + size + 1); //filename
    *(str + size + 1) = 0; //father directory
    ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, str);
}

```

```

*(str + size + 1) = tmp;
if (ret == -1) {
    pcb[current].regs.eax = -1;
    return;
}
ret = allocInode(&sBlock, gDesc,
                &fatherInode, fatherInodeOffset,
                &destInode, &destInodeOffset, str + size + 1,
REGULAR_TYPE);
}

```

如果创建一个目录，先处理一个 `str` 的最后一个字符，如果是 `'/'`，将其改成结束符。

还是和之前的一样利用 `stringChrR()` 函数可以查看路径中是否有 `'/'`，如果没有返回-1，否则这个函数返回最后一个 `'/'` 的元素下标，可以提取新目录名和父目录。这次用一个 `parent_path` 来存储父路径，如果不存在返回-1，否则利用 `allocInode()` 函数创建一个新的 `DIRECTORY_TYPE` 的目录。

```

//TODO: if O_DIRECTORY set
length = strlen(str);
if (str[length - 1] == '/')
{
    cond = 1;
    str[length - 1] = 0;
}
ret = stringChrR(str, '/', &size);
if (ret == -1)
{
    pcb[current].regs.eax = -1;
    return;
}
char parent_path[128];
for (int i = 0; i < size + 1; ++i)
    parent_path[i] = *(str + i);
parent_path[size + 1] = 0;
ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, parent_path);
if (ret == -1)
{
    if (cond == 1)
        str[length - 1] = '/';
    pcb[current].regs.eax = -1;
    return;
}
ret = allocInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
&destInodeOffset, str + size + 1, DIRECTORY_TYPE);
if (cond == 1)
    str[length - 1] = '/';

```

最后如果还是 `ret == -1`，返回-1.

否则在全局分配一个文件描述符，如果找不到 `state == 0` 的，返回-1.

```

if (ret == -1) {
    pcb[current].regs.eax = -1;
}

```

```

    return;
}
for (int i = 0; i < MAX_FILE_NUM; i++) {
    if (file[i].state == 0) { // not in use
        file[i].state = 1;
        file[i].inodeOffset = destInodeOffset;
        file[i].offset = 0;
        file[i].flags = sf->edx;
        pcb[current].regs.eax = MAX_DEV_NUM + i;
        return;
    }
}
pcb[current].regs.eax = -1; // create success but no available file[]
return;

```

## write

首先在syscall.c中模仿open进行系统调用，将 fd 传入 ecx，buffer 传入 edx，size 传入 ebx

```

int write (int fd, uint8_t *buffer, int size) {
    //TODO: Complete the function 'write' just like the function 'open'.
    return syscall(SYS_WRITE, fd, (uint32_t)buffer, size, 0, 0);
}

```

在 void syscallwrite(struct StackFrame \*sf) 函数中，如果 fd 即 ecx 指向一个已经open的普通文件，则调用 syscallwriteFile 读它，否则返回-1.

```

// TODO: if refer to a file
if (sf->ecx >= MAX_DEV_NUM && sf->ecx < MAX_DEV_NUM + MAX_FILE_NUM) {
    if (file[sf->ecx - MAX_DEV_NUM].state == 1)
        syscallwriteFile(sf);
    else
        pcb[current].regs.eax = -1;
}

```

来看 syscallwriteFile() 函数。

如果传入的 size <= 0，则不需要读，返回0.

遍历传入的 str，将数据传递给 buffer，buffer 一块块地将数据放进文件，如果 (remainder + i) % sBlock.blockSize == 0 即读完一块，如果块数满了，则调用 allocBlock() 函数来分配一些块，如果分配失败，那只能调用 diskwrite() 往磁盘中写入现有的内容了。如果块数没有满，就调用 writeBlock 正常写入块就行了。

部分代码如下：

```

if (size <= 0) {
    pcb[current].regs.eax = 0;
    return;
}

```

```

if (quotient + j < inode.blockCount)
    readBlock(&sBlock, &inode, quotient + j, buffer);
while(i < size) {
    buffer[(remainder + i) % sBlock.blockSize] = str[i];
    i++;
    if ((remainder + i) % sBlock.blockSize == 0) {
        if (quotient + j == inode.blockCount) {
            ret = allocBlock(&sBlock, gDesc, &inode, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
            if (ret == -1) {
                inode.size = inode.blockCount * sBlock.blockSize;
                diskwrite(&inode, sizeof(Inode), 1, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
                pcb[current].regs.eax = inode.size - file[sf->ecx -
MAX_DEV_NUM].offset;
                file[sf->ecx - MAX_DEV_NUM].offset = inode.size;
                return;
            }
        }

        writeBlock(&sBlock, &inode, quotient + j, buffer);
        j++;
        if (quotient + j < inode.blockCount)
            readBlock(&sBlock, &inode, quotient + j, buffer);
    }
}
if (quotient + j == inode.blockCount) {
    ret = allocBlock(&sBlock, gDesc, &inode, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
    if (ret == -1) {
        inode.size = inode.blockCount * sBlock.blockSize;
        diskwrite(&inode, sizeof(Inode), 1, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
        pcb[current].regs.eax = inode.size - file[sf->ecx - MAX_DEV_NUM].offset;
        file[sf->ecx - MAX_DEV_NUM].offset = inode.size;
        return;
    }
}
}

```

## read

首先在syscall.c和在 `void syscallRead(struct StackFrame *sf)` 函数中, 进行的操作和write类似, 在此不多赘述。

还是先判断size, 如果 `size <= 0`, 则没有读的必要, 返回0.

如果size超出了范围, 将其限定为最大即 `inode.size - offset`

利用 `readBlock` 函数将文件里的数据一块块地发送给buffer, 在将buffer的数据传递给str即 `edx`, 也就是用户的缓冲区。

代码如下:

```

int idx = sf->ecx - MAX_DEV_NUM;
if (size <= 0) {

```

```

    pcb[current].regs.eax = 0;
    return;
}
if (size > inode.size - file[idx].offset)
    size = inode.size - file[idx].offset;
readBlock(&sBlock, &inode, quotient + j, buffer);
j++;
while(i < size) {
    str[i] = buffer[(remainder + i) % sBlock.blockSize];
    i++;
    if ((remainder + i) % sBlock.blockSize == 0) {
        readBlock(&sBlock, &inode, quotient + j, buffer);
        j++;
    }
}
}

```

## seek

首先在syscall.c中，进行的操作和 open 类似，在此不多赘述。

如果传入的 whence == SEEK\_SET，即文件头，那需要判断 offset 是否合法，不能为负也不能超出 inode.size，合法则调整 offset，返回0，否则返回-1

```

case SEEK_SET
    // TODO: if SEEK_SET
    if (offset >= 0 && offset <= inode.size) {
        file[idx].offset = offset;
        pcb[current].regs.eax = 0;
    }
    else
        pcb[current].regs.eax = -1;
    break;

```

如果传入的 whence == SEEK\_CUR，即现在的文件 offset，判断加上新的偏移量之后是否正常就行了。

```

case SEEK_CUR:
    // TODO: if SEEK_CUR
    if(file[idx].offset + offset >= 0 && file[idx].offset + offset <=
inode.size){
        file[idx].offset = offset;
        pcb[current].regs.eax = 0;
    }
    else
        pcb[current].regs.eax = -1;
    break;

```

如果传入的 whence == SEEK\_END，即文件尾，判断加上新的偏移量之后是否正常就行了。

```

case SEEK_END:
    // TODO: if SEEK_END
    if(inode.size + offset >= 0 && inode.size + offset <= inode.size){
        file[idx].offset = inode.size + offset;
        pcb[current].regs.eax = 0;
    }
    else
        pcb[current].regs.eax = -1;
    break;

```

## close

首先在syscall.c中，进行的操作和 open 类似，在此不多赘述，只是传入的是文件号 fd

如果文件号不合法或者这个文件没有open即 state == 0，关闭失败，返回-1

```

if (i < MAX_DEV_NUM || i >= MAX_DEV_NUM + MAX_FILE_NUM) {
    // TODO: dev, can not be closed, or out of range
    pcb[current].regs.eax = -1;
    return;
}
if (file[i - MAX_DEV_NUM].state == 0) {
    // TODO: not in use
    pcb[current].regs.eax = -1;
    return;
}

```

否则，可以close文件。

将文件的描述符归零，返回0

```

int idx = i - MAX_DEV_NUM;
file[idx].state = 0;
file[idx].inodeOffset = 0;
file[idx].offset = 0;
file[idx].flags = 0;
pcb[current].regs.eax = 0;

```

## remove

首先在syscall.c中，进行的操作和 open 类似，在此不多赘述。

文件肯定得要存在即 ret == 0。

在此基础上，先查看是否是设备文件，如果是返回-1



```

for (i = 0; i < MAX_DEV_NUM; i++) {
    if (dev[i].inodeOffset == destInodeOffset) {
        pcb[current].regs.eax = -1;
        return;
    }
}

```

如果指向一个正在使用的文件，将文件的描述符归零

```

for (i = 0; i < MAX_FILE_NUM; i++) {
    if (file[i].inodeOffset == destInodeOffset && file[i].state == 1) {
        file[i].state = 0;
        file[i].inodeOffset = 0;
        file[i].offset = 0;
        file[i].flags = 0;
    }
}

```

否则如果删去的是普通文件，首先利用 `stringChrR()` 函数可以查看路径中是否有 '/'，如果没有返回 -1，否则这个函数返回最后一个 '/' 的元素下标，可以提取文件名和父目录。如果父目录不存在返回 -1，否则利用 `freeInode()` 函数进行删除。

```

if (destInode.type == REGULAR_TYPE) {
    // TODO: If REGULAR_TYPE
    ret = stringChrR(str, '/', &size);
    if (ret == -1) { // no '/' in file path
        pcb[current].regs.eax = -1;
        return;
    }
    tmp = *(str + size + 1);
    *(str + size + 1) = 0;
    ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, str);
    *(str + size + 1) = tmp;
    if (ret == -1) {
        pcb[current].regs.eax = -1;
        return;
    }
    ret = freeInode(&sBlock, gDesc,
        &fatherInode, fatherInodeOffset,
        &destInode, &destInodeOffset, str + size + 1, REGULAR_TYPE);
}

```

如果删去的是目录，先处理一个 `str` 的最后一个字符，如果是 '/'，将其改成结束符。

还是和之前的一样利用 `stringChrR()` 函数可以查看路径中是否有 '/'，如果没有返回 -1，否则这个函数返回最后一个 '/' 的元素下标，可以提取目录名和父目录。这次用一个 `parent_path` 来存储父路径，如果不存在返回 -1，否则利用 `freeInode()` 函数进行删除。

```

else if (destInode.type == DIRECTORY_TYPE) {
    // TODO: If DIRECTORY_TYPE

```

```

length = strlen(str);
if (str[length - 1] == '/')
{
    cond = 1;
    str[length - 1] = 0;
}
if (stringChrR(str, '/', &size) == -1)
{
    pcb[current].regs.eax = -1;
    return;
}
char parent_path[128];
for (int i = 0; i < size + 1; ++i)
    parent_path[i] = *(str + i);
parent_path[size + 1] = 0;
ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset,
parent_path);
if (ret == -1)
{
    if (cond == 1)
        str[length - 1] = '/';
    pcb[current].regs.eax = -1;
    return;
}
ret = freeInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
&destInodeOffset, str + size + 1, DIRECTORY_TYPE);
if (cond == 1)
    str[length - 1] = '/';
}
if (ret == -1) {
    pcb[current].regs.eax = -1;
    return;
}
pcb[current].regs.eax = 0;

```

## ls

可以利用 `DirEntry` 结构来访问目录，遍历目录中的 `inode`，只要不是0，就打印文件名。

```

while (ret != 0) {
    // TODO: Complete 'ls'.
    dirEntry = (DirEntry *)buffer;
    for (i = 0; i < 8; ++i)
    {
        //printf("%d\n", i);
        if (dirEntry[i].inode != 0)
            printf("%s ", dirEntry[i].name);
    }
    ret = read(fd, buffer, 512 * 2);
}

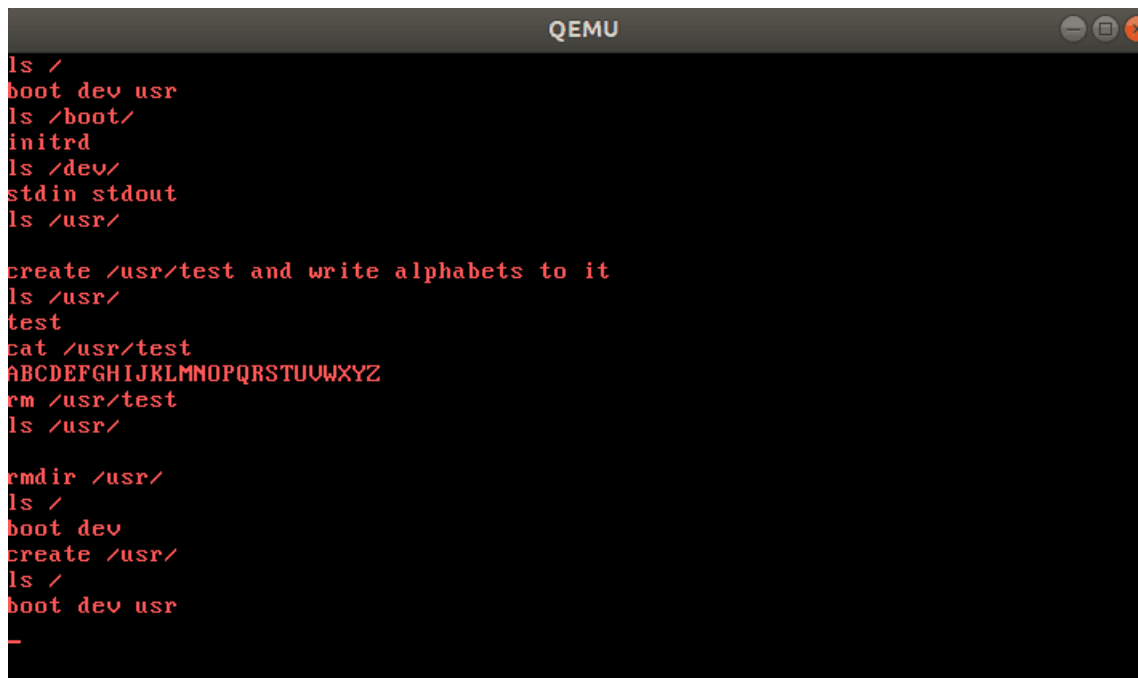
```

## cat

cat命令用来输出文件内容，读取文件内容到缓冲区，将 `STD_OUT` 传入 `write` 函数进行打印

```
while (ret != 0) {  
    // TODO: Complete 'cat'  
    write(STD_OUT, buffer, ret);  
    ret = read(fd, buffer, 512 * 2);  
}
```

## 实验结果



```
QEMU  
ls /  
boot dev usr  
ls /boot/  
initrd  
ls /dev/  
stdin stdout  
ls /usr/  
  
create /usr/test and write alphabets to it  
ls /usr/  
test  
cat /usr/test  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
rm /usr/test  
ls /usr/  
  
rmdir /usr/  
ls /  
boot dev  
create /usr/  
ls /  
boot dev usr  
-
```

## 实验心得

通过代码模拟更加了解了操作系统中读写文件的过程，`inode` 实在是非常重要的。

框架代码的很多接口函数在说明文档中没有提及，导致调用的时候非常困难和耗时间，希望在之后的实验中可以说的更加详细一点。