

《操作系统》lab 5实验报告

姓名	学号	邮箱	日期
薛飞阳	181860121	502126785@qq.com	2020/6/22

1. 实验进度

我完成的内容有：

- 完善了格式化程序，添加了rm和rmdir函数
- 内核支持文件读写，包括open,read,write,lseek,close,remove的调用
- 实现了ls和cat这两个用户函数

2. 实验结果及测试所用代码分析

2.1. 测试格式化程序中的rm和rmdir

```
//create /usr/testdir/  
stringCpy("/usr/testdir", destFilePath, NAME_LENGTH - 1);  
mkdir(driver, destFilePath);  
//put a file in /testdir  
stringCpy("/usr/testdir/test", destFilePath, NAME_LENGTH - 1);  
touch(driver, destFilePath);  
//put a file in /usr  
stringCpy("/usr/test2", destFilePath, NAME_LENGTH - 1);  
touch(driver, destFilePath);  
//ls /usr  
stringCpy("/usr", destFilePath, NAME_LENGTH - 1);  
ls(driver, destFilePath);  
//delete file  
stringCpy("/usr/test2", destFilePath, NAME_LENGTH - 1);  
rm(driver, destFilePath, 0);  
//ls /usr  
stringCpy("/usr", destFilePath, NAME_LENGTH - 1);  
ls(driver, destFilePath);  
//delete dir  
stringCpy("/usr/testdir", destFilePath, NAME_LENGTH - 1);  
rmdir(driver, destFilePath);  
//ls /usr  
stringCpy("/usr", destFilePath, NAME_LENGTH - 1);  
ls(driver, destFilePath);
```

测试代码如上所示，位于 `/utils/genFS/main.c`，该部分依次做了如下操作：

- 创建/usr下的/testdir目录
- /testdir目录内创建文件/test
- /usr内创建文件/test2
- ls /usr，删除test2后再次ls /usr

- ls /usr, 删除/testdir后再次ls /usr

测试结果如下两图所示, 可以看到, 不仅ls正确显示, 而且在删除空的test2文件后, 如图1, block释放0个, inode释放1个; 在删除包含一个空文件的文件夹后, 如图2, block释放1个, inode释放2个。

```
ls /usr
Name: ., Inode: 4, Type: 2, LinkCount: 3, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: print, Inode: 5, Type: 1, LinkCount: 1, BlockCount: 19, Size: 18460.
Name: testdir, Inode: 6, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: test2, Inode: 8, Type: 1, LinkCount: 1, BlockCount: 0, Size: 0.
LS success.
1016 inodes and 3916 data blocks available.
rm /usr/test2
RM success.
ls /usr
Name: ., Inode: 4, Type: 2, LinkCount: 3, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: print, Inode: 5, Type: 1, LinkCount: 1, BlockCount: 19, Size: 18460.
Name: testdir, Inode: 6, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
LS success.
1017 inodes and 3916 data blocks available.
```

```
rm test in dir /usr/testdir.
RMDIR success.
ls /usr
Name: ., Inode: 4, Type: 2, LinkCount: 3, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: print, Inode: 5, Type: 1, LinkCount: 1, BlockCount: 19, Size: 18460.
LS success.
1019 inodes and 3917 data blocks available.
```

2.2. 测试用户程序

```
#include "lib.h"
#include "types.h"

int uEntry(void)
{
    /**
    //main test
    int fd = 0;
    int i = 0;
    char tmp = 0;
    //test ls
    ls("/");
    ls("/boot/");
    ls("/usr/");
    printf("create /usr/test and write alphabets to it\n");
    //test open
    fd = open("/usr/test", O_READ | O_WRITE | O_CREATE);
    ls("/usr/");
    //test write, cat and close
    for (i = 0; i < 26; i++)
    {
        tmp = (char)(i + 'A');
        write(fd, (uint8_t *)&tmp, 1);
    }
    close(fd);
    cat("/usr/test");
    //test lseek
```

```

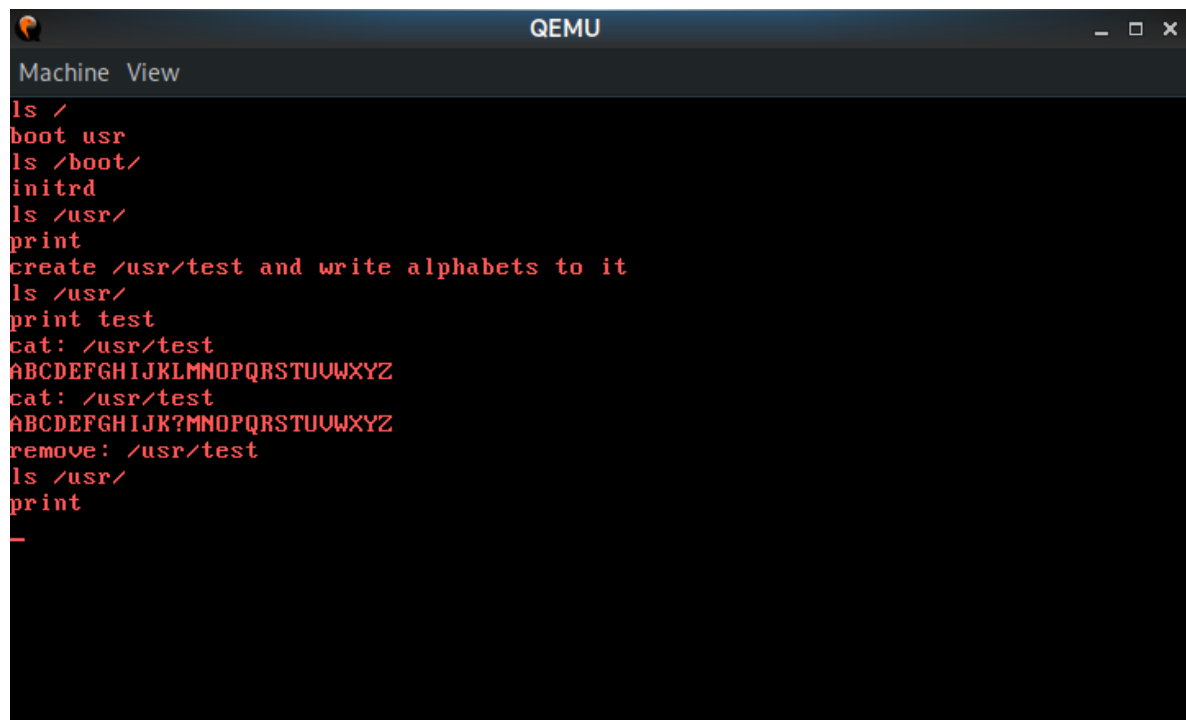
    fd = open("/usr/test", O_READ | O_WRITE);
    lseek(fd, 11, SEEK_SET);
    char C = '?';
    write(fd, (uint8_t *)&C, 1);
    close(fd);
    cat("/usr/test");
    //test remove
    remove("/usr/test");
    ls("/usr/");
    exit();
    return 0;
}

```

测试代码如下所示，位于 `/app/main.c`，该部分依次做了如下操作：

- ls数个目录并查看
- open一个不存在的文件，将创建新文件
- ls查看是否创建了这个文件
- 在文件中用write写入26个字母，关闭后用cat查看
- 再次打开文件（不带有参数O_CREATE），lseek定位到11
- 覆盖写入一个'?'，关闭后用cat查看
- 删除文件，并ls确认

测试结果如下图所示：



```

QEMU
Machine View
ls /
boot usr
ls /boot/
initrd
ls /usr/
print
create /usr/test and write alphabets to it
ls /usr/
print test
cat: /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
cat: /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
remove: /usr/test
ls /usr/
print
-

```

3. 实验修改的代码

3.1. 实现格式化程序中的rm和rmdir函数

修改位置：`lab5/utis/genFS/func.c`和`func.h`

在 `fun.c.c` 中，完成 `rmdir()` 和 `rm()` 函数，为函数增加格式化输入功能。注意到 `rmdir()` 的本质可认为是删除了文件夹内的所有文件，然后删除文件夹本身，即数次 `rm()` 的调用，故主要考虑 `rm()` 的实现。

3.1.1 rm()函数

为了调试需要，我们在 `rm()` 中引入新的参数 `state`，该参数仅用于指明是否为 `rmdir()` 调用，并输出不同的信息。

删除一个文件主要要做以下操作：

- 找到当前inode和父节点的inode
- 根据当前inode及其位置确定需要修改的inodeBitmap和BlockBitmap位置，并将其置为0，同时增加superBlock中的剩余inode和block数
- 根据父节点的inode定位其DirEntry入口，找到当前inode对应的表项，将其inode置为0

我们将需要修改信息的后两者抽离出来，写在函数 `myremove()` 中，以方便阅读。而寻找inode的实现完全可以参考 `mkdir()` 和 `cp()` 中关于super、inode、fatherinode及其offset的寻找，在此不再赘述。

`myremove()` 函数传入如下参数：

```
int myremove(FILE *file, Inode *inode, Inode *fatherinode, SuperBlock
*superBlock, int fatherInodeOffset, int inodeOffset)
```

该函数首先释放inode信息：

```
//free inode
int inodeTableOffset = superBlock->inodeTable;
int inodeBitmapOffset = superBlock->inodeBitmap;
InodeBitmap inodeBitmap;
fseek(file, inodeBitmapOffset * SECTOR_SIZE, SEEK_SET);
fread((void *)&inodeBitmap, sizeof(InodeBitmap), 1, file);
int i = (inodeOffset - inodeTableOffset * SECTOR_SIZE) / sizeof(Inode);
int j = i / 8;
int k = i % 8;
inodeBitmap.byte[j] ^= (1 << (7 - k));
superBlock->availInodeNum++;
fseek(file, inodeBitmapOffset * SECTOR_SIZE, SEEK_SET);
fwrite((void *)&inodeBitmap, sizeof(InodeBitmap), 1, file);
```

如上，先读取inodeBitmap的入口，再根据偏移量计算出是位图中的哪一个inode，将该位置为0，同时将可用inode数++。由于inodeBitmap的每一个byte有8位，故我们先计算出i（即第几个inode）后，i/8即为第几个byte，i%8为该byte的第几位。

接下来释放block信息：

```
//free blocks
int blockBitmapOffset = superBlock->blockBitmap;
BlockBitmap blockBitmap;
fseek(file, blockBitmapOffset * SECTOR_SIZE, SEEK_SET);
fread((void *)&blockBitmap, sizeof(BlockBitmap), 1, file);
int divider0 = superBlock->blockSize / 4;
int bound0 = POINTER_NUM;
```

```

int bound1 = bound0 + divider0;
uint32_t singlyPointerBuffer[divider0];
for (int index = 0; index < inode->blockCount; index++)
{
    int insideblockOffset;
    if (index < bound0) //level 1
        insideblockOffset = (inode->pointer[index] - superBlock->blocks) *
SECTOR_SIZE;
    else if (index < bound1) //level 2
    {
        fseek(file, inode->singlyPointer * SECTOR_SIZE, SEEK_SET);
        fread((void *)singlyPointerBuffer, sizeof(uint8_t), superBlock-
>blockSize, file);
        insideblockOffset = (singlyPointerBuffer[index - bound0] - superBlock-
>blocks) * SECTOR_SIZE;
    }
    else
        return -1;
    int i = (insideblockOffset) / superBlock->blockSize;
    int j = i / 8;
    int k = i % 8;
    blockBitmap.byte[j] ^= (1 << (7 - k));
}
superBlock->availBlockNum += inode->blockCount;

```

与释放inode较为类似，但是多了一个计算block偏移量的过程，对于一级和二级索引，其计算方法有所不同。同时，由于同一个inode可能含有多个block，需要通过循环实现。循环次数通过 `inode->blockCount` 获得。

根据inode表项中的一级和二级索引依次得到每一个block的偏移量，再根据每一个block的偏移量计算出需要修改的位图位置，将其置为0。偏移量的计算与inode类似，依次计算出i、j、k后定位。

最后不要忘了将修改后的 `superBlock` 写回对应位置：

```

fseek(file, 0, SEEK_SET);
fwrite((void *)superBlock, sizeof(SuperBlock), 1, file);
return 0;

```

3.1.2 rmdir()的实现

前面说，该函数的实现可以调用数次 `rm()` 函数，但是我们需要为 `rm()` 传入指定路径，故需要首先获取 `DirEntry` 的入口，依次遍历每个表项，为其执行 `rm()` 函数。故，`rmdir()` 需要完成：

- 读取inode和fatherinode
- 依次读取每个DirEntry表项，得到childinode的路径名
- 将child路径名作为参数传入 `rm()` 函数，删除该文件
- 全部子文件删除后，将自身路径作为参数传入 `rm()` 函数，删除目录

同样的，第一部分不在这里介绍。第二部分的实现我参考了 `ls()` 函数，该函数同样获得了所有子文件的名称，我们将当前路径 + '/' + 子文件名称进行拼接，就得到了子文件路径，执行 `rm()` 即可。拼接方法如下：

```
char childFilePath[NAME_LENGTH];
strcpy(destDirPath, childFilePath, NAME_LENGTH);
int length = strlen(childFilePath);
childFilePath[length] = '/';
strcpy(dirEntry.name, childFilePath + length + 1, NAME_LENGTH - length - 1);
ret = rm(driver, childFilePath, 1);
```

不过，这里需要额外注意的是，跳过了"."和".."两个文件。由于暂未实现隐藏文件（'.'开头的文件），我们可以直接这样判断：

```
if (dirEntry.name[0] == '.')
    continue;
```

在删除所有子文件后，再删除自身即可：

```
ret = rm(driver, destDirPath, 1);
```

上面的rm中额外传入的数字1仅用于帮助rm输出不同的信息，方便调试。

3.2. 实现内核的文件系统调用

修改位置：

1. lab5/lib/lib.h lab5/lib/syscall.c
2. lab5/kernel/include/fs.h lab5/kernel/kernel/fs.c
lab5/kernel/include/fs/minix.h
3. lab5/kernel/kernel/kvm.c
4. lab5/kernel/kernel/irqHandle.c

3.2.1. 修改内核调用，完善文件相关函数

3.2.1.1. 添加供用户使用的系统调用接口

在 syscall.c 中，添加open等函数提供给用户，同时在 lib.h 中增加函数的声明，再加上新的#define，如下：

```
//文件系统调用
#define SYS_OPEN 8
#define SYS_CLOSE 9
#define SYS_LSEEK 10
#define SYS_REMOVE 11
#define SYS_GETFILESIZE 12
//文件打开状态
#define O_WRITE 0x01
#define O_READ 0x02
#define O_CREATE 0x04
//定义方式
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
```

用户态的系统调用函数实现可以参考lab3中 fork() 等函数的实现，这里仅举 open() 为例：

```
int open(char *path, int flags)
{
    int size = 0;
    while (path[size] != '\0')
        size++;
    return syscall(SYS_OPEN, (uint32_t)path, (uint32_t)flags, size, 0, 0);
}
```

open中同时统计了一下文件名的长度，方便后续处理

除了手册中提到的函数外，还有一个方便实现ls和cat的函数 `int getfilesize(int fd)`，如函数名，该函数传入fd，返回fd指向的文件的大小。

3.2.1.2. 完善文件相关底层函数

在 `fs.c` 和 `fs.h` 中，我们加入了之前 `func.c` 中的那些文件操作函数的定义和声明，例如 `getAvailBlock()` 等等一系列函数；同时，修改函数内容，取消对FILE类型的操作，将读写由 `fread`、`fwrite` 换为 `diskRead`、`diskWrite`。由于这部分代码量较大但操作简单，基本是Ctrl C + Ctrl V完成，在这里不作详细阐述。

在 `minix.h` 中添加新的数据结构File，如下：

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

与手册相比新加入了length用于指明文件长度，方便实现lseek等。

3.2.1.3. 引入FCB

在 `kvm.c` 中加入文件控制块fcb

```
File fcb[MAX_FILE_NUM];
```

这里将MAX_FILE_NUM姑且设为256。

3.2.1.4. 修改内核调用过程

在 `irqHandle.c` 中首先补充之前define的那些变量，并为每一个新的系统调用新建独自の `syscallHandle` 函数，例如 `syscallOpen()` 等，然后，

修改 `syscallHandle()`，为每个新增的函数提供入口；

修改 `syscallRead()` 和 `syscallWrite()`，使其对文件操作时指向新的调用 `syscallFileRead()` 和 `syscallFileWrite()`。这里需要注意，按照UNIX设计思想，应该将之前的ShMem和标准输入输出也视为文件（fd号为3、0、1），但由于我没有实现这样的功能，故依然沿用之前的设计，而对文件类型进行单独的处理。以 `syscallWrite()` 为例：

```
void syscallWrite(struct TrapFrame *tf)
{
    switch (tf->ecx)
    { // file descriptor
    case STD_OUT:
        if (dev[STD_OUT].state == 1)
            syscallWriteStdOut(tf);
        break; // for STD_OUT
    case SH_MEM:
        if (dev[SH_MEM].state == 1)
            syscallWriteShMem(tf);
        break; // for SH_MEM
    default:
        syscallWriteFile(tf);
        break;
    }
}
```

3.2.2. open

在完成了之前的准备工作之后，我们只需要实现 `syscallOpen()` 的内容即可完成用户态调用 `open` 的全过程。

在这个函数中，我们要做的事情是：

- 读inode
- 如果失败，则没有该文件，检查是否有权限创建新文件，有则创建，没有则返回
- 否则，找到一个空闲的文件描述符，完成相关修改；若寻找中发现该文件已打开，则返回打开过后的索引
- 对于创建新文件，需要读取fatherinode，然后调用相关函数

对于读取inode之类，与3.1中的[实现](#)相比，仅多了一个读取文件路径的过程，这是由于有用户态向内核态的转变导致的。

```
char destFilePath[256];
int sel = tf->ds;
uint8_t *str = (uint8_t *)tf->ecx;
int size = tf->ebx;
int i = 0;
asm volatile("movw %0, %%es" :: "m"(sel));
uint8_t c;
while (i < size)
{
    asm volatile("movb %%es:(%1), %0" : "=r"(c) : "r"(str + i));
    destFilePath[i] = c;
    i++;
}
destFilePath[i] = '\0';
```


对于寻找索引，与之前pcb类似，遍历fcb即可。在找到后将对应state置为1，修改相关变量，再设置返回值即可：

```
pcb[current].regs.eax = index;
```

新文件的创立较为复杂，首先判断是否具有权限：

```
if ((flag & O_CREATE) != O_CREATE)
{
    putString("Failed to Find file ");putString(destFilePath);
    return;
}
```

然后读取父节点，先通过 `allocInode()` 分配一个inode，并修改父节点的DirEntry，再判断要创建的文件是否是目录类型，如果是，则需要执行一次 `mkdir()` 初始化目录。

这里需要注意，`allocInode()` 接收的参数为文件名而非其绝对路径，故还需要利用 `StringChrR()` 等函数对其进行一定的转换。而在创建新文件过后，依然需要执行之前的寻找fcb索引的过程。

3.2.3. close、lseek和remove

这三者相对来说实现较为简单。close只需要将索引对应的表项state置为0，remove读出路径后调用一次rm就行，这里都不再赘述。lseek相对复杂一点，但由于增加了File的length变量，也较为简单，根据不同的whence，使用不同的修改offset方式即可。

```
if (whence == SEEK_SET)
    fcb[fd].offset = offset;
else if (whence == SEEK_CUR)
    fcb[fd].offset += offset;
else if (whence == SEEK_END)
    fcb[fd].offset = fcb[fd].length + offset;
else
{
    putString("wrong seek parameter!");
    return;
}
```

在修改后进行判断是否合法：

```
if (fcb[fd].offset > fcb[fd].length || fcb[fd].offset < 0)
{
    pcb[current].regs.eax = -1;
    putString("Seek a wrong offset!");
    return;
}
```

3.2.4. 文件的read和write

与简单的ShMem相比，读写文件多了block的概念。读操作相对来说较为简单。通过：

```
int blockIndex = offset / superblock.blockSize;
int insideoffset = offset % superblock.blockSize;
```

我们可以得到对应的是哪一个块，以及块内的偏移量，然后通过 `readBlock()` 读取相应块到 `buffer` 中，在从 `buffer` 中从 `insideoffset` 开始读取字符，这时可能会出现读取到了新的块，还需要修改相关值后重新读取。在全部读取完毕后，还需要修改 `pcb[fd].offset`。逐字符读取的过程如下：

```
while (i < size)
{
    if (insideoffset == superblock.blockSize) //read a new block
    {
        blockIndex++;
        int ret = readBlock(&superblock, &inode, blockIndex, buffer);
        if (ret == -1)
            break;
        insideoffset = 0;
        continue;
    }
    c = buffer[insideoffset];
    asm volatile("movb %0, %%es:(%1)" :: "r"(c), "r"(str + i));
    i++;
    insideoffset++;
}
```

与之前不同的是，写操作不再是读的简单镜像。在写的过程中，可能会出现写到了新的block，这时就要通过 `allocBlock()` 分配block，同时，由于文件可能并非新文件，我们不能简单的创建一个空buffer直接写，而是需要先 `readBlock()`，将已有数据读出，再在对应位置修改，对该block的所有操作完成之后，通过 `writeBlock()` 写回block。下面是创建新的block的过程。

```
while (blockIndex + 1 > inode.blockCount)
{
    int ret = allocBlock(&superblock, &inode, inodeOffset);
    if (ret == -1)
    {
        putString("Failed to alloc block while writing file!");
        pcb[current].regs.eax = -1;
        return;
    }
}
```

在写操作成功完成之后，除了修改 `pcb`，还需要修改 `inode` 的 `size`，这样才算彻底完成对文件的修改。

```
pcb[current].regs.eax = i;
pcb[fd].offset += i;
pcb[fd].length += i;
inode.size += i;
diskwrite((void *)&inode, sizeof(Inode), 1, inodeoffset);
return;
```

3.2.5. 额外实现的GetFileSize

这个函数会返回文件长度，用于辅助 `ls` 和 `cat` 的实现。由于 `length` 已经记录在 `pcb` 内，故代码较为简单：

```
void syscallGetFileSize(struct TrapFrame *tf)
{
    int fd = tf->ecx;
    pcb[current].regs.eax = fcb[fd].length;
}
```

3.3. 实现ls和cat用户函数

修改位置: lab5/lib/lib.h lab5/lib/type.h lab5/lib/syscall.c

在前面一系列函数的支持下，ls和cat实现起来就会变得非常简单。

3.3.1. ls

通过open和read，我们可以很容易获得当前目录对应的DirEntry的全部内容，但是这个内容是以数组形式存储的，需要将其强制类型转换，因此，首先，我们在type.h中补充上DirEntry类型的定义：

```
union DirEntry {
    uint8_t byte[128];
    struct
    {
        int32_t inode; // index in inode table, started from 1, 0 for unused.
        char name[64];
    };
};
typedef union DirEntry DirEntry;
```

这样的话，我们在ls中作的操作就非常简单了：

- 使用open打开文件，操作符为O_READ | O_DIRECTORY
- 读取文件长度size
- 使用read从文件中读取size个字节到buffer
- 将buffer转换为DirEntry*，逐个读取子文件名称并打印

注意读取的时候对于inode=0的会直接忽略（视为被删除），同时，如果不显示.和..，循环将从2开始，如下所示：

```
for (int i = 2; i < ret / 128; i++)
{
    if (dirent[i].inode != 0)
    {
        printf("%s ", dirent[i].name);
    }
}
```

3.3.2. cat

cat的逻辑与ls并无太大差别，同样地将文件整个读取到buffer中，但不需要类型转换，而是直接逐个打印即可。

```
int ret = read(fd, buffer, size);
int i = 0;
while(buffer[i]!='\0')
{
    printf("%c", buffer[i]);
    i++;
}
```