

OS Lab

Lab5

191220008 陈南瞳

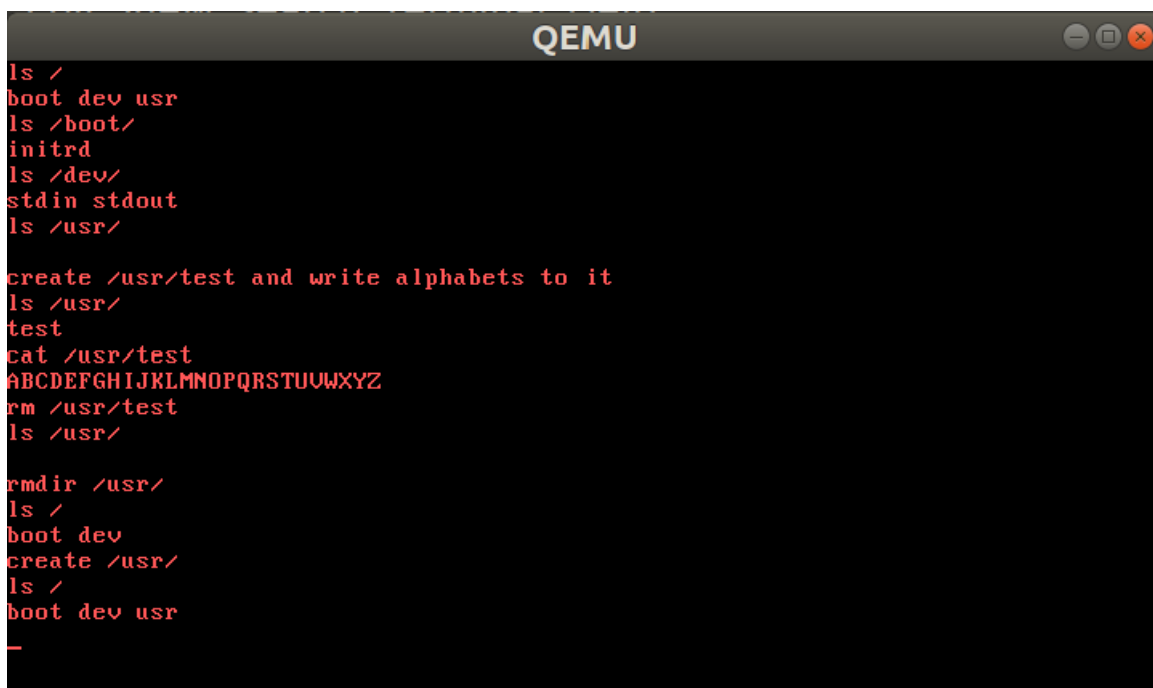
924690736@qq.com

一、实验进度

完成了Lab5中的如下内容：

- 格式化程序（阅读）
- 内核支持文件读写：open, read, write, lseek, close, remove
- 用户程序：基于open、read等系统调用实现ls、cat

二、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
-
```

三、实验修改的代码位置

(一) 格式化程序

仅需阅读。

(二) 内核支持文件读写

完成文件系统相关的调用：open, write, read, lseek, close, remove。

1、kernel/irqHandle.c

(1) syscallOpen

- 首先读取被打开文件的inode。

若文件不存在，则查看权限，进行创建或者返回失败。

创建时，检查输入的文件的上一级目录是否存在。若存在，则分配inode节点。

若文件存在，则检查打开方式和inode指示的文件类型是否匹配，若不匹配的话则返回错误。

- 然后再检查被打开的文件是否正在使用。
- 最后在FCB列表中填写相关信息。

(2) syscallWrite, syscallWriteStdOut, syscallWriteFile

框架代码提供了从标准输入输出，所以只需要完成从文件中输出。

定位到文件后，计算出需要写入的块的位置，然后依次输出。当写满一个块后，需要再分配新的块然后新的块上继续输出。

在输出完成后，更新FCB中对应的offset。

(3) syscallRead, syscallReadStdIn, syscallReadFile

框架代码提供了从标准输入读取，所以只需要完成从文件中读取。

定位到文件后，计算出需要读取的块的位置和序列，然后依次读取。

在读取完成后，更新FCB中对应的offset。

(4) syscallLseek

将FCB中对应元素的offset修改为对应值。若offset越界，则返回错误。

(5) syscallClose

修改FCB中对应表项。将state和其他相关参数置为0。

(6) syscallRemove

与syscallOpen的过程相反。

- 首先读取待删除文件的Inode。

若待删除的文件不存在，返回错误。

- 然后检查输入的待删除文件的上一级目录，并回收inode节点。
- 最后在FCB列表中还原相关信息。

2、lib/syscall.c

```
int open (char *path, int flags) {
    return syscall(SYS_OPEN, (uint32_t)path, (uint32_t)flags, 0, 0, 0);
}

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

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

int lseek (int fd, int offset, int whence) {
    //TODO: Complete the function 'lseek' just like the function 'open'.
    return syscall(SYS_LSEEK, (uint32_t)fd, (uint32_t)offset, (uint32_t)whence,
0, 0);
}

int close (int fd) {
    //TODO: Complete the function 'close' just like the function 'open'.
    return syscall(SYS_CLOSE, (uint32_t)fd, 0, 0, 0, 0);
}

int remove (char *path) {
    //TODO: Complete the function 'remove' just like the function 'open'.
    return syscall(SYS_REMOVE, (uint32_t)path, 0, 0, 0, 0);
}
```

(三) 用户程序

基于open、read等系统调用实现ls、cat两个函数。

1、app/main.c

(1) ls

```
int ls(char *destFilePath) {
    ...
    ret = read(fd, buffer, 512 * 2);
    while (ret != 0) {
        // TODO: Complete 'ls'.
        dirEntry = (DirEntry *)buffer;
```

```

        for (i = 0; i < 512 * 2 / sizeof(DirEntry); i++) {
            if (dirEntry[i].inode != 0) {
                printf("%s", dirEntry[i].name);
                printf(" ");
            }
        }
        ret = read(fd, buffer, 512 * 2);
    }
    ...
}

```

读取并逐个打印即可。

(2) cat

```

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

```

从文件中读出，输出到标准输出设备上即可。

四、对讲义或框架代码中某些思考题的看法

思考题

1、为什么使用文件描述符？我们可以直接使用文件名做为文件的标识：

```

int read(const char *filename, void *buffer, int size);
int write(const char *filename, void *buffer, int size);

```

这样做有什么缺陷吗？

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。

缺陷是明显的，文件系统中不能确保没有同名文件的情况，此时若用文件名作为标识，则无法确定目标文件。

2、不需要进行更新的exec

为什么内核在处理exec时，不需要对进程文件描述符表和系统文件打开表进行任何修改？

调用exec时，并不会对文件描述符表和系统文件打开表中的参数造成影响，与其无关。

3、cd程序在哪里

我们可以通过which命令来查看一个shell命令的程序所在的绝对路径, 例如

```
$ which gcc
/usr/bin/gcc
```

输出结果告诉我们, 我们平时使用的gcc命令的程序其实位于/usr/bin/目录下. 但对于我们使用得最多的cd命令, which却找不到它所在的目录, 你知道这是为什么吗? 如果你感到困惑, 请到互联网上搜索相关资料.

cd是内置命令, 是bash内建的命令, 不在PATH环境变量内, 而 which 默认是找 PATH 内所规范的目录, 所以无法找到。

六、实验心得或对提供帮助的同学的感谢

本次实验难度较上次有所提升，主要是文件调用函数中需要处理和注意的细节很多。但好在助教给出了TODO和对应需要做得事情，所以不至于一头雾水，只需针对每个函数进行琢磨即可。

通过本次实验也加深了对文件系统内部结构的认识和理解，对相关内容的学习有一定的巩固作用。