

OS Lab5 Report

1. 个人信息

姓名：余帅杰

学号：181860077

邮箱：3121416933@qq.com

2. 实验进度

1. 完成rm和rmdir
2. 完成open read write lseek close remove文件系统调用的实现
3. 实现ls和cat用户程序
4. 实现shell，并移植部分linux的命令程序

实验内容

1. 补全格式化程序rm, rmdir
2. 完成系统调用实现
3. 完成ls, cat用户程序
4. 实现shell

代码修改位置

(这次实验耗时很长，修改了大量的位置，同时中间有反复的代码修改，部分修改可能和原框架代码混淆了(不记得了 qwq))

genFS/func.c

增加了rm和rmdir的实现

增加了释放inode和释放block的函数，包括有freeinode, freeblock, freeLastBlock, setAllocBlock, setAllocInode等函数

kernel/irqhandle.c

实现syscall开头的readFile, writeFile, Open, Close, Remove, Lseek, Ls函数

同时对原先的syscallWrite和syscallRead函数进行了修改，以对接新的文件读写接口

修改syscallHandle函数，在中断处理程序里加上新实现的函数

在代码的首部加上所需的宏定义(包括文件权限，新的中断标识符等)

代码的首部加上file数组以及superblock等所需数据的声明

kernel/fs.c

在这个文件里迁移和修改了func.c里的文件系统处理函数，包括readInode, freeInode等等

lib/syscall.c

创建新的文件系统的用户接口open, close等

kernel/include/fs/minix.h

kernel/include/x86/memory.h

添加新的文件系统结构体以及必要的数据和函数声明

整体修改

最后增加了shell和testcase文件夹

并对应的增加了main文件和makefile，同时在lab5下的makefile中对编译选项增加了新的编译对象

3. 实验思路

框架分析

之前的实验对文件系统的了解不够的深入，本次实验耗时极长，期间遇到了大量的问题，经过反复的阅读框架代码，这里简单的记录一下自己的认识。

整体架构分析

整个文件系统的设定是superblock在第一块，同时func.c里的file对应的driver实际上就是磁盘，也就是这个读写使用fseek和diskread的效果相同，superblock在第一块

在本次实验里inode的功能做了简化，14块只有一个一级拓展地址（14块不够了作为指针）

同时整个文件系统同时维护着三方面的数据结构

一个是FCB数组，记录的是当前的文件读写的指针位置，以及对应inode的偏移量，于是可以顺利找到inode，然后进行一系列的操作

另一个结构是inode，本身管理着多个block的位置信息，是基本管理单位

还有一个结构是dirEntry结构，这个结构在getDirEntry函数中有着很大的使用，对于目录inode，block里保存的不是直接的数据，而是目录信息dirEntry，对应的读取得到可以得到自己的文件目录信息和子节点的inode的位置信息。

以若干重要函数为例。

getAvailableInode

这个函数对于理解位图和inode位置以及如何申请释放inode有着很大的教育意义

inode和block的位图记录着使用的情况，对应的inode或者block被使用，那么就会被置为1，这就解释了如何对位图进行操作，以及如何用位图反推inode位置（下图举例子就是getAvailableInode函数里）

对应的这个处理的思路就反过来推进了freeInode的处理思路

```

inodeBitmapOffset = superBlock->inodeBitmap;
inodeTableOffset = superBlock->inodeTable;
fseek(file, inodeBitmapOffset * SECTOR_SIZE, SEEK_SET);
fread((void *)&inodeBitmap, sizeof(InodeBitmap), 1, file);
for (j = 0; j < superBlock->availInodeNum / 8; j++) {
    if (inodeBitmap.byte[j] != 0xff) {
        break;
    }
}
for (k = 0; k < 8; k++) {
    if ((inodeBitmap.byte[j] >> (7-k)) % 2 == 0) {
        break;
    }
}
inodeBitmap.byte[j] = inodeBitmap.byte[j] | (1 << (7 - k));

*inodeOffset = inodeTableOffset * SECTOR_SIZE + (j * 8 + k) * sizeof(Inode);

```

下图为反推得到释放inode的时候的处理例程，对应的block的申请和释放就不难理解

```

j=(inodeOffset-inodeTableOffset * SECTOR_SIZE )/sizeof(Inode)/8;
k=(inodeOffset-inodeTableOffset * SECTOR_SIZE )/sizeof(Inode)%8;

fseek(file, inodeBitmapOffset * SECTOR_SIZE, SEEK_SET);
fread((void*)&inodeBitmap, sizeof(InodeBitmap), 1, file);
if ((inodeBitmap.byte[j] >> (7-k)) % 2 == 0)
    return -1;

superBlock->availInodeNum ++;
inodeBitmap.byte[j] = inodeBitmap.byte[j] ^ (1 << (7-k));

```

readBlock

这个函数对于理解inode如何管理和使用block有着很大的意义

可以看到划分出的分界线，如果小于的第一道界限就可以直接把inode里的数组里作为下标进行读取，也就是inode里的数组里存的是block的位置，如果block较多管不过来，就是第二道界限通过读取指针，得到二级地址（拓展地址），然后再去读取block，类似于数组的指针

```

int readBlock (FILE *file, SuperBlock *superBlock, Inode *inode, int blockIndex, uint8_t *buffer) {
    // calculate the index and bound
    int divider0 = superBlock->blockSize / 4;
    int bound0 = POINTER_NUM;
    int bound1 = bound0 + divider0;

    uint32_t singlyPointerBuffer[divider0];

    if (blockIndex < bound0) {
        fseek(file, inode->pointer[blockIndex] * SECTOR_SIZE, SEEK_SET);
        fread((void *)buffer, sizeof(uint8_t), superBlock->blockSize, file);
        return 0;
    }
    else if (blockIndex < bound1) {
        fseek(file, inode->singlyPointer * SECTOR_SIZE, SEEK_SET);
        fread((void *)singlyPointerBuffer, sizeof(uint8_t), superBlock->blockSize, file);
        fseek(file, singlyPointerBuffer[blockIndex - bound0] * SECTOR_SIZE, SEEK_SET);
        fread((void *)buffer, sizeof(uint8_t), superBlock->blockSize, file);
        return 0;
    }
    else
        return -1;
}

```

getDirEntry

这个函数如整体分析所述，揭示了父子目录的管理方案，以及DirEntry的作用和使用方法

通过读取DirEntry获得子节点的信息和当前节点的路径名。同时可以通过子节点信息轻松的访问和修改子节点

这个函数为后续的实现ls做了极大的铺垫

```
1 int getDirEntry (FILE *file, SuperBlock *superBlock, Inode *inode, int dirIndex, DirEntry *destDirEntry) {
2     int i = 0;
3     int j = 0;
4     int ret = 0;
5     int dirCount = 0;
6     DirEntry *dirEntry = NULL;
7     uint8_t buffer[superBlock->blockSize];
8
9     for ([i = 0; i < inode->blockCount; i++])
10    {
11        ret = readBlock(file, superBlock, inode, i, buffer);
12        if (ret == -1)
13            return -1;
14        dirEntry = (DirEntry *)buffer;
15        for (j = 0; j < superBlock->blockSize / sizeof(DirEntry); j++)
16        {
17            //printf("Test in get %s\n", dirEntry[j].name);
18            if (dirEntry[j].inode != 0)
19            {
20                if (dirCount == dirIndex)
21                    break;
22                else
23                    dirCount++;
24            }
25        }
26        if (j < superBlock->blockSize / sizeof(DirEntry))
27            break;
28    }
29    if (i == inode->blockCount)
30        return -1;
31    else
32    {
33        destDirEntry->inode = dirEntry[j].inode;
34        strcpy(dirEntry[j].name, destDirEntry->name, NAME_LENGTH);
35        return dirCount;
36    }
37 }
```

具体思路

Rm和Rmdir

Rm和Rmdir最核心的内容就是释放inode以及block，还有最后的对位图和superblock的修改

实际上上文已经进行了分析，另外的内容只要对照着Alloc的时候做法反着来就行

Open

对于Open要做的事情有

1. 打开文件
2. 文件不存在，查看权限创建或者失败
3. 文件存在
4. 不管存不存在都要在FCB里找到空位，填好相关的信息

打开文件可以想到文件的ID：Inode，也就是readInode，通过地址索引到inode，读取的成功与否表明文件是否已经存在，最后在文件FCB列表里填写信息即可，同时设置state=1表明活跃

Close

close要做的事情最为简单，针对FCB进行处理，设置state=0表明文件不活跃，被关闭

Remove

Remove函数实际上完全可以参考rm，本质上也是读取信息然后Inode和block的回收

Lseek

要做的事情也相对简单，只需要针对FCB表项里的对于的offset进行修改即可，这个指示的是文件指针的位置，对应修改即可

Read

在read之前就已经打开了文件，得到了在文件系统里的标识符（下标）

对应的标识符可以索引到表项，进而得到了inode的位置进行读写得到文件信息

同时注意到文件的offset，有可能不是读/写不发生在文件头，也就是不能序列化读写，要计算好开始读写的块的位置和序列，然后进行读写即可，对应的inode有size进行指示，如果读的当前块满了，就下一块，最后设置好文件的指针位置即可

最后的结果无非是读完了文件但是size没到，报错，要么就是size读满了返回即可

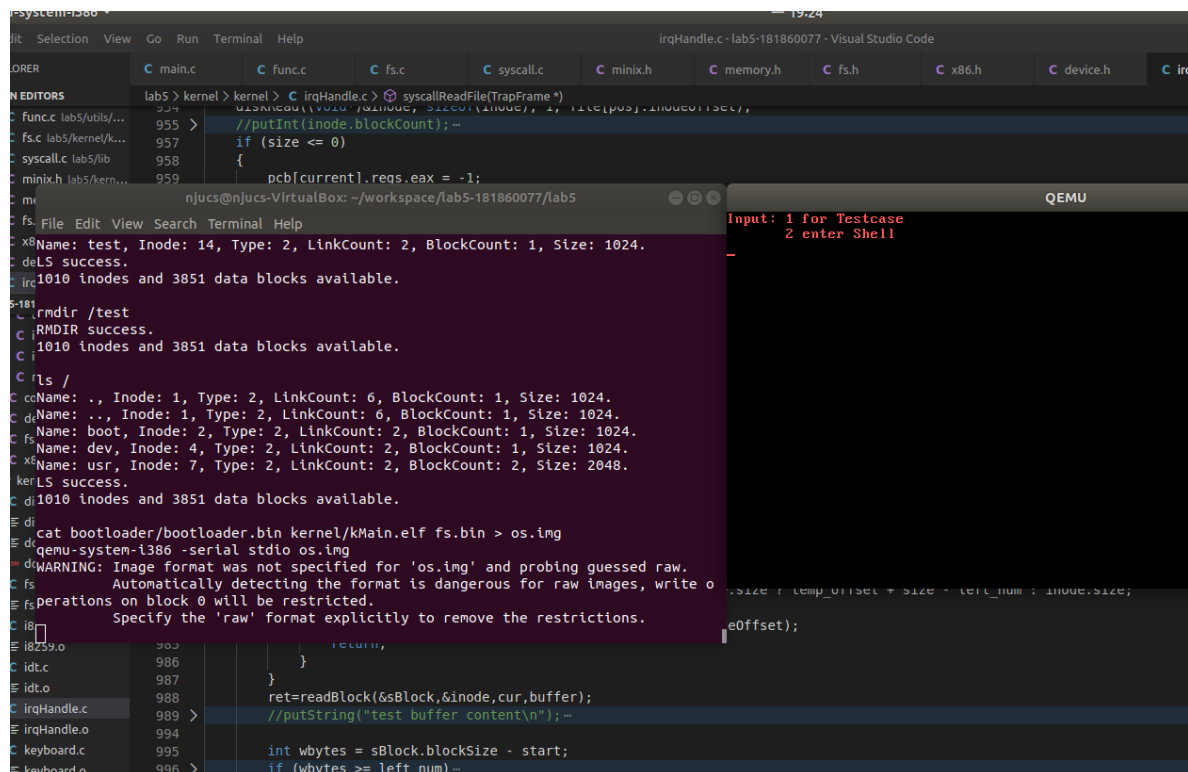
Write

Write的本质和Read是相似的，前半部分一样，只不过后半部分的时候，如果一个块写完了要继续AllocBlock得到新的块，然后继续读写即可，其中注意维持数据的一致性，先读再写即可。

4. 实验结果

make play后结果如下

输入1进入测试样例，输入2进入实现的shell。



```
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write off
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

cat bootloader/bootloader.bin kernel/kMain.elf fs.bin > os.img
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write off
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

ls /
ccName: .., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
eName: .., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
fsName: boot, Inode: 2, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
fsName: dev, Inode: 4, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
fsName: usr, Inode: 7, Type: 2, LinkCount: 2, BlockCount: 2, Size: 2048.
kerLS success.
1010 inodes and 3851 data blocks available.

cat bootloader/bootloader.bin kernel/kMain.elf fs.bin > os.img
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write off
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

Input: 1 for Testcase
      2 enter Shell
```

Rm和Rmdir

touch temp前后

```
ls /usr
Name: ., Inode: 7, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 5, BlockCount: 1, Size: 1024.
Name: print, Inode: 8, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13716.
Name: bounded_buffer, Inode: 9, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
Name: philosopher, Inode: 10, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13824.
Name: reader_writer, Inode: 11, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13760.
Name: Shell, Inode: 12, Type: 1, LinkCount: 1, BlockCount: 18, Size: 18192.
Name: testcase, Inode: 13, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
LS success.
1011 inodes and 3853 data blocks available.

touch /usr/temp
TOUCH success.
1010 inodes and 3852 data blocks available.

ls /usr
Name: ., Inode: 7, Type: 2, LinkCount: 2, BlockCount: 2, Size: 2048.
Name: .., Inode: 1, Type: 2, LinkCount: 5, BlockCount: 1, Size: 1024.
Name: print, Inode: 8, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13716.
Name: bounded_buffer, Inode: 9, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
Name: philosopher, Inode: 10, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13824.
Name: reader_writer, Inode: 11, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13760.
Name: Shell, Inode: 12, Type: 1, LinkCount: 1, BlockCount: 18, Size: 18192.
Name: testcase, Inode: 13, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
Name: temp, Inode: 14, Type: 1, LinkCount: 1, BlockCount: 0, Size: 0.
LS success.
1010 inodes and 3852 data blocks available.
```

rm temp之后很明显没了

```
RM /usr/temp
RM success.
1011 inodes and 3852 data blocks available.

ls /usr
Name: ., Inode: 7, Type: 2, LinkCount: 2, BlockCount: 2, Size: 2048.
Name: .., Inode: 1, Type: 2, LinkCount: 5, BlockCount: 1, Size: 1024.
Name: print, Inode: 8, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13716.
Name: bounded_buffer, Inode: 9, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
Name: philosopher, Inode: 10, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13824.
Name: reader_writer, Inode: 11, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13760.
Name: Shell, Inode: 12, Type: 1, LinkCount: 1, BlockCount: 18, Size: 18192.
Name: testcase, Inode: 13, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13752.
LS success.
1011 inodes and 3852 data blocks available.
```

mkdir以及rmdir前后

```
mkdir /test
MKDIR success.
1010 inodes and 3851 data blocks available.

ls /test
Name: ., Inode: 14, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
LS success.
1010 inodes and 3851 data blocks available.

ls /
Name: ., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
Name: boot, Inode: 2, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: dev, Inode: 4, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: usr, Inode: 7, Type: 2, LinkCount: 2, BlockCount: 2, Size: 2048.
Name: test, Inode: 14, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
LS success.
1010 inodes and 3851 data blocks available.

rmdir /test
RMDIR success.
1010 inodes and 3851 data blocks available.

ls /
Name: ., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
Name: .., Inode: 1, Type: 2, LinkCount: 6, BlockCount: 1, Size: 1024.
Name: boot, Inode: 2, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: dev, Inode: 4, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: usr, Inode: 7, Type: 2, LinkCount: 2, BlockCount: 2, Size: 2048.
LS success.
1010 inodes and 3851 data blocks available.
```

LS和CAT测试

ls的输出格式就是一行输出，用空格隔开

可以看到输出的结果和预期以及之前的结果一致

cat直接输出文件内容，显然正确

下图是分别设置写入26个和512个的结果


```
c lab5/kernel/k... 9 do ls(destFilePath, info);
scall.c lab5/lib 10 printf("%s\n", info);
return 0;

QEMU
Input: 1 for Testcase
2 enter Shell
Start Test !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ls /
.. boot dev usr
ls /boot/
.. initrd
ls /dev/
.. stdin stdout
ls /usr/
.. print bounded_buffer philosopher reader_writer Shell testcase
ls /usr/
.. print bounded_buffer philosopher reader_writer Shell testcase
cat /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
test end

34 ls("/");
35 ls("/boot/");
36 ls("/dev/");
37 ls("/usr/");
38 fd = open("/usr/test", O_WRITE | O_READ | O_CREATE);
39 for ([i = 0; i < 26; i ++])
40 {
41     tmp = (char)(i % 26 + 'A');
42     write(fd, (uint8_t*)&tmp, 1);
43 }
44 close(fd);
45 ls("/usr/");
46 cat("/usr/test");
47 printf("test end\n");
48 exit();
49 return 1;
50 }
```

```
955 > //putint(inode.blockCount);-
957 if (size <= 0)
958 {
QEMU
Input: 1 for Testcase
2 enter Shell
Start Test !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ls /
.. boot dev usr
ls /boot/
.. initrd
ls /dev/
.. stdin stdout
ls /usr/
.. print bounded_buffer philosopher reader_writer Shell testcase
ls /usr/
.. print bounded_buffer philosopher reader_writer Shell testcase
cat /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
test end

982 inode.size = temp_offset + size - left_num > inode->
983 tf->eax = size - left_num; // bytes have written
984 diskWrite(&inode, sizeof(inode), 1, file[pos].ino
985 return;
986 }
987 }
988 ret=readBlock(&sBlock,&inode,cur,buffer);
989 > //putString("test buffer content\n");-
994
995 int wbytes = sBlock.blockSize - start;
996 > if (wbytes >= left_num)-
1016 > else-
1035 }
1036 //putString("Hit unk Bad Trap in Test Write\n");
1037 tf->eax=-1;
1038 return;
```

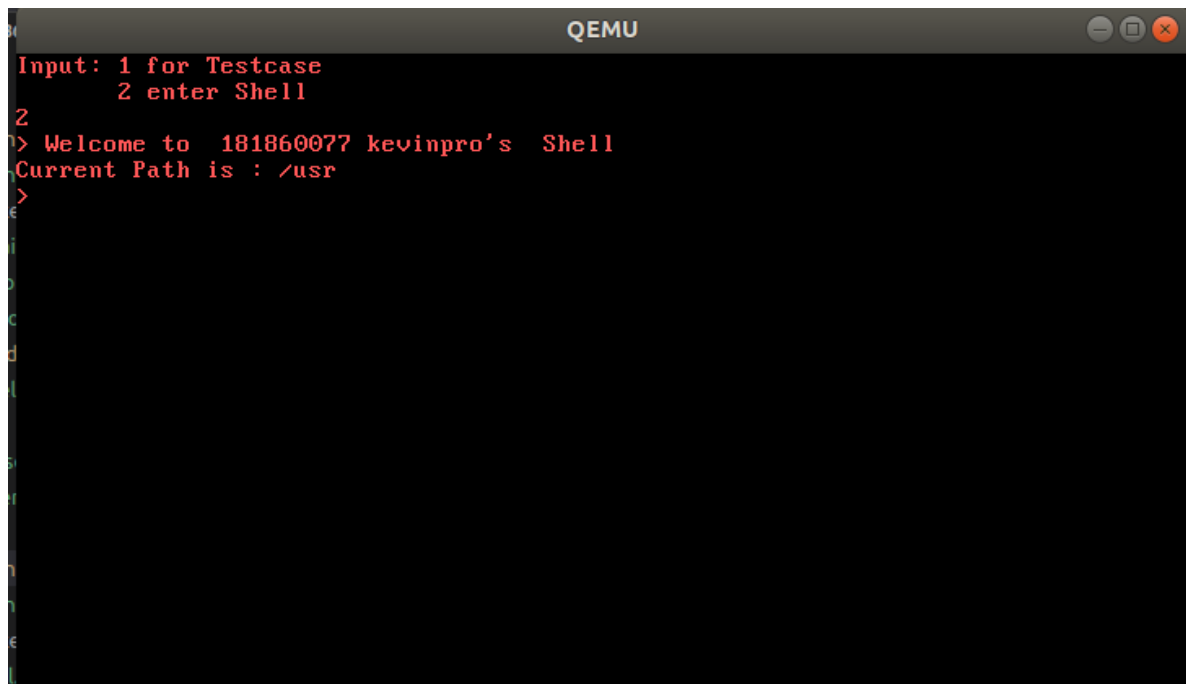
Shell

本次实现的Shell移植了Linux下的ls, cat, cd, pwd, rm, echo, quit

围绕文件构造了一个可用系统

(注1: ">"开头的是我输入指令的位置, 后面的截图里此标志符后的字符串是我输入的指令)

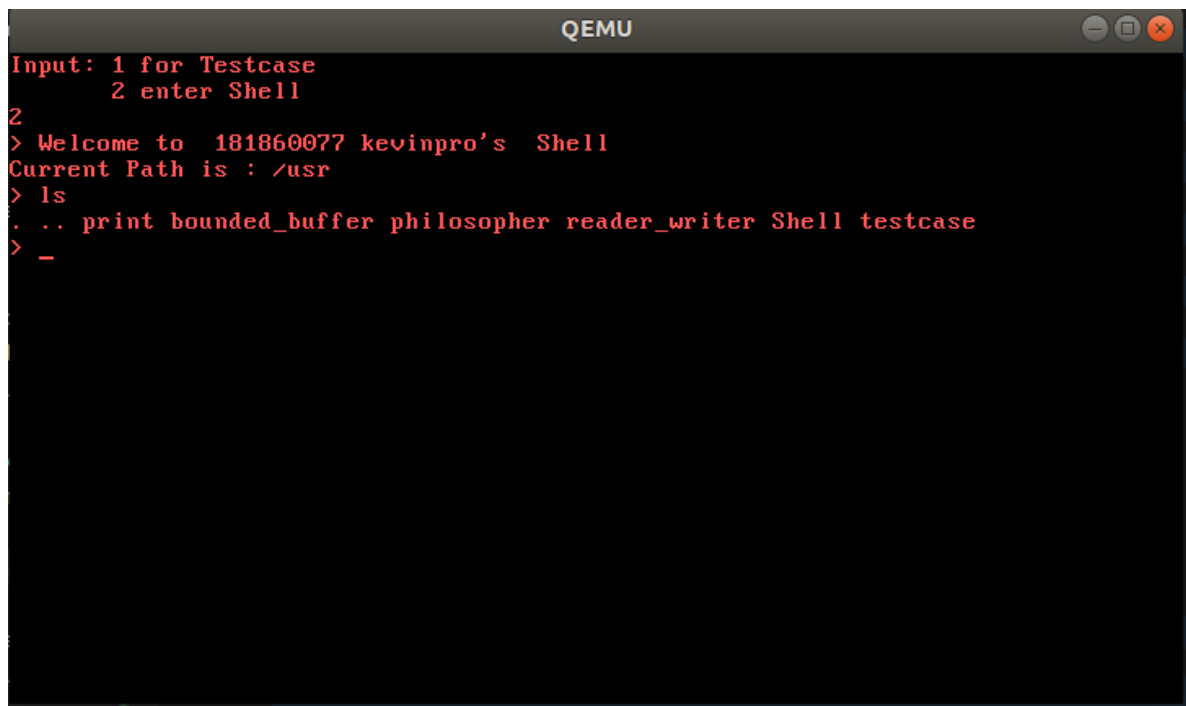
(注2: 本次实验的框架代码直接继承于Lab4, 保留了源文件系统的文件)



```
QEMU
Input: 1 for Testcase
      2 enter Shell
2
> Welcome to 181860077 kevinpro's Shell
Current Path is : /usr
>
```

ls

支持输出目录下的所有的文件和文件夹



```
QEMU
Input: 1 for Testcase
      2 enter Shell
2
> Welcome to 181860077 kevinpro's Shell
Current Path is : /usr
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase
> _
```

cd

cd的实现符合原Linux的使用习惯

对于“cd..” 返回上一级以及“cd .”在原目录，以及根目录的特殊处理等进行了适配，和原linux一致
具体的实验结果如下可见

```
QEMU
Input: 1 for Testcase
       2 enter Shell
2
> Welcome to 181860077 kevinpro's Shell
Current Path is : /usr
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase
> cd ..
> ls
. .. boot dev usr
> cd dev
> ls
. .. stdin stdout
> cd .
> ls
. .. stdin stdout
> _
```

Echo

原linux中Echo可以用于重定向直接向文件中写入数据

这里复刻了两条规则

Echo "A" > B, 则若B存在则覆盖写入, 若B不存在则创建文件

Echo "A" >> B 则不覆盖, 在后面增加文件

下面测试样例可见

1. 文件不存在, 则创建了hh这个文件
2. cat输出文件内容kkk, 符合
3. 使用||||覆写文件, 覆盖了
4. 使用学号增加数据在文件末
5. cat打印出结果一致

```
QEMU
Input: 1 for Testcase
      2 enter Shell
2
> Welcome to 181860077 kevinpro's Shell
Current Path is : /usr
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase
> echo "kkk" > hh
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase hh
> cat hh
kkk
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase hh
> echo "llll" > hh
> cat hh
llll
> echo "181860077" >> hh
> cat hh
llll181860077
> _
```

Cat

测试可见上面的结果，可知正确

rm和pwd

这两个函数一起测试，结果如下

可以看到文件被删除

```
QEMU
Input: 1 for Testcase
      2 enter Shell
2
> Welcome to 181860077 kevinpro's Shell
Current Path is : /usr
> echo "kk" > hh
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase hh
> rm hh
> ls
. .. print bounded_buffer philosopher reader_writer Shell testcase
> pwd
Current Path is : /usr
>
```

结合了上述实现的指令，这里就构造了一个功能相对完全一些的文件系统功能了

1. 可以写和创建文件。
2. 自由写文件可以覆写也可以在末尾添加。

3. 文件的输出和删除指令都具有了。
4. 可以灵活的在各级目录之间切换

5.自由报告

1. 本次实验耗时很久，重读代码收获很大，把文件系统整明白了
2. shell挺有意思的，但是时间有限，很多指令没时间实现，已经实现的指令在功能上和鲁棒性上都
有所欠缺
3. 最后感谢助教抽出时间验收附加实验