

Lab 5 实验报告

231880101 孙俊晖

邮箱: 2033282932@qq.com

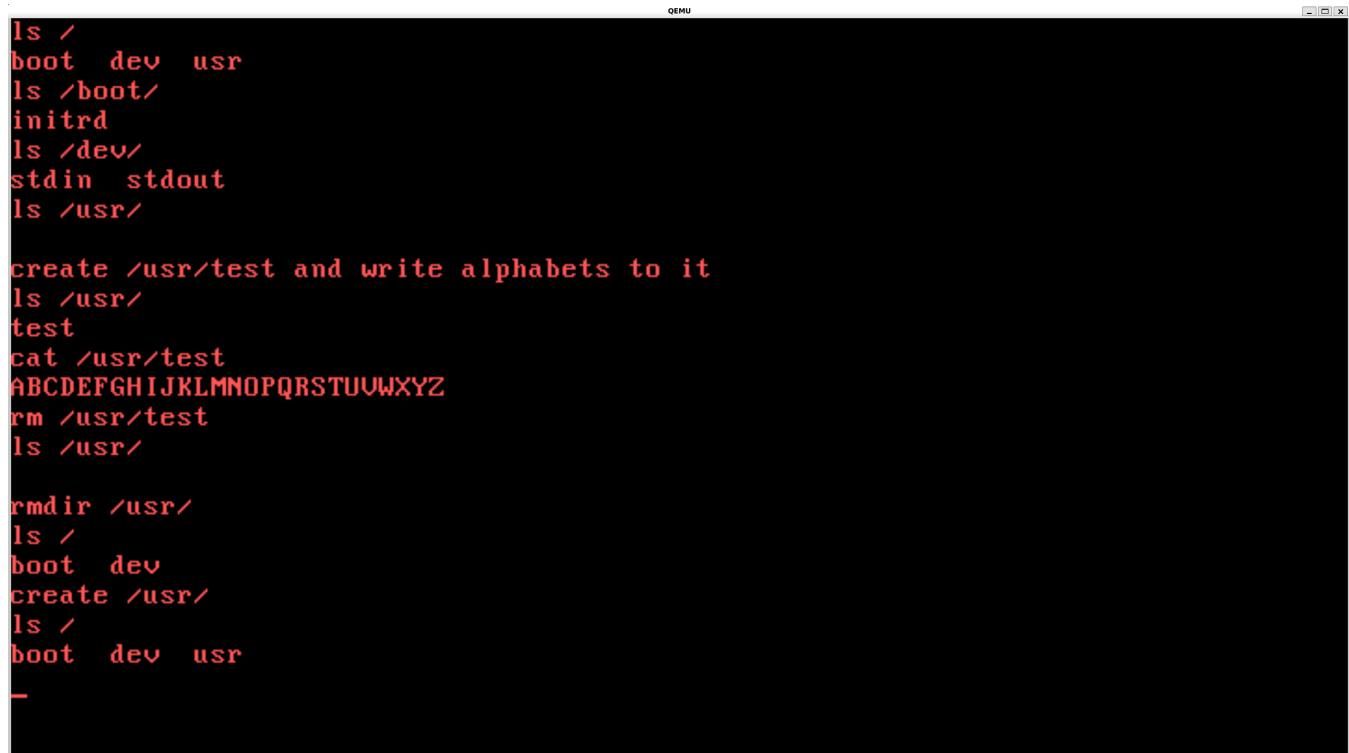
一、实验完成情况

1. 实验进度

我完成了除选做内容以外的所有内容。

2. 实验结果

我顺利通过了用户程序的测试，输出结果与预期结果一致。



```
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
-
```

二、实验过程

1. 完成库函数

首先完成 Lab5/lib/syscall1.c 中的五个库函数 *write*, *read*, *lseek*, *close*, *remove*，这部分非常简单，只需要参考库函数 *open* 的实现，将每个库函数的系统调用号和相应的参数传递给内核即可。

```
1 int open (char *path, int flags) {
2     return syscall(SYS_OPEN, (uint32_t)path, (uint32_t)flags, 0, 0, 0);
3 }
4
5 int write (int fd, uint8_t *buffer, int size) {
```

```

6     //TODO: Complete the function 'write' just like the function 'open'.
7     return syscall(SYS_WRITE, (uint32_t)fd, (uint32_t)buffer, (uint32_t)size, 0, 0);
8 }
9
10 int read (int fd, uint8_t *buffer, int size) {
11     //TODO: Complete the function 'read' just like the function 'open'.
12     return syscall(SYS_READ, (uint32_t)fd, (uint32_t)buffer, (uint32_t)size, 0, 0);
13 }
14
15 int lseek (int fd, int offset, int whence) {
16     //TODO: Complete the function 'lseek' just like the function 'open'.
17     return syscall(SYS_LSEEK, (uint32_t)fd, (uint32_t)offset, (uint32_t)whence, 0, 0);
18 }
19
20 int close (int fd) {
21     //TODO: Complete the function 'close' just like the function 'open'.
22     return syscall(SYS_CLOSE, (uint32_t)fd, 0, 0, 0, 0);
23 }
24
25 int remove (char *path) {
26     //TODO: Complete the function 'remove' just like the function 'open'.
27     return syscall(SYS_REMOVE, (uint32_t)path, 0, 0, 0, 0);
28 }

```

2. 完善系统调用处理

(1) 完善 syscallOpen 函数

首先，函数 syscallOpen 初始化了一系列变量，例如用户进程基地址baseAddr、文件路径str、父目录inode偏移量fatherInodeOffset、目标inode偏移量destInodeOffset，这部分的代码助教已经提前帮我们写好了。

```

1 int i;
2 char tmp = 0;
3 int length = 0;
4 int cond = 0;
5 int ret = 0;
6 int size = 0;
7 int baseAddr = (current + 1) * 0x100000; // base address of user process
8 char *str = (char*)sf->ecx + baseAddr; // file path
9 Inode fatherInode;
10 Inode destInode;
11 int fatherInodeOffset = 0;
12 int destInodeOffset = 0;

```

然后，使用函数 readInode 尝试读取目标文件的inode，返回值为 ret，用于判断目标文件是否存在。

```

1 | ret = readInode(&sBlock, gDesc, &destInode, &destInodeOffset, str);

```

如果目标文件存在(ret==0)，首先检查 O_DIRECTORY 标志位和文件类型是否匹配，如果文件以目录模式打开

((sf->edx >> 3) % 2 == 1)但实际不是目录(destInode.type != DIRECTORY_TYPE)，或者文件是目录(destInode.type == DIRECTORY_TYPE)但没有用目录模式打开((sf->edx >> 3) % 2 == 0))，则返回错误。

然后，寻找空闲的文件描述符，如果找到了，则将其标记为使用中，记录 inode 位置，设置偏移量为0，保存打开的标志 sf->edx，完成了这些工作后返回文件描述符。如果最后没有找到空闲的文件描述符，则返回错误。

```
1 if (ret == 0) { // file exist
2     // TODO: You need to consider the situations that O_DIRECTORY is set or not set,
3     // the file refer to a device or a file which is in use or not.
4     // 检查 O_DIRECTORY 标志和文件类型是否匹配
5     if((destInode.type != DIRECTORY_TYPE && (sf->edx >> 3) % 2 == 1) || (destInode.type
6     == DIRECTORY_TYPE && (sf->edx >> 3) % 2 == 0))
7     {
8         pcb[current].regs.eax = -1;
9         return;
10    }
11    // 寻找空闲文件描述符
12    for(i = 0; i < MAX_FILE_NUM; i++)
13    {
14        if(file[i].state == 0)
15        {
16            file[i].state = 1;
17            file[i].inodeOffset = destInodeOffset;
18            file[i].offset = 0;
19            file[i].flags = sf->edx;
20            pcb[current].regs.eax = MAX_DEV_NUM + i;
21            return;
22        }
23    }
24    // 没有找到空闲的文件描述符
25    pcb[current].regs.eax = -1;
26    return;
27 }
```

如果文件不存在(ret==1)，并且允许创建(O_CREATE 标志位为1)，则尝试创建文件。首先检查 O_CREATE 标志位，如果为0直接返回错误。接着，判断文件的类型，如果是常规文件((sf->edx >> 3) % 2 == 0)，先解析父目录路径，size 表示 str 中最后一个'/'的位置，如果 size == length，说明 str 中尾部的'/'后面无内容，路径无效，返回错误。如果 size == 0，说明文件路径是根目录，则将父目录设定为根目录，否则就将父目录路径设为当前文件路径。最后，读取父目录inode，设定创建文件的类型为常规文件，并分配新inode。如果文件类型是目录，首先检查路径末尾是否有'/'，如果有的话将末尾的'/'去除。后面的大致逻辑与常规文件相同，如果 str 的最后一个字符为'/'，路径无效，返回错误。初始化父目录路径为根目录，如果str不为根目录，就将父目录路径设为str。最后，读取父目录inode，设定创建文件的类型为目录，并分配新inode。

```
1 else { // try to create file
2     if ((sf->edx >> 2) % 2 == 0) {
3         //TODO: if O_CREATE not set
4         pcb[current].regs.eax = -1;
5         return;
6     }
7     // 文件类型是常规文件
8     if ((sf->edx >> 3) % 2 == 0) {
9         //TODO: if O_DIRECTORY not set
```

```
10     length = stringLen(str);
11     char fatherPath[NAME_LENGTH];
12     // 解析父目录路径
13     ret = stringChrR(str, '/', &size);
14     cond = size == length;
15     // 路径无效
16     if(cond)
17     {
18         pcb[current].regs.eax = -1;
19         return;
20     }
21     if(size == 0)
22     {
23         fatherPath[0] = '/';
24         fatherPath[1] = '\0';
25     }
26     // 复制父目录路径
27     else
28     {
29         stringCpy(str, fatherPath, size);
30     }
31     // 读取父目录inode
32     ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, fatherPath);
33     tmp = REGULAR_TYPE;
34     // 分配新inode
35     ret = allocInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
&destInodeOffset, str + size + 1, tmp);
36 }
37 // 文件类型是目录
38 else {
39     //TODO: if O_DIRECTORY set
40     length = stringLen(str);
41     char fatherPath[NAME_LENGTH] = "/";
42     if(str[length - 1] == '/')
43     {
44         str[length - 1] = '\0';
45         length--;
46     }
47     ret = stringChrR(str, '/', &size);
48     cond = size == length;
49     if(cond)
50     {
51         pcb[current].regs.eax = -1;
52         return;
53     }
54     // 不为根目录
55     if(size != 0)
56     {
57         stringCpy(str, fatherPath, size);
58     }
59     ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, fatherPath);
60     if(ret == -1)
61     {
```

```

62         pcb[current].regs.eax = -1;
63     }
64 }
65 tmp = DIRECTORY_TYPE;
66 ret = allocInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
67 &destInodeOffset, str + size + 1, tmp);
68 }
```

无论文件类型是常规文件还是目录，如果文件创建失败(`ret== -1`)，返回错误。然后，与之前相同，寻找空闲的文件描述符，如果找到了，则将其标记为使用中，记录 inode 位置，设置偏移量为0，保存打开的标志 `sf->edx`，完成了这些工作后返回文件描述符。如果最后没有找到空闲的文件描述符，则返回错误。

```

1 if (ret == -1) {
2     pcb[current].regs.eax = -1;
3     return;
4 }
5 for (i = 0; i < MAX_FILE_NUM; i++) {
6     if (file[i].state == 0) { // not in use
7         file[i].state = 1;
8         file[i].inodeOffset = destInodeOffset;
9         file[i].offset = 0;
10        file[i].flags = sf->edx;
11        pcb[current].regs.eax = MAX_DEV_NUM + i;
12        return;
13    }
14 }
15 pcb[current].regs.eax = -1; // create success but no available file[]
16 return;
```

(2) 完善 syscallWrite 函数

`syscallWrite` 函数主要包括设备写入和文件写入两部分。首先处理设备写入，判断文件描述符 `sf->ecx` 是否为 `STD_OUT`，如果是的话再判断设备是否处于就绪状态，若设备已就绪就调用标准输出函数 `syscallWriteStdOut`。然后处理文件写入，如果文件描述符指向文件而非设备($\text{MAX_DEV_NUM} \leq \text{sf}-\text{>} \text{ecx} < \text{MAX_DEV_NUM} + \text{MAX_FILE_NUM}$)，且文件已打开，调用处理文件写入的函数 `syscallWriteFile`。

```

1 void syscallWrite(struct StackFrame *sf) {
2     switch(sf->ecx) { // file descriptor
3         case STD_OUT:
4             if (dev[STD_OUT].state == 1)
5                 syscallWriteStdOut(sf);
6             break; // for STD_OUT
7         default:break;
8     }
9     // TODO: if refer to a file
10    if(sf->ecx >= MAX_DEV_NUM && sf->ecx < MAX_DEV_NUM + MAX_FILE_NUM)
11    {
12        // 文件已打开
13        if(file[sf->ecx - MAX_DEV_NUM].state == 1)
14        {
15            syscallWriteFile(sf);
```

```
16     }
17 }
18 return;
19 }
```

接下来的重点是实现函数 `syscallWriteFile`。首先验证目标文件是否以可写模式(O_WRITE标志)打开，若未设置则直接返回错误。接着，从用户栈帧中解析缓冲区地址和写入大小，并结合当前文件偏移量计算目标磁盘块的索引及块内偏移。随后，读取文件的Inode元数据以获取块分配信息，若写入位置超出当前文件块数则动态分配新磁盘块。在分块写入阶段，函数通过循环逐块处理数据将用户数据复制到内核缓冲区，处理跨块写入时自动切换块并重置缓冲区偏移，最终将修改后的块写回磁盘。写入过程中实时更新文件大小，最后更新文件偏移量并返回实际写入的字节数。

```
1 void syscallWriteFile(struct StackFrame *sf) {
2     if (file[sf->ecx - MAX_DEV_NUM].flags % 2 == 0) { // if O_WRITE is not set
3         pcb[current].regs.eax = -1;
4         return;
5     }
6
7     int i = 0;
8     int j = 0;
9     int ret = 0;
10    int baseAddr = (current + 1) * 0x100000; // base address of user process
11    uint8_t *str = (uint8_t*)sf->edx + baseAddr; // buffer of user process
12    int size = sf->ebx;
13    uint8_t buffer[SECTOR_SIZE * SECTORS_PER_BLOCK];
14    int quotient = file[sf->ecx - MAX_DEV_NUM].offset / sBlock.blocksize;
15    int remainder = file[sf->ecx - MAX_DEV_NUM].offset % sBlock.blocksize;
16
17    Inode inode;
18    diskRead(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
19
20    // TODO: Complete the process of write to file.
21    // 处理无效写入大小
22    if(size <= 0)
23    {
24        pcb[current].regs.eax = -1;
25        return;
26    }
27    // 已写入字节数
28    int used = 0;
29    // 当前块内起始偏移
30    int start = remainder;
31    // 缓冲区写入位置
32    j = remainder;
33    while(used < size)
34    {
35        // 若当前块未分配
36        if(quotient >= inode.blockCount)
37        {
38            ret = allocBlock(&sBlock, gDesc, &inode, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
39            // 分配失败 (
40            if(ret == -1)
```

```

41     {
42         if(file[sf->ecx - MAX_DEV_NUM].offset + used > inode.size)
43         {
44             inode.size = file[sf->ecx - MAX_DEV_NUM].offset + used;
45         }
46         size = used;
47         break;
48     }
49 }
// 读取当前块到缓冲区
readBlock(&sBlock, &inode, quotient, buffer);
// 当前块足够容纳剩余数据
if(sBlock.blockSize - start >= size - used)
{
    for(int k = 0; k < size - used; k++)
    {
        buffer[j++] = str[i++];
    }
    writeBlock(&sBlock, &inode, quotient, buffer);
    break;
}
// 当前块空间不足, 填充满后换块
else
{
    for(int k = 0; k < size - used; k++)
    {
        buffer[j++] = str[i++];
    }
    writeBlock(&sBlock, &inode, quotient, buffer);
    used += sBlock.blockSize - start;
    quotient++;
    j = 0;
    start = 0;
}
}
if (size > inode.size - file[sf->ecx - MAX_DEV_NUM].offset)
    inode.size = size + file[sf->ecx - MAX_DEV_NUM].offset;
diskwrite(&inode, sizeof(Iinode), 1, file[sf->ecx - MAX_DEV_NUM].inodeoffset);
pcb[current].regs.eax = size;
file[sf->ecx - MAX_DEV_NUM].offset += size;
return;
}

```

(3) 完善 syscallRead 函数

与 syscallWrite 函数相同, syscallRead 函数的核心逻辑包括设备读取和文件读取。首先处理设备读取, 判断文件描述符 sf->ecx 是否为 STD_IN, 如果是的话再判断设备是否处于就绪状态, 若设备已就绪就调用标准输入函数 syscallReadStdIn。然后处理文件读取, 如果文件描述符指向文件而非设备, 且文件已打开, 调用处理文件读取的函数 syscallReadFile。

```

1 void syscallRead(struct StackFrame *sf) {
2     switch(sf->ecx) { // file descriptor

```

```

3     case STD_IN:
4         if (dev[STD_IN].state == 1)
5             syscallReadStdIn(sf);
6         break; // for STD_IN
7     default:break;
8 }
9 // TODO: if refer to a file
10 if(sf->ecx >= MAX_DEV_NUM && sf->ecx < MAX_DEV_NUM + MAX_FILE_NUM)
11 {
12     // 文件已打开
13     if(file[sf->ecx - MAX_DEV_NUM].state == 1)
14     {
15         syscallReadFile(sf);
16     }
17 }
18 }

```

同样地，syscallRead 函数的实现关键在于 syscallReadFile 函数的实现。首先验证目标文件是否以可读模式(O_READ 标志)打开，若未设置则直接返回错误。接着，计算用户进程基地址以转换用户空间缓冲区地址，并通过当前文件偏移量确定目标磁盘块的索引quotient 和块内偏移remainder。随后读取文件的Inode元数据以获取块分配信息和文件总大小，若读取位置已超出文件末尾，则返回错误。在分块读取循环中，函数逐块将磁盘数据读入内核缓冲区，并根据剩余空间动态调整读取策略，当接近文件末尾时截断读取长度以避免越界，若当前块空间不足则自动切换到下一块并重置缓冲区偏移，最终将数据从缓冲区复制到用户内存。最终，读取完成后更新文件偏移量并返回实际读取的字节数。

```

1 void syscallReadFile(struct StackFrame *sf) {
2     if ((file[sf->ecx - MAX_DEV_NUM].flags >> 1) % 2 == 0) { // if O_READ is not set
3         pcb[current].regs.eax = -1;
4         return;
5     }
6
7     int i = 0;
8     int j = 0;
9     int baseAddr = (current + 1) * 0x100000; // base address of user process
10    uint8_t *str = (uint8_t*)sf->edx + baseAddr; // buffer of user process
11    int size = sf->ebx; // MAX_BUFFER_SIZE, don't need to reserve last byte
12    uint8_t buffer[SECTOR_SIZE * SECTORS_PER_BLOCK];
13    int quotient = file[sf->ecx - MAX_DEV_NUM].offset / sBlock.blocksize;
14    int remainder = file[sf->ecx - MAX_DEV_NUM].offset % sBlock.blocksize;
15
16    Inode inode;
17    diskRead(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
18
19    // TODO: Complete the process of read to a file.
20    // 已到文件末尾
21    if(file[sf->ecx - MAX_DEV_NUM].offset >= inode.size)
22    {
23        pcb[current].regs.eax = -1;
24        return;
25    }
26    int used = 0;
27    j = remainder;
28    int start = remainder;

```

```

29     while(used < size)
30     {
31         readBlock(&sBlock,&inode,quotient,buffer);
32         int rest = sBlock.blocksize - start;
33         // 处理文件末尾
34         if(rest + file[sf->ecx - MAX_DEV_NUM].offset >= inode.size)
35         {
36             for(int k = 0; k < inode.size - file[sf->ecx - MAX_DEV_NUM].offset; k++)
37             {
38                 str[i++] = buffer[j++];
39             }
40             size = inode.size - file[sf->ecx - MAX_DEV_NUM].offset;
41             break;
42         }
43         // 处理块内剩余空间
44         if(rest >= size - used)
45         {
46             for(int k = 0; k < size - used; k++)
47             {
48                 str[i++] = buffer[j++];
49             }
50             break;
51         }
52     else
53     {
54         for(int k = 0; k < rest; k++)
55         {
56             str[i++] = buffer[j++];
57         }
58         used += rest;
59         quotient++;
60         j = 0;
61         start = 0;
62     }
63 }
64
65 pcb[current].regs.eax = size;
66 file[sf->ecx - MAX_DEV_NUM].offset += size;
67 return;
68 }

```

(4) 完善 syscallLseek 函数

syscallLseek 函数需要补充的部分就是根据参数 whence(sf->ebx) 设置文件偏移量。若 whence 是 SEEK_SET，则将文件偏移量设置为 offset；若 whence 是 SEEK_CUR，则将文件偏移量设置为当前文件偏移量加上 offset；若 whence 是 SEEK_END，则将文件偏移量设置为文件长度加上 offset。最后，返回新的文件偏移量。

```

1 void syscallLseek(struct StackFrame *sf) {
2     int i = (int)sf->ecx; // file descriptor
3     int offset = (int)sf->edx;
4     Inode inode;

```

```

5
6     if (i < MAX_DEV_NUM || i >= MAX_DEV_NUM + MAX_FILE_NUM) {
7         pcb[current].regs.eax = -1;
8         return;
9     }
10    if (file[i - MAX_DEV_NUM].state == 0) {
11        pcb[current].regs.eax = -1;
12        return;
13    }
14
15    diskRead(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeoffset);
16
17    switch(sf->ebx) { // whence
18        case SEEK_SET:
19            // TODO: if SEEK_SET
20            file[i - MAX_DEV_NUM].offset = offset;
21            pcb[current].regs.eax = file[i - MAX_DEV_NUM].offset;
22            break;
23        case SEEK_CUR:
24            // TODO: if SEEK_CUR
25            file[i - MAX_DEV_NUM].offset += offset;
26            pcb[current].regs.eax = file[i - MAX_DEV_NUM].offset;
27            break;
28        case SEEK_END:
29            // TODO: if SEEK_END
30            file[i - MAX_DEV_NUM].offset = inode.size + offset;
31            pcb[current].regs.eax = file[i - MAX_DEV_NUM].offset;
32            break;
33        default:
34            break;
35    }
36}

```

(5) 完善 syscallClose 函数

该函数要补充的内容非常简单，如果文件描述符没有指向文件(sf->ecx< MAX_DEV_NUM或sf->ecx>= MAX_DEV_NUM + MAX_FILE_NUM)，或者文件没有打开(file[i - MAX_DEV_NUM].state == 0)，则返回-1，表示返回错误。

```

1 void syscallClose(struct StackFrame *sf) {
2     int i = (int)sf->ecx;
3     if (i < MAX_DEV_NUM || i >= MAX_DEV_NUM + MAX_FILE_NUM) {
4         // TODO: dev, can not be closed, or out of range
5         pcb[current].regs.eax = -1;
6         return;
7     }
8     if (file[i - MAX_DEV_NUM].state == 0) {
9         // TODO: not in use
10        pcb[current].regs.eax = -1;
11        return;
12    }
13    file[i - MAX_DEV_NUM].state = 0;

```

```

14     file[i - MAX_DEV_NUM].inodeOffset = 0;
15     file[i - MAX_DEV_NUM].offset = 0;
16     file[i - MAX_DEV_NUM].flags = 0;
17     pcb[current].regs.eax = 0;
18     return;
19 }

```

(6) 完善 syscallRemove 函数

syscallRemove 函数需要补充部分的是它的核心逻辑。首先，使用函数 readInode 尝试读取目标文件的inode，利用返回值ret判断目标文件是否存在。若目标文件不存在，直接返回错误。若目标文件存在，首先去除文件路径尾部的'/'。如果 size == length，说明 str 中尾部的'/'后面无内容，路径无效，返回错误。如果 size == 0，说明文件路径是根目录，则将父目录设定为根目录，否则就将父目录路径设为当前文件路径。然后尝试读取父目录inode，若读取失败，直接返回错误。接下来，根据文件类型释放inode。如果释放失败，返回-1；如果释放成功，返回0。

```

1 void syscallRemove(struct StackFrame *sf) {
2     int length = 0;
3     int cond = 0;
4     int ret = 0;
5     int size = 0;
6     int baseAddr = (current + 1) * 0x100000; // base address of user process
7     char *str = (char*)sf->ecx + baseAddr; // file path
8     Inode fatherInode;
9     Inode destInode;
10    int fatherInodeOffset = 0;
11    int destInodeOffset = 0;
12
13    ret = readInode(&sBlock, gDesc, &destInode, &destInodeOffset, str);
14
15    if (ret == 0) { // file exist
16        // TODO: You need to consider the situations that the file refer to a device or
17        // a file in use.
18        length = stringLen(str);
19        // 去除尾部 '/'
20        if(str[length - 1] == '/')
21        {
22            str[length - 1] = '\0';
23            length--;
24        }
25        ret = stringChrR(str, '/', &size);
26        cond = size == length;
27        // 路径无效
28        if(cond)
29        {
30            pcb[current].regs.eax = -1;
31            return;
32        }
33        char fatherPath[NAME_LENGTH] = "/";
34        if(size != 0)
35        {
36            stringCpy(str, fatherPath, size);
37        }
38    }
39 }

```

```

37     // 读取父目录inode
38     ret = readInode(&sBlock, gDesc, &fatherInode, &fatherInodeOffset, fatherPath);
39     if(ret == -1)
40     {
41         pcb[current].regs.eax = -1;
42         return;
43     }
44     // free inode
45     if (destInode.type == REGULAR_TYPE) {
46         // TODO: If REGULAR_TYPE
47         ret=freeInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
48         &destInodeOffset, str + size + 1, REGULAR_TYPE);
49     }
50     else if (destInode.type == DIRECTORY_TYPE) {
51         // TODO: If DIRECTORY_TYPE
52         ret=freeInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
53         &destInodeOffset, str + size + 1, DIRECTORY_TYPE);
54     }
55     if (ret == -1) {
56         pcb[current].regs.eax = -1;
57         return;
58     }
59     pcb[current].regs.eax = 0;
60     return;
61 }
62 else { // file not exist
63     pcb[current].regs.eax = -1;
64     return;
65 }

```

3. 完善函数 ls 和 cat

(1) 完善函数 ls

ls 函数要补充的部分的核心逻辑为遍历缓冲区中的所有目录项，跳过 inode=0 的无效项，并输出有效文件名。

```

1 int ls(char *destFilePath) {
2     printf("ls %s\n", destFilePath);
3     int i = 0;
4     int fd = 0;
5     int ret = 0;
6     DirEntry *dirEntry = 0;
7     uint8_t buffer[512 * 2];
8     fd = open(destFilePath, O_READ | O_DIRECTORY);
9     if (fd == -1)
10        return -1;
11     ret = read(fd, buffer, 512 * 2);
12     while (ret != 0) {
13         // TODO: Complete 'ls'.
14         i = 0;
15         while(i < ret)

```

```

16    {
17        // 定位到当前目录项
18        dirEntry = (DirEntry *) (buffer + i);
19        // 跳过未使用项
20        if (dirEntry->inode != 0) {
21            printf("%s ", dirEntry->name);
22        }
23        // 移动到下一个目录项
24        i += sizeof(DirEntry);
25    }
26    // 继续读取下一块
27    ret = read(fd, buffer, 512 * 2);
28    if (ret == -1)
29    {
30        break;
31    }
32    printf("\n");
33    close(fd);
34    return 0;
35}
36

```

(2) 完善函数 cat

cat 函数要补充的部分的核心逻辑为将缓冲区内容逐字节输出，代码如下：

```

1 int cat(char *destFilePath) {
2     printf("cat %s\n", destFilePath);
3     int fd = 0;
4     int ret = 0;
5     uint8_t buffer[512 * 2];
6     fd = open(destFilePath, O_READ);
7     if (fd == -1)
8         return -1;
9     ret = read(fd, buffer, 512 * 2);
10    while (ret != 0) {
11        // TODO: Complete 'cat'
12        // 将缓冲区内容逐字节输出
13        for (int i = 0; i < ret; i++)
14        {
15            printf("%c", buffer[i]);
16        }
17        // 继续读取下一块
18        ret = read(fd, buffer, 512 * 2);
19        if (ret == -1)
20        {
21            break;
22        }
23    }
24    close(fd);
25    return 0;
26}

```

三、思考题解答

1. 为什么使用文件描述符？如果直接使用文件名作为文件的标识，有什么缺陷吗？

文件描述符是操作系统用于高效管理进程打开文件的关键机制。它通过一个简短的整数标识文件，避免了每次操作都要解析长文件名的性能损耗，同时直接关联文件状态信息（如打开模式、偏移量），简化了资源管理。

若直接使用文件名作为标识，则存在三大缺陷：一是频繁解析长文件名会增加系统开销；二是文件名本身不包含文件状态信息，导致管理复杂；三是在多进程并发场景下，直接操作文件名可能引发访问冲突，而文件描述符提供了更细粒度的控制。

2. 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？

内核在处理 exec 时无需修改文件描述符表和系统文件打开表，核心原因在于 exec 的系统调用本质。exec 会替换当前进程映像为新程序，但会保留原进程的关键属性，如进程ID和文件描述符表。新程序直接继承原进程的文件描述符表，意味着已打开的文件（如标准输入/输出）及其状态（如偏移量）在新程序中保持有效。同时，系统文件打开表作为全局资源，记录了所有打开文件的信息，exec 不会改变其状态。因此，内核无需进行额外修改即可确保文件操作的连续性。

3. 如何使用重定向和管道来实现输出到缓冲区的system函数？

在子进程中关闭标准输出，创建指向缓冲区的文件描述符（如内存映射文件），或通过管道创建读写端。利用dup2系统调用将管道写端或缓冲区描述符复制为标准输出，随后执行目标命令。此时，命令输出会被重定向到缓冲区或管道。父进程则通过等待子进程结束，并从缓冲区或管道读端读取结果，从而完成输出捕获。

代码如下：

```
1 int capture_output(const char* cmd, char* buffer, size_t size) {
2     int pipefd[2];
3     // 创建管道
4     pipe(pipefd);
5     pid_t pid = fork();
6     // 子进程
7     if (pid == 0)
8     {
9         // 关闭读端
10    close(pipefd[0]);
11    // 重定向标准输出到管道
12    dup2(pipefd[1], STDOUT_FILENO);
13    close(pipefd[1]);
14    execl("/bin/sh", "sh", "-c", cmd, NULL);
15    exit(1);
16 }
17 // 父进程
18 else
19 {
20     // 关闭写端
21     close(pipefd[1]);
```

```
22     ssize_t bytes_read = 0;
23     size_t total = 0;
24     do {
25         bytes_read = read(pipefd[0], buffer + total, size - total);
26         if (bytes_read > 0) total += bytes_read;
27     } while (bytes_read > 0 && total < size);
28     close(pipefd[0]);
29     // 等待子进程结束
30     waitpid(pid, NULL, 0);
31     buffer[total] = '\0';
32     return total;
33 }
34 }
```

4. 为什么使用which命令找不到cd命令的程序所在的目录？

which 命令无法定位 cd 命令，是因为 cd 是 shell 内置命令，而非独立的可执行程序。which 通过搜索 PATH 环境变量指定的目录来查找可执行文件，但 cd 直接由 shell 解释执行，用于改变当前工作目录，其功能不依赖于外部可执行文件。因此，cd 不会出现在系统路径的任何目录中，导致 which 无法找到其“所在目录”。这一设计也避免了频繁创建子进程的开销，提升了 shell 的执行效率。