Lab5_challenge实验报告

19376273 陈厚伦

解决writef输出被打断的问题

> 实现思路

通过输出可以看出进程向控制台的输出不是原子的，因为时钟中断导致的进程中断会使两个进程的输出拼接在一起。

实现锁机制可以解决这个问题，将控制台设计成临界资源，writef代码设计成临界区，通过PV操作实现进程的互斥访问。

实验中，我主要实现了如下三个系统调用：`sys_p sys_V sys_createSemaphore`，分别实现P操作，V操作与创建一个信号量。

信号量是一个结构体，内核可以访问。

```
struct Semaphore {
    int id; // 信号量的id
    int value; // 信号量的值
    int countPointer; // 信号量进程等待队列的进程添加指针
    int freePointer; // 信号量进程等待队列的进程获取指针
    struct Env* queue[SEMAPHORE_ENV_MAXNUM]; // 信号量上的进程等待队列
};
```

在信号量上维护了一个属于该信号量的**环形等待队列**，存储被阻塞的进程，该队列通过一个头指针与为指针管理。

创建信号量的系统调用是`sys_createSemaphore`，创建一个id为dstId，信号量值为initValue的信号量。

```
int sys_createSemaphore(int sysno, int dstId, int initValue) {
        /* 如果该id的信号量已被申请则直接返回dstId */
        if (semaphoreList.occupy[dstId] == 1) {
                return dstId;
        }
        static int button = 0;
        /* 首次调用该函数时初始化 */
        if (button == 0) {
                int k;
                for (k = 0; k < SEMA_MAXNUM; k++) {
                        semaphoreList.occupy[k] = 0;
                }
                button += 1;
        }
        // 初始化信号量
        int retId = dstId;
        semaphoreList.list[retId].id = retId;
        semaphoreList.list[retId].value = initValue;
        semaphoreList.list[retId].countPointer = 0;
```

```
        semaphoreList.list[retId].freePointer = 0;
        semaphoreList.occupy[dstId] = 1;
        int i;
        // 初始化等待进程队列
        for (i = 0; i < SEMAPHORE_ENV_MAXNUM; i++) {
                semaphoreList.list[retId].queue[i] = NULL;
        }
        if (DEBUG) {
                printf("retId %d\n",retId);
                printf("value %d\n",semaphoreList.list[retId].value);
        }
        return retId;
}
```

P操作是`sys_P`，semaId是信号量的id，对信号量的值尝试自减，如果大于等于0则成功返回0。**如果小于0则要对调用P操作的进程阻塞**，经过如下步骤：

①将进程放入信号量的阻塞队列

②countPointer指针模加法自增

③设置进程状态为`ENV_NOT_RUNNABLE`

④调用`sys_yield()`进程切换

```
int sys_P(int sysno, int semaId) {
        struct Semaphore* semaphore = &(semaphoreList.list[semaId]);
        semaphore->value -= 1;
        if (DEBUG) {
                printf("\n---------------信号量值%d----------------------
\n",semaphore->value);
        }

        if (semaphore->value >= 0) {return 0;}
        if (semaphore->queue[semaphore->countPointer] != NULL) {
                panic("ENV TOO MANY!\n");
        }
        semaphore->queue[semaphore->countPointer] = curenv;
        semaphore->countPointer = (semaphore->countPointer + 1) %
SEMAPHORE_ENV_MAXNUM;
        curenv->env_status = ENV_NOT_RUNNABLE;
        if (DEBUG) {
                printf("\n---------------blocked!-------------\n");
        }
        sys_yield();
}
```

V操作是`sys_V`，semaId是信号量的id，对信号量的值自增，如果变成1成功返回，否则要唤醒一个阻塞的进程，具体步骤如下：

①从信号量阻塞队列中取一个进程设置状态为`ENV_RUNNABLE`

②进程阻塞队列freePointer进程置NULL，freePointer模加法自增

③调用`sys_yield()`进程切换。

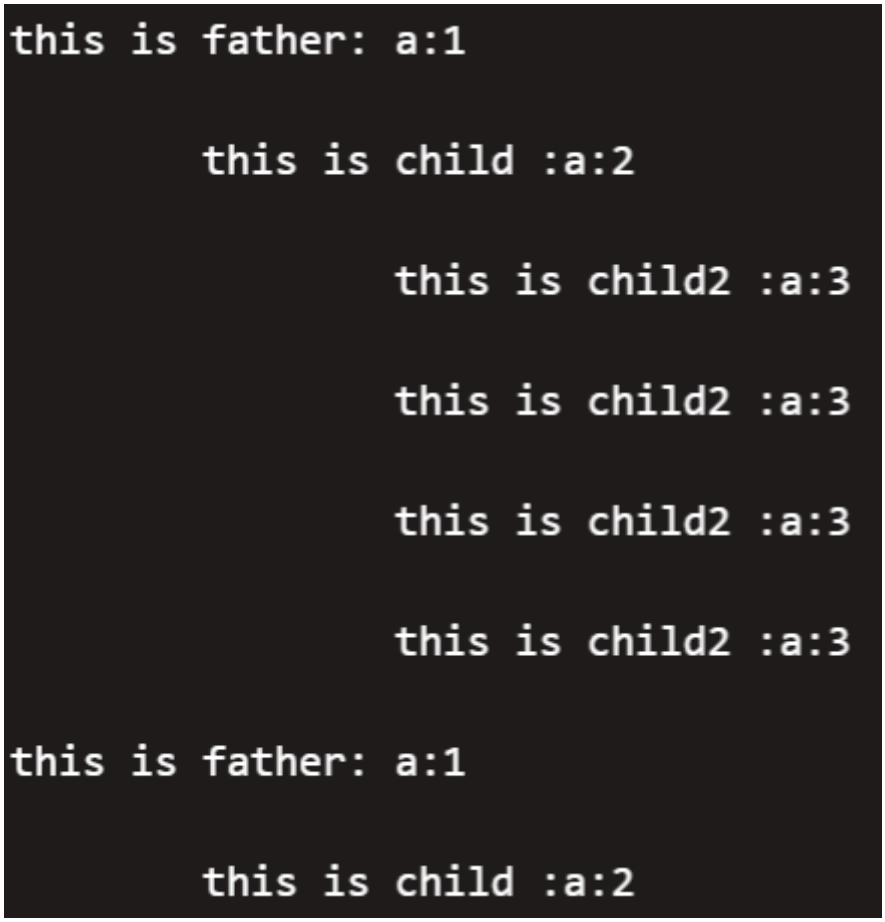
```
int sys_V(int sysno, int semaId) {
        if (DEBUG) {
                printf("\n+++++++++++++执行V操作+++++++++++++++\n");
        }
        struct Semaphore* semaphore = &(semaphoreList.list[semaId]);
        semaphore->value += 1;
        if (semaphore->value == 1) return 0;
        (semaphore->queue[semaphore->freePointer])->env_status = ENV_RUNNABLE;
        semaphore->queue[semaphore->freePointer] = NULL;
        semaphore->freePointer = (semaphore->freePointer + 1) %
SEMAPHORE_ENV_MAXNUM;
        sys_yield();
}
```

在writef中创建一个信号量，即可解决writef被打断的问题。

> 功能测试

```
void writef(char *fmt, ...)
{
        static int button = 0;
        static int writeMutex;
        // 创建并使用0号信号量
        if (button == 0) {
                writeMutex = syscall_createSemaphore(0,1);
                button += 1;
        }
        syscall_P(writeMutex);
        /* ___________临界区___________ */
        va_list ap;
        va_start(ap, fmt);
        user_lp_Print(user_myoutput, 0, fmt, ap);
        va_end(ap);
        /* ___________临界区_____________ */
        syscall_V(writeMutex);
}
```

测试发现进程切换出每个进程的输出都没有被打断。



IDE磁盘读取限制

> 实现思路

实现只有文件系统能够读写IDE磁盘，由于MOS中使用固定的进程号表示文件系统，所以一个简单的实现是在`sys_read_dev`和`sys_write_dev`中对进程ID进行判断即可。通过输出得到文件系统的id是4097.

```
if (curenv->env_id != 4097) {
    printf("only file system can read/write ide dev.\n");
    return -1;
}
```

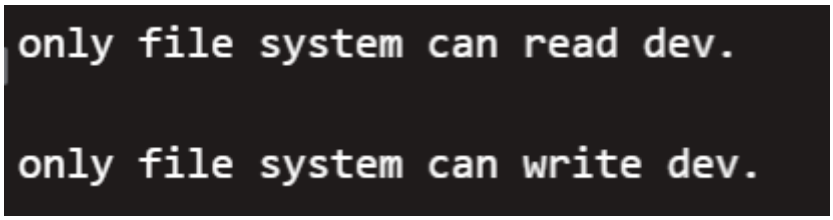
所有的系统都对访问内核地址进行了检查，所以文件系统进程也不会读写其他内核地址。

```
if (va >= ULIM) {
    return -E_INVAL;
}
```

> 功能测试

```
// 用户进程
syscall_read_dev(0x50000000,0x13000000,4);
syscall_write_dev(0x50000000,0x13000000,4);
```

测试结果如下：



虚拟设备机制

- **设备分配控制**

> 实现思路

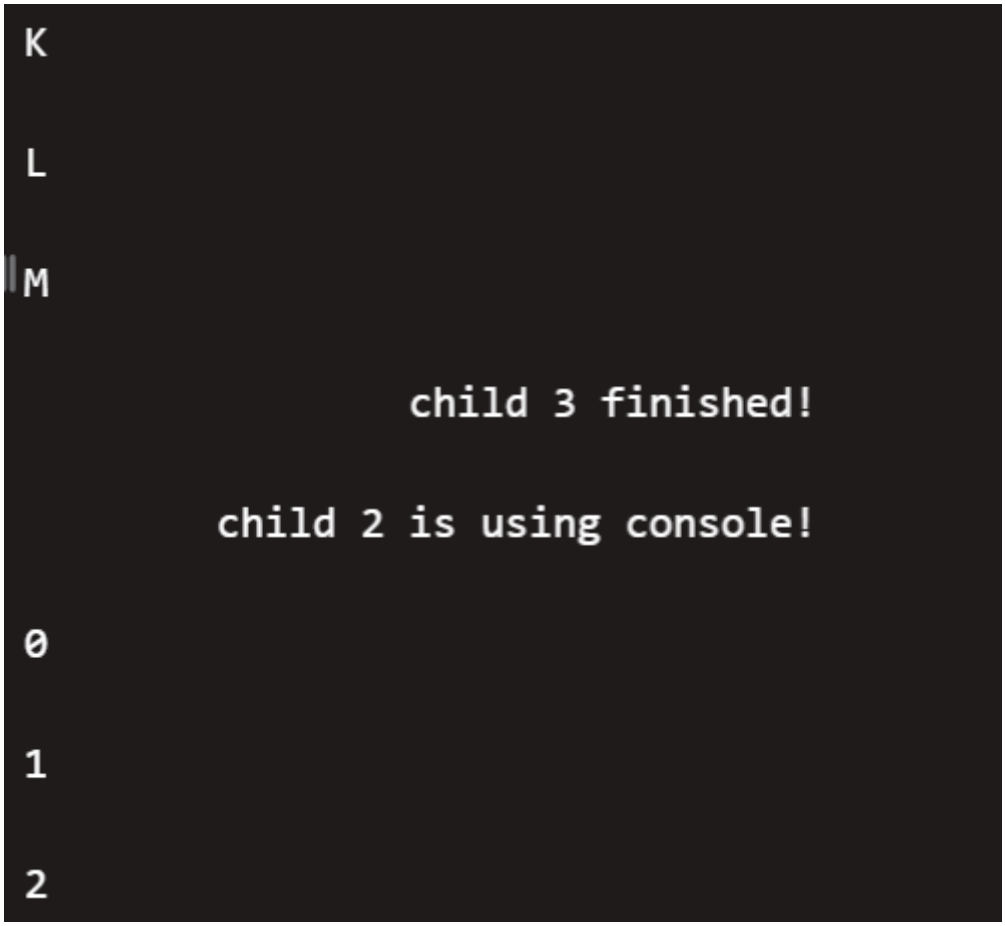
上述的PV机制可以解决这个问题，只需要扩大临界区的范围即可，保证进程使用设备的多次操作都不会被打断。

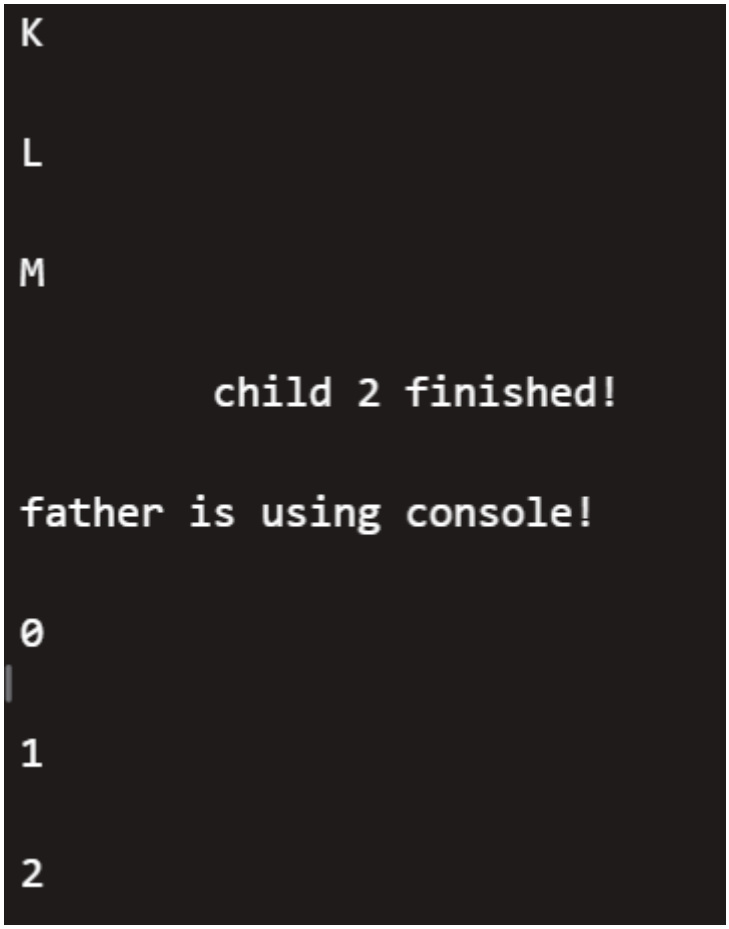
> 功能测试

以`console`设备的输出为例，进行测试。

```
for (;;) {
    syscall_P(writeMutex);
    // 每个进程连续使用控制台设备进行输出
    char* buf = "\t\tchild 3 is using console!\n";
    write(fdnum,buf,100);
    char a[2]={'\n','\n'};
    int i;
    // 连续输出从'0'开始的30个字符
    for (i=0;i<30;i++) {
        a[0] = i + '0';
        write(fdnum,a,2);
    }
    char* buf2 = "\t\tchild 3 finished!\n";
    write(fdnum,buf2,100);
    syscall_V(writeMutex);
}
```

测试结果看到，每个进程都完整的输出了30个字符才会轮到下一个进程输出，实现了多个进程轮流互斥使用控制台资源。





- 模拟**SPOOLing**技术

`SPOOLing`技术的核心是**脱机处理输入输出**，实现专门的输入进程与输出进程，IO进程与用户进程并发运行，并且向用户提供一个统一的输入输出接口。用户进程将输入输出请求发给输入输出进程，当有请求时，输入输出进程会执行。实验中，采取了**通过内核**来实现请求的传递与接收，在内核维护了一个输入请求队列与输出请求队列。

> 实现思路

以控制台设备模拟SPOOLing技术，分别实现**控制台打印文件**和**控制台读入**的虚拟设备共享。用户进程通过系统调用发出输入输出申请，输入进程与输出进程通过系统调用获取申请，并且执行输入输出，实现“假脱机”。

实现了输入和输出请求结构体：

```
struct WriteReq {
    int envid;  // 申请输出的进程号
    int fdnum;  // 申请输出的文件
    int length; // 申请输出的长度
    int valid;  // 该申请是否有效
};

struct ReadReq {
    int envid;  // 申请输出的进程号
    char* buf;  // 申请输入的目标地址
    int length; // 申请输入的长度
    int valid;  // 该申请是否有效
}
```

实现了四个系统调用：`sys_writeSpooling sys_getWriteReq sys_readSpooling sys_getReadReq`，分别实现发送输出请求，获取输出请求，发送输入请求，获取输入请求。

```
void
sys_writeSpooling(int sysno, int fdnum, u_int n) {
        static int button = 0;
        /* 初始化输出请求队列 */
        if (button == 0) {
                int i;
                for (i = 0; i < REQ_NUM; i++) {
                        writeReqs[i].valid = 0;
                }
                pushPointer = 0;
                getPointer = 0;
                button++;
        }
        if (writeReqs[pushPointer].valid == 1) {
                panic("SPOOLing的write请求池满了\n");
        }
        /* 添加一个输出请求 */
        writeReqs[pushPointer].envid = curenv->env_id;
        writeReqs[pushPointer].fdnum = fdnum;
        writeReqs[pushPointer].length = n;
        writeReqs[pushPointer].valid = 1;
        pushPointer = (pushPointer+1) % REQ_NUM;
}

void
```

```
sys_getWriteReq(int sysno,struct WriteReq* preq) {
        if (writeReqs[getPointer].valid != 1) {
                /* 如果输出请求池没有请求，则切换进程 */
                preq->valid = 0;
                sys_yield();
        }
        else {
                /* 获取一个输出请求 */
                preq->envid = writeReqs[getPointer].envid;
                preq->fdnum = writeReqs[getPointer].fdnum;
                preq->length = writeReqs[getPointer].length;
                preq->valid = 1;
                writeReqs[getPointer].valid = 0;
                getPointer = (getPointer+1) % REQ_NUM;
        }
}

void
sys_readSpooling(int sysno,char* buf,u_int n) {
        static int button = 0;
        if (button == 0) {
                int i;
                for (i=0;i<REQ_NUM;i++) {
                        readReqs[i].valid = 0;
                }
                readPushPointer = 0;
                readGetPointer = 0;
                button++;
        }
        if (readReqs[readPushPointer].valid==1) {
                panic("SPOOLing的read请求池满了\n");
        }
        readReqs[readPushPointer].envid = curenv->env_id;
        readReqs[readPushPointer].buf = buf;
        readReqs[readPushPointer].length = n;
        readReqs[readPushPointer].valid = 1;
        readPushPointer = (readPushPointer+1) % REQ_NUM;
}

void
sys_getReadReq(int sysno,struct ReadReq* preq) {
        if (readReqs[readGetPointer].valid != 1) {
                preq->valid = 0;
                sys_yield();
        }
        else {
                preq->envid = readReqs[readGetPointer].envid;
                preq->buf = readReqs[readGetPointer].buf;
                preq->length = readReqs[readGetPointer].length;
                preq->valid = 1;
                readReqs[readGetPointer].valid = 0;
                readGetPointer = (readGetPointer+1) % REQ_NUM;
        }
}
```

实现了输入与输出的用户接口：

```
/* 输出接口 */
void
writeSpooling(int fdnum, u_int n) {
    syscall_writeSpooling(fdnum,n);
}
/* 输入接口 */
void
readSpooling(char* buf, int length) {
        syscall_readSpooling(buf,length);
}
```

实现了输入进程与输出进程处理输入输出的接口：

```
/* 进程处理输出接口 */
void
execWriteReq() {
        struct WriteReq req;
        // 获取输出请求
        syscall_getWriteReq(&req);
        if (req.valid == 0) {
                // 获取输出请求无效则返回
                return;
        }
        int envid = req.envid;
        int fdnum = req.fdnum;
        int length = req.length;

        writef("SERVE: ENV %d .Output file %d to console!\n",envid,fdnum);

        char buf[1024] = {0};
        // 获取待输出内容
        seek(fdnum,0);
        read(fdnum,buf,length);
        writef("fdnum:%d length:%d buflen:%d\n",fdnum,length,strlen(buf));
        int consId;
        // 打开一个控制台设备并输出
        consId = opencons();
        write(consId,buf,length);
        close(consId);
}
/* 进程处理输入接口 */
void
execReadReq() {
        struct ReadReq req;
        // 获取一个输入请求
        syscall_getReadReq(&req);
        if (req.valid == 0) {
                // 获取输入请求无效直接返回
                return;
        }
        int envid = req.envid;
        char* buf = req.buf;
        int length = req.length;

        writef("\nSERVE ENV %d. Input from console!\n",envid);
        writef("read chars nums: %d\n",length);
```

```
        int consId;
        consId = opencons();
        // 从控制台读
        read(consId,buf,length);
        close(consId);
}
```

官方代码中`cons_read`只能实现单个字符的输入，实现的过程中对`cons_read`函数进行了改写。

上述只是一个对SPOOLing原理的模拟，扩展性不是很强。接下来完善输入输出井，将输入与输出的内容专门放入磁盘固定位置，可以对输入输出设备进行泛化。

> 功能测试

该部分的测试代码如下：

```
void umain()
{
        int a = 0;
        int id = 0;
        int fdnum;
        struct Fd* fd;
        writef("开始测试\n");
        fdnum = opencons();
        fd = num2fd(fdnum);
        int writeMutex = syscall_createSemaphore(1,1);
        // 创建两个文件用于打印
        int fdnum1 = open("/print1",O_APPEND|O_CREAT|O_RDWR);
        int fdnum2 = open("/print2",O_APPEND|O_CREAT|O_RDWR);
        char* buf1 = "this is in file print1.\n";
        int i = 0;
        // 向两个文件写不同的内容
        for (i = 0; i < 5; i++) {
                write(fdnum1,buf1,strlen(buf1));
        }
        char* buf2 = "this is in file print2.\n";
        for (i = 0; i < 3; i++) {
                write(fdnum2,buf2,strlen(buf2));
        }
        if ((id = fork()) == 0) {
                if ((id = fork()) == 0) {
                        a += 3;
                        int k=0;
                        for (;;) {
                                if (k > 5) continue;
                                k++;
                                writeSpooling(fdnum1,128);
//                              writef("提交输入请求");
//                              char buf[128]={0};
//                              readSpooling(buf,1);
                                // 切换进程
                                syscall_yield();
                        }
                }
                a += 2;
                int k =0;
                for (;;) {
```

```
                        if (k > 10) continue;
                        k++;
                        writeSpooling(fdnum2,128);
//                      char buf[128]={0};
//                      readSpooling(buf,1);
                        syscall_yield();
                }
        }
        a++;
        for (;;) {
                // 父进程充当输出/输入进程
                execWriteReq();
//              execReadReq();
        }
}
```

向控制台打印文件

测试结果如下，可以看出输出进程按照输出请求的提交顺序，依次服务。

```
                SERVE: ENV 8195 .Output file 1 to console!

this is in file print1.

this is in file print1.

this is in file print1.

this is in file print1.

this is in file print1.

                SERVE: ENV 6146 .Output file 2 to console!

this is in file print2.

this is in file print2.

this is in file print2.

                SERVE: ENV 8195 .Output file 1 to console!

this is in file print1.
```

控制台读入

测试结果如下，可以看到输入进程按照用户进程发来的请求的顺序，依次为用户进程服务。

```
SERVE ENV 6146. Input from console!

read chars nums: 1

r

SERVE ENV 8195. Input from console!

read chars nums: 1

k

SERVE ENV 6146. Input from console!

read chars nums: 1

d

SERVE ENV 8195. Input from console!

read chars nums: 1
```

完善文件系统功能

> 实现思路

- `getcwd(char* buf, int size)`

getcwd获取当前进程的工作目录，为此在进程结构体`struct Env`中添加了`char cwd[MAxPATHLEN]`存储进程的当前工作目录地址，创建进程的时候初始化为`"/"`，`getcwd`函数通过一个系统调用`syscall_getcwd`获取当前进程的cwd，如果超过了size，返回NULL，反之返回buf的地址。

- `chdir(const char* path)`

该函数可以接收绝对路径，也可以接收相对路径。这个函数要对路径的合法性进行检查，这个路径要合法并且要引用的是目录。

实现上类似于一个字符串处理的函数，如果是绝对路径直接使用，如果是相对路径要先通过getcwd获取当前工作目录并拼接相对路径，调用`open(path,O_DIRECTORY)`函数，检查路径合法性并是否是目录。

```
int
chdir(const char* path) {
        // 如果path是绝对路径直接替换即可
        int r;
        char* p = path;
        struct Fd* fd;
        if (p[0] == '/') {
                // 路径应当是一个合法的目录的路径
                r = open(path,O_DIRECTORY | O_RDONLY);
                if (r < 0) {
                        return -1;
                }
                fd = num2fd(r);
                r = file_close(fd);
                syscall_chdir(path);
                return 0;
        }
        else {
                // 相对路径 拼接
                char temp[1024] = {0};
                getcwd(temp,1024);
                int len;
                len = strlen(temp);
                if (temp[len-1] != '/') {
                        temp[len] = '/';
                        len++;
                }
                strcpy(temp+len,path);
                r = open(temp,O_DIRECTORY|O_RDWR);
                if (r < 0) {
                        return -1;
                }
                fd = num2fd(r);
                r = file_close(fd);
                syscall_chdir(temp);
                return 0;
        }
}
```

实现上对于相对路径支持了`.`与`..`，对fs.c中的`walk_path`进行了修改以支持该实现。

```
if (strcmp(name,".") == 0) {
    // do nothing
}
else if (strcmp(name,"..") == 0) {
    if (strcmp(dir->f_name,"/") == 0) {
        // 如果已经是根目录 do nothing
    }
    else {
        // 回退到上一层目录
        file = dir->f_dir;
    }
}
else if ((r == dir_lookup(dir,name,&file)) < 0) {
    // ...
}
```

- `open(const char *path, int mode)`

对open的mode指定了三个参数：`O_CREAT,O_APPEND,O_DIRECTORY`，分别在serve_open中加以修改即可。对`O_CREAT`，如果`file_open`则执行`file_create`；对`O_APPEND`，在调整文件流指针为`f_size`，对`O_DIRECTORY`，则需要对打开的文件类型检查是否是目录。

- `openat(int fd, const char* path, int mode)`

如果是绝对路径(`path[0] == '/'`)，直接调用open。如果是AT_FDCWD，则getcwd（）获取当前进程工作目录并拼接相对路径，如果是fd，则获取fd对应打开的文件的目录并拼接。这里对`struct Fd`进行了修改，添加了`char path[MAXPATHLEN]`，记录被打开文件的路径（确切地说是所用的链接的路径，这一点在实现了link之后会更加明确）。

- `link(const char* existingpath, const char* newpath)`

MOS系统的FCB结构对于link的实现可以说非常不友好，**将FCB结构改装成inode结构**，**目录文件存储目录项**，**目录项**记录链接名称，链接类型，上级目录和其inode的地址；**inode**存放文件的大小，文件的索引结构，文件的链接次数。需要对MOS进行非常大的重构。

- `symlink(const char* actualpath, const char* sympath)`

符号链接实质上是一个文件的内容是一个链接，所以可以先创建一个文件，并设置文件类型为`FTYPE_SYM`，将`actualpath`写入该文件。支持符号链接的操作会先判断文件类型是否为符号链接，如果是读取其内容并对其链接的地址再次执行操作。

- `mkdir(const char* path, u_int mode)`

通过文件系统的ipc机制，创建一个文件并且设置为目录，并设置mode，在File结构体中添加了mode项，对目录文件，支持读、写、执行三种权限。

- `mkdirat(int fd, const char* path, u_int mode)`

实现过程与`openat`类似，通过地址拼接得到绝对路径，调用mkdir.

> 功能测试

该部分的测试代码如下：

对`open`的mode参数的测试：

```
#include "lib.h"

static char *msg = "This is the NEW message of the day!\n";
static char *diff_msg = "This is a different massage of the day!\n";

void umain() {
        int r;
        int fdnum;
        char buf[512] = {0};
```

```
        int n;

        r = open("/test_o_creat",O_RDWR);
        if (r < 0) {
                if (r == -9) {
                        writef("文件不存在\n");
                }
        }

        r = open("/test_o_creat",O_CREAT | O_RDWR);
        if (r < 0) {
                writef("%d\n",r);
                user_panic("o_creat failed!\n");
        }
        fdnum = r;
        writef("o_creat open is good\n");
        r = write(fdnum, msg, strlen(msg));
        //writef("write ok\n");
        seek(fdnum,0);
        n = read(fdnum,buf,511);
        if (n < 0) {user_panic("read failed.\n");}
        writef("%s\n",buf);

        r = close(fdnum);
        r = open("/test_o_creat",O_APPEND | O_CREAT | O_RDWR);
        fdnum = r;
        r = write(fdnum,msg,strlen(msg));
        // 检查追加
        r = write(fdnum,msg,strlen(msg));

        seek(fdnum,0);
        n = read(fdnum,buf,511);
        if (n < 0) {user_panic("read failed.\n");}
        writef("----------如果正确,应该有三句------------\n");
        writef("%s\n",buf);
        close(fdnum);
        // 检查O_DIRECTORY
        r = open("/test_o_creat",O_DIRECTORY | O_RDWR);
        if (r < 0) {
                writef("pass o_directory check.\n");
        }
}
```

测试结果：

```
文件不存在
This is the NEW message of the day!
此时的文件流指针位置是36                // 原句36个字符，所以文件流指针指向36是正确的
----------如果正确,应该有三句-----------
This is the NEW message of the day!
This is the NEW message of the day!
This is the NEW message of the day!
O_DIRECTORY模式下打开的文件不是目录文件!
pass o_directory check.
```

对**其余的操作**的测试代码：

```c
#include "lib.h"
#include "fd.h"

void
umain() {
        writef("-----------------myDemo.c------------------\n");
        /* ********************** */
        user_create("/file1",0);
        user_create("/file2",0);
        user_create("/dir1",1);
        user_create("/dir2",1);
        user_create("/dir1/file3",0);
        user_create("/dir1/dir3",1);
        user_create("/dir1/dir3/file4",0);
        user_create("/dir1/dir3/dir4",1);
        user_create("/dir1/dir3/dir4/file5",0);
        user_create("/dir1/dir3/dir4/dir5",1);
        user_create("/dir1/dir3/dir4/dir5/file6",0);
        int fdnum = open("/dir1/dir3/../dir3/file4",O_RDWR);
        struct Fd* fd = num2fd(fdnum);
        struct Filefd* filefd = (struct Filefd*) fd;
        struct File* file = &(filefd->f_file);
        writef("\t\t\tfilename %s\n",file->f_name);
        char buf[1024] = {0};
        writef("\t\t\taddr is 0x%x\n",file->f_dir);
        getcwd(buf,1024);
        writef("\t\t\tpath is %s\n",buf);
        chdir("/dir1/dir3");
        getcwd(buf,1024);
        writef("\t\t\tpath is %s\n",buf);
        chdir("dir4");
        getcwd(buf,1024);
        writef("\t\t\tpath is %s\n",buf);
        chdir("/");
        getcwd(buf,1024);
        writef("\t\t\tpath is %s\n",buf);
        writef("-----------------------------------\n");
        chdir("dir1");
        getcwd(buf,1024);
        writef("\t\t\tpath is %s\n",buf);
        chdir("./dir3/../dir3/dir4");
        getcwd(buf,1024);
        writef("\t\t\t支持相对路径:path is %s\n",buf);
        close(fdnum);
        ////////////////////////////////////////////
        fdnum = openat(AT_FDCWD,"dir5",O_RDWR);
        fd = num2fd(fdnum);
        filefd = (struct Filefd*) fd;
        file = &(filefd->f_file);
        writef("\t\t\tfilename %s\n",file->f_name);

        int fdnum2 = openat(fdnum,"file6",O_RDWR);
        fd = num2fd(fdnum2);
        filefd = (struct Filefd*) fd;
        file = &(filefd->f_file);
        writef("\t\t\tfilename %s\n",file->f_name);
        close(fdnum);
```

```
        symlink("/dir1/file3","/symlink1");
        char tmpbuf[10000];
        fdnum = open("/dir1",O_RDONLY);
        read(fdnum,tmpbuf,10000);

        int t;
        struct Stat stat1;
        fstat(fdnum,&stat1);
        writef("\t\t\tdir1的大小:%d\n",stat1.st_size);
        writef("\t\t\tdir1 内容:");
        for (t=0;t<10000;t++) {
                writef("%c",tmpbuf[t]);
        }
        writef("\n");
        close(fdnum);

        stat("/dir1/dir3/dir4/dir5",&stat1);
        writef("\t\t\t删除file6前,dir5的大小:%d\n",stat1.st_size);
        remove("/dir1/dir3/dir4/dir5/file6");
        stat("/dir1/dir3/dir4/dir5",&stat1);
        writef("\t\t\t删除file6后,dir5的大小:%d\n",stat1.st_size);
        mkdir("/d1",O_EXEC);
        stat("/d1",&stat1);
        writef("\t\t\t/d1的名字是%s\n",stat1.st_name);
        writef("\t\t\t/d1的大小是%d\n",stat1.st_size);
        mkdir("/d1/d2",O_EXEC);
        remove("/d1/d2");
        stat("/d1",&stat1);
        writef("\t\t\t/d1的大小是%d\n",stat1.st_size);
        int r = open("/d1/d2",O_RDWR);
        writef("r = %d\n",r);
}
```

测试结果如下所示:

```
                        filename file4
                        addr is 0x10007100
                        path is /
                        path is /dir1/dir3
                        path is /dir1/dir3/dir4
                        path is /
-----------------------------------
                        path is /dir1
                        支持相对路径:path is /dir1/./dir3/../dir3/dir4
openat 路径 /dir1/./dir3/../dir3/dir4/dir5
                        filename dir5
openat(fd模式) 路径 /dir1/./dir3/../dir3/dir4/dir5/file6
                        filename file6
                        dir1的大小:4096
                        dir1 内容:file34dir4
                        删除file6前,dir5的大小:4096
                        删除file6后,dir5的大小:4096
                        /d1的名字是d1
                        /d1的大小是0
                        /d1的大小是4096
r = -9
```

遇到的问题及解决方案

对整个MOS系统不熟悉，之前的几个lab的基础并不牢固，所以实现的过程中遇到了许多问题。

① 对新添加的c文件h文件无法编译，更新后无法重新编译。通过改写Makefile文件解决，仿照已有的可以正常编译的文件，对新文件进行仿写。

② make之后ipc机制失效，发现要make clean之后在重新make才能够恢复。理论上所有被改动的文件make的时候都会重新编译，但是出现这个问题原因始终很模糊。

③ fsformat.c文件出错只报段错误，没有具体的报错信息。fsformat.c文件使用较少，对磁盘的贴近硬件部分了解不够深入，根据经验段错误大概率是空指针等存储空间的问题，所以仔细检查了访问存储空间的语句来定位bug。

④ 结构设计很重要。前面的设计仅仅满足了需求，后面的实现的时候会发现扩展性非常差，解决这个问题的好办法是重构，要充分把握了代码的基础之上再重构。

实验感想

从修改一个Makefile到进程的创建与切换，再改写文件系统核心代码，挑战性任务把lab0-lab5完整的串联了起来，非常考验对之前的lab的熟悉程度与整体把握程度。

挑战性没有具体的指导材料，对于lab5的代码没有完全熟悉，导致在实现的时候遇到了很大的困难。但是通过不断的尝试、试错，加深了对官方代码的理解程度，也将代码进行修改以适应新功能的扩展，锻炼了创新能力。