Lab6实验报告

思考题

T1

示例代码中，父进程操作管道的写端，子进程操作管道的读端。如果现在想让父进程作为“读者”，代码应当如何修改?

调换一下`fork()`函数后父子进程的执行代码即可。

```
int main(){
    ...
    status = pipe(fildes);
    ...
    switch (fork()) {
        case -1: /* Handle error */
            break;
        case 0: /* Child - writes to pipe */
            close(fildes[0]); //关闭读端
            write(fildes[1], "hello world\n", 12);//将12个字节的buf写入fileds[1]
            close(fildes[1]);
            exit(EXIT_SUCCESS);
        default: /* Parent - reads from pipe */
            close(fildes[1]);//关闭写端
            read(fildes[0], buf, 100);//从fildes[0]中读入100个字节到buf中
            printf("father-process read:%s",buf);
            close(fildes[0]);
            exit(EXIT_SUCCESS);
    }
}
```

T2

上面这种不同步修改 pp_ref 而导致的进程竞争问题在 user/fd.c 中的 dup 函数中也存在。请结合代码模仿上述情景，分析一下我们的 dup 函数中为什么 会出现预想之外的情况?

`int dup(int oldfdnum, int newfdnum)`函数作用为将编号为`oldfdnum`的文件描述符完全复制给`newfdnum`文件描述符。

如果执行`dup(fd[0], newfd)`，将管道的读端复制给fdnew，由于先执行将fd[0]所在的页面映射给`newfd`，若`pageref[fd[0]]`之前等于1，则`pageref(fd[0])`就加1，等于2。而如果此时在下方代码中标有注释的位置发生时钟中断，由于pipe所在的页面还未映射，如果此时`pageref(pipe)`为2，那么就有`pageref(fd[0]) == pageref(pipe)`，造成写端关闭的假象。

```
int dup(int oldfdnum, int newfdnum){
    ...
    if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
```

```
                                ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R | PTE_LIBRARY))) <
0) {
        goto err;
    }
    //这里发生时钟中断
    if ((* vpd)[PDX(ova)]) {
        for (i = 0; i < PDMAP; i += BY2PG) {
            pte = (* vpt)[VPN(ova + i)];

            if (pte & PTE_V) {
                // should be no error here -- pd is already allocated
                if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
                                         pte & (PTE_V | PTE_R | PTE_LIBRARY))) < 0) {
                    goto err;
                }
            }
        }
    }
    ...
}
```

T3

阅读上述材料并思考：为什么系统调用一定是原子操作呢？如果你觉 得不是所有的系统调用都是原子操作，请给出反例。希望能结合相关代码进行分析。

系统调用一定是原子操作，内核处理系统调用相关指令时，在函数`handle_sys`中会执行宏`CLI`，关闭中断，保证不会被时钟中断打断。

T4

仔细阅读上面这段话，并思考下列问题

- **按照上述说法控制 pipeclose 中 fd 和 pipe unmap 的顺序，是否可以解决上 述场景的进程竞争问题？给出你的分析过程。**

可以，`fork`函数执行后，pageref(fd[0]) == pageref(fd[1]) == 2， pageref(pipe) == 4,当关闭fd时，若先解除fd的映射，即使发生时钟中断，那么pageref(fd)是一定小于 pipe的，不可能出现pageref(fd) == pageref(pipe)的情况

- **我们只分析了 close 时的情形，在 fd.c 中有一个 dup 函数，用于复制文件内容。试想，如果要复制的是一个管道，那么是否会出现与 close 类似的问题？ 请模仿上述材料写写你的理解。**

会，详见**Thinking 2**

T5

bss 在 ELF 中并不占空间，但 ELF 加载进内存后，bss 段的数据占 据了空间，并且初始值都是 0。请回答你设计的函数是如何实现上面这点的?

为bss段分配物理页，之后映射到对应的虚拟地址上，然后将该页数据清零。具体调用了函数`page_alloc(&p)`，`page_insert(env->env_pgdir, p, va, PTE_R)`，`bzero(page2kva(p), size)`。

T6

为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值?

`user/user.lds`文件下规定了.text段的偏移值，为0x00400000

```
    . = 0x00400000;
  _text = .;            /* Text and read-only data */
```

T7

在 shell 中执行的命令分为内置命令和外部命令。在执行内置命令时 shell 不需要 fork 一个子 shell，如 Linux 系统中的 cd 指令。在执行外部命令时 shell 需要 fork 一个子 shell，然后子 shell 去执行这条命令。 据此判断，在 MOS 中我们用到的 shell 命令是内置命令还是外部命令？请思考为什么 Linux 的 cd 指令是内部指令而不是外部指令?

外部命令，因为我们是通过一个可执行文件执行了子进程。Shell的内置命令，就是 Shell 自带的命令，这些命令是没有执行文件的，内置命令在解析时直接根据shell内部代码执行。

T8

在哪步，0 和 1 被 “安排” 为标准输入和标准输出？请分析代码执行流程，给出答案

首先，`icode`进程执行`spawnl("init.b", "init", "initarg1", "initarg2", (char*)0))`，执行进程`init.b`， `init.b`进程通过 spawn 生成 shell 进程。

在`user/init.c`的`umain`函数中，有

```
    close(0);//关闭0，使得opencons打开的文件描述符编号为0
    if ((r = opencons()) < 0)
        user_panic("opencons: %e", r);
    if (r != 0)
        user_panic("first opencons used fd %d", r);
    if ((r = dup(0, 1)) < 0)
        user_panic("dup: %d", r);
```

将0和1安排为标准输入和标准输出

T9

在你的 shell 中输入指令 ls.b | cat.b > motd。

- **请问你可以在你的 shell 中观察到几次 spawn？分别对应哪个进程?**

两个， 分别为ls.b和cat.b

- **请问你可以在你的 shell 中观察到几次进程销毁？分别对应哪个进程?**

两次，为cat.b和ls.b

```
$ ls.b | cat.b > motd

[00001c03] pipecreate

```

```
[00001c03] SPAWN: ls.b

serve_open 00001c03 ffff000 0x0

serve_open 00002404 ffff000 0x101

[00002404] SPAWN: cat.b

serve_open 00002404 ffff000 0x0

serve_open 00003406 ffff000 0x0

serve_open 00003406 ffff000 0x0

[00003406] destroying 00003406

[00003406] free env 00003406

i am killed ...

[00002c05] destroying 00002c05

[00002c05] free env 00002c05

i am killed ...
```

实验学习

管道

管道又叫做匿名管道，只能用在具有公共祖先的进程之间使用，通常使用在父子进程之间通信，创建过程如下：

```
int pipe(int fd[2]);//成功返回0， 否则-1
//参数fd返回两个文件描述块编号，fd[0]对应读端，fd[1]对应写端
```

> 管道是一种只在内存中的文件。在 UNIX 中使用 pipe 系统调用时，进 程中会打开两个新的文件描述符：一个只读端和一个只写端，而这两个文件描述符都映射到了同一片内存区域。但这样建立的管道的两端都在同一进程中，而且构建出的管道两端是两个匿名的文件描述符，这就让其他进程无法连接该管道。在 fork 的配合下，才 能在父子进程间建立起进程间通信管道，这也是匿名管道只能在具有亲缘关系的进程间通信的原因。

创建管道

pipe函数：

返回两个文件描述符编号，两个文件描述符都对应着相同的物理内存，存有struct Pipe结构体。

```

int pipe(int pfd[2])
{
    int r, va;
    struct Fd *fd0, *fd1;
```

```
6
7       // allocate the file descriptor table entries
8       if ((r = fd_alloc(&fd0)) < 0//获取一个空闲的文件描述符
9           ||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
10          goto err;//文件描述符页面被 父子 进程共享
11
12      if ((r = fd_alloc(&fd1)) < 0
13          ||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
14          goto err1;
15
16      // allocate the pipe structure as first data page in both
17      va = fd2data(fd0);//获取fd0文件要存储的虚拟地址
18      if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
19          goto err2;
20      if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) < 0)
21          goto err3;//将fd0对应的物理内存映射到fd1对应的物理内存，这一物理内存存储struct Pipe
22
23      // set up fd structures
24      fd0->fd_dev_id = devpipe.dev_id;
25      fd0->fd_omode = O_RDONLY;
26
27      fd1->fd_dev_id = devpipe.dev_id;
28      fd1->fd_omode = O_WRONLY;
29
30      pfd[0] = fd2num(fd0);
31      pfd[1] = fd2num(fd1);
32      return 0;
33      err...
34  }
```

创建管道后，读描述符占一页，写描述符占一页，管道占一页。

管道读写

```
1  struct Pipe {
2      u_int p_rpos;        // read position
3      u_int p_wpos;        // write position
4      u_char p_buf[BY2PIPE];  // data buffer，32B，类似一个环形缓冲区
5  };
```

当管道为空，读者不能读。要保证 `p_rpos < p_wpos`

当管道满时，不可写。要保证`p_wpos - p_rpos < BY2PIPE` 。

当出现缓冲区空或满的情况时，如果另一端已经关闭，进程返回 0 即可； 如果没有关闭，让出CPU

如何判断管道是否关闭了呢？通过等式`pageref(rfd) + pageref(wfd) = pageref(pipe)`。

_pipeisclosed函数，检查`fd`所在的管道`p`的另一端是否关闭。关闭返回1，未关闭返回0。

> 为了保证读取fd和p的pageref过程中没有发生中断，即同步读。env_runs 记录了一个进程 env_run 的次数，可以根据某个操作 do() 前后进程 env_runs 值是否相等，来判断在 do() 中进程是否发生了切换。

```c
static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    int pfd,pfp,runs;
    do{
        runs = env->env_runs;
        pfd = pageref(fd);//返回fd对应的虚拟页所在物理页的pp_ref
        pfp = pageref(p);
    } while(runs != env->env_runs);

    if(pfd == pfp)
        return 1;
    return 0;
}
```

管道读函数

piperead函数，fd为读端文件描述符，vbuf为读出的缓冲区，n为要求读的字节数，offset是啥？？？。返回实际读到的字符数。

> 只有当没有读到东西，才yield；否则直接返回。

```c
static int piperead(struct Fd *fd, void *vbuf, u_int n, u_int offset)
{
    int i;
    struct Pipe *p;
    char *rbuf;
    p = fd2data(fd);//fd为读端文件描述符，对应的页为管道所在页
    rbuf = (char *)vbuf;
    for(i=0;i<n;i++){
        while (p->p_rpos == p->p_wpos){//如果管道为空
            if(_pipeisclosed(fd, p) || i>0)//若写端已关闭或者读到字符了已经
                return i;//管道为空，写端已关闭或已经读到字符时返回读到字符的个数，什么也没读到就是0啦
            syscall_yield();//管道为空，没有读到任何东西且管道没关闭时让出进程
        }
        rbuf[i] = p->p_buf[p->p_rpos % BY2PIPE];//环形缓冲区
        p->p_rpos++;
    }
    return n;
}
```

管道写函数

pipewrite函数，fd为写端文件描述符，vbuf为写入的缓冲区，n为要求写的字节数。返回实际写入的字符数。

```c
static int
pipewrite(struct Fd *fd, const void *vbuf, u_int n, u_int offset)
{
    int i;
    struct Pipe *p;
    char *wbuf;
    p = fd2data(fd);
```

```
    wbuf = (char *)vbuf;
    for(i=0;i<n;i++){
        while(p->p_wpos - p->p_rpos == BY2PIPE){//缓冲区已满
            if(_pipeisclosed(fd, p))//读端关闭，返回写入的字符数
                return i;
            syscall_yield();//缓冲区满且读端未关闭，让出进程
        }
        p->p_buf[p->p_wpos % BY2PIPE] = wbuf[i];
        p->p_wpos++;
    }
    return n;
```

shell

shell是一个命令行式命令解释器。

运行流程

`init.b`进程通过 spawn 生成 shell 进程。

文件`sh.c`可编译链接成可执行文件`sh.b`执行，其主函数为

> ARGBEGIN为宏定义
>
> ```
> #define ARGBEGIN
> for((argv ? 0:(argv=(void*)&argc)),argv++,argc--;//如果argv为空，执行:后面内容
>                         argv[0] && argv[0][0]=='-' && argv[0][1];
>                         argc--, argv++) {
>     char *_args, *_argt;
>     char _argc;
>     _args = &argv[0][1];
>     if(_args[0]=='-' && _args[1]==0){
>         argc--; argv++; break;
>     }
>     _argc = 0;
>     while(*_args && (_argc = *_args++))
>         switch(_argc)
> ```

```
void umain(int argc, char **argv)//argc为命令行参数个数，argv为参数（包含命令本身）
{
    int r, interactive, echocmds;
    interactive = '?';
    echocmds = 0;

    ARGBEGIN{
    case 'd':
        debug_++;
        break;
    case 'i':
        interactive = 1;
        break;
```

```
    case 'x':
        echocmds = 1;
        break;
    default:
        usage();
    }ARGEND

    if(argc > 1)
        usage();
    if(argc == 1){
        close(0);
        if ((r = open(argv[1], O_RDONLY)) < 0)
            user_panic("open %s: %e", r);
        user_assert(r==0);
    }
    if(interactive == '?')
        interactive = iscons(0);
    for(;;){
        if (interactive)
            fwritef(1, "\n$ ");
        readline(buf, sizeof buf);

        if (buf[0] == '#')
            continue;
        if (echocmds)
            fwritef(1, "# %s\n", buf);
        if ((r = fork()) < 0)
            user_panic("fork: %e", r);
        if (r == 0) {
            runcmd(buf);
            exit();
            return;
        } else
            wait(r);
    }
}
```

spawn

`int spawn(char *prog, char **argv)`函数，产生一个子进程，`prog`为程序路径，`argv`为其参数

```
//user/spawn.c
int spawn(char *prog, char **argv)
{
    u_char elfbuf[512];
    int r;
    int fd;
    u_int child_envid;
    int size, text_start;
    u_int i, *blk;
    u_int esp;
```

```
    Elf32_Ehdr* elf;
    Elf32_Phdr* ph;
    // Note 0: some variable may be not used,you can cancel them as you like
    // Step 1: Open the file specified by `prog` (prog is the path of the program)
    char progname[32];
    int name_len = strlen(prog);
    strcpy(progname, prog);
    if (name_len <= 2 || progname[name_len-1] != 'b' || progname[name_len-2] != '.'){
        strcat(progname, ".b");
    }

    if((r=open(progname, O_RDONLY))<0){
        progname[name_len] = 0;//需要确保没有.b
        writef("command [%s] is not found.\n", progname);
        return r;
    }
    // Your code begins here
    fd = r;//fd is prog's
    if((r = readn(fd, elfbuf, sizeof(Elf32_Ehdr))) < 0)//从prog文件中读Ehdr到elfbuf
        return r;
    elf = (Elf32_Ehdr*) elfbuf;
    if(!usr_is_elf_format(elf) || elf->e_type != 2)//2 means executable bin
        return -E_INVAL;

    // Step 2: Allocate an env (Hint: using syscall_env_alloc())
    r = syscall_env_alloc();
    if(r<0) return;
    if(r == 0) {//son executes this
        env = envs+ENVX(syscall_getenvid());
        return 0;
    }
    //father executes this for son
    child_envid = r;
    // Step 3: Using init_stack(...) to initialize the stack of the allocated env
    init_stack(child_envid, argv, &esp);//esp是子进程目前的栈底
    // Step 3: Map file's content to new env's text segment
    //        Hint 1: what is the offset of the text segment in file? try to use objdump to find out.
    //        Hint 2: using read_map(...)
    //        Hint 3: Important!!! sometimes ,its not safe to use read_map ,guess why
    //                If you understand, you can achieve the "load APP" with any method
    // Note1: Step 1 and 2 need sanity check. In other words, you should check whether
    //       the file is opened successfully, and env is allocated successfully.
    // Note2: You can achieve this func in any way ，remember to ensure the correctness
    //        Maybe you can review lab3
    text_start = elf->e_phoff;
    size = elf->e_phentsize;
    if((r = seek(fd, text_start)) < 0)
```

```
        return r;
    for(i=0; i<elf->e_phnum; i++){
        if((r = readn(fd, elfbuf, size)) < 0)
            return r;
        ph = (Elf32_Phdr*)elfbuf;
        if(ph->p_type == PT_LOAD){
            r = usr_load_elf(fd, ph, child_envid);
            if(r < 0)
                return r;
        }
    }
    // Your code ends here

    struct Trapframe *tf;

    writef("\n::::::::::spawn size : %x  sp : %x::::::::\n",size,esp);
    tf = &(envs[ENVX(child_envid)].env_tf);
    tf->pc = UTEXT;
    tf->regs[29]=esp;

    // Share memory
    u_int pdeno = 0;
    u_int pteno = 0;
    u_int pn = 0;
    u_int va = 0;
    for(pdeno = 0;pdeno<PDX(UTOP);pdeno++)
    {
        if(!((* vpd)[pdeno]&PTE_V))
            continue;
        for(pteno = 0;pteno<=PTX(~0);pteno++)
        {
            pn = (pdeno<<10)+pteno;
            if(((* vpt)[pn]&PTE_V)&&((* vpt)[pn]&PTE_LIBRARY))
            {
                va = pn*BY2PG;

                if((r = syscall_mem_map(0,va,child_envid,va,
(PTE_V|PTE_R|PTE_LIBRARY)))<0)
                {

                    writef("va: %x   child_envid: %x   \n",va,child_envid);
                    user_panic("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");
                    return r;
                }
            }
        }
    }

    if((r = syscall_set_env_status(child_envid, ENV_RUNNABLE)) < 0)
    {
        writef("set child runnable is wrong\n");
        return r;
    }
```

```
    return child_envid;

}
```

init_stack函数，初始化子进程栈空间。

```
//利用临时页TMPPAGE，先把argv等保存后，最后映射到USTACK
int init_stack(u_int child, char **argv, u_int *init_esp)
{
    int argc, i, r, tot;
    char *strings;
    u_int *args;

    // Count the number of arguments (argc)
    // and the total amount of space needed for strings (tot)
    tot = 0;
    for (argc = 0; argv[argc]; argc++)//argc和argv的实际个数相等
        tot += strlen(argv[argc]) + 1;

    // Make sure everything will fit in the initial stack page
    if (ROUND(tot, 4) + 4 * (argc + 3) > BY2PG)
        return -E_NO_MEM;

    // Determine where to place the strings and the args array
    strings = (char *)TMPPAGETOP - tot;
    args = (u_int *)(TMPPAGETOP - ROUND(tot, 4) - 4 * (argc + 1));//argv[0]的起始地址

    if ((r = syscall_mem_alloc(0, TMPPAGE, PTE_V | PTE_R)) < 0)
        return r;
    // Replace this with your code to:
    //
    //  - copy the argument strings into the stack page at 'strings'
    char *ctemp, *argv_temp;
    u_int j;
    //将所有argv参数保存到相应位置
    ctemp = strings;
    for (i = 0; i < argc; i++)
    {
        argv_temp = argv[i];
        for (j = 0; j < strlen(argv[i]); j++)
        {
            *ctemp = *argv_temp;
            ctemp++;
            argv_temp++;
        }
        *ctemp = 0;
        ctemp++;
    }
    //  - initialize args[0..argc-1] to be pointers to these strings
    //    that will be valid addresses for the child environment
    //    (for whom this page will be at USTACKTOP-BY2PG!).
```

```
    ctemp = (char *)(USTACKTOP - TMPPAGETOP + (u_int)strings);//ctemp是USTACKTOP位置处的strings
    for (i = 0; i < argc; i++)
    {
        args[i] = (u_int)ctemp;//args初始位置是在TMPPAGE的相应位置
        ctemp += strlen(argv[i]) + 1;
    }
    //  - set args[argc] to 0 to null-terminate the args array.
    ctemp--;
    args[argc] = ctemp;
    //  - push two more words onto the child's stack below 'args',
    //    containing the argc and argv parameters to be passed
    //    to the child's umain() function.
    u_int *pargv_ptr;
    pargv_ptr = args - 1;
    *pargv_ptr = USTACKTOP - TMPPAGETOP + (u_int)args;
    pargv_ptr--;
    *pargv_ptr = argc;
    //
    //  - set *init_esp to the initial stack pointer for the child
    //
    *init_esp = USTACKTOP - TMPPAGETOP + (u_int)pargv_ptr;
    //  *init_esp = USTACKTOP;  // Change this!

    if ((r = syscall_mem_map(0, TMPPAGE, child, USTACKTOP - BY2PG, PTE_V | PTE_R)) < 0)
        goto error;
    if ((r = syscall_mem_unmap(0, TMPPAGE)) < 0)
        goto error;

    return 0;

error:
    syscall_mem_unmap(0, TMPPAGE);
    return r;
}

```

此例中`argc`为2，`argv`指向字符串指针的指针，`argv[i]`指向字符串，`argv[2]`指向一个空字符串，表示参数的结束。

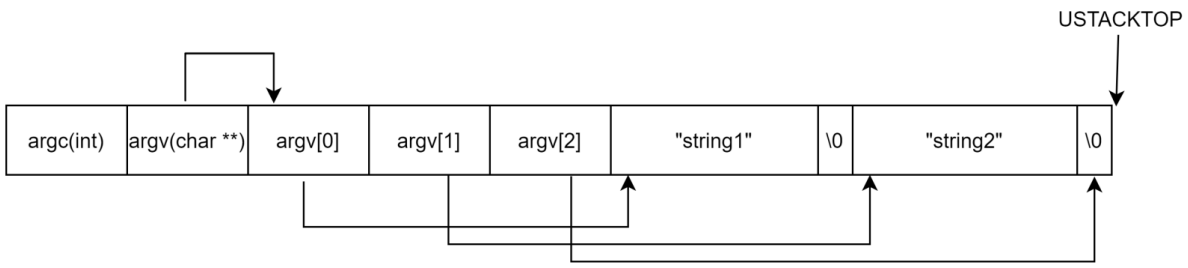

图 6.3: 子进程栈空间示意图

argc is the number of arguments

获取token

1. `int gettoken(char *s, char **p1)`，若s不为0，则开始解析s；若s为0，*p1为token，返回值为特殊字符或'w'

```
int gettoken(char *s, char **p1)
{
    static int c, nc;
    static char *np1, *np2;

    if (s) {
        nc = _gettoken(s, &np1, &np2);//nc为特殊字符
        return 0;
    }
    //若s为0
    c = nc;//返回特殊字符
    *p1 = np1;
    nc = _gettoken(np2, &np1, &np2);//接着找特殊字符
    return c;
}
```

2. `int _gettoken(char *s, char **p1, char **p2)` 函数

   s字符串中，如果找到

   - 特殊字符：返回值为特殊字符，p2 just past the token
   - 单词：返回值为'w'，p1为单词开始，p2 just past the token

```
char *strchr(const char *str, int c)
```

在字符串str中找字符c，返回包括字符c之后的字符串，若未找到返回null

```
#define SYMBOLS "<|>&;()"
int _gettoken(char *s, char **p1, char **p2)
{
    int t;

    if (s == 0) {
        return 0;
    }

    *p1 = 0;
    *p2 = 0;

    while(strchr(WHITESPACE, *s))//遇到空格跳过
        *s++ = 0;
    if(*s == 0) {//s为空串
        return 0;
    }
    if(strchr(SYMBOLS, *s)){//遇到特殊字符<|>&;()
        t = *s;
```

```
        *p1 = s;//p1为包括该特殊字符之后的字符串
        *s++ = 0;//置0
        *p2 = s;//p2为该特殊字符之后的字符串
        return t;//返回特殊字符
    }
    //不是特殊字符，说明为单词
    *p1 = s;//p1为包含单词的字符串
    while(*s && !strchr(WHITESPACE SYMBOLS, *s))
        s++;
    *p2 = s;//*p2为正好跳过单词的字符串
    return 'w';
}
```

执行命令

`void runcmd(char *s)`,执行一行指令s

```
#define MAXARGS 16
void
runcmd(char *s)
{
    char *argv[MAXARGS], *t;//argv保存参数
    int argc, c, i, r, p[2], fd, rightpipe;
    int fdnum;
    struct Stat state;
    rightpipe = 0;
    gettoken(s, 0);//解析s
again:
    argc = 0;//一条指令的参数个数，包含命令本身
    for(;;){
        c = gettoken(0, &t);//c是第一个特殊字符，如果c为'w'则*t为单词
        switch(c){
        case 0:
            goto runit;
        case 'w':
            if(argc == MAXARGS){
                writef("too many arguments\n");
                exit();
            }
            argv[argc++] = t;
            break;
        case '<':
            if(gettoken(0, &t) != 'w'){//下一个token不是单词
                writef("syntax error: < not followed by word\n");
                exit();
            }
            // Your code here -- open t for reading,
            // dup it onto fd 0, and then close the fd you got.
            r = stat(t, &state);//获得文件t打开的相关信息，返回值为负数则打开失败
            if(r<0){
                writef("cannot open file\n");
                exit();
            }
```

```
            if(state.st_isdir != 0){//如果为目录，则isdir为1，否则为0
                writef("specified path should be file\n");
                exit();
            }
            fdnum = open(t, O_RDONLY);
            dup(fdnum, 0);//dup将fdnum所在的struct Fd页面和保存文件内容的页面复制给0
            close(fdnum);
            break;
        case '>':
            if(gettoken(0, &t) != 'w'){
                writef("syntax error: < not followed by word\n");
                exit();
            }
            r = stat(t, &state);//获得文件t打开的相关信息
            if(r<0){
                writef("cannot open file\n");
                exit();
            }
            if(state.st_isdir != 0){
                writef("specified path should be file\n");
                exit();
            }
            fdnum = open(t, O_WRONLY|O_CREAT);
            dup(fdnum, 1);//输出的内容到该文件里
            close(fdnum);
            break;
        case '|':
            // Your code here.
            //  First, allocate a pipe.
            //  Then fork.
            //  the child runs the right side of the pipe:
            //      dup the read end of the pipe onto 0
            //      close the read end of the pipe
            //      close the write end of the pipe
            //      goto again, to parse the rest of the command line
            //  the parent runs the left side of the pipe:
            //      dup the write end of the pipe onto 1
            //      close the write end of the pipe
            //      close the read end of the pipe
            //      set "rightpipe" to the child envid
            //      goto runit, to execute this piece of the pipeline
            //          and then wait for the right side to finish
            pipe(p);
            rightpipe = fork();
            if(rightpipe == 0){//子进程
                dup(p[0], 0);//输入给到管道读端，子进程执行的是 | 后面的内容
                close(p[0]);
                close(p[1]);
                goto again;
            }else {
                dup(p[1], 1);//输出的内容进入管道写端
                close(p[0]);
                close(p[1]);
```

```
                goto runit;
            }
            break;
        default:
            break;
        }
    }

runit://执行命令
    if(argc == 0) {
        return;
    }
    argv[argc] = 0;
    if ((r = spawn(argv[0], argv)) < 0)
        writef("spawn %s: %e\n", argv[0], r);
    close_all();
    if (r >= 0) {
        wait(r);//等待子进程结束
    }
    if (rightpipe) {//等待管道右端的子进程结束
        wait(rightpipe);
    }

    exit();
}
```

spawn的练习思考

第一次执行`readelf -l testbss.b`，结果为：

```
Elf file type is EXEC (Executable file)
Entry point 0x400000
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  REGINFO        0x0044b0 0x004034b0 0x004034b0 0x00018 0x00018 R   0x4
  LOAD           0x001000 0x00400000 0x00400000 0x075b8 0x115c4 RWE 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .reginfo
   01     .text .reginfo .data .data.rel .data.rel.local
```

执行`size testbss.b`，结果为

```
   text    data     bss     dec     hex filename
  13512   13751   40964   68227   10a83 testbss.b
```

第二次改为`int bigarray[ARRAYSIZE]={1};`，执行`readelf`，结果为

```
Elf file type is EXEC (Executable file)
Entry point 0x400000
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  REGINFO        0x0044b0 0x004034b0 0x004034b0 0x00018 0x00018 R   0x4
  LOAD           0x001000 0x00400000 0x00400000 0x115b8 0x115c4 RWE 0x1000//FileSiz增大了

 Section to Segment mapping:
  Segment Sections...
   00     .reginfo
   01     .text .reginfo .data .data.rel .data.rel.local
```

`size`结果为

```
   text    data     bss     dec     hex filename
  13512   54711       4   68227   10a83 testbss.b//data多了40960，bss少了40960
```

实验感想

作为最后一个lab，实现一个shell还是很令人振奋的。准备挑战性任务就做lab6，实现更强大的shell。本次实验相比lab5不是很难。