林子杰_20373980_lab6

一、思考题

Thinking 6.1

修改代码如下即可。注意，对于父进程，`fork()`返回子进程的进程id；而对于子进程，函数返回0。

```
int main() {
    status = pipe(fildes);
    if (status == -1) {
/* an error occurred */
        printf("error\n");
    }
    switch (fork()) {
        case -1: /* Handle error */
            break;
        case 0: /* Child - writes to pipe */
            close(fildes[0]); /* Read end is unused */
            write(fildes[1], "Bonjour monde\n", 14); /* Get data from pipe */
            close(fildes[1]); /* Finished with pipe */
            exit(EXIT_SUCCESS);
        default: /* Parent - reads from pipe */
            close(fildes[1]); /* Write end is unused */
            read(fildes[0], buf, 100);
            printf("father-process read:%s", buf); /* Print the data */
            close(fildes[0]); /* Child will see EOF */
            exit(EXIT_SUCCESS);
    }
}
```

Thinking 6.2

观察`dup()`，发现它也有两类映射的`map`操作，先执行的是文件描述符的映射，后执行的是对文件内容的映射。可以仿照指导书构造程序：

```
if(fork() == 0) {
     dup(a);
     operation();
 }
 else {
     read(a);
 }
```

`read()`函数通过判断文件描述符`a`是否被映射来判断文件是否可以被读取。倘若程序执行顺序如下：

- `fork()`执行后，子进程执行`dup(a)`，但还没执行完所有数据的`map`操作就切换到父进程

- 父进程执行`read(a)`，但子进程仅执行了文件描述符的`map`操作，父进程误以为所有数据都已经完成映射，读取数据发生错误

  其原因是这些`map`操作并非原子操作，而可能在中途被打断；而外部的条件判断又没有适应这种"错误"。

Thinking 6.3

在MOS操作系统中，进入系统调用的时候会屏蔽中断，故而MOS操作系统的系统调用都是原子操作：

```
.macro CLI
        mfc0    t0, CP0_STATUS
        li      t1, (STATUS_CU0 | 0x1)
        or      t0, t1
        xor     t0, 0x1
        mtc0    t0, CP0_STATUS
.endm
```

不过，Linux下的`write()`系统调用并非原子的，这是一个比较特殊的情况。

Thinking 6.4

- 能够解决这个问题。`pipe`的引用次数总是大于等于`p[0]`或者`p[1]`，而问题的出现就在于`pipe`和`p[*]`都减少1时插入了一个判断操作，使得`pipe`先减少了1，出现了误判等于的情况。倘若`pipe`后减少1，则不会出现这种情况，故而能够解决这一问题。
- 倘若遇到的问题是前面Thinking描述的情景，那么可以在所有的数据都映射完之后再映射文件描述符，当热这样的操作好像不论如何都会更加安全一些。

Thinking 6.5

可以看到，我们实现的`usr_load_elf()`函数中，在结束了对其他段的加载后，有这么一段代码：

```
while (i < sgsize) {
    r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R);
    if(r < 0) return r;
    i += BY2PG;
}
```

这段代码就是用于实现`.bss`段的声明的，不断声明页面至`sgsize`。lab3中的`load_icode_mapper()`函数也是这样实现这一功能的，并且其在声明页面的时候页面会被自动填充为0，此处应当也是类似的。

Thinking 6.6

回顾`exercise 3.13`，在`./tools/secs0_3.lsd`文件下，我们会发现我们已经完成了对MOS操作系统中二进制文件各个段的地址的规定，，故而`*.b`文件的`text`段偏移值都是一样的。

Thinking 6.7

在`user/init.c`中，调用了这一段代码：

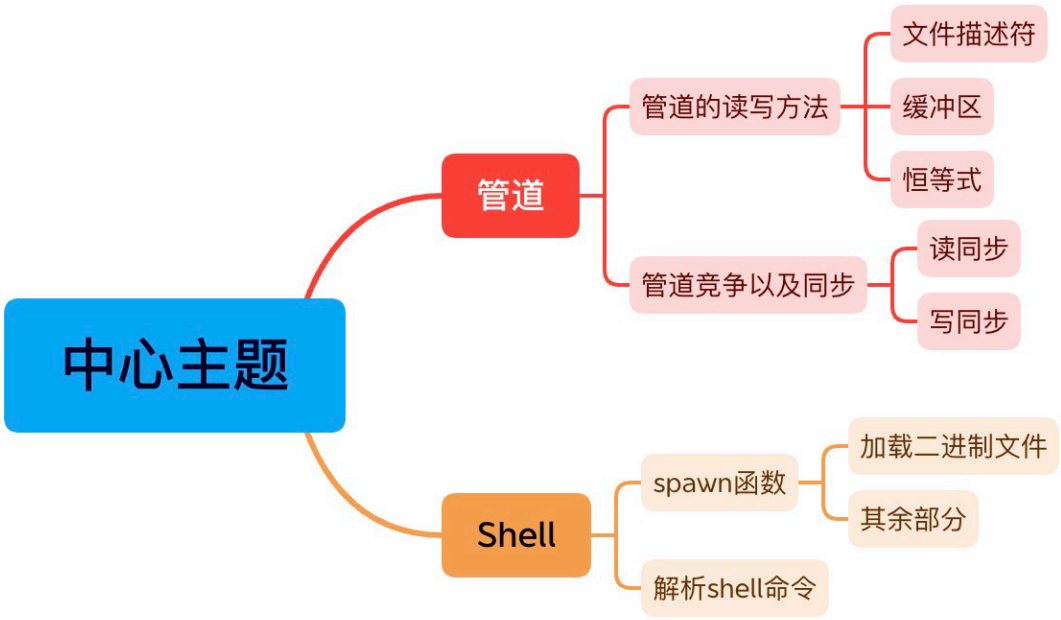
```
if ((r = opencons()) < 0)
    user_panic("opencons: %e", r);
if (r != 0)
    user_panic("first opencons used fd %d", r);
if ((r = dup(0, 1)) < 0)
    user_panic("dup: %d", r);

int opencons(void)
{
    int r;
    struct Fd *fd;

    if ((r = fd_alloc(&fd)) < 0)
        return r;
    if ((r = syscall_mem_alloc(0, (u_int)fd, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        return r;
    fd->fd_dev_id = devcons.dev_id;
    fd->fd_omode = O_RDWR;
    return fd2num(fd);
}
```

`opencons()`函数拿取了0号文件描述符，并且将其映射到`console`设备上，并且只赋只读权限。在后面的`dup()`函数中，应当是占用了1号占位符，这样就完成了对标准输入输出的"安排"。

二、实验难点



lab6的知识点不多，但需要我们有很充分的理解才能顺利地完成这一部分的内容。其中比较难以理解的地方包括管道竞争产生错误的原因以及管道同步的方法、`spawn()`函数中关于ELF解析的部分，前者需要在纸上绘图来辅助理解，后者则有较大的工程量。尤其是`spawn()`函数，需要结合着其他的代码来完成。

三、体会与感想

lab6是在烤漆的时候匆忙完成的，并且没有上机考试，故而做完后没有那么深切的感触。不过lab6的难度还是给我造成了不小的阻碍，尤其是`spawn()`函数和管道同步的部分。lab6虽然让我接触到了一些管道和shell的知识，但总让我感到一种和前面的知识发生脱节的感觉，联系并不是很紧密（或许是指导书的问题）。不论如何，我的OS之旅也随着这份报告而结束了，尽管在上机考试的关键exam中没有表现好，失去了申优的机会，但总归是学到了不少东西。按照惯例，还是把本次lab的实验笔记附在后面，并且将所有的报告和笔记附在我的Github链接中：https://github.com/Enqurance/BUAA_2022_OS_Notes

Lab6实验笔记

一、管道

1.基本印象

lab4中，我们学习过IPC的进程通信方式，相似地，管道也是一种进程通信方式。管道的工作原理和它的名字保持高度一致：一端进，一端出，且同一时刻只能往一个方向传输，因此管道是典型的单项通信方法。管道也叫做匿名管道，只能用在拥有公共祖先的进程中，通常用于父子进程间的通信。

下面举一个例子来说明管道的使用方法。

```
#include <stdlib.h>
#include <unistd.h>
```

```
#include "stdio.h"

int fildes[2];
char buf[100];
int status;

int main() {
    /* 调用pipe函数，返回两个文件描述符，fd[0]对应读，fd[1]对应写 */
    status = pipe(fildes);
    if (status == -1) {
    /* 发生错误 */
        printf("error\n");
    }
    switch (fork()) {
        case -1: /* fork()出错 */
            break;
        case 0: /* 子进程，从管道读数据 */
            close(fildes[1]); /* 关闭管道的写功能 */
            read(fildes[0], buf, 100); /* 从管道读 */
            printf("child-process read:%s", buf); /* 打印读出的数据 */
            close(fildes[0]); /* 关闭管道读，进程退出 */
            exit(EXIT_SUCCESS);
        default: /* 父进程，向管道写数据 */
            close(fildes[0]); /* 关闭读功能 */
            write(fildes[1], "Hello world\n", 12); /* 向管道写数据 */
            close(fildes[1]); /* 关闭写功能 */
            exit(EXIT_SUCCESS);
    }
}
```

这样，我们就了解了使用管道的大致流程。首先，我们将会声明一个数组用于保存文件描述符；然后，我们会调用`pipe()`函数，声明一个可使用的管道；接着，我们就可以使用`fork()`函数调用管道来读写数据了。我们发现，使用管道的时候，通常都会将本次操作中不需要用到的功能关闭。

看到文件描述符，我们就可以判断，管道的数据传输功能应当是通过文件系统实现的。由于`fork()`函数会且仅会让父子进程共享地址空间，故而文件描述符也只能在父子进程之间使用，这也是管道保证通信的匿名性的原因之一。

Thinking 6.1

修改代码如下即可。注意，对于父进程，`fork()`返回子进程的进程id；而对于子进程，函数返回0。

```
int main() {
    status = pipe(fildes);
    if (status == -1) {
/* an error occurred */
        printf("error\n");
    }
    switch (fork()) {
```

```
        case -1: /* Handle error */
            break;
        case 0: /* Child - writes to pipe */
            close(fildes[0]); /* Read end is unused */
            write(fildes[1], "Bonjour monde\n", 14); /* Get data from pipe */
            close(fildes[1]); /* Finished with pipe */
            exit(EXIT_SUCCESS);
        default: /* Parent - reads from pipe */
            close(fildes[1]); /* Write end is unused */
            read(fildes[0], buf, 100);
            printf("father-process read:%s", buf); /* Print the data */
            close(fildes[0]); /* Child will see EOF */
            exit(EXIT_SUCCESS);
    }
}
```

2.管道测试

本次的lab，在填写实验代码前，先向我们介绍了测试文件的内容。

```
#include "lib.h"
char *msg = "Now is the time for all good men to come to the aid of their party.";
void umain(void)
{
    char buf[100];
    int i, pid, p[2];
    if ((i = pipe(p)) < 0) {
        user_panic("pipe: %e", i);
    }
    if ((pid = fork()) < 0) {
        user_panic("fork: %e", i);
    }
    if (pid == 0) {   //子进程的if分支
        writef("[%08x] pipereadeof close %d\n", env->env_id, p[1]);
        close(p[1]);    //关闭写功能
        writef("[%08x] pipereadeof readn %d\n", env->env_id, p[0]);
        i = readn(p[0], buf, sizeof(buf) - 1);  //读数据
        if (i < 0) {
            user_panic("read: %e", i);
        }
        buf[i] = 0;
        if (strcmp(buf, msg) == 0) {    //比较读出的信息
            writef("\npipe read closed properly\n");
        } else {
            writef("\ngot %d bytes: %s\n", i, buf);
        }
        ......
```

可以看到，测试的方法和前面介绍的程序非常相似，都是使用了管道的读写模式。在lab4中，我们实现的`fork()`函数中，我们对所有用户态的地址空间都使用`PTE_COW`进行了保护，但在写时复制机制下，管道可能会不可用。因为在`pipe()`函数中，我们会为两个文件描述符分配空间，然后将一个管道作为这两个文件描述符的第一个数据，从而使得他们能够共享一个管道的数据缓冲区。

此处再借用一下指导书的图片，来说明管道的工作原理。图中，每当需要操作管道的时候，只允箭头指向一个方向。关闭管道的端口能够保证使用的安全性。

3.管道的读写

```
struct Pipe {
    u_int p_rpos;    //下一个将要从管道读取数据的位置
    u_int p_wpos;    //下一个将要从管道写入数据的位置
    u_char p_buf[BY2PIPE]; //管道缓冲区，大小为32字节
}
```

每次操作过程中，只有读者可以更新`Pipe`结构体的`p_rpos`，只有写者可以更新`p_wops`。缓冲区只有32字节大小，通过`p_*%32`将数据写入对应位置。当然，读写者在此处应当是协作关系：写者要先向缓冲区写入，读者才有数据可读。故而，当读者从管道读取数据时，若发现`p_rpos >= p_wpos`，即读者已经读到"最新"的数据了，那么我们需要将进程切换至写者进程。当然，写者写数据也不是无限制的写入，毕竟缓冲区的大小有限，故而写者在`p_wpos-p_rpos<32`时才可写入。

当然，仅仅如此可能还无法保证管道的正常工作。倘若写者进程已经全部结束，读者进程也已经读到管道数据的末尾，此时读写位置相同，按照上文介绍的机制，应当切换进程；然而我们已经没有写者进程了，进程切换会陷入死循环。为了解决这一问题，我们需要增加对管道另一端状态判断的机制。当出现缓冲区空或者满的情况时，判断管道的另一端是否关闭，若关闭，则返回0；若未关闭，则可切换进程。

如何判断管道另一端的情况？`_pipeisclosed()`使用恒等式进行判断。`Page`结构体当中有一个`pageref`熟悉过，用于记录页面被引用的次数。倘若此时有一个读者，一个写者在使用这个管道，那么该管道的`pageref`就是2，它被读者引用了一次，也被写者引用了一次。由此我们可以归纳出：

```
pageref(read_fd) + pageref(write_fd) == pageref(pipe)
```

倘若读写其中一端已经完全关闭，那么就会有：

```
pageref(read_fd) == pageref(pipe) || pageref(write_fd) == pageref(pipe)
```

我们可以据此来判断管道某一端是否完全关闭。下面的`exercise 6.2`需要我们完成读函数、写函数以及`_pipeisclosed()`函数。

```
static int piperead(struct Fd *fd, void *vbuf, u_int n, u_int offset)
{
    int i;
    struct Pipe *p;
    char *rbuf;
```

```
    p = (struct Pipe*)fd2data(fd);
    rbuf = (char *)vbuf;
    /* p->p_rpos == p->p_wpos,则表示没有数据可读，首先判断写端是否全部
     * 关闭，倘若全部关闭，则本函数直接退出即可；倘若没有关闭，则挂起本进程
     * 并且进入进程调度 */
    while(p->p_rpos == p->p_wpos) {
        if(_pipeisclosed(fd, p)) {
            return 0;
        } else {
            syscall_yield();
        }
    }
    /* 开始读取n字节的数据，倘若写端全部关闭，可以提前退出 */
    for (i = 0;i < n;i++) {
        while (p->p_rpos == p->p_wpos) {
            if (i > 0 || _pipeisclosed(fd, p)) {
                return i;
            }
            syscall_yield();
        }
        /* 依照缓冲区大小读取数据 */
        rbuf[i] = p->p_buf[p->p_rpos % BY2PIPE];
        p->p_rpos++;
    }
    return n;
    user_panic("piperead not implemented");
}

static int pipewrite(struct Fd *fd, const void *vbuf, u_int n, u_int offset)
{
    int i;
    struct Pipe *p;
    char *wbuf;
    p = (struct Pipe*)fd2data(fd);
    wbuf = (char *)vbuf;
    /* 开始读取n字节的数据，倘若p->p_wpos - p->p_rpos == BY2PIPE
     * 则表示缓冲区已满，根据读端的情况判断是进入进程调度，还是需要退出
     * 本函数 */
    for (i = 0; i < n; i++) {
        while (p->p_wpos - p->p_rpos == BY2PIPE) {
            if(pipeisclosed(fd2num(fd))) {
                return 0;
            }
            syscall_yield();
        }
        /* 依据缓冲区大小读取数据 */
        p->p_buf[p->p_wpos % BY2PIPE] = wbuf[i];
        p->p_wpos++;
    }
```

```
    return n;
    user_panic("pipewrite not implemented");
}

static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    int pfd,pfp,runs;
    do {
        runs = env->env_runs;
        /* 我们先不关注do...while语句。此处取出传入的文件描述符的引
         * 用次数和管道pipe的引用次数，并对其进行比较即可 */
        pfd = pageref(fd);
        pfp = pageref(p);
    } while(runs != env->env_runs);
    if (pfd == pfp) {
        return 1;
    } else {
        return 0;
    }
    user_panic("_pipeisclosed not implemented");
}
```

至此，管道读写的任务已经完成了一半。

4.管道的竞争

MOS操作系统采用的是时间片轮转算法，进程在执行的过程中随时都可能被打断。这要求我们在考虑被多个进程共享的数据的访问时，需要格外的小心。管道就是这样的类型，我们需要考虑多个进程使用管道时可能发生的竞争。

首先要明确，多个进程（尤其是读写进程同时存在时）使用管道，最好应当是同步的，以免发生错误。但考虑下面一种情况：

```
pipe(p);
if(fork() == 0) {
    close(p[1]);
    read(p[0],buf,sizeof buf);
} else {
    close(p[0]);
    write(p[1],"Hello",5);
}
```

倘若按照如下的顺序执行：

- `fork()`结束后，产生了父子两个进程，且都对管道有读写权限，此时`pageref(pipe)`为4。先执行子进程。执行完`close(p[1])`后，切换到父进程执行。此时`pageref(pipe)`为3，`pageref(p[1])`为1
- 父进程执行`close(p[0])`时，`p[0]`已经解除了对管道的映射，但还未解除进程对`p[0]`的映射时发生了进程切换。此时`pageref(pipe)`为2，`pageref(p[1])`为1。

- 此时子进程开始读数据。比较`pageref(pipe)`和`pageref(p[0])`，居然相等，写端已经全部关闭了，读函数直接退出，出现了Bug。

很显然，由于两次`unmap`不是原子操作，导致恒等式判断出现了问题。

Thinking 6.2

观察`dup()`，发现它也有两类映射的`map`操作，先执行的是文件描述符的映射，后执行的是对文件内容的映射。可以仿照指导书构造程序：

```
if(fork() == 0) {
     dup(a);
     operation();
 }
 else {
     read(a);
 }
```

`read()`函数通过判断文件描述符`a`是否被映射来判断文件是否可以被读取。倘若程序执行顺序如下：

- `fork()`执行后，子进程执行`dup(a)`，但还没执行完所有数据的`map`操作就切换到父进程
- 父进程执行`read(a)`，但子进程仅执行了文件描述符的`map`操作，父进程误以为所有数据都已经完成映射，读取数据发生错误

其原因是这些`map`操作并非原子操作，而可能在中途被打断；而外部的条件判断又没有适应这种"错误"。

Thinking 6.3

在MOS操作系统中，进入系统调用的时候会屏蔽中断，故而MOS操作系统的系统调用都是原子操作：

```
.macro CLI
        mfc0    t0, CP0_STATUS
        li      t1, (STATUS_CU0 | 0x1)
        or      t0, t1
        xor     t0, 0x1
        mtc0    t0, CP0_STATUS
.endm
```

不过，Linux下的`write()`系统调用并非原子的，这是一个比较特殊的情况。

那么如何解决这个问题？将两次`unmap`操作整合为原子操作？这显然有点费力气，不是好的选择。是否可以考虑其他的办法？我们注意到，`_pipeisclosed()`返回正确结果的条件是：

- 写端关闭当且仅当`pageref(p[0]) == pageref(pipe)`
- 读端关闭当且仅当`pageref(p[1]) == pageref(pipe)`

我们可以从反面的角度来思考这个问题，以第一条为例：当写端没有关闭时，`pageref(p[0]) != pageref(pipe)`，这个是显而易见的。而`pipe`得引用次数从理论上来讲只要两端都没有完全关闭，那么`pageref(pipe)`肯定是大于另外两个引用的。出现问题的原因则是在进行映射时，按照代码逻辑，先解除了`pipe`的映射，使其引用次数减少一，而又在解除`p[*]`的映射前发生了进程切换。既然如此，可不可以将解

除`pipe[]`映射的操作放在解除`p[*]`映射之后呢？答案是可行的。

Thinking 6.4

- 能够解决这个问题。`pipe`的引用次数总是大于等于`p[0]`或者`p[1]`，而问题的出现就在于`pipe`和`p[*]`都减少1时插入了一个判断操作，使得`pipe`先减少了1，出现了误判等于的情况。倘若`pipe`后减少1，则不会出现这种情况，故而能够解决这一问题。
- 倘若遇到的问题是前面Thinking描述的情景，那么可以在所有的数据都映射完之后再映射文件描述符，当热这样的操作好像不论如何都会更加安全一些。

`exercise 6.3`就是让我们呢根据上面得到的结论，通过调换映射顺序的方法来解决`map`或者`unmap`顺序不当导致的错误。内容并不复杂，就是把一片原来位于程序前段的代码剪切到程序后段，此处不做介绍。

然而到此，问题还没有完全解决。我们刚刚只是解决了"写数据"时可能出现的错觉，现在还需考虑读数据时的同步问题。还是刚刚那段程序：

```
pipe(p);
if(fork() == 0) {
    close(p[1]);
    read(p[0],buf,sizeof buf);
} else {
    close(p[0]);
    write(p[1],"Hello",5);
}
```

注意，如果按照指导书一步步推进，我们的`_pipeisclose()`函数目前长这样：

```
static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    int pfd,pfp,runs;

    pfd = pageref(fd);
    pfp = pageref(p);

    if (pfd == pfp) {
        return 1;
    } else {
        return 0;
    }
    user_panic("_pipeisclosed not implemented");
}
```

考虑下面一组程序执行过程：

- `fork()`函数结束，进入子进程。子进程执行完`close(p[1])`后执行`read()`。由于此时管道中尚且没有数据，故而子进程进入`_pipeisclosed()`函数判断写端是否关闭。执行到`pfd = pageref(fd)`后，进程切换。此时`pdf == 2`，因为父子进程目前都持有`p[0]`
- 父进程执行`close(p[0])`后，开始向管道写数据，尚未写完时又切换到子进程
- 子进程继续执行`_pipeisclosed()`函数，执行`pfp = pageref(p)`，发现管道的引用次数为2即`pfp == 2`

`== pdf`， `read()`函数直接退出

我们此时可以发现问题的所在：子进程在执行到`_pipeisclose()`函数的一半时发生切换，此时`p[*]`和`pipe`的引用次数都发生了更新，但子进程已经执行的那一部分却无法获悉这种更新，此时就发生了判断上的错误。

经过上述分析，我们可以得知出现此误判的根源就是没有向进程及时反馈页面引用的变化。函数中一直没用到的变量`runs`则是用于解决这个问题的。进程控制块中的`env_runs`用于记录一个进程`env_run`的次数，我们可以通过维护`runs`的值来解决这个问题。

```
static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    int pfd,pfp,runs;
    do {
        runs = env->env_runs;
        pfd = pageref(fd);
        pfp = pageref(p);
    } while(runs != env->env_runs);
    /* 如果runs和当前进程的env_runs不相等，则将全部变量都更新一次 */
    if (pfd == pfp) {
        return 1;
    } else {
        return 0;
    }
    user_panic("_pipeisclosed not implemented");
}
```

如此，`exercise 6.4`的内容也就完成了。

二、Shell

1.spawn函数

`spawn()`函数的作用是调用文件系统中的可执行文件并执行。借用指导书的内容，其执行流程如下：

- 从文件系统打开对应的文件 (二进制 ELF，即`.b`文件)。
- 用`fork()`创建子进程
- 将目标程序加载到子进程的地址空间中，并为它们分配物理页面；
- 为子进程初始化堆、栈空间，并设置栈顶指针, 以及重定向、管道的文件描述符，对于栈空间，因为我们的调用可能是有参数的，所以要将参数也安排进用户栈中。大家下个学期学习编译原理后，会对这一点有更加深刻的认识。
- 设置子进程的寄存器 (栈寄存器`sp`设置为`esp`。程序入口地址`pc`设置为`UTEXT`)
- 这些都做完后，设置子进程可执行。

这里用到了一些lab3中的知识（加载目标进程的过程），就不在此赘述了，我们直接来观察`spawn()`函数的代码。这个函数非常复杂。你可以选择先实现一个`usr_load_elf()`函数来精简你的代码。

```
int usr_load_elf(int fd , Elf32_Phdr *ph, int child_envid){
    int r, i;
```

```
    u_char *blk = NULL;
    static u_int TempPage = USTACKTOP;
    r = read_map(fd, ph -> p_offset, &blk);
    if(r < 0) {
        return r;
    }
    u_char *bin = blk;
    u_int va = ph -> p_vaddr;
    u_int sgsize = ph -> p_memsz;
    u_int bin_size = ph -> p_filesz;
    /* 求偏移，通过计算offset得到加载地址是否对齐 */
    u_int offset = va - ROUNDDOWN(va, BY2PG);
    u_int size = 0;
    /* 不能对齐，则先拷贝这一部分的数据 */
    if(offset > 0) {
        size = BY2PG - offset;
        /* 声明页面并且进行映射，然后使用user_bcopy()拷贝数据，首先消除了第一个不对齐的部分 */
        r = syscall_mem_alloc(child_envid, va - offset, PTE_V | PTE_R);
        if(r < 0) return r;
        r = syscall_mem_map(child_envid, va - offset, 0, TempPage,PTE_V|PTE_R);
        if(r < 0) return r;
        user_bcopy((void *)bin, (void *)(TempPage + offset), MIN(bin_size, size) );
        r = syscall_mem_unmap(0, TempPage);
        if(r < 0) return r;
    }
    /* 不断声明页面并且映射，然后进行拷贝，并且可同时处理尾部不对齐的情况 */
    for (i = size; i < bin_size; i += BY2PG) {
        r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R);
        if(r < 0) return r;
        r = syscall_mem_map(child_envid, va + i, 0, TempPage, PTE_V|PTE_R);
        if(r < 0) return r;
        user_bcopy((void * )(bin + i), (void *)TempPage, MIN(bin_size - i, BY2PG) );
        r = syscall_mem_unmap(0, TempPage);
        if(r < 0) return r;
    }
    /* 声明页面至sgsize */
    while (i < sgsize) {
        r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R);
        if(r < 0) return r;
        i += BY2PG;
    }
    return 0;
}

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;
    /* 以O_RDONLY权限打开对应文件，并赋值给文件描述符 */
    if((r = open(prog, O_RDONLY))<0){
        user_panic("Failed to open...\n");
        return r;
    }
    fd = r;
    /* 映射文件描述符 */
    if((r = read_map(fd, 0, &blk)) < 0) {
        writef("read_map() failed...\n");
        return r;
    }
    /* 判断文件是否是二进制文件，这一部分和lab3中相同 */
    elf = (Elf32_Ehdr *)blk;
    size = ((struct Filefd *)num2fd(fd)) -> f_file.f_size;
    if(size < 4 || !usr_is_elf_format(elf)) {
        writef("Format error...\n");
        return -1;
    }
    /* 声明一个新的子进程 */
    child_envid = syscall_env_alloc();
    if(child_envid < 0) {
        writef("Failed to alloc a env\n");
        return child_envid;
    }
    /* 为子进程初始化堆栈，这一函数已经封装好了 */
    if((r = init_stack(child_envid, argv, &esp)) < 0) {
        writef("init_stack() failed\n");
        return r;
    }
    /* 开始加载二进制文件 */
    u_char *ptr_ph_table = NULL;
    Elf32_Half ph_entry_count;
    Elf32_Half ph_entry_size;
    ptr_ph_table = (u_char *)elf + elf->e_phoff;
    ph_entry_count = elf->e_phnum;
    ph_entry_size = elf->e_phentsize;
    /* 使用之前完成的函数，加载二进制文件 */
    for(i = 0;i < ph_entry_count;i++){
        ph = (Elf32_Phdr *)ptr_ph_table;
        if(ph -> p_type == PT_LOAD) {
            r = usr_load_elf(fd, ph, child_envid);
            if( r < 0){
                return r;
```

```
            }
        }
        ptr_ph_table += ph_entry_size;
    }
    ......
}
```

这个函数要写的东西真的很多，看了半天也不知道从哪开始写起，和以前知识的联系也比较大，是一个很麻烦的函数。不过对其功能进行逐个剖析，也能够勉强理解。可以看到，我们的工作主要对应了`spawn()`函数的前四步。

Thinking 6.5

可以看到，我们实现的`usr_load_elf()`函数中，在结束了对其他段的加载后，有这么一段代码：

```
while (i < sgsize) {
    r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R);
    if(r < 0) return r;
    i += BY2PG;
}
```

这段代码就是用于实现`.bss`段的声明的，不断声明页面至`sgsize`。lab3中的`load_icode_mapper()`函数也是这样实现这一功能的，并且其在声明页面的时候页面会被自动填充为0，此处应当也是类似的。

Thinking 6.6

回顾`exercise 3.13`，在`./tools/secs0_3.lsd`文件下，我们会发现我们已经完成了对MOS操作系统中二进制文件各个段的地址的规定，，故而`*.b`文件的`text`段偏移值都是一样的。

2.解释Shell命令

接下来，我们将在shell进程中实现对管道的重定向和解释功能。搬用指导书的解释shell命令的规则：

- 如果碰到重定向符号`<`或者`>`,则读下一个单词，打开这个单词所代表的文件，然后将其复制给标准输入或者标准输出
- 如果碰到管道符号`|`， 则首先需要建立管道`pipe`，然后`fork()`。对于父进程，需要将管道的写者复制给标准输出，然后关闭父进程的读者和写者，运行`|`左边的命令，获得输出，然后等待子进程运行。对于子进程，将管道的读者复制给标准输入，从管道中读取数据，然后关闭子进程的读者和写者，继续读下一个单词。

我们直接通过代码来观察Shell命令的解析方式。

```
    ......
    for(;;){
        c = gettoken(0, &t);
        switch(c){
        case 0:
            goto runit;
        case 'w':
            if(argc == MAXARGS){
```

```
                writef("too many arguments\n");
                exit();
            }
            argv[argc++] = t;
            break;
        case '<':
            /* 读下一个单词 */
            if(gettoken(0, &t) != 'w'){
                writef("syntax error: < not followed by word\n");
                exit();
            }
            /* 打开下一个单词所代表的文件 */
            if((r = open(t, O_RDONLY)) < 0) {
                writef("This is '<', failed when open t...\n");
                exit();
            }
            fd = r;
            /* 复制到标准输入 */
            dup(fd, 0);
            close(fd);
            break;
            /* 对>的处理相同，仅仅是dup()的目的地不同 */
        ......
            case '|':
            /* 建立管道 */
            if((r = pipe(p)) < 0) {
                writef("| not implemented (pipe)\n");
                exit();
            }
            /* fork(), 创建子进程 */
            if((r = fork()) < 0) {
                writef("| not implemented (fork)\n");
                exit();
            }
            if(r == 0) {
                /* 子进程的相关操作 */
                dup(p[0], 0);
                close(p[0]);
                close(p[1]);
                goto again;
            } else {
                /* 父进程的相关操作 */
                dup(p[1], 1);
                close(p[0]);
                close(p[1]);
                rightpipe = r ;
                goto runit;
            }
            //user_panic("| not implemented");
```

```
            break;
        }
        ......
```

Shell解释这一部分的内容还相对简单，作为我们的MOS操作系统的收尾部分，也是比较令人感到愉快的了。

Thinking 6.7

在`user/init.c`中，调用了这一段代码：

```
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);

int opencons(void)
{
    int r;
    struct Fd *fd;

    if ((r = fd_alloc(&fd)) < 0)
        return r;
    if ((r = syscall_mem_alloc(0, (u_int)fd, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        return r;
    fd->fd_dev_id = devcons.dev_id;
    fd->fd_omode = O_RDWR;
    return fd2num(fd);
}
```

`opencons()`函数拿取了0号文件描述符，并且将其映射到`console`设备上，并且只赋只读权限。在后面的`dup()`函数中，应当是占用了1号占位符，这样就完成了对标准输入输出的"安排"。