Lab 6 Report

思考题

Thinking 6.1

父进程作为“读者”，需要关闭写端，而子进程作为“写者”，需要关闭读端。调换两者的语句即可。

```
switch (fork()) {
    case -1:
        break;

    case 0:     // 子进程
        close(fildes[0]); /* 关闭不用的读端*/
        write(fildes[1], "Hello world\n", 12); /* 向管道中写数据*/
        close(fildes[1]); /* 写入结束，关闭写端*/
        exit(EXIT_SUCCESS);

    default:    // 父进程
        close(fildes[1]); /* 关闭不用的写端*/
        read(fildes[0], buf, 100); /* 从管道中读数据*/
        printf("child-process read:%s",buf); /* 打印读到的数据*/
        close(fildes[0]); /* 读取结束，关闭读端*/
        exit(EXIT_SUCCESS);
}
```

Thinking 6.2

在两次syscall_mem_map()的间隙进行**进程切换**，fd的引用次数偏大，导致pageref(pipe) == pageref(fd)，系统认为管道已关闭，从而导致错误。

Thinking 6.3

系统调用是**原子操作**。因为系统调用会触发中断，系统跳转到**异常处理程序**，在处理异常时**关闭中断**，直到系统调用执行完毕。

Thinking 6.4

先执行`syscall_mem_unmap(0, fd);`，可以解决。

使得fd引用次数的-1先于pipe，在两个unmap的间隙仍有：pageref(pipe) > pageref(fd)，保证判断**管道是否关闭**的正确性。

dup()中若使得fd引用次数的+1先于pipe，会导致：pageref(pipe) == pageref(fd)，判断**管道是否关闭**的正确性受到影响。

Thinking 6.5

bss段的大小在ELF文件的头部信息中进行描述，但实际的数据在文件中没有存储。可根据p_memsz字段获取bss段在内存中的大小(sgsize)：`size_t sgsize = ph->p_memsz;`

Thinking 6.6

```
// stdin should be 0, because no file descriptors are open yet
if ((r = opencons()) != 0) {
    user_panic("opencons: %d", r);
}
// stdout
if ((r = dup(0, 1)) < 0) {
    user_panic("dup: %d", r);
}
```

- 使用opencons()函数打开控制台，返回值为0，表示**标准输入**。
- 使用dup()函数将标准输入的文件描述符0复制到标准输出的文件描述符1上。

Thinking 6.7

- MOS中的shell命令是**外部命令**，需要创建子进程执行。
- Linux中的cd命令被实现为shell的**内置命令**，执行时不需要创建子进程。这是因为cd命令是用于改变当前工作目录的命令，它需要在当前进程的环境中修改工作目录，而不是在一个新的子进程中执行。

Thinking 6.8

- 3次spawn
  - 00004803：ls.b进程
  - 00005805：cat.b进程

    - 00006006：motd输出重定向进程
- 3次进程销毁
    - 00005805：cat.b进程
    - 00006006：motd输出重定向进程
    - 00005004：pipecreate进程

```
$ ls.b | cat.b > motd
[00004803] pipecreate
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...
[00006006] destroying 00006006
[00006006] free env 00006006
i am killed ...
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
[00004803] destroying 00004803
[00004803] free env 00004803
i am killed ...
```

这些信息表明在执行命令"ls.b | cat.b > motd"时，首先通过pipecreate**创建管道**，然后通过spawn分别创建了ls.b、cat.b和motd输出重定向进程。最后，观察到这些进程被销毁并释放了相关资源。

难点分析

6.1 pipe_read() & pipe_write()

- 管道空：p->p_rpos >= p->p_wpos
- 管道满：p->p_wpos - p->p_rpos >= BY2PIPE

操作位置：p->p_buf[p->**p_rpos** % BY2PIPE] (读) // p->p_buf[p->**p_wpos** % BY2PIPE] (写)

6.5 parsecmd()

fork()函数用于创建子进程，返回值为0时表示**子进程**，父进程的返回值为**子进程的id**。

```
if ((*rightpipe = fork()) == 0) {    // child-right
    dup(p[0], 0);
    close(p[0]);
    close(p[1]);
    return parsecmd(argv, rightpipe);
} else {                             // parent-left
    dup(p[1], 1);
    close(p[0]);
    close(p[1]);
    return argc;
}
```

递归调用parsecmd(argv, rightpipe)，用于继续解析剩余的命令。

实验体会

本次实验包括了管道和shell的实现，通过实现这两个功能，我对操作系统的**进程管理**有了更深入的理解。在实现管道的过程中，需要考虑管道的读写操作，以及管道的创建和销毁。我了解到shell其实是一个**命令解析器**，它接收用户命令，然后调用相应的应用程序，最后将应用程序的输出结果显示给用户。shell相当于搭载在操作系统上的一个**软件**，很好地使用了之前完成的内核中文件系统、进程管理、内存管理等基本功能，作为本次课程的最后一次实验，形成了一个完整的可交互操作系统体系。