CHAPTER 6

管道与 SHELL

6.1 实验目的

1. 掌握管道的原理与底层细节
2. 实现管道的读写
3. 复述管道竞争情景
4. 实现基本 shell
5. 实现 shell 中涉及管道的部分

6.2 管道

在 lab4 中，我们已经学习过一种进程间通信 (IPC,Inter-Process Communication) 的方式——共享内存。而今天我们要学的管道，其实也是进程间通信的一种方式。

6.2.1 初窥管道

通俗来讲，管道就像家里的自来水管：一端用于注入水，一端用于放出水，且水只能在一个方向上流动，而不能双向流动，所以说管道是典型的单向通信。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。

在 Unix 中，管道由 pipe 函数创建，函数原型如下：

```
1  ^^I#include<unistd.h>
2  ^^I
3  ^^Iint      pipe(int fd[2]); 成功返回0，否则返回-1；
4  ^^I
5  ^^I 参数 fd 返回两个文件描述符，fd[0]对应读端，fd[1]对应写端。
```

为了更好地理解管道实现的原理，同样，我们先来做实验亲自体会一下[1]

[1]实验代码参考 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

Listing 16: 管道示例

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

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main(){

    status = pipe(fildes);

    if (status == -1 ) {
        /* an error occurred */
        printf("error\n");
    }


    switch (fork()) {
    case -1: /* Handle error */
        break;


    case 0:  /* Child - reads from pipe */
        close(fildes[1]);                       /* Write end is unused */
        read(fildes[0], buf, 100);              /* Get data from pipe */
        printf("child-process read:%s",buf);    /* Print the data */
        close(fildes[0]);                       /* Finished with pipe */
        exit(EXIT_SUCCESS);


    default:  /* Parent - writes to pipe */
        close(fildes[0]);                       /* Read end is unused */
        write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
        close(fildes[1]);                       /* Child will see EOF */
        exit(EXIT_SUCCESS);
    }
}
```

示例代码实现了从父进程向子进程发送消息”Hello,world”，并且在子进程中打印到屏幕上。它演示了管道在父子进程之间通信的基本用法：在 pipe 函数之后，调用 fork 来产生一个子进程，之后在父子进程中执行不同的操作。在示例代码中，父进程操作写端，而子进程操作读端。同时，示例代码也为我们演示了使用 pipe 系统调用的习惯：fork 之后，进程在开始读或写管道之前都会关掉不会用到的管道端。

从本质上说，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。

Thinking 6.1 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？ ■

6.2.2 管道的测试

我们下面就来填充函数实现匿名管道的功能。思考刚才的代码样例，要实现匿名管道，至少需要有两个功能：管道读取、管道写入。

要想实现管道，首先我们来看看本次实验我们将如何测试。lab6 关于管道的测试有两个，分别是`user/testpipe.c`与`user/testpiperace.c`。

首先我们来观察 testpipe 的内容

Listing 17: testpipe 测试

```
1  #include "lib.h"
2
3
4  char *msg =
5          "Now is the time for all good men to come to the aid of their party.";
6
7  void
8  umain(void)
9  {
10         char buf[100];
11         int i, pid, p[2];
12
13         if ((i = pipe(p)) < 0) {
14                 user_panic("pipe: %e", i);
15         }
16
17         if ((pid = fork()) < 0) {
18                 user_panic("fork: %e", i);
19         }
20
21         if (pid == 0) {
22                 writef("[%08x] pipereadeof close %d\n", env->env_id, p[1]);
23                 close(p[1]);
24                 writef("[%08x] pipereadeof readn %d\n", env->env_id, p[0]);
25                 i = readn(p[0], buf, sizeof buf - 1);
26
27                 if (i < 0) {
28                         user_panic("read: %e", i);
29                 }
30
31                 buf[i] = 0;
32
33                 if (strcmp(buf, msg) == 0) {
34                         writef("\npipe read closed properly\n");
35                 } else {
36                         writef("\ngot %d bytes: %s\n", i, buf);
37                 }
38
39                 exit();
40         } else {
```

```
41                  writef("[%08x] pipereadeof close %d\n", env->env_id, p[0]);
42                  close(p[0]);
43                  writef("[%08x] pipereadeof write %d\n", env->env_id, p[1]);
44
45                  if ((i = write(p[1], msg, strlen(msg))) != strlen(msg)) {
46                          user_panic("write: %e", i);
47                  }
48
49                  close(p[1]);
50          }
51
52          wait(pid);
53
54          if ((i = pipe(p)) < 0) {
55                  user_panic("pipe: %e", i);
56          }
57
58          if ((pid = fork()) < 0) {
59                  user_panic("fork: %e", i);
60          }
61
62          if (pid == 0) {
63                  close(p[0]);
64
65                  for (;;) {
66                          writef(".");
67
68                          if (write(p[1], "x", 1) != 1) {
69                                  break;
70                          }
71                  }
72
73                  writef("\npipe write closed properly\n");
74          }
75
76          close(p[0]);
77          close(p[1]);
78          wait(pid);
79
80          writef("pipe tests passed\n");
81  }
```

实际上可以看出，测试文件使用 pipe 的流程和示例代码是一致的。先使用函数 `pipe(int p[2])`创建了管道，读端的文件控制块编号[2]为 p[0]，写端的文件控制块编号为 p[1]。之后使用 fork() 创建子进程，**注意这时父子进程使用 p[0] 和 p[1] 访问到的内存区域一致**。之后子进程关闭了 p[1]，从 p[0] 读；父进程关闭了 p[0], 从 p[1] 写入管道。

lab4 的实验中，我们的 fork 实现是完全遵循 Copy-On-Write 原则的，即对于所有用户态的地址空间都进行了 PTE_COW 的设置。但实际上写时复制并不完全适用，至少在我们当前情景下是不允许写时拷贝。为什么呢？我们来看看 pipe 函数中的关键部分就能知晓答案：

[2]文件控制块编号是 int 型，user/fd.c 中 num2fd 函数可通过它定位文件控制块的地址。

```
int
pipe(int pfd[2])
{
^^int r, va;
^^struct Fd *fd0, *fd1;
^^
^^if ((r = fd_alloc(&fd0)) < 0
^^||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
^^goto err;
^^
^^if ((r = fd_alloc(&fd1)) < 0
^^||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
^^goto err1;
^^
^^va = fd2data(fd0);
^^if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
^^goto err2;
^^if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) < 0)
^^goto err3;
^^
^^...
}
```

在 pipe 中，首先分配两个文件描述符并为其分配空间，然后将一个管道作为这两个文件描述符数据区的第一页数据，从而使得这两个文件描述符能够共享一个管道的数据缓冲区。

Exercise 6.1 仔细观察 pipe 中新出现的权限位PTE_LIBRARY，根据上述提示修改 fork 系统调用，使得**管道缓冲区是父子进程共享的**，不设置为写时复制的模式。 ■

下面我们使用一张图来表示父子进程与管道的数据缓冲区的关系：

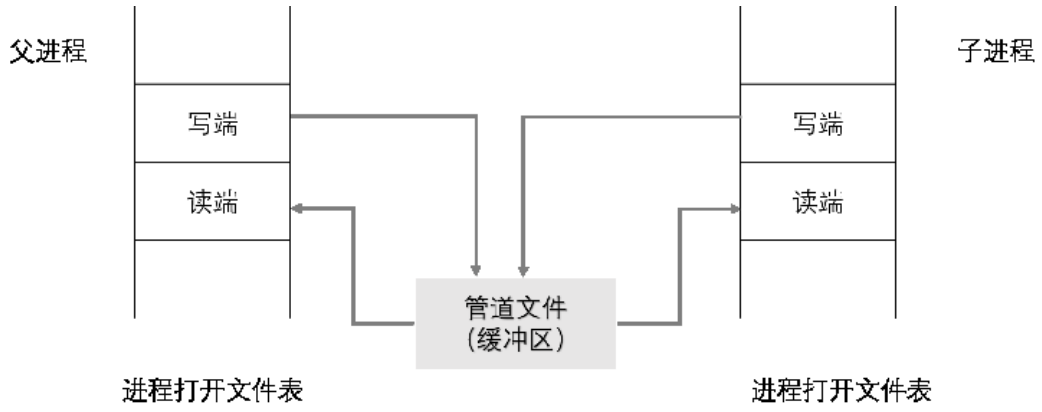

图 6.1: 父子进程与管道缓冲区

实际上，在父子进程中各自 close 掉不再使用的端口后，父子进程与管道缓冲区的关系如下图：

下面我们来讲一下`struct Pipe`，并开始着手填写操作管道端的函数。

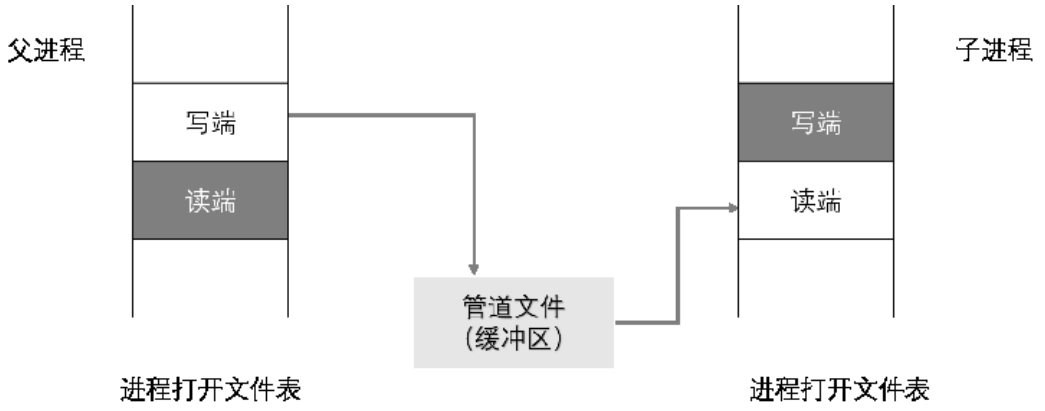

图 6.2: 关闭不使用的端口后

6.2.3 管道的读写

我们可以在 user/pipe.c 中轻松地找到 Pipe 结构体的定义，它的定义如下：

```
1 ^^Istruct Pipe {
2 ^^I^^Iu_int p_rpos;^^I^^I    // read position
3 ^^I^^Iu_int p_wpos;^^I^^I    // write position
4 ^^I^^Iu_char p_buf[BY2PIPE];^^I// data buffer
5 ^^I};
```

在 Pipe 结构体中,p_rpos 给出了下一个将要从管道读的数据的位置，而 p_wpos 给出了下一个将要向管道写的数据的位置。只有读者可以更新 p_rpos，同样，只有写者可以更新 p_wpos，读者和写者通过这两个变量的值进行协调读写。一个管道有 BY2PIPE(32Byte) 大小的缓冲区。

这个只有 BY2PIPE 大小的缓冲区发挥的作用类似于环形缓冲区，所以下一个要读或写的位置 i 实际上是 i%BY2PIPE。

读者在从管道读取数据时，要将 p_buf[p_rpos%BY2PIPE] 的数据拷贝走，然后读指针自增 1。但是需要注意的是，管道的缓冲区此时可能还没有被写入数据。所以如果管道数据为空，即当 p_rpos >= p_wpos 时，应该进程切换到写者运行。

类似于读者，写者在向管道写入数据时，也是将数据存入 p_buf[p_wpos%BY2PIPE]，然后写指针自增 1。需要注意管道的缓冲区可能出现满溢的情况，所以写者必须得在 p_wpos - p_rpos < BY2PIPE 时方可运行，否则要一直挂起。

上面这些还不足以使得读者写者一定能顺利完成管道操作。假设这样的情景：管道写端已经全部关闭，读者读到缓冲区有效数据的末尾，此时有 p_rpos = p_wpos。按照上面的做法，我们这里应当切换到写者运行。但写者进程已经结束，进程切换就造成了死循环，这时候读者进程如何知道应当退出了呢？

为了解决上面提出的问题，我们必须得知道管道的另一端是否已经关闭。不论是在读者还是在写者中，我们都需要对另一端的状态进行判断：当出现缓冲区空或满的情况时，要根据另一端是否关闭来判断是否要返回。如果另一端已经关闭，进程返回 0 即可；如果没有关闭，则切换到其他进程运行。

Note 6.2.1 如果管道的写端相关的所有的文件描述符都已经关闭，那么管道读端将会读到文件结尾并返回 0。link：http://linux.die.net/man/7/pipe

那么我们该如何知晓管道的另一端是否已经关闭了呢？这时就要用到我们的`static int _pipeisclose`数。而这个函数的核心，就是下面我们要讲的恒成立等式了。

在之前的图6.2中我们没有明确画出文件描述符所占的页，但实际上，对于每一个匿名管道而言，我们分配了三页空间：一页是读数据的文件描述符 rfd，一页是写数据的文件描述符 wfd，剩下一页是被两个文件描述符共享的管道数据缓冲区。既然管道数据缓冲区 h 是被两个文件描述符所共享的，我们很直观地就能得到一个结论：如果有 1 个读者，1 个写者，那么管道将被引用 2 次，就如同上图所示。pageref 函数能得到页的引用次数，所以实际上有下面这个等式成立：

pageref(rfd) + pageref(wfd) = pageref(pipe)

Note 6.2.2 内核会对 pages 数组成员维护一个页引用变量 pp_ref 来记录指向该物理页的虚页数量。pageref 的实现实际上就是查询虚页 P 对应的实际物理页，然后返回其 pp_ref 变量的值。

这个等式对我们而言有什么用呢？假设我们现在在运行读者进程，而进行管道写入的进程都已经结束了，那么此时就应该有：`pageref(wfd) = 0`。所以就有`pageref(rfd) = pageref(pipe)`。所以我们只要判断这个等式是否成立就可以得知写端是否关闭，对写者来说同理。

Exercise 6.2 根据上述提示与代码中的注释，填写 user/pipe.c 中的 piperead、pipewrite、_pipeisclosed 函数并通过 testpipe 的测试。 ■

Note 6.2.3 注意在本次实验中由于文件系统服务所在进程已经默认为 1 号进程 (起始进程为 0 号进程)，在测试时想启用文件系统需要注意 ENV_CREATE(fs_serv) 在 init.c 中的位置。

6.2.4 管道的竞争

我们的小操作系统采用的是时间片轮转调度的进程调度算法，这点你应该在 lab3 中就深有体会了。这种抢占式的进程管理就意味着，用户进程随时有可能会被打断。

当然，如果进程间是孤立的，随时打断也没有关系。但当多个进程共享同一个变量时，执行同一段代码，不同的进程执行顺序有可能产生完全不同的结果，造成运行结果的不确定性。而进程通信需要共享 (不论是管道还是共享内存)，所以我们要对进程中共享变量的读写操作有足够高的警惕。

实际上，因为管道本身的共享性质，所以在管道中有一系列的竞争情况。在当前这种不加锁控制的情况下，我们无法保证`_pipeisclosed`用于管道另一端关闭的判断一定返回正确的结果。

我们重新看之前写的`_pipeisclosed`函数。在这个函数中我们对`pageref(fd structure)`与`pageref(pipe structure)`进行了等价关系的判断。假如不考虑进程竞争，不论是在读

者还是写者进程中，我们会认为:

- 对 fd 和对 pipe 的 pp_ref 的**写入**是同步的。
- 对 fd 和对 pipe 的 pp_ref 的**读取**是同步的。

但现在我们处于进程竞争、执行顺序不定的情景下，上述两种情况现在都会出现不同步的现象。想想看，如果在下面这种场景下，我们前面提到的等式6.2.3还是恒成立的吗:

```
^^Ipipe(p);
^^Iif(fork() == 0 ){
^^I^^Iclose(p[1]);
^^I^^Iread(p[0],buf,sizeof buf);
^^I}else{
^^I^^Iclose(p[0]);
^^I^^Iwrite(p[1],"Hello",5);
^^I}
```

- fork 结束后，子进程先执行。时钟中断产生在 close(p[1]) 与 read 之间，父进程开始执行。
- 父进程在 close(p[0]) 中，p[0] 已经解除了对 pipe 的映射 (unmap)，还没有来得及解除对 p[0] 的映射，时钟中断产生，子进程接着执行。
- 注意此时各个页的引用情况：pageref(p[0]) = 2(因为父进程还没有解除对 p[0] 的映射)，而 pageref(p[1]) = 1(因为子进程已经关闭了 p[1])。但注意，此时 pipe 的 pageref 是 2，子进程中 p[0] 引用了 pipe，同时父进程中 p[0] 刚解除对 pipe 的映射，所以在父进程中也只有 p[1] 引用了 pipe。
- 子进程执行 read，read 中首先判断写者是否关闭。比较 pageref(pipe) 与 pageref(p[0]) 之后发现它们都是 2，说明写端已经关闭，于是子进程退出。

Thinking 6.2 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？ ■

那看到这里你有可能会问：在 close 中，既然问题出现在两次 unmap 之间，那么我们为什么不能使两次 unmap 统一起来是一个原子操作呢？要注意，在我们的小操作系统中，只有 syscall_ 开头的**系统调用函数**是原子操作，其他所有包括 fork 这些函数都是可能会被打断的。一次系统调用只能 unmap 一页，所以我们是不能保持两次 unmap 为一个原子操作的。那是不是一定要两次 unmap 是原子操作才能使得`_pipeisclosed`一定返回正确结果呢？

Thinking 6.3 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。 ■

答案当然是否定的，`_pipeisclosed`函数返回正确结果的条件其实只是：

- 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);

比如说第一个条件，写端关闭时，当然有 pageref(p[0]) == pageref(pipe)。所以我们要解决的实际上是 **当 pageref(p[0]) == pageref(pipe) 时，写端关闭**。正面如果不好解决问题，我们可以考虑从其逆否命题着手，即要满足：当写端没有关闭的时候，pageref(p[0]) ≠ pageref(pipe)。

我们考虑之前那个预想之外的情景，它出现的最关键原因在于：pipe 的引用次数总比 fd 要高。当管道的 close 进行到一半时，**若先解除 pipe 的映射，再解除 fd 的映射**，就会使得 pipe 的引用次数的-1 先于 fd。这就导致在两个 unmap 的间隙，会出现 pageref(pipe) == pageref(fd) 的情况。那么若调换 fd 和 pipe 在 close 中的 unmap 顺序，能否解决这个问题呢？

Thinking 6.4 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。
- 我们只分析了 close 时的情形，那么对于 dup 中出现的情况又该如何解决？请模仿上述材料写写你的理解。

■

根据上面的描述我们其实已经能够得出一个结论：控制 fd 与 pipe 的 map/unmap 的顺序可以解决上述情景中出现的进程竞争问题。

那么下面根据你所思考的内容进行实践吧：

Exercise 6.3 修改 user/pipe.c 中的 pipeclose 与 user/fd.c 中的 dup 函数以避免上述情景中的进程竞争情况。 ■

6.2.5 管道的同步

我们通过控制修改 pp_ref 的前后顺序避免了“写数据”导致的错觉，但是我们还得解决第二个问题：读取 pp_ref 的同步问题。

同样是上面的代码6.2.4，我们思考下面的情景：

- fork 结束后，子进程先执行。执行完 close(p[1]) 后，执行 read，要从 p[0] 读取数据。但由于此时管道数据缓冲区为空，所以 read 函数要判断父进程中的写端是否

关闭，进入到 _pipeisclosed 函数，pageref(fd) 值为 2(父进程和子进程都打开了 p[0])，时钟中断产生。

- 内核切换到父进程执行，父进程 close(p[0])，之后向管道缓冲区写数据。要写的数据较多，写到一半时钟中断产生，内核切换到子进程运行。
- 子进程继续运行，获取到 pageref(pipe) 值为 2(父进程打开了 p[1], 子进程打开了 p[0])，引用值相等，于是认为父进程的写端已经关闭，子进程退出。

上述现象出现的根源在哪里呢？fd 是一个父子进程共享的变量，但子进程中的 pageref(fd) 没有随父进程对 fd 的修改而同步，这就造成了子进程读到的 pageref(fd) 成为了“脏数据”。为了保证读的同步性，子进程应当重新读取 pageref(fd) 和 pageref(pipe)，并且要在**确认两次读取之间进程没有切换**后，才能返回正确的结果。为了实现这一点，我们要使用到之前一直都没用到的变量：env_runs。

env_runs 记录了一个进程 env_run 的次数，这样我们就可以根据某个操作 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。

Exercise 6.4 根据上面的表述，修改`_pipeisclosed`函数，使得它满足” 同步读” 的要求。注意 env_runs 变量是需要维护的。 ■

6.3 shell

首先恭喜屏幕前的你能够读到这里，你的 OS 大厦即将落成，但所谓” 行百里者半九十”，也许你此时会觉得整个系统的代码过多，OS 的大厦摇摇欲坠，每一次 DEBUG 都因找不到问题出处而心力憔悴。当你满怀希望地 make run，却发现屏幕上无限循环的 page_out 或一行简洁的 user_panic，是否让你欲哭无泪。要知道，坚持就是胜利！我们能做的，就是尽可能地引导大家正确思考，并” 正确地” 实现接下来的部分–shell。

Let’s go!

6.3.1 完善 spawn 函数

spawn 的汉译为” 产卵”，其作用是帮助我们调用文件系统中的可执行文件并执行。spawn 的流程可以分解如下：

- 从文件系统打开对应的文件 (2 进制 ELF，在我们的 OS 里是 *.b)。
- 申请新的进程描述符；
- 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。

在动手填写 spawn 函数前，我们希望你：

- 认真回看 lab5 文件系统相关代码，弄清打开文件的过程。
- 思考如何读取 elf 文件或者说如何知道二进制文件中 text 段

关于后一个问题，各位已经在 lab3 中填写了 load_icode 函数，实现了 elf 中读取数据并写入内存空间。而 text 段的位置，可以借助 readelf 的帮助（这个命令各位应该不陌生，笔者在写这部分指导书时，慨叹各位 lab1 的 exam 确实有点难 233)。

为了确保各位的大楼经得起反复折腾，我们来探讨下面的问题，以检验大家的基本功，以及前几个 lab 是否做得” 到位”，lab5 、lab3 和 lab1 的 “二进宫”：

(如果不到位，请回炉重看一遍并加以理解)

- 在 lab1 中我们介绍了 data text bss 段及它们的含义，data 存放初始化过的全局变量，bss 存放未初始化的全局变量。关于 memsize 和 filesize ，我们在 Note1.3.3 中也解释了它们的含义与特点。关于 1.3.3，注意其中关于”bss 并不在文件中占数据” 的表述。
- 在 lab3 中我们创建进程，并且在内核态加载了初始进程 (ENVCREATE(...)), 而我们的 spawn 函数则是通过和文件系统交互，取得文件描述块，进而找到 elf 在” 硬盘” 中的位置，进而读取。

在 lab3 中我们要填写 load_icode_mapper 函数，分为两部分填写，第二部分则是处理 msize 和 fsize 不相等时的情况。那么问题来了：

Thinking 6.5 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？ ■

如果你回看了上面提到的指导书内容，答案应该是” 显而易见的”。如果还不太理解，说明你需要回炉重看一遍，并实践一下下面的” 补课资料”。

6.3.2 解释 shell 命令

接下来，我们将通过一个实例，再次吸收理解上面 lab1、lab3 联动的内容。

```
#include "lib.h"
#define ARRAYSIZE (1024*10)
int bigarray[ARRAYSIZE]={0};
void
umain(int argc, char **argv)
{
int i;

writef("Making sure bss works right...\n");
    for(i = 0;i < ARRAYSIZE; i++)
        if(bigarray[i]!=0)
            user_panic("bigarray[%d] isn't cleared!\n",i);
    for (i = 0; i < ARRAYSIZE; i++)
```

```
14              bigarray[i] = i;
15          for (i = 0; i < ARRAYSIZE; i++)
16              if (bigarray[i] != i)
17                  user_panic("bigarray[%d] didn't hold its value!\n", i);
18              writef("Yes, good. Now doing a wild write off the end...\n");
19              bigarray[ARRAYSIZE+1024] = 0;
20          userpanic("SHOULD HAVE TRAPPED!!!");
21      }
```

在 user/文件夹下创建上面的文件，并在 Makefile 中添加相应信息，使得生成相应的.b 文件，在 init/init.c 中创建相应的初始进程，观察相应的实验现象。(具体可以参考 lab4 中 pingpong 和 fktest 是如何添加到 makefile 中的) 如果能正确运行，则说明我们的 load_icode 系列函数正确地保证了 bss 段的初始化。

使用 readelf 命令解析我们的 testbss.b 文件，看看 bss 段的大小并分析其原因。学习并使用 size 命令，看看我们的 testbss.b 文件的大小。

修改代码，将数组初始化为 array[SIZE]=0; 重新编译（记得常 make clean），再次分析 bss 段。再次使用 size 命令。

再次修改代码，将数组初始化为 array[SIZE]=1; 再次重新编译，再次分析。

在 Lab5 中我们实现了文件系统，lab6 的 shell 部分我们提供了几个可执行二进制文件，模拟 linux 的命令：ls.b cat.b .....。上面提到的 spawn 函数实现方法，是打开相应的文件并执行。请你思考我们是如何将文件” 烧录” 到 fs.img 中的（阅读 fs/Makefile)。如果你并没有看懂我这段话，请回看 Exercise5.4

你可以尝试将生成的 testbss.b 加载进 fs.img 中

这节” 补课” 下课以后，大家再去补全 spawn 函数，相信能如鱼得水了。

Thinking 6.6 为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值？ ■

接下来，我们需要在 shell 进程里实现对管道和重定向的解释功能。解释 shell 命令时：

1. 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。
2. 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。

在这里可以举一个使用管道符号的例子来方便大家理解，相信大家都使用过 linux 中的 ps 指令，也就是最基本的查看进程的命令，而直接使用 ps 会看到所有的进程，为了更方便的追踪某个进程，我们通常使用 ps aux|grep xxx 这条指令，这就是使用管道的例子，ps aux 命令会将所有的进程按格式输出，而 grep xxx 命令作为子进程执行，所有的进程作为他的输入，最后的输出将会筛选出含有 xxx 字符串的进程展示在屏幕上。

Exercise 6.5 根据以上描述，补充完成 user/sh.c 中的 `void runcmd(char *s)`。 ■

通过阅读代码空白段的注释我们知道，将文件复制给标准输入或输出，需要我们将其 dup 到 0 或 1 号文件描述符 (fd). 那么问题来了：

Thinking 6.7 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。 ■

Note 6.3.1 我们的测试进程从 user/icode 开始执行，里面调用了 spawn(init.b), 在完成了 spawn 后，创建了 init.b 进程.....

6.4 实验正确结果

6.4.1 管道测试

管道测试有三个文件,分别是 user/testpipe.c 、user/testpiperace.c 和 user/testelibrary.c ，以合适的次序建好进程后，在 testpipe 的测试中若出现两次 **pipe tests passed** 即说明测试通过。

testpipe 本地测试部分运行结果如图:

图 6.3: 管道测试 1

在 testpiperace 的测试中应当出现 race didn’t happen 是正确的。

testpiperace 本地测试部分运行结果如图:

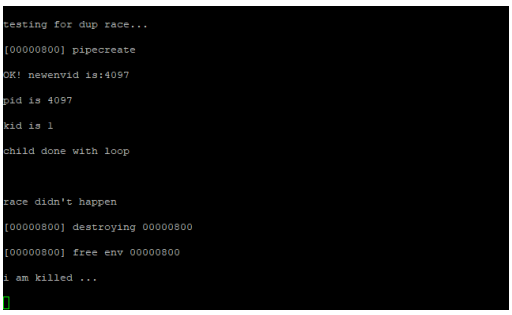

图 6.4: 管道测试 2

在 user/testptelibrary.c 的测试中，如果 fork 和 spawn 对于共享页面的处理均可得

当，即可说明测试正确。

6.4.2 shell 测试

在 init/init.c 中按照如下顺序依次启动 shell 和文件服务：

```
1 ^^IENV_CREATE(user_icode);
2 ^^IENV_CREATE(fs_serv);
```

如果正常会看到如下现象：

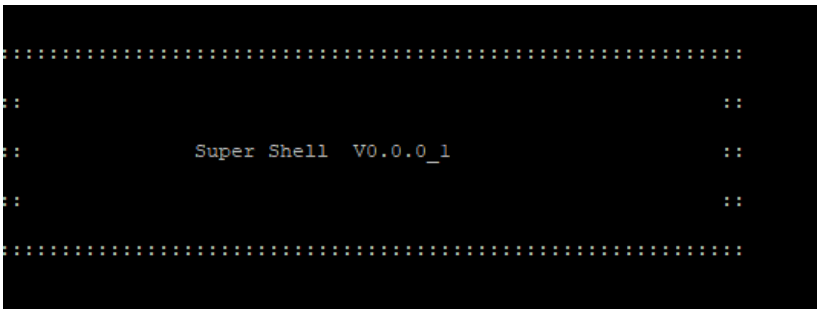

图 6.5: shell 展示界面

Note 6.4.1 Shell 部分评测提交，最好注释掉 writef(“:::⋯supershell⋯”) 部分内容。

使用不同的命令会有不同的效果：

- 输入 ls.b，会显示一些文件和文件夹;

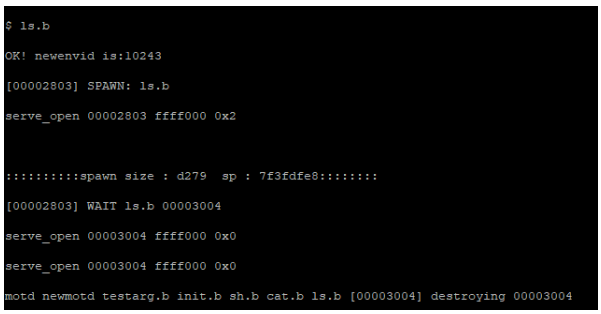

图 6.6: ls 结果

- 输入 cat.b，会有回显现象出现;

- 输入 ls.b | cat.b, 和 ls.b 的现象应当一致;

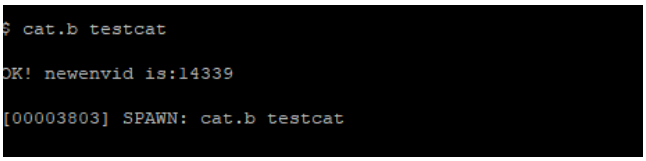

图 6.7: cat 结果

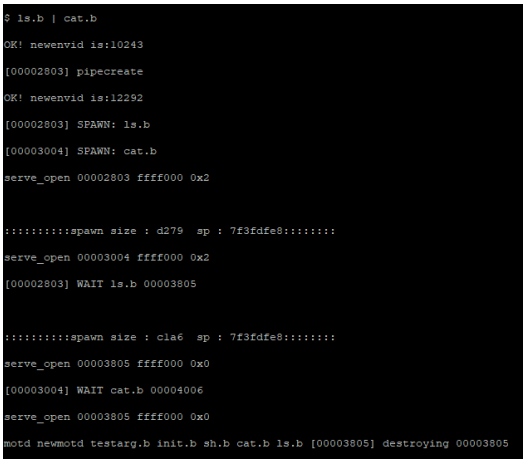

图 6.8: lscat 结果

Note 6.4.2 课程网站上有对于测试文件的解析视频，大家可以移步网站观看。

6.5 实验思考

- 思考-父进程为读者
- 思考-dup 中的进程竞争
- 思考-原子操作
- 思考-解决进程竞争
- 思考-spawn 中的内存共享

6.6 LAB6 挑战性任务

我们现在实现了一个简单的 shell。接下来，我们打一套“连续简单拳”：通过实现一些难度层次递进的小组件或附加功能，来丰富我们的 shell，加深对整个 buaa_os_lab6 的理解，让它更加强大且有逼格！

Note 6.6.1 前置要求：实现 lab5 fsformat 中“文件夹生成”的相关代码。

6.6.1 easy 任务部分

1、实现后台运行：

当我们在一个命令后增加 & 符号，代表 shell 不需要等待此命令执行完毕后再继续执行（阅读代码并思考，我们现在的 shell 的等待关系是什么样的）

2、实现一行多命令：

用 **；** 分开同一行内的两条命令

> **Note 6.6.2** 我们保留 symbol 里已经预留有 ‘;’ 和’&’ 字符

3、实现引号支持：

实现引号支持后，我们就可以处理如：echo.b ” xxx | xxx ” 这样的指令。

完成这项任务时，您可以仅实现强引用。

4、实现如下的命令：

tree mkdir

5、尝试实现清屏：

你可以通过监听 Ctrl+l 或者实现一个 clear 内建指令完成这个任务。

5、尝试彩色输出：

通过 putty/secureCRT 等终端工具 +ANIS 控制序列，我们可以控制彩色输出。如果你对这部分不是很了解，建议上网搜索一下” **字符编码 ANSI**”。或者，你可以打开任何一个 lab 的评测 Log，观察“彩色输出部分” 是如何实现的。

上述任务实现难度均不大。

6.6.2 Normal 任务部分

1、历史命令：

任务背景：

在 linux 下我们输入的 shell 命令都会被保存起来，并可以通过 up/down 键回溯指令，这极大地方便了我们……

任务目标：

实现保存在 shell 输入的指令，并可以通过 history.b 命令输出所有的历史指令。

任务要求：

第一部分要求我们将在 shell 中输入的每步指令，在解析前 (or 后) 保存进一个专用文件中 (如.history，每行一条指令)。第二部分通过写一个 UserAPP (history.b)，并写入磁盘中，使得每次调用 history.b 时，将文件 (.history) 的内容全部输出。

- 请注意，禁止使用局部变量或全局变量的形式实现保存历史指令。(这意味着，不能用堆栈区实现)
- 接上条，禁止在烧录 fs.img 时烧录一个.history 文件。
- 接上条，这也就意味着，你需要在第一次写入时，创建一个.history 文件，并在随后每次输入时，在.history 文件末尾写入

关键 NPC：

sh.c(总得在命令执行前后把 line 写进文件里吧)

serv.c 中 serve_open 的逻辑 (目前的 openfile 仅支持打开文件, 完成 easy 部分内容后, 加上了对 O_MKDIR 的支持, 现在, 需要加上对 O_CREATE 的支持) history.c(一个简单的 UserApp)

不起眼数据结构培育法 (Stat 数据结构)

6.6.3 exec 任务部分

任务背景:

在 lab6 的实验中，我们使用了 spawn 函数实现了增殖 (将目标程序加载为新的进程), 帮助我们实现了 shell。回顾 spawn，我们将**目标程序的相关信息**加载给新进程。

任务目标:

实现一个 execl、exec 函数组，使我们将目标程序加载到“本进程”

任务要求:

查找资料，理解并实现 exec。

！实际上，exec 并没有创建新的进程，它所做的是将原进程的**代码段、数据段、堆栈段等**用目标程序代替，但进程的 envid 并没有被替换。

关键 NPC:

spawn.c(师夷长技)

增加系统调用 (不能“左脚踩右脚”，借助内核态才能帮助我们将代码段替换)

3、实现 shell 环境变量:

实现 shell 变量, 支持 export [-xr] 导出环境变量和设置只读变量, 支持 unset 和 set 命令。

(建议用内建指令实现这三条指令的简易版本)

支持并在执行诸如 echo.b $variable 指令时能显示正确的值。

Hard?

本部分开放性很强，请任选 1 个实现。

1、实现“上/下”键切换历史命令 (或者 C^R 查找历史命令)

2、实现组命令, 如’ (ls.b ; echo.b ”xiaoming”) | cat.b > motd ’

实现上述任务后，请自行设计展示。