Lab6实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

thinking 6.1

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

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

int main(){

status = pipe(fildes);

if (status == -1 ) {
    printf("error\n");
}


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

    case 0: /* 子进程 - 作为管道的写者 */
        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，关闭读端 */
        exit(EXIT_SUCCESS);


    default: /* 父进程 - 作为管道的读者 */
        close(fildes[1]); /* 关闭不用的写端 */
        read(fildes[0], buf, 100);
        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

dup函数的作用是把旧文件描述符映射到新文件描述符，同时增加旧文件描述符和pipe的引用次数，并把旧文件描述符的引用次数给新文件描述符

假设我们需要将一个管道的读/写端的文件描述符（即旧文件描述符，假设为fd[0]）映射到另一个文件描述符（即新文件描述符，假设为fd[1]）。在映射前，fd[0]、fd[1]和pipe的引用次数分别是1, 1, 2。dup函数首先增加fd[0]的引用次数，如果在增加后，此时时钟中断发生并切换到另一进程，那么当这个进程调用_pipe_is_closed函数来检查管道写端是否关闭时，由于此时pageref(fd[0])和pageref(pipe)都等于2，会错误判断写端已经关闭

thinking 6.3

所谓系统调用的原子性，就是指系统调用在执行过程中的所有步骤都会作为独立操作一次性执行完毕，而不会被其他线程中断。原子操作是规避竞争状态的有效手段之一，处于竞争状态下的多个线程其执行结果是不确定的，这取决于CPU的调度，为了防止处于内核态的系统调用出现竞争，故而规定所有的系统调用都是原子操作，这本质上还是通过屏蔽中断来实现的

thinking 6.4

- 调换顺序后，先解除fd的映射，fd的引用次数先-1，而pipe的引用次数总比fd要高，所以即使在解除fd映射后发生时钟中断，pageref(pipe)>pageref(fd)的关系也不会变，不会影响判断管道是否关闭的正确性
- 在dup函数中，因为pipe的引用次数总比fd要高，如果先增加pipe的引用次数，增加后时钟中断发生，此时仍有pageref(pipe)>pageref(fd)，不会影响对管道是否关闭的判断

thinking 6.5

```
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data);
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm,
const void *src, size_t len);
```

elf_load_seg 函数会从 ph 中获取 va（该段需要被加载到的虚地址）、sgsize（该段在内存中的大小）、bin_size（该段在文件中的大小）和 perm（该段被加载时的页面权限），并根据这些信息完成以下两个步骤：

第一步 加载该段的所有数据（bin）中的所有内容到内存（va）

第二步 如果该段在文件中的内容的大小达不到为填入这段内容新分配的页面大小，即分配

了新的页面但没能填满（如 .bss 区域），那么余下的部分用 0 来填充

为了达到这一目标，elf_load_seg 的最后两个参数用于接受一个自定义的回调函数 map_page，以及需要传递给回调函数的额外参数 data。每当 elf_load_seg 函数解析到一个需要加载到内存中的页面，会将有关的信息作为参数传递给回调函数，并由它完成单个页面的加载过程，而这里 load_icode_mapper 就是 map_page 的具体实现

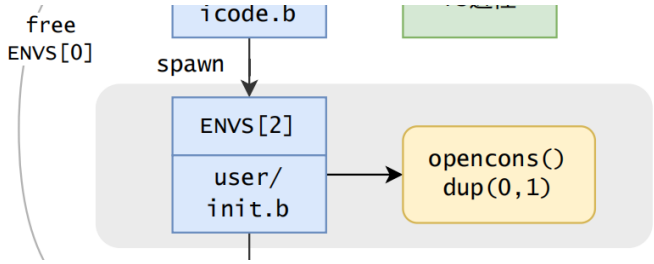
作为回调函数的 load_icode_mapper 函数，其中需要分配所需的物理页面，并在页表中建立映射。若 src 非空，还需要将该处的 ELF 数据拷贝到物理页面中

thinking 6.6

在user/init.c中

```
if ((r = opencons()) != 0) {
        user_panic("opencons: %d", r);
    }
    // stdout
if ((r = dup(0, 1)) < 0) {
        user_panic("dup: %d", r);
}
```

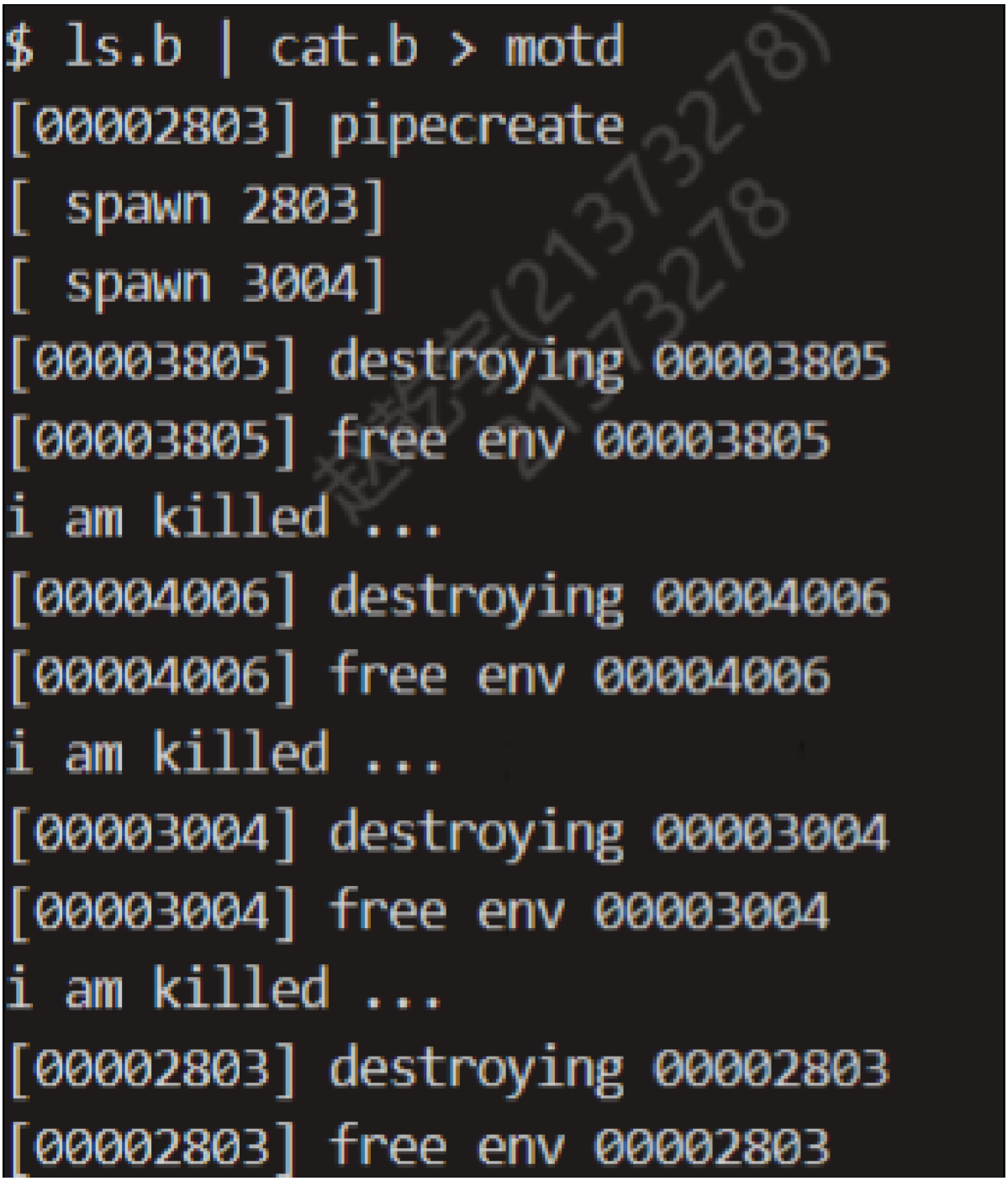
对应步骤



thinking 6.7

- 外部命令，因为shell 需要 fork 一个子 shell，然后子 shell 去执行这条命令。
- Linux 的 cd 指令是改变当前的工作目录，如果在子 shell 中执行，则改变的是子 shell 的工作目录

thinking 6.8



- 2次spawn，对应进程00002803和00003004
- 4次进程销毁，对应进程00003805，00004006，00003004，00002803

二、实验难点

在本实验中，笔者认为难点在管道的读写的实现和管道关闭的正确判断

管道的读写需要注意的就是读写过程中的判断，指导书中给出如下的提示

读者在从管道读取数据时，要将 `p_buf[p_rpos%BY2PIPE]` 中的数据拷贝走，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
我们必须知道管道的另一端是否已经关闭。不论是读者还是写者进程，我们都需要对管道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，则切换进程运行

而管道的关闭的正确判断则与进程竞争有关

因为不同步修改fd和pipe的页面引用次数而导致的进程竞争问题，反映为可能对管道关闭的判断出现错误，具体分析详见思考题6.2和6.4的内容

三、实验心得

最后一次课下os实验，也是整个os的最后一次实验，题量不多，但一如既往的有些磕磕绊绊。不过在debug中，也是对实验的一种回顾，找到错误并改正，实际加深了对那道题目的理解。

当然，本学期的os实验也有遗憾，课下没有把基础打牢打实，结果课上屡屡犯错，有时是对某个函数的使用理解不够深，有时是对某个地方的实现过程不清楚，每次都有一种近在咫尺而功亏一篑的感觉，但仔细反思，所谓“就差一点点”，可能或许“失之毫厘，差之千里”。

但无论如何，这一学期的实验是收获很多的，无论是过了的喜悦还是挂了的懊恼，都是宝贵的回忆，都是获得知识的过程中不可或缺的一部分