

操作系统lab6实验报告

姓名：高远 学号：19374242 班级：202113

实验思考题

Thinking 6.1

代码修改方式为：将 case 0:和 default 部分的语句块互换

```
#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:
        close(fildes[0]);
        write(fildes[1], "Hello world\n", 12);
        close(fildes[1]);
        exit(EXIT_SUCCESS);

    default:
        close(fildes[1]);
        read(fildes[0], buf, 100);
        printf("child-process read:%s", buf);
        close(fildes[0]);
        exit(EXIT_SUCCESS);
    }
}
```

Thinking 6.2

dup函数不是原子性的，该函数内部操作是先共享fd，后共享了data，如果在两个共享之间发生中断，执行其他进程，就会发生预料之外的情况。

Thinking 6.3

所有的系统调用都是原子操作，用户进程进行系统调用，即执行syscall后到操作系统完成操作返回的过程中，不会有其他程序执行。这是因为在系统调用开始时，操作系统就会关闭中断，使系统调用不会被打断。

Thinking 6.4

问题1：可以，因为原情况出现的原因是：对于a和b，在 $a > b$ 且先减少a再减少b时，就可能会出现 $a == b$ 的中间态。改变顺序后b先减少 就不会出现这种状态。

问题2：dup的情况与控制 pipeclose 中 fd 和 pipe unmap 的顺序类似，只不过是先增加b再增加a，改变顺序之后先增加a再增加b，也就出现这种状态。

Thinking 6.5

做法：当操作系统load二进制文件时，根据bss段数据的memsz属性分配对应的内存空间并清零，从而做到bss 段的数据占据了空间，并且初始值都是 0。

Thinking 6.6

```
6  SECTIONS
7  {
8      . = 0x00400000;
9
10     _text = .;          /* Text and read-only data */
11     .text : {
12         *(.text)
13         *(.fixup)
14         *(.gnu.warning)
15     }
16 }
```

如图，可以看到了.text在链接的开始位置是相同的，均为0x00400000，因此我们的 .b 的 text 段偏移值都是一样的。

Thinking 6.7

在 MOS 中我们用到的 shell 命令是外部命令。

因为cd命令是用来切换工作目录的命令，shell 不需要 fork 一个子 shell，因此cd命令是内部指令，也可使用type命令查看该命令类型：

```
[root@centos7 ~]# type cd
```

cd is a shell builtin

证明cd为内部指令。

Thinking 6.8

在user/init.c中，可以看到0 和 1 被 安排”为标准输入和标准输出。

```
58     close(0);
59     if ((r = opencons()) < 0)
60         user_panic("opencons: %e", r);
61     if (r != 0)
62         user_panic("first opencons used fd %d", r);
63     if ((r = dup(0, 1)) < 0)
64         user_panic("dup: %d", r);
65 }
```

Thinking 6.9

如图，在shell 中输入指令 `ls.b | cat.b > motd`

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

[00001c03] pipecreate
[00001c03] SPAWN: ls.b
serve_open 00001c03 ffff000 0x0
serve_open 00002404 ffff000 0x1
pageout: 00000000_0x7f3fcc7c_0000 ins a page
[00002404] SPAWN: cat.b
serve_open 00002404 ffff000 0x0
pageout: 00000000_0x7f3fcc7c_0000 ins a page

.....:spawn size : 2 sp : 7f3fde8:.....
.....:spawn size : 2 sp : 7f3fde8:.....
pageout: 00000000_0x40a400_0000 ins a page
pageout: 00000000_0x40c000_0000 ins a page
serve_open 00002c05 ffff000 0x0
serve_open 00002c05 ffff000 0x0
[00002c05] destroying 00002c05
[00002c05] free env 00002c05
```

```
[00002c05] free env 00002c05
i am killed ...

[00003406] destroying 00003406
[00003406] free env 00003406
i am killed ...

[00002404] destroying 00002404
[00002404] free env 00002404
i am killed ...

[00001c03] destroying 00001c03
[00001c03] free env 00001c03
i am killed ...
```

可以观察到两次spawn和四次进程销毁。

实验难点

难点1：管道的同步

`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。

`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。

如图：

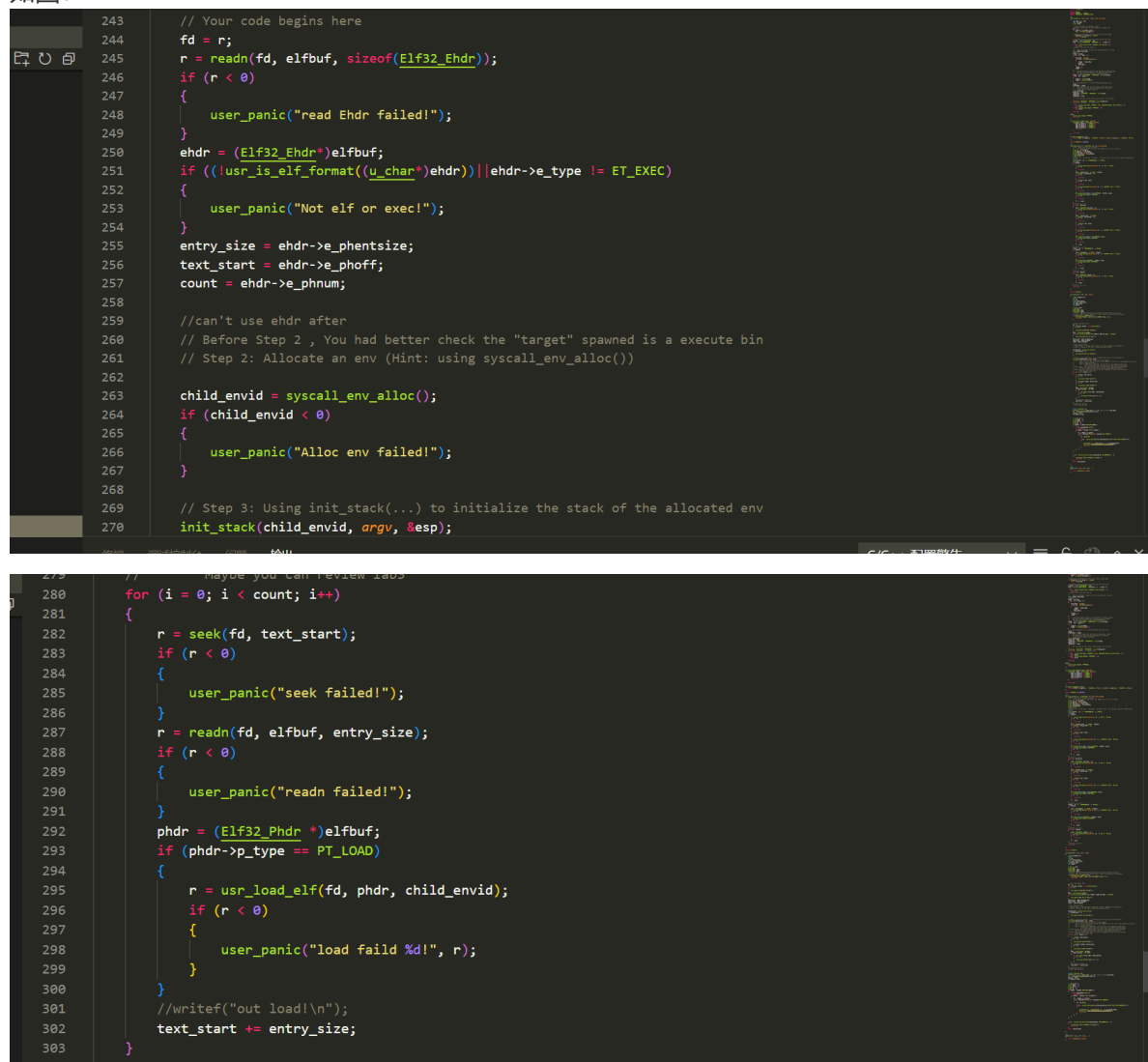
```
8
9 do {
10     runs = env->env_runs;
11     pfd = pageref(fd);
12     pfp = pageref(p);
13 } while (runs != env->env_runs);
14
```

难点2: spawn

spawn过程如下:

- 1.在文件系统中打开对应的文件。
- 2.用syscall_env_allow创建子进程
- 3.在新建系统中用syscall_load_icode将目标程序加载到子进程的地址空间中, 并为他们分配物理页面。
- 4.用init_stack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。
- 5.设置子进程可执行。

如图:



```
243 // Your code begins here
244 fd = r;
245 r = readn(fd, elfbuf, sizeof(Elf32_Ehdr));
246 if (r < 0)
247 {
248     user_panic("read Ehdr failed!");
249 }
250 ehdr = (Elf32_Ehdr*)elfbuf;
251 if (!usr_is_elf_format((u_char*)ehdr) || ehdr->e_type != ET_EXEC)
252 {
253     user_panic("Not elf or exec!");
254 }
255 entry_size = ehdr->e_phentsize;
256 text_start = ehdr->e_phoff;
257 count = ehdr->e_phnum;
258
259 //can't use ehdr after
260 // Before Step 2, You had better check the "target" spawned is a execute bin
261 // Step 2: Allocate an env (Hint: using syscall_env_alloc())
262
263 child_envid = syscall_env_alloc();
264 if (child_envid < 0)
265 {
266     user_panic("Alloc env failed!");
267 }
268
269 // Step 3: Using init_stack(...) to initialize the stack of the allocated env
270 init_stack(child_envid, argv, &esp);
271
272 // maybe you can review lab5
273 for (i = 0; i < count; i++)
274 {
275     r = seek(fd, text_start);
276     if (r < 0)
277     {
278         user_panic("seek failed!");
279     }
280     r = readn(fd, elfbuf, entry_size);
281     if (r < 0)
282     {
283         user_panic("readn failed!");
284     }
285     phdr = (Elf32_Phdr*)elfbuf;
286     if (phdr->p_type == PT_LOAD)
287     {
288         r = usr_load_elf(fd, phdr, child_envid);
289         if (r < 0)
290         {
291             user_panic("load failed %d!", r);
292         }
293     }
294     //writef("out load!\n");
295     text_start += entry_size;
296 }
```

体会与感想

本次实验难度一般, 与lab5相仿, 而且代码量也比lab5小很多, 我做的时间并不算太长, 其难点在于pipe在多进程并发环境下可能出现的一些问题和spawn函数的设计, 也花费了我一定的时间去理解。但事实上, 只有在真正测试shell才发现我们的小系统还是有太多这样那样的不完善之处, 我也会花费更多的时间去完善。

无论怎么说, Lab6是最后一次实验了, 操作系统这门课使我痛并快乐着, 现在回首望去, 也真正感受到这门课使我受益匪浅。

指导书反馈

本次指导书写的较为完备，但也可以在测试方面描述的更加详细，因为我花了较多的时间才弄明白怎么测试shell。