操作系统 Lab6实验报告

实验思考题

Thinking 6.1

代码修改如下，使父进程操作管道的读端，子进程操作管道的写端。

```
#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: /* Handle error */
            break;
        case 0: /* Child - writes to pipe */
            close(fildes[0]);                       /* Read end is unused */
            write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
            close(fildes[1]);                       /* Parent will see EOF
*/
            exit(EXIT_SUCCESS);
        default: /* Parent - reads from pipe */
            close(fildes[1]);                       /* Write end is unused
*/
            read(fildes[0], buf, 100);              /* Get data from pipe */
            printf("parent-process read:%s", buf);  /* Print the data */
            close(fildes[0]);                       /* Finished with pipe */
            exit(EXIT_SUCCESS);
    }
}
```

Thinking 6.2

`dup` 函数先复制了旧文件描述符 `oldfd` 所在的页：

```
if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
                         ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R |
PTE_LIBRARY))) < 0) {
    goto err;
}
```

然后才遍历复制了 `oldfd` 所对应的数据区：

```
if ((* vpd)[PDX(ova)]) {
    for (i = 0; i < PDMAP; i += BY2PG) {
        pte = (* vpt)[VPN(ova + i)];

        if (pte & PTE_V) {
            // should be no error here -- pd is already allocated
            if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
                                     pte & (PTE_V | PTE_R | PTE_LIBRARY))) <
0) {
                goto err;
            }
        }
    }
}
```

这两步操作中间可能产生时钟中断，若在文件描述符 `oldfd` 所在的页完成复制， `oldfd` 所对应的数据区复制之前发生了时钟中断，此时新文件描述符 `newfd` 所对应的数据区仍未空，对其进行文件操作时就会出错。

Thinking 6.3

系统调用一定是原子操作，因为系统调用是操作系统内核为用户进程提供服务的接口，不应被来自用户进程的时钟中断打断。

在实验代码中，处理系统调用的 `handle_sys` 函数通过定义的宏 `CLI` 实现了关中断，使得在执行系统调用的过程中不会产生时钟中断：

```
/* lib/syscall.S */
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                                // Macro used to save trapframe
    CLI                                     // Clean Interrupt Mask
    nop
    .set at                                 // Resume use of $at
......
```

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

Thinking 6.4

- 可以解决场景中的进程竞争问题。原情况出现的原因是在 `pageref(pipe) > pageref(fd)` 的情况下，先解除 `pipe` 的映射，再解除 `fd` 的映射，就可能出现 `pageref(pipe) == pageref(fd)` 的中间情况，导致判断出错。而先解除 `fd` 的映射，再解除 `pipe` 的映射，就能够保持 `pageref(pipe) > pageref(fd)` 的大小关系，不会出现错误判断的情况。
- 在 `dup` 函数出现类似情况的原因是在文件描述符所在的页完成复制后，其对应的数据区可能还没完成复制，此时通过文件描述符访问文件数据时会产生错误。解决方法为先复制文件对应的数据区，再复制文件描述符，即可在进程访问新文件时保证其数据已经准备好。

Thinking 6.5

`usr_load_elf` 函数实现如下：

```
int usr_load_elf(int fd , Elf32_Phdr *ph, int child_envid){
    //Hint: maybe this function is useful
    //      If you want to use this func, you should fill it ,it's not hard
    u_long va = ph->p_vaddr;
    u_int32_t sgsize = ph->p_memsz;
    u_int32_t bin_size = ph->p_filesz;
    u_long offset = va - ROUNDDOWN(va, BY2PG);
    u_char *bin;
    u_long i;
    int r;

    r = read_map(fd, ph->p_offset, &bin);
    if (r < 0)
        return r;
    if (offset != 0) {
        if ((r = syscall_mem_alloc(child_envid, va, PTE_V | PTE_R)) < 0)
            return r;
        if ((r = syscall_mem_map(child_envid, va, 0, USTACKTOP, PTE_V | PTE_R)) < 0)
            return r;
        user_bcopy(bin, USTACKTOP + offset, MIN(BY2PG - offset, bin_size));
        if ((r = syscall_mem_unmap(0, USTACKTOP)) < 0)
            return r;
    }
    for (i = offset ? MIN(BY2PG - offset, bin_size) : 0; i < bin_size; i += BY2PG) {
        if ((r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R)) < 0)
            return r;
        if ((r = syscall_mem_map(child_envid, va + i, 0, USTACKTOP, PTE_V | PTE_R)) < 0)
            return r;
        user_bcopy(bin + i, USTACKTOP, MIN(BY2PG, bin_size - i));
        if ((r = syscall_mem_unmap(0, USTACKTOP)) < 0)
            return r;
    }
    while (i < sgsize) {
        if ((r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R)) < 0)
            return r;
        i += BY2PG;
    }

    return 0;
}
```

仿照lab3中 `load_icode_mapper` 的实现，先将 `.data` 和 `.text` 加载进内存后根据 `ph->p_memsz` 的值分配 `.bss` 段的空间，再分配空间的同时就将对应空间进行了置零操作。

Thinking 6.6

因为在 `user.lds` 中，代码段 `.text` 最先被加载，因此开始位置相同，偏移值相同。

```
    . = 0x00400000;

    _text = .;          /* Text and read-only data */
    .text : {
    *(.text)
    *(.fixup)
    *(.gnu.warning)
    }
```

Thinking 6.7

- MOS中用到的shell指令需要通过 `spawn` 函数打开对应的文件来执行对应命令，因此属于外部命令。
- 在Linux中，bash创建的进程实质上都是在bash的一个子shell中运行，在子shell中执行的操作在进程死亡时并不会将对应信息返回给当前bash。而 `cd` 命令所做的是改变shell的当前文件目录，因此如果 `cd` 是一个外部命令，则它只会改变子shell中的当前文件目录，而不会对父shell产生影响。因此所有能对当前shell的环境作出改变的命令都必须是内部命令，其中自然也包括 `cd` 命令。

Thinking 6.8

在 `user/init.c` 中的 `void umain(int argc, char **argv)` 中调用了 `opencons` ：

```
close(0);
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);
```

由于此时还没有完成对文件系统的设置，因此在 `user/console.c` 中的 `int opencons(void)` 实现了对0号和1号文件描述符的设置，并约定将两者作为标准输入输出：

```
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);
}
```

Thinking 6.9

- 可以观察到2次 `spawn` ，第一次 `spawn` 对应 `ls.b` 进程，第二次 `spawn` 对应 `cat.b` 进程。
- 可以观察到4次进程销毁，第一次销毁对应文件打开进程，第二次销毁对应文件关闭进程，第三次销毁对应 `cat.b` 进程，第四次销毁对应 `ls.b` 进程。

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

[00001c03] pipecreate

[00001c03] SPAWN: ls.b

serve_open 00001c03 ffff000 0x0

serve_open 00002404 ffff000 0x1

[00002404] SPAWN: cat.b

serve_open 00002404 ffff000 0x0



::::::::::spawn size : 20  sp : 7f3fdfe8::::::::

pageout:        @@@___0x40a400___@@@  ins a page

serve_open 00002c05 ffff000 0x0

serve_open 00002c05 ffff000 0x0



::::::::::spawn size : 20  sp : 7f3fdfe8::::::::

pageout:        @@@___0x40c000___@@@  ins a page

[00002c05] destroying 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 ...
```

实验难点

管道的竞争与同步

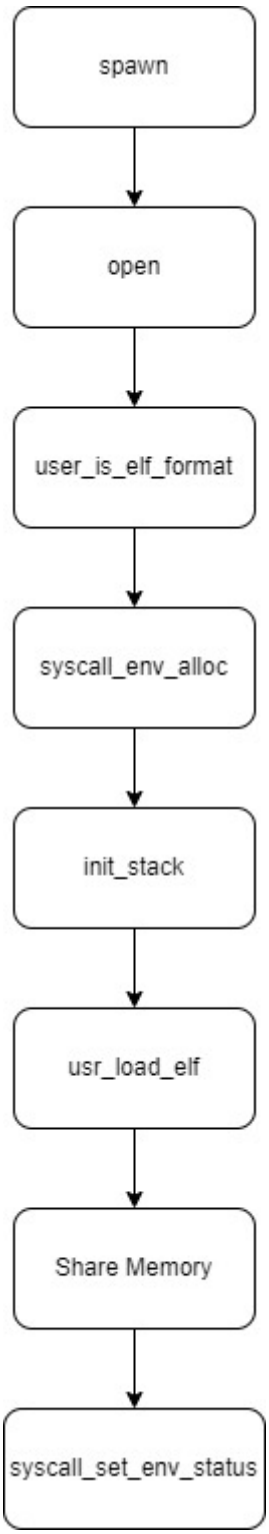
在实现管道的过程中需要理解对于 `pageref(fd)` 和 `pageref(pipe)` 的等价关系判断：

- 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)`

而在一般情况下，有 `pageref(p[0]) < pageref(pipe)` 和 `pageref(p[1]) < pageref(pipe)`，因此在解除映射时应先解除文件描述符 `fd` 的映射，再解除管道 `pipe` 的映射，以保持上述的大小关系。

`spawn`

`spawn` 函数的实现流程如下：

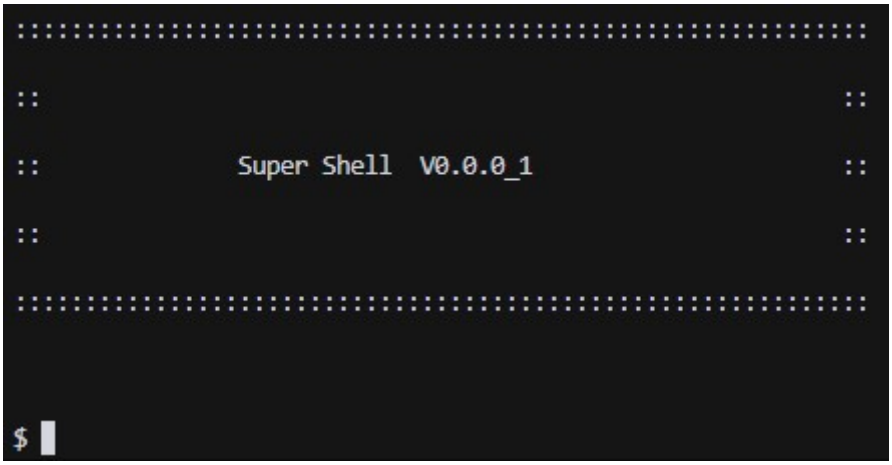


- 从文件系统打开对应的文件，并申请新的进程描述符；
- 检查文件是否为ELF格式；
- 将目标程序加载到子进程的地址空间中，并为它们分配物理页面；
- 为子进程初始化堆、栈空间，并设置栈顶指针，以及重定向、管道的文件描述符；
- 将父进程的共享页面映射到子进程的地址空间中；
- 将子进程设置为就绪状态。

体会感想

lab6的整体实现难度并不算高，但其中一些实现逻辑，如管道的竞争与同步，还是有一定理解难度，不过其实现方式的设计思路非常巧妙，值得在其他软件系统的设计层面学习借鉴。

在完成所有实验后，看到终端呈现出的shell界面还是非常有成就感的。



从内核启动，到内存管理，到进程管理，到系统调用，到文件系统，最后到管道机制和shell的搭建，虽然在逐步搭建MOS大厦的过程中遇到了很多问题，踩了很多坑，但万幸在自己的努力和老师同学的帮助下都顺利解决了。但回过头来看，我对其中的部分内容仍然理解的还不够透彻，有一些已知的祖传问题和优化方向自己也无法提出行之有效的优化解决方案。

经过一整个学期的洗礼，我不单通过MOS实验对操作系统的结构逻辑和实现机制有了更深刻的了解，同时也提高了自己对较大规模软件的系统把握能力，学会了更多测试方法，通过上机测试也锻炼了自己的思维速度和抗压能力。