OS_lab6实验报告

思考题

Thinking 6.1 父进程为读者

父进程关掉写端1，子进程关掉读端0。

Thinking 6.2 dup 中的进程竞争

dup函数的作用是将oldfdnum所代表的文件描述符指向的数据完全复制给newfdnum文件描述符。在这个过程中：首先将newfd所在的虚拟页映射到oidfd所在的物理页，然后将newfd的数据所在的虚拟页映射到oldfd的数据所在的物理页。

```
// 子进程
    read(p[0], buf, sizeof(buf));
// 父进程
    dup(p[0], newfd);
    write(p[1], "Hello", 5);
```

在上述这段代码中，如果子进程先执行，然后在read之前发生时钟中断，然后父进程开始执行。父进程在dup中，先完成了对p[0]的映射，然后发生中断，还没有映射pipe。这时回到子进程，就会出现 ref(p[0]) == ref(pipe) 的情况，认为此时写进程关闭。

Thinking 6.3 原子操作

在进行系统调用时，handle_sys程序中，汇编宏CLI将全局中断禁用了，此时时钟中断被关闭，因此进程不会被切换。

Thinking 6.4 解决进程竞争

- 可以解决。因为pageref(pipe)总是大于等于pageref(fd)的，所以为了避免解除两个映射时，短暂出现二者相等的情况，可以先解决fd的映射，这时就能保证不会出现二者相等的情况。
- 会出现上述这种问题，也可以通过调整映射的顺序来解决。

Thinking 6.5 如何实现加载 bss 段

bss段应该与text段data段连续的放在一起，但是ELF中没有空间，在分配映射页面时，text段与data段没有占满的空间置为0给了bss段，然后再给他另外分配的时候，只使用syscall_mem_alloc而不映射。

Thinking 6.6 内置命令与外部命令

在init.c的umain函数中将0和1分别被设置为了标准输入和标准输出。

```
//将0关闭，随后使用opencons函数打开的文件描述符编号就被设置为零为0
close(0);
if ((r = opencons()) < 0)
    user_panic("opencons: %e", r);
if (r != 0)
    user_panic("first opencons used fd %d", r);
// 然后通过dup函数把1设置为标准输出
if ((r = dup(0, 1)) < 0)
    user_panic("dup: %d", r);
```

Thinking 6.7 标准输入和标准输出

- MOS中，我们所有的命令执行之前都需要fork一个子进程来进行处理，所以都是外部命令。
- cd命令需要改变父进程的所在路径，不需要fork，所以是内部命令。

Thinking 6.8 解释命令执行的现象

- 2次spawn，这两次生成的子进程分别用于执行ls.b和cat.b。
- 2次销毁，分别对应spawn生成的用于执行ls.b和cat.b的子进程。

难点分析

spawn 函数的实现

spawn 函数与 fork 函数类似，其最终效果都是产生一个子进程，不过与 fork 函数不同的是，spawn 函数产生的子进程不再执行与父进程相同的程序，而是装载新的 ELF 文件，执行新的程序。

首先要做的是将文件内容加载到内存中，根据传入的文件路径参数 `prog` 打开文件，并首先通过 `readn` 函数读取其文件头的信息。

```
    int fd;
    if ((fd = open(prog, O_RDONLY)) < 0) {
        return fd;
    }

    int r;
    u_char elfbuf[512];

    if ((r = readn(fd, elfbuf, sizeof(Elf32_Ehdr))) < 0 || r !=
sizeof(Elf32_Ehdr)) {
        goto err;
    }
```

接着使用 `elf_from` 将文件头转换为 `Elf32_Ehdr` 结构体的格式。

```
    const Elf32_Ehdr *ehdr = elf_from(elfbuf, sizeof(Elf32_Ehdr));
    if (!ehdr) {
        r = -E_NOT_EXEC;
        goto err;
    }
```

接着使用系统调用 `syscall_exofork` 创建一个新的进程。并调用 `init_stack` 完成子进程栈的初始化。

```
    u_int child;
    child = syscall_exofork();
    if (child < 0) {
        r = child;
        goto err;
    }

    u_int sp;
    if ((r = init_stack(child, argv, &sp)) < 0) {
        goto err1;
    }
```

接下来遍历整个 ELF 头的程序段，将程序段的内容读到内存中。

```
    size_t ph_off;
    ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
        if ((r = seek(fd, ph_off)) < 0 || (r = readn(fd, elfbuf, ehdr-
>e_phentsize)) < 0) {
            goto err1;
        }
```

最后设定子进程为运行状态以将其加入进程调度队列，实现子进程的创建。

```
    if ((r = syscall_set_env_status(child, ENV_RUNNABLE)) < 0) {
        debugf("spawn: syscall_set_env_status %x: %d\n", child, r);
        goto err2;
    }
    return child;
```

心得体会

Lab6要求我们完成pipe机制和实现一个简单的shell，这次的lab结束之后，我们的MOS操作系统算是基本完成了。写lab6的时候，os的理论考试已经结束了，我发现理论部分和实验还是有很多相关联的地方的，可惜之前理论部分没有认真学习。经过了os的理论复习，我对os实验的部分内容也有了更加深刻的理解。