OS Lab6 - ExprimentReport

1. 思考题

> Thinking 7.1 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?

```
switch (fork()) {
case -1: /* Handle error */
    break;
case 0: /* Child - reads from pipe */
    close(fildes[0]); /* Write end is unused */
    write(fildes[1], "xxx", n); /* Get data from pipe */
    printf("child-process write:%s",buf); /* Print the data */
    close(fildes[1]); /* Finished with pipe */
    exit(EXIT_SUCCESS);

default: /* Parent - writes to pipe */
    close(fildes[1]); /* Read end is unused */
    read(fildes[0], buf, 100); /* Write data on pipe */
    close(fildes[0]); /* Child will see EOF */
    exit(EXIT_SUCCESS);
}
```

即更换父子进程关闭的管道读写端以及read、write系统调用。

> Thinking 7.2 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?

关系式：`pageref(rfd) + pageref(wfd) = pageref(pipe)` 在非原子的close函数调用时不能保证成立。而为了使得另一个进程在判断管道某端关闭时的条件 `pageref(itself-fd)=pageref(pipe)` 不会出错，在close函数中需要先将 `pagefref(itself-fd)` 减小1，再将 `pageref(pipe)` 减小1，从而保证在该函数不会导致子进程发生错误判断。

close(fd)函数会将 `fd` 和 `pipe` 所在页面的ref均减小1，为保证管道另一端在真的关闭之前，当前端的读写不会发生误判，需要先unmap `fd`，再unmap `pipe`，而dup函数则与之相反：将 `fd` 复制给另一个文件描述符，即 `fd` 所在的页面和 `pipe` 所在的页面的pp_ref数都要增加1，因此其unmap的次序也应当与close函数恰好相反，若不是这样，则以下实例可能发生同步的错误：

```
int pip[2] = {0};
pipe(pip);
p_id r = 0;
if((r=fork)==0) {
    close(pip[0]);
    write(pip[1], "a", 1);
}
if (r>0) {
    close(pip[0]);
    dup(STDOUT, pip[1]);
```

```
    close(pip[1]);
    execvp("ls", "-a", NULL);
    ...
}
```

此处本来应当由父进程调用`execvp`函数去执行`ls -a`的命令，将结果送入管道写端，但是，dup函数若执行到恰好将stdout的文件描述符被替换(map)为pip[1]的文件描述符后就被切换为子进程`此时 wfd=2, rfd=1, pipe=2`,此时子进程将判断写端已经关闭，从而write函数执行出错。

> Thinking 7.3 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

系统调用d都是原子操作，具体代码是在include/stackframe.h中定义的CLI宏，如下所示:

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

CLI宏在handle_sys函数中出现，作用是设置CP0_STATUS寄存器，因此后面的中断无法发生，因此就无法发生嵌套中断，导致系统调用也不能被打断，因此系统调用是原子操作。

对于比较特殊的`sys_ipc_receive`，也是在等待时设置为`ENV_NOT_RUNNABLE`后让步，待发送消息的进程将其接收消息的进程重新设置为env_runnable，由于`sys_ipc_can_send`是原子性的，因而其全过程也是原子性的。

> Thinking 7.4 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，在fd.c 中有一个dup 函数，用于复制文件内容。试想，如果要复制的是一个管道，那么是否会出现与close 类似的问题？请模仿上述材料写写你的理解。

已在`Thinking 7.2`中做了详细的解释

> Thinking 7.5 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?

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。

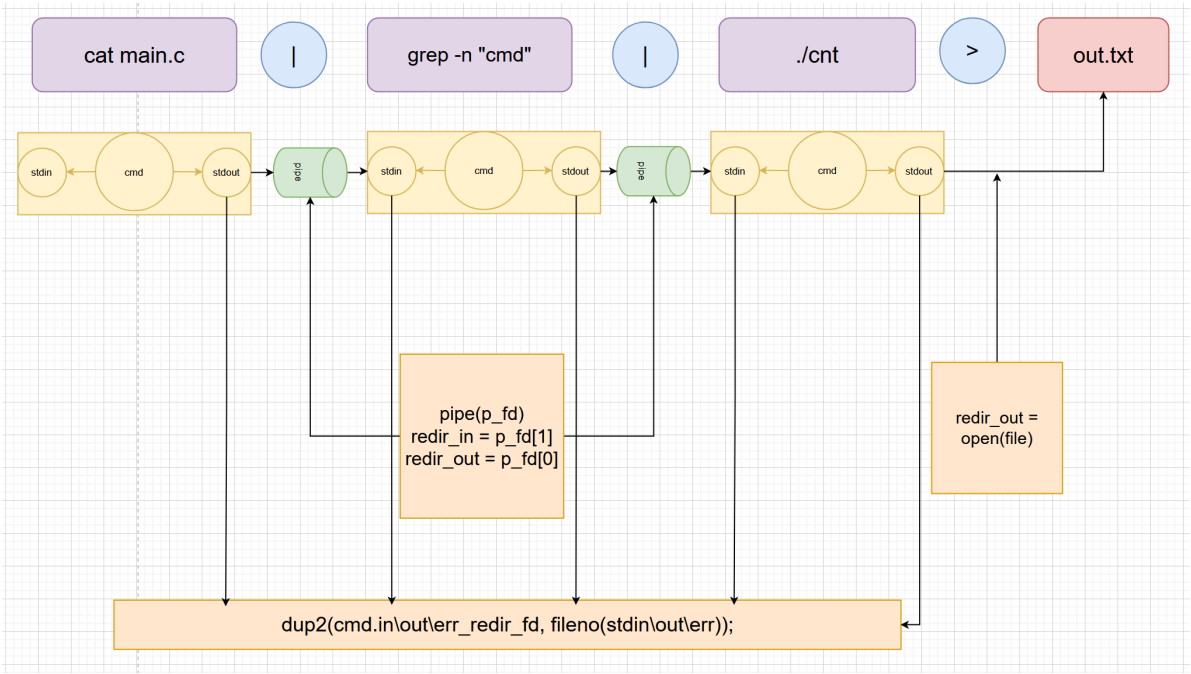
> Thinking 7.6 为什么我们的*.b 的text 段偏移值都是一样的，为固定值?

在user的lds中规定了所有加载的可执行程序的text段的偏移值都是一样的。

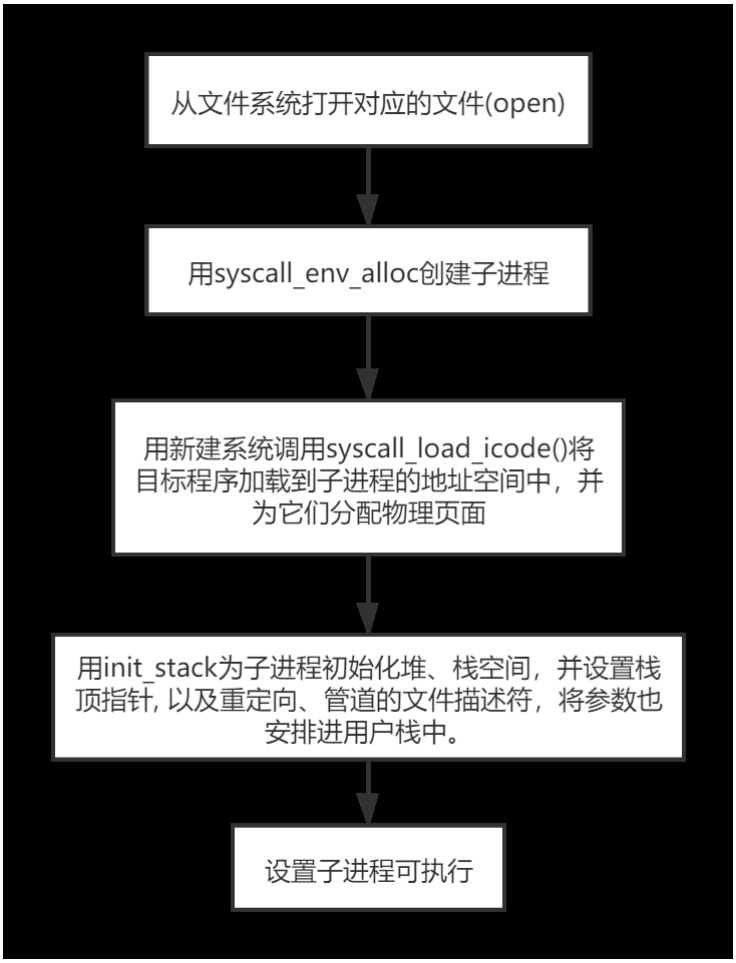
2. 实验难点图示

重定向指令管道的作用机制

当然指令是linux级的...但不影响我们理解重定向的机制：



spawn函数流程



3. 实验总结

Lab6是最后一次实验。总体感觉lab6本身的难度不大，但感觉花费了最多的时间——de之前lab的bug实在是太难了。尤其是自己在shell中出现了问题(非常奇怪：在`ls.b | cat.b`后会出现pageout此后出现一堆乱码，在大概50s的乱码刷屏后被告知`page out到了env的区域`)，总以为是`spawn`函数中加载镜像的锅才导致出现了这样的严重问题。但最后花了好久才发现是pipe的问题——pipe在写的时候无论是否能写都会在写完一个字符后调用`yield()`。虽然修改后能够解决问题，但始终都没找到问题在哪里~这个问题耽误了很长时间，因为无论断点调试还是尝试输出信息，甚至尝试panic都丝毫没有办法定位问题(不知道为什么所有的writef、user_panic都全部失效，而dump也无法找到相应函数的地址)。我也因此在祖安了好几天后认识到这样一个道理：操作系统是最接近系统底层那冰冷无情的机器的一个连贯性的存在，其出现的错误很有可能涉及多个进程、多个部分，我们不应该奢求自己在写操作系统程序的过程中出现了bug能够向OO那样在IDEA中舒舒服服地打上断点调试。同时也很难想象真正运行在真实硬件上的操作系统如果出错又会发生多么严重的问题。这样想来，真的是应该对现代OS诞生的年代中为MULTICS、UINX等操作系统贡献才智的计算机科学家们保持一种敬佩。同时反问在模拟硬件上用着交叉编译器在300页指导书和老师助教的帮助下填写一个简单到不能再简单的操作系统的不到10%的有着超长注释的代码的自己，若没有github中学长的代码，能够顺利走到最后吗?

4. 指导书反馈

感觉思考题7.2和7.3的顺序应当调换下，7.2和7.4这两个问题其实是一个连贯性的问题吧。

从lab5开始就没有在code中的注释布加上对应的exercise号了，在指导书中细致到为思考题的汇总区的每一个题目加上跳转链接的课程组(夸夸~)，应当可以画点时间在code里加上吧!

5.残留难点

就是之前所说的那个bug，真的不知道为什么会出现。这种乱码的情况应该是出现了严重问题才有可能发生吧~

此外，对user中多出来一个`user_lds`不是很理解：虽然明白它是起到用户态下加载镜像的地址偏移的基址寄存器的作用，但是为什么不设置异常入口地址呢？其外其中有些奇怪的段的作用也不是很懂~