Lab6实验报告

一、实验思考题

Thinking 6.1

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?

将case 0和default中的内容交换位置

Thinking 6.2

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?

`dup`函数的功能是将一个文件描述符（例如**fd0**）所对应的内容映射到另一个文件描述符（例如**fd1**）中。这个函数最终会将**fd0**和**pipe**的引用次数都增加**1**，将**fd1**的引用次数变为**fd0**的引用次数。若在复制了文件描述符页面后产生了**时钟中断**，pipe的引用次数没来的及增加，可能会导致另一进程调用`pipeisclosed`，发现`pageref(fd[0]) = pageref(pipe)`，误以为读/写端已经关闭。

Thinking 6.3

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。

当系统陷入内核时，会关中断，此段时间不会出现时钟中断产生的进程切换

Thinking 6.4

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。**

  可以。fd的引用先接触，只会使`pageref(fd)`和`pageref(pipe)`数值间距加大，从而不会出现暂时的相等情况而误判。

- **我们只分析了 close 时的情形，那么对于 dup 中出现的情况又该如何解决？请模仿上述材料写写你的理解。**

  先映射pipe，后映射fd

Thinking 6.5

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?

page_alloc()的时候已经把相应空间清零了，所以只需界定好bin_size和sgsize就好（一开始使用了bzero()反而会因为不支持地址未对齐情况而出错）

Thinking 6.6

为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值?

因为在user/user.lds中规定了text段的位置为`0x00400000`

Thinking 6.7

在 shell 中执行的命令分为内置命令和外部命令。在执行内置命令时 shell 不需要 fork 一个子shell，如 Linux 系统中的 cd 指令。在执行外部命令时 shell 需要 fork 一个子 shell，然后子 shell 去执行这条命令。据此判断，在 MOS 中我们用到的 shell 命令是内置命令还是外部命令？请思考为什么 Linux 的 cd 指令是内部指令而不是外部指令?

是外部命令，spawn函数使用系统调用`syscall_env_alloc()`创建了子进程；

cd这种指令过于常见基本，每次都得fork子进程开销大浪费时间和资源。

Thinking 6.8

在哪步， 0 和 1 被 “安排” 为标准输入和标准输出？请分析代码执行流程，给出答案。

```
if ((r = dup(0, 1)) < 0) user_panic("dup: %d", r);
```

在user/init.c文件中，进程的主函数将文件描述符0映射到1，于是之后我们从0写入，从1读出就相当于在操纵同一个管道文件了。

Thinking 6.9

在你的 shell 中输入指令 ls.b | cat.b > motd。

- **请问你可以在你的 shell 中观察到几次 spawn？分别对应哪个进程?**

  两次，分别是ls.b和cat.b.

  ```
  [00001c03] SPAWN: ls.b
  [00002404] SPAWN: cat.b
  ```

- **请问你可以在你的 shell 中观察到几次进程销毁？分别对应哪个进程?**

  四次，分别是：(前两个编号和被人不一样，后两个一样 -> 那时lab6没过，因为多加了重定向时的O_CREAT和stat判断)

  ```
  [00002c06] destroying 00002c06  -> [00003406] destroying 00003406
  [00003405] destroying 00003405  -> [00002c05] destroying 00002c05
  [00002404] destroying 00002404
  [00001c03] destroying 00001c03
  ```

  所以这几个envid是固定的。。。还没理解

全文：（旧版，新版就是把前两个destroy的内容改成新的）

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

[00001c03] pipecreate
```

```
[00001c03] SPAWN: ls.b

serve_open 00001c03 ffff000 0x0

serve_open 00002404 ffff000 0x0

serve_open 00002404 ffff000 0x101

[00002404] SPAWN: cat.b


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

serve_open 00002404 ffff000 0x0

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


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

serve_open 00002c06 ffff000 0x0

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

serve_open 00002c06 ffff000 0x0

[00002c06] destroying 00002c06

[00002c06] free env 00002c06

i am killed ...

[00003405] destroying 00003405

[00003405] free env 00003405

i am killed ...

[00002404] destroying 00002404

[00002404] free env 00002404

i am killed ...

[00001c03] destroying 00001c03

[00001c03] free env 00001c03

i am killed ...
```

二、实验难点

1.进程调度算法对共享变量的影响分析

- **语句运行空间顺序对结果的影响**

  pp_ref减少的先后和进程切换的原因导致进程未在程序目标空间位置进行比较，导致误判。

- **语句运行时间顺序对结果的影响**

  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”。

2.整个shell的书写

内容本身也很多，指导书和注释也不是很详细，功能上也可以有不同的拓展。写的时候参考了学长的代码，引入了一些超过标程的功能，导致测评失败，找了很久的错误呢。同时真正实现shell的代码也有好几个文件，应好好品读理解。

三、体会与感想

整个OS课设的代码就这样稀里糊涂的填完了。没能摸清MOS的全貌，也就会找猫画虎写点管中窥豹之物，有些许片段的理解。这个课设学得很不到位，希望之后能好好厘清MOS（把MOS图景解读写成一个互动APP或许是个不错的项目），然后去做一下MIT和清华的操作系统实验，也得观摩观摩Linux内核。

1 Elf32_Half e_type的宏定义在哪里呢？找不着，可能没有宏，好怪

2 关于页表项存在与否的判断，写法可以统一一下吧。第一种最清晰

- `(*vpd)[PDX(va)] & PTE_V` 和 `(*vpt)[VPN(va)] & PTE_V`
- `(*vpd)[va/PDMAP] & PTE_V` 和 `(*vpt)[va/BY2PG] & PTE_V`