BUAA-OS-lab6

一、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？

让父进程关掉管道的写端，子进程关掉管道的读端。

代码修改后如下：

```
switch(fork()) {
case-1:
   break;


case 0:
   close(fildes[0]);
   write(fildes[1],"Helloworld\n",12);
   close(fildes[1]);
   exit(EXIT_SUCCESS);


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

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/lib/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？

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

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

- 假设fork 结束后，子进程先执行。时钟中断产生在上述两条指令之间。

- 子进程 dup(p[1], newfd) 后，newfd增加了对 p[1] 的映射，但还没有来得及增加对 pipe 的映射，此时时钟中断产生，父进程接着执行。
- 父进程close(p[0])后，此时各个页的引用情况：pageref(p[0]) = 1，pageref(p[1]) = 3，此时 pipe 的pageref 是 3。
- 父进程执行 write，write 中首先判断读者是否关闭。比较 pageref(pipe)与 pageref(p[1])之后发现它们
  都是 3，说明写端已经关闭，于是父进程退出。

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说明。

系统调用一定是原子操作，因为系统调用时，系统陷入内核，关闭时钟中断以保证系统调用不会被打断。

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 pipe_close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，用于复制文件描述符。 试想，如果要复制的文件描述符指向一个管道，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。

- 可以解决，分析如下：
  - 若调换 fd 和 pipe 在 close 中的 unmap 顺序，使得 fd引用次数的-1 先于 pipe。
  - 最初pageref(p[0]) = 2，pageref(p[1]) = 2，pageref(pipe) = 4；仍考虑子进程先运行，对p[1]执行close函数时先解除了对写端fd的映射，之后发生时钟中断，此时pageref(p[0]) = 2，pageref(p[1]) = 1，pageref(pipe) = 4；父进程执行完close(p[0])后，pageref(p[0]) = 1，pageref(p[1]) = 1，pageref(pipe) = 3，父进程开始运行时仍不满足写端关闭的条件，因此竞争问题没有出现。
- 会出现，在程序执行时，总满足pageref(p[0]) ≤ pageref(pipe)，如果在dup中先对fd映射，此时fd的ref先增加，在这间隙中会出现读端已满的现象，因此也会出现与close相似的问题，需要调整映射的顺序。

Thinking 6.5

> 思考以下三个问题。
>
> - 认真回看Lab5文件系统相关代码，弄清打开文件的过程。
> - 回顾Lab1与Lab3，思考如何读取并加载ELF文件。
> - 在Lab1 中我们介绍了 data text bss 段及它们的含义，data 段存放初始化过的全 局变量，bss段存放未初始化的全局变量。关于memsize和filesize，我们在Note 1.3.4中也解释了它们的含义与特点。关于Note1.3.4，注意其中关于“bss段并不在文 件中占数据”表述的含义。回顾Lab3并思考：elf_load_seg()和load_icode_mapper() 函数是如何确保加载ELF文件时，bss段数据被正确加载进虚拟内存空间。bss段 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。请回顾elf_load_seg() 和 load_icode_mapper() 的实现，思考这一点是如何实现的?

在load_icode_mapper()函数中我们调用page_alloc函数，当src不为空指针时，对`bss`段进行内存分配，当offset + len <= PAGE_SIZE时，会将空位填0，在这段过程中为`bss`段的数据全部赋上了默认值0。

Thinking 6.6

> 通过阅读代码空白段的注释我们知道，将标准输入或输出定向到文件，需要 我们将其dup到0或1号文件描述符（fd）。那么问题来了：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。

在user/init.c中，如下代码将0和1安排为了标准输入输出：

```
if ((r = opencons()) != 0) {
        user_panic("opencons: %d", r);
    }
    // stdout
    if ((r = dup(0, 1)) < 0) {
        user_panic("dup: %d", r);
    }
```

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 命令。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 命令是内部命令而不是外部命令？

是内部命令，Linux的`cd`命令是内部命令的原因是cd改变工作目录直接影响的是当前shell会话的环境，这一操作需要直接影响到shell自身的状态。外部命令作为独立进程执行，无法直接改变父shell的环境。

Thinking 6.8

> 在你的 shell 中输入命令 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？
> - 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？

- 观察到3次spawn，分别对应进程00003805、00004006、 00002803
- 观察到了3次进程销毁，分别对应0进程0003805、00004006、00002803。

二、实验难点

进程竞争

管道的读写为了避免造成死循环，需要对 pages 数组成员维护一个页引用变量 pp_ref 来记录指向该物理页 的虚页数量。pageref 实际上就是查询虚拟地址对应的实际物理页，然后返回其pp_ref变 量的值。我们可以借助pageref(rfd) + pageref(wfd) = pageref(pipe)这个恒等式来判断管道另一端是否已经关闭。

管道关闭的正确判断

进程通过 pipe_close 函数来关闭管道的端口，该函数的实质是通过两次系统调用 unmap 解除文件描述符 fd和数据缓存区 pipe的映射。但是由于进程切换的存在，并不能保证两次系 统调用可以在同一进程时间片内被执行，两次系统调用之间可能因为进程切换而被打断。所以， fd 和对 pipe 的 pp_ref 也不能保证同步被写入，这将影响我们判断管道是否关闭的正确性。

为了避免在间隙中出现“假关闭”的现象，我们需要调整解除页面映射的顺序。

三、实验体会

整体而言，lab6根据指导完成实验的难度不大，但要理解整个实验是比较有难度的。实现完文件系统和管道后，我们可以实现自己创建文件，可以实现控制台中断，与控制台交互等，这让整个操作系统活起来了。我觉得要理解整个实验还得课下再花点时间。