Ausar的补充指导书----lab6

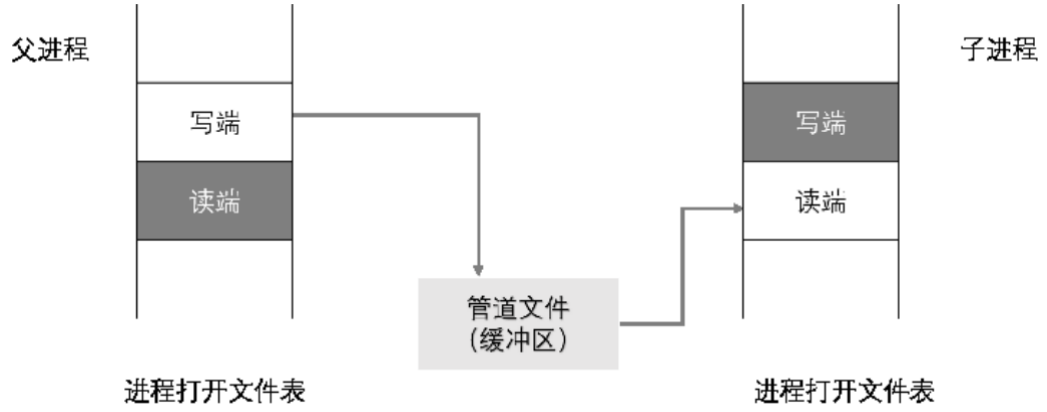
> 如果发现有错漏，请及时与本人或课程组联系，谢谢

Exercise 6.1

实验背景

咱们要开始实现管道啦。管道是操作系统中，进程传输文件的一种方式。如同其名字一样，一个管道可以一边“灌水”，另一端取水。

其实，管道的核心，是父子进程中**共享**的一块内存区域。



不过，等等，我们之前在写fork的时候，把父子进程的内存都是置为COW了，也就是写时复制。

但是这样就没法实现文件共享了啊。

但是，我们的内存权限位中，存在着一位叫做“PTE_LIBRARY”，意思是共享内存。

咱们只需要把缓冲区的权限位修改一下就好咯。

实验内容

> 仔细观察 pipe 中新出现的权限位PTE_LIBRARY，根据上述提示修改 fork 系统调用，使得管道缓冲区是父子进程共享的，不设置为写时复制的模式。Hint: 修 改 fork.c 中的 duppage 函数

实验提示

这一步的工作不是添加权限位，而是**跳过**带有共享位的内存区域。不要把这内存区域给设置为COW就行了。

Exercise 6.2

实验背景

光有了缓冲区，我们还得进行读写，关闭操作对不。

因为读写之前，首先要检查一下管道是否还开着，所以我们需要完成检查相关的函数。

怎么检查管道是否还开着呢？

我们的操作系统中，视管道为一个特殊的文件，因此读端和写端分别保留有一个rfd，wfd，即读写文件描述符。

当然，除了文件描述符，双方都需要缓冲区。因此，对于内存的引用次数，有如下关系：

- 读方文件描述符引用次数：记为A
- 写方文件描述符引用次数：记为B
- 缓冲区引用次数：A+B

很显然，如果双方都在持有这个管道，那么A和B不为0，就永远有`A<A+B`以及，`B<A+B`

那么，**什么时候会出现等于呢**？说明另一方的引用次数为0了，也就是对面关掉管道了。

因此，我们只需要判断文件描述符和管道的引用次数是否相等，就能知道管道是否被关闭。

判断完管道是否打开，那么读写操作就简单多了。

下面是Pipe结构体的定义

```
struct Pipe {
    u_int p_rpos;       // read position
    u_int p_wpos;       // write position
    u_char p_buf[BY2PIPE];  // data buffer
};
```

其中的p_buf是一个环形缓冲区，也就是说，两个读写头的位置需要对`BY2PIPE`取模之后才能使用。

当然，一定要注意一下缓冲区溢出的情况。即缓冲区已满，不应该写入。缓冲区为空，不应该读取。

实验内容

> 根据上述提示与代码中的注释，填写 user/pipe.c 中的 piperead、pipewrite、 _pipeisclosed 函数并通过 testpipe 的测试。

实验提示

- 在判断管道是否关闭时，只需要用pageref这个函数就可以获取对应的引用次数。两个引用次数相等，说明管道已关闭。
- 但是，有可能存在读数据读到一半的时候，发生了进程切换，导致读到的数据过时的情况。
    - 一个可行的做法是： 在读取两个引用数字之前，记录一次envrun的计数值。两个引用数字都读取完之后，检验一遍之前读的envrun是否和现在的envrun一致。如果一致，说明没发生进程切换。否则需要重新读取。
    - envrun变量，是在env.c/envrun函数中维护的，记得确认一下
- 在读写的时候，采用`fd2data`函数，可以从fd获取到pipe结构体。
- 开始读写之前，首先要判断一下缓冲区是否满/空。
    - 如果满/空，且管道已经关闭，那么直接退出
    - 否则需要挂起，等待别的进程

Exercise 6.3

实验背景

之前的判断管道是否关闭，我们采用了判断引用次数是否相等的方法。这个方法看起来很妙，但是并不是坚不可摧。

引用次数相等==管道关闭，这个是基于一个假设，那就是管道关闭和引用次数的修改是同步的。

但是，管道关闭的过程，实际上需要做两件事：

- 需要关闭文件描述符
- 需要关闭缓冲区

管道开启的过程与这个正好相反。

这是一个分步骤的操作，但是尴尬的事情就是，万一中途发生了进程切换，就有可能让对面进程错误的认为管道未关闭。或者是在建立管道的时候，让他认为管道未开启。

实际上，我们只需要在创建管道，和关闭管道的时候。采用特定的顺序执行上面两个命令，使得可以保证，管道开启时有`A<A+B(描述符引用数<缓冲区引用数)`，而管道关闭的时候，一定有`A=A(描述符引用数=缓冲区引用数)`

实验内容

> Exercise 6.3 修改 user/pipe.c 中 pipeclose 函数中的 unmap 顺序与 user/fd.c 中 dup 函数中的 map 顺序以避免上述情景中的进程竞争情况。

实验提示

本质上，我们修改顺序是要保证这两件事：

- 开启管道的时候，第一时间让`A<A+B(描述符引用数<缓冲区引用数)`
- 关闭管道的时候，第一时间让`A=A(描述符引用数=缓冲区引用数)`

Exercise 6.4

该部分已经在6.2中有提示，故此处只放出实验要求。

> Exercise 6.4 根据上面的表述，修改_pipeisclosed函数，使得它满足 “同步读” 的 要求。注意 env_runs 变量是需要维护的。

Exercise 6.5

实验背景

现在要开始LAB6难度最大的一个任务了，那就是完成Spawn函数。

之前我们的Fork，是建立一个新进程，但是这个进程和父进程一模一样。

而Spawn函数，则是建立一个新进程，然后给这个新进程里面填上新的程序。也就是，运行一个新的任务。

这个任务的难点就在于，为了获取新的程序内容，我们需要解析ELF文件，然后把里面的内容加载到对应的地方。

所以，个人**推荐**大家自己实现两个新的函数，一个用于解析ELF，一个用于把内容给MAP到子进程的内存空间上。

实验内容

> Exercise 6.5 根据以上描述以及注释，补充完成 user/spawn.c 中的 int spawn(char *prog, char **argv)。

实验提示

- 首先，代码中已经给了我们打开文件的代码，其中的**返回值r就是fdnum**
- 通过num2fd，fd2data这两个函数，我们可以得到**ELF的文件头指针**
- 通过syscall_env_alloc获得一个子进程，通过`init_stack(child_envid,argv,&esp);`对这个子进程进行栈的初始化
- 通过自己编写的load_elf程序来解析，并把elf文件中关键内容给加载到子进程的内存空间上
- 在load_elf中，需要编写一个MAP的函数。

- **一定要去参考`lib/kernel_elfloader.c`中的内容**
- **一定要去参考`lib/env.c/load_icode_mapper()`中的内容**
- 在map函数里面，可以先采用`syscall_mem_alloc`来生成一个临时页，再用`user_bcopy`把内容拷贝到这一页中，然后用`syscall_mem_map`来把这一页给map到子进程中，最后用`syscall_mem_unmap`来去掉父进程对这一页的挂载

Exercise 6.6

实验背景

终于到我们的交互脚本了。

其中核心的一个函数就是runcmd，就类似于我们的终端，可以输入命令，运行函数。

我们需要补充的是对<,>,|等部分的解析与具体执行工作。

这部分描述，由于比较多**请参考完整的指导书**

实验内容

> Exercise 6.6 根据以上描述，补充完成 user/sh.c 中的 void runcmd(char *s)。

实验提示

- 在代码中，已经自带了一个`gettoken`函数，可以用于获取下一个单词，或者操作符。获取到的字符串指针保存在`t`中，可以直接使用。
- 对于<,>的运行过程，实际上代码中已经给了很多提示。可能会用到的函数有：
  - open(t,O_RDONLY) //或者O_WRONLY，分别代表只读或者只写
  - dup(fdnum,0)///或者1，根据注释而定
- 对于|的处理过程：
  - 用pipe(p)可以创建管道
  - 管道0是读端，1是写端
  - 可能使用的命令：dup(p[0],0);//另一端同理
  - 父子进程dup的端不同，关闭的顺序也不同，goto的地方也不同，注意看注释