实验（2）在xv6上实现pingpong程序，即两个进程在管道两侧来回通信。父进程将”ping”写入管道，子进程从管道将其读出并打印。子进程从父进程收到字符串后，将”pong“写入另一个管道，然后由父进程从该管道读取并打印。请将代码写在user/pingpong.c文件中。

要求：

a) 使用pipe()创建管道，详见实验原理；

b) 使用fork()创建子进程，注意根据返回值，判断父子进程；

c) 利用read(), write()函数对管道进行读写。

d) 请在user/pingpong.c中实现。

e) 修改Makefile，将程序添加到UPROGS。

1、思路：

pipe-syscall是这样的：int pipe(int p[]) Create a pipe, put read/write file descriptors in p[0] and p[1].

由于管道是单向流动的，所以两次调用pipe()创建两个管道，分别对应两个方向。使用fork()创建子进程，在子进程中先从p[0] read()再向p[1] write()，父进程中先从p[1] read() 再向 p[0] write()。

2、代码：

#include "kernel/types.h"

#include "user/user.h"

int main(int *argc*, char \**argv*[]) {

    int p[2];*//parent to child.*

    int q[2];*//child to parent.*

*// int pipe(int p[]) Create a pipe, put read/write file descriptors in p[0] and p[1].*

    pipe(p);

    pipe(q);

    int child\_pid=fork();

    if(child\_pid<0){printf("fork error!"); exit(-1);}

    if(child\_pid>0){*//parent*

        close(p[0]);*//close parent read*

        close(q[1]);*//close child write*

*// so, now p[1] -> q[0]*

        write(p[1], "ping", 16 );

        sleep(5);

        char buf[1024] = {"\0"};

        read(q[0], buf, 16);

        printf("%d: received %s\n",getpid(),buf);

        close(p[1]);*//release*

        close(q[0]);*//release*

        exit(0);

    }else{*//child*

        close(p[1]);*//close parent write*

        close(q[0]);*//close child read*

*// so, now q[1] -> p[0]*

        char buf[1024] = {"\*0"}; //string end with '\0'*

        read(p[0], buf, 16);

        printf("%d: received %s\n",getpid(),buf);

        write(q[1], "pong", 16 );

        close(p[0]);*//release*

        close(q[1]);*//release*

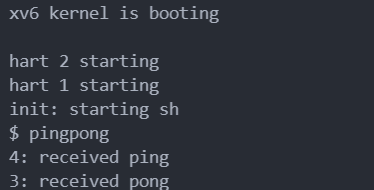
        exit(0);

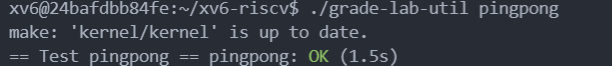
    }

    exit(0);

}

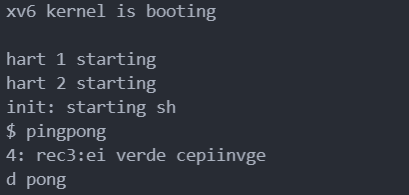
3、结果：





4、解释：如何使用close关闭描述符进行输入输出先后控制。

如果不进行先后控制，输出结果很有可能是并发的，就像这样：



原因是：调用fork()后，创建父子的task\_struct都需要耗费时间，于是无法确定父、子进程间谁将率先访问CPU，即谁将先从fork系统调用返回。

所以在我们代码中需要使用close(p[0]);//release 这种方式来关闭描述符，达到控制IO次序的目的。

在后面，可以使用锁来控制并发IO。

5、深入思考

实际上对于管道的处理，此处是使用双工的有名管道来模拟单工的匿名管道，体现在代码中就是对于管道p或q在某时刻只会使用一端。

但是pipe系统调用本身返回的就是两个可读可写的文件描述符，为什么不能把两个都使用起来呢？

于是又实现了一种写法，仅调用一次pipe：

int main(int *argc*, char \**argv*[]) {

    int pp[2];

    pipe(pp);

    int child\_pid=fork();

    if(child\_pid<0){printf("fork error!"); exit(-1);}

    if(child\_pid>0){*//parent*

        write(pp[1], "ping", 16 );

        sleep(5);

        char buf[1024] = {"\0"};*//string end with '\0'*

        read(pp[0], buf, 16);

        printf("%d: received %s\n",getpid(),buf);

        exit(0);

    }else{

        char buf[1024] = {"\0"};*//string end with '\0'*

        read(pp[0], buf, 16);

        printf("%d: received %s\n",getpid(),buf);

        write(pp[1], "pong",16);

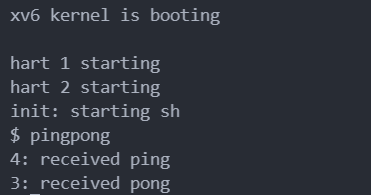
        exit(0);

    }

    exit(0);

}

输出：



但是这种写法也存在缺陷，就是不能使用独立的p管道和q管道进行类似于锁的控制，且fork调用的返回时机仍然不确定。

于是在代码中，针对parent process进行send msg之后sleep(5)挂起5个tick，来确保child process已经完成的“pong”这个msg的send。这种通过sleep(5)来手动控制并发的写法当然是不值得鼓励的。

二、了解管道模型，回答下列问题

1) 简要说明你是怎么创建管道的，又是怎么使用管道传输数据的。

答：通过pipe系统调用来获取管道的文件描述符，通过write系统调用写入文件描述符。

2) fork之后，我们怎么用管道在父子进程传输数据？

答：这就涉及到fork执行原理了。fork完全进行task\_struct的复制，所以在parent和child的task\_struct里面均是存在两个管道数组的（或一个管道数组，写法二）。

文件描述符是局部于进程的，但是向文件描述符中写入数据调用write，会先复制到内核的缓冲区，当有该打开文件关联的文件描述符被调用以read时，内核缓冲区的数据就会被复制出去，也就是到代码中的用户进程缓冲buf数组。

3) 试解释，为什么要提前关闭管道中不使用的一端？（提示：结合管道的阻塞机制）

答：在写法二中已经给出来只使用一个管道、使用sleep进行手动并发控制的写法。所以两个管道存在的意义仅仅是进行并发控制。

当然，如果结合管道的阻塞机制，可以将内核缓冲想象为管道的大小（实际上管道并不存在大小）。所以在一侧进行写入的时候，另一侧需要关闭写端口，只能进行读操作，来避免同时写入造成缓冲溢出异常。