哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2021 秋季

课程名称：操作系统

实验名称：xv6 与 Unix 实用程序

实验性质：课内实验

实验时间：2021/9/30 地点：T2210

学生班级：19 级计科 2 班

学生学号：1190303311

学生姓名：王志军

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2018 年 12 月

一、 回答问题

1. 阅读 sleep.c，回答下列问题

（1） 当用户在 xv6 的 shell 中，输入了命令“sleep hello world\n"，请问 argc 的值是多少，argv 数组大小是多少。

Argc 的值是 3，argv 数组大小是 3

（2） 请描述 main 函数参数 argv 中的指针指向了哪些字符串，他们的含义是什么。

Argv[0]指向了 sleep，是执行程序名或者命令，argv[1]指向参数，在这里就是 sleep 的时间.

（3） 哪些代码调用了系统调用为程序 sleep 提供了服务？

```
（4） sleep(ticks);
（5） exit(0);
（6） printf();
（7） atoi();
```

2. 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。

创建管道：首先定义一个大小为 2 的整型数组 p，然后调用 pipe(p)，就创建了一个管道，其中 p[0]是管道读出端的文件描述符,p[1]是管道写入端的文件描述符；

使用管道传输数据：使用 write 向 p[1]写入数据，write(p[1],data,n)，使用 read p[0]从管道读取数据，read(p[0],data,n)

（2） fork 之后，我们怎么用管道在父子进程传输数据？

创建管道之后 fork 一个子进程，子进程和父进程都有管道的两个描述符，父进程通过 p[1]和 write 向管道写入数据，子进程通过 p[0]和 read 从管道读取，由于从管道 read 时，如果管道没有数据，read 会阻塞进程直到数据写入管道或者没有进程写管道，因此直接读写即可。

（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）

管道的读和写端同时只能有一个被占用，当从管道读数据时，如果管道没有数据，就会将进程阻塞，等待数据写入管道，如果不关闭该进程中管道的写入端，

可能会因为重定位读取到进程自己写入到管道的数据，导致进程自己对管道循环读写，read 永远不会遇到文件结尾。所以要关闭管道中不使用的一端。

二、 实验详细设计

1、sleep：

处理参数然后系统调用 sleep

xv6-labs-2020 > user > C sleep.c > ...

```c
#include "kernel/types.h"
#include "user.h"

int main(int argc,char* argv[]){
    if(argc != 2){
        printf("Sleep needs one argument!\n"); //检查参数数量是否正确
        exit(-1);
    }
    int ticks = atoi(argv[1]); //将字符串参数转为整数
    sleep(ticks);              //使用系统调用sleep
    printf("(nothing happens for a little while)\n");
    exit(0); //确保进程退出
}
```

2、pingpong：

首先声明两个大小为 2 的 int 数组 p1,p2 储存两个管道的文件描述符,创建管道,p1 用于父进程向子进程传递 ping，p2 用于子进程向父进程传递 pong;

创建子进程，父进程中向 p1[1]写入”ping”,然后从 p2[0]等待读取；

子进程从 p1[0]读取父进程传来的数据，判断是否为”ping”，是则输出 pid 和数据，然后向 p2[1]写入”pong”,关闭管道文件描述符并退出；

```
    if(fork()==0)
    {
        char c[5];
        char k[] = "ping";
        char r[] = "pong";
        int n;
        n = read(p1[0],c,4);
        if(n<0)
        {
            fprintf(2,"read error from child\n");
            exit(1);
        }
        if(!strcmp(c,k))
        {
            fprintf(1,"%d: received %s\n",getpid(),c);
            n = write(p2[1],r,4);
            if(n!=4)
            {
                fprintf(2,"write error from child\n");
                exit(1);
```

父进程从 p2[0]获取数据，判断是否为”pong”并输出 pid 和数据，关闭文件描述符退出；

```
        char c[5];
        int n;
        char k[] = "pong";
        char r[] = "ping";
        n = write(p1[1],r,4);
        if(n!=4)
        {
            fprintf(2,"write error from parent\n");
            exit(1);
        }
        n = read(p2[0],c,4);
        if(n<0)
        {
            fprintf(2,"read error from parent\n");
            exit(1);
        }
        if(!strcmp(c,k))
        {
            fprintf(1,"%d: received %s\n",getpid(),c);
        }
    }
```

3、primes:

使用递归实现，递归函数为 prime,prime 参数为上一层的结果数组，和数组的大小，递归结束条件为参数数组大小为 1，也就是只剩最后一个数字；

prime 中先创建一个管道，然后创建一个子进程，父进程负责将参数数组的元素一个一个地写入到管道；

```
else//父进程，将这一层处理好的数据打包给子进程
{
    for(int i=1; i<count; i++)
    {
        int n;
        k=num[i];
        if((n=write(p[1],&k,4))!=4)
        {
            fprintf(2,"write error from parent\n");
            exit(1);
        }
    }
    close(p[1]);
    wait(&status);
    exit(0);
}
```

子进程负责从管道一个一个地读取数字，每读取一个判断是否对第一个读取的元素求余为 0，也就是素数筛选的条件，不是则记录数字，最后将数组和大小传入 prime 递归；

```
if(fork()==0)//子进程从管道读取int数组，并根据temp进
{
    close(p[1]);//读取之前必须先将write端关闭
    int counter=0,buf;
    while(read(p[0],&buf,4)!=0)
    {
        if(buf%temp!=0)
        {
            num[counter]=buf;
            counter++;
        }
    }
    primes(num,counter);
    wait(&status);
    exit(0);
}
```

每次递归都只输出数组的第一个数字。

4、find:

ls.c 修改后得到；

首先定义 fmtname 函数，用于把路径中的文件名提取出来，从后往前扫描，遇到’/’就停止；

Find 函数，当判断当前目录并不是要找的文件名后，通过 dirent 结构体类型的对象 de 扩展路径名，对于’.’和’..’直接跳过，即本级和上级目录，避免跳入死循环，扩展之后递归调用 find。

```
strcpy(buf, path);
p = buf+strlen(buf);
*p++ = '/';
while(read(fd, &de, sizeof(de)) == sizeof(de)){
  if(de.inum == 0 || de.inum == 1 || strcmp(de.name,".") == 0 || strcmp(de.name,"..") =
    continue;
  memmove(p, de.name, strlen(de.name));
  p[strlen(de.name)] = 0;
  find(buf,filename);
}
break;
```

5、xargs:

处理字符串，将|前的参数加到后面的参数列表里

Echo 的输出被 sh.c 重定位到了标准输入，xargs 的参数是”| xargs echo ….”，先将 echo 后面的参数存起来，然后从标准输入读取前面程序的输出，作为参数再添加到后面，最后调用 exec 执行程序。

```
    params[k++]=argv[i];
while((n=read(0,line,1024))>0)//从标准输入读取，sh.c已经
{
    if(fork()==0)
    {
        char *param=(char*)malloc(sizeof(line));
        int index=0;
        for(int i=0; i<n;i++)
        {
            if(line[i]==' '||line[i]=='\n')
            {
                param[index]=0;
                index=0;
                params[k++]=param;
                param=(char*)malloc(sizeof(line));
            }
            else
                param[index++]=line[i];
        }
        free(param);
        params[k]=0;
        exec(cmd,params);
    }
    else
```

三、 实验结果截图

请填写

```
1190303311@OSLabExecNode0:~/xv6-labs-2020$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.9s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.9s)
== Test find, recursive == find, recursive: OK (2.6s)
== Test xargs == xargs: OK (2.8s)
== Test time ==
time: OK
Score: 100/100
```