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

实验设计报告

开课学期：大三上学期

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：2021.10.11 地点：T2210

学生班级：1901105

学生学号：190110509

学生姓名：王铭

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 为 3 argv 数组大小为 4，最后一个参数为空

（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]指向程序名 sleep，argv[1]指向第一个参数 hello，argv[2]指向第二个参数 world\n，argv[3]为空指针

（3） 哪些代码调用了系统调用为程序 sleep 提供了服务？

sleep(ticks)提供了系统调用

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。

Int p[2]后，用提供的系统调用函数 pipe(p)创建管道，返回的 p[0]是管道的读端，p[1]是管道的写端。通过系统调用函数 write()，传入写文件的描述符为 p[1]。通过系统调用函数 read()，使用 p[0]作为读文件的文件描述符，读取管道中的数据

（2） fork 之后，我们怎么用管道在父子进程传输数据？

通过判断 fork 的返回值，若返回值大于 0，则表示是父进程，调用 write()向 p1[1]写入 ping，随后调用 read()读取 p2[0]读取子进程的数据。否则为子进程，调用 read()读取 p1[0]中父进程传入的 ping，随后调用 write()向 p2[1]写入 pong。

（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）

read 函数会在读取完管道的数据或是所有写端关闭后返回，write 会在所有读端关闭或写完数据后再返回。

若没有进行写操作且未关闭写端，那么读端可能会一直处于等待读取的状态，从而陷入阻塞导致程序卡死，因而要关闭不使用的写端。若已经没有进程在进行读操作且未正在写的进程未关闭读端，当通道写满数据后，写的这端会进入阻塞，等待读端读取，但由于无读操作，故写端会永远阻塞。

二、 实验详细设计

1.sleep

修改 makefile 即可运行。

2.pingpong

创建两个管道 p1,p2，通过 fork 函数以及返回值判断父子进程，p1 用于父进程写入 ping 并从子进程读出 ping,p2 用于子进程写入 pong 并由父进程读出 pong。

核心代码如下：

```
pipe(p1);
pipe(p2);
if((pid = fork()) ==  0) // 子进程
{
close(p1[1]);
    size1 = read(p1[0],buffer2,5);
    if(size1 < 0)
    {
    printf("child read error\n");
    exit(1);
    }
printf("%d: received %s\n",getpid(),buffer2);
close(p1[0]);
close(p2[0]);
buffer2 = "pong";
size1 = write(p2[1],buffer2,5);
if(size1 < 0)
{
    printf("child write error\n");
    exit(1);
}
close(p2[1]);
}
```

```
else if(pid > 0){                    //父进程
close(p1[0]);
buffer1 = "ping";
size2 = write(p1[1],buffer1,5);
if(size2 < 0)
{
    printf("parent write error\n");
    exit(1);
}
close(p1[1]);
close(p2[1]);
size2 = read(p2[0],buffer1,5);
if(size2 < 0)
{
    printf("parent read error\n");
    exit(1);
}
printf("%d: received %s\n",getpid(),buffer1);
close(p2[0]);
wait(&status);
}
```

3.primes

通过递归实现质数的筛选。利用管道，我们可以一次输出一个质数，具体递归过程如下：

读取传入递归函数上一级创建的管道的读端的参数，读取第一个数（质数）

并输出，随后通过 fork 创建进程，若为父进程则依次读取传入参数对应管道中的数，将能除尽该质数的数过滤，把剩余的数输入新创建的管道。若为子进程，递归调用即可。

递归函数代码如下：

```
void isPrime(int fd){
    int p[2];
    int pid,status;
    pipe(p);
    int prime;
    int n = read(fd,&prime,sizeof(int));
    if(n == 0) return ;
    if((pid = fork()) > 0){
        close(p[0]);
       printf("prime %d\n",prime);
        while(n){
            int x;
            n = read(fd,&x,sizeof(int));
        if(x % prime != 0)
                write(p[1],&x,sizeof(int));
            }
        close(p[1]);
    }
    else if(pid == 0){
        close(p[1]);
        isPrime(p[0]);
        close(p[0]);
    }
    else{
    printf("Fork error!\n");
    }
    wait(&status);
    return ;
}
```

Main 函数：

```
int main(int argc,char* argv[]){
    int pid,status;
    int p[2];
    pipe(p);
    if((pid = fork()) > 0){ //父进程
    close(p[0]);
    for(int i = 2;i < 36;i++)
            write(p[1],&i,sizeof(int));
    close(p[1]);
    }
    else if(pid == 0){
    close(p[1]);
    isPrime(p[0]);
        close(p[0]);
    }
    else{
    printf("Fork error\n");
    exit(1);
    }
    wait(&status);
    exit(0);
}
```

4.find

参考 ls 的代码，使用递归完成匹配。

可以使用 dirent 和 stat 两个结构体。首先用 open 打开传入的参数路径对应的文件得到文件描述符 fd，随后调用 fstat(fd,st)函数，根据结构体 st 中的 type 判断传入的是文件还是目录。若为文件，直接匹配要匹配的参数，若为目录，迭代调用 read(fd,&de,sizeof(de))可以得到该目录下的每一个文件名 de.name，过滤掉.和..后递归调用该函数去匹配该目录下的每一个文件或目录是否匹配输入的参数。

递归函数如下，其中 path 为输入的路径，object 为要匹配的路径：

```c
void find(char* path,char* object){
    int fd;
    char buffer[512], *p;
    struct dirent de;
    struct stat st;
    if((fd = open(path,0)) < 0){
       //  printf("open file error!\n");
        return ;
    }
    if(fstat(fd,&st) < 0){
       //  printf("read status error!\n");
        close(fd);
        return ;
    }
    switch(st.type){
        case T_FILE:
           // printf("filename:%s\n",path);
            if(match(path,object) == 1)
              printf("%s\n",path);
            break;
        case T_DIR:
            strcpy(buffer,path);
            p = buffer + strlen(buffer);
            *p++ = '/';
            while(read(fd,&de,sizeof(de)) == sizeof(de)){
                if(de.inum == 0) continue;
                if(strcmp(de.name,".") == 0 || strcmp(de.name,"..") == 0) continue;
                memmove(p,de.name,N);
                p[N] = 0;
                find(buffer,object);
            }
            break;
    }
    close(fd);
    return ;

}
```

匹配函数 match 代码如下：

```c
int match(char* path,char* object){
    char *q,*p;
    // printf("path:%s\n",path);
    q = object;
    p = path + strlen(path);
    for(;p >= path;p--){
        if(*p == '/') break;
        else continue;
    }
    p++;
    if(path + strlen(path) - p != strlen(object)) return 0;
    while((p <= path + strlen(path))){
      if(*p != *q) return 0;
      p++;
      q++;
    }
    return 1;
}
```

5.xargs

要执行的命令传递到 main 函数的参数 argv 中，通过在 while 循环中用 fork 创建子进程，用得到新的输入与 argv 中的参数进行拼接并作为新的参数调用 exec 函数。

代码如下:

```
int main(int args,char* argv[]){
    int status;
    while(1){
        char buffer[512];
        char* s = gets(buffer,512);
        if(strlen(s) == 0) break;
        s[strlen(s) - 1] = '\0';
        if(fork() == 0){
            char* arg[MAXARG] = {"0"};
            int i;
            for(i = 1;i < args;i++){
                arg[i-1] = argv[i];
            }
            arg[i - 1] = s;
            if(exec(argv[1],arg) < -1) printf("error");
            close(0);
            exit(0);
        }
        wait(&status);
    }
    exit(0);
}
```

三、 实验结果截图

1.Sleep

```
$ sleep 10
(nothing happens for a little while)
```

2.Pingpong

```
$ pingpong
5: received ping
4: received pong
```

3.Primes

```
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

4.find

```
$ echo > b
$ echo > a/b
$ find . b
./a/b
./b
$
```

5.xargs

```
$ echo 3 4 | xargs echo 1 2
1 2 3 4
$
```

通过测试截图：

```
[ming@localhost xv6-labs-2020]$ sudo ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (2.4s)
== Test sleep, returns == sleep, returns: OK (1.5s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.6s)
== Test pingpong == pingpong: OK (1.7s)
== Test primes == primes: OK (1.9s)
== Test find, in current directory == find, in current directory: OK (2.4s)
== Test find, recursive == find, recursive: OK (2.9s)
== Test xargs == xargs: OK (2.6s)
== Test time ==
time: OK
Score: 100/100
```