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

实验设计报告

开课学期：2021 年秋季学期

课程名称：

实验名称：操作系统第一次试验

实验性质：课内实验

实验时间：　　地点：

学生班级：19 级 2 班

学生学号：190110209

学生姓名：蒋晨阳

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。

答：argc 的值为 3，argv 数组大小为 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。

答：argv 中的指针指向了由命令行输入的字符串，或者是传递到该函数的参数。argv[0]为指向命令名，argv[1]、argv[2]等指向参数，最后一个指向 NULL。如果该程序名字为 A，则在命令行中输入 “A arg1 arg2 arg3”，那么 argv 中存有四个字符串指针，分别指向“A”、“arg1”、“arg2”和“arg3”，同时 argv 的第 5 个指针指向 NULL。

（3） 哪些代码调用了系统调用为程序 sleep 提供了服务？

答：sleep(ticks); 调用了系统提供的 sleep()函数。

exit(0); 调用了系统提供的 exit() 函数

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。

答：调用系统提供的 pipe 函数创建管道。使用 pipe 函数后可以得到大小为 2 的数组，数组第中存储两个文件描述符，第一个文件描述符用于读取管道，第二个文件描述符用于写管道。

可以使用 read()和 write()函数来读写数据，其函数原型分别为 read(int fd, void* buffer, int n)和 write(int fd, void* buffer, int n)，其中 fd 为文件描述符，buffer 为待传送数据起始位置指针，n 为最大传送数据大小。read 用来从 fd 指向的管道中读取最多 n 字节到 buffer 中，write 用来从 buffer 中向 fd 指向的管道写入 n 最多字节。

（2） fork 之后，我们怎么用管道在父子进程传输数据？

答：需要首先在父进程中使用 pipe()获得文件描述符，fork 之后子进程继承了父进程的环境，所以也能得到管道的文件描述符。假设文件描述符在数组 p[2]中，如果需要父进程向子进程写数据，那么需要在父进程端 close(p[0])，关掉管道读；在子进程端 close(p[1])。随后在父进程中调用 write(p[1], buf, n)向管道中从 buf 最多写入 n 个字符，在子进程中调用 read(p[0],buf,n)从管道中最多读取 n 个字符到 buf 中。

如果需要实现父子进程之间互相读写的话，可以利用管道的半双工工作

（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：假设需要从父进程向子进程中写入数据，如果不关闭子进程的写端口，则在父进程停止输入且子进程读取所有数据之后，由于存在未关闭的写端口，子进程的 read()函数会陷入阻塞。而如果关闭了子进程的写端口，则当读取所有数据之后，管道会返回文件终止符，则可以继续执行 read()之后的函数。

二、 实验详细设计

（1） 为 xv6 实现 UNIX 程序 sleep

答：sleep 接受两个参数，如果参数不等于两个，则报错。argv[0]参数为“sleep”，argv[1]参数为睡眠时长。使用 atoi 函数将字符串形式的睡眠时长转化为整数，随后调用系统函数 sleep，实现指定时长的睡眠。在程序退出前输出”(nothing happens for a little while)”即可。

（2） 在 xv6 上实现 pingpong 程序

答：该程序的关键是使用两个管道实现两个进程之间的通信。

调用 pipe 函数时，会返回两个文件描述符，分别用于管道的读写，在此时 fork 出子进程之后，子进程也会得到两个文件描述符的拷贝，从而父子进程都可以对管道进行读写，实现进程间的通信。

首先在父进程（简写为 F）中调用两次 pipe 函数，得到两个管道，其中一个用于向子进程（简写为 S）传送信息，一个用于从 S 中获取信息。两个管道的读写描述符分别存入 p1[2]和 p2[2]两个数组中。

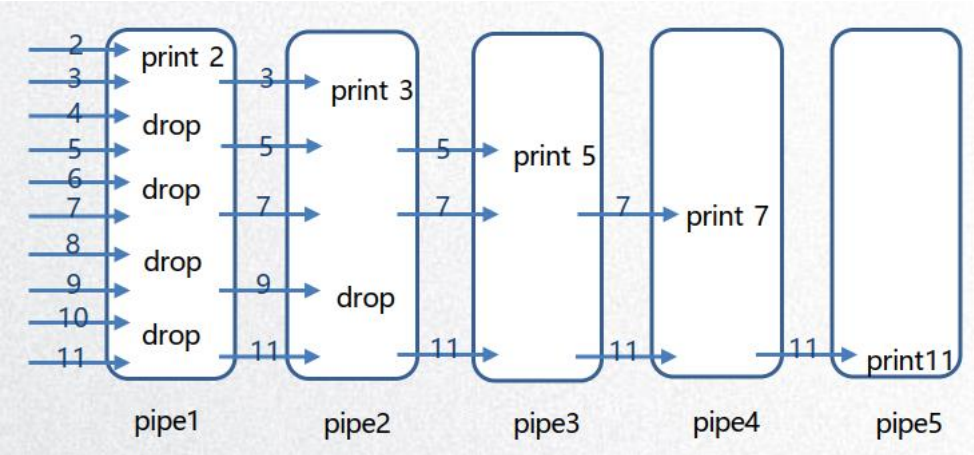
随后 F 调用 fork 函数，分离出进程 S。在父进程中关闭 p1[0]和 p2[1]，即关闭管道 1 的读端口和管道 2 的写端口。在子进程中关闭 p1[1]和 p2[0]，即管道 1 的写端口和管道 2 的读端口。随后父进程通过 write 函数由 p1[1]向子进程写入”received ping”，子进程通过 read 函数由 p1[0]从父进程中读出信息，并打印。随后子进程通过 write 函数与 p2[1]向父进程写入”received pong”，父进程通过 p2[0]读出并打印。随后在父子进程中关闭管道的所有端口。

随后父进程等待子进程退出后，结束程序。

（3） 在 xv6 上使用管道实现“质数筛选”

答：该程序的关键是递归的调用 fork 函数。并利用管道进行通信。

该程序的原理为由进程 F 向子进程 S 传入一串数字，如 2、3、4……34、35，随后子进程 S 将读取到的第 1 个数字 2 作为素数，并打印出来，随后开启子进程，并从第 2 个数字开始，判断当前数字是否为第 1 个数字的倍数，若为第 1 个数字的倍数，则必为合数，丢弃，否则将当前数字传给下一个子进程。S 的子进程得到第一个数字之后，进行同样的判断。直到所有数字判断完毕为止。



首先实现 child_process 函数，该函数的入口参数 p0 和 p1 分别为其与父进程之间管道的读写端口，由于仅需要从父进程接收数据，故关闭端口 p1。随后从 p0 端开始读取第一个数字，并输出。接下来，如果能够继续读取得到数字，则开启一个管道，将管道数据存入 p2[2]中，并 fork 出一个子进程，在子进程中调用 child_process 函数，并传入 p2[0]和 p2[1]作为参数。大体结构如下

```
void child_process(int p0, int p1) {
    // 读取第一个素数并打印，否则退出
    if (get_prime(first_prime, p0))
        print(first_prime);
    else
        exit();
```

```
    // 读取接下来的数字
    int has_child = 0;
    int p2[2];
    while (get_prime(prime, p0)) {
        if (prime % first_prime == 0) continue;
        // 当第一次读取到不是 first_prime 的倍数的数时
        // 创建管道和进程
        if (has_child == 0) {
            has_child = 1;
            pipe(p2);
            if (fork() == 0) {
                child_process(p2[0], p2[1]);
            }
        }
        // 向子进程传入数字
        write_prime(prime, p2[1]);
    }
    wait();
    exit();
}
```

接着实现一个 root_process 函数，其为主进程，用于开启第一个子进程并在子进程中调用 child_process，并传入 2-35。在 main 函数中调用 root_process 即可。

（4） 在 xv6 上实现用户程序 find

答：通过深度优先搜索的方式在指定目录及其所有子目录下寻找指定文件。

首先实现 get_name 函数，给定一个路径，返回该路径最末尾的文件名、

关键在于实现 find 函数。find 含有两个参数，path 指向目录字符串的首地址，fn 指向待查找文件名的首地址。首先打开当前目录 path，获取文件描述符 fd，随后根据 fd，通过 fstat 获取文件状态，存入 stat 结构体 st 中。

接下来判断当前 get_name(path)与 fn 是否相同，如果相同输出 path。随后判断当前 path 指向的文件类型，如果 path 指向的为目录，则从 path 指向的目录中读取文件名，并与 path 拼接成新的 path，继续调用 path 函数。实现递归查找。

（5） 在 xv6 上实现用户程序 xargs

xargs 从标准输入中读取行，并为每行运行一次指定的指令，且将该行作为命令的参数。

采用 fork 与 exec 函数组合的方式实现上述功能。大致结构如下

```
int main(int argc, char *argv[]) {
    while(readlines(lines)) {
        char *new_argv[MAXARG] = get_new_argv(argv, lines);
        if (fork() == 0) {
            exec(args[1], new_argv);
        }
        release(new_argv);
        wait(0);
    }

    exit(0);
}
```

每次从标准输入中读取一行，通过 get_new_argv 函数（仅为示意，实际代码中并未实现为独立函数），将读取的一行解析称若干个独立参数，并放在原始参数数组 argv 之后，得到新的 argv 数组，随后 fork 出子进程并调用 exec 函数即可。

其中较麻烦的部分在于解析每行输入部分。为此我实现了一个 readlines 函数（与上段中 readlines 函数作用不同），负责从标准输入中读取一行并解析成字符串指针数组。该函数每次读取一个字符，忽略除回车外的空白字符。首先读取行首的空白字符并忽略。随后持续读取字符，直到遇到空白字符，在单词后加上结束符，并为其分配一段内存空间，用于存储读取到的单词，将指针存入 char *nargv[]中。随后读取单词后的所有空白字符，直到遇到新的字符，或者遇到回车，则退出该函数。

在程序退出前，还需要释放所有申请的空间，并等待所有子进程结束。

三、 实验结果截图

```
[cs@localhost xv6-labs-2020]$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.1s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.0s)
== Test xargs == xargs: OK (1.1s)
== Test time ==
time: OK
Score: 100/100
[cs@localhost xv6-labs-2020]$
```