

# Lab1 报告

刘鸣霄 自 12 2021010584

4 月 15 日

Lab1: Xv6 and Unix utilities 实验通过在 xv6 操作系统中编程，帮助我们初步熟悉 xv6 操作系统及一些系统调用。

## 1. Boot xv6

输入命令 `make qemu` 让 xv6 在 qemu 上跑起来。

## 2. [sleep](#)

这个程序的功能是让系统休眠指定的时间，主要使用系统调用 `sleep`。

先判断 `argv` 中是否具有两个参数，如果缺少参数则输出提示信息。然后，将 `argv[1]` (字符串)用提供好的 `atoi` 函数转换为 `int`，然后以其为参数调用系统调用 `sleep`。

*[点击标题可以跳转到代码](#)*

## 3. [pingpong](#)

这个程序实现：父进程向子进程发送一个字符的信息，子进程成功接收该字符后打印一条提示信息，然后再向父进程发送该字符，父进程收到字符后再打印一条提示信息。这个程序主要使用 `fork` 和 `pipe`，使用 `fork` 创建子进程，使用 `pipe` 创建管道进行进程间通信。

首先调用 `pipe` 创建一对管道 `p1,p2`，记录他们读端和写端的文件描述符。`p1` 用于父进程向子进程传输信息，`p2` 用于子进程向父进程传递信息。由于管道中的数据只能单向流动，因此必须创建两个管道。

然后调用 `fork` 创建进程。通过其返回值是否为 0 判断进程是父进程还是子进程。对于每个进程，为了保证数据安全传输并且节省文件描述符，首先关闭不用的管道端口，对于父进程而言是 `p1` 的读端和 `p2` 的写端，子进程则是 `p2` 的读端和 `p1` 的写端。

父进程调用 `write` 向 `p1` 的写端写一个字符，通过其返回值是否等于发送的数据长度判断是否发送成功。

子进程调用 `read` 从 `p1` 的读端接收该字符。如果数据还没来，`read` 会阻塞进程。通过 `read` 的返回值是否等于其读取的数据长度判断是否读取成功。若读取成功，子进程打印一条信息（`getpid` 可以获取进程的 `PID`），随后调用 `write` 向 `p2` 的写端写入接受的字符。

父进程调用 `read` 接收到信息后，向屏幕上打印一条信息。

#### 4. [primes](#)

这个程序比较复杂，是一个并行素数筛。其原理是创建多个进程成为多个素数筛，每个素数筛得到的第一个数（是素数）作为自己主元，对于后来得到的数，把主元的倍数删除掉，不是主元的倍数传递给下一个筛子。最后剩下的数就都是素数。

用递归的方式实现，不过是递归地创建子进程。主进程向第一个素数筛传递 2~35 的数，每一个子进程（素数筛）通过“左”管道从上一个素数筛读数、得到主元、筛选、将剩余的数通过“右”管道传给下一个素数筛（为了方便描述，数字是从左往右传递的）。

对于第一个子进程，以下步骤依次进行：

(1)将右管道（目前是其父进程右管道）变为左管道，接受父进程传来的信息。通过转移文件描述符实现，因为现在父子进程的文件描述符相同；

(2)读取父进程传来的第一个数作为主元，并打印。如果父进程不传递主元(即 `read` 返回 0)，说明筛选结束，子进程 `exit`；

(3)创建右管道，以向将要创建的子进程发送信息；

(4)调用 `fork` 创建孙进程，（子进程 `continue`，原进程继续向下执行），孙进程先关闭整个左管道（因为子进程左管道和孙子已经没有关系了），之后孙进程跳转到(1)；

(5)从父进程读数，筛选，传递给孙进程。

这个题目用时较长。下面是我做这个实验犯的一些错误：

(1)一个指针的小错误。主进程传递数字时的地址应为 `(char*)(numbers+i)`，我写成了 `(char*)numbers+i` 导致传递的数字错误。

(2)在 `fork` 后子进程没有关闭左管道读端，因此导致文件描述符迅速耗尽而报错。

#### 5. [find](#)

这个程序的功能是在一个目录下查找特定文件名的文件，输出其路径，涉及 `xv6` 的文件系统及相关的一些系统调用。这个程序与 `ls` 程序比较像，参照 `ls` 修改即可。

调用 `open` 打开查找的文件或目录，调用 `fstat` 获取文件状态，根据文件类型：

(1)如果是文件或者设备，直接比较目标名称和文件名称是否相同。

(2)如果是文件夹（目录），调用 `read` 依次读取每一个目录项，然后递归调用 `find` 进行查找（注意不要对“.”和“..”查找）。

#### 6. [xargs](#)

这个程序涉及系统调用 `exec`。这里的 `xargs` 与 `linux` 的略有不同，其功能是从标准输入读取行，之后将行拼接到后方命令的末尾，然后执行。

实现思路是，先准备好 `xargs` 后方命令已有的参数列表 `args` (`args[i-1] = argv[i]`)，然后不断从标准输入中读取字符串，拼接到 `args` 后方。我做作业的时候发现一次读取的并不一定是完整的一行，因而需要对读到的字符串逐个字符地处理：

(1)是回车 `'\n'`，说明一行结束，应该执行一次命令。将缓冲区内的字符串添加到 `args` 后面，然后 `fork` 一个子进程调用 `exec(args[0], args)` 运行一次程序，即执行一次命令。

(2)是空格 `' '`，说明一个参数结束，应该切换到准备下一个参数。将缓冲区的字符串添加到 `args` 后面，令 `args` 的参数个数加一，重置缓冲区（事实上，这些操作是通过用指针在缓冲区内记录参数地址和在缓冲区内设置 `\0` 来共同完成的）。

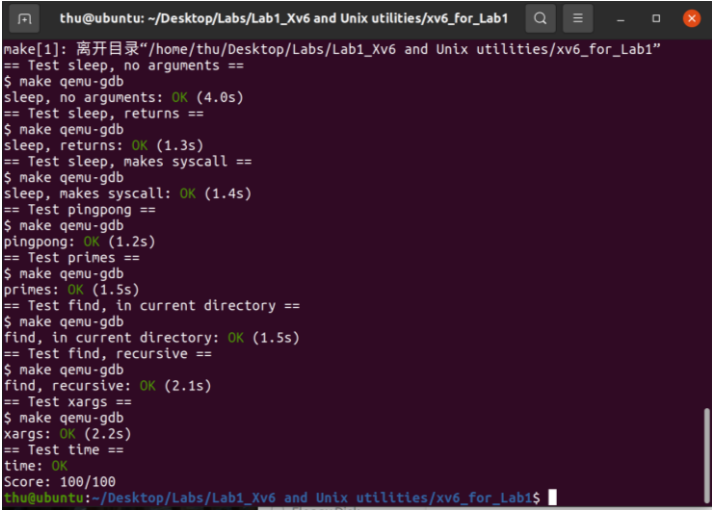
(3)是其他字符，就添加到缓冲区。

## 7. Time

该实验用时约 7 小时，主要在 `primes` 上花费了较多时间。

通过阅读 `xv6` 手册第一章和做实验，学到了一些关于 `xv6` 操作系统的初步知识（例如进程间通信，文件操作等），颇有收获。

## 8. Make grade



```
thu@ubuntu: ~/Desktop/Labs/Lab1_Xv6 and Unix utilities/xv6_for_Lab1
make[1]: 离开目录"/home/thu/Desktop/Labs/Lab1_Xv6 and Unix utilities/xv6_for_Lab1"
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (4.0s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.3s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.4s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.2s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.5s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.5s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (2.1s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (2.2s)
== Test time ==
time: OK
Score: 100/100
thu@ubuntu:~/Desktop/Labs/Lab1_Xv6 and Unix utilities/xv6_for_Lab1$
```

## 附录：代码

```
//sleep
//点击上方的 sleep 可以返回文字部分，下同。
#include ...

int
main(int argc, char *argv[])
{
    //如果参数不够则输出提示信息
    if(argc<2){
        fprintf(2, "Usage: sleep [time]\n");
        exit(1);
    }
    //将字符串转换为数字
    int time = atoi(argv[1]);
    //调用系统调用 sleep, 以 time 为参数
    sleep(time);
    exit(0);
}

//pingpong

#include ...

int
main(int argc, char *argv[])
{
    int p1[2],p2[2]; //管道读端和写端的文件描述符
    char buffer[1] = {0}; //子进程接受信息的缓冲区
    char info[1] = {0x30}; //传输的信息

    //创建两个管道
    pipe(p1);
    pipe(p2);
    if(fork() == 0){ //创建子进程
        //子进程
        //首先关闭不用的管道端口。子进程从 p1 读，向 p2 写。
        close(p1[1]);
        close(p2[0]);
        if(read(p1[0], buffer, 1) != 1){ //子进程接收错误
            fprintf(2,"child read error\n");
            exit(1);
        }
        //子进程接收到信息，输出文字
        printf("%d: received ping\n", getpid());
    }
```

```

    if(write(p2[1], buffer, 1) != 1){ //子进程发送错误
        fprintf(2,"child write error\n");
        exit(1);
    }
}
else{
    //父进程
    //关闭不用的管道端口。父进程向 p1 写，从 p2 读。
    close(p1[0]);
    close(p2[1]);
    if(write(p1[1], info, 1) != 1){ //父进程发送错误
        fprintf(2,"parent write error\n");
        exit(1);
    }
    if(read(p2[0], buffer, 1) != 1){ //父进程接收错误
        fprintf(2,"parent read error\n");
        exit(1);
    }
    //父进程接收到消息，输出文字
    printf("%d: received pong\n", getpid());
}
exit(0);
}

```

---

```

//primes

```

```

#include ...

```

```

static int numbers[34]; //储存 2~35 的数字

```

```

int

```

```

main(int argc, char *argv[])

```

```

{

```

```

    int leftp[2], rightp[2]; //每个进程的左管道，右管道

```

```

    int buffer[1]; //读入整数的缓冲区

```

```

    int res; //read 的返回值，读取数据的长度

```

```

    int i;

```

```

    int pivot[1]; //每个素数筛的“主元”

```

```

    for(i=0; i<34; ++i){

```

```

        numbers[i] = i+2; //存储 2~35

```

```

    }

```

```

    pipe(rightp); //创建主进程右管道

```

```

    if(fork() == 0){ //开辟第一个素数筛

```

```

while(1){
    //每一个素数筛：
    //父进程的右管道变为自己的左管道，通过转移文件描述符
    leftp[0] = rightp[0];
    leftp[1] = rightp[1];
    close(leftp[1]); //关闭左管道的写端，只从左管道读
    res = read(leftp[0], pivot, sizeof(int)); //读取主元
    if(res < 0){ //读取错误
        fprintf(2,"read error\n");
        close(leftp[0]);
        exit(1);
    }
    else if(res == 0) {
        //发现父进程不能给自己传递主元，说明筛选结束，直接结束进程
        close(leftp[0]);
        break;
    }
    else {
        //打印主元
        printf("prime %d\n", *pivot);
    }
    pipe(rightp); //创建右管道
    // 备注：有些注释掉的 printf 语句是用于 debug 的
    // printf("%d %d %d %d\n", leftp[0], leftp[1], rightp[0], rightp[1]);
    if(fork() == 0) {
        //新的素数筛（子进程）：
        //关闭父进程左管道的读端，父进程的左管道和自己没有关系。
        close(leftp[0]);
        continue; //跳回循环开始处
    }
    close(rightp[0]); //父进程：关闭右管道读端。
    while(1){
        //不断从左管道读数，进行筛选
        res = read(leftp[0], buffer, sizeof(int));
        if(res < 0){ //读数错误
            fprintf(2,"read error\n");
            close(leftp[0]);
            exit(1);
        }
        else if(res == 0) {
            //左管道没有数传给自己了，进程可以结束了
            close(rightp[1]);
            close(leftp[0]);
            break;
        }
    }
}

```

```

    }
    else{
        // 如果传来的数不是主元的倍数，就传给下一个素数筛
        // printf("prime %d: received %d\n", *pivot, *buffer);
        if(*buffer % *pivot != 0){
            write(rightp[1], buffer, 4);
            // printf("prime %d: send %d\n", *pivot, *buffer);
        }
    }
}
wait(0); // 同步
break;
}
}
else{
    //主进程:
    close(rightp[0]); // 关闭主进程的右管道的读端。
    for(i=0; i<34; ++i){
        // 将 2~35 传递给第一个素数筛
        write(rightp[1], numbers+i, sizeof(int));
        // printf("main: send %d\n", numbers[i]);
    }
    close(rightp[1]); // 关闭右管道写端。
    wait(0); // 同步
}

exit(0);
}

```

---

```

//find

```

```

#include ...

```

```

// ls 提供: 返回路径中的文件名

```

```

char*

```

```

fmtname(char *path)

```

```

{

```

```

    static char buf[DIRSIZ+1];

```

```

    char *p;

```

```

    // Find first character after last slash.

```

```

    for(p=path+strlen(path); p >= path && *p != '/'; p--)

```

```

    ;

```

```

    p++;

```

```

    if(strlen(p) >= DIRSIZ)
        return p;
    memmove(buf, p, strlen(p));
    //memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
    buf[strlen(p)] = 0;
    return buf;
}

void
find(char* path, char* target){

    char buf[512], *p; // 存储路径的缓冲区和操作指针
    int fd; // 打开的文件的描述符
    struct dirent de; // 目录项信息
    struct stat st; // 文件信息

    if((fd = open(path, 0)) < 0){ // 打开文件错误
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    if(fstat(fd, &st) < 0){ // 读取文件状态错误
        fprintf(2, "find: cannot stat %s\n", path);
        close(fd);
        return;
    }

    //printf("find in %s, type %d\n", path, st.type);

    switch(st.type){ // 根据文件类型
    case T_DEVICE:
    case T_FILE:
        //设备和文件:
        if(strcmp(target, fmtname(path))==0){
            // 如果文件名相符, 输出路径
            printf("%s\n", path);
        }
        break;

    case T_DIR:
        //文件夹:
        if(strlen(path) + 1 + DIRSIZ + 1 > sizeof(buf)){
            // 路径太长
            printf("find: path too long\n");
        }
    }
}

```



```

        break;
    }
    //处理路径字符串, 拷贝到 buf 并加斜杠
    strcpy(buf, path);
    p = buf+strlen(buf);
    *p++ = '/';
    while(read(fd, &de, sizeof(de)) == sizeof(de)){ //读取每一个目录项
        if(de.inum == 0)
            continue;
        // 将目录项的名称拷贝到路径结尾, 在字符串结尾添加\0
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        if(strcmp(fmtname(buf), ".") != 0 && strcmp(fmtname(buf), "..") !=
0){ //如果不是.和..
            find(buf, target); //递归调用 find 查找
        }
    }
    break;
}
close(fd);
}

int
main(int argc, char *argv[])
{
    int i;
    char path[DIRSIZ];

    if(argc < 3){ //参数过少
        printf("Usage: find [dir] [name]\n");
        exit(0);
    }
    //在给定的目录下, 依次查找给定的文件名
    memmove(path, argv[1], DIRSIZ);
    for(i=2; i<argc; i++)
        find(path, argv[i]);
    exit(0);
}

```

---

```

//xargs

#include ...

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

```

```

char *args[MAXARG+1]; //后方命令的参数列表
char line[512]; //储存 read 的数据
char buffer[512]; //缓冲区
char* tail = buffer; //buffer 中的指针，记录对应参数字串的首地址
int i, n, j, k = 0;

if (argc < 2) { // 参数太少
    fprintf(2, "Usage: xargs command [args...]\n");
    exit(1);
}

for (i = 1; i < argc && i <= MAXARG - 1; i++) {
    args[i-1] = argv[i]; //先准备 xargs 后方命令现有的参数列表
}

while ((n = read(0, line, sizeof(line))) > 0) { //从标准输入读
    for(j=0; j<n; ++j){
        if (line[j] == '\n'){ //如果是回车
            buffer[k++] = '\0'; // 最后一个参数末尾\0
            if(i >= MAXARG){ // 参数太多
                fprintf(2, "xargs: too many args");
                exit(1);
            }
            args[i-1] = tail; //最后一个参数字串的首地址
            args[i] = 0;
            k = 0; // 重置缓冲区和参数个数
            tail = buffer;
            i = argc;
            if (fork() == 0) { // fork 一个子进程执行命令。
                exec(args[0], args);
                // exec 不会在此返回，否则说明出错了
                fprintf(2, "xargs: failed to execute command %s\n", args[0]);
                exit(1);
            }
            wait(0);
        }
        else if(line[j] == ' '){ //如果是空格
            buffer[k++] = '\0'; // 一个参数的结尾
            if(i >= MAXARG){ //参数太多
                fprintf(2, "xargs: too many args");
                exit(1);
            }
            args[i-1] = tail; //设置该参数字串的首地址
            ++i; //参数列表增加 1
        }
    }
}

```

```
        tail = buffer + k; //下一个参数字符串的首地址
    }
    else{ //不是回车和空格
        buffer[k++] = line[j]; //拷进 buffer
    }
}

exit(0);
}
```