

## 实验六

### *Hollow Man*

#### 实验名称：

管道通信

#### 实验目的：

1. 掌握管道的通信原理与过程
2. 可以使用管道实现进程间通信

#### 实验时间

3 学时

#### 预备知识：

1. 概念

管道有两种类型：无名管道和有名管道。无名管道没有名字，也从来不在文件系统中出现，只是和内存中的一个索引节点相关联的两个文件描述符，该索引节点指向指向内存中的一个物理块。写进程向管道写入数据时，字节被复制到共享数据页面中，读进程从管道读出数据时，字节从共享页面中读出。无名管道是半双工的，数据只能在一个方向传送，而且只能在相关的、有共同祖先的进程之间使用。

有（命）名管道（FIFO）可以为两个不相关的进程提供通信。它不是临时对象，是文件系统中的实体，可以用 `mknod` 和 `mkfifo` 创建。在写进程使用之前，必须让读进程先打开管道，任何读进程从中读取之前必须有写进程向其写入数据。FIFO 有一个路径与之关联，故无亲缘关系的进程可以访问同一个 FIFO。

2. 无名管道工作原理

无名管道由单个进程创建，但很少在单个进程内使用。其典型用途是在一个父进程和子进程之间通信。首先由一个进程创建一个管道后调用 `fork` 派生一个自身的拷贝，如图 1：

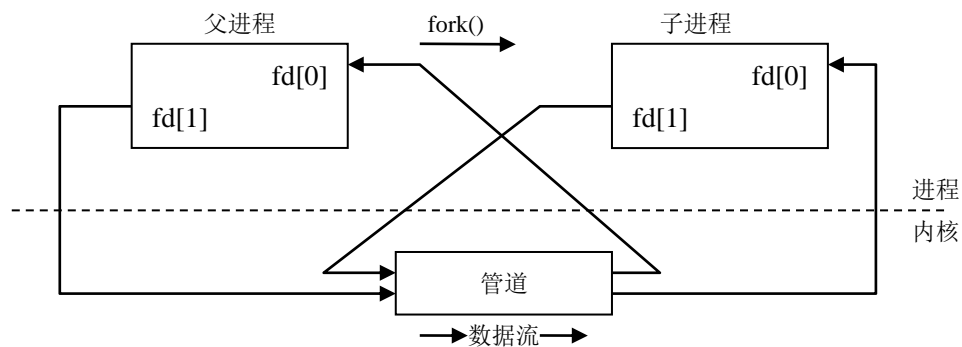


图 1：单个进程内的管道

然后，父进程关闭该管道的读出端，子进程关闭同一管道的写入端，这样就在父子进程间提供了一个单项数据流，如图 2。

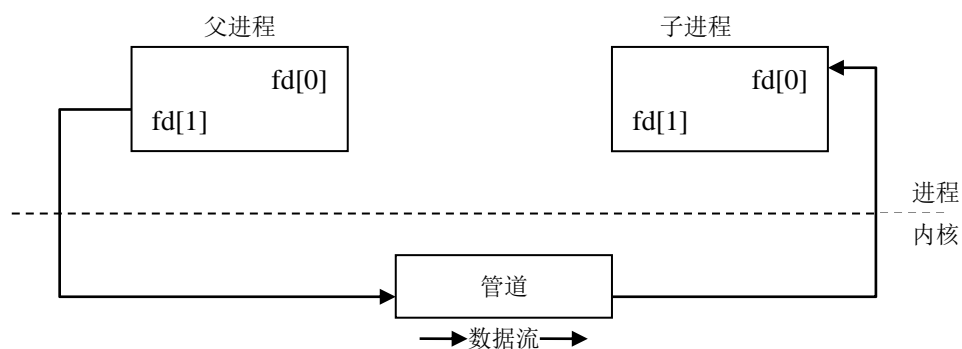


图 2：两个进程间的管道

当需要一个双向数据流时，必须建立两个管道，每个方向一个，实际步骤如下：

1. 创建管道 1（fd1[0]和 fd1[1]）和管道 2（fd2[0]和 fd2[1]）
2. fork()
3. 父进程关闭管道 1 的读出端 fd1[0]
4. 父进程关闭管道 2 的写入端 fd2[1]
5. 子进程关闭管道 1 的写入端 fd1[1]
6. 子进程关闭管道 2 的读出端 fd2[0]

管道布局如图 3：

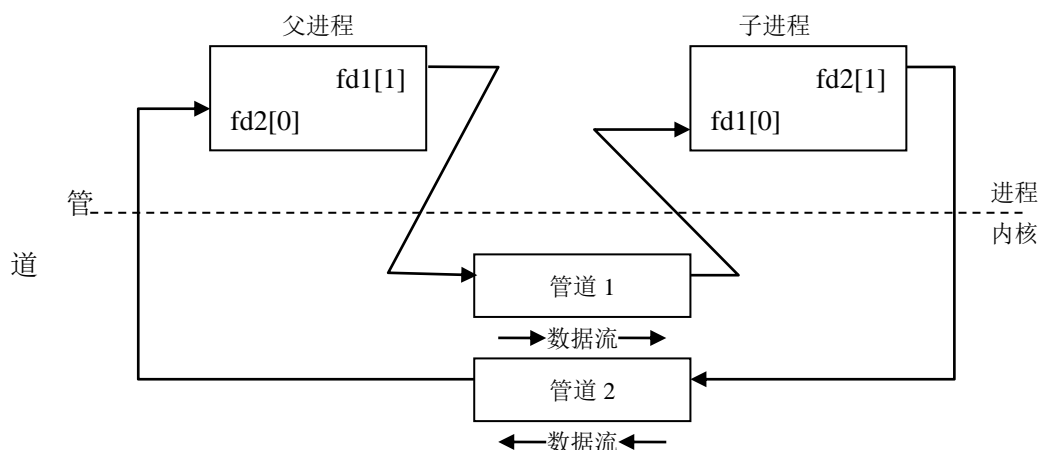


图 3：提供一个双向数据流的两个管道

读写分别使用 `read` 和 `write` 系统调用，其中读取字节数不应大于 `PIPE_BUF`(`<limits.h>`中定义，主要是为了保证原子操作)，因此通常定义缓冲区大小为 `PIPE_BUF`。管道使用完后要关闭(`close`)。

### 3. 基本命令

#### 2.1 `$command1 | command2`

将 `command1` 的标准输出作为 `command2` 的标准输入。

#### 3.2 `mknod` 创建块、字符或管道文件

#### 3.3 `mkfifo` 创建一个命名管道(FIFO)

### 4. 系统调用

#### 4.1 `pipe` (建立管道)

表头文件 `#include<unistd.h>`

定义函数 `int pipe(int filedes[2]);`

函数说明 `pipe()`会建立管道，并将文件描述词由参数 `filedes` 数组返回。`filedes[0]`为管道里的读取端，`filedes[1]`则为管道的写入端。

返回值 若成功则返回零，否则返回-1，错误原因存于 `errno` 中。

错误代码 `EMFILE` 进程已用完文件描述词最大量。

`ENFILE` 系统已无文件描述词可用。

`EFAULT` 参数 `filedes` 数组地址不合法。

#### 4.2 `mkfifo` (建立具名管道)

表头文件 `#include<sys/types.h>`

`#include<sys/stat.h>`

定义函数 `int mkfifo(const char * pathname,mode_t mode);`

函数说明 `mkfifo()`会依参数 `pathname` 建立特殊的 `FIFO` 文件，该文件必须不存在，而参数 `mode` 为该文件的权限 (`mode&~umask`)，因此 `umask` 值也会影响到 `FIFO` 文件的权限。`mkfifo()`建立的 `FIFO` 文件其他进程都可以用读写一般文件的方式存取。当使用 `open()` 来打开 `FIFO` 文件时，`O_NONBLOCK` 旗标会有影响

1、当使用 `O_NONBLOCK` 旗标时，打开 `FIFO` 文件来读取的操作会立刻返回，但是若还没有其他进程打开 `FIFO` 文件来读取，则写入的操作会返回 `ENXIO` 错误代码。

2、没有使用 `O_NONBLOCK` 旗标时，打开 `FIFO` 来读取的操作会等到其他进程打开 `FIFO` 文件来写入才正常返回。同样地，打开 `FIFO` 文件来写入的操作会等到其他进程打

开 FIFO 文件来读取后才正常返回。

返回值 若成功则返回 0，否则返回-1，错误原因存于 `errno` 中。

错误代码 `EACCESS` 参数 `pathname` 所指定的目录路径无可执行的权限

`EEXIST` 参数 `pathname` 所指定的文件已存在。

`ENAMETOOLONG` 参数 `pathname` 的路径名称太长。

`ENOENT` 参数 `pathname` 包含的目录不存在

`ENOSPC` 文件系统的剩余空间不足

`ENOTDIR` 参数 `pathname` 路径中的目录存在但却非真正的目录。

`EROFS` 参数 `pathname` 指定的文件存在于只读文件系统内。

#### 4.3 open（打开文件）

表头文件 `#include<sys/types.h>`

`#include<sys/stat.h>`

`#include<fcntl.h>`

定义函数 `int open( const char * pathname, int flags);`

`int open( const char * pathname,int flags, mode_t mode);`

函数说明 参数 `pathname` 指向欲打开的文件路径字符串。下列是参数 `flags` 所能使用的旗标:

`O_RDONLY` 以只读方式打开文件

`O_WRONLY` 以只写方式打开文件

`O_RDWR` 以可读写方式打开文件。上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用 `OR()`运算符组合。

`O_CREAT` 若欲打开的文件不存在则自动建立该文件。

`O_EXCL` 如果 `O_CREAT` 也被设置，此指令会去检查文件是否存在。文件若不存在则建立该文件，否则将导致打开文件错误。此外，若 `O_CREAT` 与 `O_EXCL` 同时设置，并且欲打开的文件为符号连接，则会打开文件失败。

返回值 若所有欲核查的权限都通过了检查则返回 0 值，表示成功，只要有一个权限被禁止则返回-1。

错误代码 `EEXIST` 参数 `pathname` 所指的文件已存在，却使用了 `O_CREAT` 和 `O_EXCL` 旗标。

`EACCESS` 参数 `pathname` 所指的文件不符合所要求测试的权限。

EROFS 欲测试写入权限的文件存在于只读文件系统内。

EFAULT 参数 `pathname` 指针超出可存取内存空间。

EINVAL 参数 `mode` 不正确。

ENAMETOOLONG 参数 `pathname` 太长。

ENOTDIR 参数 `pathname` 不是目录。

ENOMEM 核心内存不足。

ELOOP 参数 `pathname` 有过多符号连接问题。

EIO I/O 存取错误。

#### 4.4 close（关闭文件）

表头文件 `#include<unistd.h>`

定义函数 `int close(int fd);`

函数说明 当使用完文件后若已不再需要则可使用 `close()` 关闭该文件，二 `close()` 会让数据写回磁盘，并释放该文件所占用的资源。参数 `fd` 为先前由 `open()` 或 `creat()` 所返回的文件描述词。

返回值 若文件顺利关闭则返回 0，发生错误时返回 -1。

错误代码 EBADF 参数 `fd` 非有效的文件描述词或该文件已关闭。

#### 4.5 read（由已打开的文件读取数据）

表头文件 `#include<unistd.h>`

定义函数 `ssize_t read(int fd, void * buf, size_t count);`

函数说明 `read()` 会把参数 `fd` 所指的文件传送 `count` 个字节到 `buf` 指针所指的内存中。

若参数 `count` 为 0，则 `read()` 不会有作用并返回 0。返回值为实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，此外文件读写位置会随读取到的字节移动。

错误代码 EINTR 此调用被信号所中断。

EAGAIN 当使用不可阻断 I/O 时（`O_NONBLOCK`），若无数据可读取则返回此值。

EBADF 参数 `fd` 非有效的文件描述词，或该文件已关闭。

#### 4.6 write（将数据写入已打开的文件内）

表头文件 `#include<unistd.h>`

定义函数 `ssize_t write (int fd, const void * buf, size_t count);`

函数说明 write()会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。

当然，文件读写位置也会随之移动。

返回值 如果顺利 write()会返回实际写入的字节数。当有错误发生时则返回-1，错误代码存入 errno 中。

错误代码 EINTR 此调用被信号所中断。

EAGAIN 当使用不可阻断 I/O 时 (O\_NONBLOCK)，若无数据可读取则返回此值。

EADF 参数 fd 非有效的文件描述词，或该文件已关闭。

#### 4.7 unlink (删除文件)

表头文件 #include<unistd.h>

定义函数 int unlink(const char \* pathname);

函数说明 unlink()会删除参数 pathname 指定的文件。如果该文件名为最后连接点，但还有其他进程打开了此文件，则在所有关于此文件的文件描述词皆关闭后才会删除。如果参数 pathname 为一符号连接，则此连接会被删除。

返回值 成功则返回 0，失败返回-1，错误原因存于 errno

错误代码 EROFS 文件存在于只读文件系统内

EFAULT 参数 pathname 指针超出可存取内存空间

ENAMETOOLONG 参数 pathname 太长

ENOMEM 核心内存不足

ELOOP 参数 pathname 有过多符号连接问题

EIO I/O 存取错误

### 5. 标准库函数

#### 5.1 popen (建立管道 I/O)

表头文件 #include<stdio.h>

定义函数 FILE \* popen( const char \* command,const char \* type);

函数说明 popen()会调用 fork()产生子进程，然后从子进程中调用/bin/sh -c 来执行参数 command 的指令。参数 type 可使用“r”代表读取，“w”代表写入。依照此 type 值，popen()会建立管道连到子进程的标准输出设备或标准输入设备，然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入到子进程的标准输入设备中。此外，所有使用文件指针(FILE\*)操作的函数也都可以使用，除了 fclose()以外。

返回值 若成功则返回文件指针，否则返回 NULL，错误原因存于 `errno` 中。

错误代码 `EINVAL` 参数 `type` 不合法。

## 5.2 `pclose`（关闭管道 I/O）

表头文件 `#include <stdio.h>`

定义函数 `int pclose(FILE * stream);`

函数说明 `pclose()` 用来关闭由 `popen` 所建立的管道及文件指针。参数 `stream` 为先前由 `popen()` 所返回的文件指针。

返回值 返回子进程的结束状态。如果有错误则返回 -1，错误原因存于 `errno` 中。

错误代码 `ECHILD` `pclose()` 无法取得子进程的结束状态。

## 实验要求：

### 1. 阅读以下程序：

```
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

main()

{

    int filedес[2];

    char buffer[80];

    if(pipe(filedes)<0)

        printf("pipe error");

    if(fork(>0){

        char s[ ] = "hello!\n";

        close(filedes[0]);

        write(filedes[1],s,sizeof(s));

        close(filedes[1]);

    }else{

        close(filedes[1]);

        read(filedes[0],buffer,80);

        printf("%s",buffer);
```

```
        close(filedes[0]);  
    }  
}
```

编译并运行程序，分析程序执行过程和结果，注释程序主要语句。

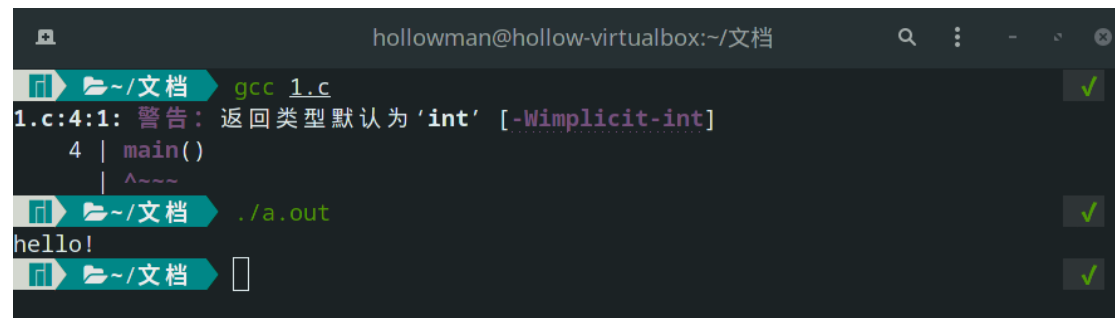
程序代码注释如下：

```
#include <unistd.h>  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
main()  
{  
  
    int filedes[2]; //filedes[0]为管道里的读取端，filedes[1]为管道的写入端  
  
    char buffer[80];  
  
    if (pipe(filedes) < 0) //成功返回 0，失败返回-1  
        printf("pipe error");  
  
    if (fork() > 0) //父进程  
    {  
        char s[] = "hello!\n";  
  
        close(filedes[0]); //关闭父进程读取端  
  
        write(filedes[1], s, sizeof(s)); //写入数据  
  
        close(filedes[1]); //关闭父进程写入端  
    }  
  
    else //子进程  
    {  
        close(filedes[1]); //关闭子进程写入端  
  
        read(filedes[0], buffer, 80); //读取数据  
  
        printf("%s", buffer); //输出读取的内容  
  
        close(filedes[0]); //关闭子进程读取端  
    }  
}
```

这段代码使用无名管道，通过从父进程中写入字符串数据”hello!\n”，并在子进程中在



屏幕上打印接收到的数据，用于测试管道通信，其执行效果如下：



```
hollowman@hollow-virtualbox:~/文档
~/文档 gcc 1.c ✓
1.c:4:1: 警告：返回类型默认为 'int' [-Wimplicit-int]
    4 | main()
      | ^~~~
~/文档 ./a.out ✓
hello!
~/文档 [ ] ✓
```

2. 阅读以下程序：

```
#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

main()
{
    char buffer[80];

    int fd;

    char *FIFO;

    unlink(FIFO);

    mkfifo(FIFO,0666);

    if(fork(>0)){

        char s[ ] = "hello!\n";

        fd = open (FIFO,O_WRONLY);

        write(fd,s,sizeof(s));

        close(fd);

    }else{

        fd= open(FIFO,O_RDONLY);

        read(fd,buffer,80);

        printf("%s",buffer);

        close(fd);

    }

}
```

编译并运行程序，分析程序执行过程和结果，注释程序主要语句。

程序代码注释如下：

```
#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

main()
{
    char buffer[80];

    int fd;

    unlink("FIFO"); //如果已经存在 FIFO 文件则先删除

    mkfifo("FIFO", 0666); //创建权限为 0666 的名称为 FIFO 的命名管道

    if (fork() > 0) //父进程
    {
        char s[] = "hello !\n";

        fd = open("FIFO", O_WRONLY);    //以只写方式打开 FIFO 命名管道

        write(fd, s, sizeof(s));    //写入数据 s

        close(fd); //关闭命名管道
    }
    else    //子进程
    {
        fd = open("FIFO", O_RDONLY);    //以只读方式打开 FIFO 命名管道

        read(fd, buffer, 80);    //读出数据存入 buffer

        printf("%s", buffer);    //打印 buffer

        close(fd); //关闭命名管道
    }
}
```

这段代码使用有名管道，创建了一个名为 FIFO 的命名管道，通过从父进程中写入字符串数据 "hello !\n"，并在子进程中在屏幕上打印接收到的数据，用于测试管道通信，其执行效果如下：

```
hollowman@hollow-virtualbox:~/文档
~/文档 nano 2.c
~/文档 gcc 2.c
2.c:5:1: 警告：返回类型默认为'int' [-Wimplicit-int]
5 | main()
  | ^~~~
2.c: 在函数 'main' 中:
2.c:22:9: 警告：隐式声明函数 'printf' [-Wimplicit-function-declaration]
22 |     printf("%s", buffer); //打印buffer
    |     ^~~~~~
2.c:22:9: 警告：隐式声明与内建函数 'printf' 不兼容
2.c:5:1: 附注：include '<stdio.h>' or provide a declaration of 'printf'
4 | #include <unistd.h>
+++ |+#include <stdio.h>
5 | main()
~/文档 ./a.out
hello !
~/文档
```

3. 阅读以下程序：

```
#include<stdio.h>

main()

{

    FILE * fp;

    char buffer[80];

    fp=popen("cat /etc/passwd","r");

    fgets(buffer,sizeof(buffer),fp);

    printf("%s",buffer);

    pclose(fp);

}
```

编译并运行程序，分析程序执行过程和结果，注释程序主要语句。

Shell 命令 cat /etc/passwd 的输出结果如下：

```
hollowman@hollow-virtualbox:~/文档
cat /etc/passwd
root:x:0:0::/root:/bin/bash
nobody:x:65534:65534:Nobody:/:usr/bin/nologin
dbus:x:81:81:System Message Bus:/:usr/bin/nologin
bin:x:1:1:/:usr/bin/nologin
daemon:x:2:2:/:usr/bin/nologin
mail:x:8:12:/:var/spool/mail:usr/bin/nologin
ftp:x:14:11:/:srv/ftp:usr/bin/nologin
http:x:33:33:/:srv/http:usr/bin/nologin
systemd-journal-remote:x:982:982:systemd Journal Remote:/:usr/bin/nologin
systemd-network:x:981:981:systemd Network Management:/:usr/bin/nologin
systemd-resolve:x:980:980:systemd Resolver:/:usr/bin/nologin
systemd-timesync:x:979:979:systemd Time Synchronization:/:usr/bin/nologin
systemd-coredump:x:978:978:systemd Core Dumper:/:usr/bin/nologin
quidd:x:68:68:/:usr/bin/nologin
dhcpcd:x:977:977:dhcpcd privilege separation:/:usr/bin/nologin
dnsmasq:x:976:976:dnsmasq daemon:/:usr/bin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:usr/bin/nologin
avahi:x:975:975:Avahi mDNS/DNS-SD daemon:/:usr/bin/nologin
colord:x:974:974:Color management daemon:/var/lib/colord:usr/bin/nologin
cups:x:209:209:cups helper user:/:usr/bin/nologin
flatpak:x:973:973:Flatpak system helper:/:usr/bin/nologin
gdm:x:120:120:Gnome Display Manager:/var/lib/gdm:usr/bin/nologin
geoclue:x:972:972:Geoinformation service:/var/lib/geoclue:usr/bin/nologin
git:x:971:971:git daemon user:/:usr/bin/git-shell
nm-openconnect:x:970:970:NetworkManager OpenConnect:/:usr/bin/nologin
nm-openvpn:x:969:969:NetworkManager OpenVPN:/:usr/bin/nologin
ntp:x:87:87:Network Time Protocol:/var/lib/ntp/bin/false
openvpn:x:968:968:OpenVPN:/:usr/bin/nologin
polkitd:x:102:102:PolicyKit daemon:/:usr/bin/nologin
rtkit:x:133:133:RealtimeKit:/proc:usr/bin/nologin
saned:x:967:967:SANE daemon user:/:usr/bin/nologin
tss:x:966:966:tss user for tpm2:/:usr/bin/nologin
usbmux:x:140:140:usbmux user:/:usr/bin/nologin
```

程序代码注释如下：

```
#include<stdio.h>

main()
{
    FILE * fp;

    char buffer[80];

    fp=popen("cat /etc/passwd","r");    //子进程执行 cat /etc/passwd，建立管道 I/O 连接到子
进程标准输出设备

    fgets(buffer,sizeof(buffer),fp);    //读取一行内容

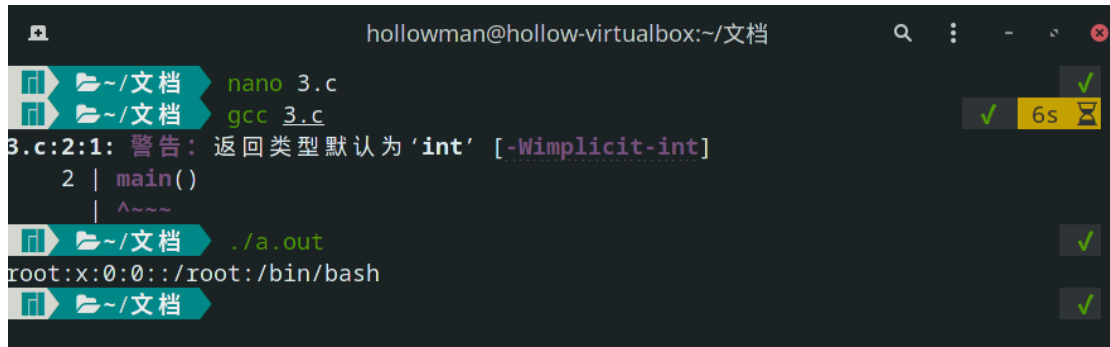
    printf("%s",buffer);    //打印 buffer 的内容

    pclose(fp); //关闭管道 I/O
```

```
}
```

这段程序使用 `popen` 建立管道 I/O 连接到执行 `cat /etc/passwd` 命令的子进程的标准输出设备，并使用 `fgets` 函数从中读取一行内容打印在屏幕上。

程序的输出结果如下：



```
hollowman@hollow-virtualbox:~/文档
nano 3.c
gcc 3.c
3.c:2:1: 警告：返回类型默认为'int' [-Wimplicit-int]
  2 | main()
    | ^~~~
./a.out
root:x:0:0::/root:/bin/bash
```

4. 编写一个程序，读取一个数据文件，对每一个数据进行某种运算，再在屏幕输出计算结果。要求以上工作用两个进程实现，父进程负责读文件和显示，子进程进行计算，进程间通信使用无名管道。（使用系统调用）

程序代码如下：

```
#include <stdio.h>

#include <unistd.h>

#include <math.h>

#include <sys/wait.h>

#include <stdlib.h>

int main()
{
    int fda[2], fdb[2];

    int pid, i;

    if(pipe(fda) < 0 || pipe(fdb) < 0)    //建立无名管道
        printf("pipe error");

    while((pid = fork()) == -1);    //创建子进程

    if(pid > 0) //父进程
    {
        FILE *fpread = fopen("data.txt", "r");    //打开文件 data.txt
```

```

float data, result;

if(fpread == NULL) //打开文件失败
{
    printf("open file error!\n");
    exit(1);
}

close(fda[0]); //关闭父进程管道 a 读取端
close(fdb[1]); //关闭父进程管道 b 写入端

for(i = 0; i < 10; i++)
{
    fscanf(fpread, "%f", &data); //读入一个数据存入 data
    write(fda[1], &data, sizeof(data)); //data 通过管道 a 发送给子进程
    read(fdb[0], &result, sizeof(result)); //通过管道 b 读计算结果到 result
    printf("sin(%f)=%f\n", data, result);
}

close(fdb[0]); //关闭父进程管道 b 读取端
close(fda[1]); //关闭父进程管道 a 写入端
fclose(fpread); //关闭文件 data.txt
}

else //子进程
{
    float data, result;

    close(fda[1]); //关闭子进程管道 a 写入端
    close(fdb[0]); //关闭子进程管道 b 读取端

    for(i = 0; i < 10; i++)
    {
        read(fda[0], &data, sizeof(data));

        result = sinf(data);

        write(fdb[1], &result, sizeof(result));
    }
}

```

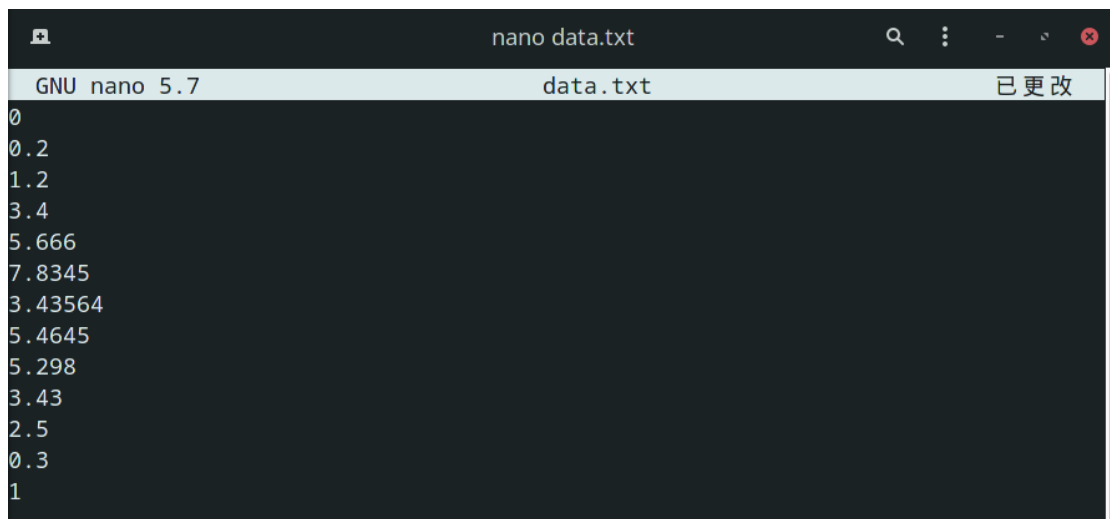
```
        close(fdb[1]); //关闭子进程管道 b 写入端

        close(fda[0]); //关闭子进程管道 a 读取端
    }

    return 0;
}
```

这段代码建立了 2 个无名管道，由于无名管道是半双工的，数据只能在一个方向传送，因此 2 个无名管道分别对应父进程写数据子进程读数据（fda）和子进程写数据父进程读数据（fdb）。

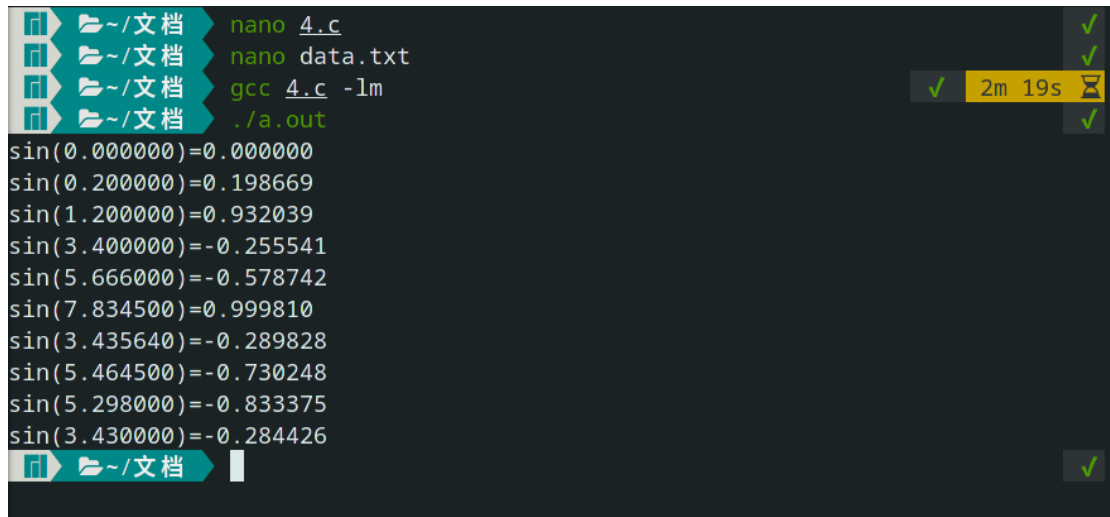
data.txt 文件中存储了 13 条数据，如下图所示：



```
0
0.2
1.2
3.4
5.666
7.8345
3.43564
5.4645
5.298
3.43
2.5
0.3
1
```

父进程和子进程中均循环了 13 次，每次循环父进程从文件中读取一行浮点型数据，使用 fda 发送给子进程，子进程接收到这个浮点数对其求正弦值并将计算结果发回给父进程并显示在屏幕上。

程序的输出结果如下：



```
~/文档 nano 4.c ✓
~/文档 nano data.txt ✓
~/文档 gcc 4.c -lm ✓ 2m 19s
~/文档 ./a.out ✓
sin(0.000000)=0.000000
sin(0.200000)=0.198669
sin(1.200000)=0.932039
sin(3.400000)=-0.255541
sin(5.666000)=-0.578742
sin(7.834500)=0.999810
sin(3.435640)=-0.289828
sin(5.464500)=-0.730248
sin(5.298000)=-0.833375
sin(3.430000)=-0.284426
~/文档 ✓
```

5. 编写两个程序，一个创建一个 FIFO，并读管道，并显示在屏幕上，另一个每过一段时间向该管道写数据（进程 PID）。运行多个写程序和一个读程序，观察运行结果。（使用系统调用）

读程序的代码如下：

```
#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

int main()
{
    int fd, pid;

    unlink("FIFO"); //如果已经存在 FIFO 文件则先删除

    mkfifo("FIFO", 0666); //创建权限为 0666 的名称为 FIFO 的命名管道

    fd = open("FIFO", O_RDONLY); //以只读方式打开 FIFO 命名管道

    while(read(fd, &pid, sizeof(pid)))
    {
        printf("pid: %d\n", pid);
    }

    close(fd);

    return 0;
}
```



写程序的代码如下：

```
#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

int main()
{
    int pid, fd;

    pid = getpid(); //获得当前进程的 pid

    fd = open("FIFO", O_WRONLY);    //以只写方式打开 FIFO 命名管道

    while(write(fd, &pid, sizeof(pid))) //持续写数据
    {
        printf("pid: %d\n", pid);

        sleep(3);    //等待 3 秒
    }

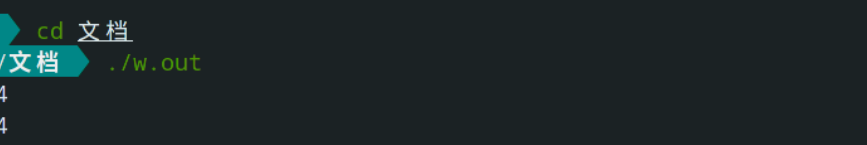
    close(fd);

    return 0;
}
```

读程序和写程序中都创建了一个名为 FIFO 的命名管道，读程序连续不断的从 FIFO 中读取数据并打印，而写程序则写入自己的 pid，最终执行写程序一段时间后通过 Ctrl+C 停止两个写程序的执行，读程序也随之停止执行。读程序输出了两个写程序的 PID，输出的数量与两个写程序写入的次数之和一致。

读程序的执行效果如下：





The screenshot shows a terminal window with the following content:

```
hollowman@hollow-virtualbox:~/文档
cd 文档
./w.out
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
pid: 6424
PIPE X 30s
```

The terminal has a dark background. The prompt is `hollowman@hollow-virtualbox:~/文档`. The first command is `cd 文档`, which is highlighted with a green bar. The second command is `./w.out`, also highlighted with a green bar. The output consists of ten lines of `pid: 6424`. At the bottom, there is a red bar with the text `PIPE X` and a yellow bar with the text `30s` and a clock icon.