# 进程间通信之管道

# **管道**

## 一、**概述**

管道是最初的Unix IPC形式，可追溯到1973年的Unix第3版。

管道根本局限在于没有名字，从而只能由有亲缘关系的进程使用。这一点随FIFO的加入后得以改正。FIFO又是成为有名管道（named pipe）

管道和FIFO都是使用通常的read和write函数访问

## 二、**分类**

匿名管道:管道没有标识符,不能被其他进程找到，因此只能用于具有亲缘关系的进程间通信。创建管道一定要在创建子进程之前。

命名管道:管道具有标识符,能够被其他进程找到，因此可以用于同一主机上的任意进程间通信。

## 三、**管道通信原理**

一个进程向管道中写入的内容被管道另一端的进程读出

写入的内容每次都是添加在管道缓冲区的末尾，读出每次都是从缓冲区的头部读出数据

单工通信：只支持信号在一个方向上传输，任何时候不能改变信号的传输方向。

双工通信：允许信号在两个方向上传输。

双工通信又分为半双工和全双工

半双工通信：允许信号在两个方向上传输，但某一时刻只允许信号在一个方向上单向传输。

全双工通信：允许信号在两个方向上传输，信号可以在两个方向上同时传输。

​​

## 四、**管道创建过程**

### （一）无名管道

无名管道由pipe函数创建，提供某个时刻一个单路（单向）数据流

头文件：#include <unistd.h>

函数原型：int pipe(int pipefd[2]);

参数：

pipefd：数组首地址

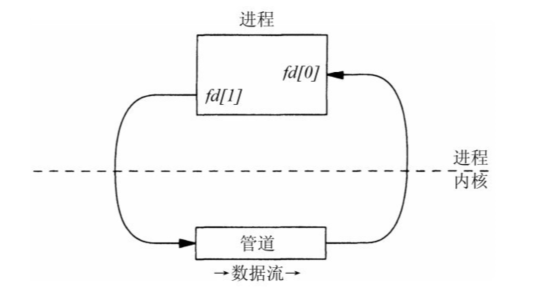
       pipefd[0]表示读端

       pipefd[1]表示写端

返回值：成功返回0

        失败返回-1

图解：单个进程中管道实例

​​

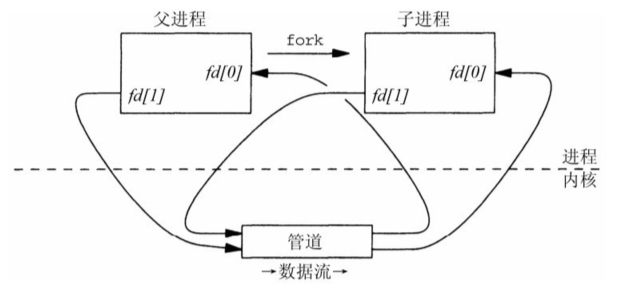
尽管管道是由单个进程创建的，但是很少在单个进程内使用

管道的经典用途是以下述方式为两个不同进程（一个是父进程，一个是子进程）提供进程间的通信手段（半双工）

步骤：

1．首先，由一个进程创建一个管道后调用fork派生一个自身的副本，如下图

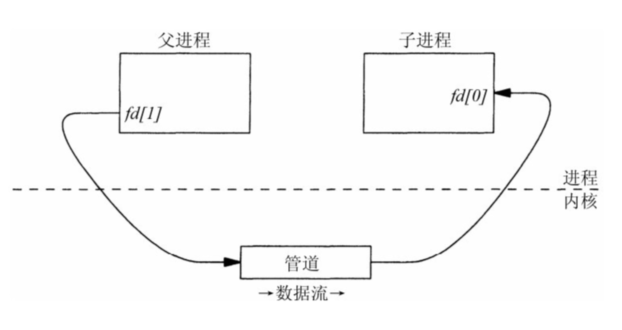
图解：单个进程内的管道，刚刚fork后

​​

2．接下，父进程关闭这个管道的读出端，子进程关闭同一管道的写入端。

       这样在父子进程间提供了一个单向数据流，如下图

图解：两个亲缘进程间的管道

​​

当需要一个双向数据流时，必须创建两个管道，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；

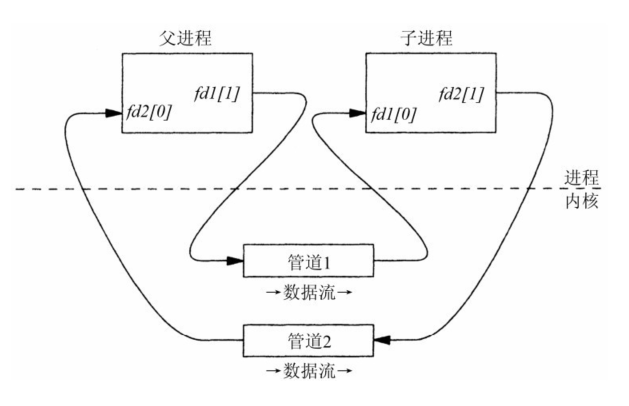
3．父进程关闭管道1的读出端（fd1[0]）；

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]）。

图解：父子进程全双工通信

​​

### （二）有名管道（FIFO）

无名管道没有名字，因此它们最大劣势是只能用于有一个共同祖先进程的各个进程之间，无法在无亲缘关系的两个进程间创建一个管道

FIFO指代先进先出（first in，first out），它是一个单向（半双工）数据流。不同于管道的是，每个FIFO有一个路径名与之关联，从而允许无亲缘关系的进程访问同一个FIFO。FIFO也称为有名管道（named pipe）。

FIFO由mkfifo函数创建

头文件：#include <sys/types.h>

#include# <sys/stat.h>

函数原型：int mkfifo(const char \*pathname, mode\_t mode);

参数：

pathname：普通的路径名，是该FIFO的名字

      mode：指定文件权限位，类似于open的第二个参数

返回值：成功返回0，失败返回-1

注意点：mkfifo函数，要么创建一个新的FIFO，要么返回一个EEXIST错误（如果所指定名字的FIFO已经存在）。如果不希望创建一个新的FIFO，则改为调用open而不是mkfifo。

要打开一个已经存在的FIFO或创建一个新的FIFO，应先调用mkfifo，再检查它是否返回EEXIST错误，若返回该错误则改为调用open。

## 五、**管道读写规则**

​​

### （一）读端存在，写管道

管道空闲，则写入数据；

管道满（管道容量为 65535），则写管道阻塞

### （二）读端不存在，写管道

写管道没有意义，内核会发送SIGPIPE信号，杀死写管道进程

需要将读写进程运行，后杀死读管道，写管道结束

### （三）读管道存在，写管道不存在

管道中有数据，读取数据，管道中没有数据，读管道不阻塞，立即返回0

### （四）读管道大小100，写端存在

管道中有数据 >= 要求读取数据 ---> 读取要求大小的数据

管道中有数据 <  要求读取数据 ---> 读取管道中实际大小数据

管道中没有数据，则读管道阻塞

## 六、**总结**

无名管道 与 有名管道异同点：

相同：

1．都是基于文件描述符的进程间通信方式

2．都可以利用read()，write()函数等文件IO函数进行读写。与普通文件读写区别是管道中数据读完后就没有,数据只能读到一次

3．都不可使用lseek()进行定位操作

4．数据都是先入先出FIFO

异同：

1．无名管道只能用于具有亲缘关系的进程间通信；有名管道可用于并不相关的两个进程之间相互通信

2．无名管道通过pipe函数创建并获得操作的一对文件描述符，具有亲缘关系的进程才能使用；而有名管道是通过mkfifo函数创建，任何进程都可以根据有名管道文件名及其所在路径利用open()函数打开，close()函数关闭，read()/write()读写

3．无名管道只在内核中有一段缓冲区，只能通过pipe函数创建并获得读写的文件描述符；而有名管道不光在内核中有一段缓冲区，而且在文件系统中有一个文件名，这个文件名对应了内核中的一段缓冲区。与普通文本文件区别在于有名管道仅仅在文件系统中有一个名字，数据是存放在内存中

## 七、**示例代码**

无名管道代码示例：

#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
#include <string.h>  
  
#define BUFFER\_SIZE 25  
  
//写文件  
void write\_file(int fd)  
{  
 char buf[100] = {0};  
 int n;  
 while(1)  
 {  
 memset(buf,0,sizeof(buf));   
 fgets(buf,sizeof(buf),stdin);  
  
 write(fd,buf,strlen(buf));  
  
 if(strncmp(buf,"quit",4) == 0)  
 break;  
 }   
}  
//读文件  
void read\_file(int fd)  
{  
 char buf[100] = {0};  
 int n;  
 while(1)  
 {  
 memset(buf,0,sizeof(buf));   
 n = read(fd,buf,sizeof(buf));  
  
 printf("read %d bytes : %s\n",n,buf);  
  
 if(strncmp(buf,"quit",4) == 0)  
 break;  
 }  
}  
int main()   
{  
 int fd[2];  
 pid\_t pid;  
 //创建管道  
 if (pipe(fd) == -1)   
 {  
 perror("pipe");  
 exit(EXIT\_FAILURE);  
 }  
 //创建子进程  
 pid = fork();  
 if (pid < 0)   
 {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 else if (pid == 0)   
 {   
 //子进程  
 //关闭写端口  
 close(fd[1]);   
 //子进程读文件  
 read\_file(fd[0]);  
  
 close(fd[0]);  
 exit(EXIT\_SUCCESS);  
 }   
 else   
 {   
 // 父进程   
 // 关闭读端口  
 close(fd[0]);  
 //父进程写文件  
 write\_file(fd[1]);  
  
 close(fd[1]);  
 exit(EXIT\_SUCCESS);  
 }  
}

有名管道：自己读写

运行方法：./a.out fifo

#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <sys/stat.h>  
#include <sys/types.h>  
#include <string.h>  
#include <errno.h>  
  
#define BUFFER\_SIZE 25  
  
int main(int argc, const char \*argv[])   
{  
 int fd;  
 char write\_buffer[BUFFER\_SIZE] = "Hello, world!\n";  
 char read\_buffer[BUFFER\_SIZE] = {0};  
 int n = 0;  
  
 if(argc != 2)  
 {  
 fprintf(stderr, "Usage:%s fifoname\n", argv[0]);  
 exit(EXIT\_FAILURE);  
 }  
  
 //创建有名管道，规避文件已存在的错误  
 if(mkfifo(argv[1], 0666) < 0 && errno != EEXIST)   
 {  
 perror("Fail to mkfifo");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("create %s FIFO is successful!\n", argv[1]);  
  
 //读写方式打开文件  
 fd = open(argv[1], O\_RDWR);  
 if(fd < 0)  
 {  
 perror("Fail to open");  
 exit(EXIT\_FAILURE);  
 }  
  
 //自身进程进行读写  
 write(fd, write\_buffer, strlen(write\_buffer));  
 lseek(fd, 0, SEEK\_SET);  
 lseek(fd, 0, SEEK\_SET);  
 n = read(fd, read\_buffer, sizeof(read\_buffer));  
  
 printf("read %d bytes:%s\n", n, read\_buffer);  
  
 close(fd);  
 return 0;  
}

## 八、**练习**

### 管道：

练习:用无名管道实现，ps -ef | grep a.out

[1] 创建无名管道，获得两个文件描述符,然后创建子进程

[2] 父亲进程:关闭读端，标准输出重定向到管道的输入端（dup2），将进程进行替换，然后写入管道

[3] 子进程 :关闭写端，标准输入重定向到管道的输出端（dup2），循环从管道中读取数据，然后打印

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

#include <string.h>

int main(int argc, const char \*argv[])

{

int pipefd[2] = {0};

int ret = pipe(pipefd);

if (ret != 0) {

perror("pipe failed");

return 1;

}

pid\_t pid = fork();

sleep(5);

if (pid < 0) {

perror("fork failed");

return 1;

} else if (pid == 0) {

close(pipefd[1]);

dup2(pipefd[0], 0);

execlp("grep", "grep", "a.out", NULL);

close(pipefd[0]);

exit(EXIT\_SUCCESS);

} else {

close(pipefd[0]);

dup2(pipefd[1], 1);

execlp("ps", "ps", "-ef", NULL);

close(pipefd[1]);

exit(EXIT\_SUCCESS);

}

return 0;

}

### 有名管道：

练习：两个进程利用管道来进行读写

[1] 写一个write\_fifo.c文件利用mkfifo创建管道，以只写的方式打开管道文件，然后创建while循环，向管道中输入数据，当输入的数据是”quit”的时候，当前进程结束。

[2] 在写一个read\_fifo.c文件利用mkfifo创建管道，以只读的方式打开read\_filo.c文件中打开的管道文件。然后把数据输出来。

[3] 当读取到”quit”数据的时候，进程结束)。两个进程结束

有名管道：两个无缘进程通信

写进程

运行方式：gcc write.c -o W

./W fifo

#include <stdio.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <string.h>  
#include <stdlib.h>  
#include <errno.h>  
#include <fcntl.h>  
#include <unistd.h>  
  
#define MAX 100  
  
void write\_file(int fd)  
{  
 char buf[MAX] = {0};  
  
 while(1)  
 {  
 memset(buf,0,sizeof(buf));  
 fgets(buf,sizeof(buf),stdin);   
  
 write(fd,buf,strlen(buf));  
  
 if(strncmp(buf,"quit",4) == 0)  
 break;  
 }  
  
 return ;  
}  
  
//./a.out fifoname  
int main(int argc, const char \*argv[])  
{  
 int fd;   
 char buf[MAX] = {0};  
 int n;  
  
 if(argc != 2)  
 {  
 fprintf(stderr,"Usage : %s fifoname\n",argv[0]);   
 exit(EXIT\_FAILURE);  
 }  
  
 //创建有名管道文件,规避掉文件已存在的错误  
 if((mkfifo(argv[1],0666) < 0) && errno != EEXIST)  
 {  
 perror("Fail to mkfifo");   
 return -1;  
 }  
  
 printf("create %s is successful!\n",argv[1]);  
  
 //只读方式打开文件  
 fd = open(argv[1], O\_WRONLY);  
 if(fd < 0)  
 {  
 perror("Fail to open");   
 return -1;  
 }  
  
 //进行写  
 write\_file(fd);  
  
 close(fd);  
  
 return 0;  
}

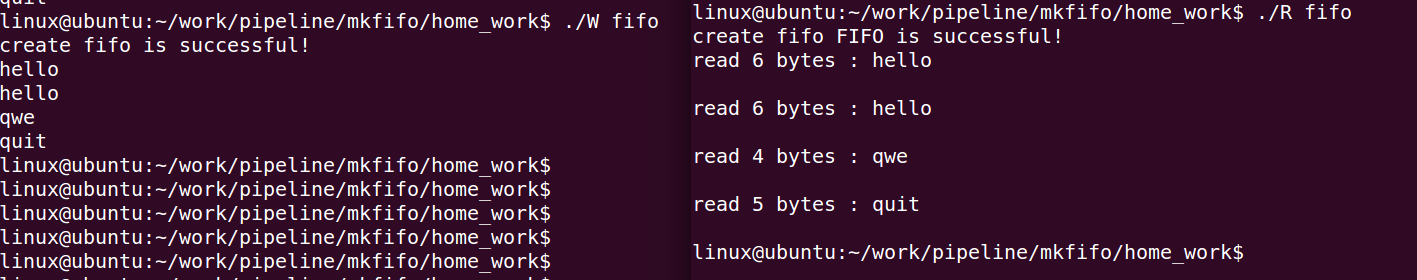
读进程

运行方式：gcc read.c -o R

./R fifo

#include <stdio.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <string.h>  
#include <stdlib.h>  
#include <errno.h>  
#include <fcntl.h>  
#include <unistd.h>  
  
#define MAX 100  
  
void read\_file(int fd)  
{  
 char buf[MAX] = {0};  
 int n = 0;  
 while(1)  
 {  
 sleep(1);  
 memset(buf,0,sizeof(buf));  
 n = read(fd,buf,sizeof(buf));  
  
 printf("read %d bytes : %s\n",n,buf);  
  
 if(strncmp(buf,"quit",4) == 0)  
 break;  
 }  
 return ;  
}  
  
int main(int argc, const char \*argv[])  
{  
 int fd;  
 int n;  
 char buf[MAX] = {0};  
  
 if(argc != 2)  
 {  
 fprintf(stderr, "Usage:%s fifoname\n", argv[0]);  
 exit(EXIT\_FAILURE);  
 }  
  
 //创建有名管道，规避文件已存在的错误  
 if(mkfifo(argv[1], 0666) < 0 && errno != EEXIST)  
 {  
 perror("Fail to mkfifo");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("create %s FIFO is successful!\n", argv[1]);  
  
 //只读方式打开文件  
 fd = open(argv[1], O\_RDONLY);  
 if(fd < 0)  
 {  
 perror("Fail to open");  
 exit(EXIT\_FAILURE);  
 }  
  
 //进行读  
 read\_file(fd);  
  
 close(fd);  
  
 return 0;  
}

运行结果：

​​

‍