# Todo

Caj73

# 环境搭建

因为不同uninx的环境，可能不同，因此需要编译一下头文件再

## 下载和安装

1. 下载源码

    W. Richard Stevens的主页：http://www.kohala.com/start/

    wget http://www.kohala.com/start/unpv22e/unpv22e.tar.gz -P /usr/local/src

2. 解压

    tar xvf /usr/local/src/unpv22e.tar.gz -C /root/bin

    备注：一定要在/root/bin文件夹下，未尝试不在这个文件夹下，在第4步会报错。无法生成/libunpipc.a文件

## 编译库文件

     cd /root/bin/unpv22e/  //备注：可能需要在root。不然权限不够

    ./configure

编辑生成config.h文件：注释以下几行、添加MSG\_R和MSG\_W定义，添加\_GNU\_SOURCE定义

diff --git a/config.h b/config.h

/\* #undef int32\_t \*/ /\* <sys/types.h> \*/

-#define uint8\_t unsigned char /\* <sys/types.h> \*/

-#define uint16\_t unsigned short /\* <sys/types.h> \*/

-#define uint32\_t unsigned int /\* <sys/types.h> \*/

+/\*#define uint8\_t unsigned char\*/ /\* <sys/types.h> \*/

+/\*#define uint16\_t unsigned short \*/ /\* <sys/types.h> \*/

+/\*#define uint32\_t unsigned int \*/ /\* <sys/types.h> \*/

+// add by keyguan@190607

+typedef unsigned long ulong\_t;

+#define MSG\_R 0400

+#define MSG\_W 0200

+

+#define \_GNU\_SOURCE

/\* #undef size\_t \*/ /\* <sys/types.h> \*/

/\* #undef ssize\_t \*/ /\* <sys/types.h> \*/

  编译wrapunix.c，使用mkstemp函数替换mktemp函数

diff --git a/lib/wrapunix.c b/lib/wrapunix.c

@@ -181,7 +181,7 @@ Mkfifo(const char \*pathname, mode\_t mode)

void

Mktemp(char \*template)

{

- if (mktemp(template) == NULL || template[0] == 0)

+ if (mkstemp(template) == NULL || template[0] == 0)

err\_quit("mktemp error");

}

 编译生成libunpipc.a

    cd lib

make

## **构建自己的编写代码的目录**

mkdir -p /root/bin/unpv2

cp /root/bin/unpv22e/libunpipc.a /root/bin/unpv22e/config.h /root/bin/unpv22e/Make.defines /root/bin/unpv2

复制各个子目录下得\*.h头文件和Makfile文件，然后执行

    cp /root/bin/unpv22e/dir/\*.h /root/bin/unpv22e/dir/Makefile /root/bin/unpv2

    make filename

     即可编译各个子目录下的代码

参看：https://blog.csdn.net/wm\_1991/article/details/49184335

# 概述

大多数重要的程序都涉及进程间通信(Tnterprflcess Communication, LPC )。这是受下述设计原则影响的自然结果:把应用程序设计为一组相互通信的小片一断比将其设计为单个庞大的程序更好。从历史角度看，应用程序有如下几种构建方法。

(1)用个庞大的程序完成全部工作。程序的各部分可以实现为函数，函数之间通过参数、返回值和全局变量来交换信息。

(2)使用多个程序，程序之间用某种形式的IPC进行通信。许多标准的Llnix工具都是按这种风格设计的，它们使用shell管道( IPC的一种形式)在程序之间传递信息。

(3)使用一个包含多个线程的程序，线程之间使用某种IPC。这里仍然便用术语IPC，尽管通信是在线程之间而不是在进程之间进行的。

本书详细描述了以下4种不同的IPC形式:

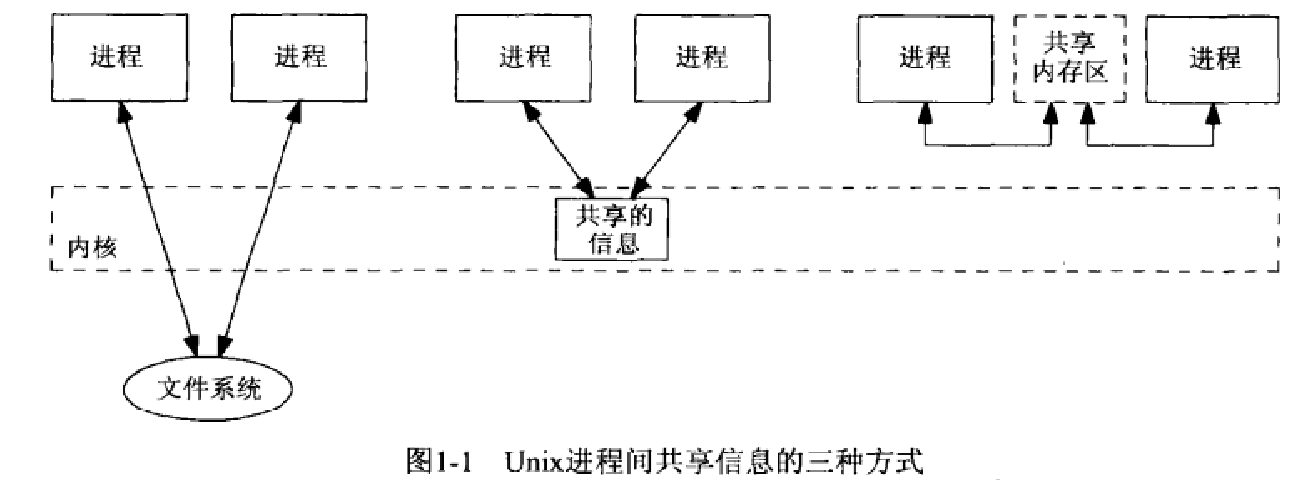
(1)消息传递(管道、FIFE和消息队列);

(2)同步(\_互斥量、条件变量、读写锁、文件和记录锁、信号量):

[3)共享内存(匿名的和具名的);

[4)远程过程调用(Soiaris门和Sun RPC )

## 1、2进程、线程与信息共享



(1)左边的两个进程共享存留7乙文件系统中某个文件l几的某峡信息。为访问这些信息，梅个进程都得穿越内核(例如read, wrl gee, leek等)。当一个文件有待更新时，某种形式的同步是必要的，这样既可保护多个，3入者。防ll:相场\_串扰，也可保护一个或多个读出者，防止写入者的干扰。

(2)中间的两个进程共享驻留尹内核中的某此信息。管道是这种共享类型的一个例子，System V消息队列和System V信一号量也是。现在访问共享信息的梅次操作涉及对内核的一次系统调用。

(3)右边的两个进程有一个双方都能访问的共享内存区0个进程一旦设置好该共享内存区，就能根本不涉及内核id}7访问其中的数据。共享该内存区的进程需要某种形式的同步。

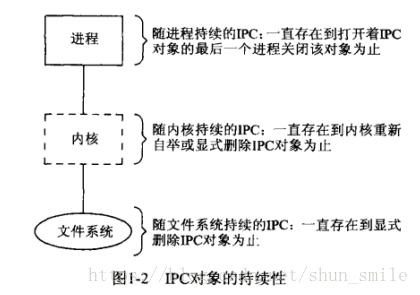
## IPC对象的持续性

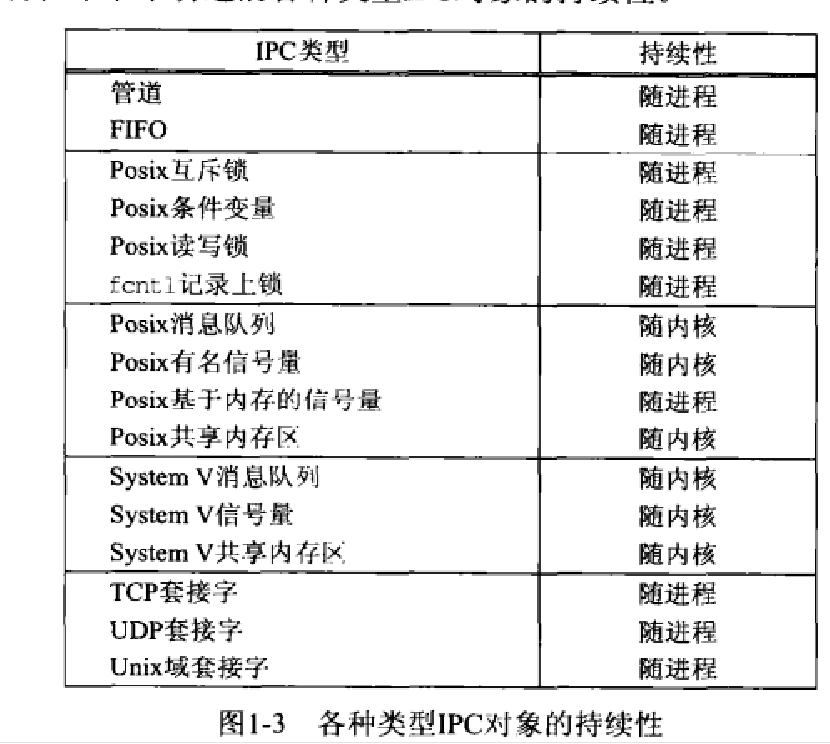
随进程持续： IPC对象一直存在到打开着该对象的最后一个进程关闭该对象为止。例如管道和FIFO。

随内核持续： IPC对象一直存在到内核重新自举或显式删除该对象为止。例如消息队里、信号量和共享内存。

随文件系统持续： IPC对象一直存在到显式删除该对象为止。例如使用映射文件实现的Posix消息队里、信号量和共享内存。

2.Posix IPC 和 System V IPC



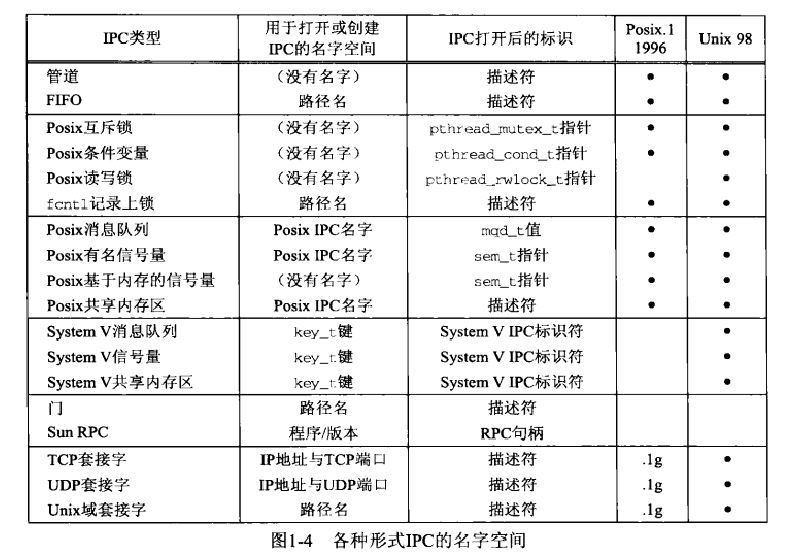


## 名字空间

当两个或多个尤亲缘关系的进程便用某种类型的IPC对象来彼此交换信息时，该IPC对象必须有一个某种形式的名字(name)或标识符C identifier?，这样其中一个进程(往往是服务器)可以创建该IPC对象了其余进程则可以指定同，个IPC对象。

管道没有名字(因此不能用于无亲缘关系的进程间)。但是FIFO有一个在文件系统中的Unix路径名作为其标一识符〔因此可用于无亲缘关系的进程问)。在以后各章具体讲述其他形式的IPC时，我们将使用另外的命名约定。对J几」种给定的IPC类型，其可能的名字的集合称为它的名字

空问(name space ) o名字空间非常重要，因为对于除普通管道以外的所有形式的IPC来说，名字是客户与服务器彼此连接以交换消息的手段。



Posix标准：知道为啥叫pthread的了。。。

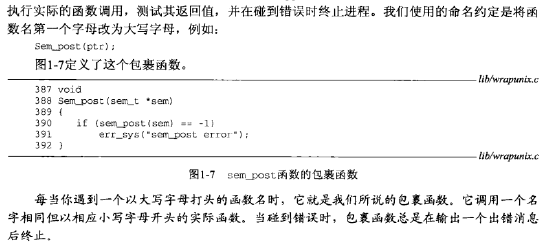
## fork exec和exit七对IPC对象的影响

d



表中多数特性将在以后讲述，不过我们需要强调几点。首先，考虑到无名同步变量(互斥锁、条件变量、读写锁、墓于内存的信号量)，从一个具有多个线程的进程中调用fork将变得混乱不堪

## 出错处理:包裹函数



Unix errno值

每当在一个Unix函数中发生错误时，全局变量errno将被设置成一个指示错误类型的正数，函数本身则通常返回-1。我们的err-sys函数检杳errno的值并输出相应的出错消息，例如，errno的值等于EAGAIN时的出错消J息为“Resource temporarily unavailahle"(资源暂时不可用)。

errno的值只在某个函数发生错误时设置。如果该函数不返回错误，errno的值就尤定义。所有正的错误值都是常值，具有以E;打头的全部为大写字母的名字，通常定义在头文件sys/errno .h》中。没有值为0的错误。

在多线程环境中，每个线程必须有自己的errno变量。提供一个局限

于线程的errno变量的隐式请求是自动处理的，不过通常需要告诉编译器所编译的程序是可重入的。给编译器指定类似-}一 REENTRAN'L'或-D\_ Pp$工x水SC]L1R[.'E-..-1 99 }U G几这样的命令行选项是较典型的方法口

errno. h>头文件往往把errno定义成」个宏，一场常值\_RELNTRANT有定义时，该宏就扩展成一个

函数，由它访问errno变最的某个局限于线程的副木。

全书使用类似“-n}sende}数返}u,IEMSGS=LE错误”的用语来简略地表示这样的意思:该函

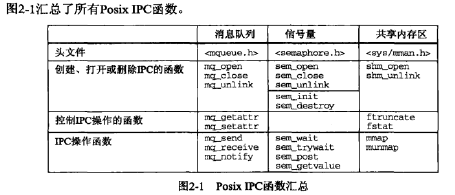
数返回个错误(典型情况是返【司值为一1)}并fL在errno中设置了指定的常值。

## Unix标准

SYSTEM V进程通信。包含消息队列、共享内存和信号量。   
POSIX 进程通信。新标准的进程通信。

# Posix IPC

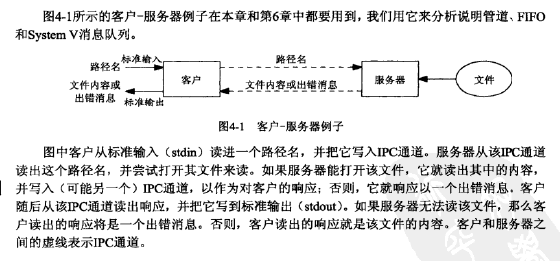
## 概述



# 管道和FIFO

但它们的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。

## 一个简单的客户一服务器例子



## 管道

所有式样的Unix都提供管道。它由pipe函数创建，提供一个单路(单向)数据流。

#include<unistd.h>

int pipe(int filedes[2]);

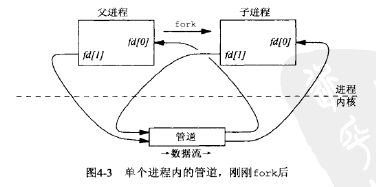
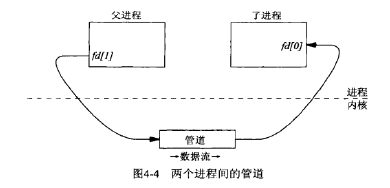
返回值：成功，返回0，否则返回-1。参数数组包含pipe使用的两个文件的描述符。fd[0]:读管道，fd[1]:写管道。

必须在fork()中调用pipe()，否则子进程不会继承文件描述符。两个进程不共享祖先进程，就不能使用pipe。但是可以使用命名管道

**特殊情况：**   
读一个空管道会阻塞。写一个满管道（塞进了65535个字符）也会阻塞。   
写一个读端被关闭的管道，会发出SIGPIPE信号。   
读一个写端关闭的管道，则read直接返回0。

### 开发

尽管管道是由单个进程创建的，’却很少在单个进程内使用(我们将在图5-14中给出在单个进程内使用i的一个例子)。管进的典型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。由一个进程(它将成为父进程)创建一个管道后调用fork派生一个自身的副本，如图4-3所示。

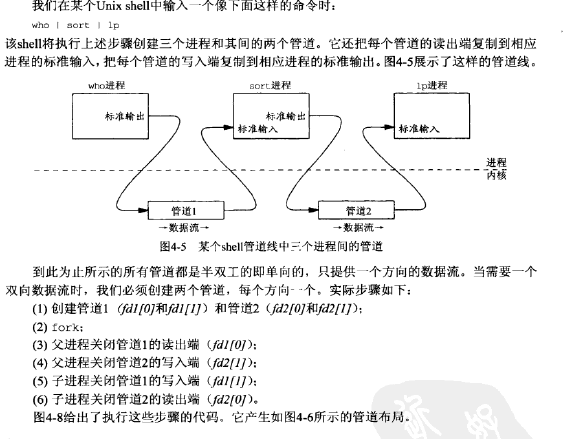
### 使用场景-shell

有些版本的Unix(例如5VR4)提供全双工管道，也就是说这些管道的两端都是既可用于读，也可用于写.创建一八全双工Ii'C'管道的另一种方法是使用UNPvI的I4.3节中讲述的socketpair函数，它在大多数现linux系统都能工柞.然而管道的最常见用途是用在各种shell中，这种情况下半双工管道足够了

key@ubuntu:~/aosp/android-9.0.0\_r35$ who | sort

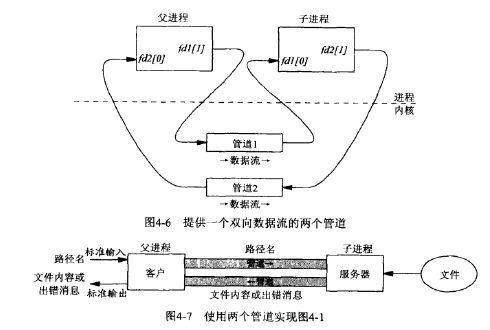
key pts/0 2019-06-07 08:36 (192.168.0.7)

key pts/1 2019-06-07 08:36 (192.168.0.7)



### 开发实现

现在使用管道实现4.2竹中描述的客户一服务器例子。main函数创建两个管道并用fork成 一个子进程口客户然后作为父进移运行，服务器则作为子进程运行。第2个管道用于从客户向服务器发送路径名，第一个管道用于从服务器向客户发送该文件的内容(或者一个出错消息)。这样设置后的布局如图4-7所示。



注意本图所示的两个管道直接连接着两个进程，然而实际上它们都是通过内核运作的，如图4-6所示。因此从客户到服务器以及从服务器到客户的所有数据都穿越了用户一内核接口两次;1次是在写入管道时，2次是在从管道读出时。

key@ubuntu:~/aosp/unpv22e/pipe$ ./mainpipe

pipe.xml

test pipe xml

#### 主程序

显然，父进程只能做客户端，方便终端用于输入数据

**int** main(**int** argc, **char** \*\*argv)  
{  
 **int** pipe1[2], pipe2[2];  
 pid\_t childpid;  
  
 Pipe(pipe1); */\* create two pipes \*/* Pipe(pipe2);  
  
 **if** ( (childpid = Fork()) == 0) { */\* child \*/* Close(pipe1[1]);  
 Close(pipe2[0]);  
 server(pipe1[0], pipe2[1]);*/\* 服务器\*/* exit(0);  
 }  
 */\* 4parent \*/* Close(pipe1[0]);  
 Close(pipe2[1]);  
  
 client(pipe2[0], pipe1[1]);//父进程作为客户端  
  
 Waitpid(childpid, NULL, 0); */\* wait for child to terminate \*/* exit(0);   
}

服务器(子进程)在往管道写入最终数据后调用exit首先终止。它随后变成了一个僵尸进程(Zombie ):白身已终止.但其父进程仍在运行且尚未等待该子进程的进程。子进程终止时，内核还给其父进程产生SIGCHID信号，不过父进程没有捕获这个信号，而该信号的默认行为就是忽略。此后不久，父进程的client函数在从管道读入最终数据后返回。父进程随后调用Waitpid取得已终止的进程(僵尸进程)的终止状态。要是父进程没有调用Waitpid，而是直接终止，那么子进程将成为托孤给init进程的孤儿进程，内核将为此向init进程发送另外1个SIGCHID信号，init进程随后将取得该僵尸进程的终止状态。

这里的管道一定要对应，close不是必须的，但是为了安全起见

#### 客户端

#include **"unpipc.h"  
  
void** client(**int** readfd, **int** writefd)  
{  
 size\_t len;  
 ssize\_t n;  
 **char** buff[MAXLINE];  
  
 */\* 4read pathname 从标准输入读进路径名\*/* Fgets(buff, MAXLINE, stdin);  
 len = strlen(buff); */\* fgets() guarantees null byte at end \*/* **if** (buff[len-1] == **'\n'**)  
 len--; */\* delete newline from fgets() \*/  
  
 /\* 4write pathname to IPC channel ,向写管道写入路径\*/* Write(writefd, buff, len);  
  
 */\* 4read from IPC, write to standard output 从读管道复制到标准输出\*/* **while** ( (n = Read(readfd, buff, MAXLINE)) > 0)  
 Write(STDOUT\_FILENO, buff, n);  
}

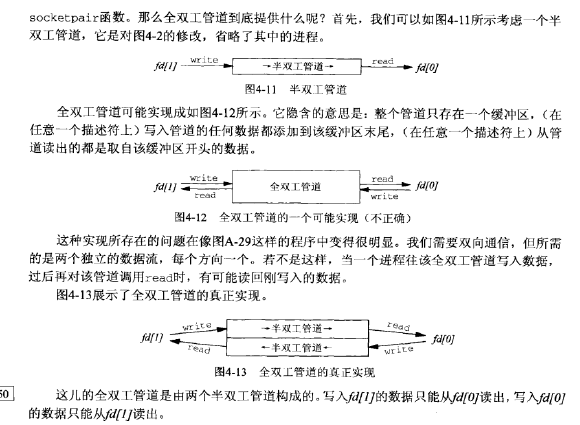
#### 服务端

#include **"unpipc.h"  
  
void** server(**int** readfd, **int** writefd)  
{  
 **int** fd;  
 ssize\_t n;  
 **char** buff[MAXLINE+1];  
  
 */\* 4read pathname from IPC channel  
 从管道读出客户写入的路径名，并以空字节作为其结尾。  
 注意，对一个管道的read只要该管道中存在一些数据就会马返回 ,  
 已不必等待达到所请求的字节数(本例中为MAXLINE)\*/* **if** ( (n = Read(readfd, buff, MAXLINE)) == 0)  
 err\_quit(**"end-of-file while reading pathname"**);  
 buff[n] = **'\0'**; */\* null terminate pathname \*/* **if** ( (fd = open(buff, O\_RDONLY)) < 0) {  
 */\* 4error: must tell client ，打开文件，处理错误\*/* snprintf(buff + n, **sizeof**(buff) - n, **": can't open, %s\n"**,  
 strerror(errno));*//strerror函数以返回对应的errno的出错消息串* n = strlen(buff);  
 Write(writefd, buff, n);  
  
 } **else** {  
 */\* 4open succeeded: copy file to IPC channel \*/* **while** ( (n = Read(fd, buff, MAXLINE)) > 0)  
 Write(writefd, buff, n);  
 Close(fd);  
 }  
}

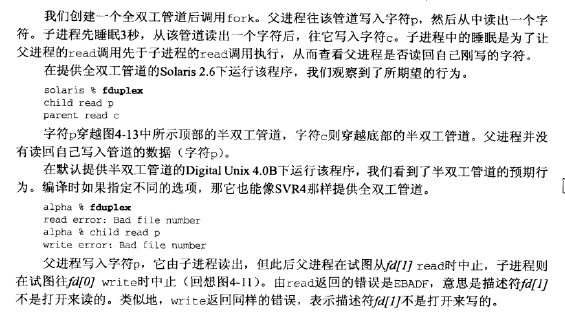
思考，为何不直接做成双工的呢？？？

## 全双工管道

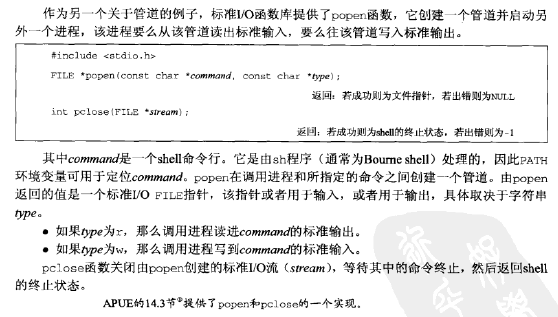
上节中我们提到，某些系统提供全}R.1}管道:5VR4的pipe函数以及许多内核都提供的，socketpair函数。那么全双工管道到底提供什么昵?首先，我们可以如图4-11所示考虑个半双工管道，它是对图4-2的修改，省略了其中的进程。



#include **"unpipc.h"  
  
int** main(**int** argc, **char** \*\*argv)  
{  
 **int** fd[2], n;  
 **char** c;  
 pid\_t childpid;  
  
 Pipe(fd); */\* assumes a full-duplex pipe (e.g., SVR4) \*/* **if** ( (childpid = Fork()) == 0) { */\* child \*/* sleep(3);  
 **if** ( (n = Read(fd[0], &c, 1)) != 1)  
 err\_quit(**"child: read returned %d"**, n);  
 printf(**"child read %c\n"**, c);  
 Write(fd[0], **"c"**, 1);  
 exit(0);  
 }  
 */\* 4parent fd[1]读和写，不会返回自己的数据哦\*/* Write(fd[1], **"p"**, 1);  
 **if** ( (n = Read(fd[1], &c, 1)) != 1)  
 err\_quit(**"parent: read returned %d"**, n);  
 printf(**"parent read %c\n"**, c);  
 exit(0);  
}



## popen和pclose函数



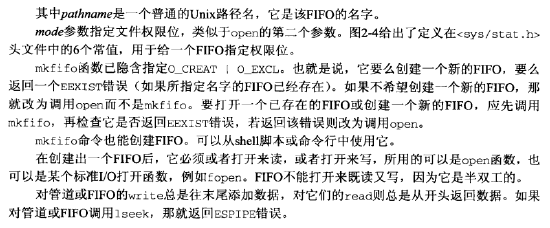
#include **"unpipc.h"  
  
int** main(**int** argc, **char** \*\*argv)  
{  
 size\_t n;  
 **char** buff[MAXLINE], command[MAXLINE];  
 FILE \*fp;  
  
 */\* 4read pathname \*/* Fgets(buff, MAXLINE, stdin);  
 n = strlen(buff); */\* fgets() guarantees null byte at end \*/* **if** (buff[n-1] == **'\n'**)  
 n--; */\* delete newline from fgets() \*/* snprintf(command, **sizeof**(command), **"cat %s"**, buff);  
 fp = Popen(command, **"r"**);  
  
 */\* 4copy from pipe to standard output \*/* **while** (Fgets(buff, MAXLINE, fp) != NULL)  
 Fputs(buff, stdout);  
  
 Pclose(fp);  
 exit(0);  
}

## FIFO

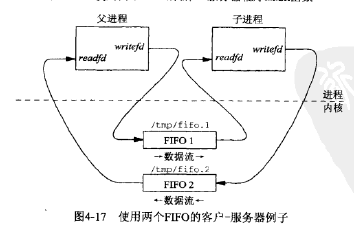
管道没有名字，因此它们的最大劣势是只能用于有一个共同祖先进程的各个进程之间。我们无法在无亲缘关系的两个进程间创建个管道井将它用作IPC通道(不考虑描述符传递)。 FIFO指代先进先出(first in First out ) } Unix中的FIFQ类似管道。’它是一个单向(半双工数据流。不同于管道的是，每个FIFO有个路径名与之关联，从而允许无亲缘关系的进程

访问同一个FIFO FIFO也称为有名管道(named pipe).

FIFD由mkfifo函数创建。

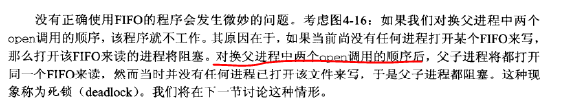


int mkfifo(const char \*pathname, mode\_t mode);

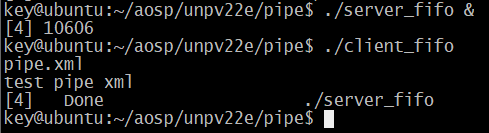


对管道FIFO的write总是往末尾添加数据，对它们的read则总是从头开始返回数据。如果对管道FIFO调用lseek，则返回ESPIPE错误。

#include **"unpipc.h"**#define FIFO1 **"/tmp/fifo.1"**#define FIFO2 **"/tmp/fifo.2"  
  
void** client(**int**, **int**), server(**int**, **int**);  
  
**int** main(**int** argc, **char** \*\*argv)  
{  
 **int** readfd, writefd;  
 pid\_t childpid;  
  
 */\* 4create two FIFOs; OK if they already exist \*/* **if** ((mkfifo(FIFO1, FILE\_MODE) < 0) && (errno != EEXIST))  
 err\_sys(**"can't create %s"**, FIFO1);  
 **if** ((mkfifo(FIFO2, FILE\_MODE) < 0) && (errno != EEXIST)) {  
 unlink(FIFO1);  
 err\_sys(**"can't create %s"**, FIFO2);  
 }  
  
 **if** ( (childpid = Fork()) == 0) { */\* child \*/* readfd = Open(FIFO1, O\_RDONLY, 0);  
 writefd = Open(FIFO2, O\_WRONLY, 0);  
  
 server(readfd, writefd);  
 exit(0);  
 }  
 */\* 4parent \*/* writefd = Open(FIFO1, O\_WRONLY, 0);  
 readfd = Open(FIFO2, O\_RDONLY, 0);  
  
 client(readfd, writefd);  
  
 Waitpid(childpid, NULL, 0); */\* wait for child to terminate \*/* Close(readfd);  
 Close(writefd);  
  
 Unlink(FIFO1);  
 Unlink(FIFO2);  
 exit(0);  
}



### 例子无亲缘关系的客户与服务器



#include **"unpipc.h"**#define FIFO1 **"/tmp/fifo.1"**#define FIFO2 **"/tmp/fifo.2"**

#### 服务器

#include **"fifo.h"  
  
void** server(**int**, **int**);  
  
**int** main(**int** argc, **char** \*\*argv)  
{  
 **int** readfd, writefd;  
  
 */\* 4create two FIFOs; OK if they already exist \*/* **if** ((mkfifo(FIFO1, FILE\_MODE) < 0) && (errno != EEXIST))  
 err\_sys(**"can't create %s"**, FIFO1);  
 **if** ((mkfifo(FIFO2, FILE\_MODE) < 0) && (errno != EEXIST)) {  
 unlink(FIFO1);  
 err\_sys(**"can't create %s"**, FIFO2);  
 }  
  
 readfd = Open(FIFO1, O\_RDONLY, 0);  
 writefd = Open(FIFO2, O\_WRONLY, 0);  
  
 server(readfd, writefd);  
 exit(0);  
}

#### 客户端

#include **"fifo.h"  
  
void** client(**int**, **int**);  
  
**int** main(**int** argc, **char** \*\*argv)  
{  
 **int** readfd, writefd;  
  
 writefd = Open(FIFO1, O\_WRONLY, 0);  
 readfd = Open(FIFO2, O\_RDONLY, 0);  
  
 client(readfd, writefd);  
  
 Close(readfd);  
 Close(writefd);  
  
 Unlink(FIFO1);  
 Unlink(FIFO2);  
 exit(0);  
}

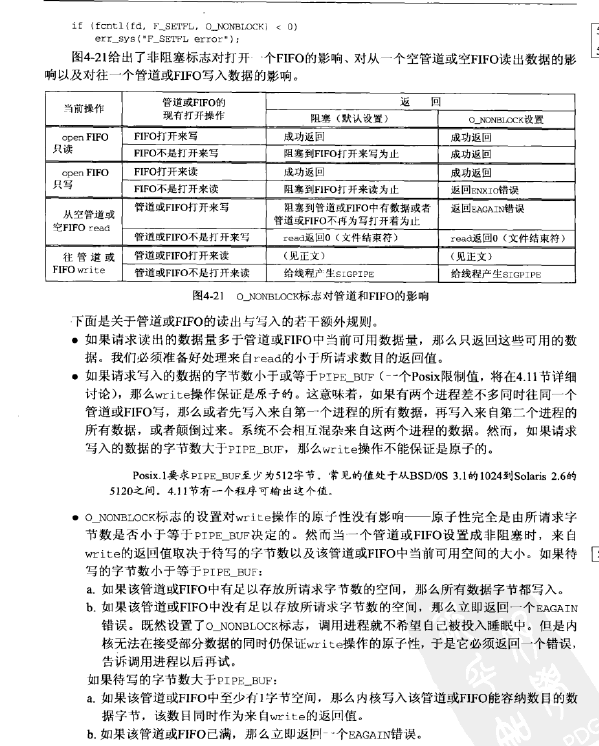
## 管道和FiFo的额外属性

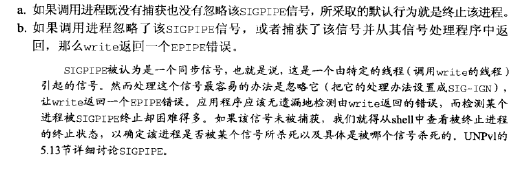
两种方式设置成非阻塞

（1）open时指定O\_NONBLOCK标志。

writefd = open(FIFO1,O\_WRONLY | O\_NONBLOCK,0);

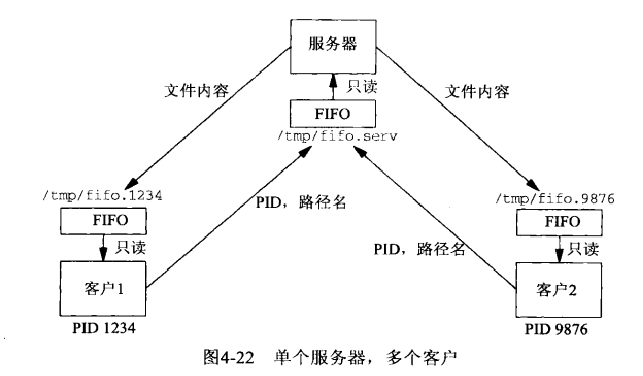
（2）如果描述符已经打开，用fcntl启用O\_NONBLOCK标志。





## 4.8单个服务器，多个客户

FIFO的真正优势表现在服务器可以是一个长期运行的进程(例如守护进程，第12章所述)，而且与其客户可以无亲缘关系。作为服务器的守护进程以某个众所周知的路径名创建一个FIFO，并打开该FIFO来读。此后某个时刻启动的客户打开该FIFO来写，并将其命令或给守护进程的其他柱何东西通过该FIFO发送出去。使用FIFO很齐易实现这种形式的单向通信(从客户到服务器)，但是如果守护进程需要往客客户端发送i东西，事情就困难了。



服务器以一个众所周知的路径名〔本例中为I}mplfifo.二二})创建一个FIF(7，它将从这

个FIFO读入客户的请求。每个客户在启动时创建自己的FIFE,所用的路径名含有自己的进程ID口每个客户把白己的请求写入服务器的众所周知FIF(7中，该请求含有客户的进程In以及个路径名，具有该路径名的文件就是客户希}J服务器打J「并发C}}的文件口

### 服务端代码

#include "fifo.h"

void server(int, int);

int main(int argc, char \*\*argv)

{

int readfifo, writefifo, dummyfd, fd;

char \*ptr, buff[MAXLINE], fifoname[MAXLINE];

pid\_t pid;

ssize\_t n;

/\* 4create server's well-known FIFO; OK if already exists \*/

if ((mkfifo(SERV\_FIFO, FILE\_MODE) < 0) && (errno != EEXIST))

err\_sys("can't create %s", SERV\_FIFO);

/\* 4open server's well-known FIFO for reading and writing \*/

readfifo = Open(SERV\_FIFO, O\_RDONLY, 0);

dummyfd = Open(SERV\_FIFO, O\_WRONLY, 0); /\* never used \*/

while ( (n = Readline(readfifo, buff, MAXLINE)) > 0) {

if (buff[n-1] == '\n')

n--; /\* delete newline from readline() \*/

buff[n] = '\0'; /\* null terminate pathname \*/

if ( (ptr = strchr(buff, ' ')) == NULL) {

err\_msg("bogus request: %s", buff);

continue;

}

\*ptr++ = 0; /\* null terminate PID, ptr = pathname \*/

pid = atol(buff);

snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);

if ( (writefifo = open(fifoname, O\_WRONLY, 0)) < 0) {

err\_msg("cannot open: %s", fifoname);

continue;

}

if ( (fd = open(ptr, O\_RDONLY)) < 0) {

/\* 4error: must tell client \*/

snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",

strerror(errno));

n = strlen(ptr);

Write(writefifo, ptr, n);

Close(writefifo);

} else {

/\* 4open succeeded: copy file to FIFO \*/

while ( (n = Read(fd, buff, MAXLINE)) > 0)

Write(writefifo, buff, n);

Close(fd);

Close(writefifo);

}

}

}

### 客户端代码

#include "fifo.h"

int main(int argc, char \*\*argv)

{

int readfifo, writefifo;

size\_t len;

ssize\_t n;

char \*ptr, fifoname[MAXLINE], buff[MAXLINE];

pid\_t pid;

/\* 4create FIFO with our PID as part of name \*/

pid = getpid();

snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);

if ((mkfifo(fifoname, FILE\_MODE) < 0) && (errno != EEXIST))

err\_sys("can't create %s", fifoname);

/\* 4start buffer with pid and a blank \*/

snprintf(buff, sizeof(buff), "%ld ", (long) pid);

len = strlen(buff);

ptr = buff + len;

/\* 4read pathname \*/

Fgets(ptr, MAXLINE - len, stdin);

len = strlen(buff); /\* fgets() guarantees null byte at end \*/

/\* 4open FIFO to server and write PID and pathname to FIFO \*/

writefifo = Open(SERV\_FIFO, O\_WRONLY, 0);

Write(writefifo, buff, len);

/\* 4now open our FIFO; blocks until server opens for writing \*/

readfifo = Open(fifoname, O\_RDONLY, 0);

/\* 4read from IPC, write to standard output \*/

while ( (n = Read(readfifo, buff, MAXLINE)) > 0)

Write(STDOUT\_FILENO, buff, n);

Close(readfifo);

Unlink(fifoname);

exit(0);

}

#### FIFQ writa的原子性

## 4.9对比迭代服务器与并发服务器

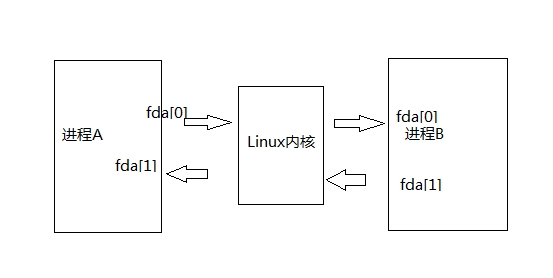
## 字节流与消息

## pipe()函数

管道是一种把两个进程之间的标准输入和标准输出连接起来的机制，从而提供一种让多个进程间通信的方法，当进程创建管道时，每次都需要提供两个文件描述符来操作管道。其中一个对管道进行写操作，另一个对管道进行读操作。对管道的读写与一般的IO系统函数一

致，使用write()函数写入数据，使用read()读出数据。

# 消息队列



## 参考

https://www.cnblogs.com/kunhu/p/3608109.html