

第四天管道与命名管道

一、回顾知识点

1.1 ps ja 查看进程状态

- a 显示终端上的所有进程，包含其他用户的进程
- j 显示父进程号、进程组号、会话号（sid）等信息

1.2 进程控制

```
#include <unistd.h>
_exit(int status_value) 进程的终止及状态值

#include <stdlib.h>
int atexit(void (*function)(void)); 退出清理回调

pid_t vfork(void) 创建新进程但不复制父进程的内存空间，且子进程先执行
```

进程替换：不会创建新的进程，只是将进程中执行的程序替换成新的程序

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ..., NULL);
int execlp(const char *filename, const char *arg0, ..., NULL);
int execl_e(const char *pathname, const char *arg0, ..., NULL, char *const envp[]);
int execl_v(const char *pathname, char *const argv[]);
int execl_p(const char *filename, char *const argv[]);

int execve(const char *pathname, char *const argv[], char *const envp[])
```

基于fork+exec执行外部命令或程序

```
#include <stdlib.h>
int system(const char *command);
```

1.3 进程间通信方式

进程间通信功能：

- 数据传输：一个进程需要将它的数据发送给另一个进程。
- 资源共享：多个进程之间共享同样的资源。
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件。
- 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有操作，并能够及时知道它的状态改变。

Linux 操作系统支持的主要进程间通信的通信机制：

UNIX: 信号、无名管道、有名管道、Socket
System V/POSIX: 消息队列、共享内存、信号量

1.4 信号

信号是**软件中断**，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。

每个信号的名字都以字符 SIG 开头, 命令: `kill -l` 查看所有信号

信号的基本操作

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signum);
int raise(int signum);

#include <unistd.h>
unsigned int alarm(unsigned int seconds);
int pause(void); 将调用进程挂起直至捕捉到信号为止

#include <stdlib.h>
void abort(void); 向进程发送一个 SIGABRT（无法阻塞的）信号，默认情况下进程会退出。
```

处理信号

程序中可用函数 `signal()` 改变信号的处理方式。

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

注册信号处理函数（**不可用于 SIGKILL、SIGSTOP 信号**），即确定收到信号后处理函数的入口地址。

handler 的取值：

忽略该信号：	SIG_IGN
执行系统默认动作：	SIG_DFL
自定义信号处理函数：	信号处理函数名

返回值：

成功：返回函数地址，该地址为此信号上一次注册的信号处理函数的地址。
失败：返回 SIG_ERR

使用 `sigaction()` 处理信号时，必须在第一行声明宏：

```
#define _XOPEN_SOURCE 700
```

```
#include <signal.h>
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

参数:

signum: 要操作的信号。
act: 要设置的对信号的新处理方式（传入参数）。
oldact: 原来对信号的处理方式（传出参数）。

如果 **act** 指针非空，则要改变指定信号的处理方式（设置），如果 **oldact** 指针非空，则系统将此前指定信号的处理方式存入 **oldact**，可以为NULL。

返回值: 成功: 0 失败: -1

```
struct sigaction
{
    void (*sa_handler)(int);           //旧的信号处理函数指针
    void (*sa_sigaction)(int, siginfo_t *, void *); //新的信号处理函数指针
    sigset_t sa_mask;                 //信号阻塞集
    int sa_flags;                     //信号处理的方式
    void (*sa_restorer)(void);        //已弃用
};
```

1) **sa_handler**、**sa_sigaction**: 信号处理函数指针，和 **signal()** 里的函数指针用法一样，应根据情况给 **sa_sigaction**、**sa_handler** 两者之一赋值，其取值如下:

- a) **SIG_IGN**: 忽略该信号
- b) **SIG_DFL**: 执行系统默认动作
- c) 处理函数名: 自定义信号处理函数

2) **sa_mask**: 信号阻塞集，在信号处理函数执行过程中，临时屏蔽指定的信号。

3) **sa_flags**: 用于指定信号处理的行为，通常设置为 0，表使用默认属性。它可以是以下值的“按位或”组合:

SA_RESTART: 使被信号打断的系统调用自动重新发起（已经废弃）
SA_NOCLDSTOP: 使父进程在它的子进程暂停或继续运行时不会收到 **SIGCHLD** 信号。
SA_NOCLDWAIT: 使父进程在它的子进程退出时不会收到 **SIGCHLD** 信号，这时子进程如果退出也不会成为僵尸进程。
SA_NODEFER: 使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。
SA_RESETHAND: 信号处理之后重新设置为默认的处理方式。
SA_SIGINFO: 使用 **sa_sigaction** 成员而不是 **sa_handler** 作为信号处理函数

信号处理函数:

```
void(*sa_sigaction)(int signum, siginfo_t *info, void *context);
```

参数说明:

signum: 信号的编号。
info: 记录信号发送进程信息的结构体。
context: 可以赋给指向 **ucontext_t** 类型的一个对象的指针，以引用在传递信号时被中断的接收进程

1.5 可重入函数

可重入函数是指函数可以由多个任务并发使用，而不必担心数据错误。

编写可重入函数：

- 1、不使用（返回）静态的数据、全局变量（除非用信号量互斥）。
- 2、不调用动态内存分配、释放的函数。
- 3、不调用任何不可重入的函数（如标准 I/O 函数）。

1.6 信号集

在 PCB 中有两个非常重要的信号集。一个称之为“阻塞信号集”，另一个称之为“未决信号集”。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对其进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改。

信号集是用来表示多个信号的数据类型。

信号集数据类型： `sigset_t`

信号集相关的操作主要有如下几个函数：

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signum);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

1.7 信号阻塞集

创建一个阻塞集合

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

参数：how：信号阻塞集合的修改方法

SIG_BLOCK：向信号阻塞集合中添加 `set` 信号集
SIG_UNBLOCK：从信号阻塞集合中删除 `set` 集合
SIG_SETMASK：将信号阻塞集合设为 `set` 集合

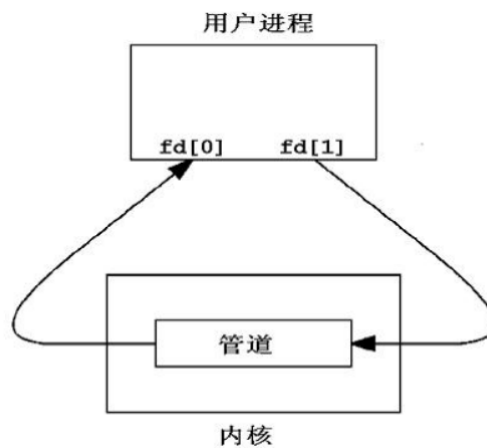
二、管道PIPE

2.1 管道概述

管道(pipe)又称无名管道。

```
int fd[2]; fd[0] 读, fd[1] 写
```

无名管道是一种特殊类型的文件，在应用层体现为两个打开的文件描述符。



特点：

- 1、半双工，数据在同一时刻只能在一个方向上流动。
- 2、数据只能从管道的一端写入，从另一端读出。
- 3、写入管道中的数据遵循先入先出的规则。
- 4、管道所传送的数据是无格式的，这要求管道的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
- 5、管道不是普通的文件，不属于某个文件系统，其只存在于内存中。
- 6、管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
- 7、从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据。
- 8、管道没有名字，只能在具有公共祖先的进程之间使用

2.2 创建pipe

```
#include <unistd.h>
int pipe(int filedes[2]);
```

功能：经由参数 filedes 返回两个文件描述符

filedes[0]为读而打开，filedes[1]为写而打开管道。
filedes[0]的输出是filedes[1]的输入。

返回值：成功返回 0，失败返回-1。

【注意】利用无名管道实现进程间的通信，都是父进程创建无名管道，然后再创建子进程，子进程继承父进程的无名管道的文件描述符，然后父子进程通过读写无名管道实现通信

如：普通的父子进程通信

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    int f = pipe(fd);
    if (f == -1)
    {
        perror("pipe");
        return 1;
    }
}
```

```

int pid = fork();
if (pid == 0)
{
    // 子进程： 读
    // 关闭写的通道
    close(fd[1]);
    char buf[64] = "";
    int len = read(fd[0], buf, 64); // 可能会阻塞
    buf[strlen(buf)] = '\0';

    printf("%d 子进程读取到数据: %s\n", getpid(), buf);
    // 管道使用完之后， 记得关闭
    close(fd[0]);
    _exit(0);
}
else if (pid > 0)
{
    // 父进程， 写数据
    close(fd[0]);
    printf("父进程(%d) 等待5秒中向子进程(%d) 写数据\n", getpid(), pid);
    sleep(5);
    char buf[64] = "hello,disen!";
    write(fd[1], buf, strlen(buf));
    close(fd[1]);
}

return 0;
}

```

```

● disen@qfxxa:~/code3/day04$ ./a.out
父进程(10306) 等待5秒中向子进程(10307) 写数据
10307 子进程读取到数据: hello,disen!

```

2.3 读写数据的特点

- 1、默认用 `read` 函数从管道中读数据是阻塞的。
- 2、调用 `write` 函数向管道里写数据，当缓冲区已满时 `write` 也会阻塞。
- 3、通信过程中，读端口全部关闭后，写进程向管道内写数据时，写进程会（收到 `SIGPIPE` 信号）退出。

【扩展】编程时可通过 `fcntl` 函数设置文件的阻塞特性。

```

设置为阻塞： fcntl(fd, FSETFL, 0);
设置为非阻塞： fcntl(fd, FSETFL, O_NONBLOCK);

```

如1：写满管道 写端被阻塞

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

```

```

int main(int argc, char const *argv[])
{
    int fd[2];
    int f = pipe(fd);
    if (f == -1)
    {
        perror("pipe");
        return 1;
    }
    int pid = fork();
    if (pid == 0)
    {
        // 子进程：读
        // 关闭写的通道
        close(fd[1]);
        sleep(2);
        close(fd[0]);
        _exit(0);
    }
    else if (pid > 0)
    {
        // 父进程，写数据
        close(fd[0]);
        int bufsize = 0;
        for (int i = 0; i < 10000; i++)
        {
            char buf[64] = "hello,disen!";
            write(fd[1], buf, strlen(buf));
            bufsize += 12;
            printf("主进程(%d) 第%d 次写数据，缓冲区的字节数%d\n", getpid(), i,
bufsize);
        }

        close(fd[1]);
    }

    return 0;
}

```

```

主进程(10594) 第5445 次写数据，缓冲区的字节数65352
主进程(10594) 第5446 次写数据，缓冲区的字节数65364
主进程(10594) 第5447 次写数据，缓冲区的字节数65376
主进程(10594) 第5448 次写数据，缓冲区的字节数65388
主进程(10594) 第5449 次写数据，缓冲区的字节数65400
主进程(10594) 第5450 次写数据，缓冲区的字节数65412
主进程(10594) 第5451 次写数据，缓冲区的字节数65424
主进程(10594) 第5452 次写数据，缓冲区的字节数65436
主进程(10594) 第5453 次写数据，缓冲区的字节数65448
主进程(10594) 第5454 次写数据，缓冲区的字节数65460
主进程(10594) 第5455 次写数据，缓冲区的字节数65472
disen@qfxa:~/code3/day04$

```

测试结果说明，管道的缓冲区的大小：`64K`

如2：通信过程中 写端关闭 读端将解阻塞

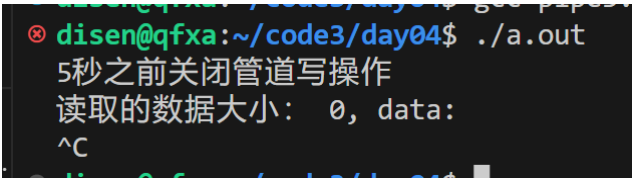
```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    int f = pipe(fd);
    if (f == -1)
    {
        perror("pipe");
        return 1;
    }
    int pid = fork();
    if (pid == 0)
    {
        // 子进程： 读
        // 关闭写的通道
        close(fd[1]);
        char buf[1024] = "";
        int len = read(fd[0], buf, 1024);
        printf("读取的数据大小: %d, data: %s\n", len, buf);
        close(fd[0]);
        _exit(0);
    }
    else if (pid > 0)
    {
        // 父进程，写数据
        close(fd[0]);
        printf("5秒之前关闭管道写操作\n");
        sleep(5);
        close(fd[1]);
        while (1)
            ;
    }

    return 0;
}

```



```

disen@qfxa:~/code3/day04$ ./a.out
5秒之前关闭管道写操作
读取的数据大小: 0, data:
^C

```

如3：通信过程中 读端关闭 写端将收到SIGPIPE信号 退出写端进程

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    int f = pipe(fd);

```

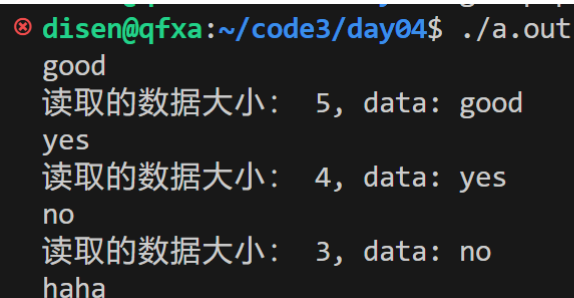


```

if (f == -1)
{
    perror("pipe");
    return 1;
}
int pid = fork();
if (pid == 0)
{
    // 子进程： 读
    // 关闭写的通道
    close(fd[1]);
    for (int i = 0; i < 3; i++)
    {
        char buf[1024] = "";
        int len = read(fd[0], buf, 1024);
        buf[len - 1] = '\0';
        printf("读取的数据大小:  %d, data: %s\n", len, buf);
    }
    close(fd[0]);
    _exit(0);
}
else if (pid > 0)
{
    // 父进程，写数据
    close(fd[0]);
    // 从键盘读数据写入到管道
    while (1)
    {
        char buf[32] = "";
        int len = read(STDIN_FILENO, buf, 32);
        write(fd[1], buf, len);
    }
    close(fd[1]);
    while (1)
        ;
}

return 0;
}

```



```

disen@qfxa:~/code3/day04$ ./a.out
good
读取的数据大小:  5, data: good
yes
读取的数据大小:  4, data: yes
no
读取的数据大小:  3, data: no
haha

```

如4：子进程从键盘读取数据，并写入管道中，父进程负责读取数据

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

```

```

#include <string.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    int f = pipe(fd);
    if (f == -1)
    {
        perror("pipe");
        return 1;
    }
    int pid = fork();
    if (pid > 0)
    {
        // 父进程： 读
        // 关闭写的通道
        close(fd[1]);
        for (int i = 0; i < 3; i++)
        {
            char buf[1024] = "";
            int len = read(fd[0], buf, 1024);
            buf[len - 1] = '\0';
            printf("读取的数据大小:  %d, data: %s\n", len, buf);
        }
        close(fd[0]);
        _exit(0);
    }
    else if (pid == 0)
    {
        // 子进程，写数据
        close(fd[0]);
        // 从键盘读数据写入到管道
        while (1)
        {
            char buf[32] = "";
            int len = read(STDIN_FILENO, buf, 32);
            write(fd[1], buf, len);
        }
        close(fd[1]);
        while (1)
            ;
    }

    return 0;
}

```

```

● disen@qfxa:~/code3/day04$ ./a.out
123
读取的数据大小:  4, data: 123
yes
读取的数据大小:  4, data: yes
good
读取的数据大小:  5, data: good
○ disen@qfxa:~/code3/day04$ █

```

【注意】当父进程结束时，子进程无法从标准输入终端获取数据了。

三、文件描述符复制

3.1 文件描述符概述

文件描述符是非负整数，是文件的标识。

用户使用文件描述符（file descriptor）来访问文件。

利用 open 打开一个文件时，内核会返回一个文件描述符

每个进程都有一张文件描述符的表，进程刚被创建时，标准输入、标准输出、标准错误输出设备文件被打开，对应的文件描述符 0、1、2 记录在表中。

在进程中打开其他文件时，系统会返回文件描述符表中最小可用的文件描述符，并将此文件描述符记录在表中

【注意】Linux 中一个进程最多只能打开 NR_OPEN_DEFAULT（即 1024）个文件，故当文件不再使用时应及时调用 close 函数关闭文件

如：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    close(1);
    // 1 -> a.txt
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_APPEND, 0755);
    // char buf[32] = "";
    // sprintf(buf, "fd=%d\n", fd);
    // write(fd, buf, 5);
    printf("hello, fd=%d\n", fd);
    printf("good\n");

    return 0;
}
```

```
● disen@qfxa:~/code3/day04$ ./a.out
● disen@qfxa:~/code3/day04$ cat a.txt
hello, fd=1
good
```

【扩展】 查看所有.c文件中read所在的行

```
disen@qfxa:~/code3/day04$ find *.c|xargs grep read -n
pipe1.c:22:      int len = read(fd[0], buf, 64); // 可能会阻塞
pipe3.c:22:      int len = read(fd[0], buf, 1024);
pipe4.c:24:      int len = read(fd[0], buf, 1024);
pipe4.c:39:      int len = read(STDIN_FILENO, buf, 32);
pipe5.c:24:      int len = read(fd[0], buf, 1024);
pipe5.c:39:      // int len = read(STDIN_FILENO, buf, 32);
```

3.2 dup和dup2函数

dup 和 dup2 是两个非常有用的系统调用，都是用来复制一个文件的描述符，使新的文件描述符也标识旧的文件描述符所标识的文件。

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup 和 dup2 经常用来重定向进程的 stdin、stdout 和 stderr。

3.2.1 dup应用

```
int dup(int oldfd);
```

复制 oldfd 文件描述符，并分配一个新的文件描述符，新的文件描述符是调用进程文件描述符表中最小可用的文件描述符。

成功：新文件描述符。

失败：返回 -1，错误代码存于 errno 中

如：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_APPEND, 0755);
    printf("新打开的fd = %d\n", fd);
    close(1);
    // 复制fd为一个新的文件描述符（最小的，1）
    int newfd = dup(fd); // 1 -> a.txt
    printf("newfd = %d\n", newfd);
    printf("hahah\n");
    close(fd);
    printf("yes\n");
    return 0;
}
```

```

● disen@qfxa:~/code3/day04$ ./a.out
新打开的fd = 3
● disen@qfxa:~/code3/day04$ cat a.txt
newfd = 1
hahah
yes

```

3.2.2 dup2应用

```
int dup2(int oldfd, int newfd)
```

复制一份打开的文件描述符 oldfd，并分配新的文件描述符 newfd，newfd 也标识 oldfd 所标识的文件。

oldfd 要复制的文件描述符，newfd 分配的新的文件描述符。

成功：返回 newfd

失败：返回-1，错误代码存于 errno 中

【注意】 newfd 是小于文件描述符最大允许值的非负整数，如果 newfd 是一个已经打开的文件描述符，则首先关闭该文件，然后再复制。

如：

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_APPEND, 0755);
    printf("新打开的fd = %d\n", fd);
    // dup2() 先关闭1，再复制fd为1
    int newfd = dup2(fd, 1); // 1 -> a.txt
    close(fd);
    printf("newfd = %d\n", newfd);
    printf("no\n");
    printf("good\n");
    return 0;
}

```

```

● disen@qfxa:~/code3/day04$ cat a.txt
newfd = 1
no
good

```

3.3 复制的新旧文件描述符的特点

复制文件描述符后新旧文件描述符的特点:

- 1) 使用 `dup` 或 `dup2` 复制文件描述符后, 新文件描述符和旧文件描述符指向同一个文件, 共享文件锁定、读写位置和各项权限。
- 2) 当关闭新的文件描述符时, 通过旧文件描述符仍可操作文件。
- 3) 当关闭旧的文件描述符时, 通过新的文件描述符仍可操作文件。

`exec` 前后文件描述符的特点:

- 1) `close_on_exec` 标志决定了文件描述符在执行 `exec` 后文件描述符是否可用。
- 2) 文件描述符的 `close_on_exec` 标志默认是关闭的, 即文件描述符在执行 `exec` 后文件描述符是可用的。
- 3) 若没有设置 `close_on_exec` 标志位, 进程中打开的文件描述符, 及其相关的设置在 `exec` 后不变, 可供新启动的程序使用

设置 `close_on_exec` 标志位的方法:

```
int flags;
flags = fcntl(fd, F_GETFD); //获得标志
flags |= FD_CLOEXEC;        //打开标志位
flags &= ~FD_CLOEXEC;       //关闭标志位
fcntl(fd, F_SETFD, flags);  //设置标志
```

如: file4_0.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    // 设置1的close_on_exec 为打开（默认exec关闭的）
    int flags = fcntl(1, F_GETFD);
    flags |= FD_CLOEXEC;
    fcntl(1, F_SETFD, flags);
    exec1("./file4_1", "file4_1", NULL);
    return 0;
}
```

file4_1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    printf("hi, file4_1\n");
    return 0;
}
```

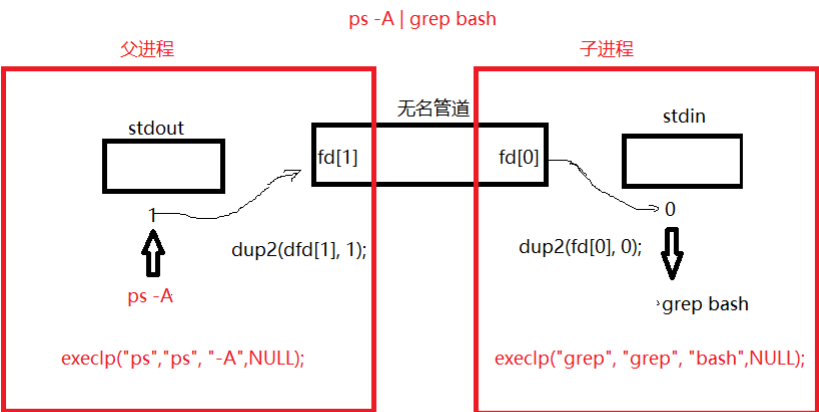
```

● disen@qfxa:~/code3/day04$ gcc file4_1.c -o file4_1
● disen@qfxa:~/code3/day04$ gcc file4_1.c -o file4_1
● disen@qfxa:~/code3/day04$ gcc file4_0.c
● disen@qfxa:~/code3/day04$ ./a.out
○ disen@qfxa:~/code3/day04$ █

```

案例1：

父子进程实现命令中管道的功能，如：`ps -A | grep bash`



如：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    if (pipe(fd) == -1)
    {
        perror("pipe");
        return 1;
    }
}
```

```

int pid = fork();
if (pid == 0)
{
    close(fd[1]);
    dup2(fd[0], 0); // 0->fd[0]
    execlp("grep", "grep", "bash", NULL);
    close(fd[0]);
    _exit(0);
}
else if (pid > 0)
{
    close(fd[0]);
    // 将命令执行的结果写入到管道到
    dup2(fd[1], 1); // 1->fd[1]
    execlp("ps", "ps", "-A", NULL);
    close(fd[1]);
}

return 0;
}

```

```

● disen@qfxa:~/code3/day04$ ./a.out
2035 pts/17    00:00:00 bash
9493 ?          00:00:00 bash
9711 pts/21    00:00:00 bash
18145 ?         00:00:00 bash

```

案例2:

借用外部命令，实现计算器功能

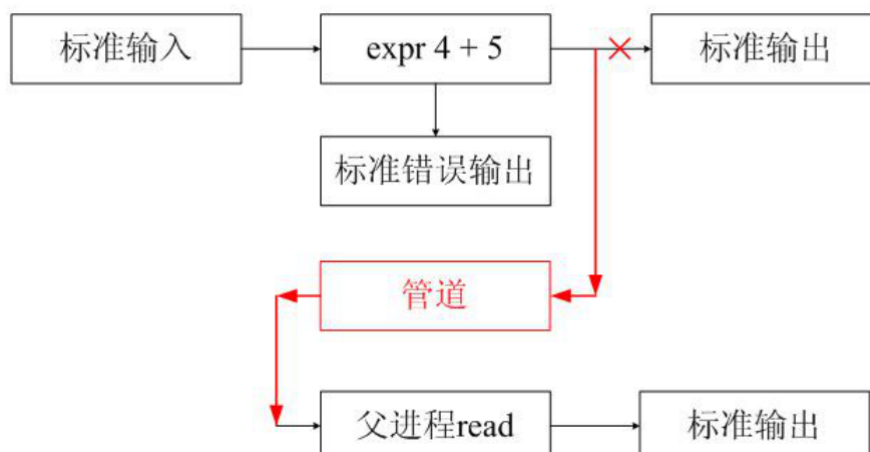
【提示】

`expr` 是个外部命令，它向标准输出打印运算结果。

创建一个管道以便让 `expr 4 + 5` 的输出到管道中

子进程 `exec` 执行 `expr 4 + 5` 命令之前重定向“标准输出”到“管道写端”。

父进程从管道读端读取数据，并显示运算结果



如:


```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
    int fd[2];
    if (pipe(fd) == -1)
    {
        perror("pipe");
        return 1;
    }

    int pid = fork();
    if (pid == 0)
    {
        close(fd[0]); // 关闭读
        // 重定向标准输出到管道
        dup2(fd[1], 1);
        execlp("expr", "expr", "4", "+", "5", NULL);
        close(fd[1]);
        _exit(0);
    }
    else if (pid > 0)
    {
        close(fd[1]);
        char buf[10] = "";
        int len = read(fd[0], buf, 10);
        buf[len - 1] = '\0';
        printf("读取子进程发送的数据: %s\n", buf);
        close(fd[0]);
    }

    return 0;
}

```

```

● disen@qfxa:~/code3/day04$ ./a.out
读取子进程发送的数据: 9

```

五、命名管道(FIFO)

5.1 FIFO概述

命名管道(FIFO)和管道(pipe)基本相同，但也有一些显著的不同。

特点：

- 1、半双工，数据在同一时刻只能在一个方向上流动。
- 2、写入 FIFO 中的数据遵循先入先出的规则。
- 3、FIFO 所传送的数据是无格式的，这要求 FIFO 的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
- 4、FIFO 在文件系统中作为一个特殊的文件而存在，但 FIFO 中的内容却存放在内存中。

- 5、管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
- 6、从 FIFO 读数据是一次性操作，数据一旦被读，它就从 FIFO 中被抛弃，释放空间以便写更多的数据。
- 7、当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
- 8、FIFO 有名字，不相关的进程可以通过打开命名管道进行通信

5.2 操作FIFO文件时的特点

系统调用的 I/O 函数都可以作用于 FIFO，如 open、close、read、write 等。

打开 FIFO 时，非阻塞标志(O_NONBLOCK)产生下列影响。

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

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

参数：pathname文件路径名，mode是文件的读写执行（r,w,x）相关的权限（可以使用0ddd数值表示，如0755）。

返回值：0成功，非0失败。

【注意】一个有名管道只能创建一次

5.2.1 特点1-不指定 O_NONBLOCK

不指定 O_NONBLOCK(即 open 没有位或 O_NONBLOCK)

- 1、open 以只读方式打开 FIFO 时，要阻塞到某个进程为写而打开此 FIFO
- 2、open 以只写方式打开 FIFO 时，要阻塞到某个进程为读而打开此 FIFO。
- 3、open 以只读、只写方式打开 FIFO 时会阻塞，调用 read 函数从 FIFO 里读数据时 read 也会阻塞。
- 4、通信过程中若写进程先退出了，则调用 read 函数从 FIFO 里读数据时不阻塞；若写进程又重新运行，则调用 read 函数从 FIFO 里读数据时又恢复阻塞。
- 5、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到 SIGPIPE 信号）退出。
- 6、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。

如1：fifo1.c, 先创建管道，再以只读方式打开【阻塞】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    // 创建fifo的有名(文件名)管道
    if (mkfifo("myfifo", 0755) != 0)
    {
        perror("mkfifo");
        return 1;
    }
}
```

```

}
printf("准备以只读或只写方式打开myfifo\n");
int fd = open("myfifo", O_RDONLY);
printf("open myfifo fd=%d\n", fd);
close(fd);
return 0;
}

```

```

● disen@qfxa:~/code3/day04$ gcc fifo1.c
○ disen@qfxa:~/code3/day04$ ./a.out
准备以只读或只写方式打开myfifo

```

如2：只写方式，打开管道（阻塞）

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("准备以只写方式打开myfifo\n");
    int fd = open("myfifo", O_WRONLY);
    printf("open myfifo fd=%d\n", fd);
    close(fd);
    return 0;
}

```

```

○ disen@qfxa:~/code3/day04$ ./a.out
准备以只写方式打开myfifo

```

如3：只读，如果没有写进程运行时，read()不会阻塞，如果写进程运行则立即阻塞

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("准备以只读方式打开myfifo\n");
    int fd = open("myfifo", O_RDONLY);
    printf("open myfifo fd=%d\n", fd);
    int n = 1;
    while (1)
    {

```

```

char buf[32] = "";
// 阻塞（写进程运行期间），非阻塞（没有写进程在运行）
int len = read(fd, buf, 32);
buf[len - 1] = '\0';
printf("第%d次读取的数据(%d): %s\n", n++, len, buf);
sleep(1);
}
close(fd);
return 0;
}

```

```

disen@qfxa:~/code3/day04$ gcc fifo1_read.c -o read
disen@qfxa:~/code3/day04$ ./read
准备以只读方式打开myfifo
open myfifo fd=3
第1次读取的数据(0):
第2次读取的数据(0):
第3次读取的数据(0):
第4次读取的数据(0):
第5次读取的数据(0):

```

如4：只写，5秒后退出

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("准备以只写方式打开myfifo\n");
    int fd = open("myfifo", O_WRONLY);
    printf("open myfifo fd=%d\n", fd);
    sleep(5);
    close(fd);
    return 0;
}

```

5.2.2 特点2-指定 O_NONBLOCK

指定 O_NONBLOCK(即 open 位或 O_NONBLOCK)

- 1、先以只读方式打开：如果没有进程已经为写而打开一个 FIFO，只读open成功，并且open不阻塞。
- 2、先以只写方式打开：如果没有进程已经为读而打开一个 FIFO，只写 open 将出错返回-1。
- 3、read、write 读写命名管道中读数据时不阻塞。
- 4、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到SIGPIPE 信号）退出

如1：非阻塞只读方式打开

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

```

```

#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("准备以只读方式打开myfifo\n");
    int fd = open("myfifo", O_RDONLY | O_NONBLOCK);
    printf("open myfifo fd=%d\n", fd);
    int n = 1;
    while (1)
    {
        char buf[32] = "";
        // 阻塞（写进程运行期间），非阻塞（没有写进程在运行）
        int len = read(fd, buf, 32);
        buf[len - 1] = '\0';
        printf("第%d次读取的数据(%d): %s\n", n++, len, buf);
        sleep(1);
    }
    close(fd);
    return 0;
}

```

如2：非阻塞只写方式打开

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("准备以只写方式打开myfifo\n");
    int fd = open("myfifo", O_WRONLY | O_NONBLOCK);
    if (fd == -1)
    {
        perror("open");
        return 1;
    }
    printf("open myfifo fd=%d\n", fd);
    sleep(5);
    close(fd);
    return 0;
}

```

```

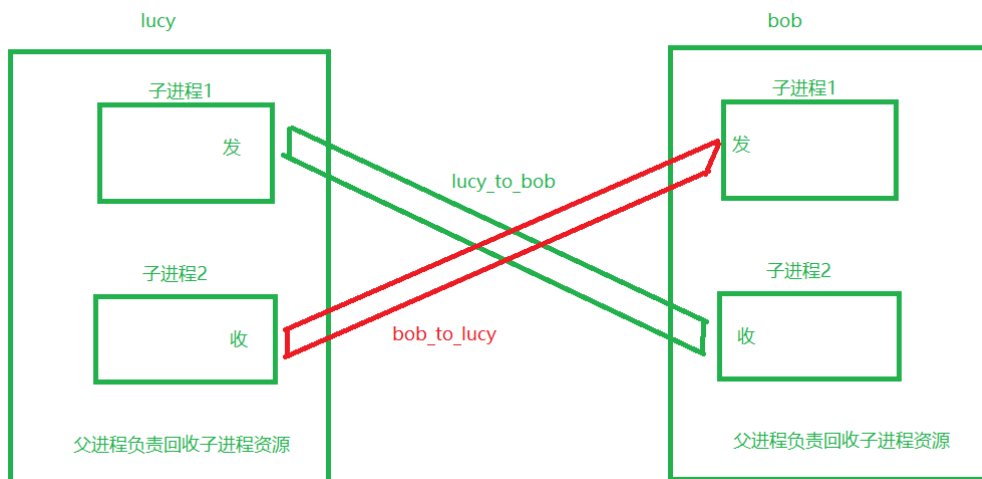
❌ disen@qfxa:~/code3/day04$ ./write2
准备以只写方式打开myfifo
open: No such device or address
○ disen@qfxa:~/code3/day04$ █

```

5.2.3 可读可写方式打开 FIFO 文件时的特点

- 1、open 不阻塞。
- 2、调用 read 函数从 FIFO 里读数据时 read 会阻塞。
- 3、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。

5.3 实现单机聊天程序



如: bbs_bob.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char const *argv[])
{
#ifdef CREATE_FIFO
    mkfifo("bob_2_lucy", 0755);
    mkfifo("lucy_2_bob", 0755);
#endif
    int i = 0;
    for (; i < 2; i++)
    {
        int pid = fork();
        if (pid == 0)
            break;
    }
    if (i == 0)
    { // 第一个子进程
        // 从键盘输入数据并发送给对方
        int fd = open("bob_2_lucy", O_WRONLY);
        while (1)
        {
            char buf[128] = "";
```

```

        int len = read(STDIN_FILENO, buf, 128);
        buf[len - 1] = '\0';
        write(fd, buf, len);
        if (strcmp(buf, "bye") == 0)
        {
            break;
        }
    }
    close(fd);
    _exit(0);
}
else if (i == 1)
{
    // 第二个子进程，读数据 (lucy->bob)
    int fd = open("lucy_2_bob", O_RDONLY);
    while (1)
    {
        char buf[128] = "";
        int len = read(fd, buf, 128);
        if (len <= 0)
            break;
        printf("接收Lucy: %s\n", buf);
    }
    close(fd);
    _exit(0);
}
else
{
    // 主进程
    while (1)
    {
        int pid_ = waitpid(0, NULL, WNOHANG);
        if (pid_ == -1)
        {
            // 所有的子进程都退出
            break;
        }
    }
}
return 0;
}

```

第一次编译时:

```
gcc bbs_bob.c -o bob -D CREATE_FIFO
```

第一次运行之后，还需要重新编译（不带 CREATE_FIFO 宏）

```
gcc bbs_bob.c -o bob
```

如: bbs_lucy.c

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

```

#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char const *argv[])
{
#ifdef CREATE_FIFO
    mkfifo("bob_2_lucy", 0755);
    mkfifo("lucy_2_bob", 0755);
#endif
    int i = 0;
    for (; i < 2; i++)
    {
        int pid = fork();
        if (pid == 0)
            break;
    }
    if (i == 0)
    { // 第一个子进程
        // 从键盘输入数据并发送给对方
        int fd = open("lucy_2_bob", O_WRONLY);
        while (1)
        {
            char buf[128] = "";
            int len = read(STDIN_FILENO, buf, 128);
            buf[len - 1] = '\0';
            write(fd, buf, len);
            if (strcmp(buf, "bye") == 0)
            {
                break;
            }
        }
        close(fd);
        _exit(0);
    }
    else if (i == 1)
    {
        // 第二个子进程，读数据 (lucy->bob)
        int fd = open("bob_2_lucy", O_RDONLY);
        while (1)
        {
            char buf[128] = "";
            int len = read(fd, buf, 128);
            if (len <= 0)
                break;
            printf("接收bob: %s\n", buf);
        }
        close(fd);
        _exit(0);
    }
    else
    {
        // 主进程
        while (1)
        {

```



```
        int pid_ = waitpid(0, NULL, WNOHANG);  
        if (pid_ == -1)  
        {  
            // 所有的子进程都退出  
            break;  
        }  
    }  
}  
return 0;  
}
```

运行结果:

```
bye  
● disen@qfxa:~/code3/day04$ ./lucy  
bob 干什么呢?  
接收bob: 我在学习 mkfifo  
接收bob: bye  
bye  
● disen@qfxa:~/code3/day04$
```

```
● disen@qfxa:~/code3/day04$ ./bob  
接收Lucy: bob 干什么呢?  
我在学习 mkfifo  
bye  
接收Lucy: bye  
● disen@qfxa:~/code3/day04$
```