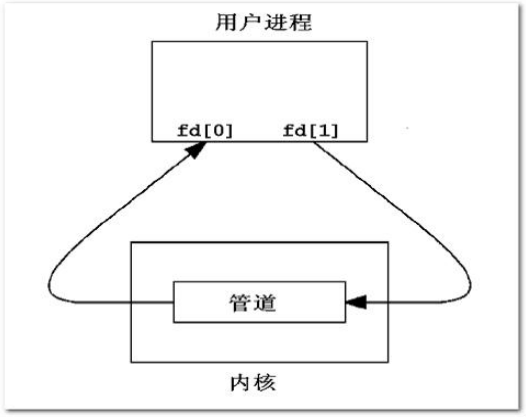
第五章：管道、命名管道

5.1 管道概述

管道(pipe)又称无名管道。
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。



管道是最古老的 UNIX IPC 方式，其特点是:
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

5.2 无名管道的创建 pipe 函数

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

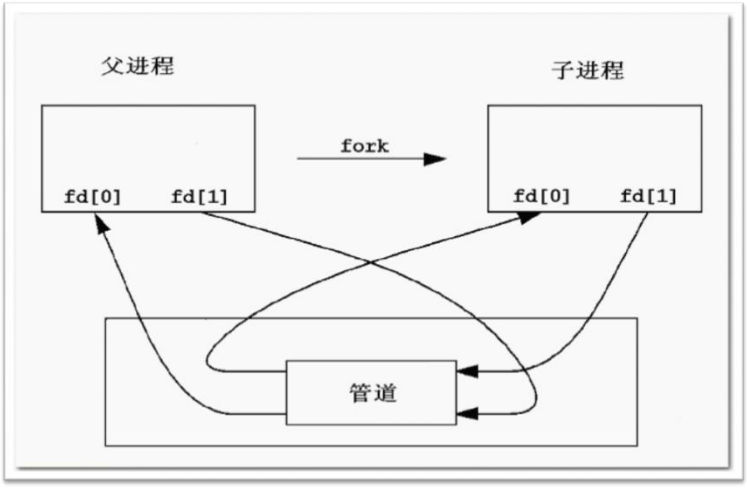
功能：经由参数 filedes 返回两个文件描述符
参数：
filedes 为 int 型数组的首地址，其存放了管道的文件描述符 fd[0]、fd[1]。
filedes[0]为读而打开，filedes[1]为写而打开管道，filedes[0]的输出是 filedes[1]的输入。
返回值：
成功：返回 0
失败：返回-1
例：01_pipe_1.c 创建管道，父子进程通过无名管道通信

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

int main(int argc, char *argv[])
{
    int fd_pipe[2];
    char buf[] = "hello world";
    pid_t pid;

    if (pipe(fd_pipe) < 0)
        perror("pipe");
    pid = fork();
    if (pid < 0)
    {
        perror("fork");
        exit(-1);
    }
    if (pid == 0)
    {
        write(fd_pipe[1], buf, strlen(buf));
        _exit(0);
    }
    else
    {
        wait(NULL);
        memset(buf, 0, sizeof(buf));
        read(fd_pipe[0], buf, sizeof(buf));
        printf("buf=[%s]\n", buf);
    }
    return 0;
}
```

父子进程通过管道实现数据的传输



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

从管道中读数据的特点

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。

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, F_SETFL, 0);

设置为非阻塞：

fcntl(fd, F_SETFL, O_NONBLOCK);

例：01_pipe_2.c　验证无名管道文件读写的阻塞和非阻塞

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

int main(int argc, char *argv[])
{
    int fd_pipe[2];
    char buf[] = "hello world";
    pid_t pid;
```

```
    if (pipe(fd_pipe) < 0)
        perror("pipe");
    pid = fork();
    if (pid < 0)
    {
        perror("fork");
        exit(-1);
    }
    if (pid == 0)
    {
        sleep(3);
        write(fd_pipe[1], buf, strlen(buf));
        _exit(0);
    }
    else
    {
        //fcntl(fd_pipe[0], F_SETFL, O_NONBLOCK);
        fcntl(fd_pipe[0], F_SETFL, 0);
        while(1)
        {
            memset(buf, 0, sizeof(buf));
            read(fd_pipe[0], buf, sizeof(buf));
            printf("buf=[%s]\n", buf);
            sleep(1);
        }
    }
    return 0;
}
```

5.3 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。

5.4 文件描述符的复制

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。

```
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。

回顾：ls > test.txt

5.4.1 dup 函数

```
#include <unistd.h>
int dup(int oldfd);
```

功能：

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。

参数：

要复制的文件描述符 oldfd。

返回值：

成功：新文件描述符。

失败：返回－1，错误代码存于 errno 中。

例：02_dup.c　重定向

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

int main(void)
{
    int fd1;
    int fd2;
```

```
    fd1 = open("test", O_CREAT|O_WRONLY, S_IRWXU);
    if (fd1 < 0)
    {
        perror("open");
        exit(-1);
    }
    close(1);
    fd2 = dup(fd1);
    printf("fd2=%d\n", fd2);
    return 0;
}
```

5.4.2 dup2 函数　重定向

```
#include <unistd.h>
int dup2(int oldfd, int newfd)
```

功能：

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。

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。

参数：

要复制的文件描述符 oldfd

分配的新的文件描述符 newfd

返回值：

成功：返回 newfd

失败：返回-1，错误代码存于 errno 中

例：03_dup2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd1;
    int fd2;
```

```
    fd2 = dup(1);//save stdout
    printf("new:fd2=%d\n",fd2);

    fd1 = open("test", O_CREAT|O_RDWR, S_IRWXU);
    close(1);
    dup(fd1);//1---> test
    printf("hello world\n");

    close(1);
    dup(fd2);//reset 1--->stdout
    printf("i love you \n");
    return 0;
}
```

复制文件描述符后新旧文件描述符的特点

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。

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。

当关闭旧的文件描述时，通过新的文件描述符仍可可操作文件。

exec 前后文件描述符的特点

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。

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。

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
```

练习：

题目:借用外部命令，实现计算器功能

提示：

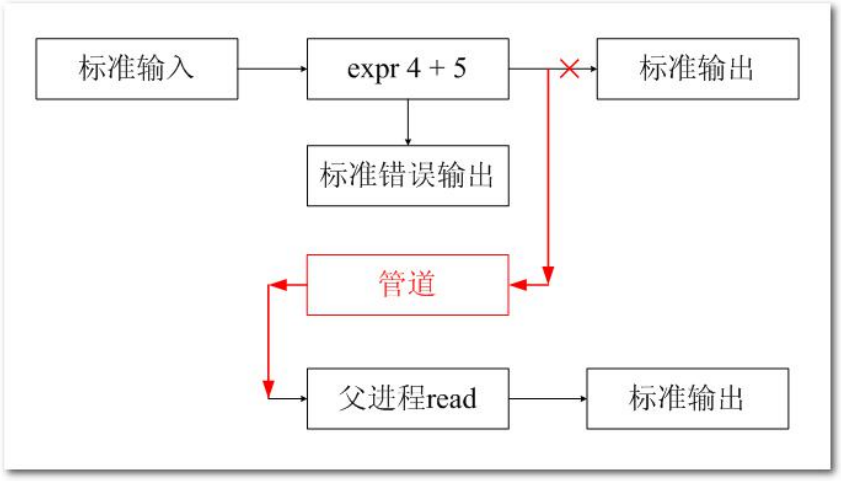
expr 是个外部命令，它向标准输出打印运算结果。

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

外部命令结果信息输出至管道：



5.5 命名管道(FIFO)

命名管道(FIFO)和管道(pipe)基本相同，但也有一些显著的不同，
其特点是:
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并且在文件系统中可见，所以有名管道可以实现不相关进程间通信，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。

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:
特点一：
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。

例:04_fifo_read_1.c　验证阻塞方式，open 的阻塞效果

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

```
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd;
    int ret;
    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    printf("before open\n");
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    return 0;
}
```

04_fifo_write_1.c　验证阻塞方式，open 的阻塞效果

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

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

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
```

```
    printf("before open\n");
    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    return 0;
}
```

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。

例:04_fifo_read_2.c 以阻塞模式，验证 read 函数也会阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    int ret;
    char recv[100];

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    printf("before open\n");
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    printf("before read\n");
    bzero(recv, sizeof(recv));
```

```
    read(fd, recv, sizeof(recv));
    printf("read from my_fifo buf=[%s]\n",recv);
    return 0;
}
```

例:04_fifo_write_2.c 以阻塞模式，验证 read 函数也会阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    printf("before open\n");
    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    printf("before write\n");
    sleep(5);
    write(fd, send, strlen(send));
    printf("write to my_fifo buf=%s\n",send);
    return 0;
}
```

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。

例:04_fifo_read_3.c　阻塞方式打开命名管道，验证 写进程退出，会导致 read 不阻塞

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

```
#include <fcntl.h>

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

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    while(1)
    {
        char recv[100];

        bzero(recv, sizeof(recv));
        read(fd, recv, sizeof(recv));
        printf("read from my_fifo buf=[%s]\n",recv);
        sleep(1);
    }
    return 0;
}
```

例：04_fifo_write_3.c 阻塞方式打开命名管道，验证 写进程退出，会导致 read 不阻塞

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

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

```
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    write(fd, send, strlen(send));
    printf("write to my_fifo buf=%s\n",send);
    while(1);
    return 0;
}
```

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。

例:04_fifo_read_4.c 阻塞方式打开命名管道，验证 读进程结束后，写进程再向管道写数据写进程会收到信号退出

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

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

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
```

```
        perror("open fifo");
    }
    while(1)
    {
        char recv[100];

        bzero(recv, sizeof(recv));
        read(fd, recv, sizeof(recv));
        printf("read from my_fifo buf=[%s]\n",recv);
        sleep(1);
    }
    return 0;
}
```

例：04_fifo_write_4.c 阻塞方式打开命名管道，验证 读进程结束后，写进程再向管道写数据写进程会收到信号退出

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    while(1)
    {
        write(fd, send, strlen(send));
        printf("write to my_fifo buf=%s\n",send);
        sleep(1);
    }
```

```
    return 0;
}
```

6、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。

特点二：

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。

例：04_fifo_read_5.c　非阻塞方式打开命名管道，验证 open 和 read 都不阻塞

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

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

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    fd = open("my_fifo", O_RDONLY|O_NONBLOCK);
    if(fd < 0)
    {
        perror("open fifo");
    }
```

```
    while(1)
    {
        char recv[100];

        bzero(recv, sizeof(recv));
        read(fd, recv, sizeof(recv));
        printf("read from my_fifo buf=[%s]\n",recv);
        sleep(1);
    }
    return 0;
}
```

例：04_fifo_write_5.c 非阻塞方式打开命名管道，验证 open 和 read 都不阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY|O_NONBLOCK);
    if(fd<0)
    {
        perror("open fifo");
    }
    write(fd, send, strlen(send));
    printf("write to my_fifo buf=%s\n",send);
    while(1);
    return 0;
}
```

注意：

open 函数以可读可写方式打开 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。