1. 无名管道（01 子目录）

- **写测试程序test1-1，再fork 子进程，父子进程间建立无名管道，父进程向子进程发送数据**

核心代码：

```
if((pid = fork()) < 0) {
    perror_exit(0, "create process failed");
} else if(pid > 0) {
    close(pfd[0]);
    write(pfd[1], writeBuff, writeLen);
    printf("parent process write: %s\n", writeBuff);
    close(pfd[1]);
    wait(&status);
} else {
    close(pfd[1]);
    sleep(1);
    read(pfd[0], readBuff, writeLen);
    printf("child process read: %s\n", readBuff);
    close(pfd[0]);
}
```

测试截图：

```
[root@RHEL74-SVR 01]# ./test1-1
parent process write: Hello child, I'm parent!
child process read: Hello child, I'm parent!
```

- **写测试程序test1-2，再fork 子进程，父子进程间建立无名管道，子进程向父进程发送数据**

核心代码：

```
if((pid = fork()) < 0) {
    perror_exit(0, "create process failed");
} else if(pid > 0) {
    close(pfd[1]);
    sleep(1);
    read(pfd[0], readBuff, writeLen);
    printf("parent process read: %s\n", readBuff);
    close(pfd[0]);
    wait(&status);
} else {
    close(pfd[0]);
    write(pfd[1], writeBuff, writeLen);
    printf("child process write: %s\n", writeBuff);
    close(pfd[1]);
}
```

测试截图：

```
[root@RHEL74-SVR 01]# ./test1-2
child process write: Hello parent, I'm child!
parent process read: Hello parent, I'm child!
```

- **写测试程序test1-3，再fork 子进程，父子进程间建立无名管道，能否双向传递数据？**

在历史上，管道是半双工的，现在某些系统提供全双工管道。当然，也可以通过建立两个无名管道进行数据传递。

在Linux RedHat7中通过一个无名管道不能能够双向传递数据的，测试核心代码如下：

```
if((pid = fork()) < 0) {
    perror_exit(0, "create process failed");
} else if(pid > 0) {
    read(pfd[0], readBuff, writeLen);
    printf("parent process read: %s\n", readBuff);
    strcpy(writeBuff, "new message data");
    write(pfd[1], writeBuff, writeLen);
    printf("parent process write: %s\n", writeBuff);
    close(pfd[0]);
    close(pfd[1]);
    wait(&status);
} else {
    write(pfd[1], writeBuff, writeLen);
    printf("child process write: %s\n", writeBuff);
    sleep(1);
    read(pfd[0], readBuff, writeLen);
    printf("child process read: %s\n", readBuff);
    close(pfd[0]);
    close(pfd[1]);
}
```

测试截图：

```
[root@RHEL74-SVR 01]# ./test1-3
child process write: child writed
parent process read: child writed
parent process write: parent writed
```

- **无名管道方式传递的数据的类型，长度是否有限制？**

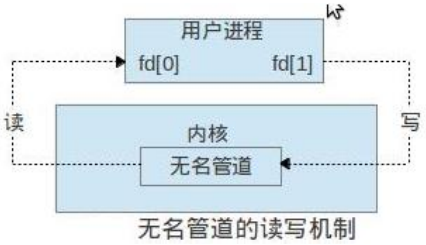
管道的缓冲区大小是受限制的，且数据以无格式字节流的方式传输，需要输入方与输出方约定好数据格式；管道是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为1页，即4K字节，使得它的大小不像文件那样不加检验地增长。

- **能否在独立进程间用无名管道通信？**

管道只能用于父子进程或者兄弟进程间通信，即只能用于有亲缘关系的进程间通信；

无名管道总结：

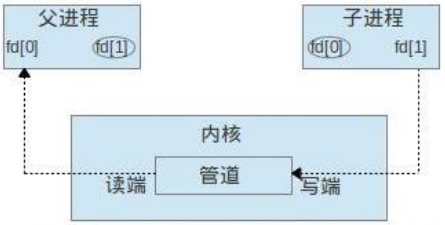
管道是基于文件描述符的通信方式，当一个管道建立时，它会创建两个文件描述符fd[0]和fd[1]，其中fd[0]固定用于读管道，而fd[1]固定用于写管道，如图所示，这样就构成了一个半双工的通道。



无名管道的读写机制

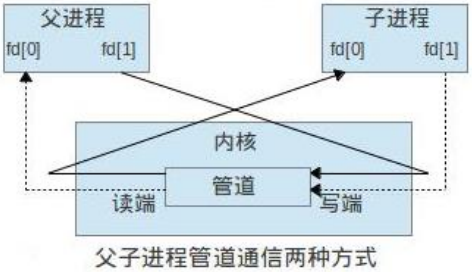
管道读写说明

用 pipe() 创建的管道两端处于同一个进程中，通常先是创建一个管道，再调用fork()函数创建一个子进程，该子进程会继承父进程所创建的管道。将父进程的写端fd[1]和子进程的读端fd[0]关闭，则父子进程之间就建立起一条“子进程写入父进程读取”的通道，父子进程管道的文件描述符对应关系如下图：



父子进程管道的文件描述符对应关系

同样，也可以将父进程的读端fd[0]和子进程的写端fd[1]关闭，则父子进程之间就建立起一条“父进程写入子进程读取”的通道。



父子进程管道通信两种方式

管道的特点：

1、管道通信是单向的，为半双工通信，因此数据只能由一个进程流向另一个进程，如果想要全双工通信，需要建立两个管道

2、写入的数据每次都添加到管道缓冲区的末尾，读数据的时候都是从缓冲区的头部读出数据。因此遵守先入先出的原则，即先写入的数据先读出。

3、管道的缓冲区大小是受限制的，且数据以无格式字节流的方式传输，需要输入方与输出方约定好数据格式

4、管道中的数据一旦被读取，将会从管道中消失。

5、管道只能用于父子进程或者兄弟进程间通信，即只能用于有亲缘关系的进程间通信。

2. 有名管道（02 子目录）

- **写测试程序test2-1，再fork 子进程，父子进程间建立有名管道，父进程向子进程发送数据**

核心代码：

```
unlink(FIFO_PATH_NAME);
ret=mkfifo(FIFO_PATH_NAME, 0777);
if(ret<0) {
    perror_exit(0, "mkfifo failed \n");
}

if((pid = fork()) < 0) {
    perror_exit(0, "create process failed \n");
} else if(pid > 0) {
    fd=open(FIFO_PATH_NAME, O_WRONLY);
    write(fd, writeBuff, writeLen);
    printf("parent process write: %s\n", writeBuff);
    wait(&status);
} else {
    fd=open(FIFO_PATH_NAME, O_RDONLY);
    read(fd, readBuff, writeLen);
    printf("child process read: %s\n", readBuff);
}
```

测试截图：

```
[root@RHEL74-SVR 02]# ./test2-1
parent process write: message data
child process read: message data
```

- **写测试程序test2-2，再fork 子进程，父子进程间建立有名管道，子进程向父进程发送数据**

核心代码：

```
unlink(FIFO_PATH_NAME);
ret=mkfifo(FIFO_PATH_NAME, 0777);
if(ret<0) {
    perror_exit(0, "mkfifo failed \n");
}

if((pid = fork()) < 0) {
    perror_exit(0, "create process failed\n");
} else if(pid > 0) {
    fd=open(FIFO_PATH_NAME, O_RDONLY);
    read(fd, readBuff, writeLen);
    printf("parent process read: %s\n", readBuff);
    wait(&status);
} else {
    fd=open(FIFO_PATH_NAME, O_WRONLY);
    write(fd, writeBuff, writeLen);
    printf("child process write: %s\n", writeBuff);
}
```

测试截图：

```
[root@RHEL74-SVR 02]# ./test2-2
child process write: message data
parent process read: message data
```

- **写测试程序test2-3，再fork 子进程，父子进程间建立有名管道，能否双向传递数据？**

有名管道也是单向传递数据，不能通过一个有名管道进行双向传递数据，若想要双向传递数据，需要建立两个有名管道。

测试中只有一个有名管道，测试核心代码：

```
unlink(FIFO_PATH_NAME);
ret=mkfifo(FIFO_PATH_NAME, 0777);
if(ret<0) {
    perror_exit(0, "mkfifo failed \n");
}

if((pid = fork()) < 0) {
    perror_exit(0, "create process failed\n");
} else if(pid > 0) {
    fd=open(FIFO_PATH_NAME, O_RDWR);
    read(fd, readBuff, writeLen);
    printf("parent process read: %s\n", readBuff);
    strcpy(writeBuff, "new message data");
    write(fd, writeBuff, strlen(writeBuff));
    printf("parent process write: %s\n", writeBuff);
    wait(&status);
} else {
    fd=open(FIFO_PATH_NAME, O_RDWR);
    write(fd, writeBuff, writeLen);
    printf("child process write: %s\n", writeBuff);
    sleep(1);
    read(fd, readBuff, INF);
    printf("child process read: %s\n", readBuff);
}
```

测试截图：

```
[root@RHEL74-SVR 02]# ./test2-3
child process write: message data
parent process read: message data
parent process write: new message data
child process read: new message data
```

- **一对测试程序test2-4-1/test2-4-2，两个进程间建立有名管道，test2-4-1 向test2-4-2 发送数据**

test2-4-1核心代码：

```
if(access(FIFO_PATH_NAME, F_OK) == -1) {
    res = mkfifo(FIFO_PATH_NAME, 0777);
    if(res != 0) {
        perror_exit(0, "mkfifo error \n");;
    }
}

fd=open(FIFO_PATH_NAME, O_WRONLY);
if(fd<0) {
    perror_exit(0, "open error errno:%d %s \n", errno, strerror(errno));
}

write(fd, writeBuff, writeLen);
printf("\n test2-4-1 write: %s\n", writeBuff);
close(fd);
```

test2-4-2核心代码：

```
fd=open(FIFO_PATH_NAME, O_RDONLY);
if(fd<0) {
    perror_exit(0, "open error errno:%d %s \n", errno, strerror(errno));
}

read(fd, readBuff, INF);
printf(" test2-4-2 read :%s\n", readBuff);

close(fd);
```

测试截图：

```
[root@RHEL74-SVR 02]# ./test2-4-2
[root@RHEL74-SVR 02]# ./test2-4-1
[root@RHEL74-SVR 02]#
 test2-4-1 write: my name is test2-4-1
 test2-4-2 read :my name is test2-4-1
```

- **一对测试程序test2-5-1/test2-5-2，两个进程间建立有名管道，双向传递数据，能否做到？**

不能。如果确实需要在程序之间双向传递数据，最好使用一对FIFO或管道，一个方向使用一个。

测试截图：

```
[root@RHEL74-SVR 02]# ./test2-5-1
[root@RHEL74-SVR 02]# ./test2-5-2
[root@RHEL74-SVR 02]#
 test2-5-1 write: test2-5-1 writed
 test2-5-2 read :test2-5-1 writed

 test2-5-2 write: test2-5-2 writed
 test2-5-1 read :
```

- **有名管道方式传递的数据的类型，长度是否有限制？和无名管道相比是否有区别？**

FIFO可以说是无名管道的推广，克服了无名管道无名字的限制，使得无亲缘关系的进程同样可以采用先进先出的通信机制进行通信。

管道和FIFO的数据是字节流，应用程序之间必须事先确定特定的传输"协议"，采用传播具有特定意义的消息。

系统对任意时刻在一个FIFO中可以存在的而数据长度是有限制的。它由#define PIPE_BUF 语句定义，通常可以在头文件limits.h 中找到他。在Linux和许多其他类UNIX 系统中，它的值通常是4096 字节，但在某些系统中他可能会小到512 字节。

3. 信号方式（03 子目录）

- **一对测试程序test3-1-1/test3-1-2，在test3-1-2 中约定截获几个信号用自定义函数处理，其中部分是截获后打印信息，程序继续；部分是截获后打印信息，程序退出。test3-1-1 中定时向test3-1-2 发送相应的信号；同时可以通过控制台手工向test3-1-2 发送相应的信号**

测试截图

```
[root@RHEL74-SVR 03]# ./test3-1-2
[root@RHEL74-SVR 03]# ./test3-1-1
[root@RHEL74-SVR 03]#
 test3-1-1 send signo:30 sival_int=1453381

 test3-1-2 recv signal, signo=30
         siginfo->si_signo :30
         siginfo->si_pid :8934
         siginfo->si_value :1453381

 test3-1-1 send signo:31 sival_int=1453381

 test3-1-2 recv signal, signo=31
         siginfo->si_signo :31
         siginfo->si_pid :8934
         siginfo->si_value :1453381
pidof test3-1-2 |xargs kill -16

 test3-1-2 recv signal, signo=16
         siginfo->si_signo :16
         siginfo->si_pid :8938
         siginfo->si_value :1453381
exit!
[root@RHEL74-SVR 03]#
```

- **信号能否带数据？**

能。

sigqueue()函数

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

int sigqueue(pid_t pid, int sig, const union sigval val)
```

调用成功返回 0；否则，返回 -1。

sigqueue的第一个参数是指定接收信号的进程ID；

第二个参数确定即将发送的信号；

第三个参数是一个联合数据结构union sigval，指定了信号传递的参数，即通常所说的4字节值。

```
typedef union sigval {
    int  sival_int;
    void *sival_ptr;
}sigval_t;
```

在调用sigqueue时，sigval_t指定的信息会拷贝到对应sig 注册的3参数信号处理函数的siginfo_t结构中，这样信号处理函数就可以处理这些信息了。由于sigqueue系统调用支持发送带参数信号，所以比kill()系统调用的功能要灵活和强大得多。

- **哪几个信号不能被截获并重定义？（kill -l 可查看系统支持的信号值）**

各种信号的默认处理情况

程序不可捕获、阻塞或忽略的信号有：SIGKILL,SIGSTOP

不能恢复至默认动作的信号有：SIGILL,SIGTRAP

默认会导致进程流产的信号有：SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGIOT、SIGQUIT、SIGSEGV、SIGTRAP、SIGXCPU、SIGXFSZ

默认会导致进程退出的信号有：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM

默认会导致进程停止的信号有：SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU

默认进程忽略的信号有：SIGCHLD、SIGPWR、SIGURG、SIGWINCH

- **一对测试程序test3-2-1/test3-2-2，要求3-2-1 写文件（此时3-2-2 为延时等待状态）后用自定义的特定信号通知3-2-2 去读，自己变为延时等待状态；3-2-2 收到此信号后去读文件的内容，读取完成后清空文件，写入新内容，再用自定义的特定信号通知3-2-1 去读，自己变为延时等待状态，循环往复（提示：用什么方法可以知道对方的pid 值并传递信号）**

用进程名获取其pid值的方法之一：

```
int executeShell(string cmd, string &ret) {
    FILE *file;
    char line[256];
    file = popen(cmd.c_str(), "r");
    if (NULL != file) {
        while (fgets(line, 256, file) != NULL) {
            ret+=string(line);
        }
    }
    pclose(file);
    return 0;
}

int getPidByName(string name) {
    string cmd="pidof "+name;
    string ret;
    executeShell(cmd, ret);
    return atoi(ret.c_str());
}
```

测试截图：

```
[root@RHEL74-SVR 03]# ./test3-2-2
[root@RHEL74-SVR 03]# ./test3-2-1
[root@RHEL74-SVR 03]#
 test3-2-2 recv signal, signo=30
         read :test3-2-1 writed, time :1511010066

 test3-2-1 recv signal, signo=30
         read :test3-2-2 writed, time :1511010066

 test3-2-2 recv signal, signo=30
         read :test3-2-1 writed, time :1511010070

 test3-2-1 recv signal, signo=30
         read :test3-2-2 writed, time :1511010070
```

4. 消息队列方式（04 子目录）

- **一对测试程序test4-1-1/test4-1-2，建立消息队列方式，然后从test4-1-1 向test4-1-2单向传递数据**

```
[root@RHEL74-SVR 04]# ./test4-1-2
[root@RHEL74-SVR 04]# ./test4-1-1
[root@RHEL74-SVR 04]#
test4-1-1 send:test4-1-1 sended
test4-1-2 recv:test4-1-1 sended
```

- **一对测试程序test4-2-1/test4-2-2，建立消息队列方式，然后双向传递数据，能否做到？**

```
[root@RHEL74-SVR 04]# ./test4-2-2
[root@RHEL74-SVR 04]# ./test4-2-1
[root@RHEL74-SVR 04]#
test4-2-1 send:test4-1-1 sended
test4-2-2 recv:test4-1-1 sended

test4-2-2 send:test4-2-2 sended
test4-2-1 recv:test4-2-2 sended
```

- **消息队列方式传递的数据的类型，长度是否有限制？ 和无名/有名管道相比是否有区别？**

消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。我们可以通过发送消息来避免命名管道的同步和阻塞问题。但是消息队列与命名管道一样，每个数据块都有一个最大长度的限制。

Linux用宏MSGMAX和MSGMNB来限制一条消息的最大长度和一个队列的最大长度。

msgque[MSGMNI]是一个msqid_ds结构的指针数组，每个msqid_ds结构指针代表一个系统消息队列，msgque[MSGMNI]的大小为MSGMNI=128，也就是说系统最多有MSGMNI=128个消息队列。

消息队列与命名管道的比较

消息队列跟命名管道有不少的相同之处，通过与命名管道一样，消息队列进行通信的进程可以是不相关的进程，同时它们都是通过发送和接收的方式来传递数据的。在命名管道中，发送数据用write，接收数据用read，则在消息队列中，发送数据用msgsnd，接收数据用msgrcv。而且它们对每个数据都有一个最大长度的限制。

与命名管道相比，消息队列的优势在于：1、消息队列也可以独立于发送和接收进程而存在，从而消除了在同步命名管道的打开和关闭时可能产生的困难。2、同时通过发送消息还可以避免命名管道的同步和阻塞问题，不需要由进程自己来提供同步方法。3、接收程序可以通过消息类型有选择地接收数据，而不是像命名管道中那样，只能默认地接收。

5. 共享内存方式（05 子目录）

- **一对测试程序test5-1/test5-2，要求共享一段内存，test5-1 向这段内存写入内容后，test5-2能读到写入的内容，反之亦然**

```
[root@RHEL74-SVR 05]# ./test5-2
[root@RHEL74-SVR 05]# ./test5-1
[root@RHEL74-SVR 05]# test 5-2 recv: test 5-1's data to test5-2
test 5-1 recv: test 5-2's data to test5-1
```

- **是否只能在父子进程间共享内存？还是可以独立进程间共享？**

共享内存不仅仅能在父子进程间共享，共享内存也可以在两个或多个进程间共享一个给定的内存区域，因为数据不需要在进程之间复制，相比于pipe、socket、file等共享通信方式，共享内存是最快的一种共享机制。

- **如果两个进程同时写，共享内存内容是否会乱？如何防止共享内存内容乱？**

若不采取相应措施，两个进程同时写，共享内存可能会乱。

如果在多个进程间共享一个资源，则可使用这 3 种技术中的一种来协调访问。我们可以使用映射到两个进程地址空间中的信号量、记录锁或者互斥量。对这 3 种技术两两之间在时间上的差别进行比较是有益的。

若使用信号量，则先创建一个包含一个成员的信号量集合，然后将该信号量值初始化为 1。为了分配资源，以 sem_op 为−1 调用 semop。为了释放资源，以 sem_op 为+1 调用 semop。对每个操作都指定 SEM_UNDO，以处理在未释放资源条件下进程终止的情况。

若使用记录锁，则先创建一个空文件，并且用该文件的第一个字节（无需存在）作为锁字节。[为了分配资源，先对该字节获得一个写锁。释放该资源时，则对该字节解锁。记录锁的性质确保了当一个锁的持有者进程终止时，内核会自动释放该锁。

若使用互斥量，需要所有的进程将相同的文件映射到它们的地址空间里，并且使用 PTHREAD_PROCESS_SHARED 互斥量属性在文件的相同偏移处初始化互斥量。为了分配资源，我们对互斥量加锁。为了释放锁，我们解锁互斥量。如果一个进程没有释放互斥量而终止，恢复将是非常困难的，除非我们使用鲁棒互斥量（回忆 12.4.1 节中讨论的 pthread_mutex_consistent 函数）。

6. Unix 套接字方式（06 子目录）

- **一对测试程序test6-1-1/test6-1-2，要求在两个进程间建立Unix 类型的Socket（类似TCP方式），然后通过socket 读写来实现进程的双向通信（通信内容自行定义即可）**

代码逻辑同TCP的Socket类似，使用UNIX Domain Socket的过程和网络socket十分相似，也要先调用socket()创建一个socket文件描述符，address family指定为AF_UNIX，type可以选择SOCK_DGRAM或SOCK_STREAM，protocol参数仍然指定为0即可。

不同的地方大致如下：

server：

```
listen_fd=socket(PF_UNIX,SOCK_STREAM,0);
```

```
//set server addr_param
srv_addr.sun_family=AF_UNIX;
strncpy(srv_addr.sun_path,UNIX_DOMAIN,sizeof(srv_addr.sun_path)-1);
unlink(UNIX_DOMAIN);
//bind sockfd & addr
ret=bind(listen_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
```

client:

```
connect_fd=socket(PF_UNIX,SOCK_STREAM,0);
if(connect_fd<0) {
    perror("cannot create communication socket");
    return 1;
}
srv_addr.sun_family=AF_UNIX;
strcpy(srv_addr.sun_path,UNIX_DOMAIN);

ret=connect(connect_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
if(ret==-1) {
    perror("cannot connect to the server");
    close(connect_fd);
    return 1;
}
```

测试截图：

```
[root@RHEL74-SVR 06]# ./test6-1-1
[root@RHEL74-SVR 06]# ./test6-1-2
[root@RHEL74-SVR 06]# test6-1-1 server read 20 bytes    :1111111111111111111
test6-1-1 write 20 bytes         :20个'1'
test6-1-2 write 20 bytes         :20个'1'
test6-1-2 read 20 bytes          :11111111111111111111
test6-1-2 write 20 bytes         :20个'2'
test6-1-1 server read 20 bytes   :22222222222222222222
test6-1-1 write 20 bytes         :20个'2'
test6-1-2 read 20 bytes          :22222222222222222222
test6-1-2 write 20 bytes         :20个'3'
test6-1-1 server read 20 bytes   :33333333333333333333
test6-1-1 write 20 bytes         :20个'3'
test6-1-2 read 20 bytes          :33333333333333333333
```

- **一对测试程序test6-2-1/test6-2-2，要求在两个进程间建立Unix 类型的Socket（类似UDP方式），然后通过socket 读写来实现进程的双向通信（通信内容自行定义即可）**

代码逻辑同UDP的Socket类似，不同的代码大致如下：

```
listen_fd=socket(AF_UNIX,SOCK_DGRAM,0);
if(listen_fd<0) {
    perror("cannot create communication socket \n");
    return 1;
}

int reuse=1;
setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));

srv_addr.sun_family=AF_UNIX;
strncpy(srv_addr.sun_path,UNIX_DOMAIN,sizeof(srv_addr.sun_path)-1);
unlink(UNIX_DOMAIN);

ret=bind(listen_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
```

client

```
connect_fd=socket(AF_UNIX,SOCK_DGRAM,0);
if(connect_fd<0) {
    perror_exit(0, "cannot create communication socket \n");
    return 1;
}

int reuse=1;
setsockopt(connect_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));

srv_addr.sun_family=AF_UNIX;
strcpy(srv_addr.sun_path,UNIX_MY_DOMAIN);
unlink(UNIX_MY_DOMAIN);

ret=bind(connect_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
```

```
struct sockaddr_un sendaddr;
int len=sizeof(sendaddr);
sendaddr.sun_family=AF_UNIX;
strcpy(sendaddr.sun_path,UNIX_DOMAIN);
ret=connect(connect_fd,(struct sockaddr*)&sendaddr,sizeof(sendaddr));
if(ret==-1) {
    perror_exit(0, "cannot connect to the server \n");
    close(connect_fd);
    return 1;
}
```

测试截图：

```
[root@RHEL74-SVR 06]# ./test6-2-1
[root@RHEL74-SVR 06]# ./test6-2-2
[root@RHEL74-SVR 06]# test6-2-2 client write 20 bytes    :20个'1'
test6-2-1 server read 20 bytes  :11111111111111111111
test6-2-1 server write 20 bytes          :20个'1'
test6-2-2 client read 20 bytes  :11111111111111111111
test6-2-2 client write 20 bytes          :20个'2'
test6-2-1 server read 20 bytes  :22222222222222222222
test6-2-1 server write 20 bytes          :20个'2'
test6-2-2 client read 20 bytes  :22222222222222222222
test6-2-2 client write 20 bytes          :20个'3'
test6-2-1 server read 20 bytes  :33333333333333333333
test6-2-1 server write 20 bytes          :20个'3'
test6-2-2 client read 20 bytes  :33333333333333333333
```

- **Unix 类型Socket 的建立、使用等与TCP/UDP Socket 相比有什么相同和不同点？**

UNIX Domain Socket是在socket架构上发展起来的用于同一台主机的进程间通讯（IPC），它不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。UNIX Domain Socket有SOCK_DGRAM或SOCK_STREAM两种工作模式，类似于UDP和TCP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。

二者编程的不同如下：

（1）address family为AF_UNIX

（2）因为应用于IPC，所以UNIXDomain socket不需要IP和端口，取而代之的是文件路径来表示“网络地址”。这点体现在两个方面：地址格式不同，UNIXDomain socket用结构体sockaddr_un表示，是一个socket类型的文件在文件系统中的路径；UNIX Domain Socket客户端一般要显式调用bind函数，而网络socket依赖系统自动分配的地址。

- **Unix 类型Socket 是否有阻塞和非阻塞方式？是否能通过select 来读写？写满后是返回不可写还是继续可写而导致数据丢失？在测试程序中表现出来**

Unix类型Socket也有阻塞和非阻塞方式，原理同网络通信socket相同。非阻塞能通过select来读写。

在使用类似TCP数据流传递数据方式时，写满后会不可写；

在使用类似TUDP数据报传递数据方式时，写满后会不可写；

两种方式都是可靠的，不会导致数据丢失。

7. 文件锁机制（07 子目录）

- **本小题要求文件的打开及读写方式为open/close/read/write，不准用fopen/fclose系列**
- **一对测试程序test7-1-1/test7-1-2，要求7-1-1 对某个特定文件新建/打开后加写锁，延时一段时间（此时启动7-1-2）后再向文件中写入一些内容，写入完成后释放锁，进入死循环等待状态；7-1-2 同样打开该文件后加读锁，此时会被阻塞，直到7-1-1 释放写锁后，7-1-2 才会退出阻塞状态，读取7-1-1 刚才写入的内容后进入死循环等待状态**

相关函数封装如下：

```
int lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len) {
    struct flock lock;

    lock.l_type = type;
    lock.l_start = offset;
    lock.l_whence = whence;
    lock.l_len = len;

    return (fcntl(fd, cmd, &lock));
}

int File_ReadLck(int fd, off_t offset, int whence, off_t len) {
    return lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len);
}

int File_WriteLck(int fd, off_t offset, int whence, off_t len) {
    return lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len);
}

int File_ReadLckW(int fd, off_t offset, int whence, off_t len) {
    return lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len);
}

int File_WriteLckW(int fd, off_t offset, int whence, off_t len) {
    return lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len);
}

int File_Unlock(int fd, off_t offset, int whence, off_t len) {
    return lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len);
}
```

test7-1-1核心代码如下：

```
fd = open("/tmp/testfilelock", O_RDWR|O_CREAT|O_TRUNC, 0666);

while(1){
    File_WriteLckW(fd, 0, SEEK_SET, 0);
    printf("test7-1-1 add write lockw success \n");
    sleep(5);
    write(fd, writeBuff, sizeof(writeBuff));
    File_Unlock(fd, 0, SEEK_SET, 0);
    printf("test7-1-1 unlock \n");
    sleep(5);
}
```

test7-1-1核心代码如下：

```
fd = open("/tmp/testfilelock", O_RDWR|O_CREAT|O_TRUNC, 0666);

while(1){
    File_ReadLckW(fd, 0, SEEK_SET, 0);
    printf("test7-1-2 add read lockw success \n");
    sleep(5);
    read(fd, readBuff, sizeof(readBuff));
    File_Unlock(fd, 0, SEEK_SET, 0);
    printf("test7-1-2 unlock \n");
    sleep(5);
}
```

在两个测试程序中，读写锁均是阻塞函数，即若一个程序加了写锁，另一个尝试加读锁，会在前者释放锁之前一直处于阻塞状态，直到前者释放了锁，反之亦然。

测试截图：

```
[root@RHEL74-SVR 07]# ./test7-1-1
[root@RHEL74-SVR 07]# test7-1-1 add write lockw success
./test7-1-2
[root@RHEL74-SVR 07]# test7-1-1 unlock
test7-1-2 add read lockw success
test7-1-2 unlock
test7-1-1 add write lockw success
test7-1-1 unlock
test7-1-2 add read lockw success
```

- **一对测试程序test7-2-1/test7-2-2，在程序中打开文件后将fd 设置为非阻塞方式，其余要求同test7-1-1/test7-1-2**

test7-2-1核心代码如下：

```
fd = open("/tmp/testfilelock", O_RDWR|O_CREAT|O_TRUNC, 0666);
setNonBlock(fd);

while(1) {
    sleep(3);
    res=File_WriteLck(fd, 0, SEEK_SET, 0);
    if(res<0){
        printf("test7-2-1 add write lock error errno: %d %s\n", errno, strerror(errno));
        continue;
    }
    else{
        printf("test7-2-1 add write lock success \n");
    }
    sleep(5);
    write(fd, writeBuff, sizeof(writeBuff));
    File_Unlock(fd, 0, SEEK_SET, 0);
    printf("test7-2-1 unlock \n");
}
```

test7-2-2核心代码如下：

```
fd = open("/tmp/testfilelock", O_RDWR|O_CREAT|O_TRUNC, 0666);
setNonBlock(fd);

while(1) {
    sleep(3);
    res=File_ReadLck(fd, 0, SEEK_SET, 0);
    if(res<0){
        printf("test7-2-2 add read lock error errno: %d %s \n", errno, strerror(errno));
        continue;
    }
    else{
        printf("test7-2-2 add read lock success \n");
    }
    sleep(5);
    read(fd, readBuff, sizeof(readBuff));
    File_Unlock(fd, 0, SEEK_SET, 0);
    printf("test7-2-2 unlock \n");
}
```

采用非阻塞加锁函数，会立即返回-1，

错误号errno: 11 Resource temporarilyunavailable，可以间隔检查是否可加锁，然后加锁。

测试截图：

```
[root@RHEL74-SVR 07]# ./test7-2-1
[root@RHEL74-SVR 07]# ./test7-2-2
[root@RHEL74-SVR 07]# test7-2-1 add write lock success
test7-2-2 add read lock error errno: 11 Resource temporarily unavailable
test7-2-2 add read lock error errno: 11 Resource temporarily unavailable
test7-2-1 unlock
test7-2-2 add read lock success
test7-2-1 add write lock error errno: 11 Resource temporarily unavailable
test7-2-1 add write lock error errno: 11 Resource temporarily unavailable
test7-2-2 unlock
test7-2-1 add write lock success
```

- **锁定文件有几种方法？不同的方法对阻塞/非阻塞方式的fd 是否有区别？**

可用fcntl、lockf、flock三个函数锁定文件。

flock和fcntl是系统调用，而lockf是库函数。lockf实际上是fcntl的封装，所以lockf和fcntl的底层实现是一样的，对文件加锁的效果也是一样的。

int flock(int fd, int operation); 只是建议性锁，其中fd是系统调用open返回的文件描述符，operation的选项有：

LOCK_SH ：共享锁
LOCK_EX ：排他锁或者独占锁
LOCK_UN ： 解锁。
LOCK_NB：非阻塞（与以上三种操作一起使用）
flock函数只能对整个文件上锁，而不能对文件的某一部分上锁，flock只能产生劝告性锁。

int lockf(int fd, int cmd, off_t len);
fd为通过open返回的打开文件描述符。cmd的取值为：
F_LOCK：给文件互斥加锁，若文件以被加锁，则会一直阻塞到锁被释放。
F_TLOCK：同F_LOCK，但若文件已被加锁，不会阻塞，而回返回错误。
F_ULOCK：解锁。
F_TEST：测试文件是否被上锁，若文件没被上锁则返回0，否则返回-1。
len：为从文件当前位置的起始要锁住的长度。
通过函数参数的功能，可以看出lockf只支持排他锁，不支持共享锁。

int fcntl(int fd, int cmd, ... /* arg */);
文件记录加锁相关的cmd 分三种：
F_SETLK：申请锁（读锁F_RDLCK，写锁F_WRLCK）或者释放所（F_UNLCK），但是如果kernel无法将锁授予本进程（被其他进程抢了先，占了锁），不傻等，返回error。
F_SETLKW：和F_SETLK几乎一样，唯一的区别，这厮是个死心眼的主儿，申请不到，就傻等。
F_GETLK：这个接口是获取锁的相关信息： 这个接口会修改我们传入的struct flock。
通过函数参数功能可以看出fcntl是功能最强大的，它既支持共享锁又支持排他锁，即可以锁住整个文件，又能只锁文件的某一部分。

- **在一个程序对文件加写锁后，另一个程序不加锁而直接读写（都不设置为非阻塞方式），是阻塞在read/write 上，还是read/write 直接返回失败？**
- **在一个程序对文件加写锁后，另一个程序不加锁而直接读写（都设置为非阻塞方式），是阻塞在read/write 上，还是read/write 直接返回失败？**

文件锁可以进行很多的分类，最常见的主要有读锁与写锁，前者也叫共享锁，后者也叫排斥锁，值得注意的是，多个读锁之间是不会相互干扰的，多个进程可以在同一时刻对同一个文件加读锁；但是，如果已经有一个进程对该文件加了写锁，那么其他进程则不能对该文件加读锁或者写锁，直到这个进程将写锁释放，因此可以总结为：对于同一个文件而言，它可以同时拥有多个读者，但是在某一时刻，他只能拥有一个写者。
根据加锁区域范围，可以分成整个文件锁与区域文件锁（记录锁），二者很好区分，前者可以锁定整个文件，而后者则可以锁定文件中的某一区域，甚至是某几个字节。
根据内核行为来分，文件锁可以分成劝告锁与强制锁两大类：
劝告锁：
劝告锁讲究的是一种协同工作，内核仅负责对文件加锁以及检查文件是否已经上锁等操作，而不亲自去参与文件锁的控制与协调，而这些都需要程序员首先要检查所要访问的文件

之前是否已经被其他进程加锁来实现并发控制，劝告锁不仅可以对文件的任一部分加锁，也可以对整个文件加锁。下面是加锁规则：

当前所加锁	共享锁	排他锁
无	允许加锁	允许加锁
共享锁	允许加锁	EAGAIN(Resource temporarily unavailable)
排他锁	EAGAIN(Resource temporarily unavailable)	EAGAIN(Resource temporarily unavailable)

强制锁：

强制锁则是内核强制使用的一种文件锁，每当有进程违反锁规则，内核将会进行阻止，具体的加锁规则如下：

（1）若一个文件已经加上共享锁，那么其他进程在对这个文件进行写操作时将会被内核阻止；

（2）若一个文件已经加上了排他锁，那么其他进程对这个文件的读取与写操作都将被阻止；

下表总结了进程试图访问已经加有强制锁的文件，进程行为如下：

当前锁类型	阻塞读	阻塞写	非阻塞读	非阻塞写
共享锁	正常读数据	阻塞	读取正常	EAGAIN
排他锁	阻塞	阻塞	EAGAIN	EAGAIN

若进程要访问文件的锁类型与要进行的操作存在冲突，那么若操作时在阻塞时进行，则进程将会阻塞；若操作时在非阻塞时进程，则进程将会立即返回EAGIN错误（表示资源临时不可达）。