进程间通信

1551265 张伯阳

1-1

```
[root@RHEL-zby test]# ./test1-1
[root@RHEL-zby test]# hello world!
```

1-2

```
[root@RHEL-zby test]# ./test1-2
[root@RHEL-zby test]# hello world1!
```

1-3

```
[root@RHEL-zby test]# ./test1-3
[root@RHEL-zby test]# hello world1!
hello world2!
```

无名管道方式传递的数据的类型，长度是否有限制？

管道传递的是更加本质的纯字节流。这个字节流可以表达任意内容，两方程序协调一致就好，管道不关心其具体内容。　当管道一端不断地读取数据，另一端却不输出数据。根据 linux 的实现机制当管道读满是输出端自动阻塞。所以这个管道是有大小的 PIPE_SIZE 是管道的最大值 一般为 64K

能否在独立进程间用无名管道通信？

无名管道通信的进程，它们的关系一定是父子进程的关系，所以不能使用无名管道在非父子独立进程间通信

2-1 2-2 2-3

```
[root@RHEL-zby test]# ./test2-1
hello world!
[root@RHEL-zby test]# ./test2-2
hello world!
[root@RHEL-zby test]# ./test2-3
parent:hello world!
child:hello world!
```

2-4

```
[root@RHEL-zby test]# ./test2-4-2
Process 118476 opening FIFO O_RDONLY
read:Hello world!
[root@RHEL-zby test]#
```

```
[root@RHEL-zby test]# ./test2-4-1
Process 118486 opening FIFO O_WRONLY
write:Hello world!
[root@RHEL-zby test]#
```

2-5

```
[root@RHEL-zby test]# ./test2-5-1
[root@RHEL-zby test]# ./test2-5-2
[root@RHEL-zby test]# write:Hello world!
read:Hello world!
write:Hello world!
read:Hello world!
```

能否双向传递数据？

命名管道可以在任意进程间通信,通信是双向的,任意一端都可读可写,但是在同一时间只能有一端读,一端写。一个管道不能同时双向读写，只能建立两个管道实现双向读写或者串行在一个管道轮流读写。（这里实现轮流读写）

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？

管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式 管道的缓冲区是有限的（管道制存在于内存中，在管道创建时，为缓冲区分配一个页面大小）当要写入的数据量大于 PIPE_BUF 时，linux 将不再保证写入的原子性。在写满所有 FIFO 空闲缓冲区后，写操作返回。当要写入的数据量不大于 PIPE_BUF 时，linux 将保证写入的原子性。

3-1

信号 SIGUSR1 可以让程序输出并继续运行

信号 SIGUSR2 可以让程序输出并退出

```
[root@RHEL-zby test]# ./test3-1-2
[root@RHEL-zby test]# ./test3-1-1
[root@RHEL-zby test]#
OUCH1! - I got signal 10

OUCH2! - I got signal 12
```

控制台也可以输入 SIGINT 信号 使程序输出并退出

```
[root@RHEL-zby test]# ./test3-1-2
[root@RHEL-zby test]# ./test3-1-1
[root@RHEL-zby test]#
OUCH1! - I got signal 10
kill -2 21917

OUCH3! - I got signal 2
[root@RHEL-zby test]#
```

信号可以带数据

信号 SIGKILL 和 SIGSTOP 既不能被捕捉，也不能被忽略

3-2

```
[root@RHEL-zby test]# ./test3-2-1
[root@RHEL-zby test]# ./test3-2-2
[root@RHEL-zby test]# read:hello world!
write:hello world!
read:hello world!
write:hello world!
read:hello world!
write:hello world!
```

4-1

```c
void ouch(int sig)
{
    if(msgctl(msgid, IPC_RMID, 0) == -1)
    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(0);
}
```

```
[root@RHEL-zby test]# ./test4-1-1
send:hello world!
send:hello world!
send:hello world!
```

```
[root@RHEL-zby test]# ./test4-1-2
```

```
[root@RHEL-zby test]#
```

```
[root@RHEL-zby test]# ./test4-1-2
```

```
[root@RHEL-zby test]# ./test4-1-1
send:hello world!
send:hello world!
send:hello world!
```

```
[root@RHEL-zby test]# ./test4-1-2
recv:hello world!
recv:hello world!
recv:hello world!
```

在 4-1-2 中加入捕获退出信号删除消息队列的代码段

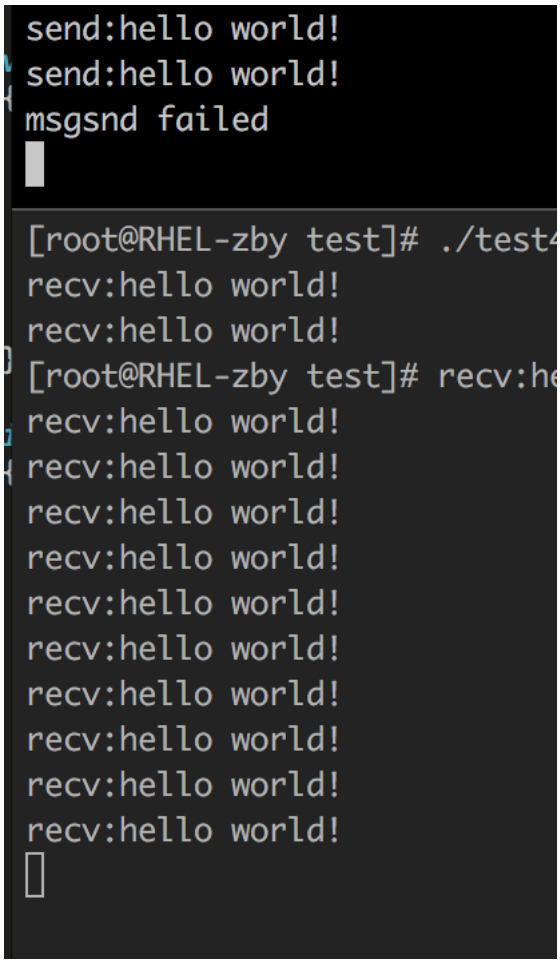
```
void ouch(int sig)
{
    if(msgctl(msgid, IPC_RMID, 0) == -1)
    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(0);
}
```

杀死 4-1-1 4-1-2 不再能继续接收到

```
[root@RHEL-zby ~]# ps -ef|grep test
root      23443      1  0 10:53 pts/0    00:00:00 ./test4-1-1
root      23446      1  0 10:53 pts/2    00:00:00 ./test4-1-2
root      23450   2453  0 10:53 pts/1    00:00:00 grep --color=auto test
[root@RHEL-zby ~]# kill -9 23443
[root@RHEL-zby ~]#
```

退出 4-1-2　4-1-1 检测到消息队列被删除 退出

```
[root@RHEL-zby ~]# ps -ef|grep test
root      24229      1  0 11:07 pts/0    00:00:00 ./test4-1-1
root      24232      1  0 11:07 pts/2    00:00:00 ./test4-1-2
root      24236   2453  0 11:07 pts/1    00:00:00 grep --color=auto test
[root@RHEL-zby ~]# kill -2 24232
[root@RHEL-zby ~]#
```


send:hello world!
send:hello world!
msgsnd failed
[root@RHEL-zby test]# ./test4
recv:hello world!
recv:hello world!
[root@RHEL-zby test]# recv:he
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!
recv:hello world!

4-2

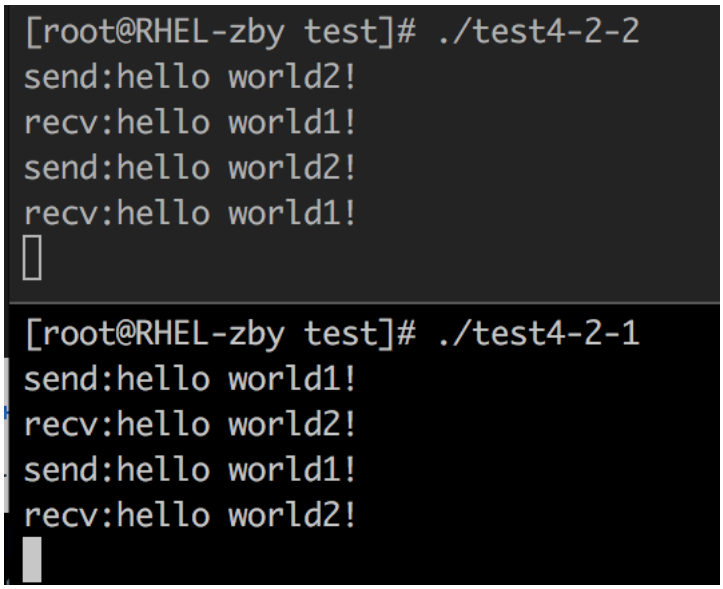
建立两个消息队列就能够做到

```
//建立消息队列
msgid1 = msgget((key_t)4321, 0777 | IPC_CREAT);
if(msgid1 == -1)
{
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}
msgid2 = msgget((key_t)1234, 0666 | IPC_CREAT);
if(msgid2 == -1)
{
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}
```

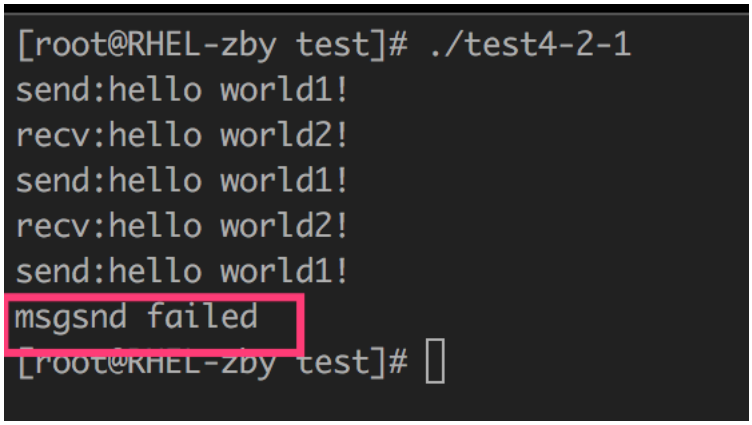
其他修改办法和前一题一样

```
void ouch(int sig)
{
    if(msgctl(msgid1, IPC_RMID, 0) == -1)
    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    if(msgctl(msgid2, IPC_RMID, 0) == -1)
    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }

    exit(0);
}
```



同样的方法 退出 4-2-2 后 删除消息队列 4-2-1 找不到消息队列并退出



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？

消息数据的类型一般为 char，自己构建时也可以存放任意形式的任意数据

消息的大小存在一个内部的限制。在 Linux 中，它在 Linux/msg.h 中的定义如下：#define MSGMAX 8192 消息的总大小不能超过 8192 个字节，这其中包括了 mtype 成员，它的长度为 4 个字节（long 类型）

消息队列和管道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

由于要做到两边都能够收和发

还不能使读写同时进行 设置写入锁 一端写入时另一端无法读取

这道题设置了两个文本储存区 使两端写入的内容不会冲突和覆盖

```
struct shared_use_st
{
    int written1;//作为一个标志，非0：表示可读，0表示可写
    int written2;
    char text1[text_SZ];//记录写入和读取的文本
    char text2[text_SZ];
};
```

```
[root@RHEL-zby test]# ./test5-1
[root@RHEL-zby test]# ./test5-2
send:hello world1!
recv:hello world2!
[root@RHEL-zby test]# send:hello world2!
recv:hello world1!
send:hello world1!
recv:hello world2!
send:hello world2!
recv:hello world1!
send:hello world1!
recv:hello world2!
```

可以在独立进程间共享

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？

同时写共享内存会混乱。共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。

我们通常需要用其他的机制来同步对共享内存的访问，例如信号，信号量等。在代码中实现了简单化的信号来控制同步读写。严格实现可以每次进入读写时发送一个进入读写信号给另一个进程，读写退出时再发送一个退出读写信号给另一个进程，另一个进程可以根据这些信号做出读写的控制，防止共享内存内容混乱。

6-1

```
[root@RHEL-zby test]# ./test6-1-1
[root@RHEL-zby test]# ./test6-1-2
[root@RHEL-zby test]# 连接成功!
server_send:hello client!
连接成功
client_send:hello server!
client_recv:hello client!
server_recv:hello server!
server_send:hello client!
client_send:hello server!
client_recv:hello client!
server_recv:hello server!
server_send:hello client!
client_send:hello server!
client_recv:hello client!
server_recv:hello server!
```

6-2

```
[root@RHEL-zby test]# ./test6-2-1
[root@RHEL-zby test]# server_send:hello client!
./test6-2-2
client_send:hello server!
server_recv:hello server!
[root@RHEL-zby test]# server_send:hello client!
client_recv:hello client!
client_send:hello server!
server_recv:hello server!
server_send:hello client!
client_recv:hello client!
client_send:hello server!
server_recv:hello server!
server_send:hello client!
client_recv:hello client!
```

相同:几乎都是相同的，有两种类型的套接字，字节流套接字和数据报套接字，字节流套接字类似于 TCP，数据报套接字类似于 UDP。

不同:tcp/udp 是在两台主机之间进行通信,其 ip 地址不同

进程间通信是在一台主机上两进程之间的通信 client 连接本机的 ip 地址

Unix 域套接字与 TCP 套接字相比较，在同一台主机的传输速度前者是后者的两倍

Unix 域套接字可以在同一台主机上的各进程之间传递描述符

Unix 域套接字与传统套接字的区别是用路径名来表示协议族的描述

Unix 域套接字的地址结构不同，在 connect()函数发现监听套接字队列满时，会立刻返回错误与 TCP 不同

与 UDP 套接字不同的是，未绑定的 Unix 域套接字上发送数据报不会给它捆绑一个路径名

有阻塞和非阻塞

```
int flags = fcntl(s, F_GETFL, 0);                    //获取文件的flags1值。
fcntl(s, F_SETFL, flags | O_NONBLOCK);    //设置成非阻塞模式;
```

可以用 select

```
FD_SET(s, &fdR);
select(s + 1, &fdR, NULL,NULL, NULL);//非阻塞recv
recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *) &si
```

写满后不可写 write 写入错误

7-1

```
[root@RHEL-zby test]# write:11
Release lock by 48401
Read lock set by 48406
read:11
process 1st
Release lock by 48406
```

7-2

```
[root@RHEL-zby test]# ./test7-2-1
Write lock set by 49087
[root@RHEL-zby test]# ./test7-2-2
Write lock already set by 49087
[root@RHEL-zby test]# write:11
Release lock by 49087
Read lock set by 49093
read:11
process 1st
Release lock by 49093
```

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？
flock 主要三种操作类型： LOCK_SH，共享锁，多个进程可以使用同一把锁，常被用作读共享锁； LOCK_EX，排他锁，同时只允许一个进程使用，常被用作写锁； LOCK_UN，释放锁；进程使用 flock 尝试锁文件时，如果文件已经被其他进程锁住，进程会被阻塞直到锁被释放掉，或者在调用 flock 的时候，采用 LOCK_NB 参数，在尝试锁住该文件的时候，发现已经被其他服务锁住，会返回错误，errno 错误码为 EWOULDBLOCK。即提供两种工作模式：阻塞与非阻塞类型。

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？
另一个进程不阻塞，read 直接正常返回空内容
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？
非阻塞情况也相同，read 直接正常返回空内容