

同济大学计算机系

计算机网络课程实验报告



学 号 1552239

姓 名 岳昊玮

专 业 计算机科学与技术

授课老师 沈坚

1. 补充知识

0.1 将 RHEL7 虚拟机克隆一个/多个，新的虚拟机如何设置网卡并使生效？

使用连接克隆，

临时性增加 ip 的两种方式，

```
[root@Anokoro ~]# ip addr add 192.168.80.220/24 dev ens32
[root@Anokoro ~]#
[root@Anokoro ~]#
[root@Anokoro ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:d4:da:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.230/24 brd 192.168.70.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 192.168.80.220/24 scope global ens32
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fed4:da24/64 scope link
        valid_lft forever preferred_lft forever

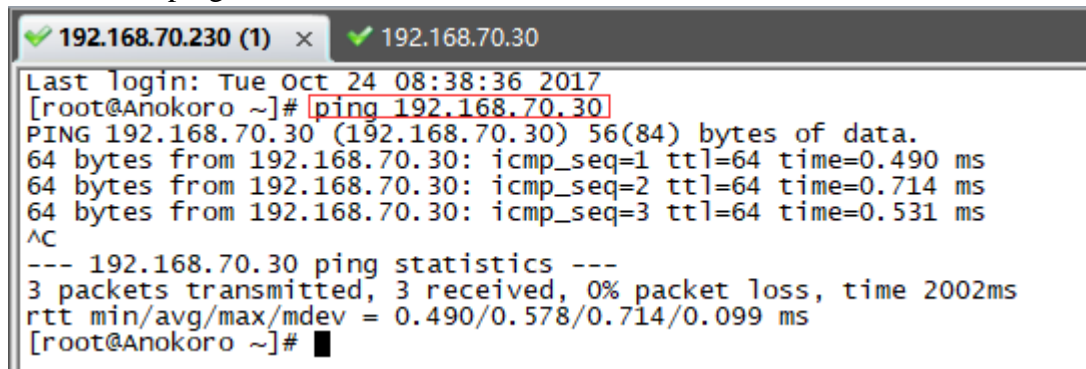
[root@Anokoro ~]# ifconfig ens32:2 192.168.90.22 netmask 255.255.255.0
[root@Anokoro ~]#
[root@Anokoro ~]#
[root@Anokoro ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:d4:da:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.230/24 brd 192.168.70.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 192.168.80.220/24 scope global ens32
        valid_lft forever preferred_lft forever
    inet 192.168.90.22/24 brd 192.168.90.255 scope global ens32:2
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fed4:da24/64 scope link
        valid_lft forever preferred_lft forever
[root@Anokoro ~]# █
```

永久性增加 IP

在 ifcfg-ens32 文件里面添加如下语句

```
ONBOOT=yes
IPADDR0=192.168.70.230
IPADDR1=192.168.70.80/26
IPADDR2=192.168.80.280/26
IPADDR3=192.168.90.300/27
PREFIX0=24
TYPE=BRIDGE
```

同网段 ping 通



```
192.168.70.230 (1) x 192.168.70.30
Last login: Tue Oct 24 08:38:36 2017
[root@Anokoro ~]# ping 192.168.70.30
PING 192.168.70.30 (192.168.70.30) 56(84) bytes of data.
64 bytes from 192.168.70.30: icmp_seq=1 ttl=64 time=0.490 ms
64 bytes from 192.168.70.30: icmp_seq=2 ttl=64 time=0.714 ms
64 bytes from 192.168.70.30: icmp_seq=3 ttl=64 time=0.531 ms
^C
--- 192.168.70.30 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.490/0.578/0.714/0.099 ms
[root@Anokoro ~]# █
```

```
[root@Anokoro 03]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    link/ether 00:50:56:31:82:9e brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.30/24 brd 192.168.70.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 172.18.12.230/26 brd 172.18.12.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 202.96.32.120/26 brd 202.96.32.127 scope global ens32
        valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe31:829e/64 scope link
        valid_lft forever preferred_lft forever
```

服务器端

```
[root@Anokoro 03]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    link/ether 00:0c:29:c7:51:da brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.230/24 brd 192.168.70.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 172.18.12.200/24 brd 172.18.12.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet 192.168.70.222/24 brd 192.168.70.255 scope global ens32
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fec7:51da/64 scope link
        valid_lft forever preferred_lft forever
```

客户端

增加新的网卡，直接把 ifcfg-ens32 拷贝一份放到相同目录下，命令 ifcfg-ens32:1，对文件内的 ip 等进行修改即可

2. （01 子目录）写一对 TCP Socket 的测试程序，分为 client 和 server，分别运行在不同虚拟机上

服务器端先初始化 Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用 accept 阻塞，等待客户端连接。在这时如果有个客户端初始化一个 Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

如果服务端绑定的端口号已被使用，（比如两次运行 ./tcp_server1 4000 或 ./tcp_server180），则无法进入 LISTEN 状态，会在哪个函数上出错？

会在端口绑定函数 bind()上出错，如下图所示

```

[root@Anokoro 01]# ./tcp_server1 4000 &
[1] 2649
[root@Anokoro 01]# =====waiting for client's request=====
[root@Anokoro 01]#
[root@Anokoro 01]# ./tcp_server1 4000 &
[2] 2650
[root@Anokoro 01]# bind socket error: Address already in use (error 98)
[2]+ 完成
[root@Anokoro 01]# ./tcp_server1 4000
[root@Anokoro 01]#

```

第一次运行，绑定4000端口，放在后台
等待客户端连接状态
再次绑定4000端口
提示端口已经被使用

查看端口的占用情况，`netstat -tunlp | grep 4000`

```

[root@Anokoro 01]# netstat -tunlp | grep 4000
tcp        0      0 0.0.0.0:4000        0.0.0.0:*          LISTEN      2649/./tcp_server1

```

测试程序 `tcp_client1`，运行时带入服务端 IP 地址及端口号，即可向服务端发起连接，要求 IP 地址、端口号通过 `main` 函数带参数的方式传入

这步就算是一个正确的演示，具体演示和后面重叠，在后面详细描述

如果 `client` 端连接时的 IP 地址不正确（例如不存在的 IP 地址），会在哪一步出错？如果连接的端口号不正确，会在哪一步出错？

如果 `client` 端连接时的 IP 地址不正确，如下图所示，（正确的 ip 为 192.168.70.30），则会提示 “Network is unreachable”

```

[root@Anokoro 01]# ./tcp_client1 192.168.80.30 4000
connect error: Network is unreachable(errno: 101)

```

如果 ip 正确而端口不正确，如下图所示（正确的端口为 4000）会提示 “Connection refused”

```

[root@Anokoro 01]# ./tcp_client1 192.168.70.30 400
connect error: Connection refused(errno: 111)

```

连接成功后，双方给出相应的提示信息，双方均进入 `read(recv)` 状态，此时 `read/recv` 函数会阻塞

Server 运行后进入等待连接状态

```

[root@Anokoro 01]#
[root@Anokoro 01]# ./tcp_server1 4000 &
[1] 3171
[root@Anokoro 01]# =====waiting for client's request=====
[root@Anokoro 01]#
[root@Anokoro 01]#

```

启动 `client` 之后，

Server 进入 `ecv` 状态

```

[root@Anokoro 01]# =====waiting for client's request=====
[root@Anokoro 01]#
[root@Anokoro 01]# get connection from one client, reading...

```

Client 阻塞在信息输入状态

```

[root@Anokoro 01]# ./tcp_client1 192.168.70.30 4000
connect successfully...
send msg to server:

```

然后 `client` 端发送一个数据

Client 端的情况，收到回复之后即断开

```

[root@Anokoro 01]# ./tcp_client1 192.168.70.30 4000
connect successfully...
send msg to server: 你好啊，服务器！
send message successfully
Received : Wow, connect successfully

```

输入的要发送的数据
 信息发送成功
 收到服务器的连接
 成功的回复

服务器端的情况

```

[root@Anokoro 01]# ./tcp_server1 4000
recv message from client: 你好啊，服务器！
a message be sent back to client

```

这是从客户端发来的消息
 服务器发送一个回执消息到客户机，告诉它连接成功了

连接成功后，用 CTRL+C 中断 client (server) 端，Server (client) 端能否检测到连接已中断？

如果 client 中断，server 可以立刻检测到。这个时候服务端 recv() 函数读取的数据长度为 0，以此来判断客户端已经退出，虽然有可能客户端真的就只是发了一个空的消息，但是这里做的时候简化为只有一次发送消息的机会，所以空的消息发完之后也会退出，因此，通过空的消息来判断客户端已经退出。

```

connect successfully...
send msg to server:
^C
[root@Anokoro 01]#

```

这时 server 没有收到任何信息，会自动判断客户端已经退出。

```

[root@Anokoro 01]# ./tcp_server1 4000
=====waiting for client's request=====

get connection from one client, reading...
Client exited! Connection break!

```

如果 server 中断，client 似乎是不能立刻检测到的，当它发送一个消息之后，发现收到的服务器的回复为空，以此来判断服务器已经断开了连接。

```

[root@Anokoro 01]# ./tcp_server1 4000
=====waiting for client's request=====
get connection from one client, reading...
^C
server exited !, connect break!
[root@Anokoro 01]# ./tcp_client1 192.168.70.30 4000
connect successfully...
send msg to server: fda
send message successfully
server exited !, connect break!

```

连接成功后，用 kill -9 杀死 client (server) 端，Server (client) 端能否检测到连接已中断？（另外启动一个 SecureCRT 的会话来做 kill）

Client 被 kill 掉，server 是可以检测到的。


```

[root@Anokoro 01]# ps
  PID TTY          TIME CMD
 3632 pts/0    00:00:00 bash
 5696 pts/0    00:00:00 tcp_client1
 5698 pts/0    00:00:00 ps
[root@Anokoro 01]# kill -9 5696
[root@Anokoro 01]# ps
  PID TTY          TIME CMD
 3632 pts/0    00:00:00 bash
 5699 pts/0    00:00:00 ps
[1]+  已杀死                  ./tcp_client1 192.168.70.30 4000

[root@Anokoro 01]# ./tcp_server1 4000
=====waiting for client's request=====
get connection from one client, reading...

client exited! connection break!

```

同样的，server 被 kill 掉之后，client 也是可以侦测到的，但是和上面一样，不是立刻侦测到，也是等到发送一个消息之后，收到了一个空的回复。

```

[root@Anokoro 01]# ./tcp_client1 192.168.70.30 4000
connect successfully...
send msg to server:
fsadfda
send message successfully
server exited !, connect break!

```

在双方连接成功后，再新的会话中再启动一个 tcp_client1 连接 server，会出现什么情况？

Server 会再 fork 出一个子进程去为新的 client 服务，因为每过来一个 client 的请求，server 都会分裂出一个子进程去和它进行信息交互，直到某一方主动中断连接。

tcp_server1 运行终止后，立即再次启动，绑定相同端口号，能否成功？（REUSEADDR 选项的作用，加或不加的区别是什么？）

```

[root@Anokoro 01]# kill 3171
[root@Anokoro 01]#
[1]+  已终止                  ./tcp_server1 4000
[root@Anokoro 01]#
[root@Anokoro 01]#
[root@Anokoro 01]# ./tcp_server1 4000
bind socket error: Address already in use (error 98)

```

如图所示，tcp_server1 运行中止之后，立即再次启动，绑定相同的端口号是会失败的，原因也是“Address already in use”。

一般来说，一个端口释放后会等待两分钟之后才能再被使用，SO_REUSEADDR 是让端口释放后立即就可以被再次使用。

3. （02 子目录）写一对 TCP Socket 的测试程序，分为 client 和 server，分别运行在不同虚拟机上

测试程序 tcp_server2，与 tcp_server1 功能相同，但接受连接时打印 client 端的 IP 地址和端口号

可以看到 IP 都是客户端主机的 IP，端口号是从一个较大的端口开始，间隔一个进行分配。

```

[root@Anokoro 01]# ./tcp_server1 4000
=====waiting for client's request=====
get connection from one client IP: 192.168.70.230 Port : 56916 reading...
recv message from client:

a message be sent back to client
get connection from one client IP: 192.168.70.230 Port : 56918 reading...
recv message from client: fasd

a message be sent back to client
get connection from one client IP: 192.168.70.230 Port : 56920 reading...
recv message from client: fdafas

a message be sent back to client
get connection from one client IP: 192.168.70.230 Port : 56922 reading...
recv message from client: fadsf

```

测试程序 `tcp_client2`，要求连接 `server` 端的时候使用固定端口号，通过 `main` 函数带参数的方式传入（例：`./tcp_client2 12345 192.168.80.230 4000` 则表示 `client` 的 12345 端口连接 `server` 的 4000 端口）

`Client1.cpp` 在编客户端的时候没有进行 `bind()`，实际上是可以 `bind` 的，不过不 `bind` 后就会系统自动分配端口，增加代码如下

```

struct sockaddr_in clientaddr;

memset(&clientaddr,0,sizeof(clientaddr));
clientaddr.sin_family = AF_INET;
clientaddr.sin_port = htons(atoi(argv[1]));
if(bind(sockfd,(struct sockaddr *)&clientaddr,sizeof(clientaddr)) == -1){
    printf("bind socket error: %s (error %d)\n",strerror(errno),errno);
    exit(0);
}

```

执行结果如下

```

[root@Anokoro 02]#
[root@Anokoro 02]# ./tcp_client2 12345 192.168.70.30 4000
connect successfully...
send msg to server:
fadf
send message successfully
Received : wow, connect successfully
[root@Anokoro 02]#

```

服务器端

```

[root@Anokoro 02]# ./tcp_server2 4000
=====waiting for client's request=====
get connection from one client , IP: 192.168.70.230, Port :12345 reading...
recv message from client: fadf

a message be sent back to client

```

- （03 子目录）写一对 TCP Socket 的测试程序，分为 `client` 和 `server`，分别运行在不同虚拟机上

两台用于测试的 `RHEL7` 虚拟机均设置多个地址（例如：`192.168.80.230/172.18.12.230`）

具体的设置已经在刚开始补充知识部分完成了，不再赘述。

测试程序 `tcp_server3`，能读到本机所有网卡的所有 IP 地址，然后只绑定其中的某个 IP 地址的某个端口，要求 IP 地址和端口号通过 `main` 函数带参数的方式传入（例：`./tcp_sevrer3 172.18.12.230 4000` 表示只绑定 `172.18.12.230` 的

4000 端口)

代码修改如下

```
servaddr.sin_addr.s_addr=inet_addr(argv[1]);
```

测试程序 `tcp_client3`，运行时带入服务端 IP 地址及其中任意一个端口号，即可向服务端发起连接，要求 IP 地址、端口号通过 `main` 函数带参数的方式传入（例：`./tcp_client3 172.18.12.230 4000` 则表示连接 172.18.12.230 的 TCP 4000 端口）

```
[root@Anokoro 03]# ./tcp_server3 172.18.12.230 4000
lo IP Address:127.0.0.1
ens32 IP Address:192.168.70.30
ens32 IP Address:172.18.12.230
ens32 IP Address:202.96.32.120
=====waiting for client's request=====
get connection from one client , IP: 172.18.12.200, Port :38730 reading...
```

```
[root@Anokoro 03]# ./tcp_client3 172.18.12.230 4000
connect successfully...
send msg to server:
```

如果 `tcp_client3` 连接未绑定的 IP 地址（例如：192.168.70.230），会怎样？

服务端

```
[root@Anokoro 03]# ./tcp_server3 172.18.12.230 4000
lo IP Address:127.0.0.1
ens32 IP Address:192.168.70.30
ens32 IP Address:172.18.12.230
ens32 IP Address:202.96.32.120
=====waiting for client's request=====
```

← 绑定的IP

客户端 会出现“拒绝连接错误”

```
[root@Anokoro 03]# ./tcp_client3 192.168.70.30 4000
connect error: Connection refused(errno: 111)
```

5. （04 子目录）写一对 TCP Socket 的测试程序，分为 client 和 server，分别运行在不同虚拟机上

测试程序 `tcp_server4-1`，接受 client 的连接成功后，用 `read` 函数一次读 20 字节（此时应进入阻塞状态，即 `read` 函数执行后，不读满 20 字节一直不返回，如何做到？注：不允许采用自己写循环保证读满 20 字节）

Read 是做不到的....，和下一问一起回答

测试程序 `tcp_client4-1`，连接服务端成功后，用 `write` 函数向服务端写入 20 字节，要求每次写两字节，然后延时 1 秒，再写 2 字节...，观察 server 端的 `read` 函数何时返回并执行后续语句，打印 `read` 函数读到的内容，是否与 client 发送的内容相同？

4-1-1 一次发送超过 20 字节


```
[root@Anokoro 04]# ./tcp_server4-1 4000
=====waiting for client's request=====
get connection from one client , reading...
waiting for 20 bytes msg...
recv message from client: 一 二 三 四 五 六 七 八 九 十
a message be sent back to client
```

收到的内容

```
[root@Anokoro 04]# ./tcp_client4-1-2 192.168.70.30 4000
connect successfully...
write msg to server:
一 二 三 四 五 六 七 八 九 十 九 八 七 六 五 四 三 二 一
2 bytes have been written...
4 bytes have been written...
[root@Anokoro 04]#
```

发送的内容

4-1-2 每次发送 2 个字节

```
get connection from one client , reading...
waiting for 20 bytes msg...
recv message from client: 一
a message be sent back to client
```

收到的内容

```
[root@Anokoro 04]# ./tcp_client4-1-1 192.168.70.30 4000
connect successfully...
write msg to server:
一 二 三 四 五 六 七 八 九 十 九 八 七 六 五 四 三 二 一
Received : server : hello!
[root@Anokoro 04]#
```

发送的内容

Read 在收到客户端发来的第一个 2bytes 数据之后，立即返回，所以只收到两个字节就断开了连接，因此客户端发送了第二次之后，服务已经断开了，所以程序也就中止了。

测试程序 tcp_server4-2/tcp_client_4-2，将 read/write 换成 recv/send 函数，用法是否相同？结果是否相同？

用法和结果都不同，(不过当第 4 个参数 flags 的值设置为 0 的时候，就和 read/write 没有区别了)。

在使用 recv 的时候，比 read 多了一个参数 int flags，将其设置为

MSG_WAITALL：通知内核直到读到请求的数据字节数时，才返回。

4-2-1 recv 接收 一次发送超过 20 个字节，结果和 read 一样的

```
[root@Anokoro 04]# ./tcp_server4-2 4000
=====waiting for client's request=====
get connection from one client , recving...
recv , waiting for 20 bytes msg...
recv message from client: 一 二 三 四 五 六 七 八 九 十
a message be sent back to client
```

4-2-1 recv 接收

在客户端 20 个字节数据发送完之前，一直等待 20 个字节足够之后，才进行返回，打印得到的 20 个字节，并且向客户端返回信息

```
get connection from one client , recving...
recv , waiting for 20 bytes msg...
```

```

received : server : hello!
[root@Anokoro 04]# ./tcp_client4-2-2 192.168.70.30 4000
connect successfully...
send msg to server:
一 二 三 四 五 六 七 八 九 十 九 八 七
2 bytes have been sent...
4 bytes have been sent...
6 bytes have been sent...
8 bytes have been sent...
10 bytes have been sent...
12 bytes have been sent...
14 bytes have been sent...
16 bytes have been sent...
18 bytes have been sent...
20 bytes have been sent...
Received : server : hello!
[root@Anokoro 04]# █
get connection from one client , recving...
recv , waiting for 20 bytes msg...
recv message from client: 一 二 三 四 五 六 七 八 九 十
a message be sent back to client
█

```

20个发送之后，服务器recv才会返回，并且给客户机反馈信息

收到20个字节

给出 **read/recv** 函数的使用区别，给出 **write/send** 函数的使用区别

Recv 函数和 read 函数提供了 read 和 write 函数一样的功能，不同的是他们提供了四个参数。前面的三个参数和 read、write 函数是一样的。第四个参数可以是 0 或者是一下组合：

ssize_t **write**(int fd, const void *buf, size_t count);

int **send**(int s, const void *msg, size_t len, int flags);

flags 取值有：

0：与 write() 无异

MSG_DONTROUTE:告诉内核，目标主机在本地网络，不用查路由表

MSG_DONTWAIT:将单个 I / O 操作设置为非阻塞模式

MSG_OOB:指明发送的是带外信息

int **recv**(int s, void *buf, size_t len, int flags);

flags 取值有：

0：常规操作，与 read() 相同

MSG_DONTWAIT:将单个 I / O 操作设置为非阻塞模式

MSG_OOB:指明发送的是带外信息

MSG_PEEK:可以查看可读的信息，在接收数据后不会将这些数据丢失

MSG_WAITALL:通知内核直到读到请求的数据字节数时，才返回。

read 总是在接收缓冲区有数据时立即返回，而不是等到给定的 read buffer 填满时返回。只有当 receive buffer 为空时，blocking 模式才会等待

6. （05 子目录）写一对 TCP Socket 的测试程序，分为 client 和 server，分别运行在不同虚拟机上

测试程序 tcp_server5-1，接受 client 的连接成功后，用一句 getchar() 进入等待输入状态

```
[root@Anokoro 05]# ./tcp_server5-1 4000
=====waiting for client's request=====
getchar() status ...
```

测试程序 `tcp_client5-1`，连接服务端成功后，用 `write` 函数不断向服务端写入数据（加计数器统计写入了多少字节），大约写入多少字节后会使得 `write` 函数不再返回（阻塞状态）

客户端每次写一个字节的的数据。

```
446575 bytes have been written...
446576 bytes have been written...
446577 bytes have been written...
446578 bytes have been written...
446579 bytes have been written...
446580 bytes have been written...
```

大约写了 44.6W 个字节之后会使 `write` 函数不再返回

server 端在 `getchar()` 后用 `read` 进行读（假设每次读 `n` 个字节），读入多少字节后，client 端的 `write` 函数可以返回？这说明了什么问题？

```
85500 bytes has been read...
86000 bytes has been read...
86500 bytes has been read...
87000 bytes has been read...
87500 bytes has been read...
88000 bytes has been read...
88500 bytes has been read...
89000 bytes has been read...
```

哇....等了好久，服务端每隔 1 秒 read 500 的字节，大约读了 7-8W 左右字节之后，客户端 `write` 函数可以返回，并且不断发送消息，直到再次达到阻塞状态，如下图所示， $522470 - 446580 = 75890$ ，这个是准确近似准确的数字，

```
522468 bytes have been written...
522469 bytes have been written...
522470 bytes have been written...
```

第三次阻塞 $602030 - 522470 = 79560$

```
602025 bytes have been written...
602026 bytes have been written...
602027 bytes have been written...
602028 bytes have been written...
602029 bytes have been written...
602030 bytes have been written...
```

这说明的问题，这个比例大概是 $1/6$ ($8/45$)，就是说有至少要有 $1/6$ 的空闲空间，客户端才能发送消息到服务器。

`write` 成功返回，只是 `buf` 中的数据被复制到了 `kernel` 中的 TCP 发送缓冲区。至于数据什么时候被发往网络，什么时候被对方主机接收，什么时候被对方进程读取，系统调用层面不会给予任何保证和通知。已经发送到网络的数据依然需要暂存在 `send buffer` 中，只有收到对方的 `ack` 后，`kernel` 才从 `buffer` 中清除这一部分数据，为后续发送数据腾出空间。

在整个过程中用新会话打开终端后，用 `netstat` 命令观察 `tcp` 连接的各种信息（`netstat` 可以带哪些参数？显示的内容代表什么？）

不带任何参数

从整体上看，netstat 的输出结果可以分为两个部分：

一个是 Active Internet connections，称为有源 TCP 连接，其中"Recv-Q"和"Send-Q"指%0A 的是接收队列和发送队列。这些数字一般都应该是 0。大于 0 则表示软件包正在队列中堆积的数量。

另一个是 Active UNIX domain sockets，称为有源 Unix 域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)。

Proto 显示连接使用的协议,RefCnt 表示连接到本套接口上的进程号,Types 显示套接口的类型,State 显示套接口当前的状态,Path 表示连接到套接口的其它进程使用的路径名。

```
[root@Anokoro ~]# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp      176677 0 Anokoro:terabase
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
```

可以看到，这个等待队列是在不断变长，直到队列满（312161）便进入 write 的阻塞状态。

```
[root@Anokoro ~]# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 Anokoro:ssh           192.168.70.1:50954     ESTABLISHED
tcp      312161 0 Anokoro:terabase      192.168.70.230:56946   ESTABLISHED
tcp      0      96 Anokoro:ssh           192.168.70.1:52175     ESTABLISHED
tcp      0      0 Anokoro:ssh           192.168.70.1:50958     ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags   Type       State       I-Node  Path
unix   2      [ ]      DGRAM      -           8454     /run/systemd/notify
unix   2      [ ]      DGRAM      -           8456     /run/systemd/cgroups-age
unix   5      [ ]      DGRAM      -           8473     /run/systemd/journal/soc
```

带参数

- a (all)显示所有选项，默认不显示 LISTEN 相关
- t (tcp)仅显示 tcp 相关选项
- u (udp)仅显示 udp 相关选项
- n 拒绝显示别名，能显示数字的全部转化成数字。
- l 仅列出有在 Listen (监听) 的服务状态

- p 显示建立相关链接的程序名
- r 显示路由信息，路由表
- e 显示扩展信息，例如 uid 等
- s 按各个协议进行统计
- c 每隔一个固定时间，执行该 netstat 命令。

结合各个参数的作用，使用以下命令观察整个过程中的 tcp 连接的各种信息（可以使用 Ctrl+C 终止命令的循环执行）

可以看到 recv-Q 队列大约以 **1W 个/秒** 的数据增加（当服务器端）

```

[root@Anokoro ~]# netstat -tep -c 3
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp      0    96 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    5140      0 Anokoro:terabase
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    30606      0 Anokoro:terabase
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    61007      0 Anokoro:terabase
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    92499      0 Anokoro:terabase
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    214930      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    245926      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    276495      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    302859      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    302859      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp    302859      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh

```

再来看客户端的 netstat，在 Send-Q 里面有大约 13W+条，结合前面的数据，发现

服务端的接收队列 (30W+) + 客户端的发送队列 (13W+)
 = 客户端发送的字节总数(43W+)

```

433177 bytes have been written...
433178 bytes have been written...
433179 bytes have been written...

```

```

[root@Anokoro ~]# netstat -tep
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 Anokoro:ssh
tcp      0 130320 Anokoro:56948
tcp      0      0 Anokoro:ssh
tcp      0      0 Anokoro:ssh

```

还有一些其他的显示出的信息

服务器端

Local Address	Foreign Address	State	User	Inode	PID/Program name
Anokoro:ssh	192.168.70.1:50954	ESTABLISHED	root	26806	2475/sshd: root@pts
Anokoro:terabase	192.168.70.230:56948	ESTABLISHED	root	42774	2943/./tcp_server5-
Anokoro:ssh	192.168.70.1:52175	ESTABLISHED	root	41044	2881/sshd: root@pts
Anokoro:ssh	192.168.70.1:50958	ESTABLISHED	root	27049	2518/sshd: root@not

客户端

Local Address	Foreign Address	State	User	Inode	PID/Program name
Anokoro:ssh	192.168.70.1:40836	ESTABLISHED	root	49520	5210/sshd: root@not
Anokoro:56948	192.168.70.30:terabase	ESTABLISHED	root	175517	7594/./tcp_client5-
Anokoro:ssh	192.168.70.1:52328	ESTABLISHED	root	175824	7602/sshd: root@pts
Anokoro:ssh	192.168.70.1:50766	ESTABLISHED	root	159223	7217/sshd: root@pts

测试程序 tcp_server5-2/tcp_client_5-2，双方角色互换，即 server 写至阻塞为止，然后 client 开始读，直到 server 端解除阻塞，观察整个过程

从下图中的这组数据中（服务器端的）可以看到，在接收端的接收队列满之前，发送端的发送队列不一定是空的，原因可能是短时间要发送的消息比较多，所以会暂时进入发送队列，稍后会进行发送，而当接收端的接收队列满了之后，发送端的发送队列便不会再减小，而是不断增加，直到发送队列也满掉为止。

```
Active Internet conn
Proto Recv-Q Send-Q
tcp      0      1415
tcp      0      0
tcp      0      96
tcp      0      0
Active Internet conn
Proto Recv-Q Send-Q
tcp      0      1414
tcp      0      0
tcp      0      0
tcp      0      0
Active Internet conn
Proto Recv-Q Send-Q
tcp      0      0
tcp      0      0
tcp      0      0
tcp      0      0
Active Internet conn
Proto Recv-Q Send-Q
tcp      0      1415
tcp      0      0
tcp      0      0
tcp      0      0
Active Internet conn
Proto Recv-Q Send-Q
tcp      0      16039
tcp      0      0
```

服务端发送的总字节数

```
server has written 385757 bytes to client...
server has written 385758 bytes to client...
server has written 385759 bytes to client...
server has written 385760 bytes to client...
```

服务端发送队列的最终长度

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address
tcp      0 127424 Anokoro:terabase
tcp      0      0 Anokoro:ssh
```

客户端接收队列的最终长度

```
tcp      0      0 192.1
Active Internet connectio
Proto Recv-Q Send-Q Local
tcp      0      0 192.1
tcp      0      0 192.1
tcp      258336 0 192.1
tcp      0      0 192.1
AC
```

后两者之和等于第一个，结论和 5-1 相同。

测试程序 tcp_server5-3/tcp_client_5-3，功能同 5-1，在其中通过设置函数改变 TCP 收发缓冲区大小，通过 netstat 观察整个过程

通过 `getsockopt()`和 `setsockopt()`来实现

对于 **server** 端的 socket 一定要在 **listen** 之前设置缓冲区大小，因为，accept 时新产生的 socket 会继承监听 socket 的缓冲区大小。对于 **client** 端的 socket 一定要在 **connect** 之前设置缓冲区大小，因为 connect 时需要进行三次握手过程，会通知对方自己的窗口大小。在 connect 之后再设置缓冲区，已经没有什么意义。

```
int send_size, rcv_size;
socklen_t optlen = sizeof(rcv_size);
getsockopt(socket_fd, SOL_SOCKET, SO_SNDBUF, &send_size, &optlen); //获取发送缓冲区大小
getsockopt(socket_fd, SOL_SOCKET, SO_RCVBUF, &rcv_size, &optlen); //获取接收缓冲区大小

printf("原始发送缓冲区： %d 原始接收缓冲区： %d \n", send_size, rcv_size);

send_size = 8 * 1024; //设置发送缓冲区大小为10*1024
rcv_size = 10 * 1024; //设置接收缓冲区大小为10*1024
setsockopt(socket_fd, SOL_SOCKET, SO_SNDBUF, &send_size, optlen);
setsockopt(socket_fd, SOL_SOCKET, SO_RCVBUF, &rcv_size, optlen);
getsockopt(socket_fd, SOL_SOCKET, SO_SNDBUF, &send_size, &optlen); //获取发送缓冲区大小
getsockopt(socket_fd, SOL_SOCKET, SO_RCVBUF, &rcv_size, &optlen); //获取接收缓冲区大小

printf("设置后发送缓冲区： %d 设置后接收缓冲区： %d \n", send_size, rcv_size);
```

运行后，发现实际设置的值比我自己设置的值翻了一倍，似乎是有 `linux` 的内核算法决定的（这里服务器接收数据，因此只是把接收缓冲区设置小点，发送缓冲区没有改变，客户端反之）

```
[root@Anokoro 05]# ./tcp_server5-3 4000
原始发送缓冲区: 16384 原始接收缓冲区: 87380
设置后发送缓冲区: 16384 设置后接收缓冲区: 20480
=====waiting for client's request=====
```

```
[root@Anokoro 05]# ./tcp_client5-3 192.168.70.30 4000
原始发送缓冲区: 16384 原始接收缓冲区: 87380
设置后发送缓冲区: 8192 设置后接收缓冲区: 87380
connect successfully...
```

发现写了 21696bytes 数据之后就停止了

```
21693 bytes have been written...
21694 bytes have been written...
21695 bytes have been written...
21696 bytes have been written...
```

使用 netstat 查看

```
[root@Anokoro ~]# netstat -an | grep ESTABLISHED
```

Proto	Recv-Q	Send-Q
tcp	0	0
tcp	13008	0
tcp	0	0
tcp	0	0

服务端

```
[root@Anokoro ~]# netstat -an | grep ESTABLISHED
```

Proto	Recv-Q	Send-Q
tcp	0	0
tcp	0	8688
tcp	0	256
tcp	0	0

客户端

发现服务端的接收队列只用了 65%，而客户端的发送队列却超过了 100%。

再运行一次

```

[root@Anokoro ~]# netstat -an | grep -i tcp
Active Internet connections
Proto Recv-Q Send-Q
tcp        0      0
tcp        0 15006      0
tcp        0      0 96
tcp        0      0
客户端

[root@Anokoro ~]# netstat -an | grep -i tcp
Active Internet connections
Proto Recv-Q Send-Q
tcp        0      0
tcp        0      0
tcp        0 96
tcp        0 8688      0
tcp        0      0
服务端

```

第三次运行

服务端	<pre>[root@Anokoro ~]# netstat -an</pre> <pre>Active Internet connections</pre> <pre>Proto Recv-Q Send-Q</pre> <pre>tcp 0 0</pre> <pre>tcp 15603 0</pre> <pre>tcp 0 96</pre> <pre>tcp 0 0</pre>	客户端	<pre>[root@Anokoro ~]# netstat -an</pre> <pre>Active Internet connections</pre> <pre>Proto Recv-Q Send-Q</pre> <pre>tcp 0 0</pre> <pre>tcp 0 8688</pre> <pre>tcp 0 96</pre> <pre>tcp 0 0</pre>	总数	<pre>24288 bytes</pre> <pre>24289 bytes</pre> <pre>24290 bytes</pre> <pre>24291 bytes</pre>
-----	---	-----	--	----	---

因此发现，发送端的发送队列的最大长度是固定的，而接收端的接收队列长度却有波动。

7. （06 子目录）写一对 TCP Socket 的测试程序，分为 client 和 server，分别运行在不同虚拟机上

06-1 双方都先 read()再 write()死循环，无法正常收发数据，和设置的读写的长度无关，因为双方都阻塞在 read 阶段。

服务端 1

```
[root@Anokoro 06]# ./tcp_server6-1 4000 1000 900
```

```
=====waiting for client's request=====
```

```
get connection from one client
```

客户端 1

```
[root@Anokoro 06]# ./tcp_client6-1 192.168.70.30 4000 800 700
```

```
connect successfully...
```

服务端 2

```
[root@Anokoro 06]# ./tcp_server6-1 4000 1000 1000
```

```
=====waiting for client's request=====
```

```
get connection from one client
```

客户端 2

```
[root@Anokoro 06]# ./tcp_client6-1 192.168.70.30 4000 1000 1000
```

```
connect successfully...
```

06-2 双方都先 write()再 read()死循环，可以进行正常的通信

服务器端 1

```
[root@Anokoro 06]# ./tcp_server6-2 4000 1000 1000
```

```
=====waiting for client's request=====
```

```
get connection from one client
```

```
write 1000 read 1000
```

```
write 1000 read 1000
```

```
write 1000 read 1000
```

```
write 1000 read 1000
```

客户端 1。

```
[root@Anokoro 06]# ./tcp_client6-2 192.168.70.30 4000 1000 1000
connect successfully...
write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
```

服务器端 2

```
[root@Anokoro 06]# ./tcp_server6-2 4000 1000 500
=====waiting for client's request=====
get connection from one client
write 500 read 1000
write 500 read 1000
write 500 read 1000
write 500 read 1000
write 500 read 1000
```

客户端 2

```
[root@Anokoro 06]# ./tcp_client6-2 192.168.70.30 4000 500 1000
connect successfully...
write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
```

服务器端 3

```
[root@Anokoro 06]# ./tcp_server6-2 4000 1000 1000
=====waiting for client's request=====
get connection from one client
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
```

客户端 3

```
[root@Anokoro 06]# ./tcp_client6-2 192.168.70.30 4000 700 700
connect successfully...
write 700
read 700 write 700
read 700 write 700
read 700 write 700
read 700 write 700
```

06-3 server : 先 write 再 read , client: 先 read 再 write

服务端 1

```
[root@Anokoro 06]# ./tcp_server6-3 4000 1000 1000
=====waiting for client's request=====
get connection from one client
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
```

客户端 1

```
[root@Anokoro 06]# ./tcp_client6-3 192.168.70.30 4000 1000 1000
connect successfully...
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
```

服务端 2

```
[root@Anokoro 06]# ./tcp_server6-3 4000 1000 500
=====waiting for client's request=====
get connection from one client
write 500 read 1000
write 500 read 1000
write 500 read 1000
```

客户端 2

```
[root@Anokoro 06]# ./tcp_client6-3 192.168.70.30 4000 500 1000
connect successfully...
read 500 write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
```

服务端 3

```
[root@Anokoro 06]# ./tcp_server6-3 4000 1000 1000
=====waiting for client's request=====
get connection from one client
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
```

客户端 3

```
[root@Anokoro 06]# ./tcp_client6-3 192.168.70.30 4000 700 700
connect successfully...
read 700 write 700
read 700 write 700
read 700 write 700
read 700 write 700
read 700 write 700
```

06-4 server : 先 read 再 write , client: 先 write 再 read

服务端 1

```
[root@Anokoro 06]# ./tcp_server6-4 4000 1000 1000
=====waiting for client's request=====
get connection from one client
read 1000
write 1000 read 1000
write 1000 read 1000
write 1000 read 1000
```

客户端 1

```
[root@Anokoro 06]# ./tcp_client6-4 192.168.70.30 4000 1000 1000
connect successfully...
write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
read 1000 write 1000
```


服务端 2

```
[root@Anokoro 06]# ./tcp_server6-4 4000 1000 500
=====waiting for client's request=====
get connection from one client
read 1000
write 500    read 1000
write 500    read 1000
write 500    read 1000
■
```

客户端 2

```
~
[root@Anokoro 06]# ./tcp_client6-4 192.168.70.30 4000 500 1000
connect successfully...
write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
read 500 write 1000
```

服务端 3

```
[root@Anokoro 06]# ./tcp_server6-4 4000 1000 1000
=====waiting for client's request=====
get connection from one client
read 1000
write 1000    read 1000
write 1000    read 1000
write 1000    read 1000
write 1000    read 1000
```

客户端 3

```
~
[root@Anokoro 06]# ./tcp_client6-4 192.168.70.30 4000 700 700
connect successfully...
write 700
read 700 write 700
read 700 write 700
read 700 write 700
```

结论：除了双方都从 read 开始，其他情况都能够正常通信。