Basics of TCP implementation in Linux

Charles
2014/12/05

Outlines

- Basic skills to read kernel codes
  - Vim + Ctags
  - Grep: killer for function pointers
- BSD Socket APIs
  - Server side: socket(), bind(), listen(), accept()
  - Client side: socket(), connect()
  - send(), recv()
- Data strutures
  - socket, file descriptor(FD), sock
  - sock，inet_sock, inet_connection_sock, tcp_sock
  - sk_buff and routines operating on sk_buff
- 几个问题
  - 区分struct socket和structsock，它们的缩写分别是：sock和sk
  - 三次握手什么时候算完成了？
  - TCP/IP的包头具体是怎样封装并解封装的？
  - sock结构体嵌套了那么多层，是怎么分配内存的？

0. An simple example

```c
/* Server side */
int main(int argc, char *argv[])
{
    struct sockaddr_in serv_addr;

    int listenfd = 0, connfd = 0;

    char sendBuff[1025];

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        write(connfd, sendBuff, strlen(sendBuff));

        close(connfd);
     }
}
```

```c
/* Client side */
int main(int argc, char *argv[])
{
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);
    inet_aton(argv[1], &serv_addr.sin_addr);

    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    n = read(sockfd, recvBuff, sizeof(recvBuff));

    return 0;
}
```

1. socket()

- int socket (int family, int type, int protocol)

- listenfd = socket(AF_INET, SOCK_STREAM, 0);
  - AF_INET: ipv4 address family   SOCK_STREAM: socket type
  - If the protocol is 0, the family is instructed to select an appropriate default.

1.1 File descriptor, BSD socket and kernel sock

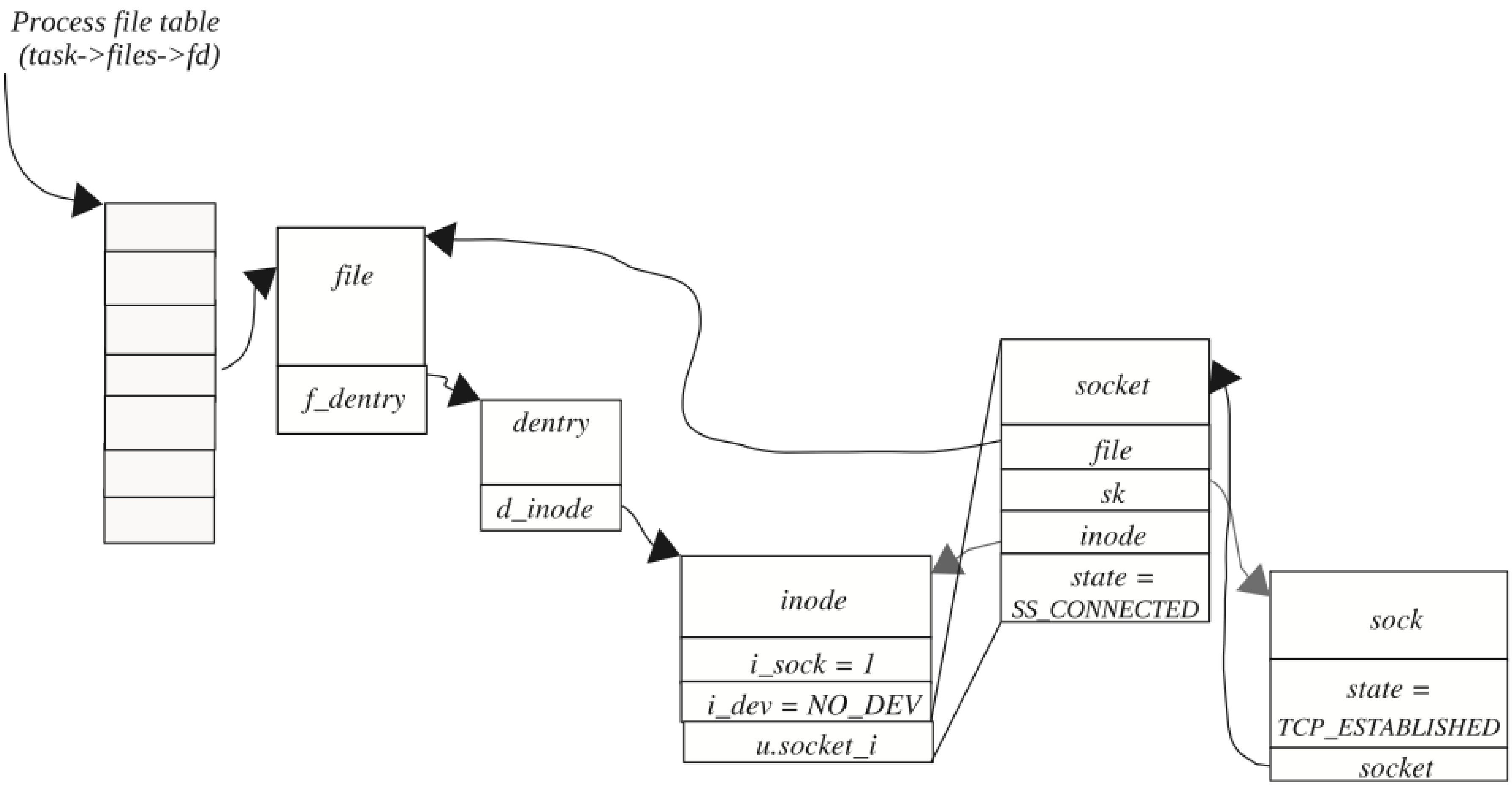

Figure 3.2. Socket accessed through process file table.

1.2 sock, inet_sock, tcp_sock

- 这些数据结构是一层一层嵌套的。
- 如果套用C++中类的概念：
  - sock 是父类，inet_sock是sock的子类
  - inet_connection_sock是inet_sock的子类
  - tcp_sock是inet_connection_sock的子类

  - 因此经常有这样的转换 （一个指向父类的指针，是可以转化成一个指向子类的指针。C++类实现不过尔尔）
    - ✓ struct tcp_sock *tp = (struct tcp_sock *)sk; // 将一个sock *转换为tcp_sock *

- 如何分配内存的？
  - sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_port)
    - ✓ answer_port == tcp_port
    - ✓ sk = kmalloc(prot->obj_size, priority)
    - ✓ .obj_size == sizeof(struct tcp_sock)

- 细节请参考代码

1.3 Main routines of socket()

```
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)  // net/socket.c
    => sock_create() // 在net/socket.c文件中
        => sock = sock_alloc()  // allocate the BSD socket
        => pf = rcu_dereference(net_families[family]);  // 根据family值，得到struct net_proto_family结构
        => pf->create() = inet_create()  // PF_INET对应的定义在net/ipv4/af_inet.c中
            => 遍历inetsw list，如果protocol没设置，则默认会匹配成IPPROTO_TCP
            => sock->ops = answer->ops  // 设置sock->ops = &inet_stream_ops
            => sk = sk_alloc()  // 分配struct sock结构体，传递了tcp_prot结构体地址作为参数，所以一次性分配了
            => sock_init_data()  // 完成sock结构体的初始化，把sock与socket关联

            => sk->sk_prot->init() = tcp_v4_init_sock()
                => tcp_init_sock()  // 完成tcp_sock结构体的初始化
    => sock_map_fd()  // bind sock with fd
        => fd = get_unused_fd_flags()
        => newfile = sock_alloc_file()  // create file 结构体，并于socket关联
```

细节请参考源码

2. bind()

- int bind(int sockfd, struct sockaddr *my_addr, int addrlen)

```
/* diff with 'sockaddr', which is more general */
struct sockaddr_in serv_addr;

memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(5000);
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```

```
struct sockaddr {
        sa_family_t     sa_family;      /* address family, AF_xxx       */
        char            sa_data[14];    /* 14 bytes of protocol address */
};

/* Structure describing an Internet (IP) socket address. */
#define __SOCK_SIZE__   16              /* sizeof(struct sockaddr)      */
struct sockaddr_in {
  __kernel_sa_family_t  sin_family;     /* Address family               */
  __be16                sin_port;       /* Port number                  */
  struct in_addr        sin_addr;       /* Internet address             */

  /* Pad to size of `struct sockaddr'. */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct in_addr)];
};

/* Internet address. */
struct in_addr {
        __be32  s_addr;
};
```

Note:
服务器端sin_addr一般设置为INADDR_ANY（0x00000000）。意思就是说可以接受来自不同网卡（服务器一般有多个网卡，也就对应对个IP地址）的连接请求。

2.1.1 Main routines of bind()

bind系统调用的过程

```
sys_bind()
    => sockfd_lookup_light() // 根据fd 获取套接口指针，并返回是否需要减少文件引用计数
    => sock->ops->bind()  == inet_bind()
        => sk->sk_prot->bind() == raw_bind()  // 如果是RAW sock
        => sk->sk_prot->get_port() == inet_csk_get_port()  // 如果是 TCP
        => sk->sk_prot->get_port() == udp_v4_get_port()  // 如果是 UDP
```

sock_fd_lookup_light的具体过程如下：

```
sock_fd_lookup_light()
    => file = fget_light(fd, fput_needed);    // 获取文件指针
    => sock = sock_from_file(file, err);      // 获取sock指针
        return file->private_data;  // private_data即是file结构体中的指向sock结构体的指针
```

2.1.2 Main routines of bind()

inet_bind的实现在net/ipv4/af_inet.c中，具体过程如下：

```
如果是RAW sock的话，直接调用bind
检查addr_len和sin_family
chk_addr_ret = inet_addr_type();  // 得到地址的类型，用于后续检查
对port进行判断，用户仅能使用Port >= 1024的端口。
根据sk->sk_state及inet->inet_num判断socket状态，检查重复绑定的错误

/* rcv_saddr用于hash lookups, inet_saddr用于transmit。正常情况下它们值相同 */
inet->inet_rcv_saddr = inet->inet_saddr = addr->sin_addr.s_addr;

调用sk->sk_prot->get_port(sk, snum);  // snum就是sin_port
```

细节请参考源码

3. listen()

- int listen (int fd, int backlog)

- listen(listenfd, 10);

- 功能很简单:
  - 初始化：icsk->icsk_accept_queue等
  - 将sk->sk_state 设置为TCP_LISTEN

- 但此时的socket与之前有着本质的区别:
  - 1. 一个调用了bind，而没有调用listen的socket，是仅仅绑定了端口（或IP）的，并不能接收连接请求。
    此时socket的状态还不是listening，如果客户端发出连接请求，服务器端会回复reset包
  - 2. 为socket调用了listen，而没有调用accept时，socket的状态是listening，
    此时如果客户端发出连接请求，能能够看到三次握手成功的【注意！】；
    同时客户端也能发送数据并收到ACK，但是在发完rwnd数量的数据后会收到rwnd等于0的确认包。
    此后客户端就会停止发送数据。最终由于服务器端不会consume接收的数据，会导致客户端的0窗口探测包超时后结束连接。

3.1 SYN queue and Accept queue

- After receiving a SYN pkt, and sent out a SYN/ACK pkt
  - Goto syn queue
- After receiving the final ACK pkt
  - Goto accept queue

3.2.1 Flow control for handling a new connection request

当TCP层收到一个IP包时，被调用的函数是tcp_v4_rcv()，该函数的实现是在net/ipv4/tcp_ipv4.c中。

```
tcp_v4_rcv(struct sk_buff *skb)
    => sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest) //根据四元组查找该IP
        => __inet_lookup()
            => __inet_lookup_established()  // 在tcp_ehash table中查找
            => __inet_lookup_listener()     // 上一步没找到再到hashinfo->listening-hash[]中查

    => ret = tcp_v4_do_rcv(sk, skb)
        => if (sk->sk_state == TCP__ESTABLISHED)
            => tcp_rcv_established()  // 后续再分析该函数
        => if (sk->sk_state == TCP_LISTEN)
            => struct sock *nsk = tcp_v4_hnd_req(sk, skb)    // 找到skb对应的sock，找不到则丢
            => if (nsk != sk)  // 如果nsk与sk不同，即说明已经为该connection request新建了sock
                => tcp_child_process(sk, nsk, skb)  // 对新建立的sock结构体做更多地处理
```

3.2.2 Flow control for handling a new connection request

```
=> tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)    // 根据不同的状态处理响应的包，此
    => case TCP_LISTEN:
        => icsk->icsk_af_ops->conn_request() == tcp_v4_conn_request()
            => inet_csk_reqsk_queue_is_full(sk)    // 判断request queue是否用满
            => sk_acceptq_is_full(sk)              // 判断accept queue是否用满
            => req = inet_reqsk_alloc()            // 为connection request分配一个request so
            => tcp_parse_options()                 // 解析TCP options
            => tcp_openreq_init()
            => ip_build_and_send_pkt()             // add an ip header to a skbuff and send
            => inet_csk_reqsk_queue_hash_add()     // add the request sock to the SYN table
    => case TCP_SYN_SENT:
        => queue = tcp_rcv_syssent_state_process(sk, skb, th, len)    // 代码里面注释较详细
            => tcp_finish_connect()     // 完成连接，进行最后的设置
                => tcp_set_state(sk, TCP_ESTABLISHED)    // 设置sk_state
                => tcp_init_congestion_control(sk)       // 设置congestion control algorithm,
                => tcp_init_buffer_space(sk)
```

细节请参考源码

4. accept()

- int accept(int fd, struct sockaddr *addr, int *addrlen)
- Connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
- 功能很简单：
  - Accept a pending connection

4.1 main routines of accept()

accept系统调用对应内核中的sys_accept()函数，具体的实现则在net/socket.c文件中。主要调用流程如下：

```
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
                int __user *, upeer_addrlen, int, flags)  == sys_accept4
    => sock = sockfd_lookup_light(fd, &err, &fput_needed)  // 根据监听的socket fd找到其sock结构
    => newsock = sock_alloc()   // 分配一个新的BSD socket
    => newfd = get_unused_fd_flags(flags)
    => newfile = sock_alloc_file(newsock, flags, sock->sk->sl_prot_creator->name)
    => err = sock->ops->accept(sock, newsock, sock->file->f_flags)  == inet_accept()
        => *sk2 = sk1->sk_prot->accept()  == inet_csk_accept()
            => if accept queue is empty, wait for connect if is blocking
            => otherwise, get the very first request

            => newsk = req->sk    // 获取request结构体中的sock结构体指针并返回
        => sock_graft(sk2, newsock)  // 将获取的sock结构体与之前新建的BSD socket关联
        => newsock->state = SS_CONNECTED;
    => fd_install(newfd, newfile);   // index newfile for the socket inode in the process fi
        => fd_install主要完成的动作就是：current->files->fd[fd] = file;
```

5. connect()

- int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)

```
memset(&serv_addr, '0', sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(5000);
inet_aton(argv[1], &serv_addr.sin_addr);

connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

- Connect的主要参数是server的IP和PORT
- Client一般不需要bind操作，因为IP是根据route确定的网卡确定的，port是选用一个空闲的（绕开TIME_WAIT状态）

5.1 main routine of connect()

```
SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr,
                int, addrlen)  == sys_connect()
    => sock = sockfd_lookup_light()
    => err = move_addr_to_kernel()
    => err = sock->ops->connect()  == inet_stream_connect()
        => Any state other then SS_UNCONNECTED is unacceptable for processing
        => err = sk->sk_prot->connect(sk, uaddr, addr_len)  == tcp_v4_connect()
            => rt = ip_route_connect()  // get the route for the dst addr. All routing entries for the system a
            => tcp_set_state(sk, TCP_SYN_SEND)
            => err = inet_hash_connect(&tcp_death_row, sk);  // 获得一个free的Port,流程与tcp_v4_get_port较类似
                => inet_get_local_port_range(&low, &high)
                => 遍历所有端口，对某个备选端口，遍历inet_bind_bucket确认是否有冲突。
                => 如果没有冲突，则tb = inet_bind_bucket_create()创建bind_bucket中的hash表项
            => Until now we got the route to destination, and obtained the local port number,
               and we have initialized remote address, remote port, local address, and local address fields of
            => err = tcp_connect(sk)  // generate a SYN packet and give it to the IP layer
                => tcp_connect_init(sk)  // do all connect socket setups that can be done AF independent
                    => tcp_select_initial_window() // determine  a window scaling and initial window to offer
                => buff = alloc_skb_fclone()  // allocate a sk_buff structure, 细节在下一章再写
                => tcp_transmit_skb(sk, buff, 1, sk->sk_allocation)  // 复制一份Buff, 然后发送出去
        => 至此已发送SYN包，然后等待SYN/ACK包从而完成三次握手
        => timeo = sock_sndtimeo(sk, flag * O_NONBLOCK)
        => inet_wait_for_connect(sk, timeo, writebias)  // 完成三次握手的最后的工作
```

6. struct sk_buff

- 整个TCP/IP stack管理数据包的一个重要数据结构

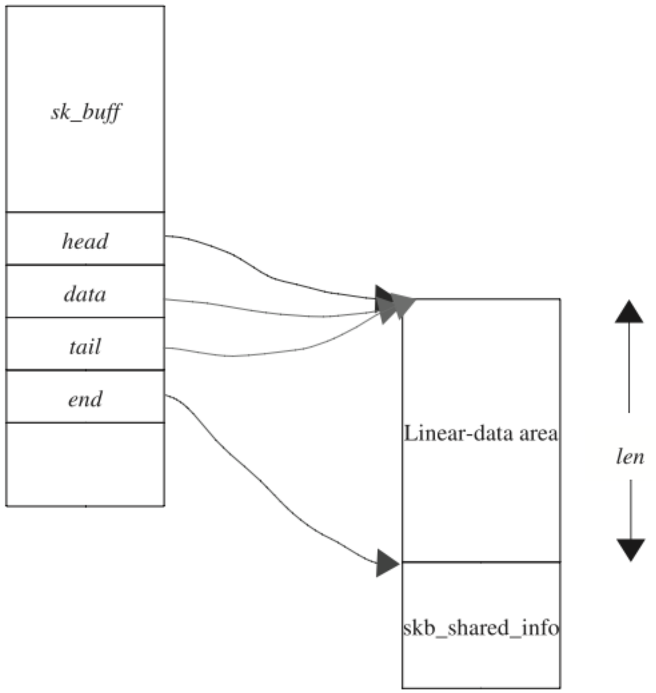

Figure 5.2. sk_buff when it is just as returned by skb_allocr().

6.1 routines operating on sk_buff

- stuct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask, int flags, int mode)
  - 该函数分配一个新的sk_buff实例，需要给定的参数是linear-data area的大小和内存分配的模式。
- static inline void skb_reserve(struct sk_buff *skb, int len)
  - 该函数将skb的data和tail指针往后移动，移动长度为len。常见的用途是为协议头部保留空间。
- unsigned char skb_put(struct sk_buff skb, unsigned int len)
  - 该函数负责将sk_buff的linear-data area的tail指针增加len。
- unsigned char skb_push(struct sk_buff skb, unsigend int len)
  - 该函数负责将sk_buff的linear-data area的data指针往前移动，距离为len。不难看出，skb_put是用来构建包的数据的，而skb_push则是当一个上层包传递到下一层后，下层在添加头部数据时调用skb_push。
- unsigned char skb_pull(struct sk_buff skb, unsigned int len)
  - 该函数与skb_push相对应，它将data指针往后移动，距离为len。也就意味着skb->len要减少len。当有数据包到达后，一层层解析包头的过程中往往会用到该函数。

7. send()

- int send (int fd, void *buff, int len, unsigned int flags)
- 一般情况下，send与write两个函数完成的功能一致
- 重点：
  - Send调用完成并不意味着数据已经发送到对端，而只是说数据放入了write queue
  - 数据的真实发送可能分为两种情形：　（接收数据也是如此）
    - ✓ 调用send时，尝试发送数据：process context
    - ✓ 接收到ack包后，尝试发送数据：IRQ context

7.1 pending a skb into write_queue

```
tcp_sendmsg()
    => mss_now = tcp_send_mss()  // 获得current mss
    => sg = !!(sk->sk_route_caps * NETIF_F_SG)  // 检查硬件是否支持scatter-gather
    => 两个循环，第一层遍历所有的buffer块，第二层遍历某一个buffer的所有数据
       /* 获取sk->sk_write_queue的最后一个skb，用于检查是否用满。
        * 用满了就新建一个skb放新数据，否则将新数据拼接到这最后一个skb中
        */
       => skb = tcp_write_queue_tail(sk)
       => if (copy <= 0) // 需要new a segment
          => sk_stream_memory_free(sk)  // 检查send buffer的配额是否超过上限，超过了要跳转到wait_for_sndbuf
             => return sk->sk_wmem_queued < sk->sk_sndbuf
          => skb = sk_stream_alloc_skb() // 为新数据新分配一个skb
          => skb_entail(sk, skb)    // 将新生成的skb挂到sk->sk_write_queue的尾部
       => skb_can_coalesce()  // 判断最后一页能否合并更多数据

       => forced_push(tp) // 解释见接下来的note
       => tcp_mark_push(tp, skb)  // 解释见接下来的note
       /* push out any pending frames which were held back due to TCP_CORK
        * or attempt at coalescing tiny packets
        */
       => __tcp_push_pending_frame()  // 如果是设置了PSH flag，会调用该函数尽快的将数据发送出去
```

7.1 send a skb

```
if (copied) tcp_push()  //发送数据
 => check sk->sk_send_head is NULL or not // 不为空表示有数据待发送
 => __tcp_push_pending_frames()  // 大部分数据应该是走这条流程被发送出去的
     => tcp_write_xmit()  // this rountine write packets to the network
         /* 只要有数据pending在write queue里面就继续发送，
          * 当然循环内部有各种条件判断是否应该终止循环
          */
         => while (skb = tcp_send_head(sk))
             => cwnd_quota = tcp_cwnd_test(tp, skb)  // 根据cwnd与packet in flight的差得到配额
             => if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)) break;  // 判断是否受限于rwnd
             => if (unlikely(!tcp_nagle_test()) break;  // return true if allow by Nagle
             => if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)) break;  // 发送一个skb，不成功则break。
                 => 细节见后续章节
             => tcp_event_new_data_sent(sk, skb)  // 更新sk->sk_send_head, tp->snd_nxt, tp->packet_out
             => if (push_one) break;  // 如果之前只允许发送一个，则break
```

8. recv()

- int recv(int fd, void *buf, int len, unsigned int flags)

- 重点
  - 多个接收队列：
    - ✓ prequeue,：如果有进程正在睡眠等待新数据的到来，则可以放入prequeue
    - ✓ receive_queue：所有按序到达的包，被解去包头后都是放在receive queue
    - ✓ backlog_queue：当sock结构体被锁定，则数据包会放入backloag queue
  - 数据的处理顺序
    - ✓ 优先处理receive queue中的数据
    - ✓ 若receive queue == NULL && prequeue != NULL, 处理prequeue
    - ✓ 最后处理backlog queue。由于backlog的特殊性，基本都是在release sock lock的时候处理它。

Thank you !

欢迎访问：perthcharles.github.com