高性能网络编程3----TCP消息的接收

2013年08月26日 18:55:02 阅读数：32869

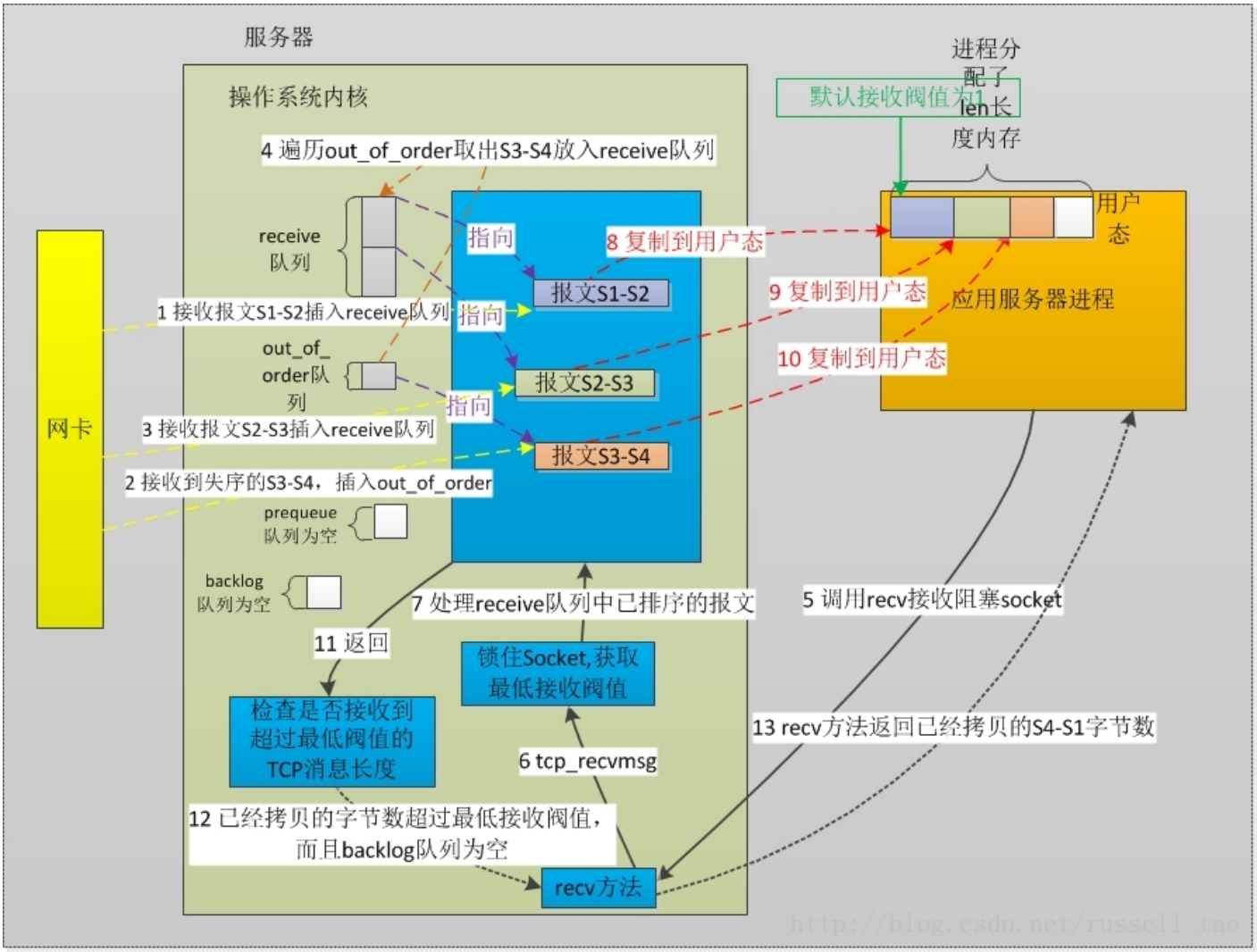
这篇文章将试图说明应用程序如何接收网络上发送过来的TCP消息流，由于篇幅所限，暂时忽略ACK报文的回复和接收窗口的滑动。
为了快速掌握本文所要表达的思想，我们可以带着以下问题阅读：
1、应用程序调用read、recv等方法时，socket套接字可以设置为阻塞或者非阻塞，这两种方式是如何工作的？
2、若socket为默认的阻塞套接字，此时recv方法传入的len参数，是表示必须超时（SO_RCVTIMEO）或者接收到len长度的消息，recv方法才会返回吗？而且，socket上可以设置一个属性叫做SO_RCVLOWAT，它会与len产生什么样的交集，又是决定recv等接收方法什么时候返回？
3、应用程序开始收取TCP消息，与程序所在的机器网卡上接收到网络里发来的TCP消息，这是两个独立的流程。它们之间是如何互相影响的？例如，应用程序正在收取消息时，内核通过网卡又在这条TCP连接上收到消息时，究竟是如何处理的？若应用程序没有调用read或者recv时，内核收到TCP连接上的消息后又是怎样处理的？
4、recv这样的接收方法还可以传入各种flags，例如MSG_WAITALL、MSG_PEEK、MSG_TRUNK等等。它们是如何工作的？
5、1个socket套接字可能被多个进程在使用，出现并发访问时，内核是怎么处理这种状况的？
6、linux的sysctl系统参数中，有类似tcp_low_latency这样的开关，默认为0或者配置为1时是如何影响TCP消息处理流程的？

书接上文。本文将通过三幅图讲述三种典型的接收TCP消息场景，理清内核为实现TCP消息的接收所实现的4个队列容器。当然，了解内核的实现并不是目的，而是如何使用socket接口、如何配置操作系统内核参数，才能使TCP传输消息更高效，这才是最终目的。

很多同学不希望被内核代码扰乱了思维，如何阅读本文呢？
我会在图1的步骤都介绍完了才来从代码上说明tcp_v4_rcv等主要方法。像flags参数、非阻塞套接字会产生怎样的效果我是在代码介绍中说的。然后我会介绍图2、图3，介绍它们的步骤时我会穿插一些上文没有涉及的少量代码。不喜欢了解内核代码的同学请直接看完图1的步骤后，请跳到图2、图3中，我认为这3幅图覆盖了主要的TCP接收场景，能够帮助你理清其流程。

接收消息时调用的系统方法要比上一篇发送TCP消息复杂许多。接收TCP消息的过程可以一分为二：首先是PC上的网卡接收到网线传来的报文，通过软中断内核拿到并且解析其为TCP报文，然后TCP模块决定如何处理这个TCP报文。其次，用户进程调用read、recv等方法获取TCP消息，则是将内核已经从网卡上收到的消息流拷贝到用户进程里的内存中。

第一幅图描述的场景是，TCP连接上将要收到的消息序号是S1（TCP上的每个报文都有序号，详见《TCP/IP协议详解》），此时操作系统内核依次收到了序号S1-S2的报文、S3-S4、S2-S3的报文，注意后两个包乱序了。之后，用户进程分配了一段len大小的内存用于接收TCP消息，此时，len是大于S4-S1的。另外，用户进程始终没有对这个socket设置过SO_RCVLOWAT参数，因此，接收阀值SO_RCVLOWAT使用默认值1。另外，系统参数tcp_low_latency设置为0，即从操作系统的总体效率出发，使用prequeue队列提升吞吐量。当然，由于用户进程收消息时，并没有新包来临，所以此图中prequeue队列始终为空。先不细表。
图1如下：



上图中有13个步骤，应用进程使用了阻塞套接字，调用recv等方法时flag标志位为0，用户进程读取套接字时没有发生进程睡眠。内核在处理接收到的TCP报文时使用了4个队列容器（当链表理解也可），分别为receive、out_of_order、prequeue、backlog队列，本文会说明它们存在的意义。下面详细说明这13个步骤。

1、当网卡接收到报文并判断为TCP协议后，将会调用到内核的tcp_v4_rcv方法。此时，这个TCP连接上需要接收的下一个报文序号恰好就是S1，而这一步里，网卡上收到了S1-S2的报文，所以，tcp_v4_rcv方法会把这个报文直接插入到receive队列中。

注意：receive队列是允许用户进程直接读取的，它是将已经接收到的TCP报文，去除了TCP头部、排好序放入的、用户进程可以直接按序读取的队列。由于socket不在进程上下文中（也就是没有进程在读socket），由于我们需要S1序号的报文，而恰好收到了S1-S2报文，因此，它进入了receive队列。

2、接着，我们收到了S3-S4报文。在第1步结束后，这时我们需要收到的是S2序号，但到来的报文却是S3打头的，怎么办呢？进入out_of_order队列！从这个队列名称就可以看出来，所有乱序的报文都会暂时放在这。

3、仍然没有进入来读取socket，但又过来了我们期望的S2-S3报文，它会像第1步一样，直接进入receive队列。不同的时，由于此时out_of_order队列不像第1步是空的，所以，引发了接来的第4步。

4、每次向receive队列插入报文时都会检查out_of_order队列。由于收到S2-S3报文后，期待的序号成为了S3，这样，out_of_order队列里的唯一报文S3-S4报文将会移出本队列而插入到receive队列中（这件事由tcp_ofo_queue方法完成）。

5、终于有用户进程开始读取socket了。做过应用端编程的同学都知道，先要在进程里分配一块内存，接着调用read或者recv等方法，把内存的首地址和内存长度传入，再把建立好连接的socket也传入。当然，对这个socket还可以配置其属性。这里，假定没有设置任何属性，都使用默认值，因此，此时socket是阻塞式，它的SO_RCVLOWAT是默认的1。当然，recv这样的方法还会接收一个flag参数，它可以设置为MSG_WAITALL、MSG_PEEK、MSG_TRUNK等等，这里我们假定为最常用的0。进程调用了recv方法。

6、无论是何种接口，C库和内核经过层层封装，接收TCP消息最终一定会走到tcp_recvmsg方法。下面介绍代码细节时，它会是重点。

7、在tcp_recvmsg方法里，会首先锁住socket。为什么呢？因此socket是可以被多进程同时使用的，同时，内核中断也会操作它，而下面的代码都是核心的、操作数据的、有状态的代码，不可以被重入的，锁住后，再有用户进程进来时拿不到锁就要休眠在这了。内核中断看到被锁住后也会做不同的处理，参见图2、图3。

8、此时，第1-4步已经为receive队列里准备好了3个报文。最上面的报文是S1-S2，将它拷贝到用户态内存中。由于第5步flag参数并没有携带MSG_PEEK这样的标志位，因此，再将S1-S2报文从receive队列的头部移除，从内核态释放掉。反之，MSG_PEEK标志位会导致receive队列不会删除报文。所以，MSG_PEEK主要用于多进程读取同一套接字的情形。

9、如第8步，拷贝S2-S3报文到用户态内存中。当然，执行拷贝前都会检查用户态内存的剩余空间是否足以放下当前这个报文，不足以时会直接返回已经拷贝的字节数。
10、同上。

11、receive队列为空了，此时会先来检查SO_RCVLOWAT这个阀值。如果已经拷贝的字节数到现在还小于它，那么可能导致进程会休眠，等待拷贝更多的数据。第5步已经说明过了，socket套接字使用的默认的SO_RCVLOWAT，也就是1，这表明，只要读取到报文了，就认为可以返回了。
做完这个检查了，再检查backlog队列。backlog队列是进程正在拷贝数据时，网卡收到的报文会进这个队列。此时若backlog队列有数据，就顺带处理下。图3会覆盖这种场景。

12、在本图对应的场景中，backlog队列是没有数据的，已经拷贝的字节数为S4-S1，它是大于1的，因此，释放第7步里加的锁，准备返回用户态了。

13、用户进程代码开始执行，此时recv等方法返回的就是S4-S1，即从内核拷贝的字节数。

图1描述的场景是最简单的1种场景，下面我们来看看上述步骤是怎样通过内核代码实现的（以下代码为2.6.18内核代码）。

我们知道，linux对中断的处理是分为上半部和下半部的，这是处于系统整体效率的考虑。我们将要介绍的都是在网络软中断的下半部里，例如这个tcp_v4_rcv方法。图1中的第1-4步都是在这个方法里完成的。

```
int tcp_v4_rcv(struct sk_buff *skb)
{
        ... ...
        //是否有进程正在使用这个套接字，将会对处理流程产生影响
        //或者从代码层面上，只要在tcp_recvmsg里，执行lock_sock后只能进入else，而release_
        if (!sock_owned_by_user(sk)) {
                {
                        //当 tcp_prequeue 返回0时，表示这个函数没有处理该报文
                        if (!tcp_prequeue(sk, skb))//如果报文放在prequeue队列，即表示
                            ret = tcp_v4_do_rcv(sk, skb);//不使用prequeue或者没有用户
                }
        } else
                sk_add_backlog(sk, skb);//如果进程正在操作套接字，就把skb指向的TCP报文
        ... ...
}
```

图1第1步里，我们从网络上收到了序号为S1-S2的包。此时，没有用户进程在读取套接字，因此，sock_owned_by_user(sk)会返回0。所以，tcp_prequeue方法将得到执行。简单看看它：

```
static inline int tcp_prequeue(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);

        //检查tcp_low_latency，默认其为0，表示使用prequeue队列。tp->ucopy.task不为0，表
        if (!sysctl_tcp_low_latency && tp->ucopy.task) {
                //到这里，通常是用户进程读数据时没读到指定大小的数据，休眠了。直接将报文插
                __skb_queue_tail(&tp->ucopy.prequeue, skb);
                tp->ucopy.memory += skb->truesize;
                //当然，虽然通常是延后处理，但如果TCP的接收缓冲区不够用了，就会立刻处理pre
                if (tp->ucopy.memory > sk->sk_rcvbuf) {
                        while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
                                //sk_backlog_rcv就是下文将要介绍的tcp_v4_do_rcv方法
                                sk->sk_backlog_rcv(sk, skb1);
                        }
                } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
                        //prequeue里有报文了，唤醒正在休眠等待数据的进程，让进程在它的上
                        wake_up_interruptible(sk->sk_sleep);
                }

                return 1;
        }
        //prequeue没有处理
        return 0;
}
```

由于tp->ucopy.task此时是NULL，所以我们收到的第1个报文在tcp_prequeue函数里直接返回了0，因此，将由 tcp_v4_do_rcv方法处理。

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
        if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
                //当TCP连接已经建立好时，是由tcp_rcv_established方法处理接收报文的
                if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
                        goto reset;

                return 0;
        }
        ... ...
}
```

tcp_rcv_established方法在图1里，主要调用tcp_data_queue方法将报文放入队列中，继续看看它又干了些什么事：

```
static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);

        //如果这个报文是待接收的报文（看seq），它有两个出路：进入receive队列，正如图1；直接
        if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
                //滑动窗口外的包暂不考虑，篇幅有限，下次再细谈
                if (tcp_receive_window(tp) == 0)
                        goto out_of_window;

                //如果有一个进程正在读取socket，且正准备要拷贝的序号就是当前报文的seq序号
                if (tp->ucopy.task == current &&
                    tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
                    sock_owned_by_user(sk) && !tp->urg_data) {
                        //直接将报文内容拷贝到用户态内存中，参见图3
                        if (!skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk))
                                tp->ucopy.len -= chunk;
                                tp->copied_seq += chunk;
                        }
                }

                if (eaten <= 0) {
queue_and_out:
                        //如果没有能够直接拷贝到用户内存中，那么，插入receive队列吧，正如
                        __skb_queue_tail(&sk->sk_receive_queue, skb);
                }
                //更新待接收的序号，例如图1第1步中，更新为S2
                tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;

                //正如图1第4步，这时会检查out_of_order队列，若它不为空，需要处理它
                if (!skb_queue_empty(&tp->out_of_order_queue)) {
                        //tcp_ofo_queue方法会检查out_of_order队列中的所有报文
                        tcp_ofo_queue(sk);
                }
        }
        ... ...

        //这个包是无序的，又在接收滑动窗口内，那么就如图1第2步，把报文插入到out_of_order队
        if (!skb_peek(&tp->out_of_order_queue)) {
                __skb_queue_head(&tp->out_of_order_queue,skb);
        } else {
                ... ...
                    __skb_append(skb1, skb, &tp->out_of_order_queue);
        }
```

```
}
```

图1第4步时，正是通过tcp_ofo_queue方法把之前乱序的S3-S4报文插入receive队列的。

```
static void tcp_ofo_queue(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        __u32 dsack_high = tp->rcv_nxt;
        struct sk_buff *skb;
        //遍历out_of_order队列
        while ((skb = skb_peek(&tp->out_of_order_queue)) != NULL) {
                ... ...

            //若这个报文可以按seq插入有序的receive队列中，则将其移出out_of_order队列
                __skb_unlink(skb, &tp->out_of_order_queue);
            //插入receive队列
            __skb_queue_tail(&sk->sk_receive_queue, skb);
            //更新socket上待接收的下一个有序seq
            tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;         }
}
```

下面再介绍图1第6步提到的tcp_recvmsg方法。

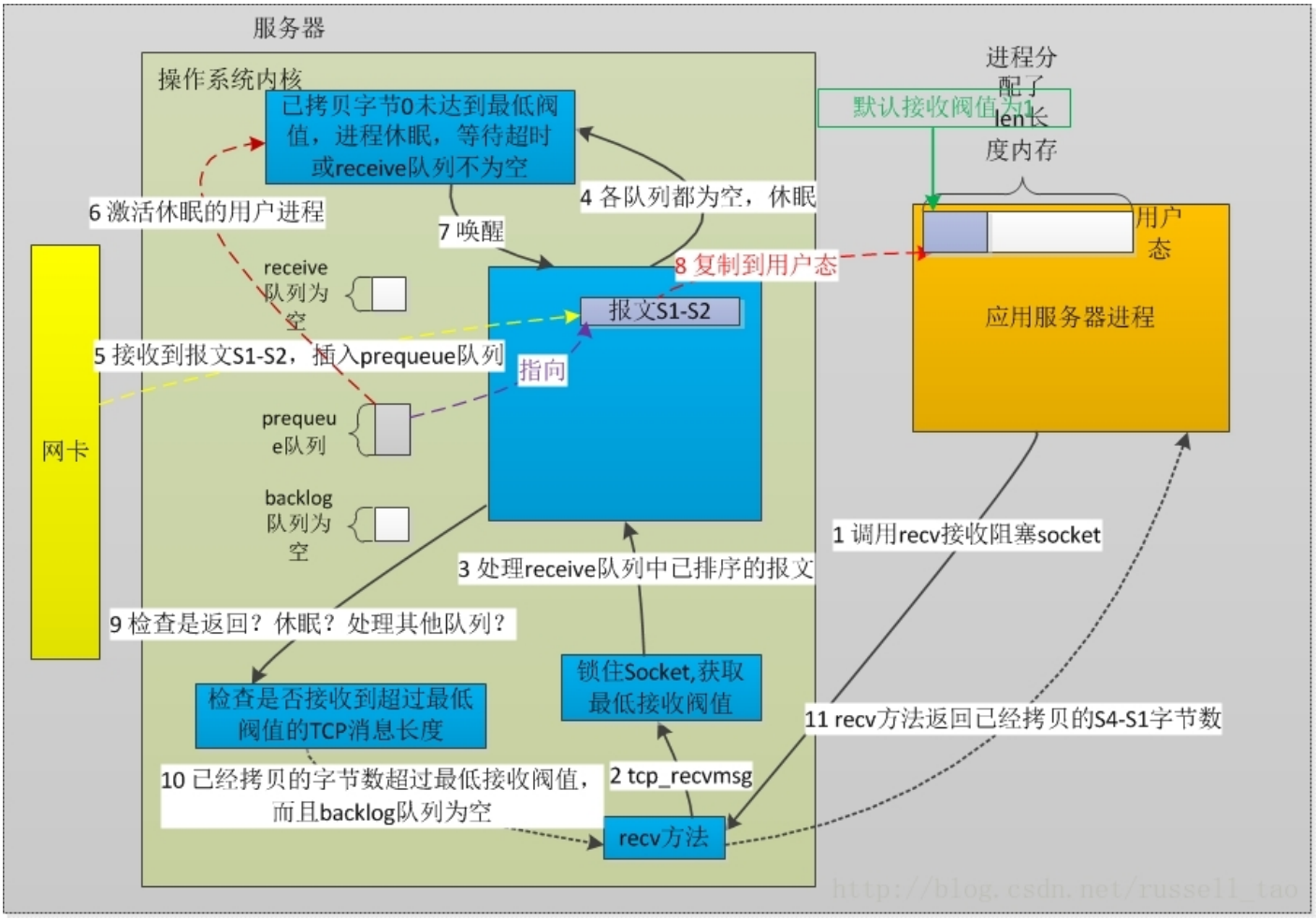
```
//参数里的len就是read、recv方法里的内存长度，flags正是方法的flags参数，nonblock则是阻塞、
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len, int nonblock, int flags, int *addr_len)
{
        //锁住socket，防止多进程并发访问TCP连接，告知软中断目前socket在进程上下文中
        lock_sock(sk);

        //初始化errno这个错误码
        err = -ENOTCONN;

        //如果socket是阻塞套接字，则取出SO_RCVTIMEO作为读超时时间；若为非阻塞，则timeo为
        timeo = sock_rcvtimeo(sk, nonblock);

        //获取下一个要拷贝的字节序号
        //注意：seq的定义为u32 *seq;，它是32位指针。为何？因为下面每向用户态内存拷贝后，
        seq = &tp->copied_seq;
        //当flags参数有MSG_PEEK标志位时，意味着这次拷贝的内容，当再次读取socket时（比如
        if (flags & MSG_PEEK) {
                //所以不会更新copied_seq，当然，下面会看到也不会删除报文，不会从receive
                peek_seq = tp->copied_seq;
```

```
                seq = &peek_seq;
                        }

        //获取SO_RCVLOWAT最低接收阀值，当然，target实际上是用户态内存大小len和SO_RCVLOWA
        //注意：flags参数中若携带MSG_WAITALL标志位，则意味着必须等到读取到len长度的消息才
        target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);

        //以下开始读取消息
        do {
                //从receive队列取出1个报文
                skb = skb_peek(&sk->sk_receive_queue);
                do {
                        //没取到退出当前循环
                        if (!skb)
                                break;


                        //offset是待拷贝序号在当前这个报文中的偏移量，在图1、2、3中它都为
                        offset = *seq - TCP_SKB_CB(skb)->seq;
                        //有些时候，三次握手的SYN包也会携带消息内容的，此时seq是多出1的
                        if (skb->h.th->syn)
                                offset--;
                        //若偏移量还有这个报文之内，则认为它需要处理
                        if (offset < skb->len)
                                goto found_ok_skb;

                        skb = skb->next;
                } while (skb != (struct sk_buff *)&sk->sk_receive_queue);

                //如果receive队列为空，则检查已经拷贝的字节数，是否达到了SO_RCVLOWAT或者
                if (copied >= target && !sk->sk_backlog.tail)
                        break;

                //在tcp_recvmsg里，copied就是已经拷贝的字节数
                if (copied) {
                        ... ...
                } else {
                        //一个字节都没拷贝到，但如果shutdown关闭了socket，一样直接返回。
                        if (sk->sk_shutdown & RCV_SHUTDOWN)
                                break;

                        //如果使用了非阻塞套接字，此时timeo为0
                        if (!timeo) {
                                //非阻塞套接字读取不到数据时也会返回，错误码正是EAGAIN
                                copied = -EAGAIN;
                                break;
                        }
```

```
                ... ...                 }

                //tcp_low_latency默认是关闭的，图1、图2都是如此，图3则例外，即图3不会走
                if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
                        //prequeue队列就是为了提高系统整体效率的，即prequeue队列有可能
                        if (!skb_queue_empty(&tp->ucopy.prequeue))
                                goto do_prequeue;
                }

                //如果已经拷贝了的字节数超过了最低阀值
                if (copied >= target) {
                        //release_sock这个方法会遍历、处理backlog队列中的报文
                        release_sock(sk);
                        lock_sock(sk);
                } else
                        sk_wait_data(sk, &timeo);//没有读取到足够长度的消息，因此会进程

                if (user_recv) {
                        if (tp->rcv_nxt == tp->copied_seq &&
                            !skb_queue_empty(&tp->ucopy.prequeue)) {
do_prequeue:
                                //接上面代码段，开始处理prequeue队列里的报文
                                tcp_prequeue_process(sk);
                        }
                }

                //继续处理receive队列的下一个报文
                continue;

        found_ok_skb:
                /* Ok so how much can we use? */
                //receive队列的这个报文从其可以使用的偏移量offset，到总长度len之间，可以
                used = skb->len - offset;
                //len是用户态空闲内存，len更小时，当然只能拷贝len长度消息，总不能导致内存
                if (len < used)
                        used = len;

                //MSG_TRUNC标志位表示不要管len这个用户态内存有多大，只管拷贝数据吧
                if (!(flags & MSG_TRUNC)) {
                        {
                                //向用户态拷贝数据
                                err = skb_copy_datagram_iovec(skb, offset,
                                                msg->msg_iov, used);
                        }
                }

                //因为是指针，所以同时更新copied_seq--下一个待接收的序号
```

```
                *seq += used;                                   //更新已经拷贝的长度
                copied += used;
                //更新用户态内存的剩余空闲空间长度
                len -= used;

                ... ...
        } while (len > 0);

        //已经装载了接收器
        if (user_recv) {
                //prequeue队列不为空则处理之
                if (!skb_queue_empty(&tp->ucopy.prequeue)) {
                        tcp_prequeue_process(sk);
                }

                //准备返回用户态，socket上不再装载接收任务
                tp->ucopy.task = NULL;
                tp->ucopy.len = 0;
        }

        //释放socket时，还会检查、处理backlog队列中的报文
        release_sock(sk);
        //向用户返回已经拷贝的字节数
        return copied;
}
```

图2给出了第2种场景，这里涉及到prequeue队列。用户进程调用recv方法时，连接上没有任何接收并缓存到内核的报文，而socket是阻塞的，所以进程睡眠了。然后网卡中收到了TCP连接上的报文，此时prequeue队列开始产生作用。图2中tcp_low_latency为默认的0，套接字socket的SO_RCVLOWAT是默认的1，仍然是阻塞socket，如下图：



简单描述上述11个步骤：
1、用户进程分配了一块len大小的内存，将其传入recv这样的函数，同时socket参数皆为默认，即阻塞的、SO_RCVLOWAT为1。调用接收方法，其中flags参数为0。

2、C库和内核最终调用到tcp_recvmsg方法来处理。

3、锁住socket。

4、由于此时receive、prequeue、backlog队列都是空的，即没有拷贝1个字节的消息到用户内存中，而我们的最低要求是拷贝至少SO_RCVLOWAT为1长度的消息。此时，开始进入阻塞式套接字的等待流程。最长等待时间为SO_RCVTIMEO指定的时间。
这个等待函数叫做sk_wait_data，有必要看下其实现：

```
int sk_wait_data(struct sock *sk, long *timeo)
{
        //注意，它的自动唤醒条件有两个，要么timeo时间到达，要么receive队列不为空
        rc = sk_wait_event(sk, timeo, !skb_queue_empty(&sk->sk_receive_queue));
}
```

sk_wait_event也值得我们简单看下：

```
#define sk_wait_event(__sk, __timeo, __condition)               \
({      int rc;                                                 \
        release_sock(__sk);                                     \
        rc = __condition;                                       \
        if (!rc) {                                              \
```

```
                *(__timeo) = schedule_timeout(*(__timeo));      \

        }                                                       \
        lock_sock(__sk);                                        \
        rc = __condition;                                       \
        rc;                                                     \ })
```

注意，它在睡眠前会调用release_sock，这个方法会释放socket锁，使得下面的第5步中，新到的报文不再只能进入backlog队列。

5、这个套接字上期望接收的序号也是S1，此时网卡恰好收到了S1-S2的报文，在tcp_v4_rcv方法中，通过调用tcp_prequeue方法把报文插入到prequeue队列中。

6、插入prequeue队列后，此时会接着调用wake_up_interruptible方法，唤醒在socket上睡眠的进程。参见tcp_prequque方法。

7、用户进程被唤醒后，重新调用lock_sock接管了这个socket，此后再进来的报文都只能进入backlog队列了。

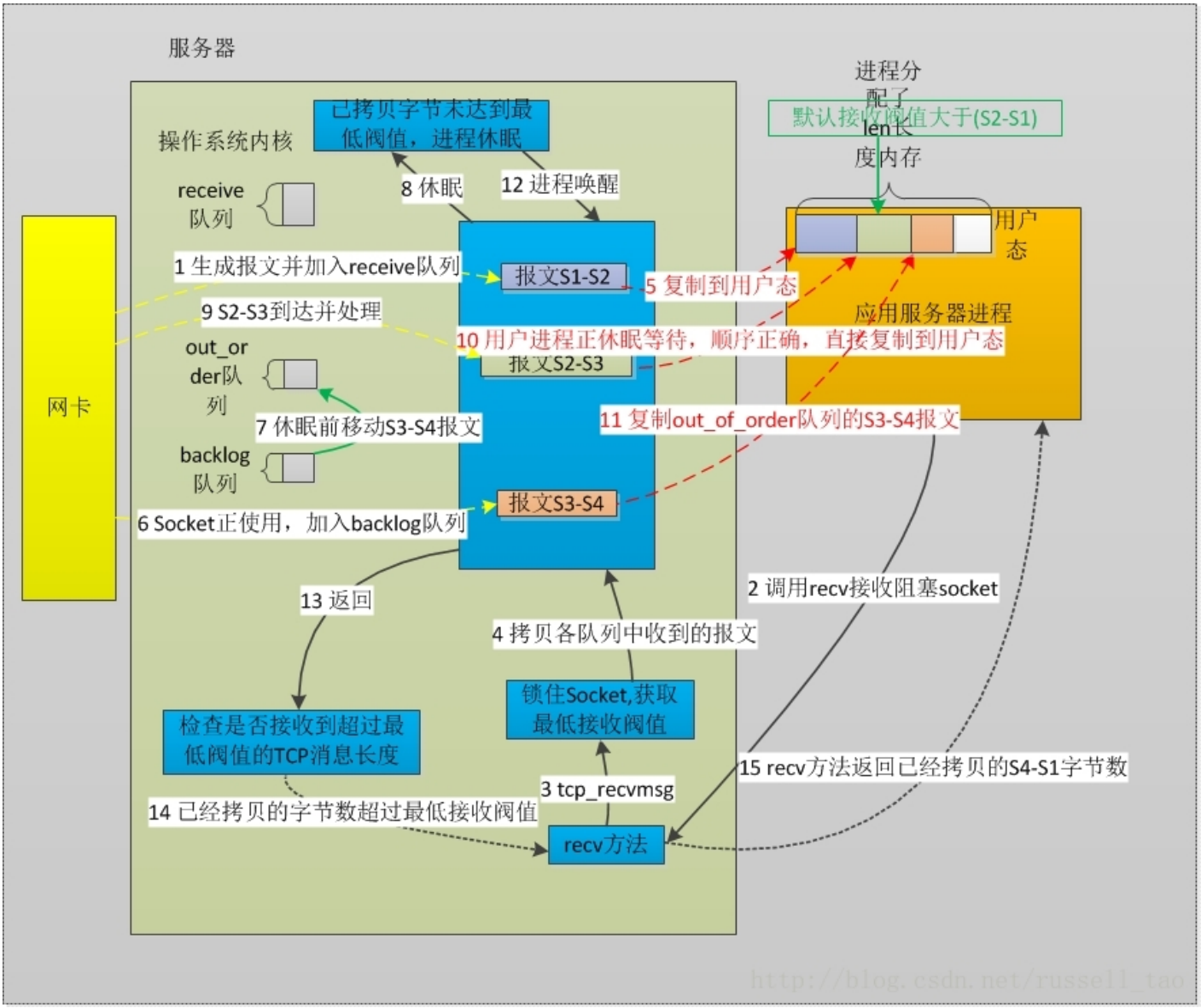
8、进程醒来后，先去检查receive队列，当然仍然是空的；再去检查prequeue队列，发现有一个报文S1-S2，正好是socket连接待拷贝的起始序号S1，于是，从prequeue队列中取出这个报文并把内容复制到用户内存中，再释放内核中的这个报文。

9、目前已经拷贝了S2-S1个字节到用户态，检查这个长度是否超过了最低阀值（即len和SO_RCVLOWAT的最小值）。

10、由于SO_RCVLOWAT使用了默认的1，所以准备返回用户。此时会顺带再看看backlog队列中有没有数据，若有，则检查这个无序的队列中是否有可以直接拷贝给用户的报文。当然，此时是没有的。所以准备返回，释放socket锁。

11、返回用户已经拷贝的字节数。

图3给出了第3种场景。这个场景中，我们把系统参数tcp_low_latency设为1，socket上设置了SO_RCVLOWAT属性的值。服务器先是收到了S1-S2这个报文，但S2-S1的长度是小于SO_RCVLOWAT的，用户进程调用recv方法读套接字时，虽然读到了一些，但没有达到最小阀值，所以进程睡眠了，与此同时，在睡眠前收到的乱序的S3-S4包直接进入backlog队列。此时先到达了S2-S3包，由于没有使用prequeue队列，而它起始序号正是下一个待拷贝的值，所以直接拷贝到用户内存中，总共拷贝字节数已满足SO_RCVLOWAT的要求！最后在返回用户前把backlog队列中S3-S4报文也拷贝给用户了。如下图：



简明描述上述15个步骤：

1、内核收到报文S1-S2，S1正是这个socket连接上待接收的序号，因此，直接将它插入有序的receive队列中。

2、用户进程所处的linux操作系统上，将sysctl中的tcp_low_latency设置为1。这意味着，这台服务器希望TCP进程能够更及时的接收到TCP消息。用户调用了recv方法接收socket上的消息，这个socket上设置了SO_RCVLOWAT属性为某个值n，这个n是大于S2-S1，也就是第1步收到的报文大小。这里，仍然是阻塞socket，用户依然是分配了足够大的len长度内存以接收TCP消息。

3、通过tcp_recvmsg方法来完成接收工作。先锁住socket，避免并发进程读取同一socket的同时，也在告诉内核网络软中断处理到这一socket时要有不同行为，如第6步。

4、准备处理内核各个接收队列中的报文。

5、receive队列中的有序报文可直接拷贝，在检查到S2-S1是小于len之后，将报文内容拷贝到用户态内存中。

6、在第5步进行的同时，socket是被锁住的，这时内核又收到了一个S3-S4报文，因此报文直接进入backlog队列。注意，这个报文不是有序的，因为此时连接上期待接收序号为S2。

7、在第5步，拷贝了S2-S1个字节到用户内存，它是小于SO_RCVLOWAT的，因此，由于socket是阻塞型套接字（超时时间在本文中忽略），进程将不得不转入睡眠。转入睡眠之前，还会干一件事，就是处理backlog队列里的报文，图2的第4步介绍过休眠方法sk_wait_data，它在睡眠前会执行release_sock方法，看看是如何实现的：

```
void fastcall release_sock(struct sock *sk)
{
```

```
        mutex_release(&sk->sk_lock.dep_map, 1, _RET_IP_);

        spin_lock_bh(&sk->sk_lock.slock);
        //这里会遍历backlog队列中的每一个报文
        if (sk->sk_backlog.tail)
                __release_sock(sk);
        //这里是网络中断执行时，告诉内核，现在socket并不在进程上下文中
        sk->sk_lock.owner = NULL;
        if (waitqueue_active(&sk->sk_lock.wq))
                wake_up(&sk->sk_lock.wq);
        spin_unlock_bh(&sk->sk_lock.slock);
}
```

再看看__release_sock方法是如何遍历backlog队列的：

```
static void __release_sock(struct sock *sk)
{
        struct sk_buff *skb = sk->sk_backlog.head;

        //遍历backlog队列
        do {
                sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
                bh_unlock_sock(sk);

                do {
                        struct sk_buff *next = skb->next;

                        skb->next = NULL;
                        //处理报文，其实就是tcp_v4_do_rcv方法，上文介绍过，不再赘述
                        sk->sk_backlog_rcv(sk, skb);

                        cond_resched_softirq();

                        skb = next;
                } while (skb != NULL);

                bh_lock_sock(sk);
        } while((skb = sk->sk_backlog.head) != NULL);
}
```

此时遍历到S3-S4报文，但因为它是失序的，所以从backlog队列中移入out_of_order队列中（参见上文说过的tcp_ofo_queue方法）。

8、进程休眠，直到超时或者receive队列不为空。

9、内核接收到了S2-S3报文。注意，这里由于打开了tcp_low_latency标志位，这个报文是不会进入prequeue队列以待进程上下文处理的。

10、此时，由于S2是连接上正要接收的序号，同时，有一个用户进程正在休眠等待接收数据中，且它要等待的数据起始序号正是S2，于是，这种种条件下，使得这一步同时也是网络软中断执行上下文中，把S2-S3报文直接拷贝进用户内存。

11、上文介绍tcp_data_queue方法时大家可以看到，每处理完1个有序报文（无论是拷贝到receive队列还是直接复制到用户内存）后都会检查out_of_order队列，看看是否有报文可以处理。那么，S3-S4报文恰好是待处理的，于是拷贝进用户内存。然后唤醒用户进程。

12、用户进程被唤醒了，当然唤醒后会先来拿到socket锁。以下执行又在进程上下文中了。

13、此时会检查已拷贝的字节数是否大于SO_RCVLOWAT，以及backlog队列是否为空。两者皆满足，准备返回。

14、释放socket锁，退出tcp_recvmsg方法。

15、返回用户已经复制的字节数S4-S1。

好了，这3个场景读完，想必大家对于TCP的接收流程是怎样的已经非常清楚了，本文起始的6个问题也在这一大篇中都涉及到了。下一篇我们来讨论TCP连接的关闭。