

## 第30章 TCP的用户需求

### 30.1 引言

本章介绍TCP的用户请求处理函数 `tcp_usrreq`，它被协议的 `pr_usrreq` 函数调用，处理各种与TCP插口有关的系统调用。此外，还将介绍 `tcp_ctloutput`，应用进程调用 `setsockopt` 设定TCP插口选项时，会用到它。

### 30.2 `tcp_usrreq` 函数

TCP的用户请求函数用于处理多种操作。图 30-1给出了 `tcp_usrreq` 函数的基本框架，其中 `switch` 的语句体部分将在后续部分逐一展开。图 15-17中列出了函数的参数，其具体含义取决于所处理的用户请求。

---

*tcp\_usrreq.c*

```

45 int
46 tcp_usrreq(so, req, m, nam, control)
47 struct socket *so;
48 int req;
49 struct mbuf *m, *nam, *control;
50 {
51     struct inpcb *inp;
52     struct tcpcb *tp;
53     int s;
54     int error = 0;
55     int ostate;
56     if (req == PRU_CONTROL)
57         return (in_control(so, (int) m, (caddr_t) nam,
58             (struct ifnet *) control));
59     if (control && control->m_len) {
60         m_freem(control);
61         if (m)
62             m_freem(m);
63         return (EINVAL);
64     }
65     s = splnet();
66     inp = sotoinpcb(so);
67     /*
68      * When a TCP is attached to a socket, then there will be
69      * a (struct inpcb) pointed at by the socket, and this
70      * structure will point at a subsidiary (struct tcpcb).
71      */
72     if (inp == 0 && req != PRU_ATTACH) {
73         splx(s);
74         return (EINVAL);          /* XXX */
75     }
76     if (inp) {
77         tp = intotcpcb(inp);

```

图30-1 `tcp_usrreq` 函数体

```

78      /* WHAT IF TP IS 0? */
79      ostate = tp->t_state;
80  } else
81      ostate = 0;
82  switch (req) {

      /* switch cases */

276  default:
277      panic("tcp_usrreq");
278  }
279  if (tp && (so->so_options & SO_DEBUG))
280      tcp_trace(TA_USER, ostate, tp, (struct tcphdr *) 0, req);
281  splx(s);
282  return (error);
283 }

```

tcp\_usrreq.c

图30-1 (续)

### 1. in\_control处理ioctl请求

45-58 PRU\_CONTROL请求来自于ioctl系统调用，函数in\_control负责处理这一请求。

### 2. 控制信息无效

59-64 如果试图调用sendmsg，为TCP插口配置控制信息，代码将释放mbuf，并返回EINVAL差错代码，声明这一操作无效。

65-66 函数接着执行splnet。这种做法极为保守，因为并非在所有情况下都需要锁定，只是为了防止在case语句中单个地调用splnet。我们在图23-15中曾提到，调用splnet设定处理器的优先级，唯一的作用是阻止软中断执行IP输入处理(它会接着调用tcp\_input)，但无法阻止接口层接收输入数据分组并放入到IP的输入队列中。

通过指向插口结构的指针，可得到指向Internet PCB的指针。只有在应用进程调用socket系统调用，发出PRU\_ATTACH请求时，该指针才允许为空。

67-81 如果inp非空，当前连接状态将保存在ostate中，以备函数结束时可能会调用tcp\_trace。

下面我们开始讨论单独的case语句。应用进程调用socket系统调用，或者监听服务器收到连接请求(图28-7)，调用sonewconn函数时，都会发出PRU\_ATTACH请求，图30-2给出了这一请求的处理代码。

```

83      /*
84      * TCP attaches to socket via PRU_ATTACH, reserving space,
85      * and an internet control block.
86      */
87  case PRU_ATTACH:
88      if (inp) {
89          error = EISCONN;
90          break;
91      }
92      error = tcp_attach(so);
93      if (error)

```

tcp\_usrreq.c

图30-2 tcp\_usrreq 函数：PRU\_ATTACH 和PRU\_DETACH 请求

```

94         break;
95         if ((so->so_options & SO_LINGER) && so->so_linger != 0)
96             so->so_linger = TCP_LINGERTIME;
97         tp = sototcpb(so);
98         break;

99         /*
100        * PRU_DETACH detaches the TCP protocol from the socket.
101        * If the protocol state is non-embryonic, then can't
102        * do this directly: have to initiate a PRU_DISCONNECT,
103        * which may finish later; embryonic TCB's can just
104        * be discarded here.
105        */
106        case PRU_DETACH:
107            if (tp->t_state > TCPS_LISTEN)
108                tp = tcp_disconnect(tp);
109            else
110                tp = tcp_close(tp);
111            break;

```

tcp\_usrreq.c

图30-2 (续)

### 3. PRU\_ATTACH请求

83-94 如果插口结构已经指向某个PCB，则返回EISCONN差错代码。调用tcp\_attach完成处理：分配并初始化Internet PCB和TCP控制块。

95-96 如果选用了SO\_LINGER插口选项，且延迟时间为0，则将其设为120 (TCP\_LINGERTIME)。

为什么在PRU\_ATTACH请求发出之前，就可以设定插口选项？尽管不可能在调用socket之前就设定插口选项，但sonewconn也会发送PRU\_ATTACH请求。它在把监听插口的so\_options复制到新建插口之后，才会发送PRU\_ATTACH请求。此处的代码防止新建连接从监听插口中继承延迟时间为0的SO\_LINGER选项。

请注意，此处的代码有错误。常量TCP\_LINGERTIME在tcp\_timer.h中初始化为120，该行的注释为“最多等待2分钟”。但SO\_LINGER值也是内核tsleep函数(由soclose调用)的最后一个参数，从而成为内核的timeout函数的最后一个参数，单位为滴答，而非秒。如果系统的滴答频率(hz)等于100，则延迟时间将变为1.2秒，而非2分钟。

97 现在，tp已指向插口的TCP控制块。这样，如果选定了SO\_DEBUG插口选项，函数结束时就可以输出所需信息。

### 4. PRU\_DETACH请求

99-111 close系统调用在PRU\_DISCONNECT请求失败后，将发送PRU\_DETACH请求。如果连接尚未建立(连接状态小于ESTABLISHED)，则无需向对端发送任何信息。但如果连接已建立，则调用tcp\_disconnect初始化TCP的连接关闭过程(发送所有缓存中的数据，之后发送FIN)。

代码if语句的测试条件要求状态大于LISTEN，这是不正确的。因为如果连接状态等于SYN\_SENT或者SYN\_RCVD，两者都大于LISTEN，此时tcp\_disconnect会直接调用tcp\_close。实际上，这个case语句可以简化为直接调用tcp\_disconnect。

图30-3给出了bind和listen系统调用的处理代码。

```

112      /*
113      * Give the socket an address.
114      */
115      case PRU_BIND:
116          error = in_pcbbind(inp, nam);
117          if (error)
118              break;
119          break;

120      /*
121      * Prepare to accept connections.
122      */
123      case PRU_LISTEN:
124          if (inp->inp_lport == 0)
125              error = in_pcbbind(inp, (struct mbuf *) 0);
126          if (error == 0)
127              tp->t_state = TCPS_LISTEN;
128          break;

```

tcp\_usrreq.c

图30-3 tcp\_usrreq 函数：PRU\_BIND 和PRU\_LISTEN 请求

112-119 PRU\_BIND请求的处理只是简单地调用 in\_pcbbind。

120-128 对于PRU\_LISTEN请求，如果插口还未绑定在某个本地端口上，则调用 in\_pcbbind自动为其分配一个。这种情况十分少见，因为多数服务器会明确地绑定一个知名端口，尽管RPC(远端过程调用)服务器一般是绑定在一个临时端口上，并通过 Port Mapper 向系统注册该端口(卷1的29.4节介绍了Port Mapper)。连接状态变迁到LISTEN，完成了listen调用的主要目的：设定插口的状态，以便接受到达的连接请求(被动打开)。

图30-4给出了connect系统调用的处理代码：客户发起的主动打开。

```

129      /*
130      * Initiate connection to peer.
131      * Create a template for use in transmissions on this connection.
132      * Enter SYN_SENT state, and mark socket as connecting.
133      * Start keepalive timer, and seed output sequence space.
134      * Send initial segment on connection.
135      */
136      case PRU_CONNECT:
137          if (inp->inp_lport == 0) {
138              error = in_pcbbind(inp, (struct mbuf *) 0);
139              if (error)
140                  break;
141          }
142          error = in_pcbconnect(inp, nam);
143          if (error)
144              break;

145          tp->t_template = tcp_template(tp);
146          if (tp->t_template == 0) {
147              in_pcbdisconnect(inp);
148              error = ENOBUFS;

```

tcp\_usrreq.c

图30-4 tcp\_usrreq 函数：PRU\_CONNECT 请求

```

149         break;
150     }
151     /* Compute window scaling to request. */
152     while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
153           (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
154         tp->request_r_scale++;
155     soisconnecting(so);
156     tcpstat.tcps_connattempt++;
157     tp->t_state = TCPS_SYN_SENT;
158     tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;

159     tp->iss = tcp_iss;
160     tcp_iss += TCP_ISSINCR / 2;
161     tcp_sendseqinit(tp);

162     error = tcp_output(tp);
163     break;

```

tcp\_usrreq.c

图30-4 (续)

### 5. 分配临时端口

129-141 如果插口还未绑定在某个本地端口上，调用 `ip_pcbbind` 自动为其分配一个。对于客户端，这是很常见的，因为客户一般不关心本地端口值。

### 6. 连接PCB

142-144 调用 `in_pcbconnect`，获取到达目的地的路由，确定外出接口，验证插口对不重复。

### 7. 初始化IP和TCP首部

145-150 调用 `tcp_template` 分配 `mbuf`，保存IP和TCP的首部，并初始化两个首部，填入尽可能多的信息。会造成函数失败的唯一原因是内核耗尽了 `mbuf`。

### 8. 计算窗口缩放因子

151-154 计算用于接收缓存的窗口缩放因子：左移 65535(`TCP_MAXWIN`)，直到它大于或等于接收缓存的大小(`so_rcv.sb_hiwat`)。得到的位移次数(0~14之间)，就是需要在SYN中发送的缩放因子值(图28-7处理被动打开时，有相同的代码)。应用进程必须在调用 `connect` 之前，设定 `SO_RCVBUF` 插口选项，TCP才会在SYN中添加窗口大小选项，否则，将使用接收缓存大小的默认值(图24-3中的 `tcp_recvspace`)。

### 9. 设定插口和连接的状态

155-158 调用 `soisconnecting`，置位插口状态变量中恰当的比特，设定TCP连接状态为 `SYN_SENT`，从而在后续的 `tcp_output` 调用中发送SYN(参见图24-16的 `tcp_outlags` 值)。连接建立定时器启动，时限初始化为5秒。`tcp_output` 还会启动SYN的重传定时器，如图25-16所示。

### 10. 初始化序号

159-161 令初始序号等于全局变量 `tcp_iss`，之后令 `tcp_iss` 增加 64 000 (`TCP_ISSINCR` 除以2)。在监听服务器收到SYN并初始化ISS时(图28-17)，对 `tcp_iss` 的相同的操作。接着调用 `tcp_sendseqinit` 初始化发送序号。

### 11. 发送初始SYN

162 调用 `tcp_output` 发送初始SYN，以建立连接。如果 `tcp_output` 返回错误(例如，`mbuf` 耗尽或没有到达目的地的路由)，该差错代码将成为 `tcp_usrreq` 的返回值，报告给应用进程。

图30-5给出了PRU\_CONNECT2、PRU\_DISCONNECT和PRU\_ACCEPT请求的处理代码。

164-169 PRU\_CONNECT2请求，来自于socketpair系统调用，对TCP协议无效。

170-183 close系统调用会发送PRU\_DISCONNECT请求。如果连接已建立，应调用tcp\_disconnect，发送FIN，执行正常的TCP关闭操作。

```

164      /*
165      * Create a TCP connection between two sockets.
166      */
167      case PRU_CONNECT2:
168          error = EOPNOTSUPP;
169          break;

170      /*
171      * Initiate disconnect from peer.
172      * If connection never passed embryonic stage, just drop;
173      * else if don't need to let data drain, then can just drop anyway,
174      * else have to begin TCP shutdown process: mark socket disconnecting,
175      * drain unread data, state switch to reflect user close, and
176      * send segment (e.g. FIN) to peer. Socket will be really disconnected
177      * when peer sends FIN and acks ours.
178      *
179      * SHOULD IMPLEMENT LATER PRU_CONNECT VIA REALLOC TCPCB.
180      */
181      case PRU_DISCONNECT:
182          tp = tcp_disconnect(tp);
183          break;

184      /*
185      * Accept a connection. Essentially all the work is
186      * done at higher levels; just return the address
187      * of the peer, storing through addr.
188      */
189      case PRU_ACCEPT:
190          in_setpeeraddr(inp, nam);
191          break;

```

tcp\_usrreq.c

图30-5 tcp\_usrreq 函数：PRU\_CONNECT2、PRU\_DISCONNECT 和PRU\_ACCEPT 请求

请注意以“应该实现”起头的注释，这是因为无法接着使用出现错误的插口。

例如，客户调用connect，并得到一个错误，它就无法在同一个插口上再次调用connect，而必须首先关闭插口，调用socket创建新的插口，在新的插口上才能再次调用connect。

184-191 与accept系统调用有关的工作全部由插口层和协议层完成。PRU\_ACCEPT请求只简单地向应用进程返回对端的IP地址和端口号。

图30-6给出了PRU\_SHUTDOWN、PRU\_RCVD和PRU\_SEND请求的处理代码。

## 12. PRU\_SHUTDOWN请求

192-200 应用进程调用shutdown，禁止更多的输出时，soshutdown会发送PRU\_SHUTDOWN请求。调用socantsendmore置位插口的标志，禁止继续发送报文段。接着调用tcp\_usrclosed，根据图24-15的状态变迁图，设定正确的连接状态。tcp\_output发送FIN之前，如果发送缓存中仍有数据，会首先发送等待数据。

```

192      /*
193      * Mark the connection as being incapable of further output.
194      */
195      case PRU_SHUTDOWN:
196          socantsendmore(so);
197          tp = tcp_usrclosed(tp);
198          if (tp)
199              error = tcp_output(tp);
200          break;

201      /*
202      * After a receive, possibly send window update to peer.
203      */
204      case PRU_RCVD:
205          (void) tcp_output(tp);
206          break;

207      /*
208      * Do a send by putting data in output queue and updating urgent
209      * marker if URG set.  Possibly send more data.
210      */
211      case PRU_SEND:
212          sbappend(&so->so_snd, m);
213          error = tcp_output(tp);
214          break;

```

tcp\_usrreq.c

图30-6 tcp\_usrreq 函数：PRU\_SHUTDOWN、PRU\_RCVD 和 PRU\_SEND 请求

## 13. PRU\_RCVD请求

201-206 应用进程从插口的接收缓存中读取数据后，soreceive会发送这个请求。此时接收缓存已扩大，也许会有足够的空间，让TCP发送更大的窗口通告。tcp\_output会决定是否发送窗口更新报文段。

## 14. PRU\_SEND请求

207-214 图23-14中给出的5个写函数，都以这一请求结束。调用sbappend，向插口的发送缓存中添加数据（它将一直保存在缓存中，直到被确认），并调用tcp\_output发送新报文段（如果条件允许）。

图30-7给出了PRU\_ABORT和PRU\_SENSE请求的处理代码。

```

215      /*
216      * Abort the TCP.
217      */
218      case PRU_ABORT:
219          tp = tcp_drop(tp, ECONNABORTED);
220          break;

221      case PRU_SENSE:
222          ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;
223          (void) splx(s);
224          return (0);

```

tcp\_usrreq.c

图30-7 tcp\_usrreq 函数：PRU\_ABORT 和 PRU\_SENSE 请求

## 15. PRU\_ABORT请求

215-220 如果插口是监听插口（如服务器），并且存在等待建立的连接，例如已发送初始



SYN或已完成三次握手过程，但还未被服务器 `accept` 的连接，调用 `sockclose` 会导致发送 `PRU_ABORT` 请求。如果连接已同步，`tcp_drop` 将发送 RST。

#### 16. `PRU_SENSE` 请求

221-224 `fstat` 系统调用会生成 `PRU_SENSE` 请求。TCP 返回发送缓存的大小，保存在 `stat` 结构的成员变量 `st_blksize` 中。

图30-8给出了 `PRU_RCVOOB` 的处理代码。当应用进程置位 `MSG_OOB` 标志，试图读取带外数据时，`soreceive` 会发送这一请求。

```

225     case PRU_RCVOOB:
226         if ((so->so_oobmark == 0 &&
227             (so->so_state & SS_RCVATMARK) == 0) ||
228             so->so_options & SO_OOBINLINE ||
229             tp->t_oobflags & TCPOOB_HADDATA) {
230             error = EINVAL;
231             break;
232         }
233         if ((tp->t_oobflags & TCPOOB_HAVEDATA) == 0) {
234             error = EWOULDBLOCK;
235             break;
236         }
237         m->m_len = 1;
238         *mtod(m, caddr_t) = tp->t_iobc;
239         if (((int) nam & MSG_PEEK) == 0)
240             tp->t_oobflags ^= (TCPOOB_HAVEDATA | TCPOOB_HADDATA);
241         break;

```

*tcp\_usrreq.c*

*tcp\_usrreq.c*

图30-8 `tcp_usrreq` 函数：`PRU_RCVOOB` 请求

#### 17. 能否读取带外数据

225-232 如果下列3个条件有一个为真，应用进程读取带外数据的努力就会失败。

1) 如果插口的带外数据分界点 (`so_oobmark`) 等于0，并且插口的 `SS_RCVATMARK` 标志未设定；或者

2) 如果 `SO_OOBINLINE` 插口选项设定；或者

3) 如果连接的 `TCPOOB_HADDATA` 标志设定 (例如，连接的带外数据已被读取)。

如果上述3个条件中任何一个为真，则返回差错代码 `EINVAL`。

#### 18. 是否有带外数据到达

233-236 如果上述3个条件全假，但 `TCPOOB_HAVEDATA` 标志置位，说明尽管 TCP 已收到了对端发送的紧急方式通告，但尚未收到序号等于紧急指针减 1 的字节 (图29-17)，此时返回差错代码 `EWOULDBLOCK`，有可能因为发送方发送紧急数据通告时，紧急数据偏移量指向了尚未发送的字节。卷1的图26-7举例说明了这种情况，发送方的数据传输被对端的零窗口通告停止时，常出现这种现象。

#### 19. 返回带外数据字节

237-238 `tcp_pulloutofband` 向应用进程返回存储在 `t_iobc` 中的一个字节的带外数据。

#### 20. 更新标志

239-241 如果应用进程已读取了带外数据 (而不是仅大致了解带外数据的情况，`MSG_PEEK` 标志置位)，TCP 清除 `HAVE` 标志，并置位 `HAD` 标志。`case` 语句执行到此处时，通过前面的代码可



以肯定，HAVE标志已置位，而HAD标志被清除。置位HAD标志的目的是防止应用进程试图再次读取带外数据。一旦HAD标志置位，在收到新的紧急指针之前，它不会被清除(图29-17)。

代码使用了让人费解的异或运算，而不是简单的

```
tp->t_oobflags = TCPOOB_HADDATA;
```

是为了能够在t\_oobflags中定义更多的比特。但Net/3中，实际只用到了上面提及的两个标志比特。

图30-9中的PRU\_SENDOOB请求，是在应用进程写入数据并置位MSG\_OOB时，由sosend发送的。

```

242     case PRU_SENDOOB:                                     tcp_usrreq.c
243         if (sbospace(&so->so_snd) < -512) {
244             m_freem(m);
245             error = ENOBUFS;
246             break;
247         }
248         /*
249          * According to RFC961 (Assigned Protocols),
250          * the urgent pointer points to the last octet
251          * of urgent data. We continue, however,
252          * to consider it to indicate the first octet
253          * of data past the urgent section.
254          * Otherwise, snd_up should be one lower.
255          */
256         sbappend(&so->so_snd, m);
257         tp->snd_up = tp->snd_una + so->so_snd.sb_cc;
258
259         tp->t_force = 1;
260         error = tcp_output(tp);
261         tp->t_force = 0;
262
263         break;

```

图30-9 tcp\_usrreq 函数：PRU\_SENDOOB 请求

## 21. 确认发送缓存中有足够空间并添加新数据

242-247 发送带外数据时，允许应用进程写入数据后，待发送数据量超过发送缓存大小，超出量最多为512字节。插口层的限制要宽松一些，写入带外数据后，最多可超出发送缓存1024字节(图16-24)。调用sbappend向发送缓存末端添加数据。

## 22. 计算紧急指针

248-257 紧急指针(snd\_up)指向写入的最后一个字节之后的字节。图26-30举例说明了这一点，假定发送缓存为空，应用进程写入3字节的数据，且置位了MSG\_OOB标志。这是考虑到若应用进程置位MSG\_OOB标志，且写入的数据量超过1字节，如果接收方为伯克利系统，则只有最后一个字节会被认为是带外数据。

## 23. 强制TCP输出

258-261 令t\_force等于1，并调用tcp\_output。即使收到了对端的零窗口通告，TCP也会发送报文段，URG标志置位，紧急指针偏移量非零。卷1的图26-7说明了如何向一个关闭的接收窗口发送紧急报文段。

图30-10给出了最后3个请求的处理。

```

262     case PRU_SOCKADDR:
263         in_setsockaddr(inp, nam);
264         break;

265     case PRU_PEERADDR:
266         in_setpeeraddr(inp, nam);
267         break;

268     /*
269      * TCP slow timer went off; going through this
270      * routine for tracing's sake.
271      */
272     case PRU_SLOWTIMO:
273         tp = tcp_timers(tp, (int) nam);
274         req |= (int) nam << 8; /* for debug's sake */
275         break;

```

tcp\_usrreq.c

图30-10 tcp\_usrreq 函数：PRU\_SOCKADDR、PRU\_PEERADDR 和 PRU\_SLOWTIMO 请求

262-267 getsockname和getpeername系统调用分别发送 PRU\_SOCKADDR和 PRU\_PEERADDR请求。调用in\_setsockaddr和in\_setpeeraddr函数，从PCB中获取需要信息，存储在addr参数中。

268-275 执行tcp\_slowtimo函数会发送 PRU\_SLOWTIMO函数。如同注释所指出的，tcp\_slowtimo不直接调用 tcp\_timers的唯一原因是为了能够在函数结尾处调用 tcp\_trace，跟踪记录定时器超时事件(图30-1)。为了在记录中指明是4个TCP定时器中的哪一个超时，tcp\_slowtimo通过nam参数传递了t\_timer数组(图25-1)的指针，并左移8位后与请求值(req)逻辑或。trpt程序了解这种做法，并据此完成相应的处理。

### 30.3 tcp\_attach函数

tcp\_attach函数，在处理PRU\_ATTACH请求(例如，插口系统调用，或者监听插口上收到了新的连接请求)时，由tcp\_usrreq调用。图30-11给出了它的代码。

#### 1. 为发送缓存和接收缓存分配资源

361-372 如果还未给插口的发送和接收缓存分配空间，sbreserve将两者都设为8192，即全局变量tcp\_sendspace和tcp\_recvspace的默认值(图24-3)。

这些默认值是否够用，取决于连接两个传输方向上的 MSS，后者又取决于 MTU。

例如，[Comer and lin 1994]论证了，如果发送缓存小于3倍的MSS，则会出现异常，严重降低系统性能。某些实现定义的默认值很大，如 61 444字节，已考虑到这些默认值对性能的影响，尤其对较大的 MTU(如FDDI和ATM)更是如此。

#### 2. 分配Internet PCB和TCP控制块

373-377 in\_pcballoc分配Internet PCB，而tcp\_newtcpcb分配TCP控制块，并将其与对应的PCB相连。

378-384 如果tcp\_newtcpcb调用malloc时失败，则执行注释为“XXX”的代码。前面已介绍过，PRU\_ATTACH请求是插口系统调用或监听插口收到新的连接请求(sonewconn)的结果。对于后一种情况，插口标志 SS\_NOFDREF置位。如果此标志置位，in\_pcballoc调用sofree时会释放插口结构。但我们在 tcp\_input中看到，除非该函数已完成接收报文段

的处理(图29-27中的 `dropsocket` 标志), 否则, 不应释放插口结构。因此, 调用 `in_pcbdetach` 时, 应将 `SS_NOFDREF` 标志的当前值保存在变量 `nofd` 中, 并在 `tcp_attach` 返回前重设该标志。

385-386 TCP连接状态初始化为CLOSED。

```

361 int
362 tcp_attach(so)
363 struct socket *so;
364 {
365     struct tcpcb *tp;
366     struct inpcb *inp;
367     int error;

368     if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
369         error = sorereserve(so, tcp_sendspace, tcp_recvspace);
370         if (error)
371             return (error);
372     }
373     error = in_pcballoc(so, &tp);
374     if (error)
375         return (error);
376     inp = sotoinpcb(so);
377     tp = tcp_newtcpcb(inp);
378     if (tp == 0) {
379         int nofd = so->so_state & SS_NOFDREF; /* XXX */

380         so->so_state &= ~SS_NOFDREF; /* don't free the socket yet */
381         in_pcbdetach(inp);
382         so->so_state |= nofd;
383         return (ENOBUFS);
384     }
385     tp->t_state = TCPS_CLOSED;
386     return (0);
387 }

```

*tcp\_usrreq.c*

图30-11 `tcp_attach` 函数：创建新的TCP插口

## 30.4 `tcp_disconnect`函数

图30-12给出的 `tcp_disconnect` 函数, 准备断开TCP连接。

### 1. 连接未同步

396-402 如果连接还未进入 ESTABLISHED 状态(如 LISTEN、SYN\_SENT 或 SYN\_RCVD), `tcp_close` 只释放 PCB 和 TCP 控制块。无需向对端发送任何报文段, 因为连接尚未同步。

### 2. 硬性断开

403-404 如果连接已同步, 且 `SO_LINGER` 插口选项置位, 拖延时间(`SO_LINGER`)设为零, 则调用 `tcp_drop` 丢弃连接。连接不经过 `TIME_WAIT`, 直接更新为 `CLOSED`, 向对端发送 RST, 释放 PCB 和 TCP 控制块。调用 `close` 会发送 `PRU_DISCONNECT` 请求, 丢弃仍在发送或接收缓存中的任何数据。

如果 `SO_LINGER` 插口选项置位, 且拖延时间非零, 则调用 `soclose` 进行处理。

### 3. 平滑断开

405-406 如果连接已同步, 且 `SO_LINGER` 选项未设定, 或者选项设定且拖延时间不为零,

则执行TCP正常的连接终止步骤。soisdisconnecting设定插口状态。

```

396 struct tcpcb *
397 tcp_disconnect(tp)
398 struct tcpcb *tp;
399 {
400     struct socket *so = tp->t_inpcb->inp_socket;

401     if (tp->t_state < TCPS_ESTABLISHED)
402         tp = tcp_close(tp);
403     else if ((so->so_options & SO_LINGER) && so->so_linger == 0)
404         tp = tcp_drop(tp, 0);
405     else {
406         soisdisconnecting(so);
407         sbflush(&so->so_rcv);
408         tp = tcp_usrclosed(tp);
409         if (tp)
410             (void) tcp_output(tp);
411     }
412     return (tp);
413 }

```

tcp\_usrreq.c

tcp\_usrreq.c

图30-12 tcp\_disconnect 函数：准备断开TCP连接

#### 4. 丢弃滞留的接收数据

407 调用sbflush，丢弃所有滞留在接收缓存中的数据，因为应用进程已关闭了插口。发送缓存中的数据仍保留，tcp\_output将试图发送剩余的数据。我们说“试图”，因为不能保证数据还能成功地被发送。在收到并确认这些数据之前，对端可能已崩溃，即使对端的TCP模块能够接收并确认这些数据，在应用程序读取数据之前，系统也可能崩溃。因为本地进程已关闭了插口，即使TCP放弃发送仍滞留在发送缓存中的数据（因为重传定时器最终超时），也无法向应用进程通告错误。

#### 5. 改变连接状态

408-410 tcp\_usrclosed基于连接的当前状态，促使其进入下一状态。通常情况下，连接将转移到FIN\_WAIT\_1状态，因为连接关闭时一般都处于ESTABLISHED状态。后面会看到，tcp\_usrclosed通常返回当前控制块的指针(tp)。因为状态必须先同步才会执行此处的代码，所以总需要调用tcp\_output发送报文段。如果连接从ESTABLISHED转移到FIN\_WAIT\_2，将发送FIN。

### 30.5 tcp\_usrclosed函数

图30-13给出的这个函数，在PRU\_SHUTDOWN处理中，由tcp\_disconnect调用。

#### 1. 未收到SYN时的简单关闭

429-434 如果连接上还未收到SYN，则无需发送FIN。新的状态等于CLOSED，tcp\_close将释放Internet PCB和TCP控制块。

#### 2. 转移到FIN\_WAIT\_1状态

435-438 如果连接当前状态等于SYN\_RCVD和ESTABLISHED，新的状态将等于FIN\_WAIT\_1，再次调用tcp\_output时，将发送FIN(图24-16中的tcp\_outflags值)。

#### 3. 转移到LAST\_ACK状态

439-441 如果连接当前状态等于 CLOSE\_WAIT，新状态等于 LAST\_ACK，则再次调用 tcp\_output 时，将发送 FIN。

443-444 如果连接当前状态等于 FIN\_WAIT\_2 或 TIME\_WAIT，soisdisconnected 将正确地标注插口的状态。

```

424 struct tcpcb *
425 tcp_usrclosed(tp)
426 struct tcpcb *tp;
427 {
428     switch (tp->t_state) {
429     case TCPS_CLOSED:
430     case TCPS_LISTEN:
431     case TCPS_SYN_SENT:
432         tp->t_state = TCPS_CLOSED;
433         tp = tcp_close(tp);
434         break;
435     case TCPS_SYN_RECEIVED:
436     case TCPS_ESTABLISHED:
437         tp->t_state = TCPS_FIN_WAIT_1;
438         break;
439     case TCPS_CLOSE_WAIT:
440         tp->t_state = TCPS_LAST_ACK;
441         break;
442     }
443     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
444         soisdisconnected(tp->t_inpcb->inp_socket);
445     return (tp);
446 }

```

tcp\_usrreq.c

图30-13 tcp\_usrclosed 函数：基于连接关闭的处理进程，将连接转移到下一状态

## 30.6 tcp\_ctloutput 函数

tcp\_ctloutput 函数被 getsockopt 和 setsockopt 函数调用，如果它们的描述符参数指明了一个 TCP 插口，且 level 不是 SOL\_SOCKET。图30-14列出了 TCP 支持的两个插口选项。

选项名	变量	存取	描述
TCP_NODELAY	t_flags	读、写	Nagel算法(图26-8)
TCP_MAXSEG	t_maxseg	读、写	TCP将发送的最大报文段长度

图30-14 TCP支持的插口选项

图30-15给出了函数的第一部分。

```

284 int
285 tcp_ctloutput(op, so, level, optname, mp)
286 int op;
287 struct socket *so;
288 int level, optname;

```

tcp\_usrreq.c

图30-15 tcp\_ctloutput 函数：第一部分

```

289 struct mbuf **mp;
290 {
291     int     error = 0, s;
292     struct inpcb *inp;
293     struct tcpcb *tp;
294     struct mbuf *m;
295     int     i;

296     s = splnet();
297     inp = sotoinpcb(so);
298     if (inp == NULL) {
299         splx(s);
300         if (op == PRCO_SETOPT && *mp)
301             (void) m_free(*mp);
302         return (ECONNRESET);
303     }
304     if (level != IPPROTO_TCP) {
305         error = ip_ctloutput(op, so, level, optname, mp);
306         splx(s);
307         return (error);
308     }
309     tp = intotcpcb(inp);

```

tcp\_usrreq.c

图30-15 (续)

296-303 函数执行时，处理器优先级设为 `splnet`，`inp` 指向插口的 Internet PCB。如果 `inp` 为空，且操作类型是设定插口选项，则释放 `mbuf` 并返回错误。

304-308 如果 `level` (`getsockopt` 和 `setsockopt` 系统调用的第二个参数) 不等于 `IPPROTO_TCP`，说明操作的是其他协议 (如 IP)。例如，可以创建一个 TCP 插口，并设定其 IP 源选路插口选项。此时应由 IP 处理这个插口选项，而不是 TCP。`ip_ctloutput` 处理命令。

309 如果是对 TCP 选项进行操作，`tp` 将指向 TCP 控制块。

函数的剩余部分是一个 `switch` 语句，有两个分支：一个处理 `PRCO_SETOPT` (图30-16中给出)，另一个处理 `PRCO_GETOPT` (图30-17中给出)。

tcp\_usrreq.c

```

310     switch (op) {
311     case PRCO_SETOPT:
312         m = *mp;
313         switch (optname) {
314         case TCP_NODELAY:
315             if (m == NULL || m->m_len < sizeof(int))
316                 error = EINVAL;
317             else if (*mtod(m, int *))
318                 tp->t_flags |= TF_NODELAY;
319             else
320                 tp->t_flags &= ~TF_NODELAY;
321             break;
322         case TCP_MAXSEG:
323             if (m && (i = *mtod(m, int *)) > 0 && i <= tp->t_maxseg)
324                 tp->t_maxseg = i;
325             else
326                 error = EINVAL;
327             break;

```

图30-16 `tcp_ctloutput` 函数：设定插口选项

```

328         default:
329             error = ENOPROTOOPT;
330             break;
331     }
332     if (m)
333         (void) m_free(m);
334     break;

```

—tcp\_usrreq.c

图30-16 (续)

315-316 m是一个mbuf，保存了setsockopt的第四个参数。对于两个TCP插口选项，mbuf中都必须都是整数。如果任何一个mbuf指针为空，或者mbuf中的数据长度小于整数大小，则返回错误。

#### 1. TCP\_NODELAY选项

317-321 如果整数值非零，则置位TF\_NODELAY标志，从而取消图26-8中的Nagle算法。如果整数值等于0，则使用Nagle算法(默认值)，并清除TF\_NODELAY标志。

#### 2. TCP\_MAXSEG选项

322-327 应用进程只能减少MSS。TCP插口创建时，tcp\_newtcpcb初始化t\_maxseg为默认值512。当收到对端SYN中包含的MSS选项时，tcp\_input调用tcp\_mss，t\_maxseg最高可等于外出口的MTU(减去40字节，IP和TCP首部的默认值)，以太网等于1460。因此，调用插口之后，连接建立之前，应用进程只能以默认值512为起点，减少MSS。连接建立后，应用进程可以从tcp\_mss选取的任何值起，减少MSS。

4.4BSD是伯克利版本中第一次支持MSS做为插口选项，以前的版本只允许利用getsockopt读取MSS值。

#### 3. 释放mbuf

332-333 释放mbuf链。

图30-17给出了PRCO\_GETOPT命令的处理。

```

335     case PRCO_GETOPT:
336         *mp = m = m_get(M_WAIT, MT_SOOPTS);
337         m->m_len = sizeof(int);
338
339     switch (optname) {
340     case TCP_NODELAY:
341         *mtod(m, int *) = tp->t_flags & TF_NODELAY;
342         break;
343     case TCP_MAXSEG:
344         *mtod(m, int *) = tp->t_maxseg;
345         break;
346     default:
347         error = ENOPROTOOPT;
348         break;
349     }
350     splx(s);
351     return (error);
352 }

```

—tcp\_usrreq.c

—tcp\_usrreq.c

图30-17 tcp\_ctloutput 函数：读取插口选项



335-337 两个TCP插口选项都向应用进程返回一个整数值，因此，调用 `m_get` 得到一个 `mbuf`，其长度等于整数长度。

339-341 `TCP_NODELAY` 返回 `TF_NODELAY` 标志的当前状态：如果标志未置位（使用 Nagel 算法），则等于 0；如果标志置位，则等于 `TF_NODELAY`。

342-344 `TCP_MAXSEG` 选项返回 `t_maxseg` 的当前值。前面讨论 `PRCO_SETOPT` 命令时曾提到，返回值取决于插口是否已进入连接状态。

## 30.7 小结

`tcp_usrreq` 函数处理逻辑很简单，因为绝大多数处理都由其他函数完成。`PRU_xxx` 请求是独立于协议的系统调用与 TCP 协议处理间的桥梁。

`tcp_ctlsocketopt` 函数也很简单，因为 TCP 只支持两个插口选项：使用或取消 Nagel 算法，设置或读取最大报文段长度。

## 习题

- 30.1 现在，我们已经结束了对 TCP 的讨论，如果某个客户执行了正常的 `socket`、`connect`、`write`（向服务器请求）和 `read`（读取服务器响应），分别列出客户端和服务器的处理步骤及 TCP 状态变迁。
- 30.2 如果应用进程设定 `SO_LINGER` 插口选项，且拖延时间等于 0，之后调用 `close`，我们给出了如何调用 `tcp_disconnect`，从而发送 RST。如果应用进程设定了这个插口选项，且拖延时间等于 0，之后进程被某个信号杀死（kill），而非调用 `close`，会发生什么？还会发送 RST 报文段吗？
- 30.3 图 25-4 中描述 `TCP_LINGERTIME` 时，称之为“`SO_LINGER` 插口选项的最大秒数”。根据图 30-2 中的代码，这个说法正确吗？
- 30.4 某个 Net/3 客户调用 `socket` 和 `connect`，主动与服务器建立连接，使用了客户的默认路由。客户主机向服务器发送了 1 129 个报文段。假定到达目的地的路由未变，为了这条连接，客户主机需要搜索多少次路由表？解释你的结论。
- 30.5 找到卷 1 的附录 C 中提到的 `sock` 程序。把该程序做为服务器运行，读取数据前有停顿（-p），且有较大的接收缓存。之后在另一个系统中运行同一个程序，但做为客户。通过 `tcpdump` 查看数据。确认 TCP “确认所有其他报文段”的属性未出现，服务器送出的 ACK 全部是延迟 ACK。
- 30.6 修改 `SO_KEEPAALIVE` 插口选项，从而能够配置每个连接的参数。
- 30.7 阅读 RFC 1122，了解为什么它建议 TCP 应该允许 RST 报文段携带数据。修改 Net/3 代码以实现此功能。