第20章 选路插口

20.1 引言

一个进程使用选路域(routing domain)中的一个插口来发送和接收前一章所描述的选路报文。socket系统调用需要指定一个PF_ROUTE的族类型和一个SOCK_RAW的插口类型。

接着，该进程可以向内核发送以下五种选路报文：

1) RTM_ADD：增加一条新路由。

2) RTM_DELETE：删除一条已经存在的路由。

3) RTM_GET：取得有关一条路由的所有信息。

4) RTM_CHANGE：改变一条已经存在路由的网关、接口或者度量。

5) RTM_LOCK：说明内核不应该修改哪个度量。

除此之外，该进程可以接收其他七个选路报文，这些报文是在发生某些事件时，如接口下线和收到重定向报文等等，由内核生成的。

本章简介选路域、为每个选路插口创建的选路控制块、处理进程产生的报文的函数(route_output)、发送选路报文给一个或多个进程的函数(raw_input)、以及不同的支持一个选路插口上所有插口操作的函数。

20.2 routedomain和protosw结构

在描述选路插口函数之前，我们需要讨论有关选路域的其他一些细节；在选路域中支持的SOCK_RAW协议；以及每个选路插口所附带的选路控制块。

图20-1列出了称为routedomain的PF_ROUTE域的domain结构。

成 员	值	描 述
dom_family	*PF_ROUTE*	域的协议族
dom_name	*route*	名字
dom_init	*route_init*	域的初始化，图18-30
dom_externalize	*0*	在选路域中不使用
dom_dispose	*0*	在选路域中不使用
dom_protosw	*routesw*	协议交换结构，图20-2
dom_protoswNPROTOSW		指向协议交换结构之后的指针
dom_next		由domaininit填入，图7-15
dom_rtattch	*0*	在选路域中不使用
dom_rtoffset	*0*	在选路域中不使用
dom_maxrtkey	*0*	在选路域中不使用

图20-1 routedomain 结构

与支持多个协议(TCP、UDP和ICMP等)的Internet域不一样，在选路域中只支持SOCK_RAW类型的一种协议。图20-2列出了PF_ROUTE域的协议转换项。

成 员	routesw[0]	描 述
pr_type	*SOCK_RAW*	原始插口
pr_domain	*&routedomain*	选路域部分
pr_protocol	*0*	
pr_flags	*PR_ATOMI|PR_ADDR*	插口层标志，协议处理时不使用
pr_input	*raw_input*	不使用这项，raw_input直接调用
pr_output	*route_output*	PRU_SEND请求所调用
pr_ctlinput	*raw_ctlinput*	控制输入函数
pr_ctloutput	*0*	不使用
pr_usrreq	*route_usrreq*	对一个进程通信请求的响应
pr_init	*raw_init*	初始化
pr_fasttimo	*0*	不使用
pr_slowtimo	*0*	不使用
pr_drain	*0*	不使用
pr_sysctl	*sysctl_rtable*	用于sysctl(8)系统调用

图20-2 选路协议protosw 的结构

20.3 选路控制块

每当采用如下形式的调用创建一个选路插口时，

```
socket(PF_ROUTE, SOCK_RAW, protocol);
```

对协议的用户请求函数(route_usrreq)的一个对应的PRU_ATTACH请求分配一个选路控制块，并且将它链接到插口结构上。 *protocol*参数可以将发送给这个插口上的进程的报文类型限制为一个特定族。例如，如果将 *protocol*参数说明为AF_INET，只有包含了Internet地址的选路报文将被发送给这个进程。 *protocol*参数为0将使得来自内核的所有选路报文都发送给这个插口。

记住我们把这些结构称为选路控制块，而不是原始控制块 (raw control block)，是为了避免与第32章中的原始IP控制块相混淆。

图20-3显示了rawcb结构的定义。

raw_cb.h

```
39 struct rawcb {
40     struct rawcb *rcb_next;     /* doubly linked list */
41     struct rawcb *rcb_prev;
42     struct socket *rcb_socket;  /* back pointer to socket */
43     struct sockaddr *rcb_faddr; /* destination address */
44     struct sockaddr *rcb_laddr; /* socket's address */
45     struct sockproto rcb_proto; /* protocol family, protocol */
46 };

47 #define sotorawcb(so)       ((struct rawcb *)(so)->so_pcb)
```

raw_cb.h

图20-3 rawcb 结构

另外，分配了一个相同名字的全局结构， rawcb，作为这个双向链表的头。图 20-4显示了这种情况。

39-47 我们在图19-26中显示了sockproto的结构。它的sp_family成员变量被设置为PF_ROUTE，sp_protocol成员变量被设置为 socket系统调用的第三个参数。

rcb_faddr成员变量被永久性地设置为指向route_src的指针，我们在图19-26中描述了route_src。rcb_laddr总是一个空指针。

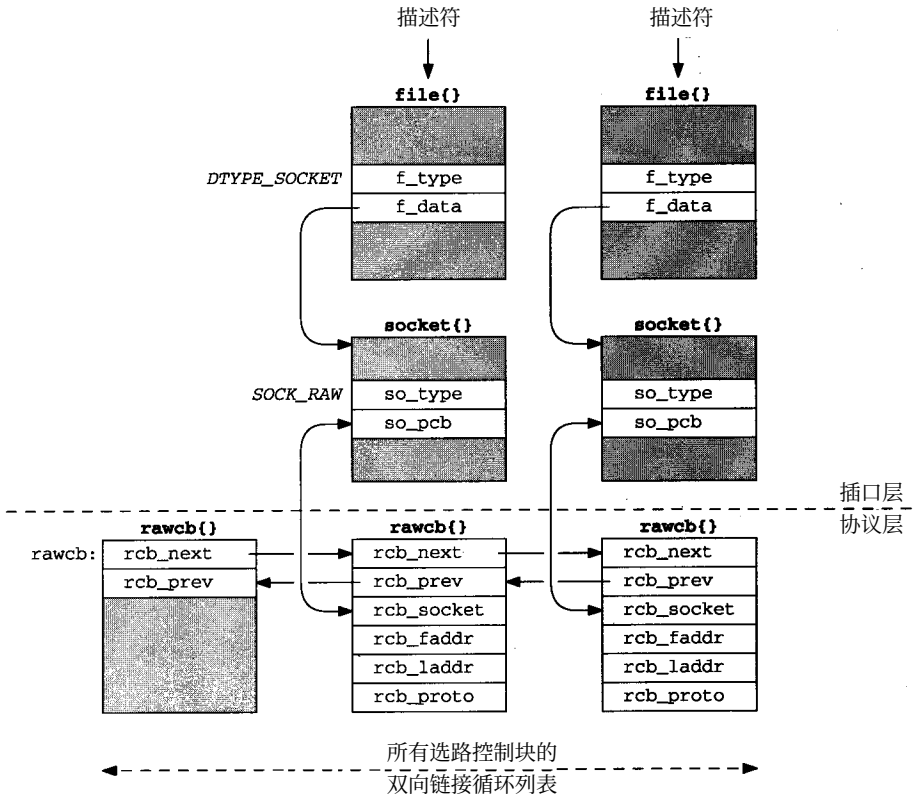

图20-4 原始协议控制块与其他数据结构的关系

20.4 raw_init函数

在图20-5中显示的raw_init函数是图20-2中的protosw结构的协议初始化函数。我们在图18-29中描述了选路域的完整初始化过程。

38-42 这个函数将头结构的下一个和前一个指针设置为指向自身来对这个双向链表进行初始化。

raw_usrreq.c

```
38 void
39 raw_init()
40 {
41     rawcb.rcb_next = rawcb.rcb_prev = &rawcb;
42 }
```

raw_usrreq.c

图20-5 raw_init 函数：初始化选路控制块的双向链表

20.5 route_output函数

如同我们在图18-11所显示的，当给协议的用户请求函数发送 PRU_SEND请求时，就会调

用route_output，这是一个进程向一个选路插口进行写操作所引起的。在图 18-9中，我们给出了内核接受的、由进程发出的五种不同类型的选路报文。

因为这个函数是由一个进程的写操作激活的，来自于该进程的数据 (发送给进程的选路报文)被放在一个由sosend开始的mbuf链中。图20-6显示了大概的处理步骤，假定进程发送了一个RTM_ADD命令，说明三个地址：目的地址、它的网关和一个网络掩码 (因此，这是一个网络路由，而不是一个主机路由)。

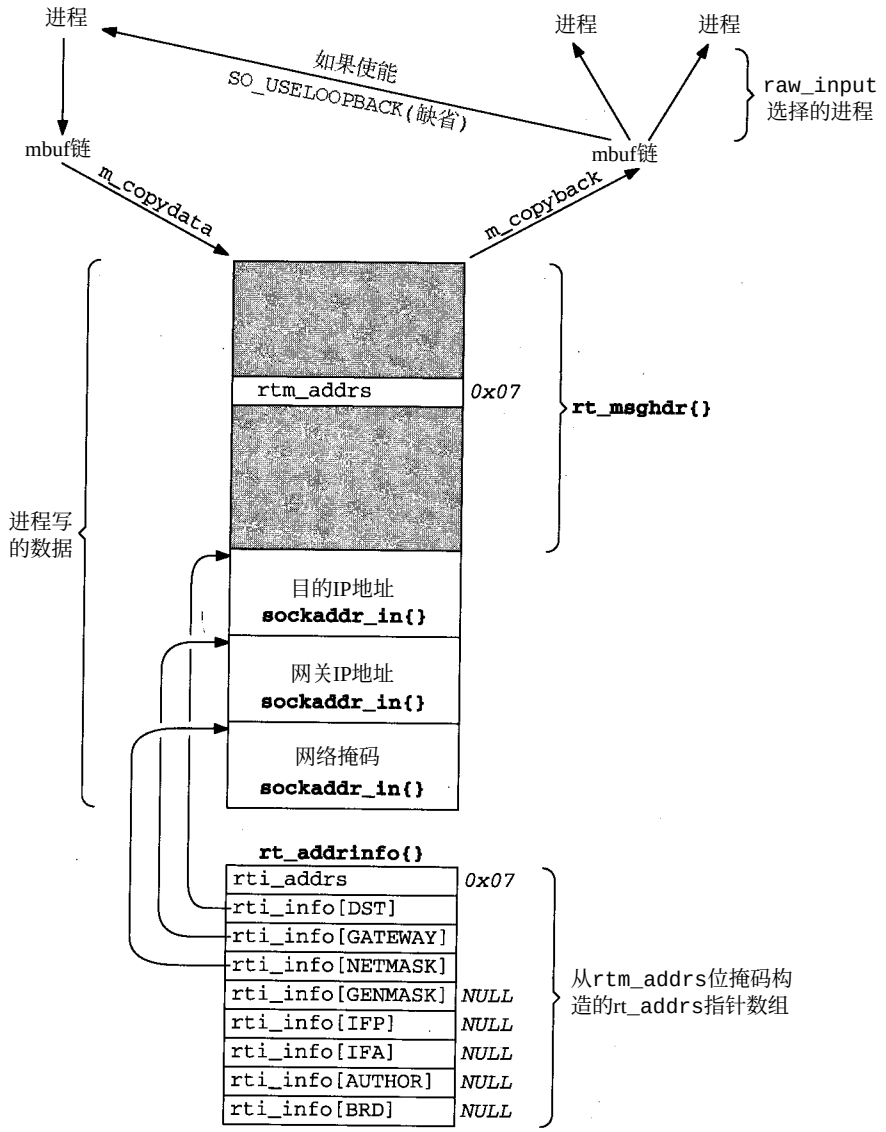

图20-6 一个进程发出的RTM_ADD 命令的处理过程示例

在这个图中有几点需要引起注意，我们在介绍 route_output的源代码时讨论了这里需要注意的大多数情况。另外，为了节省空间，我们省略了 rt_addrinfo结构中每个数组下标的RTAX_前缀。

• 进程通过设置比特掩码rtm_addrs来说明在定长的rt_msghdr结构之后有哪些插口地

址结构。我们显示了一个值为0x07的比特掩码，表示有一个目的地址、一个网关地址和一个网络掩码(图19-19)。RTM_ADD命令需要前两个地址；第三个地址是可选的。另一个可选的地址，genmask说明了用来产生克隆路由的掩码。

- write系统调用(sosend函数)将来自进程的一个缓存数据复制到内核的一个mbuf链中。
- m_copydata将mbuf链中数据复制到route_output使用malloc获得的一个缓存中。访问存储在单个连续缓存中的结构以及结构后面的若干插口地址结构，比访问一个mbuf链更容易。
- route_output调用rt_xaddrs函数取得比特掩码，并且构造一个指向缓存的rt_addrinfo结构。route_output中的代码使用图19-19中第五栏显示的名字来引用这些结构。比特掩码也要复制到rti_addrs成员中。
- route_output一般要修改rt_msghdr结构。如果发生了一个错误，相应的errno值被返回到rtm_errno中(例如，如果路由已经存在，则返回EEXIST)；否则，RTF_DONE标志与进程提供的rtm_flags执行逻辑或操作。
- rt_msghdr结构以及接着的地址成为0个或多个正在读选路插口的进程的输入。首先使用m_copyback将缓存转换为一个mbuf链。raw_input经过所有的选路PCB，并且传递一个复制给对应的进程。我们还显示了一个带有选路插口的进程，如果该进程没有禁止SO_USELOOPBACK插口选项，就会收到它写给那个插口的每个报文的一个复制。

为了避免收到它们自己的选路报文的一个复制，有些程序，如route，将第二个参数置为0来调用shutdown，以防止从选路插口上收到任何数据。

我们分成七个部分分析route_output的源代码。图20-7显示了这个函数的大概流程。

```
int
route_output()
{
    R_Malloc() to allocate buffer;
    m_copydata() to copy from mbuf chain into buffer;
    rt_xaddrs() to build rt_addrinfo{};

    switch (message type) {
    case RTM_ADD:
        rtrequest(RTM_ADD);
        rt_setmetrics();
        break;
    case RTM_DELETE:
        rtrequest(RTM_DELETE);
        break;

    case RTM_GET:
    case RTM_CHANGE:
    case RTM_LOCK:
        rtalloc1();

        switch (message type) {
        case RTM_GET:
            rt_msg2(RTM_GET);
            break;

        case RTM_CHANGE:
```

图20-7 route_output 处理步骤小结

```
            change appropriate fields;
            /* fall through */

        case RTM_LOCK:
            set rmx_locks;
            break;
        }
        break;
    }

    set rtm_error if error, else set RTF_DONE flag;

    m_copyback() to copy from buffer into mbuf chain;

    raw_input();    /* mbuf chain to appropriate processes */
}
```

图20-7 (续)

route_output的第一部分显示在图20-8中。

rtsock.c

```
113 int
114 route_output(m, so)
115 struct mbuf *m;
116 struct socket *so;
117 {
118     struct rt_msghdr *rtm = 0;
119     struct rtentry *rt = 0;
120     struct rtentry *saved_nrt = 0;
121     struct rt_addrinfo info;
122     int     len, error = 0;
123     struct ifnet *ifp = 0;
124     struct ifaddr *ifa = 0;

125 #define senderr(e) { error = e; goto flush;}
126     if (m == 0 || ((m->m_len < sizeof(long)) &&
127                    (m = m_pullup(m, sizeof(long))) == 0))
128                 return (ENOBUFS);
129     if ((m->m_flags & M_PKTHDR) == 0)
130         panic("route_output");
131     len = m->m_pkthdr.len;
132     if (len < sizeof(*rtm) ||
133         len != mtod(m, struct rt_msghdr *)->rtm_msglen) {
134         dst = 0;
135         senderr(EINVAL);
136     }
137     R_Malloc(rtm, struct rt_msghdr *, len);
138     if (rtm == 0) {
139         dst = 0;
140         senderr(ENOBUFS);
141     }
142     m_copydata(m, 0, len, (caddr_t) rtm);
143     if (rtm->rtm_version != RTM_VERSION) {
144         dst = 0;
145         senderr(EPROTONOSUPPORT);
146     }
147     rtm->rtm_pid = curproc->p_pid;

148     info.rti_addrs = rtm->rtm_addrs;
149     rt_xaddrs((caddr_t) (rtm + 1), len + (caddr_t) rtm, &info);
```

图20-8 route_output 函数：初始化处理，从mbuf链中复制报文

```
150     if (dst == 0)
151         senderr(EINVAL);
152     if (genmask) {
153         struct radix_node *t;
154         t = rn_addmask((caddr_t) genmask, 1, 2);
155         if (t && Bcmp(genmask, t->rn_key, *(u_char *) genmask) == 0)
156             genmask = (struct sockaddr *) (t->rn_key);
157         else
158             senderr(ENOBUFS);
159     }
```

rtsock.c

图20-8 (续)

1. 检查mbuf的合法性

113-136 检查mbuf的合法性：它的长度必须至少是一个rt_msghdr结构的大小。从mbuf的数据部分取出第一个长字，里面包含了rtm_msglen的值。

2. 分配缓存

137-142 分配一个缓存来存放整个报文，m_copydata将报文从mbuf链复制到缓存。

3. 检查版本号

143-146 检查报文的版本号。如果将来引入了新版本的选路报文，这个成员变量可以用来对早期版本提供支持。

147-149 进程的ID被复制到rtm_pid，进程提供的比特掩码被复制到该函数的一个内部结构info.rti_addrs。函数rt_xaddrs(在下一节显示)填入info结构的第8个插口地址指针来指示当前包含报文的缓存。

4. 需要的目的地址

150-151 所有的命令都需要一个目的地址。如果info.rti_info[RTAX_DST]项是一个空指针，就需要一个EINVAL。记住dst引用了这个数组成员(图19-19)。

5. 处理可选的genmask

152-159 genmask是可选的，它是在设置了RTF_CLONING标志后(图19-8)，用作所创建路由的网络掩码。rn_addmask将这个掩码加入到掩码树中，并首先在掩码树中查找是否存在与这个掩码相同的条目，如果找到，就引用那个条目。如果在掩码树中找到或者将这个掩码加入到掩码树中，还要再检查一下掩码树中的那个条目是否真等于genmask的值，如果等于，则genmask指针就被替代为掩码树中那个掩码的指针。

图20-9显示了route_output函数处理RTM_ADD和RTM_DELETE的下一部分。

162-163 RTM_ADD命令要求进程说明一个网关。

164-165 rtrequest处理该请求。如果输入的路由是一个主机路由，则netmask指针可以为空。如果一切OK，则saved_nrt返回新的路由表项的指针。

166-172 将rt_metrics结构从调用者缓存中复制到路由表项中。引用计数减1，并且保存genmask指针(可能是一个空指针)。

173-176 处理RTM_DELETE命令非常简单，因为所有的工作都由rtrequest来完成。既然最后一个参数是一个空指针，如果引用计数为0，rtrequest就调用rtfree从路由表中删除指定的项(图19-7)。

下一步的处理过程显示在图20-10中，它显示了RTM_GET、RTM_CHANGE和RTM_LOCK命

令的公共代码。

rtsock.c

```
160     switch (rtm->rtm_type) {

161     case RTM_ADD:
162         if (gate == 0)
163             senderr(EINVAL);
164         error = rtrequest(RTM_ADD, dst, gate, netmask,
165                           rtm->rtm_flags, &saved_nrt);
166         if (error == 0 && saved_nrt) {
167             rt_setmetrics(rtm->rtm_inits,
168                           &rtm->rtm_rmx, &saved_nrt->rt_rmx);
169             saved_nrt->rt_refcnt--;
170             saved_nrt->rt_genmask = genmask;
171         }
172         break;

173     case RTM_DELETE:
174         error = rtrequest(RTM_DELETE, dst, gate, netmask,
175                           rtm->rtm_flags, (struct rtentry **) 0);
176         break;
```

rtsock.c

图20-9 route_output 函数：进程RTM_ADD 和RTM_DELETE 命令

rtsock.c

```
177     case RTM_GET:
178     case RTM_CHANGE:
179     case RTM_LOCK:
180         rt = rtalloc1(dst, 0);
181         if (rt == 0)
182             senderr(ESRCH);
183         if (rtm->rtm_type != RTM_GET) {     /* XXX: too grotty */
184             struct radix_node *rn;
185             extern struct radix_node_head *mask_rnhead;

186             if (Bcmp(dst, rt_key(rt), dst->sa_len) != 0)
187                 senderr(ESRCH);
188             if (netmask && (rn = rn_search(netmask,
189                                            mask_rnhead->rnh_treetop)))
190                 netmask = (struct sockaddr *) rn->rn_key;
191             for (rn = rt->rt_nodes; rn; rn = rn->rn_dupedkey)
192                 if (netmask == (struct sockaddr *) rn->rn_mask)
193                     break;
194             if (rn == 0)
195                 senderr(ETOOMANYREFS);
196             rt = (struct rtentry *) rn;
197         }
```

rtsock.c

图20-10 route_output 函数：RTM_GET 、RTM_CHANGE 和RTM_LOCK 的公共处理部分

6. 查找已经存在的项

177-182 因为三个命令都引用了一个已经存在的项，所以用 rtalloc1函数来查找这个项。如果没有找到，则返回一个ESRCH。

7. 不允许网络匹配

183-187 对于RTM_CHANGE和RTM_LOCK命令，一个网络匹配是不合适的：需要一个路由表关键字的精确匹配。因此，如果 dst参数不等于路由表关键字，这个匹配就是一个网络匹配，返回一个ESRCH。

8. 使用网络掩码来查找正确的项

188-193 即使是一个精确的匹配，如果存在网络掩码不同的重复表项，仍然必须查找正确的项。如果提供了一个netmask参数，就要在掩码表中查找它(mask_rnhead)。如果找到了，netmask指针就被替代为掩码树中相应掩码的指针。检查重复表项列表的每个叶结点，查找一个rn_mask指针等于netmask的项。这个测试只是比较指针，而不是指针所指向的结构。这是因为所有的掩码都出现在掩码树中，并且每个不同的掩码只有一个副本出现在这个掩码树中。大多数情况下，表项不会重复，因此 for循环只执行一次。如果一个主机路由项被修改了，不应该提供一个网络掩码，因此， netmask和rn_mask都是空指针(两者是相等的)。但是，如果有一个附带掩码的项被修改了，那个掩码必须作为 netmask参数提供。

194-195 如果for循环终止时没有找到一个匹配的网络掩码，就返回 ETOOMANYREFS。

注释XXX表示这个函数必须做所有的工作来找到需要的项。所有这些细节在其他一些类似rtalloc1的函数中都应该被隐藏，rtalloc1检测网络匹配，并且处理掩码参数。

这个函数的下一部分继续处理 RTM_GET命令，显示在图 20-11中。这个命令与

rtsock.c

```
198         switch (rtm->rtm_type) {

199         case RTM_GET:
200             dst = rt_key(rt);
201             gate = rt->rt_gateway;
202             netmask = rt_mask(rt);
203             genmask = rt->rt_genmask;
204             if (rtm->rtm_addrs & (RTA_IFP | RTA_IFA)) {
205                 if (ifp = rt->rt_ifp) {
206                     ifpaddr = ifp->if_addrlist->ifa_addr;
207                     ifaaddr = rt->rt_ifa->ifa_addr;
208                     rtm->rtm_index = ifp->if_index;
209                 } else {
210                     ifpaddr = 0;
211                     ifaaddr = 0;
212                 }
213             }
214             len = rt_msg2(RTM_GET, &info, (caddr_t) 0,
215                           (struct walkarg *) 0);
216             if (len > rtm->rtm_msglen) {
217                 struct rt_msghdr *new_rtm;
218                 R_Malloc(new_rtm, struct rt_msghdr *, len);
219                 if (new_rtm == 0)
220                     senderr(ENOBUFS);
221                 Bcopy(rtm, new_rtm, rtm->rtm_msglen);
222                 Free(rtm);
223                 rtm = new_rtm;
224             }
225             (void) rt_msg2(RTM_GET, &info, (caddr_t) rtm,
226                            (struct walkarg *) 0);
227             rtm->rtm_flags = rt->rt_flags;
228             rtm->rtm_rmx = rt->rt_rmx;
229             rtm->rtm_addrs = info.rti_addrs;
230             break;
```

rtsock.c

图20-11 route_output 函数：RTM_GET 处理过程

route_output支持的其他命令的区别在于它能够返回比传递给它的更多的数据。例如，只需要输入一个插口地址结构，即目的地址，但至少返回两个插口地址结构，即目的地址和它的网关。参看图20-6，这就意味着为m_copydata复制数据所分配的缓存可能需要扩充大小。

9. 返回目的地址、网关和掩码

198-203 rti_info数组中存储了四个指针：dst、gate、netmask和genmask。后两个可能是空指针。info结构中的这些指针指向进程将要返回的各个插口地址结构。

10. 返回接口信息

204-213 进程可以在rtm_flags比特掩码中设置RTA_IFP和RTA_IFA掩码。如果设置了一项或两项，就表示进程想要接收这个路由表项所指示的 ifaddr结构：接口的链路层地址(由rt_ifp->addrlist指向)以及这个路由项的协议地址(由rt_ifa->ifa_addr指向)的内容。接口索引也会被返回。

11. 构造应答报文

214-224 将第三个指针置为空，调用rt_msg2来计算相应于RTM_GET的选路报文和info结构所指向的地址的长度。如果结果报文的长度超过了输入报文的长度，就会分配一个新的缓存，输入报文被复制到新的缓存中，老的缓存被释放， rtm被重新设置为指向新缓存。

225-230 再次调用rt_msg2，这次调用时第三个指针非空，因为在缓存中已经构造了一个结果报文。这次调用填入rt_msghdr结构的最后三个成员项。

图20-12显示了RTM_CHANGE和RTM_LOCK命令的处理过程。

12. 改变网关

231-233 如果进程传递了一个gate地址，rt_setgate就被调用来改变这个路由表项的网关。

13. 查找新的接口

234-244 新的网关(如果被改变)可能也需要rt_ifp和rt_ifa指针。进程可以通过传递一个ifpaddr插口地址结构或者一个ifaaddr插口地址结构来说明这些新的值。先看第一个，然后再看第二个。如果进程两个结构都没传递， rt_ifp和rt_ifa指针就被忽略。

14. 检验接口是否改变

245-256 如果找到了一个接口(ifa非空)，则该路由的现有rt_ifa指针要和新的值进行比较。如果数值已经改变了，则两个针对 rt_ifp和rt_ifa的新值需要存储到路由表的表项中去。在这样做之前，先要用 RTM_DELETE命令调用该接口的请求函数 (如果定义了该函数的话)。这个删除动作是必须的，因为从一种类型的网络到另一种类型的网络，它们的链路层信息可能会有很大的差别(比如说从一个X.25网络改变成以太网的路由)，同时我们还必须通知输出例程。

15. 更新度量

257-258 rt_setmetrics修改路由表项的度量。

16. 调用接口请求函数

259-260 如果定义了一个接口请求函数，它就会和 RTM_ADD命令一起被调用。

17. 保存克隆生成的掩码

261-262 如果进程指定了 genmask参数，就将在图 20-8中获得的掩码的指针保存在rt_genmask中。

18. 修改加锁度量的比特掩码

266-270 RTM_LOCK命令修改保存在rt_rmx.rmx_locks中的比特掩码。图20-13显示了这个比特掩码中不同比特的值，每个度量一个值。

rtsock.c

```
231         case RTM_CHANGE:
232             if (gate && rt_setgate(rt, rt_key(rt), gate))
233                 senderr(EDQUOT);
234             /* new gateway could require new ifaddr, ifp; flags may also be
235                different; ifp may be specified by ll sockaddr when protocol
236                address is ambiguous */
237             if (ifpaddr && (ifa = ifa_ifwithnet(ifpaddr)) &&
238                 (ifp = ifa->ifa_ifp))
239                 ifa = ifaof_ifpforaddr(ifaaddr ? ifaaddr : gate,
240                                        ifp);
241             else if ((ifaaddr && (ifa = ifa_ifwithaddr(ifaaddr))) ||
242                      (ifa = ifa_ifwithroute(rt->rt_flags,
243                                             rt_key(rt), gate)))
244                 ifp = ifa->ifa_ifp;
245             if (ifa) {
246                 struct ifaddr *oifa = rt->rt_ifa;
247                 if (oifa != ifa) {
248                     if (oifa && oifa->ifa_rtrequest)
249                         oifa->ifa_rtrequest(RTM_DELETE,
250                                             rt, gate);
251                     IFAFREE(rt->rt_ifa);
252                     rt->rt_ifa = ifa;
253                     ifa->ifa_refcnt++;
254                     rt->rt_ifp = ifp;
255                 }
256             }
257             rt_setmetrics(rtm->rtm_inits, &rtm->rtm_rmx,
258                           &rt->rt_rmx);
259             if (rt->rt_ifa && rt->rt_ifa->ifa_rtrequest)
260                 rt->rt_ifa->ifa_rtrequest(RTM_ADD, rt, gate);
261             if (genmask)
262                 rt->rt_genmask = genmask;
263             /*
264              * Fall into
265              */
266         case RTM_LOCK:
267             rt->rt_rmx.rmx_locks &= ~(rtm->rtm_inits);
268             rt->rt_rmx.rmx_locks |=
269                 (rtm->rtm_inits & rtm->rtm_rmx.rmx_locks);
270             break;
271         }
272         break;
273     default:
274         senderr(EOPNOTSUPP);
275     }
```

rtsock.c

图20-12 route_output 函数：RTM_CHANGE 和RTM_LOCK 处理过程

路由表项中rt_metrics结构的rmx_locks成员是告诉内核哪些度量不要管的比特掩码。即，rmx_locks指定的那些度量内核不能修改。内核惟一能使用这些度量的地方是和TCP一起，如图27-3所示。rmx_pksent度量不能被初始化或加锁，但是内核也从来没有引用或修改过这个成员。

进程发出的报文中的 rtm_inits值是一个比特掩码，指出哪些度量刚刚被

常 量	值	描 述
RTV_MTU	0x01	初始化或者锁住rmx_mtu
RTV_HOPCOUNT	0x02	初始化或者锁住rmx_hopcount
RTV_EXPIRE	0x04	初始化或者锁住rmx_expire
RTV_RPIPE	0x08	初始化或者锁住rmx_recvpipe
RTV_SPIPE	0x10	初始化或者锁住rmx_sendpipe
RTV_SSTHRESH	0x20	初始化或者锁住rmx_ssthresh
RTV_RTT	0x40	初始化或者锁住rmx_rtt
RTV_RTTVAR	0x80	初始化或者锁住rmx_rttvar

图20-13 对度量初始化或加锁的常量

rt_setmetrics初始化过。报文中的rtm_rmx.rmx_locks值是一个指出哪些度量现在应该加锁的比特掩码。rt_rmx.rmx_locks的值是一个指出路由表中哪些度量当前被加锁的比特掩码。首先，任何将要初始化的比特 (rtm_inits)都要解锁。任何既被初始化(rtm_inits)又被加锁(rtm_rmx.rmx_locks)的比特都必须加锁。

273-275 这个default是用于图20-9开始的switch语句，用来处理进程发出的报文中除了所支持的五个命令以外的其他选路命令。

route_output的最后一部分显示在图20-14中，用来发送应答给raw_input。

rtsock.c

```
276   flush:
277     if (rtm) {
278         if (error)
279             rtm->rtm_errno = error;
280         else
281             rtm->rtm_flags |= RTF_DONE;
282     }
283     if (rt)
284         rtfree(rt);
285     {
286         struct rawcb *rp = 0;
287         /*
288          * Check to see if we don't want our own messages.
289          */
290         if ((so->so_options & SO_USELOOPBACK) == 0) {
291             if (route_cb.any_count <= 1) {
292                 if (rtm)
293                     Free(rtm);
294                 m_freem(m);
295                 return (error);
296             }
297             /* There is another listener, so construct message */
298             rp = sotorawcb(so);
299         }
300         if (rtm) {
301             m_copyback(m, 0, rtm->rtm_msglen, (caddr_t) rtm);
302             Free(rtm);
303         }
304         if (rp)
305             rp->rcb_proto.sp_family = 0;     /* Avoid us */
306         if (dst)
307             route_proto.sp_protocol = dst->sa_family;
308         raw_input(m, &route_proto, &route_src, &route_dst);
```

图20-14 route_output 函数：将结果传递给raw_input

```
309         if (rp)
310             rp->rcb_proto.sp_family = PF_ROUTE;
311     }
312     return (error);
313 }
```
rtsock.c

图20-14 (续)

19. 返回错误或OK

276-282 flush是该函数开头定义的senderr宏所跳转的标号。如果发生了一个错误，错误就在rtm_errno成员中返回；否则，就设置RTF_DONE标志。

20. 释放拥有的路由

283-284 如果拥有一条路由，就要被释放。如果找到，在图 20-10的开始位置对rtalloc1的调用拥有这条路由。

21. 没有进程接收报文

285-296 SO_USELOOPBACK插口选项的默认值为真，表示发送进程将会收到它发送给选路插口的每个选路报文的一个复制(如果发送者不接收一个复制的报文，它就不能收到 RTM_GET返回的任何信息)。如果没有设置这个选项，并且选路插口的总数小于或等于 1，就没有其他进程接收报文，并且发送者不想要一个复制报文。缓存和 mbuf链都会被释放，该函数返回。

22. 没有环回复制报文的其他监听者

297-299 至少有一个其他的监听者而不是发送进程不想要一个复制报文。指针 rp，默认是空，被设置成指向发送者的选路控制块，它也用来作为发送者不想要复制报文的一个标志。

23. 将缓存转换成mbuf链

300-303 缓存被转换成一个mbuf链(图20-6)，然后释放缓存。

24. 避免环回复制

304-305 如果设置了rp，则某个其他的进程可能想要报文，但是发送者不想要一个复制。发送者的选路控制块的sp_family成员被临时设置为0，但是报文的sp_family(route_proto结构，显示在图19-26中)有一个PF_ROUTE的族。这个技巧防止raw_input将结果的一个复制传递给发送进程，因为raw_input不会将一个复制传递给sp_family为0的任何插口。

25. 设置选路报文的地址族

306-308 如果dst是一个非空的指针，则那个插口地址结构的地址族成为选路报文的协议。对于Internet协议，这个值将是PF_INET。通过raw_input，一个复制被传递给合适的监听者。

309-313 如果调用进程的sp_family成员被临时设置为0，它就被复位成正常值，PF_ROUTE。

20.6 rt_xaddrs函数

在将来自进程的选路报文从 mbuf链复制到一个缓存以及将来自进程的比特掩码(rtm_addrs)复制到rt_addrinfo结构的rti_info成员之后，只从route_output中调用一次rt_xaddrs函数(图20-8)。rt_xaddrs的目的是获取这个比特掩码，并且设置rti_info数组的指针，使之指向缓存中相应的地址。图 20-15显示了这个函数。

330-340 指针数组被设置成0，因此，所有在比特掩码中不出现的地址结构的指针都将是空。

341-347 测试比特掩码中8个(RTM_MAX)可能比特的每一个(如果设置)，将相应于插口地址结构的一个指针存到rti_info数组中。ADVANCE宏以插口地址结构的sa_len字段为参数，

上舍入为4个字节的倍数，相应地增加指针cp。

rtsock.c

```
330 #define ROUNDUP(a) \
331     ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
332 #define ADVANCE(x, n) (x += ROUNDUP((n)->sa_len))
333 static void
334 rt_xaddrs(cp, cplim, rtinfo)
335 caddr_t cp, cplim;
336 struct rt_addrinfo *rtinfo;
337 {
338     struct sockaddr *sa;
339     int     i;
340     bzero(rtinfo->rti_info, sizeof(rtinfo->rti_info));
341     for (i = 0; (i < RTAX_MAX) && (cp < cplim); i++) {
342         if ((rtinfo->rti_addrs & (1 << i)) == 0)
343             continue;
344         rtinfo->rti_info[i] = sa = (struct sockaddr *) cp;
345         ADVANCE(cp, sa);
346     }
347 }
```

rtsock.c

图20-15 rt_xaddrs 函数：将指针填入rti_info 数组

20.7 rt_setmetrics函数

这个函数在route_output中调用了两次：增加一条新路由时和改变一条已经存在的路由时。来自进程的选路报文的rtm_inits成员说明了进程想要初始化rtm_rmx数组中的哪些度量。比特掩码中的比特的值显示在图20-13中。

请注意，rtm_addrs和rtm_inits都是来自进程的报文中的比特掩码，前者说明了接下来的插口地址结构，而后者说明哪些度量将被初始化。为了节省空间，在rtm_addrs中没有设置比特的插口地址结构也不会出现在选路报文中。但是整个rt_metrics总是以定长的rt_msghdr结构的形式出现——在rtm_inits中没有设置比特的数组成员将被忽略。

图20-16显示了rt_setmetrics函数。

rtsock.c

```
314 void
315 rt_setmetrics(which, in, out)
316 u_long  which;
317 struct rt_metrics *in, *out;
318 {
319 #define metric(f, e) if (which & (f)) out->e = in->e;
320     metric(RTV_RPIPE, rmx_recvpipe);
321     metric(RTV_SPIPE, rmx_sendpipe);
322     metric(RTV_SSTHRESH, rmx_ssthresh);
323     metric(RTV_RTT, rmx_rtt);
324     metric(RTV_RTTVAR, rmx_rttvar);
325     metric(RTV_HOPCOUNT, rmx_hopcount);
326     metric(RTV_MTU, rmx_mtu);
327     metric(RTV_EXPIRE, rmx_expire);
328 #undef metric
329 }
```

rtsock.c

图20-16 rt_setmetrics 函数：设置rt_metrics 结构中的成员

314-318 which参数总是进程的选路报文的 rtm_inits成员。 in指向进程的 rt_metrics结构，而out指向将要创建或修改的路由表项的rt_metrics结构。

319-329 测试比特掩码中8比特的每一比特，如果该比特被设置，就复制相应的度量。请注意当使用RTM_ADD创建一个新的路由表项时， route_output调用了rtrequest，后者将整个路由表项设置为0(图19-9)。因此，在选路报文中，进程没有说明的任何度量，其默认值都是0。

20.8 raw_input函数

向一个进程发送的所有选路报文 —— 包括由内核产生的和由进程产生的 —— 都被传递给raw_input，后者选择接收这个报文的进程。图 18-11总结了调用raw_input的四个函数。

当创建一个选路插口时，族总是PF_ROUTE；而协议， socket的第三个参数，可能为0，表示进程想要接收所有的选路报文；或者是一个如同 AF_INET的值，限制插口只接收包含指定协议族地址的报文。为每个选路插口创建一个选路控制块 (20.3节)，这两个值分别存储在rcb_proto结构的sp_family和sp_protocol成员中。

图20-17显示了raw_input函数。

raw_usrreq.c

```
51 void
52 raw_input(m0, proto, src, dst)
53 struct mbuf *m0;
54 struct sockproto *proto;
55 struct sockaddr *src, *dst;
56 {
57     struct rawcb *rp;
58     struct mbuf *m = m0;
59     int     sockets = 0;
60     struct socket *last;
61     last = 0;
62     for (rp = rawcb.rcb_next; rp != &rawcb; rp = rp->rcb_next) {
63         if (rp->rcb_proto.sp_family != proto->sp_family)
64             continue;
65         if (rp->rcb_proto.sp_protocol &&
66             rp->rcb_proto.sp_protocol != proto->sp_protocol)
67             continue;
68         /*
69          * We assume the lower level routines have
70          * placed the address in a canonical format
71          * suitable for a structure comparison.
72          *
73          * Note that if the lengths are not the same
74          * the comparison will fail at the first byte.
75          */
76 #define equal(a1, a2) \
77   (bcmp((caddr_t)(a1), (caddr_t)(a2), a1->sa_len) == 0)
78         if (rp->rcb_laddr && !equal(rp->rcb_laddr, dst))
79             continue;
80         if (rp->rcb_faddr && !equal(rp->rcb_faddr, src))
81             continue;
82         if (last) {
83             struct mbuf *n;
84             if (n = m_copy(m, 0, (int) M_COPYALL)) {
85                 if (sbappendaddr(&last->so_rcv, src,
86                                  n, (struct mbuf *) 0) == 0)
```

图20-17 raw_input 函数：将选路报文传递给0个或多个进程

```
87                      /* should notify about lost packet */
88                      m_freem(n);
89                  else {
90                      sorwakeup(last);
91                      sockets++;
92                  }
93              }
94          }
95          last = rp->rcb_socket;
96      }
97      if (last) {
98          if (sbappendaddr(&last->so_rcv, src,
99                           m, (struct mbuf *) 0) == 0)
100             m_freem(m);
101         else {
102             sorwakeup(last);
103             sockets++;
104         }
105     } else
106         m_freem(m);
107 }
```

raw_usrreq.c

图20-17 (续)

51-61 在我们所看到的四个对raw_input的调用中，proto、src和dst参数指向三个全局变量route_proto、route_src和route_dst，这些变量都如同图19-26所示的那样被声明和初始化。

1. 比较地址族和协议

62-67 for循环遍历每个选路控制块来查找一个匹配。控制块里的族(一般是PF_ROUTE)必须与sockproto结构的族相匹配，否则这个控制块就被略过。接下来，如果控制块里的协议(socket的第三个参数)非空，它必须匹配sockproto结构的族；否则，这个报文被略去。因此，以0协议创建了一个选路插口的进程将收到所有的选路报文。

2. 比较本地的和外部的地址

68-81 如果指定了的话，这两个测试比较了控制块里的本地地址和外部地址。目前，进程不能设置控制块的rcb_laddr或者rcb_faddr成员。一般来说，进程使用bind设置前者，用connect设置后者，但对于Net/3中的选路插口这是不可能的。作为替代，我们将看到route_usrreq将插口固定地连接到route_src插口地址结构，这是可行的，因为它总是这个函数的src参数。

3. 将报文添加到插口的接收缓存中

82-107 如果last非空，它指向最近看到的应该接收这个报文的socket结构。如果这个变量非空，就使用m_copy和sbappendaddr将这个报文的一个复制添加到那个插口的接收缓存中，并且在这个接收缓存等待的任何进程都会被唤醒。然后，last被设置成指向在以前的测试中刚刚匹配的插口。使用last是为了在只有一个进程接收报文的情况下避免调用m_copy(一个代价昂贵的操作)。

如果有N个进程接收报文，那么前$N-1$个接收一个复制报文，最后一个进程收到的是这个报文本身。

在这个函数里递增的socket变量并没有被用到。因为只有当报文被传递给一个进程后它才会被递增，所以，如果在函数的结尾这个变量的值是0，就表示没有进程接收该报文(但是

变量值没有在任何地方保存)。

20.9 route_usrreq函数

route_usrreq是选路协议的用户请求函数。它被不同的操作调用。图 20-18显示了这个函数。

rtsock.c

```
64 int
65 route_usrreq(so, req, m, nam, control)
66 struct socket *so;
67 int     req;
68 struct mbuf *m, *nam, *control;
69 {
70     int     error = 0;
71     struct rawcb *rp = sotorawcb(so);
72     int     s;

73     if (req == PRU_ATTACH) {
74         MALLOC(rp, struct rawcb *, sizeof(*rp), M_PCB, M_WAITOK);
75         if (so->so_pcb = (caddr_t) rp)
76             bzero(so->so_pcb, sizeof(*rp));
77     }
78     if (req == PRU_DETACH && rp) {
79         int     af = rp->rcb_proto.sp_protocol;
80         if (af == AF_INET)
81             route_cb.ip_count--;
82         else if (af == AF_NS)
83             route_cb.ns_count--;
84         else if (af == AF_ISO)
85             route_cb.iso_count--;
86         route_cb.any_count--;
87     }
88     s = splnet();
89     error = raw_usrreq(so, req, m, nam, control);
90     rp = sotorawcb(so);
91     if (req == PRU_ATTACH && rp) {
92         int     af = rp->rcb_proto.sp_protocol;
93         if (error) {
94             free((caddr_t) rp, M_PCB);
95             splx(s);
96             return (error);
97         }
98         if (af == AF_INET)
99             route_cb.ip_count++;
100        else if (af == AF_NS)
101            route_cb.ns_count++;
102        else if (af == AF_ISO)
103            route_cb.iso_count++;
104        route_cb.any_count++;

105        rp->rcb_faddr = &route_src;
106        soisconnected(so);
107        so->so_options |= SO_USELOOPBACK;
108    }
109    splx(s);
110    return (error);
111 }
```

rtsock.c

图20-18 route_usrreq 函数：处理PRU_ xxx请求

1. PRU_ATTACH：分配控制块

64-77 当进程调用socket时，就会发出PRU_ATTACH请求。为一个选路控制块分配内存。MALLOC返回的指针保存在socket结构的so_pcb成员中。如果分配了内存，rawcb结构被设置成0。

2. PRU_DETACH：计数器递减

78-87 close系统调用发出PRU_DETACH请求。如果socket结构指向一个协议控制块，route_cb结构的计数器中有两个被减1：一个是any_count；另一个是基于该协议的计数器。

3. 处理请求

88-90 函数raw_usrreq被调用来进一步处理PRU_*xxx*请求。

4. 计数器递增

91-104 如果请求是PRU_ATTACH，并且插口指向一个选路控制块，就要检查raw_usrreq是否返回一个错误。然后，route_cb结构的计数器中的两个被递增：一个是any_count，另一个是基于该协议的计数器。

5. 连接插口

105-106 选路控制块里的外部地址被设置成route_src。这将永久地连接到新的插口来接收PF_ROUTE族的选路报文。

6. 默认情况下使能SO_USELOOPBACK

107-111 使能SO_USELOOPBACK插口选项。这是一个默认使能的插口选项——其他所有的选项默认都被禁止。

20.10 raw_usrreq函数

raw_usrreq完成在选路域中用户请求处理的大部分工作。在上一节中它被route_usrreq函数所调用。用户请求的处理被划分成这两个函数，是因为其他的一些协议(例如OSI CLNP)调用raw_usrreq而不是route_usrreq。raw_usrreq并不是想要成为一个协议的pr_usrreq函数，相反，它是一个被不同的pr_usrreq函数调用的公共的子例程。

图20-19显示了raw_usrreq函数的开始和结尾。其中的switch语句体在该图后面的图中单独讨论。

1. PRU_CONTROL请求是不合法的

119-129 PRU_CONTROL请求来自于ioctl系统调用，在路由选择域中不被支持。

2. 控制信息不合法

130-133 如果进程传递控制信息(使用sendmsg系统调用)，就会返回一个错误，因为路由选择域中不使用这个可选的信息。

3. 插口必须有一个控制块

134-137 如果socket结构没有指向一个选路控制块，就返回一个错误。如果创建了一个新的插口，调用者(即route_usrreq)有责任在调用这个函数之前分配这个控制块，并且将指针保存在so_pcb成员中。

262-269 这个switch语句的default子句处理case子句没有处理的两个请求：PRU_BIND

和PRU_CONNECT。这两个请求的代码是提供的，但在Net/3中被注释掉了。因此，如果在一个选路插口上发出bind或connect系统调用，就会引起一个内核的告警(panic)。这是一个程序错误(bug)。幸运的是创建这种类型的插口需要有超级用户的权限。

raw_usrreq.c

```
119 int
120 raw_usrreq(so, req, m, nam, control)
121 struct socket *so;
122 int     req;
123 struct mbuf *m, *nam, *control;
124 {
125     struct rawcb *rp = sotorawcb(so);
126     int     error = 0;
127     int     len;

128     if (req == PRU_CONTROL)
129         return (EOPNOTSUPP);
130     if (control && control->m_len) {
131         error = EOPNOTSUPP;
132         goto release;
133     }
134     if (rp == 0) {
135         error = EINVAL;
136         goto release;
137     }
138     switch (req) {

                /* switch cases */

262     default:
263         panic("raw_usrreq");
264     }
265 release:
266     if (m != NULL)
267         m_freem(m);
268     return (error);
269 }
```

raw_usrreq.c

图20-19 raw_usrreg 函数体

我们现在讨论单个的case语句。图20-20显示了对PRU_ATTACH和PRU_DETACH请求的处理。

139-148 PRU_ATTACH请求是socket系统调用的一个结果。一个选路插口只能由一个超级用户的进程创建。

149-150 函数raw_attach(图20-24)将控制块链接到双向链接列表中。 nam参数是socket的第三个参数，被存储在控制块中。

151-159 PRU_DETACH是由close系统调用发出的请求。对一个空的rp指针的测试是多余的，因为在switch语句之前已经进行过这个测试了。

160-161 raw_detach(图20-25)从双向链接表中删除这个控制块。

图20-21显示了PRU_CONNECT2、PRU_DISCONNECT和PRU_SHUTDOWN请求的处理。

186-188 PRU_CONNECT2请求来自于socketpair系统调用，在路由选择域中不被支持。

189-196 因为一个选路插口总是连接的(图20-18)，所以PRU_DISCONNECT请求在PRU_DETACH请

raw_usrreq.c

```
139         /*
140          * Allocate a raw control block and fill in the
141          * necessary info to allow packets to be routed to
142          * the appropriate raw interface routine.
143          */
144     case PRU_ATTACH:
145         if ((so->so_state & SS_PRIV) == 0) {
146             error = EACCES;
147             break;
148         }
149         error = raw_attach(so, (int) nam);
150         break;

151         /*
152          * Destroy state just before socket deallocation.
153          * Flush data or not depending on the options.
154          */
155     case PRU_DETACH:
156         if (rp == 0) {
157             error = ENOTCONN;
158             break;
159         }
160         raw_detach(rp);
161         break;
```

raw_usrreq.c

图20-20 raw_usrreq 函数：PRU_ATTACH 和PRU_DETACH 请求

求之前由close发出。插口必须已经和一个外部地址相连接，这对于一个选路插口来说总是成立的。raw_disconnect和soisdisconnected完成这个处理。

197-202 当参数指定在这个插口上没有更多的写操作时，shutdown系统调用发出PRU_SHUTDOWN请求。socantsendmore禁止以后的写操作。

raw_usrreq.c

```
186     case PRU_CONNECT2:
187         error = EOPNOTSUPP;
188         goto release;

189     case PRU_DISCONNECT:
190         if (rp->rcb_faddr == 0) {
191             error = ENOTCONN;
192             break;
193         }
194         raw_disconnect(rp);
195         soisdisconnected(so);
196         break;

197         /*
198          * Mark the connection as being incapable of further input.
199          */
200     case PRU_SHUTDOWN:
201         socantsendmore(so);
202         break;
```

raw_usrreq.c

图20-21 raw_usrreq 函数：PRU_CONNECT2 、PRU_DISCONNECT 和PRU_SHUTDOWN 请求

对一个选路插口最常见的请求：PRU_SEND、PRU_ABORT和PRU_SENSE显示在图20-22中。

203-217 当进程向插口写时，sosend发出了PRU_SEND请求。如果指定了一个nam参数，

raw_usrreq.c

```
203         /*
204          * Ship a packet out.  The appropriate raw output
205          * routine handles any massaging necessary.
206          */
207     case PRU_SEND:
208         if (nam) {
209             if (rp->rcb_faddr) {
210                 error = EISCONN;
211                 break;
212             }
213             rp->rcb_faddr = mtod(nam, struct sockaddr *);
214         } else if (rp->rcb_faddr == 0) {
215             error = ENOTCONN;
216             break;
217         }
218         error = (*so->so_proto->pr_output) (m, so);
219         m = NULL;
220         if (nam)
221             rp->rcb_faddr = 0;
222         break;

223     case PRU_ABORT:
224         raw_disconnect(rp);
225         sofree(so);
226         soisdisconnected(so);
227         break;

228     case PRU_SENSE:
229         /*
230          * stat: don't bother with a blocksize.
231          */
232         return (0);
```

raw_usrreq.c

图20-22 raw_usrreq 函数：PRU_SEND 、PRU_ABORT 和PRU_SENSE 请求

即进程使用 sendto或者sendmsg指定了一个目的地址，就会返回一个错误，因为route_usrreq总是为一个选路插口设置rcb_faddr。

218-222 m指向的mbuf链中的信息被传递给协议的pr_output函数，也就是route_output。

223-227 如果发出了一个PRU_ABORT请求，则该控制块被断开连接，插口被释放，然后被断开连接。

228-232 fstat系统调用发出PRU_SENSE请求。函数返回OK。

图20-23显示了剩下的PRU_*xxx*请求。

raw_usrreq.c

```
233         /*
234          * Not supported.
235          */
236     case PRU_RCVOOB:
237     case PRU_RCVD:
238         return (EOPNOTSUPP);

239     case PRU_LISTEN:
240     case PRU_ACCEPT:
241     case PRU_SENDOOB:
242         error = EOPNOTSUPP;
```

图20-23 raw_usrreq 函数：最后部分

```
243             break;

244         case PRU_SOCKADDR:
245             if (rp->rcb_laddr == 0) {
246                 error = EINVAL;
247                 break;
248             }
249             len = rp->rcb_laddr->sa_len;
250             bcopy((caddr_t) rp->rcb_laddr, mtod(nam, caddr_t), (unsigned) len);
251             nam->m_len = len;
252             break;

253         case PRU_PEERADDR:
254             if (rp->rcb_faddr == 0) {
255                 error = ENOTCONN;
256                 break;
257             }
258             len = rp->rcb_faddr->sa_len;
259             bcopy((caddr_t) rp->rcb_faddr, mtod(nam, caddr_t), (unsigned) len);
260             nam->m_len = len;
261             break;
```

raw_usrreq.c

图20-23 (续)

233-243 这五个请求不被支持。

244-261 PRU_SOCKADDR和PRU_PEERADDR请求分别来自于getsockname和getpeername系统调用。前者总是返回一个错误，因为设置本地地址的bind系统调用在路由选择域中不被支持。后者总是返回插口地址结构route_src的内容，这个内容是由route_usrreq作为外部地址设置的。

20.11 raw_attach、raw_detach和raw_disconnect函数

raw_attach函数，显示在图20-24中，被raw_input调用来完成PRU_ATTACH请求的处理。

raw_cb.c

```
49 int
50 raw_attach(so, proto)
51 struct socket *so;
52 int     proto;
53 {
54     struct rawcb *rp = sotorawcb(so);
55     int     error;

56     /*
57      * It is assumed that raw_attach is called
58      * after space has been allocated for the
59      * rawcb.
60      */
61     if (rp == 0)
62         return (ENOBUFS);
63     if (error = soreserve(so, raw_sendspace, raw_recvspace))
64         return (error);
65     rp->rcb_socket = so;
66     rp->rcb_proto.sp_family = so->so_proto->pr_domain->dom_family;
67     rp->rcb_proto.sp_protocol = proto;
68     insque(rp, &rawcb);
69     return (0);
70 }
```

raw_cb.c

图20-24 raw_attach 函数

49-64 调用者必须已经分配了原始的协议控制块。soreserve将发送和接收缓存的高水位标记设置为8192。这对于选路报文应该是绰绰有余了。

65-67 socket结构的一个指针和dom_family(即图20-1中用于选路域的PF_ROUTE)以及proto参数(socket调用的第三个参数)一起被存储到协议控制块中。

68-70 insque将这个控制块加入到由全局变量rawcb作为头指针的双向链接表的前面。

raw_detach函数，显示在图20-25中，被raw_input调用来完成PRU_DETACH请求的处理。

raw_cb.c

```
75 void
76 raw_detach(rp)
77 struct rawcb *rp;
78 {
79     struct socket *so = rp->rcb_socket;

80     so->so_pcb = 0;
81     sofree(so);
82     remque(rp);
83     free((caddr_t) (rp), M_PCB);
84 }
```

raw_cb.c

图20-25 raw_detach 函数

75-84 socket结构中的so_pcb指针被设置成空，然后释放这个插口。使用remque从双向链接表中删除该控制块，使用free来释放被控制块占用的内存。

raw_disconnect函数，显示在图 20-26中，被raw_input调用来完成 PRU_DISCONNECT和PRU_ABORT请求的处理。

88-94 如果该插口没有引用一个描述符，raw_detach释放该插口和控制块。

raw_cb.c

```
88 void
89 raw_disconnect(rp)
90 struct rawcb *rp;
91 {

92     if (rp->rcb_socket->so_state & SS_NOFDREF)
93         raw_detach(rp);
94 }
```

raw_cb.c

图20-26 raw_disconnect 函数

20.12 小结

一个选路插口是PF_ROUTE域中的一个原始插口。选路插口只能被一个超级用户进程创建。如果一个没有权限的进程想要读内核包含的选路信息，可以使用选路域所支持的 sysctl 系统调用(我们在前一章中描述过)。

在本章中，我们第一次碰到了与插口相联系的协议控制块 (PCB)。在选路域中，一个专门的rawcb包含了有关选路插口的信息：本地和外部的地址、地址族和协议。我们将在第 22章中看到用于UDP、TCP和原始IP插口的更大的Internet协议控制块(inpcb)。然而概念是相同的：socket结构被插口层使用，而 PCB，一个rawcb或一个inpcb，被协议层使用。socket结构指向该PCB，后者也指向前者。

route_output函数处理进程可以发出的五个请求。依赖于协议和地址族，raw_input将一个选路报文发送给一个或多个选路插口。对一个选路插口的不同的 PRU_*xxx*请求由raw_usrreq和route_usrreq处理。在后面的章节中，我们将碰到另外的 *xxx*_usrreq函数，每个协议(UDP、TCP和原始IP)对应一个，每个函数都由一个 switch语句组成用来处理每一个请求。

习题

20.1 当进程向一个选路插口写一个报文时，列出两种进程可以从 route_output收到返回值的方法。哪种方法更可靠？

20.2 因为routesw结构的pr_protocol成员为0，所以当进程对socket系统调用指定了一个非0的*protocol*参数时，会发生什么情况？

20.3 路由表中的路由(和ARP项不同)永远不会超时。试在路由上实现一个超时机制。