第17章 Unix域协议：实现

17.1 概述

在uipc_usrreq.c文件中实现Unix域协议的源代码包含16个函数，总共大约有1000行C语言源程序，这与在卷2中实现UDP的800行源程序长度差不多，比实现TCP的4500行源程序要短得多。

我们分两章来描述Unix域协议的实现，下一章讨论I/O和描述符传递，其他的内容都在本章讨论。

17.2 代码介绍

在一个C文件中有16个Unix域函数，在其他C文件和两个头文件中还有其他有关的定义，如图17-1所示。

文　件	说　明
sys/un.h	sockaddr_un结构的定义
sys/unpcb.h	unpcb结构的定义
kern/uipc_proto.c	Unix域protosw{}和domain{}的定义
kern/uipc_usrreq.c	Unix域函数
kern/uipc_syscalls.c	pipe和socketpair系统调用

图17-1 在本章中讨论的文件

在本章我们也介绍pipe和socketpair系统调用，它们都使用本章描述的Unix域函数。

全局变量

图17-2列出了在本章和下一章中讨论的11个全局变量。

变　量	数 据 类 型	说　明
unixdomain	struct domain	域定义(图17-4)
unixsw	struct protosw	协议定义(图17-5)
sun_noname	struct sockaddr	包含空路径名的插口地址结构
unp_defer	int	延迟入口的无用单元收集计数器
unp_gcing	int	如果当前执行无用单元收集函数，就设置
unp_ino	ino_t	下一个分配的伪i_node号的值
unp_rights	int	当前传送中的文件描述符数
unpdg_recvspace	u_long	数据报插口接收缓存的默认范围，4096字节
unpdg_sendspace	u_long	数据报插口发送缓存的默认范围，2048字节
unpst_recvspace	u_long	流插口接收缓存的默认范围，4096字节
unpst_sendspace	u_long	流插口接收缓存的默认范围，4096字节

图17-2 在本章中介绍的全局变量

17.3 Unix domain和protosw结构

图17-3表示了Net/3系统中常见的三个domain结构，同时还有相应的protosw数组。

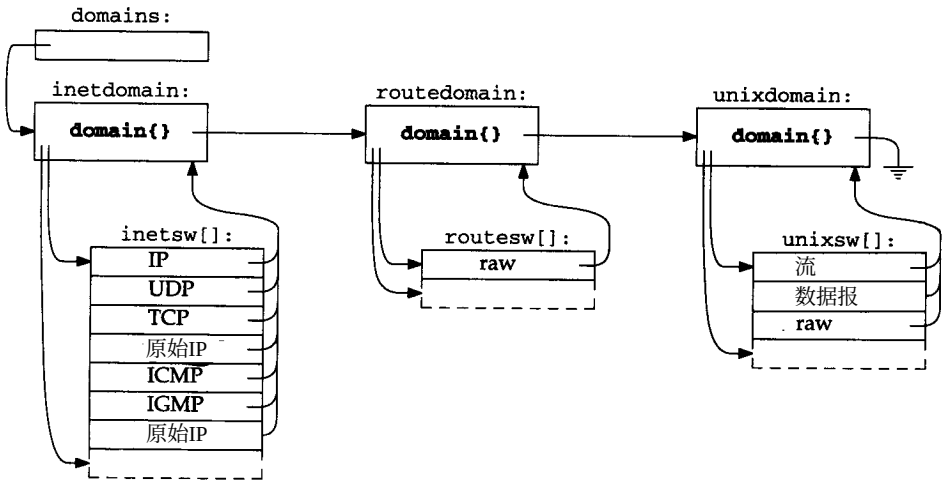

图17-3 domain 表和protosw 数组

卷2描述了Internet和路由选择域，图17-4描述了Unix域协议使用的domain结构(卷2图7-5)中的字段。

由于历史的原因，两个raw IP记录项在卷2图7-12中描述。

单 元	值	说 明
dom_family	*PF_UNIX*	域协议族
dom_name	*unix*	名字
dom_init	0	在Unix域中没有使用
dom_externalize	*unp_externalize*	外部化访问权(图18-12)
dom_dispose	*unp_dispose*	释放内部化权利(图18-14)
dom_protosw	*unixsw*	协议转换数组(图17-5)
dom_protoswNPROTOSW		协议转换数组的尾部指针
dom_next		由domaininit填充，卷2
dom_rtattach	0	在Unix域中没有使用
dom_rtoffset	0	在Unix域中没有使用
dom_maxrtkey	0	在Unix域中没有使用

图17-4 unixdomain 结构

仅有Unix domain定义了dom_externalize和dom_dispose两个函数，我们在第18章中讨论描述符传递时再描述这两个函数，由于 Unix 域没有路由选择表，所以domain结构的最后三个元素没有定义。

图17-5描述了unixsw结构的初始化(卷2图7-13描述了Internet协议的对应结构)。

定义三个协议：

- 与TCP相似的流协议；
- 与UDP相似的数据报协议；
- 与原始IP相似的raw协议。

uipc_proto.c

```
41 struct protosw unixsw[] =
42 {
43     {SOCK_STREAM, &unixdomain, 0, PR_CONNREQUIRED | PR_WANTRCVD | PR_RIGHTS,
44      0, 0, 0, 0,
45      uipc_usrreq,
46      0, 0, 0, 0,
47     },
48     {SOCK_DGRAM, &unixdomain, 0, PR_ATOMIC | PR_ADDR | PR_RIGHTS,
49      0, 0, 0, 0,
50      uipc_usrreq,
51      0, 0, 0, 0,
52     },
53     {0, 0, 0, 0,
54      raw_input, 0, raw_ctlinput, 0,
55      raw_usrreq,
56      raw_init, 0, 0, 0,
57     },
58 };
```

uipc_proto.c

图17-5 unixsw 数组的初始化

由于Unix 域支持访问权(就是我们在下一章要讲的描述符传递), Unix域流协议和数据报协议都设置PR_RIGHTS标志。流协议的另外两个标志PR_CONNREQUIRED和PR_WANTRCVD与TCP的标志一样；数据报协议的两个标志PR_ATOMIC和PR_ADDR与UDP的标志一样。需要注意的是流协议与数据报协议定义的唯一一个函数指针是 uipc_usrreq，用它处理所有的用户请求。

在raw协议的protosw结构中的四个函数指针都是以raw_开头，与PR_ROUTE域中的一样，这些内容在卷2的第20章介绍。

作者从来没有听到过一个应用程序使用 Unix域的raw协议。

17.4 Unix域插口地址结构

图17-6描述了一个Unix 域插口地址结构的定义，一个sockaddr_un结构长度为106个字节。

un.h

```
38 struct sockaddr_un {
39     u_char  sun_len;            /* sockaddr length including null */
40     u_char  sun_family;         /* AF_UNIX */
41     char    sun_path[104];      /* path name (gag) */
42 };
```

un.h

图17-6 Unix 域插口地址结构

开始的两个域与其他的插口地址结构一样：地址族 (AF_UNIX)后紧跟着一个长度字节。

自从4.2BSD以来，注解“ gag ”就存在了，也许原作者并不喜欢使用路经名来标识Unix域插口，或者是因为一个完整的路径名太长以至在 mbuf中写不下(路经名的长度能达到1024字节)。

我们将要看到 Unix 域插口使用文件系统中的路经名来标识插口，并且路径名存储在sun_path中。sun_path的大小为104字节，一个mbuf的大小为128个字节，刚好存放插口地址结构和一个表示终止的空字节。如图 17-7所示。

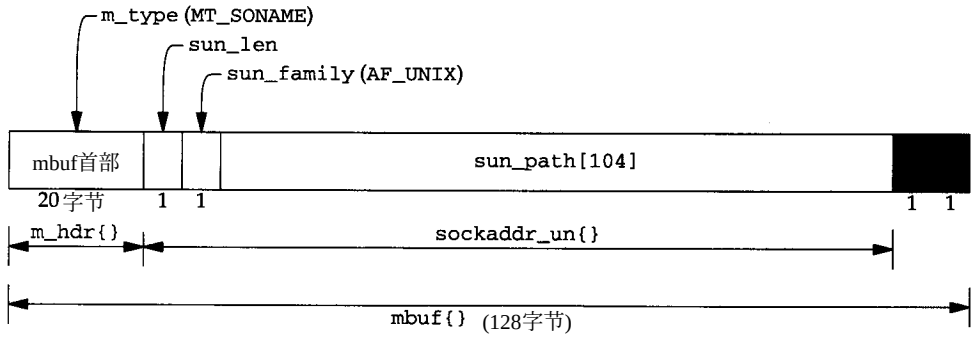

图17-7 存储在一个mbuf中的Unix域插口地址结构

我们将mbuf中的m_type字段设置成MT_SONAME，因为当mbuf含有一个插口地址结构时m_type就是这个普通值。虽然从图上看，最后两个字节没有使用，并且与这些插口相联系的最长路径名是104字节，但是我们将看到unp_bind和unp_connect两个函数允许一个路径名后面跟一个空字节时可以长达105字节。

Unix域插口在一些地方需要一个命名空间，由于文件系统的命名空间已经存在，所以就选定了路径名。与其他例子一样， Internet协议使用IP地址和端口号作为命名空间，系统V IPC([Stevens1992]的第14章)使用32比特密钥。由于Unix域客户进程用路径名来与服务器进程同步，从而通常使用绝对路径名 (以/开头)。如果使用相对路径名，客户程序和服务器程序必须在相同的目录中，或者服务器程序的绑定路经名不会被客户程序的connect和sendto发现。

17.5 Unix域协议控制块

Unix域插口有一个相关联的协议控制块 (PCB)，一个unpcb结构，我们在图17-8中描述了这个36字节的结构。

unpcb.h

```
60 struct unpcb {
61     struct socket *unp_socket;   /* pointer back to socket structure */
62     struct vnode *unp_vnode;     /* nonnull if associated with file */
63     ino_t   unp_ino;             /* fake inode number */
64     struct unpcb *unp_conn;      /* control block of connected socket */
65     struct unpcb *unp_refs;      /* referencing socket linked list */
66     struct unpcb *unp_nextref;   /* link in unp_refs list */
67     struct mbuf *unp_addr;       /* bound address of socket */
68     int     unp_cc;              /* copy of rcv.sb_cc */
69     int     unp_mbcnt;           /* copy of rcv.sb_mbcnt */
70 };

71 #define sotounpcb(so)    ((struct unpcb *)((so)->so_pcb))
```

unpcb.h

图17-8 Unix域协议控制块

不像路由域中使用的Internet PCB和控制块，这两者都是通过内核MALLOC函数来分配的(分别见卷2图20-18和图22-6)，而unpcb结构却存储在mbuf中，这可能是一个历史的人为因素。

另一个不同点是除了Unix域协议控制块以外，所有的控制块都保留在一个双向循环链表上，当数据到达时能通过查找这个链表将数据传递给相应的插口。对于所有的Unix域协议控制块而言，没有必要维护这样的链表，因为同样的操作，也就是当客户进程调用connect时，查找服务器的控制块是通过内核中已有的路径名查找函数来实现的。一旦找到服务器的unpcb，就可以将它的地址存储在客户进程的unpcb中，因为Unix域插口的客户进程与服务器进程在相同的主机上。

图17-9描述了处理Unix域插口的不同数据结构的关系，在这个图中我们描述了两个Unix

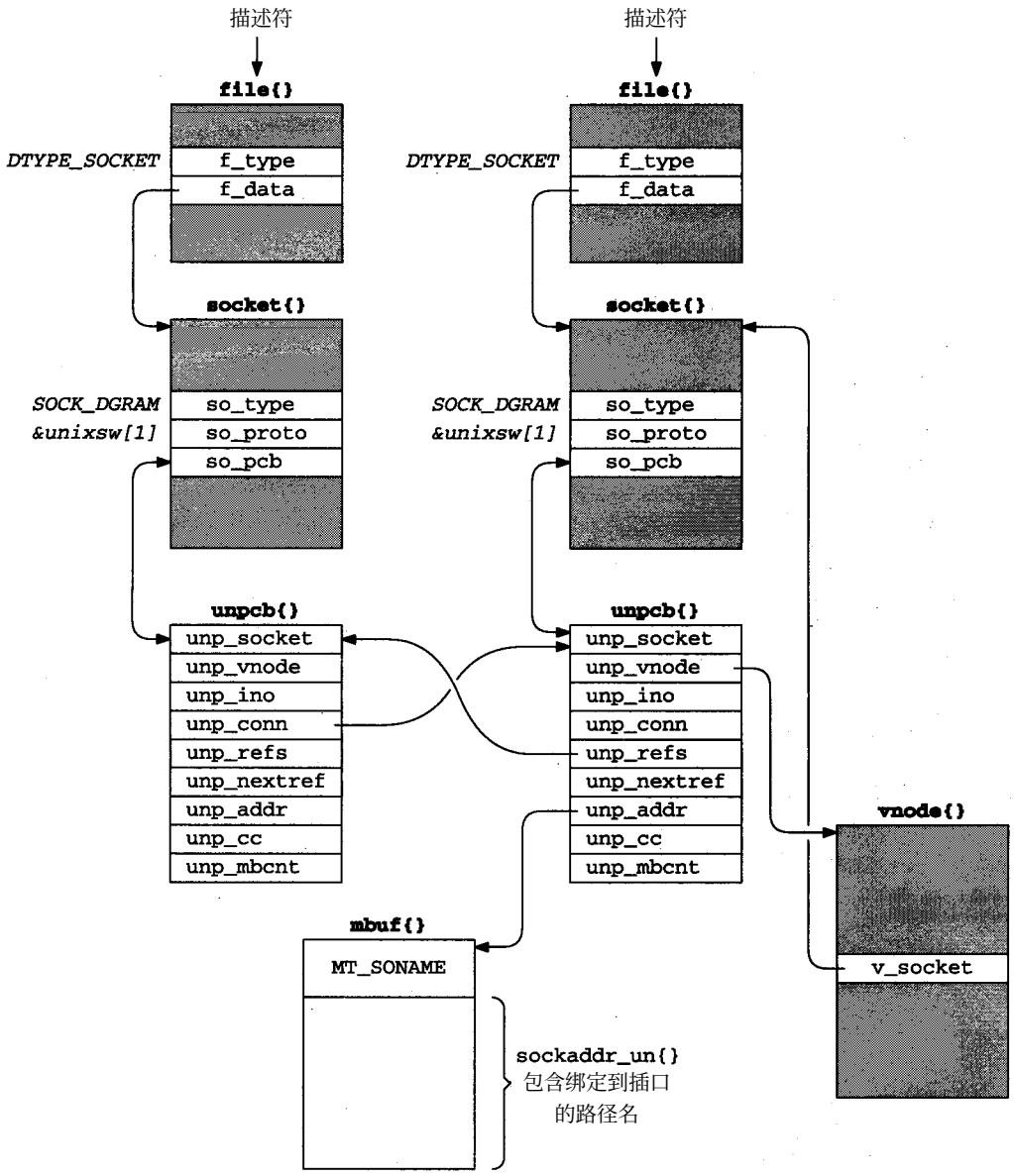

图17-9 互相连接的两个Unix域数据报插口

域数据报插口，我们假定右边的(服务器进程)插口已经绑定了一个路径名到它的插口，左边的(客户进程)插口已经连接到服务器的路径名上。

客户进程PCB的unp_conn单元指向服务器进程的PCB，服务器进程的unp_refs指向连接到这个PCB上的第一个客户进程（不像流插口，多个数据报客户进程可以连接到同一个服务器进程上，在17.11节我们要详细讨论Unix域数据报插口的连接)。

服务器的unp_vnode单元指向vnode，vnode与绑定到服务器插口的路径名相联系，它的v_socket单元指向服务器的socket，这就是定位一个已经绑定了路径名的unpcb所需的链接。例如，当服务器绑定了一个路径名到它的Unix域插口时，就会创建一个vnode结构，并且将unpcb的指针存储在v_node的v_socket中。当客户进程连接到服务器上时，内核中的路径名查找代码定位v-node，然后从v_socket指针获得服务器进程的unpcb指针。

被绑定到服务器插口的名字包含在socksddr_un结构中，sockaddr_un结构本身包含在unp_sddr指向的mbuf结构中。Unix 的v-node从来没有包含指向v-node的路径名，因为在一个Unix文件系统中多个名字(即目录记录项)能同时指向一个给定的文件(即v-node)。

图17-9表示两个连接的数据报插口，在图 17-26中我们将看到，处理流插口时与这里有些不同。

17.6 uipc_usrreq函数

在图17-5中我们看到，对于流和数据报协议，unixsw结构中引用的唯一函数是uipc_usrreq，图17-10给出了这个函数的要点。

uipc_usrreq.c

```
47 int
48 uipc_usrreq(so, req, m, nam, control)
49 struct socket *so;
50 int     req;
51 struct mbuf *m, *nam, *control;
52 {
53     struct unpcb *unp = sotounpcb(so);
54     struct socket *so2;
55     int     error = 0;
56     struct proc *p = curproc;    /* XXX */

57     if (req == PRU_CONTROL)
58         return (EOPNOTSUPP);
59     if (req != PRU_SEND && control && control->m_len) {
60         error = EOPNOTSUPP;
61         goto release;
62     }
63     if (unp == 0 && req != PRU_ATTACH) {
64         error = EINVAL;
65         goto release;
66     }
67     switch (req) {

            /* switch cases (discussed in following sections) */

246     default:
```

图17-10 uipc_usrreq 函数体

```
247             panic("piusrreq");
248         }
249     release:
250         if (control)
251             m_freem(control);
252         if (m)
253             m_freem(m);
254         return (error);
255 }
```
uipc_usrreq.c

图17-10 (续)

1. 无效的PRU_CONTROL请求

57-58 PRU_CONTROL请求来自ioctl系统调用，不被Unix域支持。

2. 仅为PRU_SEND支持的控制信息

59-62 如果进程传送控制信息(使用sendmsg系统调用)，请求必须是PRU_SEND；否则，返回一个错误。描述符在使用该请求的控制信息的进程间传递，这部分我们在第18章中讨论。

3. 插口必须有一个控制块

63-66 如果socket结构没有指向一个Unix域控制块，请求必须是PRU_ATTACH；否则，返回一个错误。

67-248 在下面几节中我们讨论这个函数的每一个case语句，以及调用的不同unp_*xxx*函数。

249-255 释放任何控制信息和数据mbuf，然后函数返回。

17.7 PRU_ATTACH请求和unp_attach函数

当一个连接请求到达一个处于监听状态的流插口时，socket系统调用和sonewconn函数(卷2图15-29)产生PRU_ATTACH请求，如图17-11所示。

uipc_usrreq.c
```
68      case PRU_ATTACH:
69          if (unp) {
70              error = EISCONN;
71              break;
72          }
73          error = unp_attach(so);
74          break;
```
uipc_usrreq.c

图17-11 PRU_ATTACH 请求

unp_attach函数完成这个请求的所有处理工作，如图17-12所示。socket结构已经被插口层分配和初始化，现在轮到协议层分配和初始化自身的协议控制块，在本例中这个协议控制块为unpcb结构。

uipc_usrreq.c
```
270 int
271 unp_attach(so)
272 struct socket *so;
273 {
274     struct mbuf *m;
275     struct unpcb *unp;
276     int     error;
```

图17-12 unp_attack 函数

```
277     if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
278         switch (so->so_type) {

279         case SOCK_STREAM:
280             error = soreserve(so, unpst_sendspace, unpst_recvspace);
281             break;

282         case SOCK_DGRAM:
283             error = soreserve(so, unpdg_sendspace, unpdg_recvspace);
284             break;

285         default:
286             panic("unp_attach");
287         }
288         if (error)
289             return (error);
290     }
291     m = m_getclr(M_DONTWAIT, MT_PCB);
292     if (m == NULL)
293         return (ENOBUFS);
294     unp = mtod(m, struct unpcb *);
295     so->so_pcb = (caddr_t) unp;
296     unp->unp_socket = so;
297     return (0);
298 }
```

uipc_usrreq.c

图17-12 (续)

1. 设置插口高水位标记

277-290 如果插口发送和接收的高水位标记为0，则soreserve将它们设置成图17-2所示的默认值，高水位标记限制了存放在插口发送和接收缓存中的数据量。当通过socket系统调用来调用unp_attach时，这两个标记都为0，但是当通过sonewconn调用unp_attach时，它们等于监听插口中的值。

2. 分配并初始化PCB

291-296 m_getclr获得一个mbuf用于unpcb结构，将mbuf清零并将类型设置成MT_PCB。注意所有的PCB单元都被初始化为0。通过so_pcb和unp_socket指针将socket和unpcb结构连接起来。

17.8 PRU_DETACH请求和unp_detach函数

当一个插口关闭时发出PRU_DETACH请求(卷2图15-39)，这个请求跟随在PRU_DISCONNECT请求(仅针对有连接的插口)的后面，如图17-13所示。

uipc_usrreq.c

```
75      case PRU_DETACH:
76          unp_detach(unp);
77          break;
```

uipc_usrreq.c

图17-13 PRU_DETACH 请求

75-77 图17-14中的unp_detach函数完成PRU_DETACH请求的所有处理工作。

1. 释放v-node

303-307 如果插口与一个v-node相联系，那么将指向PCB结构的指针置为0，并且调用vrele释放v_node。

uipc_usrreq.c

```
299 void
300 unp_detach(unp)
301 struct unpcb *unp;
302 {
303     if (unp->unp_vnode) {
304         unp->unp_vnode->v_socket = 0;
305         vrele(unp->unp_vnode);
306         unp->unp_vnode = 0;
307     }
308     if (unp->unp_conn)
309         unp_disconnect(unp);
310     while (unp->unp_refs)
311         unp_drop(unp->unp_refs, ECONNRESET);
312     soisdisconnected(unp->unp_socket);
313     unp->unp_socket->so_pcb = 0;
314     m_freem(unp->unp_addr);
315     (void) m_free(dtom(unp));
316     if (unp_rights) {
317         /*
318          * Normally the receive buffer is flushed later,
319          * in sofree, but if our receive buffer holds references
320          * to descriptors that are now garbage, we will dispose
321          * of those descriptor references after the garbage collector
322          * gets them (resulting in a "panic: closef: count < 0").
323          */
324         sorflush(unp->unp_socket);
325         unp_gc();
326     }
327 }
```

uipc_usrreq.c

图17-14 unp_detach 函数

2. 如果插口连接了其他插口，则断开连接

308-309 如果关闭的插口连接到另一个插口上，那么unp_disconnect就要断开这两个插口的连接，这种情况在流和数据报插口中都会发生。

3. 断开连接到关闭插口的插口

310-311 如果其他的数据报插口连接到这个插口，则调用unp_drop断开这些连接，那些插口就会接收到ECONNRESET错误。while循环检查连接到这个unpcb的所有unpcb结构链表。函数unp_drop调用unp_disconnect，它改变PCB的unp_refs单元去指向链表的下一个单元。当整个链表已经被处理后，PCB的unp_refs指针将为0。

312-313 被关闭的插口由soisdisconnect断开连接，指向PCB的socket结构中的指针置为0。

4. 释放地址和PCB mbuf

314-315 如果插口已经绑定到一个地址，m_freem就释放存储这个地址的mbuf。注意程序不检查unp_addr是否为空，因为m_freem会检查。unpcb由m_free来释放。

这个对m_free的调用应当移到函数的末尾，因为指针unp可能会在下一段程序里使用。

5. 检查被传送的描述符

316-326 如果内核里任何进程传来了描述符，则unp_rights为非0，这会导致调用

sorflush和unp_gc(无用单元收集函数)。我们将在第18章中讨论描述符的传送。

17.9 PRU_BIND请求和unp_bind函数

可以通过bind将Unix 域中的流和数据报插口绑定到文件系统中的路径名上， bind系统调用产生PRU_BIND请求，如图17-15所示。

uipc_usrreq.c

```
78      case PRU_BIND:
79          error = unp_bind(unp, nam, p);
80          break;
```

uipc_usrreq.c

图17-15 PRU_BIND 请求

78-80 所有的工作都由unp_bind函数来完成，如图17-16所示。

1. 初始化nameidata结构

338-339 unp_bind分配一个nameidata结构，这个结构封装所有传给 namei函数的参数，并使用NDINIT宏来初始化这个结构。 CREATE参数指定要创建的路径名， FOLLOW允许紧跟的符号连接， LOCKPARENT指明在返回时必须要锁定父亲的v-node (防止我们在完成工作之前其他进程修改v-node)。UIO_SYSSPACE指明路径名在内核中(由于bind系统调用将路径名从用户空间复制到一个mbuf中)。soun->sun_path是路径名的起始地址(它被作为nam参数传送给unp_bind)。最后， p是指向发布bind系统调用的进程的proc结构的指针，这个结构包含所有有关一个进程的信息，内核需要一直将该进程存放在内存中。 NDINIT宏仅仅初始化这个结构，对namei的调用在这个函数后面。

uipc_usrreq.c

```
328 int
329 unp_bind(unp, nam, p)
330 struct unpcb *unp;
331 struct mbuf *nam;
332 struct proc *p;
333 {
334     struct sockaddr_un *soun = mtod(nam, struct sockaddr_un *);
335     struct vnode *vp;
336     struct vattr vattr;
337     int     error;
338     struct nameidata nd;

339     NDINIT(&nd, CREATE, FOLLOW | LOCKPARENT, UIO_SYSSPACE, soun->sun_path, p);
340     if (unp->unp_vnode != NULL)
341         return (EINVAL);
342     if (nam->m_len == MLEN) {
343         if (*(mtod(nam, caddr_t) + nam->m_len - 1) != 0)
344             return (EINVAL);
345     } else
346         *(mtod(nam, caddr_t) + nam->m_len) = 0;
347 /* SHOULD BE ABLE TO ADOPT EXISTING AND wakeup() ALA FIFO's */
348     if (error = namei(&nd))
349         return (error);
350     vp = nd.ni_vp;
351     if (vp != NULL) {
352         VOP_ABORTOP(nd.ni_dvp, &nd.ni_cnd);
353         if (nd.ni_dvp == vp)
```

图17-16 unp_bind 函数

```
354             vrele(nd.ni_dvp);
355         else
356             vput(nd.ni_dvp);
357         vrele(vp);
358         return (EADDRINUSE);
359     }
360     VATTR_NULL(&vattr);
361     vattr.va_type = VSOCK;
362     vattr.va_mode = ACCESSPERMS;
363     if (error = VOP_CREATE(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr))
364         return (error);
365     vp = nd.ni_vp;
366     vp->v_socket = unp->unp_socket;
367     unp->unp_vnode = vp;
368     unp->unp_addr = m_copy(nam, 0, (int) M_COPYALL);
369     VOP_UNLOCK(vp, 0, p);
370     return (0);
371 }
```

uipc_usrreq.c

图17-16 (续)

历史上，在文件系统中查询路径名的函数名一直是 namei，它代表“ name-to-inode ”。这个函数要搜索整个文件系统去查找指定的名字，如果成功，就初始化内核中的inode结构，这个结构包含从磁盘上得到的文件的 i_node信息的副本。尽管 v-node已经取代了i-node，但是术语namei仍然保留了下来。

这是我们第一次涉及到BSD内核中文件系统代码。BSD内核支持许多不同的文件系统类型：标准的磁盘文件系统 (有时也叫作“ 快速文件系统 ”)，网络文件系统(NFS)，CD-ROM文件系统，MS-DOS文件系统，基于存储器的文件系统 (对于目录，例如/tmp)，等等。[Kleiman 1986]描述了一个早期的v-node实现。以VOP_作为名字开始的函数一般是v-node操作函数。这样的函数大约有 40个，当被调用时，每个函数调用一个文件系统定义的函数去执行这个操作。以一个小写字母 v开头的函数是内核函数，这些函数可能调用一个或更多的 VOP_函数。例如，vput调用VOP_UNLOCK，然后再调用vrele。vrele函数释放一个v-node:v-node的引用计数器递减，如果达到0，就调用VOP_INACTIVE。

2. 检查插口是否被绑定

340-341 如果插口PCB的unp_vnode非空，插口就已经被绑定，这是一个错误。

3. 以空字符(null)结束的路径名

342-346 如果包含sockaddr_un结构的mbuf长度是108(MLEN)，长度值是从bind系统调用的第三个参数复制的，则 mbuf的最后一个字节必须是一个空字节。这就保证路径名以空字符结尾，当在文件系统中查找路径名时这是必需的 (卷2图15-20中的sockargs函数保证由进程传送的插口地址结构长度不超过 108字节)。如果mbuf的长度小于108个字节，则在路径名的结尾存放一个空字节，以免进程没有以空字符来结束路径名。

4. 在文件系统中查找路径名

347-349 namei在文件系统中查找路径名，并且尽可能在相应的目录中为指定的路径名创建一个记录项。例如，如果绑定到插口的路径名是 /tmp/.Xll-unix/X0，那么文件名X0必

须被加到目录 /tmp/.Xll-unix中，包含X0的记录项的目录叫作父目录。如果目录/tmp/.Xll-unix不存在，或者如果存在，但是已经包含一个 X0的文件，那么就要返回一个错误。另一个可能的错误是调用进程没有权限在父目录中创建一个新的文件。从 namei想得到的结果是从函数返回一个 0值，nd.ni_vp返回的是一个空指针 (文件不存在)。如果这两个条件都正确，那么nd.ni_dvp就包含要创建新文件名的加锁父目录。

347行的注释指的是如果路径名已经存在将导致 bind返回错误。所以大部分绑定Unix域插口的应用程序在调用bind之前先调用unlink删除已存在的路径名。

5. 路径名已经存在

350-359 如果nd.ni_vp非空，那么路径名就已经存在。 v-node引用被释放，并且返回EADDRINUSE给进程。

6. 创建v-node

360-365 VATTR_NULL宏初始化vattr结构，类型被设置为VSOCK(一个插口)，访问模式设置为八进制777(ACCESSPERMS)。这九个权限比特允许文件所有者、组里的成员和其他用户(也就是每一个用户)执行读、写和执行操作。在指定的目录中，文件由文件系统的创建函数间接通过VOP_CREATE函数创建。传递给创建函数的参数是 nd.ni_dvp(父目录v-node的指针)，nd.ni_cnd(来自namei需要传送给VOP函数的附加信息)，以及vattr结构。第二个参数nd.ni_vp接收返回信息，nd.ni_vp指向新创建的v-node(如果创建成功)。

7. 链接结构

365-367 vnode和socket通过v_socket和unp_vnode指针互相指向对方。

8. 保存路径名

368-371 调用m_copy将刚刚绑定到插口的路径名复制到一个 mbuf中，PCB的unp_addr指向这个新的mbuf。将v-node解锁。

17.10 PRU_CONNECT请求和unp_connect函数

图17-17描述了PRU_LISTEN和PRU_CONNECT请求。

uipc_usrreq.c

```
81      case PRU_LISTEN:
82          if (unp->unp_vnode == 0)
83              error = EINVAL;
84          break;

85      case PRU_CONNECT:
86          error = unp_connect(so, nam, p);
87          break;
```

uipc_usrreq.c

图17-17 PRU_LISTEN 和PRU_CONNECT 请求

1. 验证监听插口是否已经被绑定

81-84 只能在一个已经绑定了一个路径名的插口上执行 listen系统调用。TCP没有这个需求，在卷2图30-3我们看到对一个没有绑定的 TCP插口调用listen时，TCP就会选择一个临时的端口，并把它分配给插口。

85-87 PRU_CONNECT请求的所有处理工作都由unp_connect函数来执行，函数的第一部分如图17-18所示。对于流插口，该函数被 PRU_CONNECT请求调用；当临时连接一个无连接

的数据报插口时，该函数被PRU_SEND请求调用。

uipc_usrreq.c

```
372 int
373 unp_connect(so, nam, p)
374 struct socket *so;
375 struct mbuf *nam;
376 struct proc *p;
377 {
378     struct sockaddr_un *soun = mtod(nam, struct sockaddr_un *);
379     struct vnode *vp;
380     struct socket *so2, *so3;
381     struct unpcb *unp2, *unp3;
382     int     error;
383     struct nameidata nd;
384     NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF, UIO_SYSSPACE, soun->sun_path, p);
385     if (nam->m_data + nam->m_len == &nam->m_dat[MLEN]) {    /* XXX */
386         if (*(mtod(nam, caddr_t) + nam->m_len - 1) != 0)
387             return (EMSGSIZE);
388     } else
389         *(mtod(nam, caddr_t) + nam->m_len) = 0;
390     if (error = namei(&nd))
391         return (error);
392     vp = nd.ni_vp;
393     if (vp->v_type != VSOCK) {
394         error = ENOTSOCK;
395         goto bad;
396     }
397     if (error = VOP_ACCESS(vp, VWRITE, p->p_ucred, p))
398         goto bad;
399     so2 = vp->v_socket;
400     if (so2 == 0) {
401         error = ECONNREFUSED;
402         goto bad;
403     }
404     if (so->so_type != so2->so_type) {
405         error = EPROTOTYPE;
406         goto bad;
407     }
```

uipc_usrreq.c

图17-18 unp_connect 函数：第一部分

2. 初始化用作路径名查找的nameidata结构

383-384 nameidata结构由NDINIT宏进行初始化。LOOKUP参数指明应当查找的路径名，FOLLOW允许紧跟的符号连接，LOCKLEAF参数指明返回时必须锁定v-node(防止在执行结束前其他进程修改这个v-node)，UIO_SYSSPACE参数指明路径名在内核中，soun->sun_path是路径名的起始地址(它被作为nam参数传递给unp_connect)。p指向发布connect或sendto系统调用的进程的proc结构。

3. 以空字节结束路径名

385-389 如果插口地址结构的长度是108字节，最后一个字节必须为空，否则在路径名的结尾要存储一个空字节。

这段代码与图17-16中的代码相似，但实际上是不同的。不仅第一个if语句不同，而且当最后一个字节非空时返回的错误也不同：这里是EMSGSIZE，而图17-16中是

EINVAL。另外，这个测试对检查数据是否包含在一个簇中有负面影响，虽然这可能是偶然的，因为sockargs函数从来不会把插口地址结构放进一个簇中。

4. 查找路径名并检验其正确性

390-398 namei在文件系统中查找路径名，如果返回值是OK，那么在nd.ni_vp中就返回vnode结构的指针。v-node的类型必须是VSOCK，并且当前进程对插口一定要有写权限。

5. 验证插口是否已绑定到路径名

399-403 一个插口当前必须被绑定到路径名上，这就是说，在v-node中的v_socket指针必须非空。如果情况不是这样，连接就要被拒绝。如果服务器当前没有运行，但是在上一次运行时路径名留在文件系统中，这种情况就有可能发生。

6. 验证插口类型

404-407 连接的客户进程插口(so)的类型必须与被连接的服务器进程插口(so2)的类型相同。也就是说，一个流插口不能连接到一个数据报插口或者相反。

图17-19描述了unp_connect函数的剩余部分，它首先处理连接流插口，然后调用unp_connect2去链接两个unpcb结构。

uipc_usrreq.c

```
408     if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
409         if ((so2->so_options & SO_ACCEPTCONN) == 0 ||
410             (so3 = sonewconn(so2, 0)) == 0) {
411             error = ECONNREFUSED;
412             goto bad;
413         }
414         unp2 = sotounpcb(so2);
415         unp3 = sotounpcb(so3);
416         if (unp2->unp_addr)
417             unp3->unp_addr =
418                 m_copy(unp2->unp_addr, 0, (int) M_COPYALL);
419         so2 = so3;
420     }
421     error = unp_connect2(so, so2);
422   bad:
423     vput(vp);
424     return (error);
425 }
```

uipc_usrreq.c

图17-19 unp_connect 函数：第二部分

7. 连接流插口

408-415 流插口需要特殊处理，因为必须根据监听插口创建一个新的插口。首先，服务器插口必须是监听插口：SO_ACCEPTCONN标志必须被设置(由卷2图15-24的solisten函数来完成)。然后调用sonewconn创建一个新的插口，sonewconn还把这个新的插口放到监听插口的未完成的连接队列中。

8. 复制绑定到监听插口的名字

416-418 如果监听插口包含一个指向mbuf的指针，mbuf包含一个sockaddr_un，并且sockaddr_un带有绑定到插口的路径名(这应当总是对的)，那么调用m_copy将该mbuf复制给新创建的插口。

图17-20给出了在so2=so3赋值之前的不同结构的状态，步骤如下：

- 服务器进程调用socket创建最右边的file、socket和unpcb结构，然后调用bind创建对vnode和包含路径名的mbuf的引用。随后调用listen，允许客户进程发起连接。
- 客户进程调用socket创建最左边的file、socket和unpcb结构，然后调用connect，connect调用unp_connect。
- 我们称中间的socket结构为“已连接的服务器插口”，它由sonewconn创建，

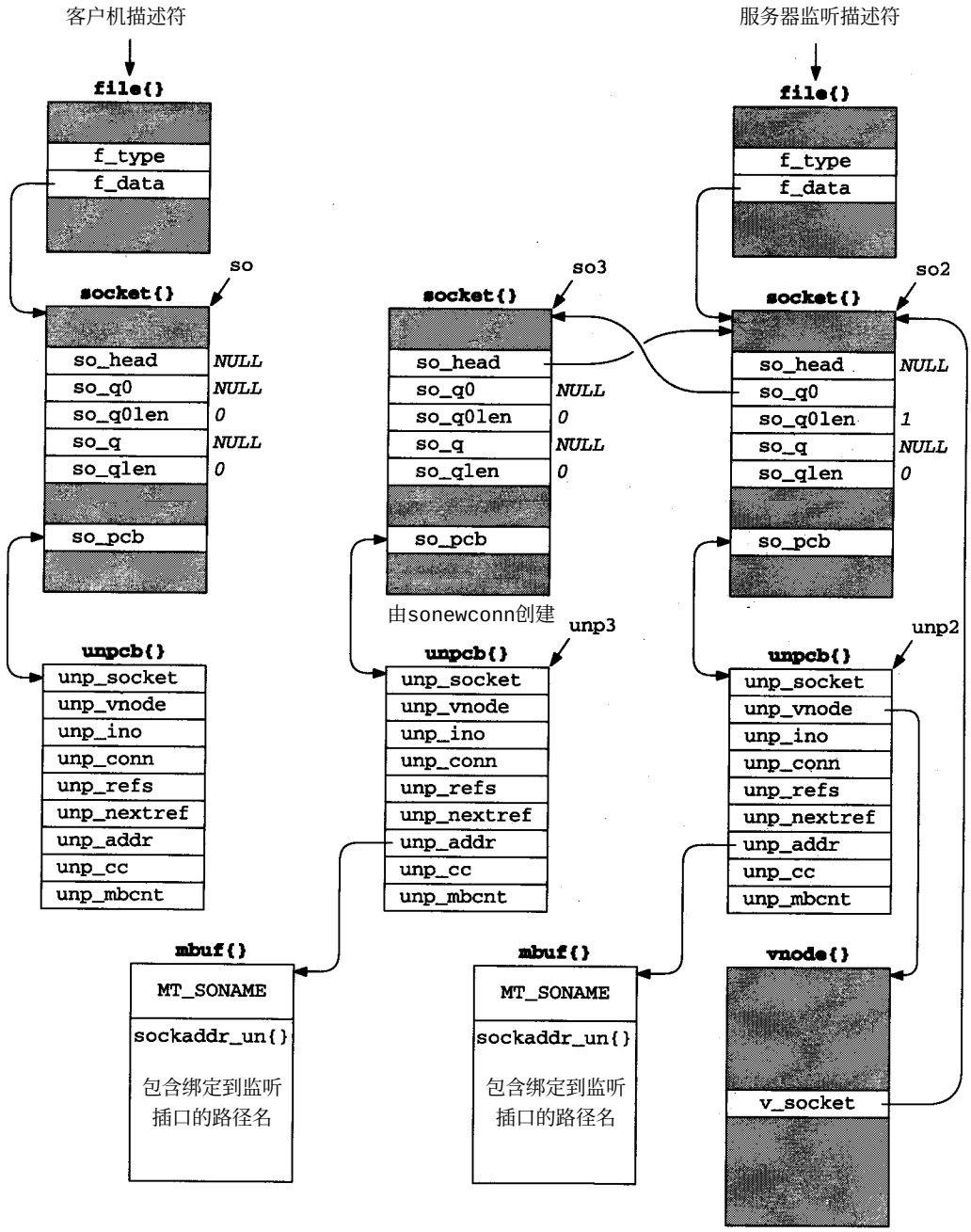

图17-20 流插口的connect 调用中的各种结构

sonewconn创建完该结构后发出PRU_ATTACH请求，创建相应的unpcb结构。

- sonewconn也调用soqinseque将刚产生的socket放入监听插口的未完成的连接队列中(我们假定队列开始是空的)。我们还看到监听插口的已完成连接队列(so_q和so_qlen)为空，新建socket的so_head指针反过来指向监听插口。
- unp_connect调用m_copy创建包含绑定到监听插口的路径名的mbuf的副本，中间的unpcb指向这个mbuf。我们将看到getpeername系统调用需要这个副本。
- 最后要注意的是，还没有一个file结构指向新建的socket(事实上是通过sonewconn设置SS_NOFDREF标志来说明这一点的)。当监听服务器进程调用accept时，就会给该socket分配一个file结构和对应的文件描述符。

vnode指针没有从监听插口复制到连接的服务器插口。vnode结构的唯一作用就是允许客户进程通过v_socket指针调用connect定位相应的服务器的socket结构。

9. 连接两个流或数据报插口

421 unp_connect中的最后一步是调用unp_connect2(下一节描述)，这对于流和数据报插口是一样的。就图17-20而言，该函数连接最左边的两个unpcb结构的unp_conn字段，并且将新创建的插口从监听服务器的socket的未完成连接队列移到已完成连接队列中，我们将在后面的章节中描述最终的数据结构(图17-26)。

17.11 PRU_CONNECT2请求和unp_connect2函数

图17-21中的PRU_CONNECT2请求仅仅作为socketpair系统调用产生的一个结果，而且这个请求只在Unix域中得到支持。

uipc_usrreq.c

```
88      case PRU_CONNECT2:
89          error = unp_connect2(so, (struct socket *) nam);
90          break;
```

uipc_usrreq.c

图17-21 PRU_CONNECT2 请求

88-90 这个请求的所有处理工作都由unp_connect2函数来完成，正如我们在图17-22中看到的一样，unp_connect2函数又是从内核中的其他两个地方调用的。

我们将在17.12节介绍socketpair系统调用和soconnect2函数，在17.13节介绍pipe系统调用。图17-23描述了unp_connect2函数。

1. 检验插口类型

426-434 两个参数都是指向socket结构的指针：so连接到so2。首先检查两个插口的类型是否相同：是流插口或者是数据报插口。

2. 把第一个插口连接到第二个插口

435-436 通过字段unp_conn将第一个unpcb连接到第二个unpcb，然而，下面的步骤在流和数据报之间是不同的。

3. 连接数据报插口

438-442 PCB的unp_nextref和unp_refs字段连接数据报插口。例如，考虑一个绑定了路径名/tmp/foo的数据报服务器插口，然后一个数据报客户进程连接到这个路径名。图17-24给出了在unp_connect2返回后得到的unpcb结构(为了简便起见，我们没有描述相应

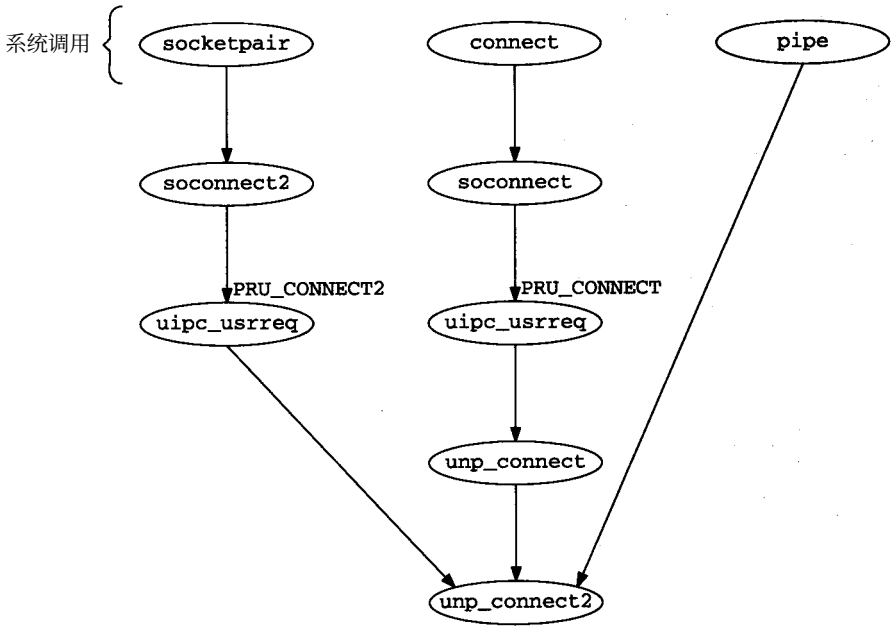

图17-22 unp_connect2 函数的调用者

uipc_usrreq.c

```
426 int
427 unp_connect2(so, so2)
428 struct socket *so;
429 struct socket *so2;
430 {
431     struct unpcb *unp = sotounpcb(so);
432     struct unpcb *unp2;
433     if (so2->so_type != so->so_type)
434         return (EPROTOTYPE);
435     unp2 = sotounpcb(so2);
436     unp->unp_conn = unp2;
437     switch (so->so_type) {
438     case SOCK_DGRAM:
439         unp->unp_nextref = unp2->unp_refs;
440         unp2->unp_refs = unp;
441         soisconnected(so);
442         break;
443     case SOCK_STREAM:
444         unp2->unp_conn = unp;
445         soisconnected(so);
446         soisconnected(so2);
447         break;
448     default:
449         panic("unp_connect2");
450     }
451     return (0);
452 }
```

uipc_usrreq.c

图17-23 unp_connect2 函数

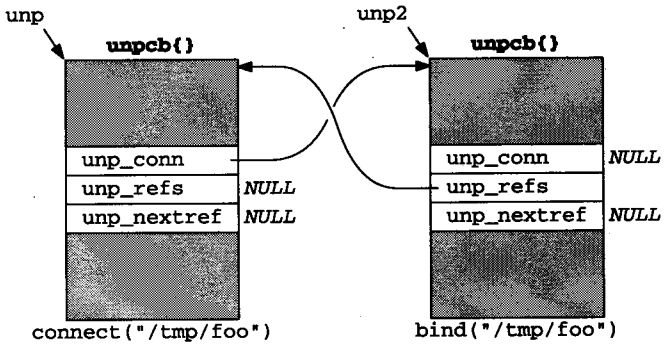

图17-24 连接的数据报插口

的file或socket结构，或者与最右边插口相连接的 vnode)。我们描述了在unp_connect2中用到的两个指针unp和unp2。

对于一个已经有连接的数据报插口，unp_refs指向连接到该插口的所有插口的链表的第一个PCB。通过unp_nextref指针遍历这个链表。

图17-25表示了第三个数据报插口 (左边的那个)连接到同一服务器后的三个 PCB的状态，绑定路径名都是/tmp/foo。

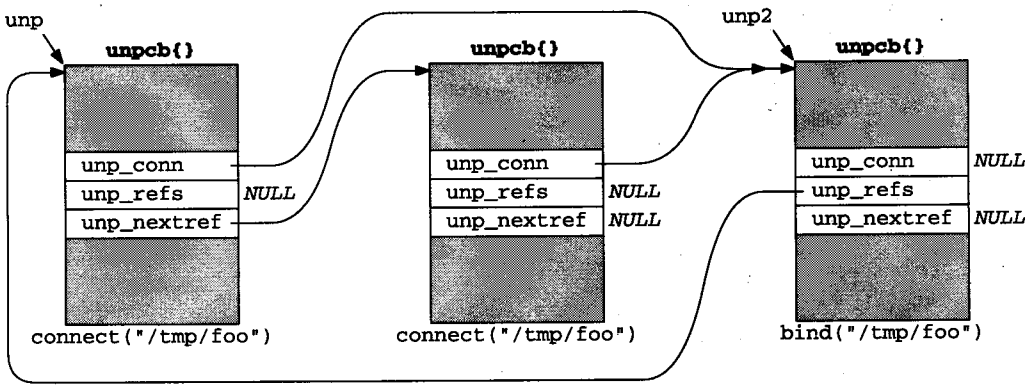

图17-25 另一个插口(左边)连接到右边的插口

两个PCB字段unp_refs和unp_nextref必须分开，因为图17-25中右边的插口自己能连接到其他的数据报插口。

4. 连接流插口

443-447 流插口的连接与数据报插口的连接是不同的，这是因为只能有一个客户进程连接到一个流插口上(服务器进程)，客户进程和服务器进程的PCB的unp_conn指针分别指向对方的PCB，如图17-26所示(这个图是图17-20的延续)。

这个图中的另一个变化是对于带有so2参数的soisconnected的调用，这个调用将插口从监听插口的未完成连接队列(图17-20中的so_q0)移到已完成连接队列(so_q)中。accept要从这个队列中获取新创建的插口(卷2图15-34)。需要注意的是，soinconnected(卷2图15-30)设置so_state中的SS_ISCONNECTED标志，仅当插口的so_head指针非空时才将socket从未完成连接队列移到已完成连接队列(如果插口的so_head指针为空时，插口不在任何一个队列中)。所以，在图17-23中，对带有so参数的soisconnected的第一次调用仅仅改变so_state。

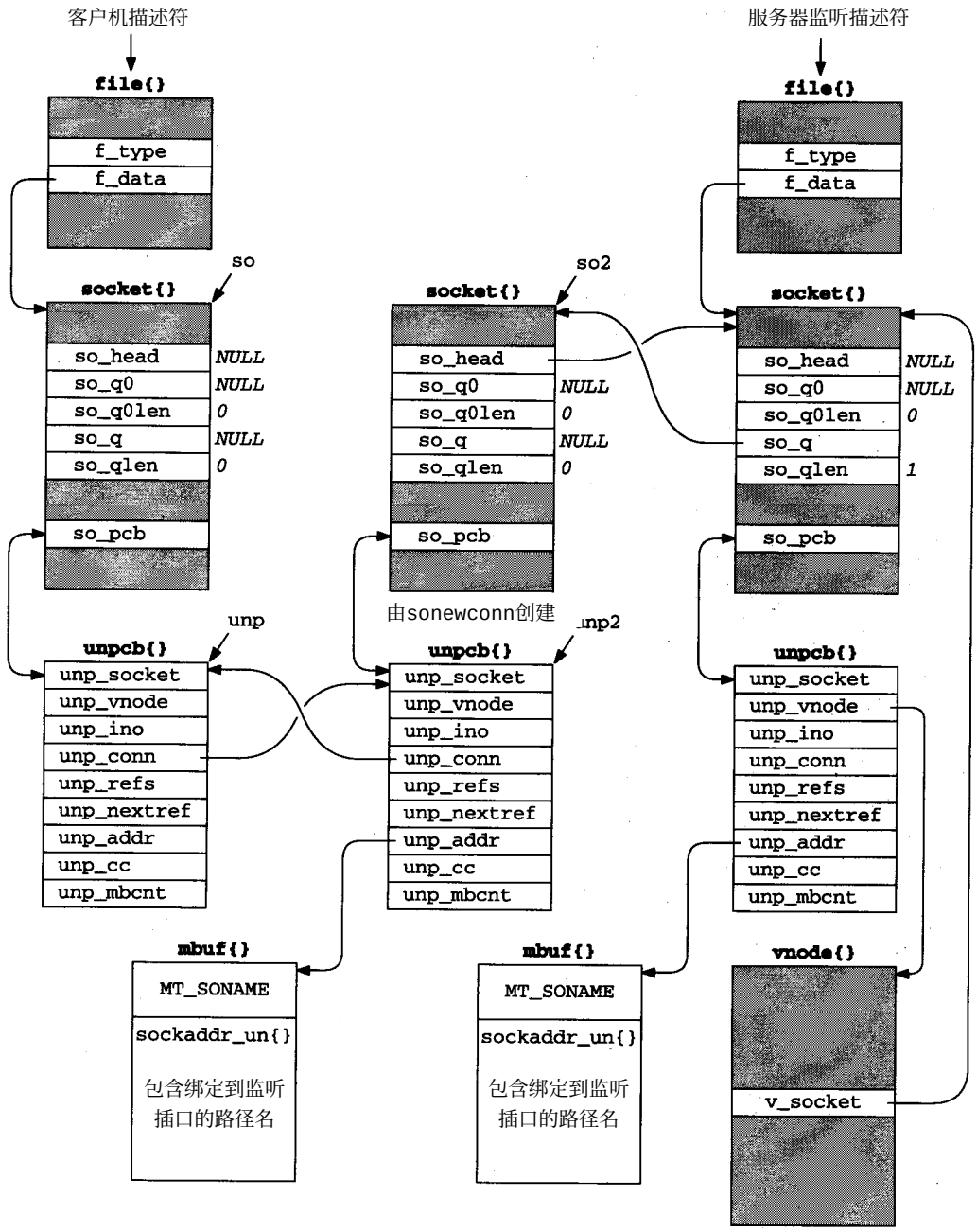

图17-26 已建连的流插口

17.12 socketpair系统调用

socketpair系统调用仅在Unix域中得到支持。它创建两个插口并连接它们，同时返回两个描述符，互相连接在一起。例如，一个用户进程发出调用：

```
int   fd[2];
socketpair(PF_UNIX, SOCK_STREAM, 0, fd);
```

创建一对连接在一起的全双工Unix域流插口。在fd[0]中返回第一个描述符，在fd[1]中返回第二个描述符。如果第二个参数是SOCK_DGRAM，则创建一对互相连接的Unix域数据报插口。如果调用成功，socketpair返回0；否则，返回-1。

图17-27描述了socketpair系统调用的实现。

uipc_syscalls.c

```
229 struct socketpair_args {
230     int     domain;
231     int     type;
232     int     protocol;
233     int    *rsv;
234 };

235 socketpair(p, uap, retval)
236 struct proc *p;
237 struct socketpair_args *uap;
238 int     retval[];
239 {
240     struct filedesc *fdp = p->p_fd;
241     struct file *fp1, *fp2;
242     struct socket *so1, *so2;
243     int     fd, error, sv[2];

244     if (error = socreate(uap->domain, &so1, uap->type, uap->protocol))
245         return (error);
246     if (error = socreate(uap->domain, &so2, uap->type, uap->protocol))
247         goto free1;

248     if (error = falloc(p, &fp1, &fd))
249         goto free2;
250     sv[0] = fd;
251     fp1->f_flag = FREAD | FWRITE;
252     fp1->f_type = DTYPE_SOCKET;
253     fp1->f_ops = &socketops;
254     fp1->f_data = (caddr_t) so1;

255     if (error = falloc(p, &fp2, &fd))
256         goto free3;
257     fp2->f_flag = FREAD | FWRITE;
258     fp2->f_type = DTYPE_SOCKET;
259     fp2->f_ops = &socketops;
260     fp2->f_data = (caddr_t) so2;
261     sv[1] = fd;

262     if (error = soconnect2(so1, so2))
263         goto free4;
264     if (uap->type == SOCK_DGRAM) {
265         /*
266          * Datagram socket connection is asymmetric.
267          */
268         if (error = soconnect2(so2, so1))
269             goto free4;
270     }
271     error = copyout((caddr_t) sv, (caddr_t) uap->rsv, 2 * sizeof(int));
272     retval[0] = sv[0];          /* XXX ??? */
273     retval[1] = sv[1];          /* XXX ??? */
274     return (error);

275 free4:
```

图17-27 socketpair 系统调用

```
276     ffree(fp2);
277     fdp->fd_ofiles[sv[1]] = 0;
278   free3:
279     ffree(fp1);
280     fdp->fd_ofiles[sv[0]] = 0;
281   free2:
282     (void) soclose(so2);
283   free1:
284     (void) soclose(so1);
285     return (error);
286 }
```

uipc_syscalls.c

图17-27 (续)

1. 参数

229-239 四个整型参数，从domian到rsv，在本节开始部分用户调用socketpair的例子中进行了描述。函数socketpair定义中描述的三个参数(p、uap和retval)是传送到内核中的系统调用的参数。

2. 创建两个插口和两个描述符

244-261 调用socreate两次，创建两个插口。两个描述符中的第一个由falloc分配。在fd中返回描述符的值，而指向相应file结构的指针在fp1中返回。设置FREAD和FWRITE标

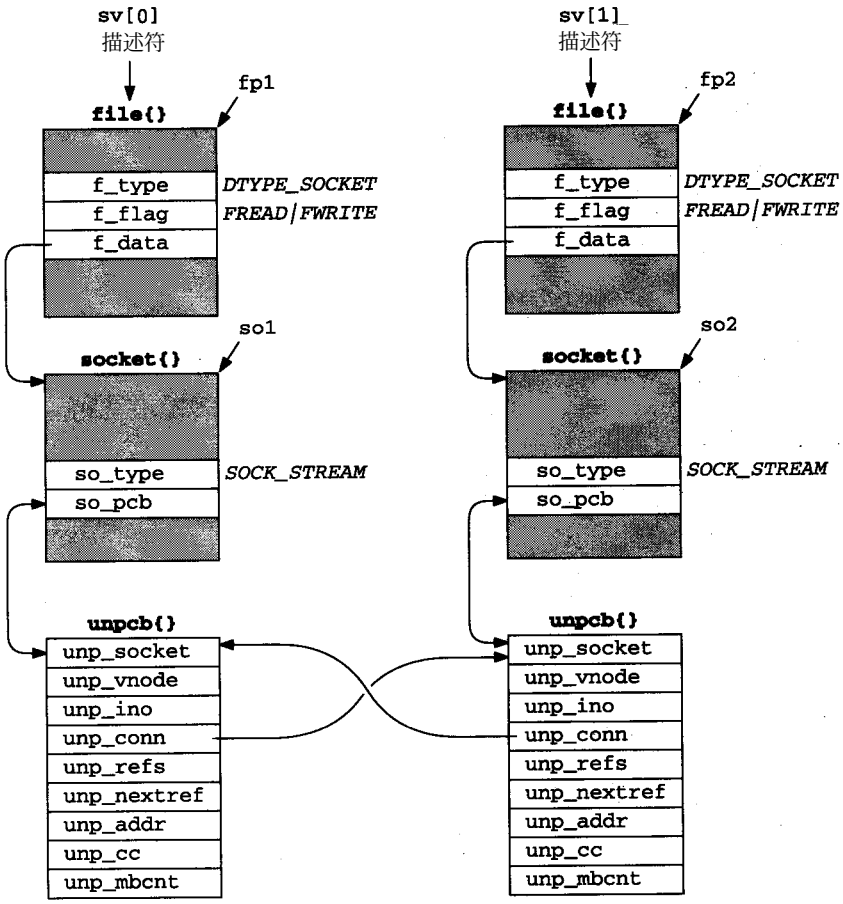

图17-28 由sockerpair 创建的两个流插口

志 (由于插口是全双工的)，文件类型设置为DTYPE_SOCKET，设置f_ops指向五个插口函数指针的数组(卷2图15-13)，设置f_data指向socket结构。第二个描述符由falloc分配，并且初始化相应的file结构。

3. 连接两个插口

262-270 soconnect2发出PRU_CONNECT2请求，这个请求仅在Unix域中得到支持。如果系统调用正在创建流插口，立即从soconnect2中返回。此时的结构如图17-28所示。

如果创建两个数据报插口，就需要调用soconnect2两次，每一次调用连接一个方向。两次调用以后，我们就有了图17-29中的结构。

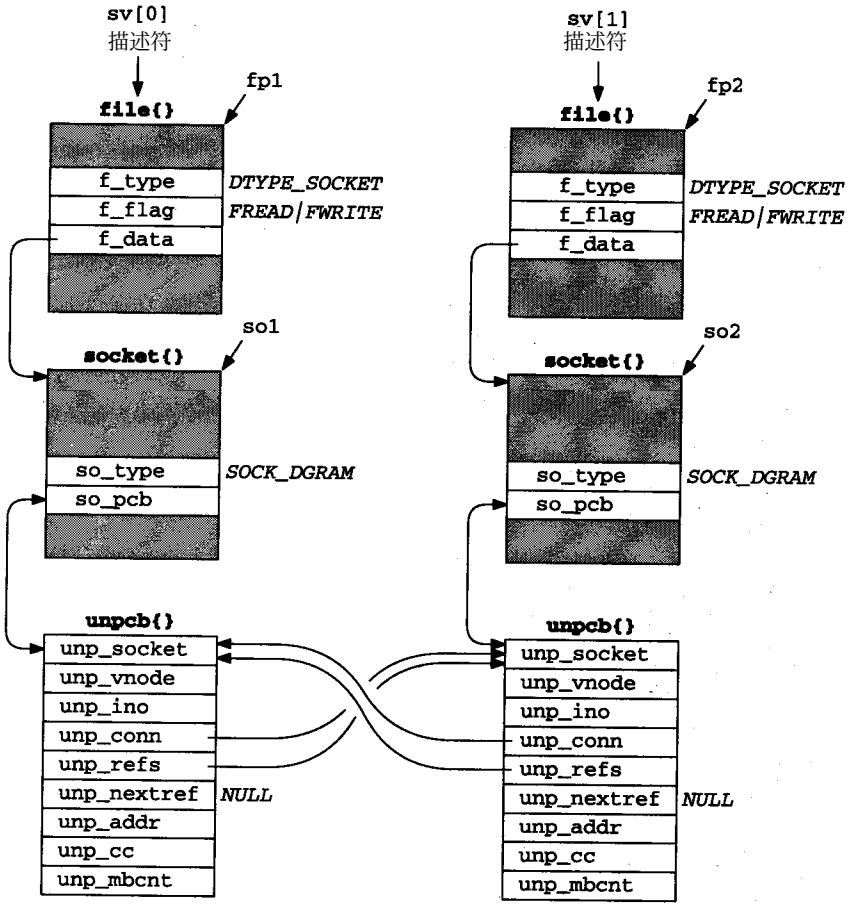

图17-29 由sockerpair 创建的两个数据报插口

4. 将两个描述符复制给进程

271-274 copyout将两个描述符复制给进程。

带有注释ＸＸＸ ？？的两个表达式第一次出现在 4.3BSD Reno版本里。因为copyout把两个描述符复制给进程，所以不需要这两个表达式。我们将看到 pipe系统调用通过设置retval[0]和retval[1]返回两个描述符，其中retval是系统调用的第三个参数。内核中处理系统调用的汇编子程序总是将两个整数 retval[0]和retval[1]放在机器的寄存器里作为任何系统调用的返回值。但是在用户进程中，

激活系统调用的汇编子程序必须查看这些寄存器并且返回进程希望得到的值。 C函数库中的pipe函数实际上是这样做的，但是socketpair函数并不这么做。

5. soconnect2函数

图17-30中的函数发出PRU_CONNECT2请求，该函数仅在socketpair系统调用中被调用。

uipc_socket.c

```
225 soconnect2(so1, so2)
226 struct socket *so1;
227 struct socket *so2;
228 {
229     int     s = splnet();
230     int     error;

231     error = (*so1->so_proto->pr_usrreq) (so1, PRU_CONNECT2,
232                 (struct mbuf *) 0, (struct mbuf *) so2, (struct mbuf *) 0);
233     splx(s);
234     return (error);
235 }
```

uipc_socket.c

图17-30 soconnect2 函数

17.13 pipe系统调用

图17-31中的pipe系统调用与socketpair系统调用几乎相同。

uipc_syscalls.c

```
645 pipe(p, uap, retval)
646 struct proc *p;
647 struct pipe_args *uap;
648 int     retval[];
649 {
650     struct filedesc *fdp = p->p_fd;
651     struct file *rf, *wf;
652     struct socket *rso, *wso;
653     int     fd, error;

654     if (error = socreate(AF_UNIX, &rso, SOCK_STREAM, 0))
655         return (error);
656     if (error = socreate(AF_UNIX, &wso, SOCK_STREAM, 0))
657         goto free1;
658     if (error = falloc(p, &rf, &fd))
659         goto free2;
660     retval[0] = fd;
661     rf->f_flag = FREAD;
662     rf->f_type = DTYPE_SOCKET;
663     rf->f_ops = &socketops;
664     rf->f_data = (caddr_t) rso;
665     if (error = falloc(p, &wf, &fd))
666         goto free3;
667     wf->f_flag = FWRITE;
668     wf->f_type = DTYPE_SOCKET;
669     wf->f_ops = &socketops;
670     wf->f_data = (caddr_t) wso;
671     retval[1] = fd;
672     if (error = unp_connect2(wso, rso))
673         goto free4;
```

图17-31 pipe 系统调用

```
674     return (0);
675   free4:
676     ffree(wf);
677     fdp->fd_ofiles[retval[1]] = 0;
678   free3:
679     ffree(rf);
680     fdp->fd_ofiles[retval[0]] = 0;
681   free2:
682     (void) soclose(wso);
683   free1:
684     (void) soclose(rso);
685     return (error);
686 }
```

uipc_syscalls.c

图17-31 (续)

654-686 调用socreate创建两个Unix域流插口，pipe系统调用与socketpair系统调用的唯一差别就是pipe把两个描述符中的第一个设置成只读 (read-only)，把第二个设置成只写 (write_only)；两个描述符由 retval参数返回，而不是通过 copyout；pipe直接调用unp_connect2，而不是通过soconnect2函数。

Unix的一些版本，特别是SVR4，创建的管道两端均可进行读写。

17.14 PRU_ACCEPT请求

对于一个流插口，接受一个新的连接所需的大部分处理工作由其他内核函数完成：sonewconn创建新的socket结构，并发出PRU_ATTACH请求，accept系统调用将插口从已完成连接队列中删除并调用 soaccept。soaccept(卷2)仅发出PRU_ACCEPT请求，用于Unix域的PRU_ACCEPT请求。如图17-33所示。

返回客户进程的路径名

94-108 如果客户进程调用bind，并且同客户进程的连接仍然存在，那么这个请求把含有客户进程路径名的 sockaddr_un复制到由 nam参数指向的 mbuf。否则，返回空路径名 (sun_nonname)。

```
91      case PRU_DISCONNECT:
92          unp_disconnect(unp);
93          break;
```

uipc_usrreq.c

图17-32 PRU_DISCONNECT 请求

```
94      case PRU_ACCEPT:
95          /*
96           * Pass back name of connected socket,
97           * if it was bound and we are still connected
98           * (our peer may have closed already!).
99           */
100         if (unp->unp_conn && unp->unp_conn->unp_addr) {
101             nam->m_len = unp->unp_conn->unp_addr->m_len;
102             bcopy(mtod(unp->unp_conn->unp_addr, caddr_t),
```

uipc_usrreq.c

图17-33 PRU_ACCEPT 请求

```
103                     mtod(nam, caddr_t), (unsigned) nam->m_len);
104             } else {
105                 nam->m_len = sizeof(sun_noname);
106                 *(mtod(nam, struct sockaddr *)) = sun_noname;
107             }
108             break;
```
uipc_usrreq.c

图17-33 (续)

17.15 PRU_DISCONNECT请求和unp_disconnect函数

如果插口已建连，close系统调用就发出PRU_DISCONNECT请求，如图17-32所示。

91-93 p_disconnect函数完成所有的断连工作，如图17-34所示。

uipc_usrreq.c

```
453 void
454 unp_disconnect(unp)
455 struct unpcb *unp;
456 {
457     struct unpcb *unp2 = unp->unp_conn;
458     if (unp2 == 0)
459         return;
460     unp->unp_conn = 0;
461     switch (unp->unp_socket->so_type) {
462     case SOCK_DGRAM:
463         if (unp2->unp_refs == unp)
464             unp2->unp_refs = unp->unp_nextref;
465         else {
466             unp2 = unp2->unp_refs;
467             for (;;) {
468                 if (unp2 == 0)
469                     panic("unp_disconnect");
470                 if (unp2->unp_nextref == unp)
471                     break;
472                 unp2 = unp2->unp_nextref;
473             }
474             unp2->unp_nextref = unp->unp_nextref;
475         }
476         unp->unp_nextref = 0;
477         unp->unp_socket->so_state &= ~SS_ISCONNECTED;
478         break;
479     case SOCK_STREAM:
480         soisdisconnected(unp->unp_socket);
481         unp2->unp_conn = 0;
482         soisdisconnected(unp2->unp_socket);
483         break;
484     }
485 }
```
uipc_usrreq.c

图17-34 unp_disconnect 函数

1. 检查插口是否有连接

458-460 如果插口没有连接到其他插口，则函数立即返回；否则就将 unp_conn置为0。表明这个插口没有连接到其他插口。

2. 将关闭的数据报PCB从链表中删除

462-478 这部分代码把关闭插口的PCB从已连接数据报PCB的链表中删除。例如，如果我们从图17-25开始，然后关闭最左边的插口，就得到图 17-35中的数据结构。由于 `unp2->unp_refs`等于unp(被关闭的PCB是链表的头)，所以被关闭的PCB的`unp_nextref`指针成为新的链表头。

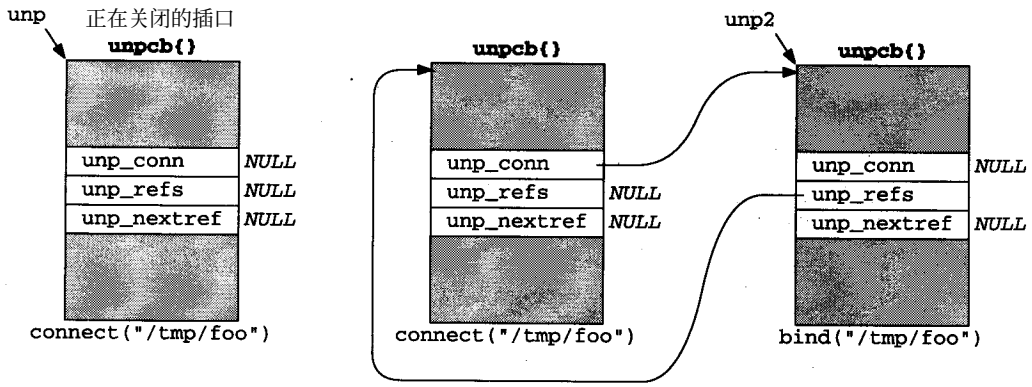

图17-35 最左边插口关闭后图17-25中的链表所发生的变化

如果我们再从图17-25开始，关闭中间的插口，就得到图 17-36中的数据结构。这一次被关闭插口的PCB就不是链表的头。unp2从链表的头开始查看被关闭的 PCB之前的PCB。删除关闭的PCB之后，unp2就指向图17-36中最左边的PCB。然后将关闭PCB的`unp_nextref`指针赋给链表(unp)上前一个PCB的`unp_nextref`。

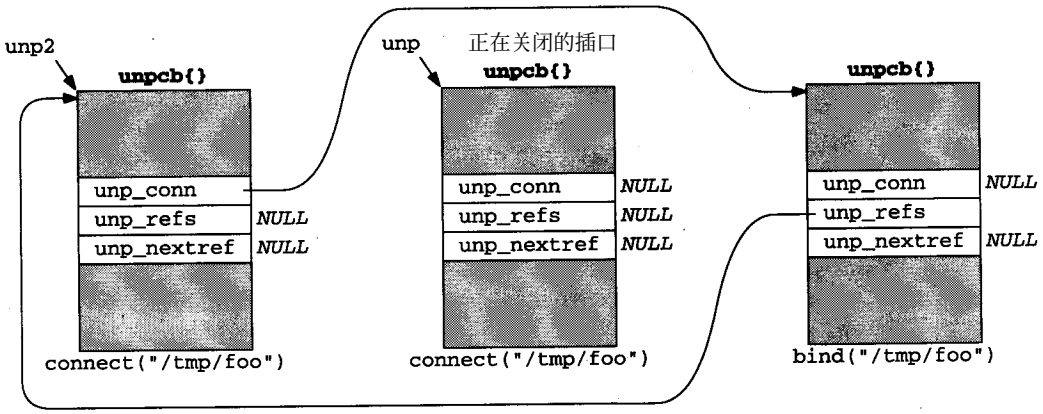

图17-36 中间插口关闭后图17-25中的链表所发生的变化

3. 完成流插口的断连

479-483 由于一个Unix域流插口只能同一个流插口建连，因而不涉及到链表，断开连接就比较简单。将连接对方的 `unp_conn`指针置为 0，并且对客户进程和服务器进程均调用`soisdisconnected`。

17.16 PRU_SHUTDOWN请求和unp_shutdown函数

当进程调用shutdown禁止任何进一步输出时，发出PRU_SHUTDOWN请求，如图17-37所示。

uipc_usrreq.c

```
109     case PRU_SHUTDOWN:
110         socantsendmore(so);
111         unp_shutdown(unp);
112         break;
```

uipc_usrreq.c

图17-37 PRU_SHUTDOWN 请求

109-112 socantsendmore设置插口标志禁止任何进一步的输出，然后调用图 17-38中的unp_shutdown函数。

uipc_usrreq.c

```
494 void
495 unp_shutdown(unp)
496 struct unpcb *unp;
497 {
498     struct socket *so;

499     if (unp->unp_socket->so_type == SOCK_STREAM && unp->unp_conn &&
500         (so = unp->unp_conn->unp_socket))
501         socantrcvmore(so);
502 }
```

uipc_usrreq.c

图17-38 unp_shutdown 函数

如果是流插口通知对等插口

499-502 对于数据报插口不需要再做什么。但是，如果这个插口是流插口，并且还与另一个插口相连，且对等端插口还有一个socket结构，则对对等端插口调用socantrcvmore。

17.17 PRU_ABORT请求和unp_drop函数

如果插口是一个监听插口，并且未完成的连接依然在队列中，那么 soclose就发出PRU_ABORT请求，如图17-39所示。soclose对在未完成连接队列和已完成连接队列中的每一个插口都发出这个请求(卷2图15-39)。

uipc_usrreq.c

```
209     case PRU_ABORT:
210         unp_drop(unp, ECONNABORTED);
211         break;
```

uipc_usrreq.c

图17-39 PRU_ABORT 请求

209-211 图17-40中的unp_drop函数产生一个ECONNABORTED错误，我们在图17-14中看到，unp_detach也调用带有参数ECONNRESET的unp_drop函数。

uipc_usrreq.c

```
503 void
504 unp_drop(unp, errno)
505 struct unpcb *unp;
506 int     errno;
507 {
508     struct socket *so = unp->unp_socket;

509     so->so_error = errno;
```

图17-40 unp_drop 函数

```
510     unp_disconnect(unp);
511     if (so->so_head) {
512         so->so_pcb = (caddr_t) 0;
513         m_freem(unp->unp_addr);
514         (void) m_free(dtom(unp));
515         sofree(so);
516     }
517 }
```

uipc_usrreq.c

图17-40 (续)

1. 保存错误，断开插口连接

509-510 设置插口的so_error值，并且如果插口上有连接，就调用unp_disconnect。

2. 如果插口在监听服务器的队列上，就删除数据结构

511-516 如果插口的so_head指针非空，那么插口当前不是在监听插口的未完成连接队列上，就是在监听插口的已完成连接队列上。从 socket到unpcb的指针都置为0，调用m_freem释放包含绑定到监听插口的路径名的mbuf(回想图17-20)，下一次调用m_free释放unpcb结构。sofree释放socket结构。由于插口在监听服务器的任何一个队列中，所以还没有与它相对应的 file结构，因为该结构是在插口从已完成连接队列中被删除时调用accept分配的。

17.18 其他各种请求

图17-41描述了其余六个尚未讨论的请求。

uipc_usrreq.c

```
212     case PRU_SENSE:
213         ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;
214         if (so->so_type == SOCK_STREAM && unp->unp_conn != 0) {
215             so2 = unp->unp_conn->unp_socket;
216             ((struct stat *) m)->st_blksize += so2->so_rcv.sb_cc;
217         }
218         ((struct stat *) m)->st_dev = NODEV;
219         if (unp->unp_ino == 0)
220             unp->unp_ino = unp_ino++;
221         ((struct stat *) m)->st_ino = unp->unp_ino;
222         return (0);

223     case PRU_RCVOOB:
224         return (EOPNOTSUPP);

225     case PRU_SENDOOB:
226         error = EOPNOTSUPP;
227         break;

228     case PRU_SOCKADDR:
229         if (unp->unp_addr) {
230             nam->m_len = unp->unp_addr->m_len;
231             bcopy(mtod(unp->unp_addr, caddr_t),
232                   mtod(nam, caddr_t), (unsigned) nam->m_len);
233         } else
234             nam->m_len = 0;
235         break;
```

图17-41 其他的PRU_ *xxx*请求

```
236     case PRU_PEERADDR:
237         if (unp->unp_conn && unp->unp_conn->unp_addr) {
238             nam->m_len = unp->unp_conn->unp_addr->m_len;
239             bcopy(mtod(unp->unp_conn->unp_addr, caddr_t),
240                   mtod(nam, caddr_t), (unsigned) nam->m_len);
241         } else
242             nam->m_len = 0;
243         break;

244     case PRU_SLOWTIMO:
245         break;
```

uipc_usrreq.c

图17-41 (续)

1. PRU_SENSE请求

212-217 这个请求是由fstat系统调用发出的。将插口发送缓存高水位标记的当前值赋给stat结构的st_blksize作为返回值。另外，如果这个插口是一个有连接的流插口，那么将对等端插口接收缓存中的字节数加到这个值上。当我们讨论 18.2节中的PRU_SEND请求时会看到，这两个值之和就是两个相连的流插口间的实际“管道”容量。

218 将st_dev置为NODEV(所有比特为全1的常数值，代表一个不存在的设备)。

219-221 I-node号标识文件系统中的文件。该值(stat结构的st_ino字段)是作为一个Unix域插口的i-node号返回的，它是从全局变量 unp_ino得到的一个唯一值。如果还没有为unpcb分配一个这类伪i-node号，就将unp_ino的当前值赋给该unpcb作为其i-node号，然后将unp_ino加1。之所以称这些i-node号为伪i-node号，是因为它们并不是文件系统中的实际文件。它们仅在需要时由一个全局计数器产生。如果要求将 Unix域插口绑定到文件系统中的一个路径名(不是这种情况)，PRU_SENSE请求就能使用st_dev和st_ino值来代替绑定路径名。

全局变量unp_ino的递增应当在赋值之前而不是在赋值之后完成。在内核重启后，对Unix域插口第一次调用fstat时，存储在插口unpcb中的值将为0。但是，如果对相同的插口再次调用 fstat，由于unpcb中的当前值是 0，所以将全局变量unp_ino的非0值保存在其PCB中。

2. PRU_RCVOOB和PRU_SENDOOB请求

223-227 Unix域不支持带外数据。

3. PRU_SOCKADDR请求

228-235 这个请求返回绑定到插口的协议地址 (在Unix域插口中为路径名)。如果路径名绑定到插口，unp_addr就指向包含存储路径名的 sockaddr_un的mbuf。uipc_usrreq的nam参数指向由调用者分配的、用于接收结果的 mbuf。调用m_copy产生插口地址结构的副本。如果路径名没有绑定到插口，那么将 mbuf的长度域设置为0。

4. PRU_PEERADDR请求

236-243 处理这个请求与前一个请求相似，但是期望的路径名是绑定到与发起连接的插口相连的插口的名字。如果发起连接的插口已连接到一个对等端插口，那么 unp_conn非空。

没有绑定路径名的插口对这两个请求的处理与 PRU_ACCEPT请求的处理不同(图17-33)。当没有名字存在时，getsockname和getpeername系统调用通过第三个

参数返回0。而accept函数通过第三个参数返回16，通过第二个参数返回包含在sockaddr_un中由空字节组成的路径名(sun_noname是一个通用的sockaddr结构，它的长度是16个字节)。

5. PRU_SLOWTIMO请求

244-245 由于Unix域协议不使用定时器，所以从来不会发出这个请求。

17.19 小结

我们在本章看到的Unix域协议实现简单直观。它提供了流和数据报插口，其中流协议类似于TCP，数据报协议类似于UDP。

路径名能绑定到Unix域插口。服务器进程绑定其知名的路径名，客户进程连接到这个路径名。数据报插口也可以建连，与UDP一样，多个客户进程可以连接到同一个服务器进程上。Socketpair函数也可以创建尚未命名的Unix域插口。Unix pipe系统调用能创建两个互相连接的Unix域流插口，源于伯克利系统的管道实际上就是Unix域流插口。

与Unix域插口有关的协议控制块是unpcb结构。与其他域不同的是这些PCB并不保存在一个链表中。然而，当一个Unix域插口需要与另一个Unix域插口同步时(connect或sendto)，通过内核中的路径名查找函数namei来定位目的unpcb，函数namei得到一个vnode结构，通过这个结构得到目的unpcb。