

第18章 Unix 域协议：I/O和描述符的传递

18.1 概述

本章继续描述上一章的 Unix域协议实现。本章的第一节讲述 I/O、PRU_SEND和 PRU_RCVD请求，其余部分介绍描述符传递。

18.2 PRU_SEND和PRU_RCVD请求

无论什么时候，当一个进程给 Unix域插口发送数据或者控制信息时都要发出 PRU_SEND请求。请求的第一部分首先处理控制信息，然后处理数据报插口，如图 18-1所示。

```
140     case PRU_SEND:
141         if (control && (error = unp_internalize(control, p)))
142             break;
143         switch (so->so_type) {
144             case SOCK_DGRAM:{
145                 struct sockaddr *from;
146                 if (nam) {
147                     if (unp->unp_conn) {
148                         error = EISCONN;
149                         break;
150                     }
151                     error = unp_connect(so, nam, p);
152                     if (error)
153                         break;
154                 } else {
155                     if (unp->unp_conn == 0) {
156                         error = ENOTCONN;
157                         break;
158                     }
159                 }
160                 so2 = unp->unp_conn->unp_socket;
161                 if (unp->unp_addr)
162                     from = mtod(unp->unp_addr, struct sockaddr *);
163                 else
164                     from = &sun_noname;
165                 if (sbappendaddr(&so2->so_rcv, from, m, control)) {
166                     sorwakeup(so2);
167                     m = 0;
168                     control = 0;
169                 } else
170                     error = ENOBUFS;
171                 if (nam)
172                     unp_disconnect(unp);
173                 break;
174             }
175         }
```

uipc_usrreq.c

uipc_usrreq.c

图18-1 数据报插口的 PRU_SEND 请求

1. 初始化所有控制信息

141-142 如果进程使用 `sendmsg` 发送控制信息，函数 `unp_internalize` 将嵌入的描述符转换成 `file` 指针，我们将在 18.4 节中描述这个函数。

2. 暂时连接一个无连接的数据报插口

146-153 如果进程传送一个带有目的地址的插口地址结构（也就是说，`nam` 参数非空），那么插口必须是无连接的，否则返回 `EISCONN` 错误。通过 `unp_connect` 连接无连接的插口。暂时连接一个无连接的数据报插口的代码与卷 2 图 23-15 的 UDP 代码相似。

154-159 如果进程没有传递一个目的地址，那么对于一个无连接的插口就返回 `ENOTCONN` 错误。

3. 传递发送者的地址

160-164 `so2` 指向目的插口的 `socket` 结构。如果发送插口 (`unp`) 已经绑定了一个路径名，`from` 就指向包含路径名的 `sockaddr_un` 结构；否则，`from` 指向 `sun_noname`，`sun_noname` 是一个以空字节作为路径名首字符的 `sockaddr_un` 结构。

如果一个 Unix 域数据报的发送者没有绑定一个路径名到它的插口，数据报的接收者由于没有目的地址（例如，路径名）而不能用 `sendto` 发送应答。这就与 UDP 不同，当数据报第一次到达一个未绑定的数据报插口时，协议就会自动为其分配一个临时的端口号。UDP 能为应用程序自动选择端口号的一个原因是这些端口号仅由 UDP 使用。然而，文件系统中的路径名并不是仅为 Unix 域插口保留。因而为一个没有绑定的 Unix 域插口自动选择路径名可能会在后面产生冲突。

是否需要一个应答取决于应用程序。例如，`syslog` 函数没有绑定一个路径名到它的 Unix 域数据报插口，它仅发送报文到本地 `syslogd` 守护进程而不想得到一个应答。

4. 把控制、地址和数据 mbuf 添加到插口接收队列

165-170 `sbappendaddr` 将控制信息（如果需要）、发送者地址和数据添加到接收插口的接收队列。如果函数调用成功，`sorwakeup` 就要唤醒所有等待这些数据的接收者，为了防止 `mbuf` 指针 `m` 和 `control` 在函数结束时被释放，将它们全置为 0（图 17-10）。如果出现错误（可能因为在接收队列上没有足够空间来存放数据、地址和控制信息），就返回 `ENOBUFS`。

处理这种错误与 UDP 不同。如果在接收队列上没有足够的空间，使用 Unix 域数据报插口的 `sender` 就会收到从它的输出操作返回的错误。同 UDP 一样，如果在接口输出队列上有足够的空间，那么发送者的输出操作就会成功。如果接收 UDP 发现在接收插口的接收队列上没有空间，它通常发送一个 ICMP 端口不可达的错误给发送者，但是如果发送者没有连接到接收者，它也就不可能收到这个错误（如同卷 2 第 600~601 页描述的一样）。为什么当接收者的缓存满时 Unix 域发送者不阻塞，而是收到 `ENOBUFS` 错误？传统上，数据报不保证可靠的数据传输。[Rago1993] 认为，在 SVR4 下编译内核时，是否给 Unix 域数据报插口提供流量控制是由厂家来决定的。

5. 暂时断开与相连插口的连接

171-172 `unp_disconnect` 断开暂时连接的插口。

图18-2给出了对于流插口的PRU_SEND请求的处理。

```

175      case SOCK_STREAM:
176 #define rcv (&so2->so_rcv)
177 #define snd (&so->so_snd)
178         if (so->so_state & SS_CANTSENDMORE) {
179             error = EPIPE;
180             break;
181         }
182         if (unp->unp_conn == 0)
183             panic("uipc 3");
184         so2 = unp->unp_conn->unp_socket;
185         /*
186          * Send to paired receive port, and then reduce
187          * send buffer hiwater marks to maintain backpressure.
188          * Wake up readers.
189          */
190         if (control) {
191             if (sbappendcontrol(rcv, m, control))
192                 control = 0;
193         } else
194             sbappend(rcv, m);
195         snd->sb_mbmax -=
196             rcv->sb_mbcnt - unp->unp_conn->unp_mbcnt;
197         unp->unp_conn->unp_mbcnt = rcv->sb_mbcnt;
198         snd->sb_hiwat -= rcv->sb_cc - unp->unp_conn->unp_cc;
199         unp->unp_conn->unp_cc = rcv->sb_cc;
200         sorwakep(so2);
201         m = 0;
202 #undef snd
203 #undef rcv
204         break;
205     default:
206         panic("uipc 4");
207     }
208     break;

```

图18-2 流插口的PRU_SEND 请求

6. 验证插口状态

175-183 如果插口的发送方已经关闭,就返回EPIPE。因为sosend验证需要一个连接的插口是否已建立连接,所以这个插口必须已建连,否则调用panic(卷2图16-24)。

第一次测试好像是一个早期版本中遗留下来的, sosend已经做了这个测试(卷2图16-24)。

7. 把mbuf添加到接收缓存

184-194 so2指向接收插口的socket结构。如果进程使用sendmsg传送了控制信息,那么控制mbuf和任何数据mbuf都要通过sbappendcontrol添加到接收插口的接收缓存。否则, sbappend将数据mbuf添加到接收缓存。如果sbappendcontrol失败,为了防止在函数结尾调用m_freem,将control指针设置为0(图17-10),因为sbappendcontrol已经释放了mbuf。

8. 更新发送者和接收者的计数器(端到端的流量控制)

195-199 对于发送者要更新两个变量: sb_mbmax(缓存中所有mbuf允许的最大字节数)和

`sb_hiwat`(缓存中允许存放实际数据的最大字节数),在卷2的图16-24中我们注意到,对`mbuf`所做的限制防止了大量小报文消耗太多的`mbuf`。

对于Unix域流插口,这两个限制指的是接收缓存和发送缓存中的两个计数器的和。例如,一个Unix域流插口的发送缓存和接收缓存的`sb_hiwat`初始值都是4096(图17-2)。如果发送者把1024字节写到插口上,不仅接收者的`sb_cc`(插口缓存中的当前字节数)从0增长到1024(正如我们所希望的),而且发送者的`sb_hiwat`从4096减到3072(这是我们所不希望的)。对于其他协议如TCP,如果没有显式设置插口的选项,缓存的`sb_hiwat`值决不会变化。`sb_mbmax`也是一样:当接收者的`sb_mbcnt`值增加时,发送者的`sb_mbmax`值下降。

因为发送给Unix域流插口的数据从来不会放在发送插口的发送缓存中,所以要改变发送者的缓存限制和接收者的当前计数。数据被立即加到接收插口的接收缓存中,没有必要浪费时间把数据放到发送插口的发送队列上,然后立即或晚些时候把它发送到接收队列上。如果接收缓存中没有空闲空间,发送者就要被阻塞。但是,如果`sosend`阻塞发送者,发送缓存中的空间大小必须反映相应接收缓存中的空间大小。代替修改发送缓存数,当发送缓存中没有数据时,很容易修改发送缓存限制来反映相应接收缓存中的空间大小。

198-199 如果我们只是检验发送者的`sb_hiwat`和接收者的`unp_cc`的操作(`sb_mbmax`和`unp_mbcnt`的操作也基本相同),在这一点上由于数据刚被添加到接收缓存,所以`rcv->sb_cc`就等于接收缓存中的字节数。`unp->unp_conn->unp_cc`是`rcv->sb_cc`的前一个值,所以它们之间的差值就是刚刚添加到接收缓存的字节数(也就是写的字节数)。同时,将`snd->sb_hiwat`的值减去相同的字节数(刚写的字节数)。接收缓存中的当前字节数保存在`unp->unp_conn->unp_cc`中,所以下一次通过这段代码我们能计算出写了多少数据。

例如,当创建插口时,发送者的`sb_hiwat`是4096,接收者的`sb_cc`和`unp_cc`都为0。如果写了1024字节,那么发送者的`sb_hiwat`变为3072,接收者的`sb_cc`和`unp_cc`都是1024。在图18-3中我们还将看到,当接收进程读这1024个字节时,发送者的`sb_hiwat`增加到4096,而接收者的`sb_cc`和`unp_cc`都降为0。

9. 唤醒等待数据的所有进程

200-201 `sorwakeup`唤醒等待数据的任何进程,由于`mbuf`现在在接收队列上,所以为了防止在函数结尾调用`m_freem`,将`m`设置为0。

图18-3中I/O代码的最后部分是`PRU_RCVD`请求,当从一个插口读数据并且协议设置`PR_WANTRCVD`标志时,`soreceive`发出这个请求(卷2图16-51),图17-5中对Unix域流协议设置这个标志。这个请求的目的是当插口层把数据从一个插口的接收缓存中移走时让协议层获得控制。例如,由于插口接收缓存中现在有更多的自由空间,TCP使用这个请求来判断是否应该将新的窗口宽度发送到对端,Unix域流协议使用这个请求去更新发送者和接收者的缓存计数器。

10. 检查对等实体是否终止

121-122 如果写数据的对等实体已经结束,不需做任何工作。注意,接收者的数据并不丢弃;然而,由于发送进程关闭了它的插口,所以发送者的缓存计数器就不能更新。由于发送者不再往插口写任何数据,所以没有必要更新缓存计数器。

11. 更新缓存计数器

123-131 `so2`指向发送者`socket`结构。根据读到的数据来更新发送者的`sb_mbmax`和`sb_hiwat`。例如,`unp->unp_cc`减去`rcv->sb_cc`就是所读到的数据字节数。

```

113     case PRU_RCVD:
114         switch (so->so_type) {
115             case SOCK_DGRAM:
116                 panic("uipc 1");
117                 /* NOTREACHED */
118             case SOCK_STREAM:
119 #define rcv (&so->so_rcv)
120 #define snd (&so2->so_snd)
121                 if (unp->unp_conn == 0)
122                     break;
123                 so2 = unp->unp_conn->unp_socket;
124                 /*
125                  * Adjust backpressure on sender
126                  * and wake up any waiting to write.
127                  */
128                 snd->sb_mbmax += unp->unp_mbcnt - rcv->sb_mbcnt;
129                 unp->unp_mbcnt = rcv->sb_mbcnt;
130                 snd->sb_hiwat += unp->unp_cc - rcv->sb_cc;
131                 unp->unp_cc = rcv->sb_cc;
132                 sowwakeup(so2);
133 #undef snd
134 #undef rcv
135                 break;
136             default:
137                 panic("uipc 2");
138             }
139             break;

```

uipc_usrreq.c

uipc_usrreq.c

图18-3 PRU_RCVD 请求

12. 唤醒任何发送数据进程

132 当从接收队列读数据时，增加发送者的 `sb_hiwat`。由于可能有空间，所以任何等待往插口写数据的进程都被唤醒。

18.3 描述符的传递

描述符的传递对于进程间通信来说是一项重大的技术。[Stevens 1992]的第15章在4.4BSD和SVR4下有使用这种技术的例子。虽然在这两种实现中的系统调用不同，但是那些例子提供了对应用程序屏蔽实现差异的库函数。

历史上描述符传递一直被称为访问权 (access right)。描述符代表一种对底层对象执行 I/O 的权力 (如果我们没有这个权力，内核就不会为我们打开描述符)。但是这个能力仅在打开描述符的进程环境中才有意义。例如，将描述符号，假定等于 4，从一个进程传到另一个进程，但并不传递这些权力，因为在接收进程中描述符 4 也许并没有打开，并且即使已经打开了，它代表的文件也可能与发送进程中所代表的文件不相同。描述符只是一个在给定进程中才有意义的标识符。一个描述符以及与其相联系的权力从一个进程传送到另一个进程需要从内核得到额外的支持。唯一能从一个进程传到另一个进程的访问权就是描述符。

图18-4显示了涉及到将描述符从一个进程传到另一个进程的数据结构。传送过程如下：

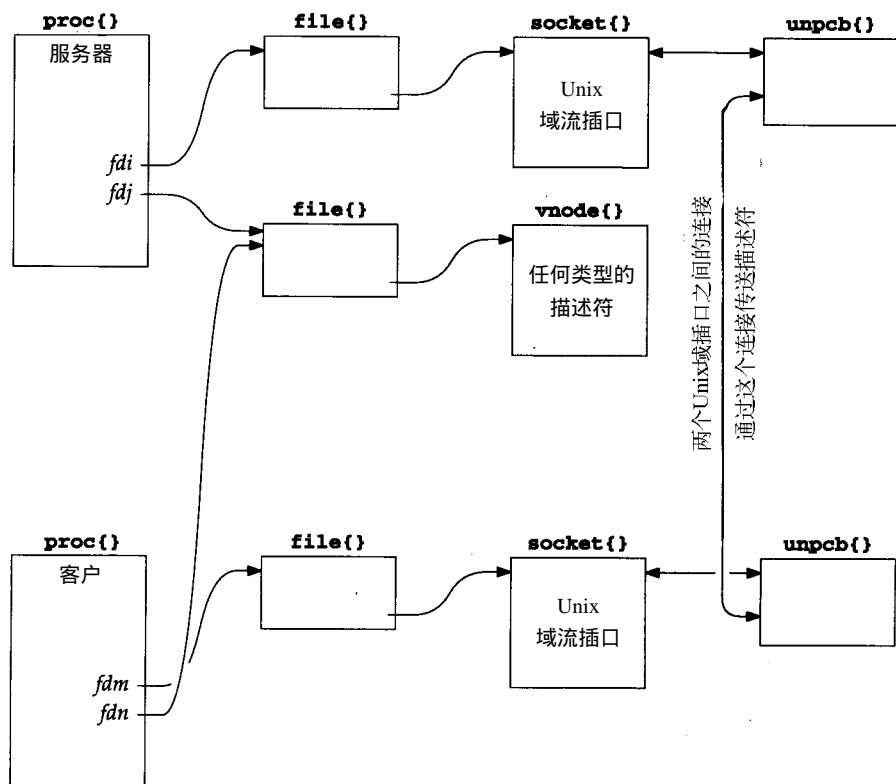


图18-4 在描述符传递中涉及到的数据结构

- 1) 我们假定最上面进程是一个从 Unix域流插口上接受连接的服务器进程。客户进程是最下面的进程，它创建一个 Unix域流插口并与服务器进程的插口建连。客户进程用 *fdm* 引用它的插口，而服务器进程用 *fdi* 来引用它的插口。在这个例子中我们用的是流插口，但是我们将看到描述符传递也能在 Unix域数据报插口间进行。我们也假定 17.10 节中 *accept* 返回的 *fdi* 作为服务器进程的连接插口，为了简单起见，我们不显示服务器进程监听插口的结构。
- 2) 服务器进程还打开另一个文件，并用 *fdj* 来访问它。通过描述符访问的文件可能是任何类型的文件：文件、设备、插口，等等，我们用 *vnode* 来表示这类文件。文件的访问计数，也就是它的 *file* 结构的 *f_count* 字段，在文件第一次打开时等于 1。
- 3) 服务器在 *fdi* 上调用 *sendmsg* 发送包含一个类型值 *SCM_RIGHTS* 和 *fdj* 值的控制信息。从而将描述符传送给接收者，即客户进程中的 *fdm*。将与 *fdj* 相联系的 *file* 结构中的引用数增加到 2。
- 4) 客户进程在 *fdm* 上调用带有控制信息缓存的 *recvmsg*，返回的控制信息有一个类型值 *SCM_RIGHTS* 和 *fdn* 值，在客户进程中 *fdn* 是最低的、尚未使用的描述符。
- 5) 在服务器进程中，当 *sendmsg* 返回后，服务器通常会关闭刚才传送的描述符 (*fdj*)。这会导致引用计数减到 1。我们说在 *sendmsg* 和 *recvmsg* 之间描述符在“传送中” (*in flight*)。三个计数器由内核负责维护，描述符传递中要用到它们。

- 1) `f_count`是`file`结构的一个字段，用来记录该结构的引用次数。当多个描述符共享相同的`file`结构时，这个字段等于描述符数。例如当一个进程打开一个文件时，该文件的`f_count`为1。如果进程接着调用`fork`，由于`file`结构在父进程和子进程间共享，所以`f_count`的值变为2，并且父进程和子进程都有一个描述符指向相同的文件结构。当一个描述符被关闭时，`f_count`值以1递减，如果值减到0，相应的文件或插口被关闭，并且`file`结构能重新使用。
- 2) `f_msgcount`也是`file`结构的一个字段，但是它仅在传送描述符时等于非0。当描述符由`sendmsg`传送时，`f_msgcount`以1递增。当`recvmsg`接收到描述符时，`f_msgcount`以1递减。`f_msgcount`值是这个`file`结构的引用数，`file`结构由插口接收队列中的描述符保持着（即目前是在传送中）。
- 3) `unp_rights`是一个全局变量，用来记录当前正被传送的描述符个数，也就是当前插口接收队列中的描述符总数。

对于一个已打开，但还没有被传送的描述符，`f_count`的值大于0，`f_msgcount`的值等于0。

图18-5显示了当一个描述符传送时三个变量的值，我们假定当前内核没有传送其他的描述符。

	<code>f_count</code>	<code>f_msgcount</code>	<code>unp_rights</code>
发送方执行 <code>open</code> 后	1	0	0
发送方执行 <code>sendmsg</code> 后	2	1	1
在接收方的队列上	2	1	1
接收方执行 <code>recvmsg</code> 后	2	0	0
发送方执行 <code>close</code> 后	1	0	0

图18-5 描述符传送过程中内核变量的值

在这个图中我们假定，接收者的`recvmsg`返回后发送者关闭描述符。但是在接收者调用`recvmsg`之前，允许发送者在描述符传递过程中关闭它，图18-6表示了这种情况发生时三个变量的值。

	<code>f_count</code>	<code>f_msgcount</code>	<code>unp_rights</code>
发送方执行 <code>open</code> 后	1	0	0
发送方执行 <code>sendmsg</code> 后	2	1	1
在接收方的队列上	2	1	1
发送方执行 <code>close</code> 后	1	1	1
在接收方的队列上	1	1	1
接收方执行 <code>recvmsg</code> 后	1	0	0

图18-6 描述符传送过程中内核变量的值

无论发送者在接收者调用`recvmsg`之前或之后关闭描述符，最终结果都是一样的。我们从上面两个图中也能看到，`sendmsg`增加所有的三个计数器，而`recvmsg`只减少表中的最后两个计数器。

用来传送描述符的内核代码从概念上看是比较简单的。将传送的描述符转换成相应的`file`指针并传送到Unix域插口的另一端。在接收进程中，接收者把`file`指针转换为最低的、没有使用的描述符。然而在处理可能的错误时就有问题了，例如，当一个描述符在它的接收

队列上时，接收进程就能关闭它的 Unix域插口。

将一个描述符转换成相应的 file 指针叫做内部化 (internalizing)，在接收进程中，随后的 file 指针转换成最低的。没有使用的描述符叫做外部化 (externalizing)。如果进程传送控制信息，图 18-1 中的 PRU_SEND 请求将调用 unp_internalize 函数。如果进程正在读 MT_CONTROL 类型的一个 mbuf，soreceive 就调用 unp_externalize 函数 (卷 2 图 16-44)。

图 18-7 显示了被进程传送到 sendmsg 的控制信息的定义，这里控制信息用于传送描述符。当接收到一个描述符时，recvmsg 填充相同类型的一个结构。

```

251 struct cmsghdr {
252     u_int    cmsg_len;           /* data byte count, including hdr */
253     int      cmsg_level;        /* originating protocol */
254     int      cmsg_type;         /* protocol-specific type */
255 /* followed by u_char cmsg_data[]; */
256 };

```

socket.h

图 18-7 cmsghdr 结构

例如，如果进程发送两个描述符，它们的值分别是 3 和 7，图 18-8 给出了控制信息的格式。我们还给出了 msghdr 结构中描述控制信息的两个字段。

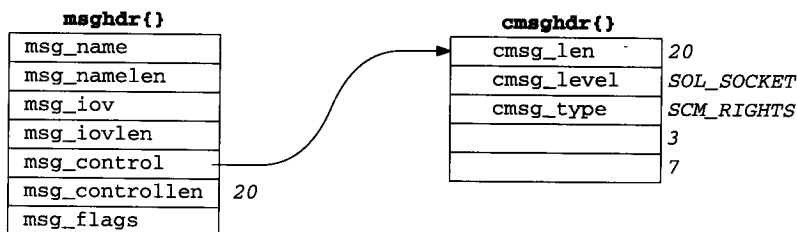


图 18-8 传送两个描述符的控制信息的例子

通常一个进程使用一个 sendmsg 能发送任意个描述符，但是传送描述符的应用程序典型情况下只传送一个描述符。有一个内部约束限制着控制信息总的大小必须适合一个 mbuf (由 sockargs 函数强加的，这个 sockargs 函数又是被 sendit 函数调用的，分别见卷 2 图 15-20 和图 16-21)，这样就限制了任何进程最多只能传送 24 个描述符。

在 4.3BSD Reno 之前，msghdr 结构的 msg_control 和 msg_controllen 字段分别为 msg_accrightrights 和 msg_accrightrightslen。

明显冗余的 cmsg_len 字段总是等于 msg_controllen，这其中的原因是允许多条控制信息出现在同一个控制缓存中，但是我们将看到源代码不支持这种情况，而是要求每个控制缓存仅有一个控制报文。

对于一个 UDP 数据报，Internet 域中支持的唯一控制信息是返回目的 IP 地址 (卷 2 图 23-25)。对于各种特定 OSI 用途的 OSI 协议支持四种不同类型的控制信息。

图 18-9 总结了在发送和接收描述符过程中调用的函数，带阴影的函数在本卷中讲述，其余的函数见卷 2。

图 18-10 总结了 unp_internalize 和 unp_externalize 对用户控制缓存和内核 mbuf 中的描述符和 file 指针的各种操作。

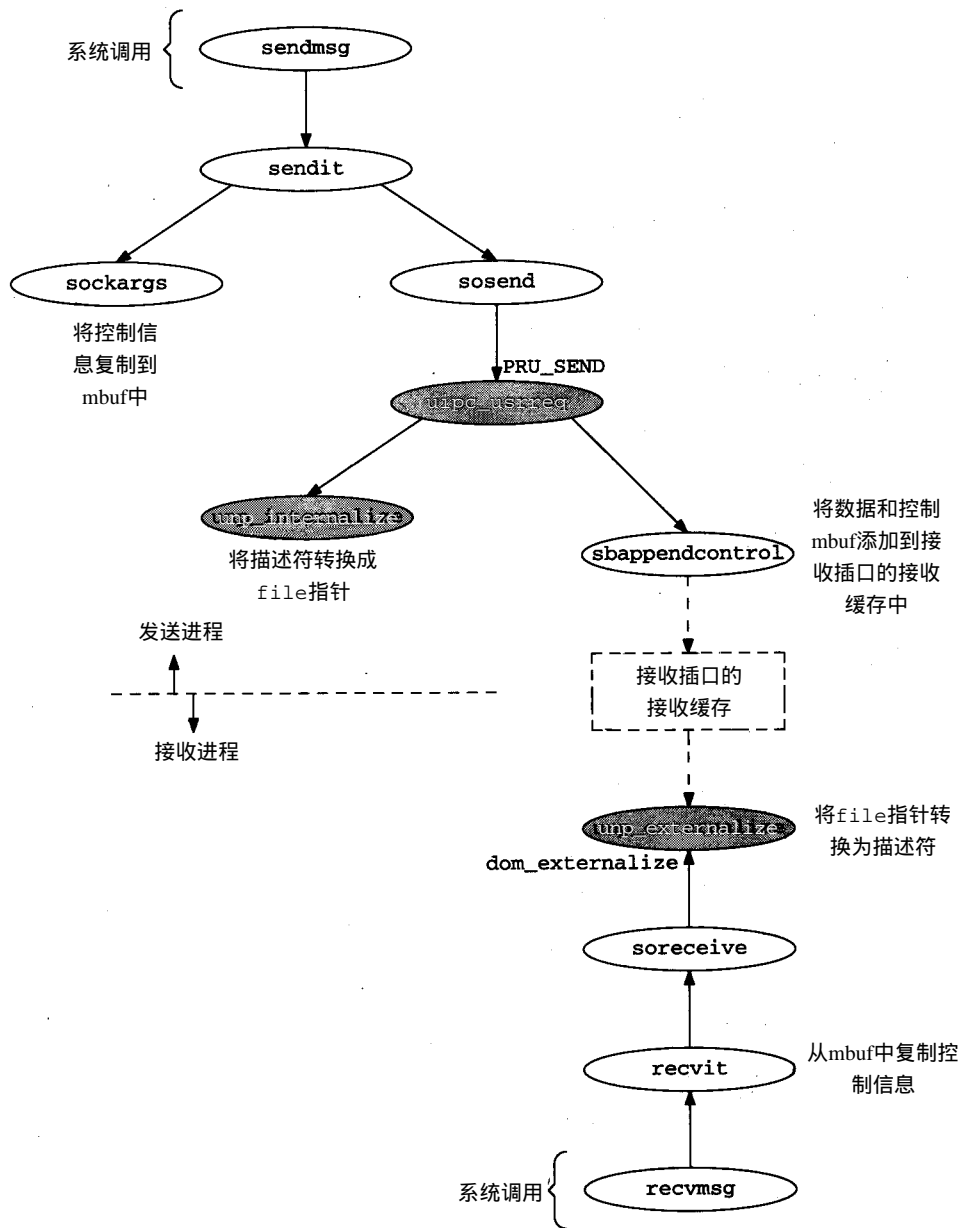


图18-9 在传送描述符过程中涉及到的函数

18.4 unp_internalize函数

图18-11描述了 `unp_internalize` 函数。正如我们在图 18-1中看到的一样，当发出 `PRU_SEND` 请求并且进程正传送描述符时，`uipc_usrreq` 调用这个函数。

1. 验证 `cmsghdr` 字段

564-566 用户的 `cmsghdr` 结构必须指定类型 `SCM_RIGHTS` 和级别 `SOL_SOCKET`，并且它的长度字段必须等于 `mbuf` 中的数据量（这是 `msg_hdr` 结构中 `msg_controllen` 字段的一个副本，`msg_hdr` 结构由进程传送到 `sendmsg`）。

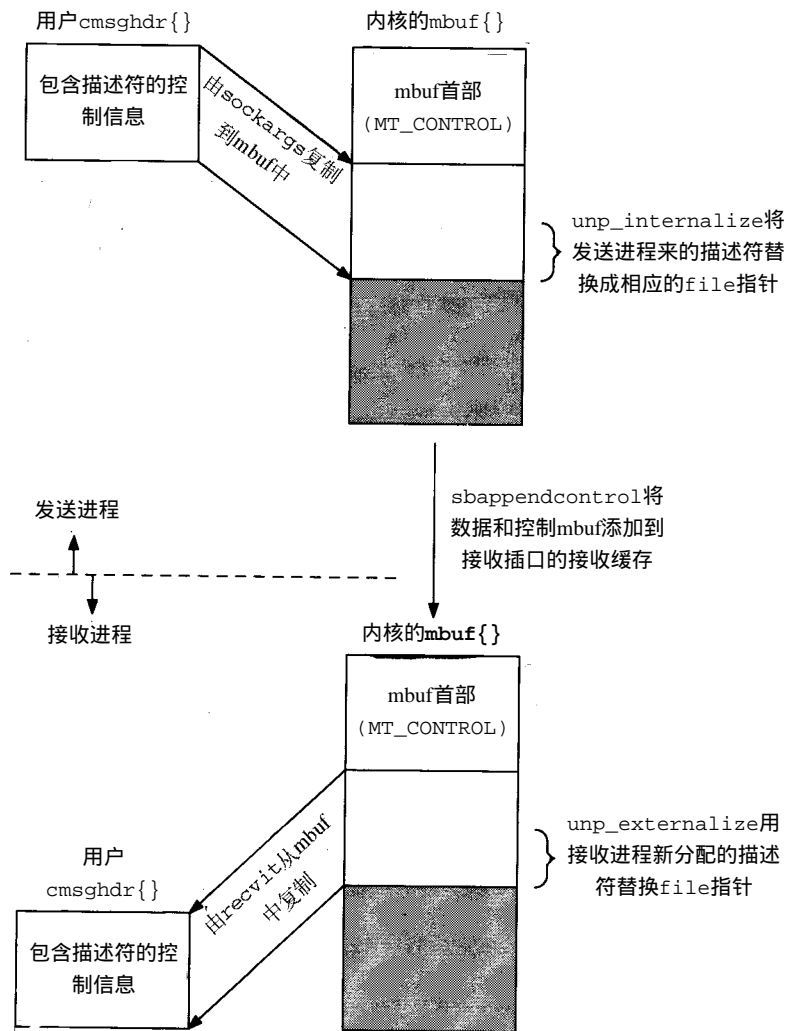


图18-10 由unp_internalize 和unp_externalize 执行的操作

2. 验证传送描述符的有效性

567-574 oldfds设置为被传送的描述符数，rp指向第一个描述符。对于每一个被传送的描述符，for循环验证这个描述符不会比当前被进程使用的最大描述符还大，以及指针非空（即描述符已打开）。

3. 用file指针替换描述符

575-578 将rp重新设置为指向第一个描述符，for循环用引用的file指针fp替换每一个描述符。

4. 增加三个计数器

579-581 file结构的f_count和f_msgcount元素递增，前者在每一次描述符关闭时递减，而后者由unp_externalize递减。另外，对于每一个由unp_internalize传送的描述符来说，全局变量unp_rights递增。我们将看到，对于每一个由unp_externalize接受描述符，unp_rights将递减。任何时候它的值都是当前内核中正在传送的描述符数。

uipc_usrreq.c

```

553 int
554 unp_internalize(control, p)
555 struct mbuf *control;
556 struct proc *p;
557 {
558     struct filedesc *fdp = p->p_fdp;
559     struct cmsghdr *cm = mtod(control, struct cmsghdr *);
560     struct file **rp;
561     struct file *fp;
562     int i, fd;
563     int oldfds;

564     if (cm->cmsg_type != SCM_RIGHTS || cm->cmsg_level != SOL_SOCKET ||
565         cm->cmsg_len != control->m_len)
566         return (EINVAL);
567     oldfds = (cm->cmsg_len - sizeof(*cm)) / sizeof(int);
568     rp = (struct file **) (cm + 1);
569     for (i = 0; i < oldfds; i++) {
570         fd = *(int *) rp++;
571         if ((unsigned) fd >= fdp->fd_nfiles ||
572             fdp->fd_ofiles[fd] == NULL)
573             return (EBADF);
574     }
575     rp = (struct file **) (cm + 1);
576     for (i = 0; i < oldfds; i++) {
577         fp = fdp->fd_ofiles[(int *) rp];
578         *rp++ = fp;
579         fp->f_count++;
580         fp->f_msgcount++;
581         unp_rights++;
582     }
583     return (0);
584 }

```

uipc_usrreq.c

图18-11 unp_internalize 函数

我们在图 17-14 中看到，当任何 Unix 域插口关闭，并且计数器非 0 时，调用无用单元收集函数 unp_gc，以免关闭的插口在它的接收队列上包含任何正在传送的描述符。

18.5 unp_externalize 函数

图 18-12 表示了 unp_externalize 函数，当一个类型为 MT_CONTROL 的 mbuf 在插口的接收队列上，并且进程正准备接收控制信息时，soreceive 像调用 dom_externalize 函数一样调用 unp_externalize (卷 2 图 16-44)。

1. 验证接收进程是否有足够的可用描述符

532-541 newfds 是外部化的 mbuf 中 file 指针的数目。fdavail 是检验进程是否有足够可用描述符的一个内核函数。如果没有足够的可用描述符，那么对于每一个描述符调用 unp_discard (在下一节描述)，并且返回 EMSGSIZE 给进程。

2. 把 file 指针转换成描述符

542-546 对于进程中每一个传送的 file 指针，最小的没有使用的描述符由 fdalloc 来分配。fdalloc 的第二个参数 0 告诉它不需要分配一个 file 结构，因为此时需要的只是一个描述符。fdalloc 通过 f 返回描述符。进程中的描述符指向 file 指针。

3. 递减两个计数器

547-548 对于每一个传送的描述符，两个计数器 `f_msgcount` 和 `unp_rights` 都要递减。

4. 用描述符替换 `file` 指针

549 新分配的描述符替换 `mbuf` 中的 `file` 指针，这是作为控制信息返回到进程中的值。

如果由进程传送到 `recvmsg` 的控制缓存不够，接收传送的描述符怎么办？

`unp_externalize` 仍然分配进程中需要的描述符数，描述符全部指向正确的 `file` 结构，但是 `recvit` (卷2的图16-44) 仅仅返回与进程分配的缓存相适应的控制信息。如果导致控制信息的不完整截断，那么就要置上 `msg_flags` 字段中的 `MSG_CTRUNC` 标志，进程通过测试这个标志来判断 `recvmsg` 返回的控制信息的完整性。

```

523 int
524 unp_externalize(rights)
525 struct mbuf *rights;
526 {
527     struct proc *p = curproc;    /* XXX */
528     int i;
529     struct cmsghdr *cm = mtod(rights, struct cmsghdr *);
530     struct file **rp = (struct file **) (cm + 1);
531     struct file *fp;
532     int newfds = (cm->cmsgh_len - sizeof(*cm)) / sizeof(int);
533     int f;

534     if (!fdavail(p, newfds)) {
535         for (i = 0; i < newfds; i++) {
536             fp = *rp;
537             unp_discard(fp);
538             *rp++ = 0;
539         }
540         return (EMSGSIZE);
541     }
542     for (i = 0; i < newfds; i++) {
543         if (fdalloc(p, 0, &f))
544             panic("unp_externalize");
545         fp = *rp;
546         p->p_fd->fd_ofiles[f] = fp;
547         fp->f_msgcount--;
548         unp_rights--;
549         *(int *) rp++ = f;
550     }
551     return (0);
552 }

```

— *uipc_usrreq.c*

— *uipc_usrreq.c*

图18-12 `unp_externalize` 函数

18.6 `unp_discard` 函数

当判断出接收进程没有足够的可用描述符时，在图 18-12 中对于每一个传送的描述符调用图18-13中的 `unp_discard`。

1. 递减两个计数器

730-731 `f_msgcount` 和 `unp_rights` 两个计数器全都递减。

```

726 void
727 unip_discard(fp)
728 struct file *fp;
729 {
730     fp->f_msgcount--;
731     unip_rights--;
732     (void) closef(fp, (struct proc *) NULL);
733 }

```

uipc_usrreq.c

图18-13 unip_discard 函数

2. 调用closef

732 closef关闭file，如果f_count现在是0，closef就减小f_count，并且调用描述符的fo_close函数(卷2的图15-38)。

18.7 unip_dispose函数

回想图17-14中，如果全局变量unip_rights非0(即有描述符在传送中)，那么当关闭一个Unix域插口时，unip_detach函数就要调用sorflush。如果有定义，并且协议设置了PR_RIGHTS标志，sorflush(卷2图15-37)执行的最后操作之一就是调用域的dom_dispose函数。因为将要刷新(释放)的mbuf也许包含正在传送中的描述符，所以需要执行这个调用。由于file结构中的两个计数器f_count和f_msgcount以及全局变量unip_rights都要由unip_internalize来递增，对于已传送但没有被接收的描述符，这些计数器全都必须要调整。

Unix域的dom_dispose函数就是unip_dispose(图17-4)，如图18-14所示。

```

682 void
683 unip_dispose(m)
684 struct mbuf *m;
685 {
686     if (m)
687         unip_scan(m, unip_discard);
688 }

```

uipc_usrreq.c

图18-14 unip_dispose 函数

调用unip_scan

686-687 unip_scan完成的所有工作我们在下一节描述。该调用的第二个参数是指向函数unip_discard的一个指针，正如我们在上一节看到的一样，unip_discard删除在插口接收队列上unip_scan发现的控制缓存中的任何描述符。

18.8 unip_scan函数

从unip_dispose调用unip_scan函数，其第二个参数为unip_discard，并且这个函数在后面的unip_gc中也会被调用，其第二个参数为unip_mark。我们在图18-15中给出了unip_scan。

```

689 void
690 unproc_scan(m0, op)
691 struct mbuf *m0;
692 void (*op) (struct file *);
693 {
694     struct mbuf *m;
695     struct file **rp;
696     struct cmsghdr *cm;
697     int i;
698     int qfds;

699     while (m0) {
700         for (m = m0; m; m = m->m_next)
701             if (m->m_type == MT_CONTROL &&
702                 m->m_len >= sizeof(*cm)) {
703                 cm = mtod(m, struct cmsghdr *);
704                 if (cm->cmsg_level != SOL_SOCKET ||
705                     cm->cmsg_type != SCM_RIGHTS)
706                     continue;
707                 qfds = (cm->cmsg_len - sizeof *cm)
708                     / sizeof(struct file *);
709                 rp = (struct file **) (cm + 1);
710                 for (i = 0; i < qfds; i++)
711                     (*op) (*rp++);
712                 break; /* XXX, but saves time */
713             }
714         m0 = m0->m_nextpkt;
715     }
716 }

```

uipc_usrreq.c

uipc_usrreq.c

图18-15 unproc_scan 函数

1. 查找控制mbuf

699-706 这个函数检查插口接收队列上(m0参数)所有的分组，并且扫描每一个分组的mbuf链去查找一个类型为MT_CONTROL的mbuf。当发现一个控制报文时，如果层次是SOL_SOCKET，类型是SCM_RIGHTS，那么mbuf包含没有被接收的传送中的描述符。

2. 释放保持的file引用

707-716 qfds是控制信息中file表指针的数量，对每一个file指针调用op函数(unproc_discard或unproc_mark)。op函数的参数是控制信息中的file指针。当处理完该控制mbuf时，执行break，跳出循环，处理接收缓存中的下一个分组。

712行的注释XXX表示：因为break假定每个mbuf链仅有一个控制mbuf，这实际上是对的。

18.9 unproc_gc函数

我们已经看到用来处理传送中的描述符的无用单元收集函数的一种形式：在unproc_detach中，无论什么时候关闭一个Unix域插口，并且描述符在传送中，sorflush就释放任何传送中的、包含在关闭插口接收队列上的描述符。然而，在Unix域插口间传送的描述符也有可能“丢失”，在三种情况下这种事情可能发生。

- 1) 当描述符被传送时，一个类型为MT_CONTROL的mbuf由sbappendcontrol(图18-2)放在插口接收队列上。但是，如果接收进程调用recvmsg却没有说明想接收控制信息，

或者调用一个不能接收控制信息的其他输入函数，`soreceive`就调用`MFREE`，从插口接收缓存中删除类型为`MT_CONTROL`的`mbuf`，并释放它(卷2图15-44)。但是，当由这个`mbuf`引用的`file`结构被发送者关闭时，它的`f_count`和`f_msgcount`将全为1(回想图18-6)，全局变量`unp_rights`仍然表明这个描述符在传送中。这是一个没有被其他任何描述符引用的`file`结构，并且将来也不会被一个描述符引用，但是仍在内核的活动`file`结构链表上。

[Leffler et al.1989]的第305页讲到，问题是在报文被传送到插口层等待传送之后，内核不允许协议再访问该报文；他们还后见之明地讲到，当一个类型为`MT_CONTROL`的`mbuf`被释放时，这个问题应当由触发的每个域的处理函数来处理。

- 2) 当描述符被传送，但是接收插口没有空间存放这个报文时，不需要任何说明就丢弃这个传送中的描述符。这种情况在一个 Unix 域流插口中应当不会发生，因为在 18.2 节中我们看到，发送者的高水位标记反映了接收者缓存中的空间大小，使得在接收缓存有了空间之前发送者的高水位标记一直阻塞发送者。但是在一个 Unix 域数据报插口中可能会失败，如果接收缓存没有足够的空间，`sbappendaddr`(在图18-1中调用)返回0，`error`设置为`ENOBUFFS`，在标号`release`处的代码会删除包含控制报文的 `mbuf`，这就如同在前一个例子中一样导致相同的情况：一个没有被任何描述符引用的 `file` 结构，并且将来也不会被一个描述符引用。
- 3) 当一个 Unix 域插口 `fdi` 在另一个 Unix 域插口 `fdj` 上传送时，`fdj` 也在 `fdi` 上传送。如果两个 Unix 域插口在没有接收到传送的描述符时关闭，这些描述符就有可能丢失。我们将看到 4.4BSD 直接处理了这个问题(图18-18)。

开始两种情况的关键事实是，“丢失的”`file`结构的`f_count`等于它的`f_msgcount`(即对这个描述符的引用是在控制报文中)，并且`file`结构当前没有被内核中所有 Unix 域插口的接收队列中任何控制报文引用。如果 `file` 结构的 `f_count` 超过了它的 `f_msgcount`，那么差别就是在引用结构的进程中描述符数，所以结构没有丢失(一个 `file` 的 `f_count` 值必须不能小于它的 `f_msgcount` 值，否则某些事情就要受到破坏)。如果 `f_count` 等于 `f_msgcount`，但是 `file` 结构被 Unix 域插口上的控制报文引用，由于一些进程仍然能从该插口接收描述符，因而不会出现问题。

无用单元收集函数 `unp_gc` 找到这些丢失的 `file` 结构，并回收它们。调用 `closef` 来回收 `file` 结构，如图18-13所示，因为 `closef` 返回一个无用的 `file` 结构给内核的空闲缓存池。注意这个函数仅在传送中描述符时才调用，这就是说，仅当 `unp_rights` 非0(图17-14)和一些 Unix 域插口关闭时才调用这个函数。因而由于这个函数似乎涉及过多的开销，它应当很少调用。

`unp_gc` 使用标记-回收(mark-and-sweep)算法去执行无用单元收集，这个函数的前一部分，即标记阶段，检查内核中的每一个 `file` 结构，并把那些正在使用的置上标志：`file` 结构要么被进程中的描述符引用，要么被 Unix 域插口的接收队列上的控制报文引用(这就是说，`file` 结构对应一个当前在传送中的描述符)。函数的后一部分，即回收阶段，回收所有尚未置上标志的 `file` 结构，因为这些 `file` 结构不在使用中。

图18-16给出了 `unp_gc` 的前半部分。

1. 防止函数被递归调用

594-596 全局变量 `unp_gcing` 防止函数被递归调用，因为 `unp_gc` 能调用 `sorflush`，而

sockflush能调用unp_dispose, unp_dispose能调用unp_discard, unp_discard能调用closef, closef能调用unp_detach, unp_detach又能再次调用unp_gc。

2. 清除FMARK和FDEFER标志

598-599 第一个循环检验内核里面的所有file结构, 并且清除FMARK和FDEFER标志。

3. 循环到unp_defer等于零

600-622 只要unp_defer标志非0, 就执行do while循环。我们将看到, 一旦发现一个以前处理过的file结构, 就置上这个标志, 我们认为这个file结构不在使用中, 但是实际上是在使用的。一旦这种情况发生, 我们需要再次回过头来检查所有的file结构, 因为有一个可能, 就是我们刚才标志为忙的结构本身就是一个Unix域插口, 并且这个Unix域插口在它的接收队列上包括file引用。

4. 循环检查所有的file结构

601-603 这个循环检查内核中的所有file结构, 如果一个结构不在使用中(f_count等于0), 我们就跳过去。

```

587 void
588 unp_gc()
589 {
590     struct file *fp, *nextfp;
591     struct socket *so;
592     struct file **extra_ref, **fpp;
593     int nunref, i;

594     if (unp_gcing)
595         return;
596     unp_gcing = 1;
597     unp_defer = 0;
598     for (fp = filehead.lh_first; fp != 0; fp = fp->f_list.le_next)
599         fp->f_flag &= ~(FMARK | FDEFER);
600     do {
601         for (fp = filehead.lh_first; fp != 0; fp = fp->f_list.le_next) {
602             if (fp->f_count == 0)
603                 continue;
604             if (fp->f_flag & FDEFER) {
605                 fp->f_flag &= ~FDEFER;
606                 unp_defer--;
607             } else {
608                 if (fp->f_flag & FMARK)
609                     continue;
610                 if (fp->f_count == fp->f_msgcount)
611                     continue;
612                 fp->f_flag |= FMARK;
613             }
614             if (fp->f_type != DTYPE_SOCKET ||
615                 (so = (struct socket *) fp->f_data) == 0)
616                 continue;
617             if (so->so_proto->pr_domain != &unixdomain ||
618                 (so->so_proto->pr_flags & PR_RIGHTS) == 0)
619                 continue;
620             unp_scan(so->so_rcv.sb_mb, unp_mark);
621         }
622     } while (unp_defer);

```

— uipc_usrreq.c

图18-16 unp_gc 函数：第一部分，标记阶段

5. 处理延迟的结构

604-606 如果已经置上FDEFER标志,那么就要关闭这个标志,并且unp_defer计数器也要减小。当unp_mark置上FDEFER标志时,FMARK标志也被置上,这样我们就知道这个记录项在使用中,并且我们还将检查在if语句的末尾是否是一个Unix域插口。

6. 跳过已经处理过的结构

607-609 如果设置了FMARK标志,那么记录项正在使用中,并且已经被处理过了。

7. 不标记丢失的结构

610-611 如果f_count等于f_msgcount,则这个记录项会可能丢失。它没有被标记,并被跳过去了。由于它似乎不在使用中,所以我们不能检查它是否是一个在接收队列上有传送中描述符的Unix域插口。

8. 标记使用中的结构

612 在这一点上我们知道这个记录项在使用中,所以要置上FMARK标志。

9. 检验结构是否与一个Unix域插口相连

614-619 既然这个记录项在使用中,我们就检验看它是否是一个有socket结构的插口。下一次检验确定这个插口是否是带有PR_RIGHTS标志集的Unix域插口。设置这个标志是为了Unix域流和数据报协议。如果任何一个测试结果是错的,就要跳过这个记录项。

10. 扫描Unix域插口接收队列上传送中的描述符

620 在这一点上file结构对应一个Unix域插口。unp_scan遍历插口接收队列,寻找包含传送中描述符的类型为MT_CONTROL的mbuf。如果发现,就调用unp_mark。

在此处,源代码也应当能处理 Unix域插口的已完成连接队列(so_q)[McKusick et al.1996],一个客户进程把描述符传送给一个新创建的还在等着接收的服务器插口是完全可能的。

图18-17给出了一个标记阶段的例子,并且在标记阶段可能需要多次扫描file结构链表。这个图描述了在标记阶段第一次扫描完成时的结构状态,此时unp_defer为1,需要再一次扫描所有的file结构。当从左至右处理四个file结构时,开始下列处理过程。

- 1) file结构在引用它的进程中有一个描述符(f_count等于2),但是没有引用传送中的描述符(f_msgcount等于0)。图18-16中的代码设置f_flag字段中的FMARK比特位。这个结构指向一个vnode(我们忽略了f_type值的DTYPE前缀,另外我们仅仅给出了f_flag字段中的FMARK和FDEFER标志,实际上其他标志值也有可能在这个字段上出现)。
- 2) 因为f_count等于f_msgcount,所以这个结构好像没有被引用。当被标记阶段处理后,f_flag字段不变。
- 3) 因为这个结构被进程中的一个描述符引用,所以要置上FMARK标志。还有,由于这个结构对应于一个Unix域插口,unp_scan还要处理插口接收队列上的任何控制报文。控制报文中的第一个描述符指向第二个file结构,并且由于在第二步中没有设置FMARK标志,unp_mark就要置上FMARK和FDEFER这两个标志。因为这个结构已经处理过,并且发现没有被引用,所以unp_defer也要增加到1。控制报文中的第二个描述符指向第四个file结构,并且由于没有置上FMARK标志(它甚至还没有被处理过),因此就要置上FMARK和FDEFER标志,unp_defer增加到2。

- 4) 设置这个结构的FDEFER标志，所以图18-16中的代码关闭这个标志，并将unp_defer减小到1。即使这个结构也被进程中的描述符引用，但是因为已经知道这个结构被传送中的描述符引用，从而也就不需要检查它的f_count和f_msgcount值。

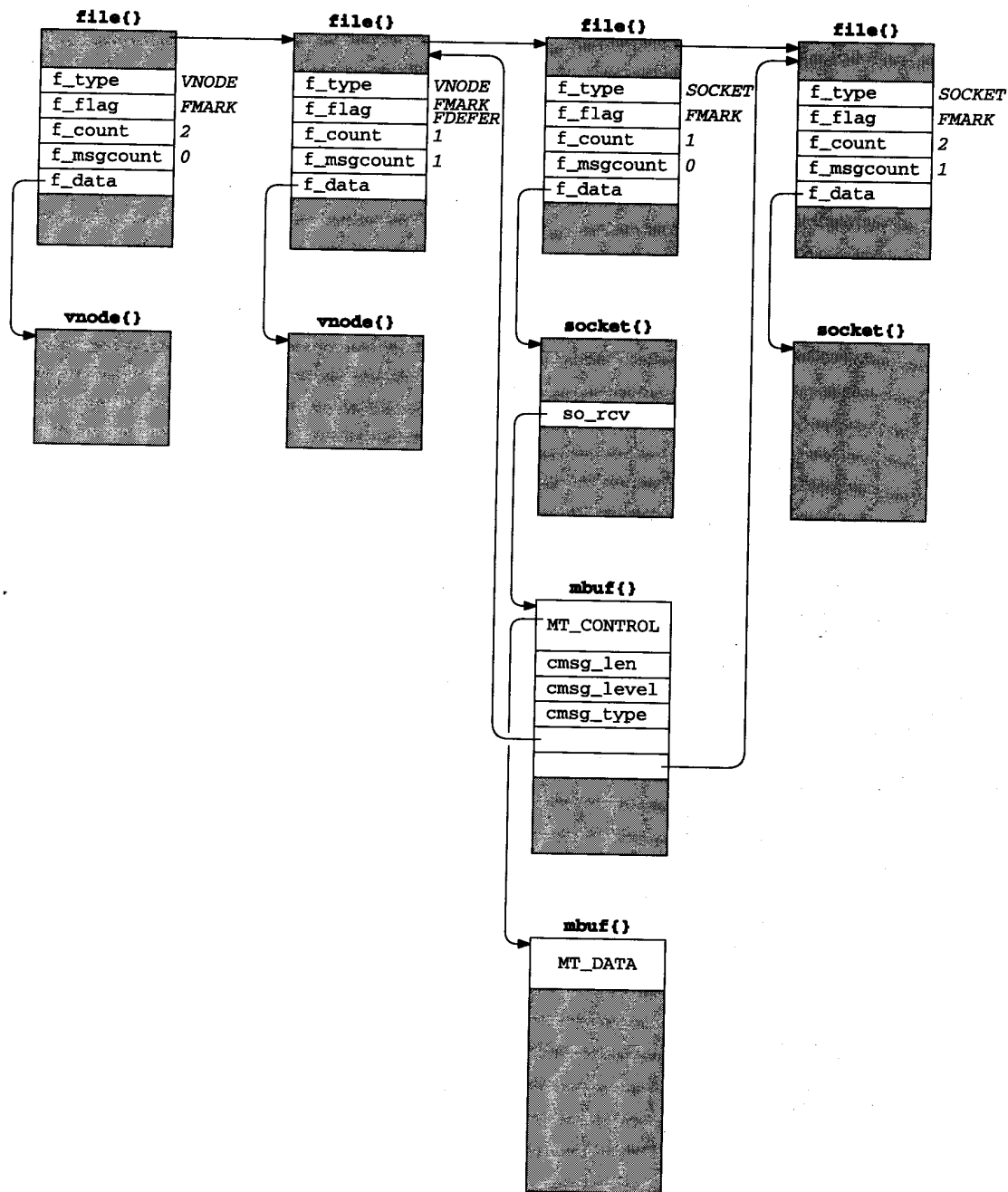


图18-17 标记阶段第一次扫描后的数据结构

此时，所有四个`file`结构都被处理过了，但是`unp_defer`等于1，所以需要再一次扫描所有的结构。因为确信第一次循环没有引用过的第二个结构也许是一个 Unix域插口，并且

在这个Unix域插口的接收队列上有控制报文，所以要产生再一次循环（这不在我们的例子中）。那个结构需要被再次处理，并且当情况是这样时，这次循环可能会在认为没有被引用的链表中靠前的一些结构中置上FMARK和FDEFER标志。

在标记阶段的结尾涉及到多次扫描内核的file结构链表，其中没有标志的结构不在使用中。函数的第二段，即回收(sweep)部分，如图18-18所示。

```

623      /*
624      * We grab an extra reference to each of the file table entries
625      * that are not otherwise accessible and then free the rights
626      * that are stored in messages on them.
627      *
628      * The bug in the original code is a little tricky, so I'll describe
629      * what's wrong with it here.
630      *
631      * It is incorrect to simply unp_discard each entry for f_msgcount
632      * times -- consider the case of sockets A and B that contain
633      * references to each other. On a last close of some other socket,
634      * we trigger a gc since the number of outstanding rights (unp_rights)
635      * is non-zero. If during the sweep phase the gc code unp_discards,
636      * we end up doing a (full) closef on the descriptor. A closef on A
637      * results in the following chain. Closef calls soclose, which
638      * calls soclose. Soclose calls first (through the switch
639      * uipc_usrreq) unp_detach, which re-invokes unp_gc. Unp_gc simply
640      * returns because the previous instance had set unp_gcing, and
641      * we return all the way back to soclose, which marks the socket
642      * with SS_NOFDREF, and then calls sofree. Sofree calls sorflush
643      * to free up the rights that are queued in messages on the socket A,
644      * i.e., the reference on B. The sorflush calls via the dom_dispose
645      * switch unp_dispose, which unp_scans with unp_discard. This second
646      * instance of unp_discard just calls closef on B.
647      *
648      * Well, a similar chain occurs on B, resulting in a sorflush on B,
649      * which results in another closef on A. Unfortunately, A is already
650      * being closed, and the descriptor has already been marked with
651      * SS_NOFDREF, and soclose panics at this point.
652      *
653      * Here, we first take an extra reference to each inaccessible
654      * descriptor. Then, we call sorflush ourselves, since we know
655      * it is a Unix domain socket anyhow. After we destroy all the
656      * rights carried in messages, we do a last closef to get rid
657      * of our extra reference. This is the last close, and the
658      * unp_detach etc will shut down the socket.
659      *
660      * 91/09/19, bsy@cs.cmu.edu
661      */
662      extra_ref = malloc(nfiles * sizeof(struct file *), M_FILE, M_WAITOK);
663      for (nunref = 0, fp = filehead.lh_first, fpp = extra_ref; fp != 0;
664           fp = nextfp) {
665          nextfp = fp->f_list.le_next;
666          if (fp->f_count == 0)
667              continue;
668          if (fp->f_count == fp->f_msgcount && !(fp->f_flag & FMARK)) {
669              *fpp++ = fp;
670              nunref++;
671              fp->f_count++;

```

图18-18 unp_gc 函数：第二部分，回收阶段


```

672     }
673 }
674 for (i = nunref, fpp = extra_ref; --i >= 0; ++fpp)
675     if ((*fpp)->f_type == DTYPE_SOCKET)
676         sorflush((struct socket *) (*fpp)->f_data);
677 for (i = nunref, fpp = extra_ref; --i >= 0; ++fpp)
678     closef(*fpp, (struct proc *) NULL);
679 free((caddr_t) extra_ref, M_FILE);
680 unp_gcing = 0;
681 }

```

uipc_usrreq.c

图18-18 (续)

11. 更正错误的注释

623-661 注释涉及到4.3BSD Reno和Net/2版本里的一个错误，这个错误在4.4BSD里由Bennet S. Yee更正，注释里提到的旧代码如图18-19所示。

12. 分配临时区域

662 malloc为指向内核中所有file结构的指针数组分配空间。nfiles是当前使用中的file结构数量。M_FILE标识使用内存的目的(vmstat -m命令输出关于内核存储器使用的信息)。如果当前得不到可用内存，那么M_WAITOK导致进程转入睡眠状态。

13. 遍历所有的file结构

663-665 为了发现所有没有引用的(丢失的)结构，这个循环再次检查内核中的所有file结构。

14. 跳过没有使用过的结构

666-667 如果file结构的f_count是0，就跳过这个结构。

15. 检查未引用的结构

668 如果在标记阶段，f_count等于f_msgcount(唯一的引用来自于传送中的描述符)，并且没有设置FMARK标志(传送中的描述符没有出现在任何Unix域插口接收队列上)，那么这个记录项是没有被引用的。

16. 保存指向file结构的指针

669-671 fp的一个副本，即指向file结构的指针，保存在分配的数组中，递增计数器nunref，递减file结构的f_count。

17. 对没有引用的插口调用sorflush

674-676 对每一个没有被引用的插口文件调用sorflush函数。函数sorflush(卷2的图15-37)调用域的dom_dispose和unp_dispose函数，unp_dispose调用unp_scan删除当前插口接收队列上任何传送中的描述符。unp_discard递减f_msgcount和unp_rights，并且对在插口接收队列上控制报文中的所有file结构调用closef。由于我们对这个file结构(早些时候完成f_count的递增)有一个额外的引用，而且由于那个循环忽略了f_count为0的结构，从而我们确信f_count等于2或者比2还要大。所以作为sorflush的结果去调用closef将把file结构的f_count减小到一个非0值，从而避免完全关闭该结构。这就是为什么对结构的额外引用进行得比较早。

18. 执行最后的关闭

677-678 对所有没有引用的file结构调用closef。这是最后一次关闭，也就是说，

f_count应当从1减到0,从而导致插口关闭,并返回file结构给内核的空闲缓存池。

19. 返回临时数组

679-680 返回早些时候由malloc分配的数组,并清除unp_gcing标志。

图18-19表示了unp_gc函数的回收阶段,同Net/2版本一样,这部分代码被图18-18中的代码替换。

```

for (fp = filehead; fp; fp = fp->f_filef) {
    if (fp->f_count == 0)
        continue;
    if (fp->f_count == fp->f_msgcount && (fp->f_flag & FMARK) == 0)
        while (fp->f_msgcount)
            unp_discard(fp);
}
unp_gcing = 0;
}

```

图18-19 Net/2版本中unp_gc 函数回收阶段的错误代码

这就是在图18-18开始部分的注释中谈到的代码。

不幸的是,虽然在本节讨论的 Net/3代码对图18-19中的代码进行了改进,并且在图18-18的开始部分描述了错误的更正,但是代码仍然是不正确的,在本节开始部分提到的前两种情况下,file结构仍是有可能丢失的。

18.10 unp_mark函数

当unp_gc调用unp_scan时,unp_mark函数被unp_scan调用去标记一个file结构。当在插口接收队列上发现传送中的描述符时完成标记过程,图18-20给出了这个函数。

```

717 void
718 unp_mark(fp)
719 struct file *fp;
720 {
721     if (fp->f_flag & FMARK)
722         return;
723     unp_defer++;
724     fp->f_flag |= (FMARK | FDEFER);
725 }

```

uipc_usrreq.c

uipc_usrreq.c

图18-20 unp_mark 函数

717-720 参数fp是指向file结构的指针,这个file结构是在Unix域插口接收队列上的控制报文里发现的。

1. 如果记录项已经被标记,就返回

721-722 如果file结构已经被标记,则不需做任何工作,因为已经知道file结构在使用过程中。

2. 设置FMARK和FDEFER标志

723-724 递减unp_defer计数器,并且设置FMARK和FDEFER标志。如果在内核列表里这

一个file结构比Unix域插口file结构出现得早(也就是说,这个file结构已经由unp_gc处理过了,并且似乎不在使用过程中,所以没有被标记),那么在 unp_gc函数的标记阶段 unp_defer的增加会导致另一次对所有file结构的遍历。

18.11 性能(再讨论)

我们已经讨论了Unix域协议的实现,现在返回到它们的性能上来看看,为什么要比 TCP快两倍(图16-2)。

所有的插口I/O都调用sosend和soreceive,与协议无关。这有利有弊,有利是因为这两个函数满足许多不同协议的需要,从字节流(TCP)到数据报协议(UDP),以及基于记录的协议(OSI TP4)。不利的原因是其一般性降低了性能,并使代码复杂化。对于不同的协议形式,这两个函数的优化版本会提高性能。

比较输出性能,对于TCP,通过sosend的路径几乎与Unix域流协议的路径相同。假定大的应用程序进行写操作(图16-2中用32 768字节写),sosend函数把用户数据打包放到mbuf簇中,然后通过PRU_SEND请求将每一个2 048字节簇传送给协议。所以,无论是TCP还是Unix域都要处理相同数量的PRU_SEND请求。对于速度上的差异,Unix域PRU_SEND(图18-2)的输出应当比TCP输出(它调用IP输出把每一段添加到环回驱动器输出队列)简单。

由于PRU_SEND请求把数据放到接收插口的接收缓存,所以在接收方唯一与 Unix域插口有关的函数是soreceive。尽管如此,对于TCP,环回驱动器把每一段数据放到IP输入队列上,后面紧跟着IP处理,再后面跟着TCP输入处理把每一段分解到正确的插口,然后将数据放到插口的接收缓存。

18.12 小结

当把数据写到一个Unix域插口时,立即将数据添加到接收插口的接收缓存中,没有必要将发送插口发送缓存里的数据进行缓存。基于这个原因,为了使流插口能正确地工作,PRU_SEND和PRU_RCVD请求操纵发送缓存的高水位标记,从而使得这个标记总是反映对等端接收缓存中的空间数量。

Unix域插口提供了将描述符从一个进程传送到另一个进程的机制。对于进程间通信来说,这是一项强大的技术。当一个描述符从一个进程传送到另一个进程时,首先这个描述符要内部化(转换成对应的file指针),再将这个指针传送到接收插口。当接收进程读到控制信息时,file指针要外部化(转换成接收进程中最小的、没有编号的描述符)。然后将描述符再返回到这个进程。

容易处理的一个错误情况是,当Unix域插口的接收缓存中有传送中描述符的控制信息时,插口关闭。不幸的是还有其他两种不容易处理的错误情况:一种是接收进程没有请求接收在其接收缓存中的控制信息;另一种是接收缓存中没有足够的空间来保存控制信息。在这两种错误情况下就会丢失file结构,这就是说,它们既不在内核的空闲缓存池中,也不在使用中。从而需要无用单元收集函数回收这些丢失的结构。

无用单元收集函数执行一个标记阶段,在这个标记阶段中扫描所有内核的file结构,同时把在使用中的描述符置上标志,在后面紧跟着回收阶段,回收所有没有被标记的结构。虽然需要这个函数,但是很少使用它。