第16章 插 口 I/O

16.1 引言

本章讨论有关从网络连接上读写数据的系统调用，分三部分介绍。

第一部分介绍四个用来发送数据的系统调用：write、writev、sendto和sendmsg。第二部分介绍四个用来接收数据的系统调用：read、readv、recvfrom和recvmsg。第三部分介绍select系统调用，select调用的作用是监控通用描述符和特殊描述符(插口)的状态。

插口层的核心是两个函数：sosend和soreceive。这两个函数负责处理所有插口层和协议层之间的I/O操作。在后续的章节中我们将看到，因为这两个函数要处理插口层和各种类型的协议之间的I/O操作，使得这两个函数特别长和复杂。

16.2 代码介绍

图16-1中列出了本章后续章节要用到的三个头文件和四个 C源文件。

文 件 名	说 明
sys/socket.h	插口API中的结构和宏定义
sys/socketvar.h	socket结构和宏定义
sys/uio.h	uio结构定义
kern/uipc_syscalls.c	socket系统调用
kern/uipc_socket.c	插口层处理
kern/sys_generic.c	select系统调用
kern/sys_socket.c	select对插口的处理

图16-1 本章涉及的头文件和C源文件

全局变量

图16-2列出了三个全局变量。前两个变量由 select系统调用使用，第三个变量控制分配给插口的存储器大小。

变 量	数据类型	说 明
selwait	int	select调用的等待通道
nselcoll	int	避免select调用中出现竞争的标志
sb_max	u_long	插口发送或接收缓存的最大字节数

图16-2 本章涉及的全局变量

16.3 插口缓存

从第15.3节我们已经知道，每一个插口都有一个发送缓存和一个接收缓存。缓存的类型为

sockbuf。图16-3中列出了sockbuf结构的定义(重复图15-5)。

socketvar.h

```
72      struct sockbuf {
73          u_long   sb_cc;          /* actual chars in buffer */
74          u_long   sb_hiwat;       /* max actual char count */
75          u_long   sb_mbcnt;       /* chars of mbufs used */
76          u_long   sb_mbmax;       /* max chars of mbufs to use */
77          long     sb_lowat;       /* low water mark */
78          struct mbuf *sb_mb;      /* the mbuf chain */
79          struct selinfo sb_sel;   /* process selecting read/write */
80          short    sb_flags;       /* Figure 16.5 */
81          short    sb_timeo;       /* timeout for read/write */
82      } so_rcv, so_snd;
```

socketvar.h

图16-3 sockbuf 结构

72-78 每一个缓存均包含控制信息和指向存储数据的mbuf链的指针。sb_mb指向mbuf链的第一个mbuf，sb_cc的值等于存储在mbuf链中的数据字节数。sb_hiwat和sb_lowat用来调整插口的流控算法。sb_mbcnt等于分配给缓存中的所有mbuf的存储器数量。

在前面的章节中提到过每一个 mbuf可存储0~2048个字节的数据 (如果使用了外部簇)。sb_mbmax是分配给插口mbuf缓存的存储器数量的上限。默认的上限在socket系统调用中发送PRU_ATTACH请求时由协议设置。只要内核要求的每个插口缓存的大小不超过 262,144个字节的限制(sb_max)，进程就可以修改缓存的上限和下限。流控算法将在 16.4节和16.8节中讨论。图16-4显示了Internet协议的默认设置。

协 议	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	9×1024	2048 (忽略)	2×sb_hiwat	40×(1024+16)	1	2×sb_hiwat
TCP	8×1024	2048	2×sb_hiwat	8×1024	1	2×sb_hiwat
原始IP ICMP IGMP	8×1024	2048 (忽略)	2×sb_hiwat	8×1024	1	2×sb_hiwat

图16-4 Internet协议的默认的插口缓存限制

因为每一个进入的UDP报文的源地址同数据一起排队，所以 UDP协议的sb_hiwat的默认值设置为能容纳40个1K字节长的数据报和相应的sockaddr_in结构(每个16字节)。

79 sb_sel是一个用来实现select系统调用的selinfo结构(16.13节)。

80 图16-5列出了sb_flags的所有可能的值。

sb-flags	说 明
SB_LOCK	一个进程已经锁定了插口缓存
SB_WANT	一个进程正在等待给插口缓存加锁
SB_WAIT	一个进程正在等待接收数据或发送数据所需的缓存
SB_SEL	一个或多个进程正在选择这个缓存
SB_ASYNC	为这个缓存产生异步I/O信号
SB_NOINTR	信号不取消加锁请求
SB_NOTIFY	(SB_WAIT|SB_AEL|SB_ASYNC) 一个进程正在等待缓存的变化，如果缓存发生任何改变，用 wakeup通知该进程

图16-5 sb_flags 的值

81-82 sb_timeo用来限制一个进程在读写调用中被阻塞的时间，单位为时钟滴答（tick）。默认值为0，表示进程无限期的等待。SO_SNDTIMEO和SO_RCVTIMEO插口选项可以改变或读取sb_timeo的值。

插口宏和函数

有许多宏和函数用来管理插口的发送和接收缓存。图 16-6中列出了与缓存加锁和同步有关的宏和函数。

名 称	说 明
sblock	申请给*sb*加锁，如果*wf*等于M_WAITOK，则进程睡眠等待加锁；否则，如果不能立即给缓存加锁，就返回 EWOULDBLOCK。如果进程睡眠被一个信号中断，则返回 EINTR或ERESTART；否则返回0 `int sblock(struct sockbuf *sb, int wf);`
sbunlock	释放加在*sb*上的锁。所有等待给*sb*加锁的进程被唤醒 `void sbunlock(struct sockbuf *sb);`
sbwait	调用tsleep等待*sb*上的协议动作。返回tsleep返回的结果 `int sbwait(struct sockbuf *sb);`
sowakeup	通知插口有协议动作出现。唤醒所有匹配的调用 sbwait的进程或在*sb*上调用tsleep的进程 `void sowakeup(struct socket *so, struct sockbuf *sb);`
sorwakeup	唤醒等待*sb*上的读事件的进程，如果进程请求了I/O事件的异步通知，则还应给该进程发送SIGIO信号 `void sorwakeup(struct socket *so);`
sowwakeup	唤醒等待*sb*上的写事件的进程，如果进程请求了I/O事件的异步通知，则还应给该进程发送SIGIO信号 `void sowwakeup(struct socket *so);`

图16-6 与缓存加锁和同步有关的宏和函数

图16-7显示了设置插口资源限制、往缓存中写数据和从缓存中删除数据的宏和函数。在该表中，*m*、*m0*、*n*和*control*都是指向mbuf链的指针。*sb*指向插口的发送或接收缓存。

名 称	说 明

sbspace	*sb*中可用的空间（字节数）：
	`min(sb_hiwat - sb_cc), (sb_mbmax - sb_mbcont)`
	`long sbspace(struct sockbuf *sb);`

sballoc	将*m*加到*sb*中，同时修改*sb*中的sb_cc和sb_mbcnt `void sballoc(struct sockbuf *sb, struct mbuf *m);`
sbfree	从*sb*中删除*m*，同时修改*sb*中的sb_cc和sb_mbcnt `int sbfree(struct sockbuf *sb, struct mbuf *m);`
sbappend	将m中的mbuf加到*sb*的最后面 `int sbappend(struct sockbuf *sb, struct mbuf *m);`
sbappendrecord	将*m0*中的记录加到*sb*的最后面。调用sbcompress `int sbappendrecord(struct sockbuf *sb, struct mbuf *m0);`

图16-7 与插口缓存分配与操作有关的宏和函数

名 称	说 明
sbappendaddr	将*asa*的地址放入一个mbuf。将地址、*control*和*m0*连接成一个mbuf链，并将该链放在*sb*的最后面 `int abappendaddr(struct sb, struct sockaddr asa, struct mbuf m0, struct mbuf control);`
sbappendcontrol	将*control*和*m0*连接成一个mbuf链，并将该链放在*sb*的最后面 `int abappendcontrol(struct sb, struct mbuf m0, struct mbuf control);`
sbinsertoob	将*m0*插在没有带外数据的*sb*的第一个记录的前面 `int abinsertoob(struct sockbuf sb, struct mbuf m0);`
sbcompress	将*m*合并到*n*中并压缩没用的空间 `void abcompress(struct sockbuf sb, struct mbuf m, struct mbuf n);`
sbdrop	删除*sb*的前*len*个字节 `void sbdrop(struct sockbuf sb, int len);`
sbdroprecord	删除*sb*的第一个记录，将下一个记录移作第一个记录 `void sbdroprecord(struct sockbuf sb);`
sbrelease	调用sbflush释放*sb*中所有的mbuf。并将sb_hiwat和sb_mbmax清0 `void sbrelease(struct sockbuf sb);`
sbflush	释放*sb*中的所有mbuf `void sbflush(struct sockbuf sb);`
soreserve	设置插口缓存高、低水位标记(high-water and low-water marks)于发送缓存，调用sbreserve并传入参数*sndcc*。对于接收缓存，调用sbreserve并传入参数rcvcc。将发送缓存和接收缓存的sb_lowat初始化成默认值(图16-4)。如果超过系统限制，则返回ENOBUFS `int soreserve(struct socket so, int sndcc, int rcvcc);`
sbreserve	将*sb*的高水位标记设置成*cc*。同时将低水位标记降到*cc*。本函数不分配存储器 `int sbreserve(struct sockbuf sb, int cc);`

图16-7 (续)

16.4 write、writev、sendto和sendmsg系统调用

我们将write、writev、sendto和sendmsg四个系统调用统称为写系统调用，它们的作用是往网络连接上发送数据。相对于最一般的调用sendmsg而言，前三个系统调用是比较简单的接口。

所有的写系统调用都要直接或间接地调用sosend。sosend的功能是将进程来的数据复制到内核，并将数据传递给与插口相关的协议。图16-8给出了sosend的工作流程。

在下面的章节中，我们将讨论图16-8中带阴影的函数。其余的四个系统调用和soo_write留给读者自己去了解。

图16-9说明了这四个系统调用和一个相关的库函数(send)的特点。

在Net/3中，send被实现成一个调用sendto的库函数。为了与以前编译的程序二进制兼容，内核将旧的send调用映射成函数osend，该函数不在本书中讨论。

从图16-9的第二栏中可以看出，write和writev系统调用适用于任何描述符，而其他的系统调用只适用于插口描述符。

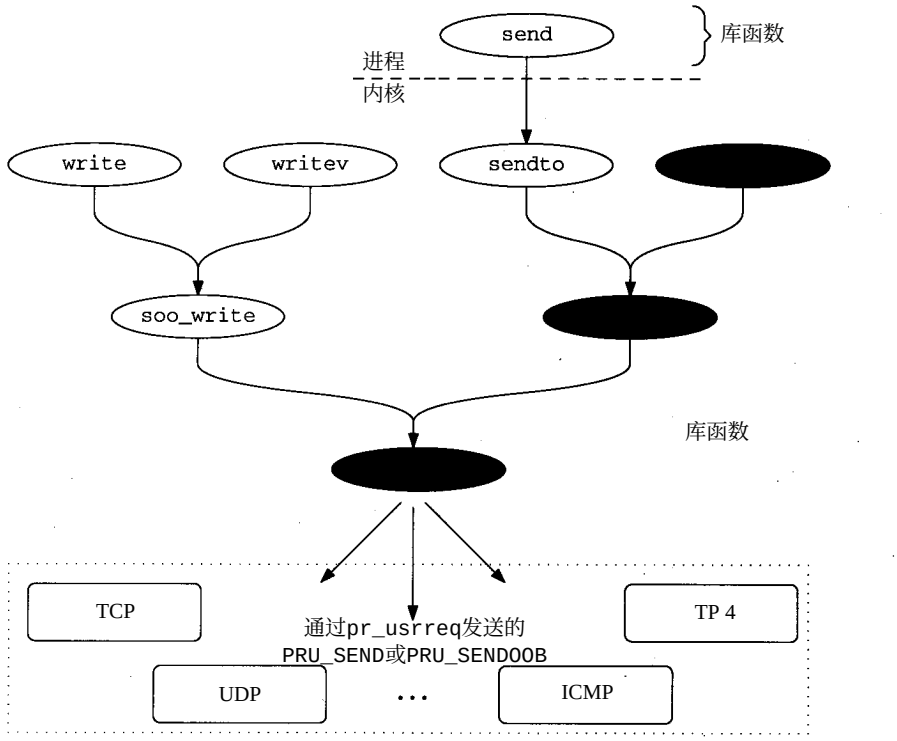

图16-8 所有的插口输出均由 sosend 处理

函 数	描述符类型	缓存数量	是否指明目的地址	标 志?	控制信息?
write	任何类型	1			
writev	任何类型	[1..UIO_MAXIOV]			
send	插口	1		•	
sendto	插口	1	•	•	
sendmsg	插口	[1..UIO-MAXIOV]	•	•	•

图16-9 写系统调用

从图16-9的第三栏中可以看出，writev和sendmsg系统调用可以接收从多个缓存中来的数据。从多个缓存中写数据称为收集 (gathering)，同它相对应的读操作称为分散(scattering)。执行收集操作时，内核按序接收类型为 iovec的数组中指定的缓存中的数据。数组最多有UIO_MAXIOV个单元。图16-10显示了类型iovec的结构。

uio.h

```
41 struct iovec {
42     char   *iov_base;          /* Base address */
43     size_t  iov_len;           /* Length */
44 };
```

uio.h

图16-10 iovec 结构

41-44 在图16-10中，iov_base指向长度为iov_len个字节的缓存的开始。

如果没有这种接口，一个进程将不得不将多个缓存复制到一个大的缓存中，或调用多个

写系统调用来发送从多个缓存来的数据。相对于用一个系统调用传送类型为 iovec的数组，这两种方法的效率更低。对于数据报协议而言，调用一次 writev就是发送一个数据报，数据报的发送不能用多个写动作来实现。

图16-11说明了iovec结构在writev系统调用中的应用，图中iovp指向数组的第一个元素，iovcnt等于数组的大小。

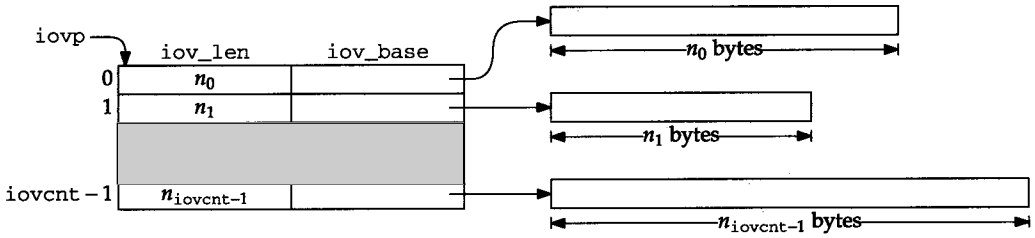

图16-11 writev 系统调用中的iovec 参数

数据报协议要求每一个写调用必须指定一个目的地址。因为 write、writev和send调用接口不支持对目的地址的指定，因此这些调用只能在调用 connect将目的地址同一个无连接的插口联系起来后才能被调用。调用 sendto或sendmsg时必须提供目的地址，或在调用它们之前调用connect来指定目的地址。

图16-9的第五栏显示send *xxx*系统调用接收一个可选的控制标志，这些标志在图 16-12中定义。

flags	描　述	参　考
MSG_DONTROUTE	发送本报文时，不查路由表	图16-23
MSG_DONTWAIT	发送本报文时，不等待资源	图16-22
MSG_EOR	标志一个逻辑记录的结束	图16-25
MSG_OOB	发送带外数据	图16-26

图16-12 send *xxx*系统调用：flags 值

如图16-9的最后一栏所示，只有 sendmsg系统调用支持控制信息。控制信息和另外几个参数是通过结构msghdr(图16-13)一次传递给sendmsg，而不是分别传递。

socket.h
```
228 struct msghdr {
229     caddr_t msg_name;           /* optional address */
230     u_int   msg_namelen;        /* size of address */
231     struct iovec *msg_iov;      /* scatter/gather array */
232     u_int   msg_iovlen;         /* # elements in msg_iov */
233     caddr_t msg_control;        /* ancillary data, see below */
234     u_int   msg_controllen;     /* ancillary data buffer len */
235     int     msg_flags;          /* Figure 16.33 */
236 };
```
socket.h

图16-13 msghdr 结构

msg_name应该被说明成一个指向sockaddr结构的指针，因为它包含网络地址。

228-236 msghdr结构包含一个目的地址(msg_name和msg_namelen)、一个分散/收集数组(msg_iov和msg_iovlen)、控制信息(msg_control和msg_controllen)和接收标志

(msg_flags)。控制信息的类型为cmsghdr结构，如图16-14所示。

socket.h

```
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16-14 cmsghdr 结构

251-256 插口层并不解释控制信息，但是报文的类型被置为 cmsg_type，且报文长度为cmsg_len。多个控制报文可能出现在控制信息缓存中。

举例

图16-15说明了在调用sendmsg时msghdr的结构。

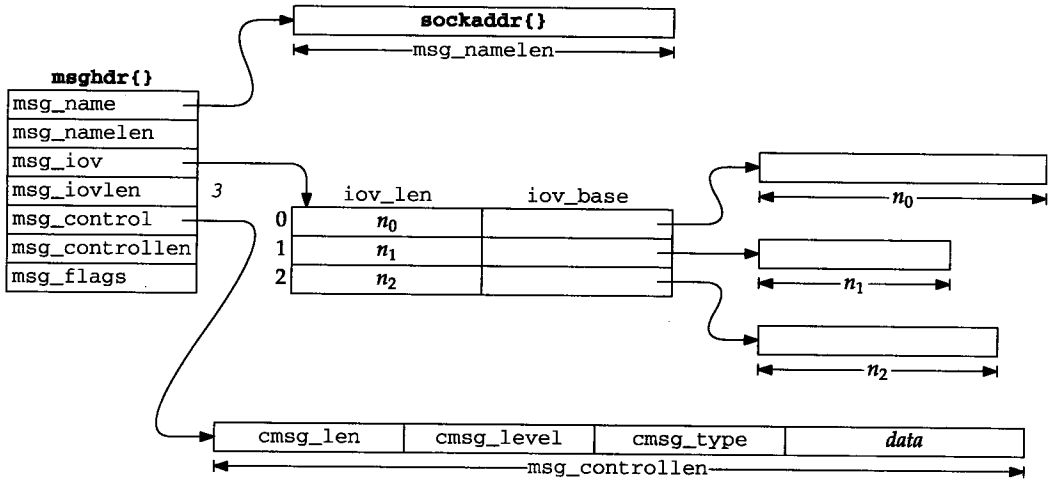

图16-15 sendmsg 系统调用的msghdr 结构

16.5 sendmsg系统调用

只有通过sendmsg系统调用才能访问到与插口 API的输出有关的所有功能。 sendmsg和sendit函数准备sosend系统调用所需的数据结构，然后由 sosend调用将报文发送给相应的协议。对SOCK_DGRAM协议而言，报文就是数据报。对 SOCK_STREAM协议而言，报文是一串字节流。对于SOCK_SEQPACKET协议而言，报文可能是一个完整的记录 (隐含的记录边界)或一个大的记录的一部分 (显式的记录边界)。对于SOCK_PDM协议而言，报文总是一个完整的记录(隐含的记录边界)。

即使一般的sosend代码处理SOCK_SEQPACKET和SOCK_PDK协议，但是在Internet域中没有这样的协议。

图16-16显示了sendmsg系统调用的源代码。

307-319 sendmsg有三个参数：插口描述符；指向 msghdr结构的指针；几个控制标志。函数copyin将msghdr结构从用户空间复制到内核。

uipc_syscalls.c

```
307 struct sendmsg_args {
308     int     s;
309     caddr_t msg;
310     int     flags;
311 };

312 sendmsg(p, uap, retval)
313 struct proc *p;
314 struct sendmsg_args *uap;
315 int    *retval;
316 {
317     struct msghdr msg;
318     struct iovec aiov[UIO_SMALLIOV], *iov;
319     int     error;

320     if (error = copyin(uap->msg, (caddr_t) & msg, sizeof(msg)))
321         return (error);
322     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
323         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
324             return (EMSGSIZE);
325         MALLOC(iov, struct iovec *,
326                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
327                M_WAITOK);
328     } else
329         iov = aiov;
330     if (msg.msg_iovlen &&
331         (error = copyin((caddr_t) msg.msg_iov, (caddr_t) iov,
332                         (unsigned) (msg.msg_iovlen * sizeof(struct iovec)))))
333                 goto done;
334     msg.msg_iov = iov;
335     error = sendit(p, uap->s, &msg, uap->flags, retval);
336   done:
337     if (iov != aiov)
338         FREE(iov, M_IOV);
339     return (error);
340 }
```

uipc_syscalls.c

图16-16 sendmsg 系统调用

1. 复制iov数组

320-334 一个有8个元素(UIO_SMALLIOV)的iovec数组从栈中自动分配。如果分配的数组不够大，sendmsg将调用MALLOC分配更大的数组。如果进程指定的数组单元大于1024(UIO_MAXIOV)，则返回EMSGSIZE。copyin将iovec数组从用户空间复制到栈中的数组或一个更大的动态分配的数组中。

这种技术避免了调用malloc带来的高代价，因为大多数情况下，数组的单元数小于等于8。

2. sendit和cleanup

335-340 如果sendit返回，则表明数据已经发送给相应的协议或出现差错。sendmsg释放iovec数组(如果它是动态分配的),并且返回sendit调用返回的结果。

16.6 sendit函数

sendit函数是被sendto和sendmsg调用的公共函数。sendit初始化一个uio结构，

将控制和地址信息从进程空间复制到内核。在讨论 sosend之前，我们必须先解释 uiomove函数和uio结构。

16.6.1 uiomove函数

uiomove函数的原型为：

```
int uiomove(caddr_t cp, int n, struct uio *uio);
```

uiomove函数的功能是在由*cp*指向的缓存与*uio*指向的类型为*iovec*的数组中的多个缓存之间传送*n*个字节。图16-7说明了uio结构的定义，该结构控制和记录uiomove的行为。

uio.h

```
45 enum uio_rw {
46     UIO_READ, UIO_WRITE
47 };

48 enum uio_seg {                    /* Segment flag values */
49     UIO_USERSPACE,                /* from user data space */
50     UIO_SYSSPACE,                 /* from system space */
51     UIO_USERISPACE                /* from user instruction space */
52 };

53 struct uio {
54     struct iovec *uio_iov;        /* an array of iovec structures */
55     int      uio_iovcnt;          /* size of iovec array */
56     off_t    uio_offset;          /* starting position of transfer */
57     int      uio_resid;           /* remaining bytes to transfer */
58     enum uio_seg uio_segflg;      /* location of buffers */
59     enum uio_rw uio_rw;           /* direction of transfer */
60     struct proc *uio_procp;       /* the associated process */
61 };
```

uio.h

图16-17 uio结构

45-61 在uio结构中，uio_iov指向类型为iovec结构的数组，uio_offset记录uiomove传送的字节数，uio_resid记录剩余的字节数。每次调用 uiomove，uio_offset增加*n*，uio_resid减去*n*。同时，uiomove根据传送的字节数调整uio_iov数组中的基指针和缓存长度，从而从缓存中删除每次调用时传送的字节。最后，每当从uio_iov中传送一块缓存，uio_iov数组的每个单元就向前进一个数组单元。uio_segflg指向uio_iov数组的基指针指向的缓存的位置。uio_rw指定数据传送的方向。缓存可能在用户数据空间，用户指令空间或内核数据空间。图 16-18对uiomove函数的操作进行了小结。图中对操作的描述用到了uiomove函数原型中的参数名。

uio_segflg	uio_rw	描 述
UIO_USERSPACE	*UIO-READ*	从内核缓存*cp*中分散*n*个字节到进程缓存
UIO_USERISPACE		
UIO_USERSPACE	*UIO-WRITE*	从进程缓存中收集*n*个字节到内核缓存*cp*
UIO_USERISPACE		
UIO_SYSSPACE	*UIO-READ*	从内核缓存*cp*中分散*n*个字节到多个内核缓存
	UIO-WRITE	从多个内核缓存中收集*n*个字节到内核缓存*cp*中

图16-18 uiomove 操作

16.6.2 举例

图16-19显示了一个调用uiomove之前的uio结构。

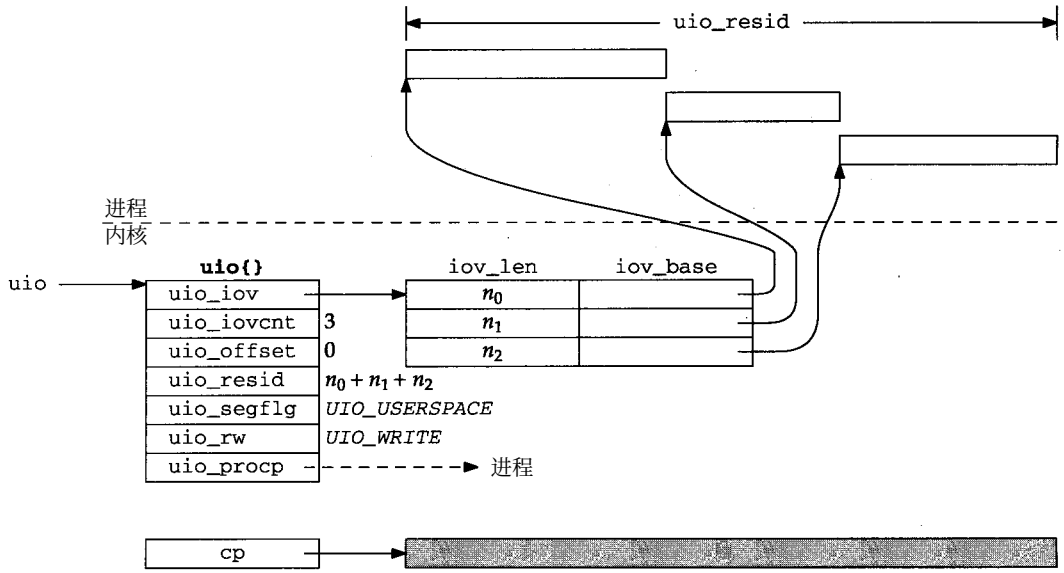

图16-19 调用uiomove 前的uio 结构

uio_iov指向iovec数组的第一个单元。iov_base指针数组的每一个单元分别指向它们在进程地址空间中的缓存的起始地址。uio_offset等于0，uio_resid等于三块缓存的总的大小。cp指向内核中的一块缓存，一般来说，这块缓存是一个 mbuf的数据区。图16-20显示了调用uiomove之后同一个uio结构的内容。

```
uiomove(cp, n, uio);
```

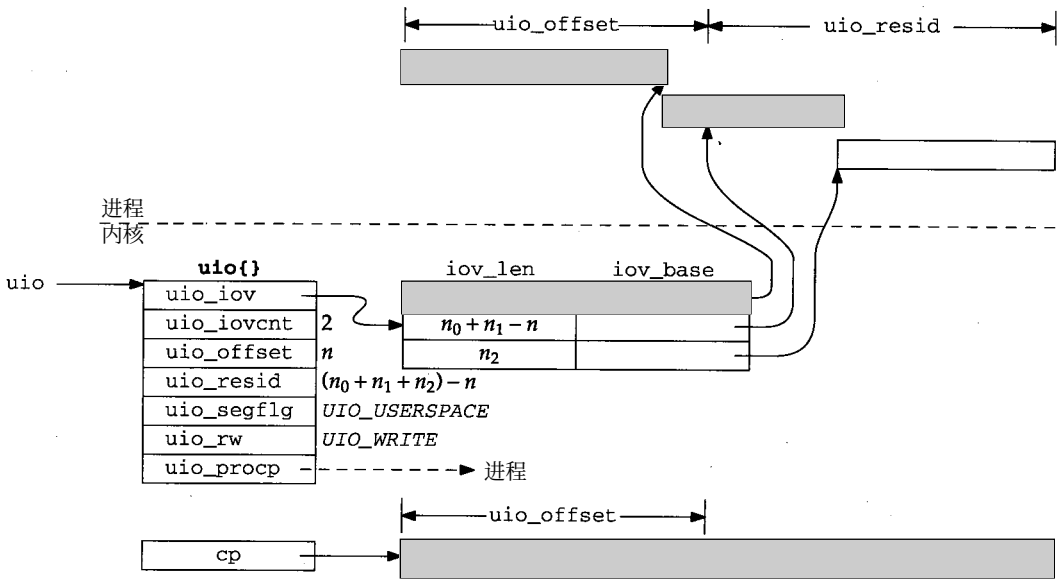

图16-20 调用uiomove 后的uio 结构

在上述调用中， n包括第一块缓存中的所有字节和第二块缓存中的部分字节 (即，$n_0<n<n_0+n_1$)。

调用uiomove后，第一块缓存的长度变为0，且它的基指针指向缓存的末端。uio_iov现在指向iovec数组的下一个单元。单元指针也前进了一个单元，长度也减少了，减少的字节数等于缓存中被传送的字节数。同时，uio_offset增加了n，uio_resid减少了n。数据已经从进程中的缓存传送到内核缓存，因为uio_rw等于UIO_WRITE.。

16.6.3 sendit代码

现在开始讨论sendit的代码，如图16-21所示。

1. 初始化auio

341-368 sendit调用getsock函数获取描述符对应的file结构，初始化uio结构，并将进程指定的输出缓存中的数据收集到内核缓存中。传送的数据的长度通过一个 for循环来计算，并将结果保存在uio_resid。循环内的第一个if保证缓存的长度非负。第二个if保证uio_resid不溢出，因为uio_resid是一个有符号的整数，且iov_len要求非负。

2. 从进程复制地址和控制信息

369-385 如果进程提供了地址和控制信息,则sockargs将地址和控制信息复制到内核缓存中。

uipc_syscalls.c

```
341 sendit(p, s, mp, flags, retsize)
342 struct proc *p;
343 int     s;
344 struct msghdr *mp;
345 int     flags, *retsize;
346 {
347     struct file *fp;
348     struct uio auio;
349     struct iovec *iov;
350     int     i;
351     struct mbuf *to, *control;
352     int     len, error;
353     if (error = getsock(p->p_fd, s, &fp))
354         return (error);
355     auio.uio_iov = mp->msg_iov;
356     auio.uio_iovcnt = mp->msg_iovlen;
357     auio.uio_segflg = UIO_USERSPACE;
358     auio.uio_rw = UIO_WRITE;
359     auio.uio_procp = p;
360     auio.uio_offset = 0;        /* XXX */
361     auio.uio_resid = 0;
362     iov = mp->msg_iov;
363     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
364         if (iov->iov_len < 0)
365             return (EINVAL);
366         if ((auio.uio_resid += iov->iov_len) < 0)
367             return (EINVAL);
368     }
369     if (mp->msg_name) {
370         if (error = sockargs(&to, mp->msg_name, mp->msg_namelen,
371                              MT_SONAME))
```

图16-21 sendit 函数

```
372             return (error);
373     } else
374         to = 0;
375     if (mp->msg_control) {
376         if (mp->msg_controllen < sizeof(struct cmsghdr)
377         ) {
378             error = EINVAL;
379             goto bad;
380         }
381         if (error = sockargs(&control, mp->msg_control,
382                              mp->msg_controllen, MT_CONTROL))
383             goto bad;
384     } else
385         control = 0;
386     len = auio.uio_resid;
387     if (error = sosend((struct socket *) fp->f_data, to, &auio,
388                        (struct mbuf *) 0, control, flags)) {
389         if (auio.uio_resid != len && (error == ERESTART ||
390                                       error == EINTR || error == EWOULDBLOCK))
391             error = 0;
392         if (error == EPIPE)
393             psignal(p, SIGPIPE);
394     }
395     if (error == 0)
396         *retsize = len - auio.uio_resid;
397   bad:
398     if (to)
399         m_freem(to);
400     return (error);
401 }
```

uipc_syscalls.c

图16-21 (续)

3. 发送数据和清除缓存

386-401 为了防止sosend不接受所有数据而无法计算传送的字节数，将uio_resid的值保存在len中。将插口、目的地址、uio结构、控制信息和标志全部传给函数sosend。当sosend返回后，sendit响应如下：

- 如果sosend传送了部分数据后，传送被信号或阻塞条件所中断，差错被丢弃，报告传送了部分数据。
- 如果sosend返回EPIPE，则发送信号SIGPIPE给进程。error设置成非0，所以如果进程捕捉到了该信号，并且从信号处理程序中返回，或进程忽略信号，写调用返回EPIPE。
- 如果没有差错出现(或差错被丢弃)，则计算传送的字节数，并将其保存在*retsize中。如果sendit返回0，syscall(第15.4节)返回*retsize给进程而不是返回差错代码。
- 如果任何其他类型的差错出现，返回相应差错代码给进程。

在返回之前，sendit释放包含目的地址的缓存。sosend负责释放control缓存。

16.7 sosend函数

sosend是插口层中最复杂的函数之一。在图16-8中已提到过所有五个写系统调用最终都要调用sosend。sosend的功能就是：根据插口指明的协议支持的语义和缓存的限制，将数

据和控制信息传递给插口指明的协议的pr_usrreq函数。sosend从不将数据放在发送缓存中；存储和移走数据应由协议来完成。

sosend对发送缓存的sb_hiwat和sb_lowat值的解释，取决于对应的协议是否实现可靠或不可靠的数据传送功能。

16.7.1 可靠的协议缓存

对于提供可靠的数据传送协议，发送缓存保存了还没有发送的数据和已经发送但还没有被确认的数据。sb_cc等于发送缓存的数据的字节数，且0≤sb_cc≤sb_hiwat。

如果有带外数据发送，则sb_cc有可能暂时超过sb_hiwat。

sosend应该确保在通过pr_usrreq函数将数据传递给协议层之前有足够的发送缓存。协议层将数据放到发送缓存中。sosend通过下面两种方式之一将数据传送给协议层：

- 如果设置了PR_ATOMIC，sosend就必须保护进程和协议层之间的边界。在这种情况下，sosend等待得到足够的缓存来存储整个报文。当获取到足够的缓存后，构造存储整个报文的mbuf链，并用pr_usrreq函数一次性传送给协议层。RDP和SPP就是这种类型的协议。
- 如果没有设置PR_ATOMIC，sosend每次传送一个存有报文的mbuf给协议，可能传送部分mbuf给协议层以防止超过上限。这种方法在SOCK_STREAM类协议如TCP中和SOCK_SEQPACKET类协议如TP4中被采用。在TP4中，记录边界通过MSG_EOR标志(图16-12)来显式指定，所以sosend没有必要保护报文边界。

TCP应用程序对外出的TCP报文段的大小没有控制。例如，在TCP插口上发送一个长度为4096字节的报文，假定发送缓存中有足够的缓存，则插口层将该报文分成两部分，每一部分长度为2048个字节，分别存放在一个带外部簇的mbuf中。然后，在协议处理时，TCP将根据连接上的最大报文段大小将数据分段，通常情况下，最大报文段大小为2048个字节。

当一个报文因为太大而没有足够的缓存时，协议允许报文被分成多段，但sosend仍然不将数据传送给协议层直到缓存中的闲置空间大小大于sb_lowat。对于TCP而言，sb_lowat的默认值为2048 (图16-4)，从而阻止插口层在发送缓存快满时用小块数据干扰TCP。

16.7.2 不可靠的协议缓存

对于提供不可靠的数据传输的协议(如UDP)而言，发送缓存不需保存任何数据，也不等待任何确认。每一个报文一旦被排队等待发送到相应的网络设备，插口层立即将它传送到协议。在这种情况下，sb_cc总是等于0，sb_hiwat指定每一次写的最大长度，间接指明数据报的最大长度。

图16-4显示了UDP协议的sb_hiwat的默认值为9216(9×1024)。如果进程没有通过SO_SNDBUF插口选项改变sb_hiwat的值，则发送长度大于9216个字节的数据报将导致差错。不仅如此，其他的协议限制也可能不允许一个进程发送大的数据报报文。卷 1的第11.10节中已讨论了在其他的TCP/IP实现中的这些选项和限制。

对于NFS写而言，9216已足够大，NFS写的数据加上协议首部的长度一般默认为8192个字节。

图16-22显示了sosend函数的概况。下面分别讨论图中四个带阴影的部分。

uipc_socket.c

```
271 sosend(so, addr, uio, top, control, flags)
272 struct socket *so;
273 struct mbuf *addr;
274 struct uio *uio;
275 struct mbuf *top;
276 struct mbuf *control;
277 int     flags;
278 {
        /* initialization (Figure 16.23) */
305   restart:
306     if (error = sblock(&so->so_snd, SBLOCKWAIT(flags)))
307         goto out;
308     do {                        /* main loop, until resid == 0 */
            /* wait for space in send buffer (Figure 16.24) */
342         do {
343             if (uio == NULL) {
344                 /*
345                  * Data is prepackaged in "top".
346                  */
347                 resid = 0;
348                 if (flags & MSG_EOR)
349                     top->m_flags |= M_EOR;
350             } else
351                 do {
                        /* fill a single mbuf or an mbuf chain (Figure 16.25) */
396                 } while (space > 0 && atomic);
                    /* pass mbuf chain to protocol (Figure 16.26) */
412         } while (resid && space > 0);
413     } while (resid);

414   release:
415     sbunlock(&so->so_snd);
416   out:
417     if (top)
418         m_freem(top);
419     if (control)
420         m_freem(control);
421     return (error);
422 }
```

uipc_socket.c

图16-22 sosend 函数：概述

271-278 sosend的参数有如下几个：so，指向相应插口的指针；addr，指向目的地址的指针；uio，指向描述用户空间的 I/O缓存的uio结构；top，保存将要发送的数据的 mbuf链；control，保存将要发送的控制信息的mbuf链；flags，包含本次写调用的一些选项。

正常情况下，进程通过uio机制将数据提供给插口层， top为空。当内核本身正在使用插口层时(如NFS)，数据将作为一个mbuf链传送给sosend， top指向该mbuf链，而uio为空。

279-304 初始化代码分别如下所述。

1. 给发送缓存加锁

305-308 sosend的主循环从restart开始，在循环的开始调用sblock给发送缓存加锁。通过加锁确保多个进程按序互斥访问插口缓存。

如果在flags中MSG_DONTWAIT被设置，则SBLOCKWAIT将返回M_NOWAIT。M_NOWAIT告知sblock，如果不能立即加锁，则返回EWOULDBLOCK。

MSG_DONTWAIT仅用于Net/3中的NFS。

主循环直到将所有数据都传送给协议 (即resid=0)后才退出。

2. 检查空间

309-341 在传送数据给协议之前，需要对各种差错情况进行检查，并且 sosend实现前面讨论的流控和资源控制算法。如果 sosend阻塞等待输出缓存中的更多的空间，则它跳回restart等待。

3. 使用top中的数据

342-350 一旦有了足够的空间并且sosend也获得了发送缓存上的锁，则准备传送给协议的数据。如果uio等于空(即数据在top指向的mbuf链中)，则sosend检查MSG_EOR，并且在链中设置M_EOR来标志逻辑记录的结束。 mbuf链是准备发送给协议层的。

4. 从进程复制数据

351-396 如果uio不空，则sosend必须从进程间复制数据。当PR_ATOMIC被设置时(例如，UDP)，循环继续，直到所有数据都被复制到一个 mbuf链中。当sosend从进程得到所有数据后，通过循环中的break(图16-22中没有显示这个break)跳出循环。跳出循环后， sosend将整个数据链一次传送给相应协议。

5. 传送数据给协议

395-414 对于PR_ATOMIC协议，当整个数据链被传送给协议后， resid总是等于0，并且控制跳出两个循环后至release处。如果PR_ATOMIC没有被置位，且当还有数据要发送并有缓存空间时，则 sosend继续往mbuf中写数据。如果缓存中没有闲置空间，但仍然有数据要发送，则 sosend回到循环开始，等待闲置空间来写下一个 mbuf。如果所有数据都发送完，则两个循环结束。

6. 释放缓存

414-422 当所有数据都传送给协议后，给插口缓存解锁，释放多余的 mbuf缓存，然后返回。

sosend的详细情况将分四个部分来描述：

- 初始化(图16-23)
- 差错和资源检查(图16-24)
- 数据传送(图16-25)
- 协议处理(图16-26)

sosend的第一部分初始化变量，如图 16-23所示。

7. 计算传送大小和语义

279-284 如果sosendallatonce等于true(任何设置了PR_ATOMIC的协议)或数据已经

通过top中的mbuf链传送给sosend，则将设置atomic。这个标志控制数据是作为一个mbuf链还是作为多个独立的mbuf传送给协议。

285-297 resid等于iovec缓存中的数据字节数或top中的mbuf链中的数据字节数。习题16.1讨论为什么resid可能等于负数的问题。

uipc_socket.c

```
279     struct proc *p = curproc;    /* XXX */
280     struct mbuf **mp;
281     struct mbuf *m;
282     long     space, len, resid;
283     int      clen = 0, error, s, dontroute, mlen;
284     int      atomic = sosendallatonce(so) || top;

285     if (uio)
286         resid = uio->uio_resid;
287     else
288         resid = top->m_pkthdr.len;
289     /*
290      * In theory resid should be unsigned.
291      * However, space must be signed, as it might be less than 0
292      * if we over-committed, and we must use a signed comparison
293      * of space and resid.  On the other hand, a negative resid
294      * causes us to loop sending 0-length segments to the protocol.
295      */
296     if (resid < 0)
297         return (EINVAL);
298     dontroute =
299         (flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
300         (so->so_proto->pr_flags & PR_ATOMIC);
301     p->p_stats->p_ru.ru_msgsnd++;
302     if (control)
303         clen = control->m_len;
304 #define snderr(errno)    { error = errno; splx(s); goto release; }
```

uipc_socket.c

图16-23 sosend 函数：初始化

8. 关闭路由

298-303 如果仅仅要求对这个报文不通过路由表进行路由选择，则设置 dontroute。clen等于在可选的控制缓存中的字节数。

304 宏snderr传送差错代码，重新使能协议处理，控制跳转到out执行解锁和释放缓存的工作。这个宏简化函数内的差错处理工作。

图16-24显示的sosend代码功能是检查差错条件和等待发送缓存中的闲置空间。

309 当检查差错情况时，为防止缓存发生改变，协议处理被挂起。在每一次数据传送之前，sosend要检查以下几种差错情况：

310-311 • 如果插口输出被禁止(即，TCP连接的写道通已经被关闭)，则返回EPIPE。

312-313 • 如果插口正处于差错状态(例如，前一个数据报可能已经产生了一个ICMP不可达的差错)，则返回so_error。如果差错出现之前数据已经被收到，则 sendit忽略这个差错(图16-21的第389行)。

314-318 • 如果协议请求连接且连接还没有建立或连接请求还没有启动，则返回ENOTCONN。sosend允许只有控制信息但没有数据的写操作，即使连接还没有建立。

Internet协议并不使用这个特点，但TP4用它在连接请求中发送数据，证实连接请

求，在断连请求中发送数据。

319-321 • 如果在无连接协议中没有指定目的地址 (例如，进程调用 send但并没有用connect建立目的地址)，则返回EDESTADDREQ。

uipc_socket.c

```
309         s = splnet();
310         if (so->so_state & SS_CANTSENDMORE)
311             snderr(EPIPE);
312         if (so->so_error)
313             snderr(so->so_error);
314         if ((so->so_state & SS_ISCONNECTED) == 0) {
315             if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
316                 if ((so->so_state & SS_ISCONFIRMING) == 0 &&
317                     !(resid == 0 && clen != 0))
318                     snderr(ENOTCONN);
319             } else if (addr == 0)
320                 snderr(EDESTADDRREQ);
321         }
322         space = sbspace(&so->so_snd);
323         if (flags & MSG_OOB)
324             space += 1024;
325         if (atomic && resid > so->so_snd.sb_hiwat ||
326             clen > so->so_snd.sb_hiwat)
327             snderr(EMSGSIZE);
328         if (space < resid + clen && uio &&
329             (atomic || space < so->so_snd.sb_lowat || space < clen)) {
330             if (so->so_state & SS_NBIO)
331                 snderr(EWOULDBLOCK);
332             sbunlock(&so->so_snd);
333             error = sbwait(&so->so_snd);
334             splx(s);
335             if (error)
336                 goto out;
337             goto restart;
338         }
339         splx(s);
340         mp = &top;
341         space -= clen;
```

uipc_socket.c

图16-24 sosend 函数：差错和资源检查

9. 计算可用空间

322-324 sbspace函数计算发送缓存中剩余的闲置空间字节数。这是一个基于缓存高水位标记的管理上的限制，但也是 sb_mbmax对它的限制，其目的是为了防止太多的小报文消耗太多的mbuf缓存(图16-6)。sosend通过放宽缓存限制到1024个字节来给予带外数据更高的优先级。

10. 强制实施报文大小限制

325-327 如果atomic被置位，并且报文大于高水位标记(high-watermark)，则返回EMSGSIZE；报文因为太大而不被协议接受，即使缓存是空的。如果控制信息的长度大于高水位标记，同样返回EMSGSIZE。这是限制数据或记录大小的测试代码。

11. 等待更多的空间吗？

328-329 如果发送缓存中的空间不够，数据来源于进程 (而不是来源于内核中的top)，并且下列条件之一成立，则sosend必须等待更多的空间：

• 报文必须一次传送给协议(atomic为真)；或
• 报文可以分段传送，但闲置空间大小低于低水位标记；或
• 报文可以分段传送，但可用空间存放不下控制信息。

当数据通过top传送给sosend (即，uio为空)时，数据已经在mbuf缓存中。因此，sosend忽略缓存高、低水位标记限制，因为不需要附加的缓存来保存数据。

如果在测试中，忽略发送缓存的低水位标记，在插口层和运输层之间将出现一种有趣的交互过程，它将导致性能下降。[Crowcroft et al. 1992]提供了有关这个问题的详细情况。

12. 等待空间

330-338 如果sosend必须等待缓存且插口是非阻塞的，则返回EWOULDBLOCK。同时，缓存锁被释放，sosend调用sbwait等待，直到缓存状态发生变化。当sbwait返回后，sosend重新使能协议处理，并且跳转到restart获取缓存锁，检查差错和缓存空间。如果条件满足，则继续执行。

默认情况下，sbwait阻塞直到可以发送数据。通过SO_SNDTIMEO插口选项改变缓存中的sb_timeo，进程可以设置等待时间的上限。如果定时器超时，则返回EWOULDBLOCK。回想一下图16-21，如果数据已经被成功发送给协议，则sendit忽略这个差错。这个定时器并不限制整个调用的时间，而仅仅是限制写两个mbuf缓存之间的不活动时间。

339-341 在这点上，sosend已经知道一些数据已传送给协议。splx使能中断，因为sosend从进程复制数据到内核相对较长的时间间隔内不应该被阻塞。mp包含一个指针，用来构造mbuf链。在sosend从进程复制任何数据之前，可用缓存的数量需减去控制信息的大小(clen)。

图16-25显示了sosend从进程复制数据到一个或多个内核中的mbuf中的代码段。

13. 分配分组首部或标准mbuf

351-360 当atomic被置位时，这段代码在第一次循环时分配一个分组首部，随后分配标准的mbuf缓存。如果atomic没有被置位，则这段代码总是分配一个分组首部，因为进入循环之前，top总是被清除。

14. 尽可能用簇

361-371 如果报文足够大使得为其分配一个簇是值得的，并且space大于或等于MCLBYTES，则调用MCLGET分配一个簇同mbuf连在一起。当space小于MCLBYTES时，额外的2048个字节将超过缓存分配限制，因为即使resid小于MCLBYTES，整个簇也将被分配。

如果调用MCLGET失败，sosend跳转到nopages，用一个标准的mbuf代替一个外部簇。

对MINCLSIZE的测试应该用>，而不是≥，因为208(MINCLSIZE)个字节的写操作只适合小于两个mbuf的情况。

如果atomic被设置(例如，UDP)，则mbuf链表示一个数据报或记录，并且在第一个簇的前面为协议首部保留max_hdr个字节。而后续的簇因为是同一条链的一部分，所以不需要再为协议首部保留空间。

如果atomic没有被置位(如，TCP)，则不需要保留空间，因为sosend不知道协议如何将发送的数据进行分段。

需要注意的是，space由簇大小(2048个字节)而不是len来决定，len等于放在簇中的数据的字节数(习题16-2)。

uipc_socket.c

```
351                 do {
352                     if (top == 0) {
353                         MGETHDR(m, M_WAIT, MT_DATA);
354                         mlen = MHLEN;
355                         m->m_pkthdr.len = 0;
356                         m->m_pkthdr.rcvif = (struct ifnet *) 0;
357                     } else {
358                         MGET(m, M_WAIT, MT_DATA);
359                         mlen = MLEN;
360                     }
361                     if (resid >= MINCLSIZE && space >= MCLBYTES) {
362                         MCLGET(m, M_WAIT);
363                         if ((m->m_flags & M_EXT) == 0)
364                             goto nopages;
365                         mlen = MCLBYTES;
366                         if (atomic && top == 0) {
367                             len = min(MCLBYTES - max_hdr, resid);
368                             m->m_data += max_hdr;
369                         } else
370                             len = min(MCLBYTES, resid);
371                         space -= MCLBYTES;
372                     } else {
373                       nopages:
374                         len = min(min(mlen, resid), space);
375                         space -= len;
376                         /*
377                          * For datagram protocols, leave room
378                          * for protocol headers in first mbuf.
379                          */
380                         if (atomic && top == 0 && len < mlen)
381                             MH_ALIGN(m, len);
382                     }
383                     error = uiomove(mtod(m, caddr_t), (int) len, uio);
384                     resid = uio->uio_resid;
385                     m->m_len = len;
386                     *mp = m;
387                     top->m_pkthdr.len += len;
388                     if (error)
389                         goto release;
390                     mp = &m->m_next;
391                     if (resid <= 0) {
392                         if (flags & MSG_EOR)
393                             top->m_flags |= M_EOR;
394                         break;
395                     }
396                 } while (space > 0 && atomic);
```

uipc_socket.c

图16-25 sosend 函数：数据传送

15. 准备mbuf

372-382 如果不用簇，存储在 mbuf中的字节数受下面三个量中最小一个量的限制： (1) mbuf中的可用空间；(2) 报文的字节数；(3) 缓存的空间。

如果atomic被置位，则利用MH_ALIGN可知数据在链中的第一个缓存的尾部。如果数据占居整个mbuf，则忽略MH_ALIGN。这一点可能导致没有足够的空间来存放协议首部，主要取决于有多少数据存放在mbuf中。如果atomic没有被置位，则没有为协议首部保留空间。

16. 从进程复制数据

383-395 uiomove从进程复制len个字节的数据到mbuf。传送完成后，更新mbuf的长度，前面的mbuf连接到新的mbuf(或top指向第一个mbuf)，更新mbuf链的长度。如果在传送过程中发生差错，则sosend跳转到release。

一旦最后一个字节传送完毕，如果进程设置了MSG_EOR，则设置分组中的M_EOR，然后sosend跳出循环。

MSG_EOR仅用于有显式的记录边界的协议，如OSI协议簇中的TP4。TCP不支持逻辑记录因而忽略MSG_EOR标志。

17. 写另一个缓存吗？

396 如果设置了atomic，sosend回到循环开始，写另一个mbuf。

对space>0的测试好像无关紧要。当atomic没有被设置时，space也是无关紧要的，因为一次只传送一个mbuf给协议。如果设置了atomic，只有当有足够的缓存空间来存放整个报文时才进入这个循环。参考习题16-2。

sosend的最后一段代码的功能是传送数据和控制mbuf给插口指定的协议，如图16-26所示。

uipc_socket.c

```
397             if (dontroute)
398                 so->so_options |= SO_DONTROUTE;
399             s = splnet();       /* XXX */
400             error = (*so->so_proto->pr_usrreq) (so,
401                                 (flags & MSG_OOB) ? PRU_SENDOOB : PRU_SEND,
402                                                 top, addr, control);
403             splx(s);
404             if (dontroute)
405                 so->so_options &= ~SO_DONTROUTE;
406             clen = 0;
407             control = 0;
408             top = 0;
409             mp = &top;
410             if (error)
411                 goto release;
412         } while (resid && space > 0);
413     } while (resid);
```

uipc_socket.c

图16-26 sosend 函数：协议分散

397-405 在传送数据到协议层的前后，可能通过SO_DONTROUTE选项选择是否利用路由表为这个报文选择路由。这是唯一的一个针对单个报文的选项，如图16-23所示，在写期间通过MSG_DONTROUTE标志来控制路由选择。

为了防止协议在处理报文期间pr_usrreq阻塞中断，pr_usrreq被放在splnet函数和splx函数之间执行。一些协议(如UDP)可能在进行输出处理期间并不阻塞中断，但插口层得不到这些信息。

如果进程传送的是带外数据，则sosend发送PRU_SENDOOB请求；否则，它发送PRU_SEND请求。同时将地址和控制mbuf传送给协议。

406-413 因为控制信息只需传送给协议一次，所以将clen、control、top和mp初始化，然后为传送报文的下一部分构造新的mbuf链。只有atomic没有被设置时(如TCP)，resid才

可能等于非0。在这种情况下，如果缓存中仍然有空间，则 sosend回到循环开始，继续写另一个mbuf。如果没有可用空间，则sosend回到循环开始，等待可用空间(图16-24)。

在第23章我们将了解到不可靠的协议，如 UDP，立即将数据排队等待发送。第 26章描述可靠的协议，如TCP，将数据放到插口发送缓存直到数据被发送和确认。

16.7.3 sosend函数小结

sosend是一个比较复杂的函数。它共有 142行，包含3个嵌套的循环，一个利用 goto实现的循环，两个基于是否设置 PR_ATOMIC的代码分支，两个并行锁。像许多其他软件一样，复杂性是多年积累的结果。 NFS加入MSG_DONTWAIT功能以及从mbuf链接收数据而不是从进程那里接收数据。 SS_ISCONFIRMING状态和MSG_EOR标志是为处理OSI协议连接和记录功能而加入的。

比较好的做法是为每一种协议实现一个独立的 sosend函数，通过分散指针 pr_send给protosw入口来实现。[Partridge and Pink 1993]中提出并实现了这种方法。

16.7.4 性能问题

如图16-25所描述的， sosend尽可能地以mbuf为单位将报文传送到协议层。与将一个报文用一个mbuf链的形式一次建立并传送给协议层的方法相比，这种做法导致了更多的调用，但是[Jacobson 1998a]说明了这种做法增加了并行性，因而获得了较好的性能。

一次传送一个 mbuf(2048个字节)允许CPU在网络硬件传输数据的同时准备一个分组。同发送一个大的 mbuf链相比：构造一个大的 mbuf链的同时，网络和接收系统是空闲的。在[Jacobson 1998a]描述的系统中，这种改变导致了网络吞吐量增加 20%。

有一点非常重要，即确保发送缓存的大小总是大于连接的带宽和时延的乘积 (卷1的第20.7节)。例如，如果TCP认为一条连接在收到确认之前能保留 20个报文段，那么发送缓存必须大到足够存储20个未被确认的报文段。如果发送缓存太小， TCP在收到第一个确认之前将用完数据，连接将在一段时间内是空闲的。

16.8 read、readv、recvfrom和recvmsg系统调用

我们将read、readv、recvfrom和recvmsg系统调用统称为读系统调用，从网络连接上接收数据。同recvmsg相比，前三个系统调用比较简单。 recvmsg因为比较通用而复杂得多。图16-27给出了这四个系统调用和一个库函数 (recv)的特点。

函 数	描述符类型	缓 存 数 量	返回发送者的地址吗?	标 志?	返回控制信息?
read	任何类型	1			
readv	任何类型	[1..UIO_MAXIOV]			
recv	插口	1		•	
recvfrom	插口	1	•	•	
recvmsg	插口	[1..UIO-MAXIOV]	•	•	•

图16-27 读系统调用

在Net/3中，recv是一个库函数，通过调用 recvfrom来实现的。为了同以前编

译的程序二进制兼容，内核将旧的recv系统调用映射到函数orecv。我们仅仅讨论recvfrom的内核实现。

只有read和readv系统调用适用于各类描述符，其他的调用只适用于插口描述符。

同写调用一样，通过iovec结构数组来指定多个缓存。对数据报协议，recvfrom和recvmsg返回每一个收到的数据报的源地址。对于面向连接的协议，getpeername返回连接对方的地址。与接收调用相关的标志参考第16.11节。

同写调用一样，读调用利用一个公共函数soreceive来做所有工作。图16-28说明读系统调用的流程。

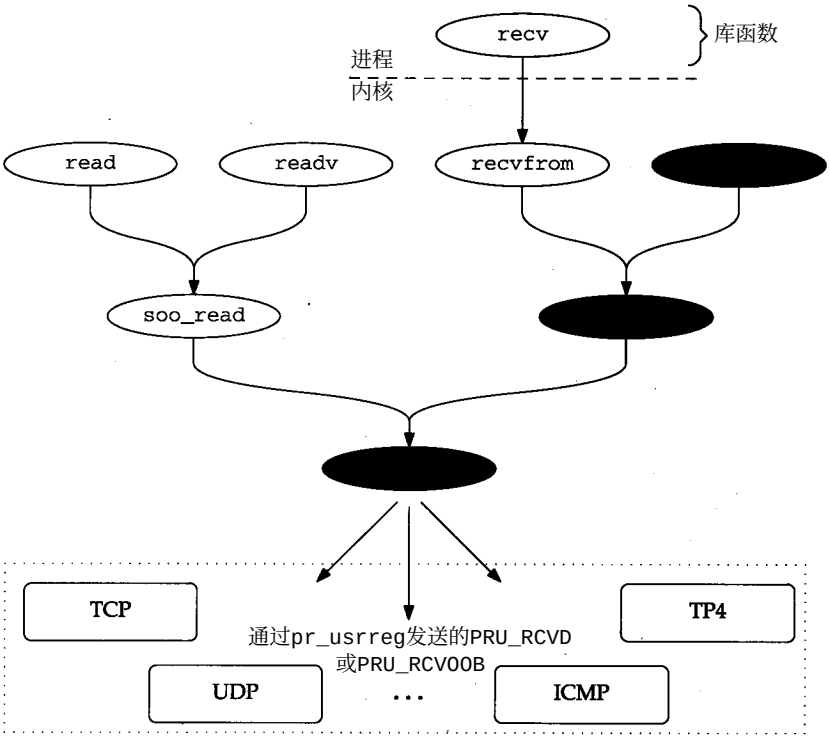

图16-28 所有插口输入都由soreceive处理

我们仅仅讨论图16-28中的带阴影的函数。其余的函数读者可以自己查阅有关资料。

16.9 recvmsg系统调用

recvmsg函数是最通用的读系统调用。如果一个进程使用任何一个其他的读系统调用，且地址、控制信息和接收标志的值还未定，则系统可能在没有任何通知的情况下丢弃它们。图16-29显示了recvmsg函数。

433-445 recvmsg的三个参数是：插口描述符；类型为msghdr的结构指针，几个控制标志。

1. 复制iov数组

446-461 同sendmsg一样，recvmsg将msghdr结构复制到内核，如果自动分配的数组

aiov太小，则分配一个更大的iovec数组，并且将数组单元从进程复制到由iov指向的内核数组(第16.4节)。将第三个参数复制到msghdr结构中。

uipc_syscalls.c

```
433 struct recvmsg_args {
434     int     s;
435     struct msghdr *msg;
436     int     flags;
437 };

438 recvmsg(p, uap, retval)
439 struct proc *p;
440 struct recvmsg_args *uap;
441 int    *retval;
442 {
443     struct msghdr msg;
444     struct iovec aiov[UIO_SMALLIOV], *uiov, *iov;
445     int     error;

446     if (error = copyin((caddr_t) uap->msg, (caddr_t) & msg, sizeof(msg)))
447         return (error);
448     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
449         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
450             return (EMSGSIZE);
451         MALLOC(iov, struct iovec *,
452                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
453                M_WAITOK);
454     } else
455         iov = aiov;
456     msg.msg_flags = uap->flags;
457     uiov = msg.msg_iov;
458     msg.msg_iov = iov;
459     if (error = copyin((caddr_t) uiov, (caddr_t) iov,
460                        (unsigned) (msg.msg_iovlen * sizeof(struct iovec))))
461                 goto done;
462     if ((error = recvit(p, uap->s, &msg, (caddr_t) 0, retval)) == 0) {
463         msg.msg_iov = uiov;
464         error = copyout((caddr_t) & msg, (caddr_t) uap->msg, sizeof(msg));
465     }
466   done:
467     if (iov != aiov)
468         FREE(iov, M_IOV);
469     return (error);
470 }
```

uipc_syscalls.c

图16-29 recvmsg 系统调用

2. recvit和释放缓存

462-470 recvit收完数据后，将更新过的缓存长度和标志的msghdr结构再复制到进程。如果分配了一个更大的iovec结构，则返回之前释放它。

16.10 recvit函数

recvit函数被recv、recvfrom和recvmsg调用，如图16-30所示。基于recv *xxx*调用提供的msghdr结构，recvit函数为soreceive的处理准备了一个uio结构。

471-500 getsock为描述符s返回一个file结构，然后recvit初始化uio结构，该结构描述从内核到进程之间的一次数据传送。通过对iovec数组中的msg_iovlen字段求和得到

传送的字节数。结果保留在uio_resid中的len中。

uipc_syscalls.c

```
471 recvit(p, s, mp, namelenp, retsize)
472 struct proc *p;
473 int     s;
474 struct msghdr *mp;
475 caddr_t namelenp;
476 int    *retsize;
477 {
478     struct file *fp;
479     struct uio auio;
480     struct iovec *iov;
481     int     i;
482     int     len, error;
483     struct mbuf *from = 0, *control = 0;
484     if (error = getsock(p->p_fd, s, &fp))
485         return (error);
486     auio.uio_iov = mp->msg_iov;
487     auio.uio_iovcnt = mp->msg_iovlen;
488     auio.uio_segflg = UIO_USERSPACE;
489     auio.uio_rw = UIO_READ;
490     auio.uio_procp = p;
491     auio.uio_offset = 0;        /* XXX */
492     auio.uio_resid = 0;
493     iov = mp->msg_iov;
494     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
495         if (iov->iov_len < 0)
496             return (EINVAL);
497         if ((auio.uio_resid += iov->iov_len) < 0)
498             return (EINVAL);
499     }
500     len = auio.uio_resid;
```

uipc_syscalls.c

图16-30 recvit 函数：初始化uio结构

recvit的第二部分调用soreceive，并且将结果复制到进程，如图16-31所示。

uipc_syscalls.c

```
501     if (error = soreceive((struct socket *) fp->f_data, &from, &auio,
502       (struct mbuf **) 0, mp->msg_control ? &control : (struct mbuf **) 0,
503                             &mp->msg_flags)) {
504         if (auio.uio_resid != len && (error == ERESTART ||
505                                       error == EINTR || error == EWOULDBLOCK))
506             error = 0;
507     }
508     if (error)
509         goto out;
510     *retsize = len - auio.uio_resid;
511     if (mp->msg_name) {
512         len = mp->msg_namelen;
513         if (len <= 0 || from == 0)
514             len = 0;
515         else {
516             if (len > from->m_len)
517                 len = from->m_len;
518             /* else if len < from->m_len ??? */
519             if (error = copyout(mtod(from, caddr_t),
```

图16-31 recvit 函数：返回结果

```
520                                     (caddr_t) mp->msg_name, (unsigned) len))
521                 goto out;
522         }
523         mp->msg_namelen = len;
524         if (namelenp &&
525             (error = copyout((caddr_t) & len, namelenp, sizeof(int)))) {
526             goto out;
527         }
528     }
529     if (mp->msg_control) {
530         len = mp->msg_controllen;
531         if (len <= 0 || control == 0)
532             len = 0;
533         else {
534             if (len >= control->m_len)
535                 len = control->m_len;
536             else
537                 mp->msg_flags |= MSG_CTRUNC;
538             error = copyout((caddr_t) mtod(control, caddr_t),
539                             (caddr_t) mp->msg_control, (unsigned) len);
540         }
541         mp->msg_controllen = len;
542     }
543   out:
544     if (from)
545         m_freem(from);
546     if (control)
547         m_freem(control);
548     return (error);
549 }
```

uipc_syscalls.c

图16-31 (续)

1. 调用soreceive

501-510 soreceive实现从插口缓存中接收数据的最复杂的功能。传送的字节数保存在*retsize中，并且返回给进程。如果有些数据已经被复制到进程后信号出现或阻塞出现(len不等于uio_resid)，则忽略差错，并返回已经传送的字节。

2. 将地址和控制信息复制到进程

511-542 如果进程传入了一个存放地址或控制信息或两者都有的缓存，则 recvit将结果写入该缓存，并且根据soreceive返回的结果调整它们的长度。如果缓存太小，则地址信息可能被截掉。如果进程在发送读调用之前保留缓存的长度，将该长度同内核返回的namelenp变量(或sockaddr结构的长度域)相比较就可以发现这个差错。通过设置msg_flags中的MSG_CTRUNC标志来报告这种差错，参考习题16-7。

3. 释放缓存

543-549 从out开始，释放存储源地址和控制信息的mbuf缓存。

16.11 soreceive函数

soreceive函数将数据从插口的接收缓存传送到进程指定的缓存。某些协议还提供发送者的地址，地址可以同可能的附加控制信息一起返回。在讨论它的代码之前，先来讨论接收操作，带外数据和插口接收缓存的组织的含义。

图16-32 列出了在执行soreceive期间内核知道的一些标志。

flags	描 述	参 考
MSG_DONTWAIT	在调用期间不等待资源	图16-38
MSG_OOB	接收带外数据而不是正常的数据	图16-39
MSG_PEEK	接收数据的副本而不取走数据	图16-43
MSG_WAITALL	在返回之前等待数据写缓存	图16-50

图16-32 recv *xxx*系统调用：传递给内核的标志值

recvmsg是唯一返回标志字段给进程的读系统调用。在其他的系统调用中，控制返回给进程之前，这些信息被内核丢弃。图16-33列出了在msghdr中recvmsg能设置的标志。

msg_flags	描 述	参 考
MSG_CTRUNC	控制信息的长度大于提供的缓存长度	图16-31
MSG_EOR	收到的数据标志一个逻辑记录的结束	图16-48
MSG_OOB	缓存中包含带外数据	图16-45
MSG_TRUNC	收到的报文的长度大于提供的缓存长度	图16-51

图16-33 recvmsg 系统调用：内核返回的msg_flag 值

16.11.1 带外数据

带外数据(OOB)在不同的协议中有不同的含义。一般来说，协议利用已建立的通信连接来发送OOB数据。OOB数据可能与已发送的正常数据同序。插口层支持两种与协议无关的机制来实现对OOB数据的处理：标记和同步。本章讨论插口层实现的抽象的 OOB机制。UDP不支持OOB数据。TCP的紧急数据机制与插口层的OOB数据之间的关系在TCP一章中描述。

发送进程通过在send*xxx*调用中设置MSG_OOB标志将数据标记为OOB数据。sosend将这个信息传递给插口协议，插口层收到这个信息后，对数据进行特殊处理，如加快发送数据或使用另一种排队策略。

当一个协议收到OOB数据后，并不将它放进插口的接收缓存而是放在其他地方。进程通过设置recv*xxx*调用中的MSG_OOB标志来接收到达的 OOB数据。另一种方法是，通过设置SO_OOBINLINE插口选项(见第17.3节)，接收进程可以要求协议将OOB数据放在正常的数据之内。当SO_OOBINLINE被设置时，协议将收到的OOB数据放进正常数据的接收缓存。在这种情况下，MSG_OOB不用来接收OOB数据。读调用要么返回所有的正常数据，要么返回所有的OOB数据。两种类型的数据从来不会在一个输入调用的输入缓存中混淆。进程使用 recvmsg来接收数据时，可以通过检查MSG_OOB标志来决定返回的数据是正常数据还是OOB数据。

插口层支持OOB数据和正常数据的同步接收，采用的方法是允许协议在正常数据流中标记OOB数据起始点。接收者可以在每一个读系统调用的后面，通过 SIOCATMARKioctl命令来检查是否已经达到 OOB数据的起始点。当接收正常的数据时，插口层确保在一个报文中只有在标记前的正常数据才会收到，使得接收者接收的数据不会超过标记。如果在接收者到达标记之前收到一些附加的OOB数据，标记就自动向前移。

16.11.2 举例

图16-34说明两种接收带外数据的方法。在两个例了中，字节 A~I作为正常数据接收，字

节J作为带外数据接收，字节K~L作为正常数据接收。接收进程已经接收了A之前(不包括A)的所有数据。

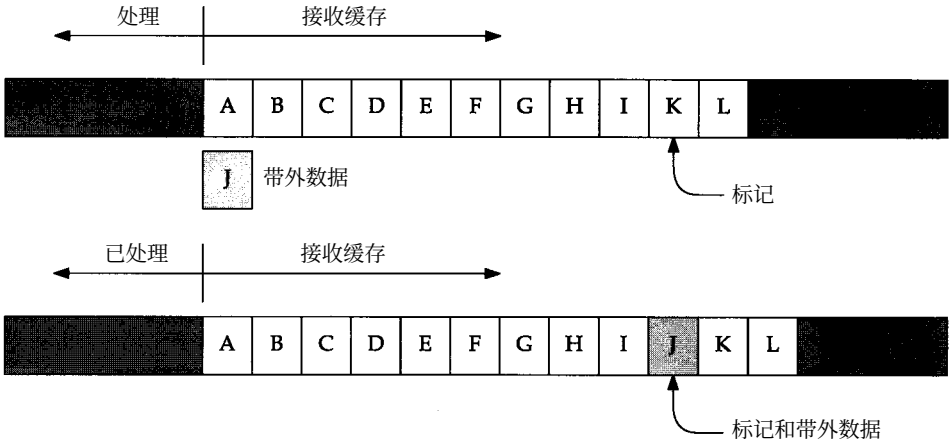

图16-34 接收带外数据

在第一个例子中，进程能够正确读出字节A~I，或者如果设置MSG_OOB，也能读出字节J。即使读请求的长度大于9个字节(A~I)，插口层也只返回9个字节，以免超过带外数据的同步标记。当读出字节I后，SIOCATMARK为真；对于到达带外数据标记的进程，不必读出字节J。

在第二个例了中，在SIOCATMARK为真时只能读字节A~I。第二次调用读字节J~L。

在图16-34中，字节J不是TCP的紧急数据指针指示的字节。在本例中，紧急指针指向的是字节K。有关细节请参考第29.7节。

16.11.3 其他的接收操作选项

进程能够通过设置标志MSG_PEEK来查看是否有数据到达。而数据仍然留在接收队列中，被下一个不设置MSG_PEEK的读调用读出。

标志MSG_WAITALL指示读调用只有在读到指定数量的数据后才返回。即使soreceive中有一些数据可以返回给进程，但它仍然要等到收到剩余的数据后才返回。

当标志MSG_WAITALL被设置后，soreceive只有在下列情况下可以在没有读完指定长度的数据时返回：

- 连接的读通道被关闭；
- 插口的接收缓存小于所读数据的大小；
- 在进程等待剩余的数据时差错出现；
- 带外数据到达；或
- 在读缓存被写满之前，一个逻辑记录的结尾出现。

NFS是Net/3中唯一使用MSG_WAITALL和MSG_DONTWAIT标志的软件。进程可以不通过ioctl或fcntl来选择非阻塞的I/O操作而是设置MSG_DONTWAIT标志来实现非阻塞的读系统调用。

16.11.4 接收缓存的组织：报文边界

对于支持报文边界的协议，每一个报文存放在一个 mbuf链中。接收缓存中的多个报文通

过m_nextpkt指针链接成一个mbuf队列(图2-21)。协议处理层加数据到接收队列，插口层从接收队列中移走数据。接收缓存的高水位标记限制了存储在缓存中的数据量。

如果PR_ATOMIC没有被置位，协议层尽可能多地在缓存中存放数据，丢弃输入数据中的不合要求的部分。对于 TCP，这就意味着到达的任何数据如果在接收窗口之外都将被丢弃。如果PR_ATOMIC被置位，缓存必须能够容纳整个报文，否则协议层将丢弃整个报文。对于UDP而言，如果接收缓存已满，则进入的数据报都将被丢弃，缓存满的原因可能是进程读数据报的速度不够快。

PR_ADDR被置位的协议使用sbappendaddr构造一个mbuf链，并将其加入到接收队列。缓存链包含一个存放报文源地址的 mbuf，0个或更多的控制mbuf，后面跟着0个或更多的包含数据的mbuf。

对于SOCK_SEQPACKET和SOCK_RDM协议，它们为每一个记录建立一个 mbuf链。如果PR_ATOMIC被置位，则调用 sbappendrecord，将记录加到接收缓存的尾部。如果PR_ATOMIC没有被置位(OSI的TP4)，则用sbappendrecord产生一个新的记录，其余的数据用sbappend加到这个记录中。

假定PR_ATOMIC就是表示缓存的组织结构是不正确的。例如， TP4中并没有PR_ATOMIC，而是用M_EOR标志来支持记录边界。

图16-35说明了由三个mbuf链(即三个数据报)组成的UDP接收缓存的结构。每一个 mbuf中都标有m_type的值。

在图16-35中，第三个数据报中有一些控制信息。三个 UDP插口选项能够导致控制信息被存入接收缓存。详细情况参考图 22-5和图23-7。

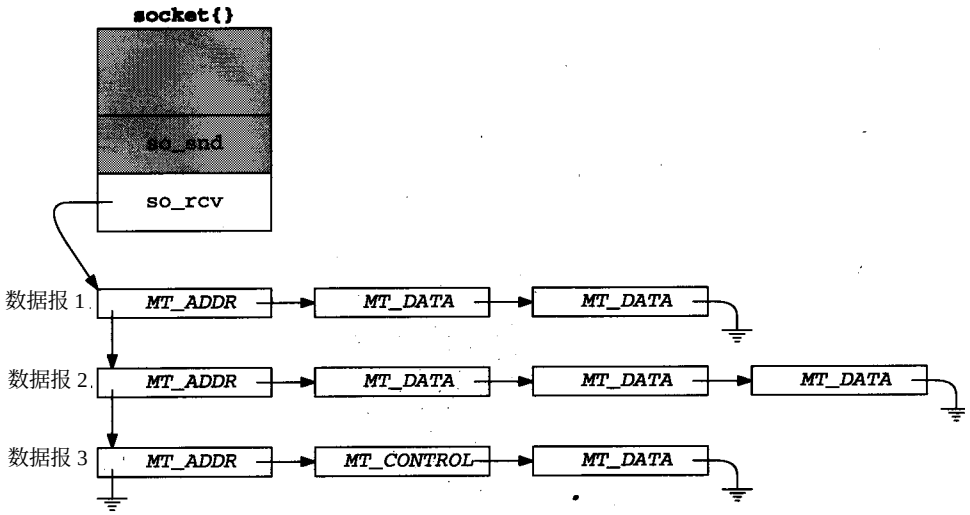

图16-35 包含三个数据报的UDP接收缓存

对于PR_ATOMIC协议，当收到数据时， sb_lowat被忽略。当没有设置PR_ATOMIC时，sb_lowat的值等于读系统调用返回的最小的字节数。但也有一些例外，如图 16-41所示。

16.11.5 接收缓存的组织：没有报文边界

当协议不需维护报文边界 (即SOCK_STREAM协议，如TCP)时，通过sbappend将进入的

数据加到缓存中的最后一个mbuf链的尾部。如果进入的数据长度大于缓存的长度，则数据将被截掉，sb_lowat为一个读系统调用返回的字节数设置了一个下限。

图16-36说明了仅仅包含正常数据的TCP接收缓存的结构。

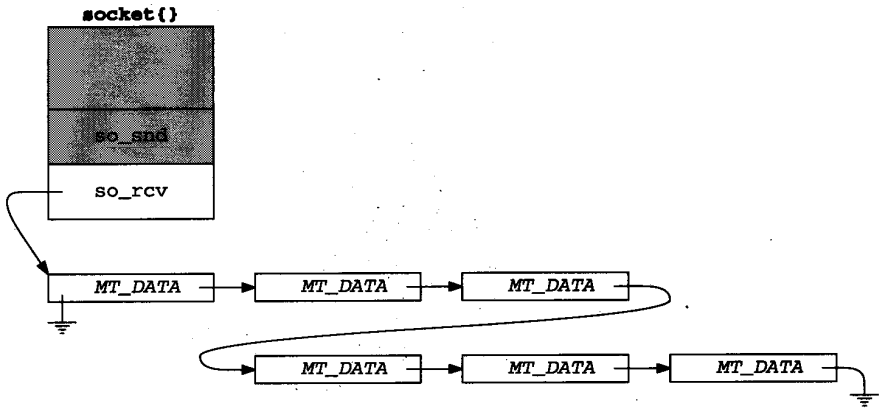

图16-36 TCP的so_rcv 缓存

16.11.6 控制信息和带外数据

不像TCP，一些流协议支持控制信息，并且调用sbappendcontrol将控制信息和相关数据作为一个新的mbuf链加入接收缓存。如果协议支持内含OOB数据，则调用sbinsertoob插入一个新的mbuf链到任何包含OOB数据的mbuf链之后，但在任何包含正常数据的mbuf链之前。这一点确保进入的OOB数据总是排在正常数据之前。

图16-37说明包含控制信息和OOB数据的接收缓存的结构。

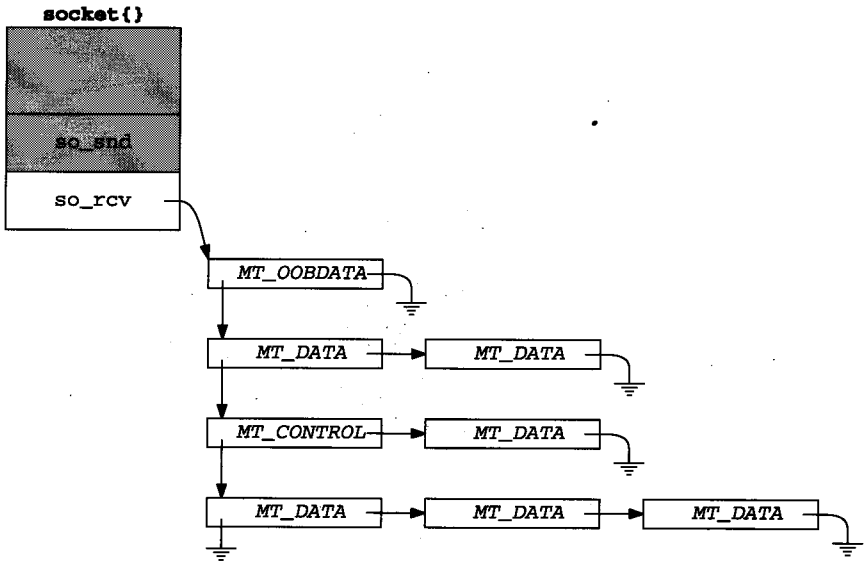

图16-37 带有控制信息和OOB数据的so_rcv 缓存

Unix域流协议支持控制信息，OSI TP4协议支持MT_OOBDATA mbuf。TCP既不支持控制信息，也不支持MT_OOBDATA形式的带外数据。如果TCP的紧急指针指向的字节存储在数据内(SO_OOBINLINE被设置)，那么该字节是正常数据而不是OOB数据。TCP对紧急指针和相

关数据的处理在第29.7节中讨论。

16.12 soreceive代码

我们现在有足够的背景信息来详细讨论 soreceive函数。在接收数据时， soreceive必须检查报文边界，处理地址和控制信息以及读标志所指定的任何特殊操作 (图16-32)。一般来说， soreceive的一次调用只处理一个记录，并且尽可能返回要求读的字节数。图 16-38显示了soreceive函数的大概情况。

uipc_socket.c

```
439 soreceive(so, paddr, uio, mp0, controlp, flagsp)
440 struct socket *so;
441 struct mbuf **paddr;
442 struct uio *uio;
443 struct mbuf **mp0;
444 struct mbuf **controlp;
445 int    *flagsp;
446 {
447     struct mbuf *m, **mp;
448     int     flags, len, error, s, offset;
449     struct protosw *pr = so->so_proto;
450     struct mbuf *nextrecord;
451     int     moff, type;
452     int     orig_resid = uio->uio_resid;

453     mp = mp0;
454     if (paddr)
455         *paddr = 0;
456     if (controlp)
457         *controlp = 0;
458     if (flagsp)
459         flags = *flagsp & ~MSG_EOR;
460     else
461         flags = 0;

                    /* MSG_OOB processing and */
                /* implicit connection confirmation */

483   restart:
484     if (error = sblock(&so->so_rcv, SBLOCKWAIT(flags)))
485         return (error);
486     s = splnet();
487     m = so->so_rcv.sb_mb;

                /* if necessary, wait for data to arrive */

542   dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;

                /* process address and control information */

591     if (m) {
```

图16-38 soreceive 函数：概述

```
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
                        /* process data */
693     }                           /* while more data and more space to fill */
                        /* cleanup */
715   release:
716     sbunlock(&so->so_rcv);
717     splx(s);
718     return (error);
719 }
```

uipc_socket.c

图16-38 (续)

439-446 soreceive有六个参数。指向插口的so指针。指向存放接收地址信息的mbuf缓存的指针*paddr。如果mp0指向一个mbuf链，则soreceive将接收缓存中的数据传送到*mp0指向的mbuf缓存链。在这种情况下，uio结构中只有用来记数的uio_resid字段是有意义的。如果mp0为空，则soreceive将数据传送到uio结构中指定的缓存。*controlp指向包含控制信息的mbuf缓存。soreceive将图16-33中描述的标志存放在*flagsp。

447-453 soreceive一开始将pr指向插口协议的交换结构，并将uio_resid(接收请求的大小)保存在orig_resid。如果将控制或地址信息从内核复制到进程，则将orig_resid清0。如果复制的是数据，则更新uio_resid。不管哪一种情况，orig_resid都不可能等于uio_resid。soreceive函数的最后处理要利用这一事实(图16-51)。

454-461 在这一段代码中，首先将*paddr和*controlp置空。在将MSG_EOR标志清0后，将传给soreceive的*flagsp的值保存在flags中(习题16.8)。flagsp是一个用来返回结果的参数，但是只有recvmsg系统调用才能收到结果。如果flagsp为空，则将flags清0。

483-487 在访问接收缓存之前，调用sblock给缓存加锁。如果flags中没有设置MSG_DONTWAIT标志，则soreceive必须等待加锁成功。

支持在内核中从NFS发调用到插口层带来了另一个负作用。

挂起协议处理，使得在检查缓存过程中soreceive不被中断。m是接收缓存中的第一个mbuf链上的第一个mbuf。

1. 如果需要，等待数据

488-541 soreceive要检查几种情况，并且如果需要，它可能要等待接收更多的数据才继续往下执行。如果soreceive在这里进入睡眠状态，则在它醒来后跳转到restart查看是否有足够的数据到达。这个过程一直继续，直到收到足够的数据为止。

542-545 当soreceive已收到足够的数据来满足读请求所要求的数据量时，就跳转到dontblock。并将指向接收缓存中的第二个mbuf链的指针保存在nextrecord中。

2. 处理地址和控制信息

542-545 在传送数据之前，首先处理地址信息和控制信息。

3. 建立数据传送

591-597 因为只有OOB数据或正常数据是在一次soreceive调用中传送，这段代码的功能就是记住队列前端的数据的类型，这样在类型改变时，soreceive能够停止传送。

4. 传送数据循环

598-692 只要缓存中还有mbuf (m不空)，请求的数据还没有传送完毕(uio_resid>0)，且没有差错出现，本循环就不会退出。

退出处理

693-719 剩余的代码主要是更新指针、标志和偏移；释放插口缓存锁；使能协议处理并返回。

图16-39说明soreceive对OOB数据的处理。

uipc_socket.c

```
462     if (flags & MSG_OOB) {
463         m = m_get(M_WAIT, MT_DATA);
464         error = (*pr->pr_usrreq) (so, PRU_RCVOOB,
465                 m, (struct mbuf *) (flags & MSG_PEEK), (struct mbuf *) 0);
466         if (error)
467             goto bad;
468         do {
469             error = uiomove(mtod(m, caddr_t),
470                             (int) min(uio->uio_resid, m->m_len), uio);
471             m = m_free(m);
472         } while (uio->uio_resid && error == 0 && m);
473       bad:
474         if (m)
475             m_freem(m);
476         return (error);
477     }
```

uipc_socket.c

图16-39 soreceive 函数：带外数据

5. 接收OOB数据

462-477 因为OOB数据不存放在接收缓存中，所以soreceive为其分配一块标准的mbuf，并给协议发送PRU_RCVOOB请求。while循环将协议返回的数据复制到uio指定的缓存中。复制完成后，soreceive返回0或差错代码。

对于PRU_RCVOOB请求，UDP协议总是返回EOPNOTSUPP。关于TCP的紧急数据的处理的详细情况参考第30.2节。图16-40说明soreceive对连接信息的处理。

uipc_socket.c

```
478     if (mp)
479         *mp = (struct mbuf *) 0;
480     if (so->so_state & SS_ISCONFIRMING && uio->uio_resid)
481         (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
482                           (struct mbuf *) 0, (struct mbuf *) 0);
```

uipc_socket.c

图16-40 soreceive 函数：连接信息

6. 连接证实

478-482 如果返回的数据存放在 mbuf链中，则将 *mp初始化成空。如果插口处于

SO_ISCONFIRMING状态，PRU_RCVD请求告知协议进程想要接收数据。

SO_ISCONFIRMING状态仅用于OSI的流协议，TP4。在TP4中，直到一个用户级进程通过发送或接收数据的方式来证实连接，该连接才被认为已完全建立。在通过调用getpeername来获取对方的身份后，进程可能调用 shutdown或close来拒绝连接。

图16-38显示了图16-41中的代码在检查接收缓存时，接收缓存被加锁。 soreceive的这部分代码的功能是查看接收缓存中的数据是否能满足读系统调用的要求。

uipc_socket.c

```
488     /*
489      * If we have less data than requested, block awaiting more
490      * (subject to any timeout) if:
491      *   1. the current count is less than the low water mark, or
492      *   2. MSG_WAITALL is set, and it is possible to do the entire
493      *    receive operation at once if we block (resid <= hiwat).
494      *   3. MSG_DONTWAIT is not set
495      *
496      * If MSG_WAITALL is set but resid is larger than the receive buffer,
497      * we have to do the receive in sections, and thus risk returning
498      * a short count if a timeout or signal occurs after we start.
499      */
500     if (m == 0 || ((flags & MSG_DONTWAIT) == 0 &&
501                    so->so_rcv.sb_cc < uio->uio_resid) &&
502         (so->so_rcv.sb_cc < so->so_rcv.sb_lowat ||
503        ((flags & MSG_WAITALL) && uio->uio_resid <= so->so_rcv.sb_hiwat)) &&
504          m->m_nextpkt == 0 && (pr->pr_flags & PR_ATOMIC) == 0) {
```

uipc_socket.c

图16-41 soreceive 函数：数据够吗？

7. 读调用的请求能满足吗？

488-504 一般情况下，soreceive要等待直到接收缓存中有足够的数据来满足整个读请求。但是，有几种情况可能导致差错或返回比读请求要求少的数据。

- 接收缓存没有数据(m等于0)。
- 缓存中的数据不能满足读请求要求的数量 (sb_cc<uio_resid)并且没有设置MSG_DONTWAIT标志，最少的数据也得不到(sb_cc<sb_lowat)，且当该链到达时更多的数据能够加到链的后面(m_nextpkt等于0，且没有设置PR_ATOMIC)。
- 缓存中的数据不能满足读请求要求的数量，能得到最少的数据量，数据能够加到链中来，但是MSG_WAITALL指示soreceive必须等待直到缓存中的数据能满足读请求。

如果最后一种情况的条件能够满足，但是因为读请求的数据太大以至如果不阻塞等待就不能满足(uio_resid≤sb_hiwat)，soreceive就不等待而是继续往下执行。

如果接收缓存有数据，并且设置了MSG_DONTWAIT，则sorecevie不等待更多的数据。

有几种原因使得等待更多的数据是不合适的。在图 16-42中，soreceive要么检查三种情况，然后返回；要么等待更多的数据到达。

8. 等待更多的数据吗？

505-534 在此处，soreceive已经决定等待更多的数据来满足读请求。在等待之前，它需要检查以下几种情况：

505-512 • 如果插口处于差错状态，且缓存为空(m为空)，则soreceive返回差错代码。如

果有差错，但是接收缓存中有数据(m非空)，则返回缓存的数据；当下一个读调用来时，如果没有数据，就返回差错。如果设置了MSG_PEEK，就不清除差错，因为设置了MSG_PEEK的读调用不能改变插口的状态。

513-518 • 如果连接的读通道已经被关闭并且数据仍在接收缓存中，则sosend不等待而是将数据返回给进程(在dontblock的情况下)。如果接收缓存为空，则soreceive跳转到release，读系统调用返回0，表示连接的读通道已经被关闭。

519-523 • 如果接收缓存中包含带外数据或出现一个逻辑记录的结尾，则soreceive不等待，而是跳转到dontblock。

524-528 • 如果协议请求中的连接不存在，则设置差错代码为ENOTCONN，函数跳转到release。

529-534 • 如果读请求读0字节或插口是非阻塞的，则函数跳转到release，并返回0或EWOULDBLOCK(后一种情况)。

uipc_socket.c

```
505         if (so->so_error) {
506             if (m)
507                 goto dontblock;
508             error = so->so_error;
509             if ((flags & MSG_PEEK) == 0)
510                 so->so_error = 0;
511             goto release;
512         }
513         if (so->so_state & SS_CANTRCVMORE) {
514             if (m)
515                 goto dontblock;
516             else
517                 goto release;
518         }
519         for (; m; m = m->m_next)
520             if (m->m_type == MT_OOBDATA || (m->m_flags & M_EOR)) {
521                 m = so->so_rcv.sb_mb;
522                 goto dontblock;
523             }
524         if ((so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING)) == 0 &&
525             (so->so_proto->pr_flags & PR_CONNREQUIRED)) {
526             error = ENOTCONN;
527             goto release;
528         }
529         if (uio->uio_resid == 0)
530             goto release;
531         if ((so->so_state & SS_NBIO) || (flags & MSG_DONTWAIT)) {
532             error = EWOULDBLOCK;
533             goto release;
534         }
535         sbunlock(&so->so_rcv);
536         error = sbwait(&so->so_rcv);
537         splx(s);
538         if (error)
539             return (error);
540         goto restart;
541     }
```

uipc_socket.c

图16-42 soreceive 函数：等待更多的数据吗？

9. 是，等待更多的数据

535-541 此处soreceive已决定等待更多的数据，并且有理由这么做(即，将有数据到达)。在进程调用sbwait进入睡眠期间，缓存被解锁。如果因为差错或信号出现使得sbwait返回，则soreceive返回相应的差错；否则soreceive跳转到restart，查看接收缓存中的数据是否能够满足读请求。

同sosend中一样，进程能够利用SO_RCVTIMEO插口选项为sbwait设置一个接收定时器。如果在数据到达之前定时器超时，则sbwait返回EWOULDBLOCK。

定时器并不能总令人满意。因为当插口上有活动时，定时器每次都被重置。如果在一个超时间隔内至少有一个字节到达，则定时器从来不会超时，一直到设置了更长的超时值的读系统调用返回。sb_timeo是一个不活动定时器，并不要求超时值上限，但为了满足读系统调用，超时值的上限可能是必要的。

在此处，soreceive准备从接收缓存中传送数据。图16-43说明了地址信息的传送。

uipc_socket.c

```
542   dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;
546     if (pr->pr_flags & PR_ADDR) {
547         orig_resid = 0;
548         if (flags & MSG_PEEK) {
549             if (paddr)
550                 *paddr = m_copy(m, 0, m->m_len);
551             m = m->m_next;
552         } else {
553             sbfree(&so->so_rcv, m);
554             if (paddr) {
555                 *paddr = m;
556                 so->so_rcv.sb_mb = m->m_next;
557                 m->m_next = 0;
558                 m = so->so_rcv.sb_mb;
559             } else {
560                 MFREE(m, so->so_rcv.sb_mb);
561                 m = so->so_rcv.sb_mb;
562             }
563         }
564     }
```

uipc_socket.c

图16-43 soreceive 函数：返回地址信息

10. dontblock

542-545 nextrecord指向接收缓存中的下一条记录。在soreceive的后面，当第一个链被丢弃后，该指针被用来将剩余的mbuf放入插口缓存。

11. 返回地址信息

546-564 如果协议提供地址信息，如UDP，则将从mbuf链中删除包含地址的mbuf，并通过*paddr返回。如果paddr为空，则地址被丢弃。

在soreceive中，如果设置了MSG_PEEK，则数据仍留在缓存中。

图16-44中的代码处理缓存中的控制mbuf。

12. 返回控制信息

565-590 每一个包含控制信息的mbuf都将从缓存中删除(如果设置了MSG_PEEK，则不删除而是复制)，并连到*controlp。如果controlp为空，则丢弃控制信息。

uipc_socket.c

```
565     while (m && m->m_type == MT_CONTROL && error == 0) {
566         if (flags & MSG_PEEK) {
567             if (controlp)
568                 *controlp = m_copy(m, 0, m->m_len);
569             m = m->m_next;
570         } else {
571             sbfree(&so->so_rcv, m);
572             if (controlp) {
573                 if (pr->pr_domain->dom_externalize &&
574                     mtod(m, struct cmsghdr *)->cmsg_type ==
575                     SCM_RIGHTS)
576                         error = (*pr->pr_domain->dom_externalize) (m);
577                 *controlp = m;
578                 so->so_rcv.sb_mb = m->m_next;
579                 m->m_next = 0;
580                 m = so->so_rcv.sb_mb;
581             } else {
582                 MFREE(m, so->so_rcv.sb_mb);
583                 m = so->so_rcv.sb_mb;
584             }
585         }
586         if (controlp) {
587             orig_resid = 0;
588             controlp = &(*controlp)->m_next;
589         }
590     }
```

uipc_socket.c

图16-44 soreceive 函数：处理控制信息

如果进程准备接收控制信息，则协议定义了一个dom_externalize函数，一旦控制信息mbuf中包含SCM_RIGHTS(访问权限)，就调用dom_externalize函数。该函数执行内核中所有接收访问权限的操作。只有Unix域协议支持访问权限，有关细节在第7.3节已讨论过。如果进程不准备接收控制信息(controlp为空)，则丢弃控制mbuf。

直到处理完所有包含控制信息的mbuf或出现差错时，循环才退出。

对于Unix协议域，dom_externalize函数通过修改接收进程的文件描述符表来实现文件描述符的传送。

处理完所有的控制mbuf后，m指向链中的下一个mbuf。如果在地址或控制信息的后面，链中没有其他的mbuf，则m为空。例如，当一个长度为0的数据报进入接收缓存时就会出现这种情况。图16-45说明了soreceive准备从mbuf链中传送数据。

uipc_socket.c

```
591     if (m) {
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
```

uipc_socket.c

图16-45 soreceive 函数：准备传送mbuf

13. 准备传送数据

591-597 处理完控制mbuf后，链中应该只剩下正常数据、带外数据 mbuf或没有任何mbuf。如果m为空，则soreceive完成处理，控制跳到while循环的底部。如果m不空，所有剩余的mbuf链(nextrecord)都将重新连接到m，并将下一个mbuf的类型赋给type。如果下一个mbuf包含OOB数据，则设置flags中的MSG_OOB标志，并在最后返回给进程。因为 TCP不支持MT_OOBDATA形式的带外数据，所以MSG_OOB不会返回给TCP插口上的读调用。

图16-47显示了传送mbuf循环的第一部分。图16-46列出了循环中更新的变量。

变 量	描 述
moff	当MSG_PEEK被置位时，将被传送的下一个字节的偏移位置
offset	当MSG_PEEK被置位时，OOB标记的偏移位置
uio_resid	还未传送的字节数
len	从本mbuf中将要传送的字节数；如果uio_resid比较小或靠OOB标记比较近，则len可能小于m_len。

图16-46 soreceive 函数：循环内的变量

uipc_socket.c

```
598     moff = 0;
599     offset = 0;
600     while (m && uio->uio_resid > 0 && error == 0) {
601         if (m->m_type == MT_OOBDATA) {
602             if (type != MT_OOBDATA)
603                 break;
604         } else if (type == MT_OOBDATA)
605             break;
606         so->so_state &= ~SS_RCVATMARK;
607         len = uio->uio_resid;
608         if (so->so_oobmark && len > so->so_oobmark - offset)
609             len = so->so_oobmark - offset;
610         if (len > m->m_len - moff)
611             len = m->m_len - moff;
612         /*
613          * If mp is set, just pass back the mbufs.
614          * Otherwise copy them out via the uio, then free.
615          * Sockbuf must be consistent here (points to current mbuf,
616          * it points to next record) when we drop priority;
617          * we must note any additions to the sockbuf when we
618          * block interrupts again.
619          */
620         if (mp == 0) {
621             splx(s);
622             error = uiomove(mtod(m, caddr_t) + moff, (int) len, uio);
623             s = splnet();
624         } else
625             uio->uio_resid -= len;
```

uipc_socket.c

图16-47 soreceive 函数：uiomove

598-600 while循环的每一次循环中，一个 mbuf中的数据被传送到输出链或 uio缓存中。一旦链中没有mbuf或进程的缓存已满或出现差错，就退出循环。

14. 检查OOB和正常数据之前的变换

600-605 如果在处理mbuf链的过程中，mbuf的类型发生变化，则立即停止传送，以确保正常数据和带外数据不会混合在一个返回的报文中。但是，这种检查不适用于 TCP。

15. 更新OOB标记

606-611 计算当前字节到oobmark之间的长度来限制传送的大小，所以oobmark的前一个字节为传送的最后一个字节。传送的大小同时还要受 mbuf大小的限制。这段代码同样适用于TCP。

612-625 如果将数据传送到uio缓存，则调用uiomove。如果数据是作为一个mbuf链返回的，则更新uio_resid的值，使其等于传送的字节数。

为了避免在传送数据过程中协议处理挂起的时间太长，在调用 uiomove过程中使能协议处理。所以，在uiomove运行的过程中，接收缓存中可能会出现新的数据。

图16-48中描述的代码说明调整指针和偏移准备传送下一个 mbuf。

uipc_socket.c

```
626         if (len == m->m_len - moff) {
627             if (m->m_flags & M_EOR)
628                 flags |= MSG_EOR;
629             if (flags & MSG_PEEK) {
630                 m = m->m_next;
631                 moff = 0;
632             } else {
633                 nextrecord = m->m_nextpkt;
634                 sbfree(&so->so_rcv, m);
635                 if (mp) {
636                     *mp = m;
637                     mp = &m->m_next;
638                     so->so_rcv.sb_mb = m = m->m_next;
639                     *mp = (struct mbuf *) 0;
640                 } else {
641                     MFREE(m, so->so_rcv.sb_mb);
642                     m = so->so_rcv.sb_mb;
643                 }
644                 if (m)
645                     m->m_nextpkt = nextrecord;
646             }
647         } else {
648             if (flags & MSG_PEEK)
649                 moff += len;
650             else {
651                 if (mp)
652                     *mp = m_copym(m, 0, len, M_WAIT);
653                 m->m_data += len;
654                 m->m_len -= len;
655                 so->so_rcv.sb_cc -= len;
656             }
657         }
```

uipc_socket.c

图16-48 soreceive 函数：更新缓存

16. mbuf处理完毕了吗

626-646 如果mbuf中的所有字节都已传送完毕，则必须丢弃 mbuf或将指针向前移。如果mbuf中包含了一个逻辑记录的结尾，还应设置 MSG_EOR。如果将 MSG_PEEK置位，则so_receive跳到下一个缓存。在没有将 MSG_PEEK置位的情况下，如果数据已通过uiomove复制完成，则丢弃这块缓存；或者如果数据是作为一个 mbuf链返回，则将缓存添加到mp中。

图16-49包含处理OOB偏移和MSG_EOR的代码段。

uipc_socket.c

```
658         if (so->so_oobmark) {
659             if ((flags & MSG_PEEK) == 0) {
660                 so->so_oobmark -= len;
661                 if (so->so_oobmark == 0) {
662                     so->so_state |= SS_RCVATMARK;
663                     break;
664                 }
665             } else {
666                 offset += len;
667                 if (offset == so->so_oobmark)
668                     break;
669             }
670         }
671         if (flags & MSG_EOR)
672             break;
```

uipc_socket.c

图16-49 soreceive 函数：带外数据标记

17. 更新OOB标记

658-670 如果带外数据标志等于非0，则将其减去已传送的字节数。如果已到达标记处，则将SS_RCVATMARK置位，soreceive跳出while循环。如果没有将MSG_PEEK置位，则更新offset，而不是so_oobmark。

18. 逻辑记录结束

671-672 如果已到达一个逻辑记录的结尾，则 soreceive跳出mbuf处理循环，因而不会将下一个逻辑记录也作为这个报文的一部分返回。

在图16-50中，当设置了MSG_WAITALL标志 ，并且读请求还没有完成，则循环将等待更多的数据到达。

uipc_socket.c

```
673         /*
674          * If the MSG_WAITALL flag is set (for non-atomic socket),
675          * we must not quit until "uio->uio_resid == 0" or an error
676          * termination.  If a signal/timeout occurs, return
677          * with a short count but without error.
678          * Keep sockbuf locked against other readers.
679          */
680         while (flags & MSG_WAITALL && m == 0 && uio->uio_resid > 0 &&
681                !sosendallatonce(so) && !nextrecord) {
682             if (so->so_error || so->so_state & SS_CANTRCVMORE)
683                 break;
684             error = sbwait(&so->so_rcv);
685             if (error) {
686                 sbunlock(&so->so_rcv);
687                 splx(s);
688                 return (0);
689             }
690             if (m = so->so_rcv.sb_mb)
691                 nextrecord = m->m_nextpkt;
692         }
693     }                           /* while more data and more space to fill */
```

uipc_socket.c

图16-50 soreceive 函数：MSG_WAITALL 处理

19. MSG_WAITALL

673-681 如果将MSG_WAITALL置位，而缓存中没有数据(m等于0)，调用者需要更多的数据，sosendallatonce为假，并且这是接收缓存中的最后一个记录（nextrecord为空），则soreceive必须等待新的数据。

20. 差错或没有数据到达

682-683 如果差错出现或连接被关闭，则退出循环。

21. 等待数据到达

684-689 当接收缓存被协议层改变时sbwait返回。如果sbwait是被信号中断(error非0)，则soreceive立即返回。

22. 用接收缓存同步m和nextrecord

690-692 更新m和nextrecord，因为接收缓存被协议层修改了。如果数据到达mbuf，则m等于非0，while循环结束。

23. 处理下一个mbuf

693 本行是mbuf处理循环的结尾。控制返回到循环开始的第600行(图16-47)。一旦接收缓存中有数据，有新的缓存空间，没有差错出现，则循环继续。

如果soreceive停止复制数据，则执行图16-51所示的代码段。

uipc_socket.c

```
694     if (m && pr->pr_flags & PR_ATOMIC) {
695         flags |= MSG_TRUNC;
696         if ((flags & MSG_PEEK) == 0)
697             (void) sbdroprecord(&so->so_rcv);
698     }
699     if ((flags & MSG_PEEK) == 0) {
700         if (m == 0)
701             so->so_rcv.sb_mb = nextrecord;
702         if (pr->pr_flags & PR_WANTRCVD && so->so_pcb)
703             (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
704                               (struct mbuf *) flags, (struct mbuf *) 0,
705                               (struct mbuf *) 0);
706     }
707     if (orig_resid == uio->uio_resid && orig_resid &&
708         (flags & MSG_EOR) == 0 && (so->so_state & SS_CANTRCVMORE) == 0) {
709         sbunlock(&so->so_rcv);
710         splx(s);
711         goto restart;
712     }
713     if (flagsp)
714         *flagsp |= flags;
```

uipc_socket.c

图16-51 soreceive 函数：退出处理

24. 被截断的报文

694-698 如果因为进程的接收缓存太小而收到一个被截断的报文（数据报或记录），则插口层将这种情况通过设置MSG_TRUNC来通知进程，报文的被截断部分被丢弃。同其他接收标志一样，进程只有通过recvmsg系统调用才能获得MSG_TRUNC，即使soreceive总是设置这个标志。

25. 记录结尾的处理

699-706 如果没有将MSG_PEEK置位，则下一个mbuf链将被连接到接收缓存，并且如果发送了PRU_RCVD协议请求，则通知协议接收操作已经完成。 TCP通过这种机制来完成对连接

接收窗口的更新。

26. 没有传送数据

707-712 如果soreceive运行完成，没有传送任何数据，没有到达记录的结尾，且连接的读通道是活动的，则将接收缓存解锁，soreceive跳回到restart继续等待数据。

713-714 soreceive中设置的任何标志都在*flagsp中返回，缓存被解锁，soreceive返回。

讨论

soreceive是一个复杂的函数。导致其复杂性的主要原因是繁锁的指针操作及对多种类型的数据(带外数据、地址、控制信息和正常数据)和多目标(进程缓存，mbuf链)的处理。

同sosend类似，soreceive的复杂性是多年积累的结果。为每一种协议编写一个特殊的接收函数将会模糊插口层和协议层之间的边界，但是可以大大简化代码。

[Partridge and Pink 1993]描述了一个专门为UDP编写的soreceive函数，其功能是将数据报从接收缓存复制到进程缓存中时给数据报求检验和。他们给出的结论是：修改通用的soreceive函数来支持这一功能将“使本来已经很复杂的插口子程序变得更加复杂。”

16.13 select系统调用

在下面的讨论中，我们假定读者熟悉 select调用的基本操作和含义。关于 select的应用接口的详细描述参考[Stevens 1992]。

图16-52列出了select能够监控的插口状态。

描 述	select监控的操作		
	读	写	例外
有数据可读	•		
连接的读通道被关闭	•		
listen插口已经将连接排队	•		
插口差错未处理	•		
缓存可供写操作用，且一个连接存在或还没有连接请求		•	
连接的写通道被关闭		•	
插口差错未处理		•	
OOB同步标记未处理			•

图16-52 select 系统调用：插口事件

我们从select系统调用的第一部分开始讨论，如图 16-53所示。

1. 验证和初始化

390-410 在堆栈中分配两个数组：ibits和obits，每个数组有三个单元，每个单元为一个描述符集合。用bzero将它们清0。第一个参数，nd，必须不大于进程的描述符的最大数量。如果nd大于当前分配给进程的描述符个数，将其减少到当前分配给进程的描述符的个数。ni等于用来存放nd个比特(1个描述符占1个比特)的比特掩码所需的字节数。例如，假设最多有256个描述符(FD_SETSIZE)，falset表示一个32 bit的整型(NFDBITS)数组，且nd等于65，那么：

$$ni=howmany(65,32)\times 4=3\times 4=12$$

在上面的公式中，howmany (x，y)返回存储x比特所需要的长度为y比特的对象的数量。

sys_generic.c

```
390 struct select_args {
391     u_int   nd;
392     fd_set *in, *ou, *ex;
393     struct timeval *tv;
394 };

395 select(p, uap, retval)
396 struct proc *p;
397 struct select_args *uap;
398 int    *retval;
399 {
400     fd_set  ibits[3], obits[3];
401     struct timeval atv;
402     int     s, ncoll, error = 0, timo;
403     u_int   ni;

404     bzero((caddr_t) ibits, sizeof(ibits));
405     bzero((caddr_t) obits, sizeof(obits));
406     if (uap->nd > FD_SETSIZE)
407         return (EINVAL);
408     if (uap->nd > p->p_fd->fd_nfiles)
409         uap->nd = p->p_fd->fd_nfiles;   /* forgiving; slightly wrong */
410     ni = howmany(uap->nd, NFDBITS) * sizeof(fd_mask);

411 #define getbits(name, x) \
412     if (uap->name && \
413         (error = copyin((caddr_t)uap->name, (caddr_t)&ibits[x], ni))) \
414         goto done;
415     getbits(in, 0);
416     getbits(ou, 1);
417     getbits(ex, 2);
418 #undef  getbits

419     if (uap->tv) {
420         error = copyin((caddr_t) uap->tv, (caddr_t) & atv,
421                        sizeof(atv));
422         if (error)
423             goto done;
424         if (itimerfix(&atv)) {
425             error = EINVAL;
426             goto done;
427         }
428         s = splclock();
429         timevaladd(&atv, (struct timeval *) &time);
430         timo = hzto(&atv);
431         /*
432          * Avoid inadvertently sleeping forever.
433          */
434         if (timo == 0)
435             timo = 1;
436         splx(s);
437     } else
438         timo = 0;
```

sys_generic.c

图16-53 Select 函数：初始化

2. 从进程复制文件描述符集

411-418 getbits宏用copyin从进程那里将文件描述符集合传送到 ibits中的三个描述符集合。如果描述符集合指针为空，则不需复制。

3. 设置超时值

419-438 如果tv为空，则将timeo置成0，select将无限期等待。如果tv非空，则将超时值复制到内核，并调用itimerfix将超时值按硬件时钟的分辨率取整。调用timevaladd将当前时间加到超时值中。调用hzto计算从启动到超时之间的时钟滴答数，并保存在timo中。如果计算出来的结果为0，将timeo置1，从而防止select阻塞，实现利用全0的timeval结构来实现非阻塞操作。

select的第二部分代码，如图16-54所示。其作用是扫描进程指示的文件描述符，当一个或多个描述符处于就绪状态或定时器超时或信号出现时返回。

sys_generic.c

```
439   retry:
440     ncoll = nselcoll;
441     p->p_flag |= P_SELECT;
442     error = selscan(p, ibits, obits, uap->nd, retval);
443     if (error || *retval)
444         goto done;
445     s = splhigh();
446     /* this should be timercmp(&time, &atv, >=) */
447     if (uap->tv && (time.tv_sec > atv.tv_sec ||
448                 time.tv_sec == atv.tv_sec && time.tv_usec >= atv.tv_usec)) {
449         splx(s);
450         goto done;
451     }
452     if ((p->p_flag & P_SELECT) == 0 || nselcoll != ncoll) {
453         splx(s);
454         goto retry;
455     }
456     p->p_flag &= ~P_SELECT;
457     error = tsleep((caddr_t) & selwait, PSOCK | PCATCH, "select", timo);
458     splx(s);
459     if (error == 0)
460         goto retry;
461   done:
462     p->p_flag &= ~P_SELECT;
463     /* select is not restarted after signals... */
464     if (error == ERESTART)
465         error = EINTR;
466     if (error == EWOULDBLOCK)
467         error = 0;
468 #define putbits(name, x) \
469     if (uap->name && \
470         (error2 = copyout((caddr_t)&obits[x], (caddr_t)uap->name, ni))) \
471         error = error2;
472     if (error == 0) {
473         int     error2;
474         putbits(in, 0);
475         putbits(ou, 1);
476         putbits(ex, 2);
477 #undef putbits
478     }
479     return (error);
480 }
```

sys_generic.c

图16-54 select 函数：第二部分

4. 扫描文件描述符

439-442 从retry开始的循环直到select能够返回时退出。在调用进程的控制块中保存

全局整数nselcoll的当前值和P_SELECT标志。如果在selscan(图16-55)扫描文件描述符期间出现任何一种变化，则这种变化表明描述符的状态因为中断处理而发生改变，select必须重新扫描文件描述符。selscan查看三个输入的描述符集合中的每一个描述符集合，如果描述符处于就绪状态，则在输出的描述符集合中设置匹配的描述符。

sys_generic.c

```
481 selscan(p, ibits, obits, nfd, retval)
482 struct proc *p;
483 fd_set *ibits, *obits;
484 int     nfd, *retval;
485 {
486     struct filedesc *fdp = p->p_fd;
487     int     msk, i, j, fd;
488     fd_mask bits;
489     struct file *fp;
490     int     n = 0;
491     static int flag[3] =
492     {FREAD, FWRITE, 0};

493     for (msk = 0; msk < 3; msk++) {
494         for (i = 0; i < nfd; i += NFDBITS) {
495             bits = ibits[msk].fds_bits[i / NFDBITS];
496             while ((j = ffs(bits)) && (fd = i + --j) < nfd) {
497                 bits &= ~(1 << j);
498                 fp = fdp->fd_ofiles[fd];
499                 if (fp == NULL)
500                     return (EBADF);
501                 if ((*fp->f_ops->fo_select) (fp, flag[msk], p)) {
502                     FD_SET(fd, &obits[msk]);
503                     n++;
504                 }
505             }
506         }
507     }
508     *retval = n;
509     return (0);
510 }
```

sys_generic.c

图16-55 selscan 函数

5. 差错或一些描述符准备就绪

443-444 如果差错出现或描述符处于就绪状态，就立即返回。

6. 超时了吗

445-451 如果进程提供的时间限制和当前时间已经超过了超时值，则立即返回。

7. 在执行selscan期间状态发生变化

452-455 selscan可以被协议处理中断。如果在中断期间插口状态改变，则将P_SELECT和nselcoll置位，且selscan必须重新扫描所有描述符。

8. 等待缓存发生变化

456-460 所有调用select的进程均在调用tsleep时用selwait作为等待通道。如图16-60所示，这种做法在多个进程等待同一个插口缓存的情况下将导致效率降低。如果tsleep正确返回，则select跳转到retry，重新扫描所有描述符。

9. 准备返回

461-480 在done处清除P_SELECT，如果差错代码为ERESTART，则修改为EINTR；如果差错代码为EWOULDBLOCK，则将差错代码置成0。这些改变确保在select调用期间若信号出现时能返回EINTR；若超时，则返回0。

16.13.1 selscan函数

select函数的核心是图16-55所示的selscan函数。对于任意一个描述符集合中设置的每一个比特，selscan找出同它相关联的描述符，并且将控制分散给与描述符相关联的so_select函数。对于插口而言，就是soo_select函数。

1. 定位被监视的描述符

481-496 第一个for循环依次查看三个描述符集合：读，写和例外。第二个for循环在每个描述符集合内部循环，这个循环在集合中每隔32 bit(NFDBITS)循环一次。

最里面的while循环检查所有被32 bit的掩码标记的描述符，该掩码从当前描述符集合中获取并保存在bits中。函数ffs返回bits中的第一个被设置的比特的位置，从最低位开始。例如，如果bits等于1000 (省略了前面的28个0)，则ffs (bits)等于4。

2. 轮询描述符

497-500 从i到ffs函数的返回值，计算与比特相关的描述符，并保存在fd中。在bits中(而不是在输入描述符集合中)清除比特，找到与描述符相对应的file结构，调用fo_select。

fo_select的第二个参数是flag数组中的一个元素。msk是外层的for循环的循环变量。所以，第一次循环时，第二个参数等于FREAD，第二次循环时等于FWRITE，第三次循环时等于0。如果描述符不正确，则返回EBADF。

3. 描述符准备就绪

501-504 当发现某个描述符的状态为准备就绪时，设置输出描述符集合中相对应的比特位。并将n (状态就绪的描述符的个数)加1。

505-510 循环继续直到轮询完所有描述符。状态就绪的描述符的个数通过*retval返回。

16.13.2 soo_select函数

对于selscan在输入描述符集合中发现的每一个状态就绪的描述符，selscan调用与描述符相关的fileops结构(参考第15.5节)中的fo_select指针引用的函数。在本书中，我们只对插口描述符和图16-56所示的soo_select函数感兴趣。

sys_socket.c

```
105 soo_select(fp, which, p)
106 struct file *fp;
107 int     which;
108 struct proc *p;
109 {
110     struct socket *so = (struct socket *) fp->f_data;
111     int     s = splnet();
112     switch (which) {
113     case FREAD:
114         if (soreadable(so)) {
```

图16-56 soo_select 函数

```
115             splx(s);
116             return (1);
117         }
118         selrecord(p, &so->so_rcv.sb_sel);
119         so->so_rcv.sb_flags |= SB_SEL;
120         break;
121     case FWRITE:
122         if (sowriteable(so)) {
123             splx(s);
124             return (1);
125         }
126         selrecord(p, &so->so_snd.sb_sel);
127         so->so_snd.sb_flags |= SB_SEL;
128         break;
129     case 0:
130         if (so->so_oobmark || (so->so_state & SS_RCVATMARK)) {
131             splx(s);
132             return (1);
133         }
134         selrecord(p, &so->so_rcv.sb_sel);
135         so->so_rcv.sb_flags |= SB_SEL;
136         break;
137     }
138     splx(s);
139     return (0);
140 }
```

sys_socket.c

图16-56 (续)

105-112 soo_select每次被调用时，它只检查一个描述符的状态。如果相对于which中指定的条件，描述符处于就绪状态，则立即返回1。如果描述符没有处于就绪状态，就用selrecord标记插口的接收缓存或发送缓存，指示进程正在选择该缓存，然后soo_select返回0。

图16-52显示了插口的读、写和例外情况。我们将看到soo_select使用了soreadable和sowriteable宏，这些宏在sys/socketvar.h中定义。

1. 插口可读吗

113-120 soreadable宏的定义如下：

```
#define soreadable(so) \
    ((so)->so_rcv.sb_cc >= (so)->so_rcv.sb_lowat || \
    ((so)->so_state & SS_CANTRCVMORE) || \
    (so)->so_qlen || (so)->so_error)
```

因为UDP和TCP的接收下限默认值为1 (图16-4)，下列情况表示插口是可读的：接收缓存中有数据，连接的读通道被关闭，可以接受任何连接或有挂起的差错。

2. 插口可写吗

121-128 sowriteable宏的定义如下：

```
#define sowriteable(so) \
    (sbspace(&(so)->so_snd) >= (so)->so_snd.sb_lowat && \
    (((so)->so_state&SS_ISCONNECTED) || \
    ((so)->so_proto->pr_flags&PR_CONNREQUIRED)==0) || \
    ((so)->so_state & SS_CANTSENDMORE) || \
    (so)->so_error)
```

TCP和UDP默认的发送低水位标记是2048。对于UDP而言，sowriteable总是为真，因

为sbspace总是等于sb_hiwat，当然也总是大于或等于so_lowat，且不要求连接。

对于TCP而言，当发送缓存中的可用空间小于 2048个字节时，插口不可写。其他的情况在图16-52中讨论。

3. 还有挂起的例外情况吗

129-140 对于例外情况，需检查标志so_oobmark和SS_RECVATMARK。直到进程读完数据流中的同步标记后，例外情况才可能存在。

16.13.3 selrecord函数

图16-57显示同发送和接收缓存存储在一起的selinfo结构的定义(图16-3中的sb_sel成员)。

select.h

```
41 struct selinfo {
42     pid_t   si_pid;             /* process to be notified */
43     short   si_flags;           /* 0 or SI_COLL */
44 };
```

select.h

图16-57 selinfo 结构

41-44 当只有一个进程对某一给定的插口缓存调用 select时，sl_pid等于等待进程的进程标志符。当其他的进程对同一缓存调用 select时，设置sl_flags中的SI_COLL标志。将这种情况称为冲突。这个标志是目前 sl_flags中唯一已定义的标志。

当soo_select发现描述符不在就绪状态时就调用 selrecord函数，如图 16-58所示。该函数记录了足够的信息，使得缓存内容发生变化时协议处理层能够唤醒进程。

sys_generic.c

```
522 void
523 selrecord(selector, sip)
524 struct proc *selector;
525 struct selinfo *sip;
526 {
527     struct proc *p;
528     pid_t   mypid;

529     mypid = selector->p_pid;
530     if (sip->si_pid == mypid)
531         return;
532     if (sip->si_pid && (p = pfind(sip->si_pid)) &&
533         p->p_wchan == (caddr_t) & selwait)
534         sip->si_flags |= SI_COLL;
535     else
536         sip->si_pid = mypid;
537 }
```

sys_generic.c

图16-58 selrecord 函数

1. 重复选择描述符

522-531 selrecord的第一个参数指向调用 select进程的proc结构。第二个参数指向selinfo记录，该记录的 so_snd.sb_sel和so_rcv.sb_sel可能会被修改。如果selinfo中已记录了该进程的信息，则立即返回。例如，进程对同一个描述符调用 select查询读和例外情况时，函数就立即返回。

2. 同另一个进程的操作冲突？

532-534 如果另一个进程已经对同一插口缓存执行select操作，则设置SI_COLL。

3. 没有冲突

535-537 如果调用没有发生冲突，则 si_pid等于0，将当前进程的进程标志符赋给si_pid。

16.13.4 selwakeup函数

当协议处理改变插口缓存的状态，并且只有一个进程选择了该缓存时，Net/3就能根据selinfo结构中记录的信息立即将该进程放入运行队列。

当插口缓存发生变化但是有多个进程选择同一插口缓存时（设置了SI_COLL），Net/3就无法确定哪些进程对这种缓存变化感兴趣。我们在讨论图16-54中的代码段时就已经指出，每一个调用select的进程在调用tsleep时使用selwait作为等待通道。这意味着对应的wakeup将唤醒所有阻塞在select上的进程——甚至是对缓存的变化不关心的进程。

图16-59说明如何调用selwakeup。

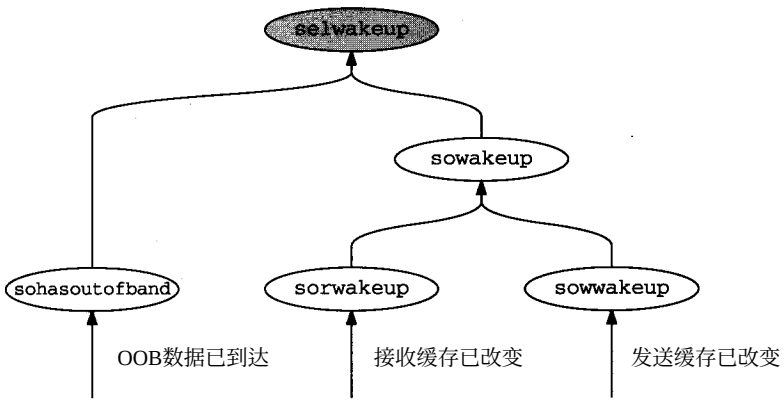

图16-59 selwakeup处理

当改变插口状态的事件出现时，协议处理层负责调用图16-59的底部列出的一个函数来通知插口层。图16-59底部显示的三个函数都将导致selwakeup被调用，在插口上选择的任何进程将被调度运行。

selwakeup函数如图16-60所示。

541-548 如果si_pid等于0，表明没有进程对该缓存执行select操作，函数立即返回。

在冲突中唤醒所有进程

549-553 如果多个进程对同一插口执行select操作，将nselcoll加1，清除冲突标志，唤醒所有阻塞在select上的进程。正如图16-54中讨论的，进程在tsleep中阻塞之前若缓存发生改变，nselcoll能使select重新扫描描述符(习题16.9)。

554-567 如果si_pid标识的进程正在selwait中等待，则调度该进程运行。如果进程是在其他等待通道中，则清除P_SELECT标志。如果对一个正确的描述符执行selrecord，则调用进程可能正在其他的等待通道中等待，然后，selscan在描述符集合中发现一个差错的文件描述符，并返回EBADF，不清除以前修改的selinfo记录。到selwakeup运行时，selwakup

可能会发现sel_pid标识的进程不再在插口缓存等待，从而忽略selinfo中的信息。

如果没有出现多个进程共享同一个描述符的情况(也就是同一块插口缓存)，当然这种情况很少，则只有一个进程被selwakeup唤醒。在作者使用的机器上，nselcoll总是等于0，这说明select冲突是很少发生的。

sys_generic.c

```
541 void
542 selwakeup(sip)
543 struct selinfo *sip;
544 {
545     struct proc *p;
546     int     s;

547     if (sip->si_pid == 0)
548         return;
549     if (sip->si_flags & SI_COLL) {
550         nselcoll++;
551         sip->si_flags &= ~SI_COLL;
552         wakeup((caddr_t) & selwait);
553     }
554     p = pfind(sip->si_pid);
555     sip->si_pid = 0;
556     if (p != NULL) {
557         s = splhigh();
558         if (p->p_wchan == (caddr_t) & selwait) {
559             if (p->p_stat == SSLEEP)
560                 setrunnable(p);
561             else
562                 unsleep(p);
563         } else if (p->p_flag & P_SELECT)
564             p->p_flag &= ~P_SELECT;
565         splx(s);
566     }
567 }
```

sys_generic.c

图16-60 selwakeup 函数

16.14 小结

本章介绍了插口的读、写和选择系统调用。

我们了解到sosend处理插口层与协议处理层之间的所有输出，而soreceive处理所有输入。

本章还介绍了发送缓存和接收缓存的组织结构，以及缓存的高、低水位标记的默认值和含义。

本章的最后一部分介绍了select系统调用。从这部分内容中我们了解到，当只有一个进程对描述符执行select调用时，协议处理层仅仅唤醒selinfo结构中标识的那个进程。当有多个进程对同一个描述符执行select操作而发生冲突时，协议层只能唤醒所有等待在该描述符上的进程。

习题

16.1 当将一个大于最大的正的有符号整数的无符号整数传给 write系统调用时，

sosend中的resid如何变化？

16.2 当sosend将小于MCLBYTES个字节的数据放入簇中时，space被减去MCLBYTES，可能会成为一个负数，这会导致为atomic协议填写mbuf的循环结束。这种结果是正常的吗？

16.3 数据报和流协议有着不同的语义。将sosend和soreceive函数分别分成两个函数，一个用来处理报文，另一个用来处理流。除了使得代码清晰外，这样做还有什么好处？

16.4 对于PR_ATOMIC协议，每一个写调用都指定了一个隐含的报文边界。插口层将这个报文作为一个整体交给协议。MSG_EOR标志允许进程显式指定报文边界。为什么仅有隐含的报文边界是不够的？

16.5 如果插口描述符没有标记为非阻塞，且进程也没有指定MSG_DONTWAIT，当sosend不能立即获取发送缓存上的锁时，结果如何？

16.6 在什么情况下，虽然sb_cc<sb_hiwat，但sbspace仍然报告没有闲置空间？为什么在这种情况下进程应该被阻塞？

16.7 为什么recvit不将控制报文的长度而是将名字长度返回给进程？

16.8 为什么soreceive要清除MSG_EOR？

16.9 如果将nselcoll代码从select和selwakeup中删除，会有什么问题？

16.10 修改select系统调用，使得select返回时返回定时器的剩余时间。