

## 第17章 插口选项

### 17.1 引言

本章讨论修改插口行为的几个系统调用，以此来结束插口层的介绍。

setsockopt和getsockopt系统调用已在第8.8节中介绍过，主要描述访问IP特点的选项。在本章中，我们将介绍这两个系统调用的实现以及通过它们来控制的插口级选项。

ioctl函数在第4.4节中已介绍过，在第4.4节中，我们描述了用于网络接口配置的与协议无关的ioctl命令。在第6.7节中，我们描述了用来分配网络掩码以及单播、多播和目的地址的IP专用的ioctl命令。本章我们将介绍ioctl的实现和fcntl函数的相关特点。

最后，我们介绍getsockname和getpeername系统调用，它们用来返回插口和连接的地址信息。

图17-1列出了实现插口选项系统调用的函数。本章描述带阴影的函数。

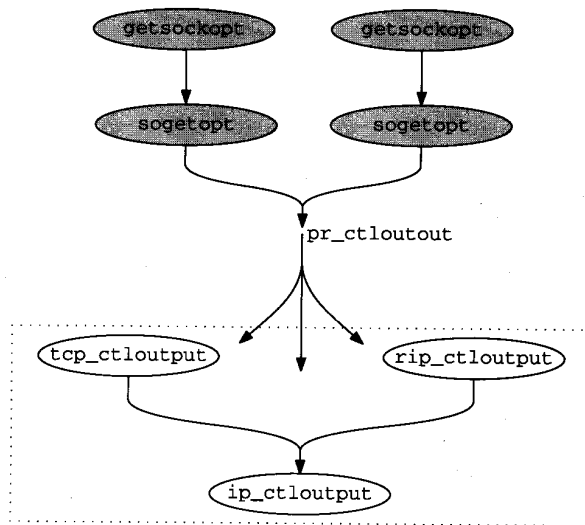


图17-1 setsockopt 和getsockopt 系统调用

### 17.2 代码介绍

本章中涉及的源代码来自于图17-2中列出的四个文件。

文 件 名	说 明
kern/kern_descrip.c	fcntl系统调用
kern/uipc_syscalls.c	setsockopt、getsockopt、getsockname和getpeername系统调用
kern/uipc_socket.c	插口层对setsockopt和getsockopt的处理
kern/sys_socket.c	ioctl系统调用对插口的处理

图17-2 本章讨论的源文件

全局变量和统计量

本章中描述的系统调用没有定义新的全局变量，也没有收集任何统计量。

17.3 setsockopt系统调用

图8-29列出了函数setsockopt (和getsockopt)能够访问的各种不同的协议层。本章主要集中在SOCKET级的选项，这些选项在图 17-3中列出。

optname	optval 类型	变 量	说 明
SO_SNDBUF	int	so_snd.sb_hiwat	发送缓存高水位标记
SO_RCVBUF	int	so_rcv.sb_hiwat	接收缓存高水位标记
SO_SNDBUF	int	so_snd.sb_lowat	发送缓存低水位标记
SO_RCVBUF	int	so_rcv.sb_lowat	接收缓存低水位标记
SO_SNDTIMEO	struct timeval	so_snd.sb_timeo	发送超时值
SO_RCVTIMEO	struct timeval	so_rcv.sb_timeo	接收超时值
SO_DEBUG	int	so_options	记录插口调试信息
SO_REUSEADDR	int	so_options	插口能重新使用一个本地地址
SO_REUSEPORT	int	so_options	插口能重新使用一个本地端口
SO_KEEPAIVE	int	so_options	协议查询空闲的连接
SO_DONTROUTE	int	so_options	旁路路由表
SO_BROADCAST	int	so_options	插口支持广播报文
SO_USELOOPBACK	int	so_options	仅用于选路域插口；发送进程接收自己的选路报文
SO_OOBINLINE	int	so_options	协议排队内联的带外数据
SO_LINGER	struct linger	so_linger	插口关闭但仍发送剩余数据
SO_ERROR	int	so_error	获取差错状态并清除；只用于getsockopt
SO_TYPE	int	so_type	获取插口类型；只用于getsockopt
其他			返回ENOPROTOOPT

图17-3 setsockopt 和getsockopt 选项

setsockopt函数原型为：

```
int setsockopt(int s, int level, int optname, void *optval, int optlen);
```

图17-4显示了setsockopt调用的源代码。

565-597 getsock返回插口描述符的file结构。如果val非空，则将valsize个字节的  
数据从进程复制到用m\_get分配的mbuf中。与选项对应的数据长度不能超过MLEN个字节，所以，如果valsize大于MLEN，则返回EINVAL。调用sosetopt，并返回其值。

```

565 struct setsockopt_args {
566     int      s;
567     int      level;
568     int      name;
569     caddr_t  val;
570     int      valsize;
571 };

572 setsockopt(p, uap, retval)
573 struct proc *p;
574 struct setsockopt_args *uap;
575 int      *retval;

```

uipc\_syscalls.c

图17-4 setsockopt 系统调用

```

576 {
577     struct file *fp;
578     struct mbuf *m = NULL;
579     int     error;

580     if (error = getsock(p->p_fd, uap->s, &fp))
581         return (error);
582     if (uap->valsize > MLEN)
583         return (EINVAL);
584     if (uap->val) {
585         m = m_get(M_WAIT, MT_SOOPTS);
586         if (m == NULL)
587             return (ENOBUFS);
588         if (error = copyin(uap->val, mtod(m, caddr_t),
589             (u_int) uap->valsize)) {
590             (void) m_free(m);
591             return (error);
592         }
593         m->m_len = uap->valsize;
594     }
595     return (so_setopt((struct socket *) fp->f_data, uap->level,
596         uap->name, m));
597 }

```

uipc\_syscalls.c

图17-4 (续)

### so\_setopt函数

so\_setopt函数处理所有插口级的选项，并将其他的选项传给与插口关联的协议的pr\_ctloutput函数。图17-5列出了so\_setopt函数的部分代码。

```

752 so_setopt(so, level, optname, m0)
753 struct socket *so;
754 int     level, optname;
755 struct mbuf *m0;
756 {
757     int     error = 0;
758     struct mbuf *m = m0;

759     if (level != SOL_SOCKET) {
760         if (so->so_proto && so->so_proto->pr_ctloutput)
761             return ((*so->so_proto->pr_ctloutput)
762                 (PRCO_SETOPT, so, level, optname, &m0));
763         error = ENOPROTOOPT;
764     } else {
765         switch (optname) {

```

/\* socket option processing \*/

```

841         default:
842             error = ENOPROTOOPT;
843             break;
844         }
845         if (error == 0 && so->so_proto && so->so_proto->pr_ctloutput) {
846             (void) ((*so->so_proto->pr_ctloutput)
847                 (PRCO_SETOPT, so, level, optname, &m0));

```

图17-5 so\_setopt 函数

```
848         m = NULL;                /* freed by protocol */
849     }
850 }
851 bad:
852     if (m)
853         (void) m_free(m);
854     return (error);
855 }
```

—uipc\_socket.c

图17-5 (续)

752-764 如果选项不是插口级的(SOL\_SOCKET)选项，则给底层协议发送PRCO\_SETOPT请求。注意：调用的是协议的pr\_ctloutput函数，而不是它的pr\_usrreq函数。图17-6说明了Internet协议调用的pr\_ctloutput函数。

协 议	pr_ctloutput函数	参 考
UDP	ip_ctloutput	第8.8节
TCP	tcp_ctloutput	第30.6节
ICMP IGMP 原始IP	rip_ctloutput和ip_ctloutput	第8.8节和第32.8节

图17-6 pr\_ctloutput 函数

765 switch语句处理插口级的选项。

841-844 对于不认识的选项，在保存它的mbuf被释放后返回ENOPROTOOPT。

845-855 如果没有出现差错，则控制总是会执行到switch。在switch语句中，如果协议层需要响应请求或插口层，则将选项传送给相应的协议。Internet协议中没有预期处理插口级的选项。

注意，如果协议收到不预期的选项，则直接将其pr\_ctloutput函数的返回值丢弃。并将m置空，以免调用m\_free，因为协议负责释放缓存。

图17-7说明了linger选项和在插口结构中设置单一标志的选项。

766-772 linger选项要求进程传入linger结构：

```
struct linger {
    int l_onoff; /* option on/off */
    int l_linger; /* linger time in seconds */
};
```

确保进程已传入长度为linger结构大小的数据后，将结构成员l\_linger复制到so\_linger中。在下一组case语句后决定是使能还是关闭该选项。so\_linger和close系统调用在第15.15节中已介绍过。

773-789 当进程传入一个非0值时，设置选项对应的布尔标志；当进程传入的是0时，将对应标志清除。第一次检查确保一个整数大小（或更大）的对象在mbuf中，然后设置或清除对应的选项。

图17-8显示了插口缓存选项的处理。

790-815 这组选项改变插口的发送和接收缓存的大小。第一个if语句确保提供给四个选项的变量是整型。对于SO\_SNDBUF和SO\_RCVBUF，sbreserve只调整缓存的高水位标记而不

分配缓存。对于SO\_SNDLOWAT和SO\_RCVLOWAT，调整缓存的低水位标记。

```

766         case SO_LINGER:
767             if (m == NULL || m->m_len != sizeof(struct linger)) {
768                 error = EINVAL;
769                 goto bad;
770             }
771             so->so_linger = mtod(m, struct linger *)->l_linger;
772             /* fall thru... */

773         case SO_DEBUG:
774         case SO_KEEPAVAIL:
775         case SO_DONTROUTE:
776         case SO_USELOOPBACK:
777         case SO_BROADCAST:
778         case SO_REUSEADDR:
779         case SO_REUSEPORT:
780         case SO_OOBINLINE:
781             if (m == NULL || m->m_len < sizeof(int)) {
782                 error = EINVAL;
783                 goto bad;
784             }
785             if (*mtod(m, int *))
786                 so->so_options |= optname;
787             else
788                 so->so_options &= ~optname;
789             break;

```

uipc\_socket.c

图17-7 setsockopt 函数：linger 和标志选项

```

790         case SO_SNDBUF:
791         case SO_RCVBUF:
792         case SO_SNDLOWAT:
793         case SO_RCVLOWAT:
794             if (m == NULL || m->m_len < sizeof(int)) {
795                 error = EINVAL;
796                 goto bad;
797             }
798             switch (optname) {
799                 case SO_SNDBUF:
800                 case SO_RCVBUF:
801                     if (sbreserve(optname == SO_SNDBUF ?
802                                 &so->so_snd : &so->so_rcv,
803                                 (u_long) * mtod(m, int *)) == 0) {
804                         error = ENOBUFS;
805                         goto bad;
806                     }
807                     break;
808                 case SO_SNDLOWAT:
809                     so->so_snd.sb_lowat = *mtod(m, int *);
810                     break;
811                 case SO_RCVLOWAT:
812                     so->so_rcv.sb_lowat = *mtod(m, int *);
813                     break;
814             }
815             break;

```

uipc\_socket.c

图17-8 setsockopt 函数：插口缓存选项

图17-9说明超时选项。

```

816         case SO_SNDTIMEO:
817         case SO_RCVTIMEO:
818             {
819                 struct timeval *tv;
820                 short   val;

821                 if (m == NULL || m->m_len < sizeof(*tv)) {
822                     error = EINVAL;
823                     goto bad;
824                 }
825                 tv = mtod(m, struct timeval *);
826                 if (tv->tv_sec > SHRT_MAX / hz - hz) {
827                     error = EDOM;
828                     goto bad;
829                 }
830                 val = tv->tv_sec * hz + tv->tv_usec / tick;

831                 switch (optname) {
832                     case SO_SNDTIMEO:
833                         so->so_snd.sb_timeo = val;
834                         break;
835                     case SO_RCVTIMEO:
836                         so->so_rcv.sb_timeo = val;
837                         break;
838                 }
839                 break;
840             }

```

uipc\_socket.c

uipc\_socket.c

图17-9 ssetopt 函数：超时选项

816-824 进程在timeval结构中设置SO\_SNDTIMEO和SO\_RCVTIMEO选项的超时值。如果传入的数值不正确，则返回EINVAL。

825-830 存储在timeval结构中的时间间隔值不能太大，因为sb\_timeo是一个短整数，当时间间隔值的单位为一个时钟滴答时，时间间隔值的大小就不能超过一个短整数的最大值。

第826行代码是不正确的。在下列条件下，时间间隔不能表示为一个短整数：

$$tv\_sec \times hz + \frac{tv\_usec}{tick} > SHRT\_MAX$$

其中，tick=1 000 000和SHRT\_MAX=32767

所以，如果下列不等式成立，则返回。

$$tv\_sec > \frac{SHRT\_MAX}{hz} - \frac{tv\_usec}{tick \times hz} = \frac{SHRT\_MAX}{hz} - \frac{tv\_usec}{100000}$$

等式的最后一项不是代码指明的hz。正确的测试代码应该是：

```

if (tv->tv_sec * hz + tv->tv_usec / tick > SHRT_MAX)
    error = EDOM;

```

习题17.3中有更详细的讨论。

831-840 将转换后的时间，val，保存在请求的发送或接收缓存中。sb\_timeo限制了进程等待接收缓存中的数据或发送缓存中的闲置空间的时间。详细讨论参考第16.7和16.11节。

超时值是传给tsleep的最后一个参数，因为tsleep要求超时值为一个整数，所以进程最多只能等待65535个时钟滴答。假设时钟频率为100 Hz，则等待时间小于11分钟。

## 17.4 getsockopt系统调用

getsockopt返回进程请求的插口和协议选项。函数原型是：

```
int getsockopt(int s, int level, int name, caddr_t val, int *valsize);
```

该调用的源代码如图17-10所示。

—uipc\_syscalls.c

```
598 struct getsockopt_args {
599     int      s;
600     int      level;
601     int      name;
602     caddr_t  val;
603     int      *avalsize;
604 };

605 getsockopt(p, uap, retval)
606 struct proc *p;
607 struct getsockopt_args *uap;
608 int      *retval;
609 {
610     struct file *fp;
611     struct mbuf *m = NULL;
612     int      valsize, error;

613     if (error = getsock(p->p_fd, uap->s, &fp))
614         return (error);
615     if (uap->val) {
616         if (error = copyin((caddr_t) uap->avalsize, (caddr_t) & valsize,
617                             sizeof(valsize)))
618             return (error);
619     } else
620         valsize = 0;
621     if ((error = sogetopt((struct socket *) fp->f_data, uap->level,
622                          uap->name, &m)) == 0 && uap->val && valsize && m != NULL) {
623         if (valsize > m->m_len)
624             valsize = m->m_len;
625         error = copyout(mtod(m, caddr_t), uap->val, (u_int) valsize);
626         if (error == 0)
627             error = copyout((caddr_t) & valsize,
628                             (caddr_t) uap->avalsize, sizeof(valsize));
629     }
630     if (m != NULL)
631         (void) m_free(m);
632     return (error);
633 }
```

—uipc\_syscalls.c

图17-10 getsockopt 系统调用

598-633 这段代码现在看上去应该很熟悉了。getsock获取插口的file结构，将选项缓存的大小复制到内核，调用sogetopt来获取选项的值。将sogetopt返回的数据复制到进程提供的缓存，可能还需修改缓存长度。如果进程提供的缓存不够大，则返回的数据可能会被截掉。通常情况下，存储选项数据的mbuf在函数返回后被释放。

### sogetopt函数

同sosetopt一样，sogetopt函数处理所有插口级的选项，并将其他的选项传给与插口关联的协议。图17-11列出了sogetopt函数的开始和结束部分的代码。

uipc\_socket.c

```

856 sogetopt(so, level, optname, mp)
857 struct socket *so;
858 int level, optname;
859 struct mbuf **mp;
860 {
861     struct mbuf *m;

862     if (level != SOL_SOCKET) {
863         if (so->so_proto && so->so_proto->pr_ctloutput) {
864             return ((*so->so_proto->pr_ctloutput)
865                 (PRCO_GETOPT, so, level, optname, mp));
866         } else
867             return (ENOPROTOOPT);
868     } else {
869         m = m_get(M_WAIT, MT_SOOPTS);
870         m->m_len = sizeof(int);

871         switch (optname) {



```

/* socket option processing */

```


872         default:
873             (void) m_free(m);
874             return (ENOPROTOOPT);
875         }
876         *mp = m;
877         return (0);
878     }
879 }

```

uipc\_socket.c

图17-11 sogetopt 函数：概述

856-871 同so\_setopt一样，函数将那些与插口级选项无关的选项立即通过PRCO\_GETOPT协议请求传递给相应的协议级。协议将被请求的选项保存在mp指向的mbuf中。

对于插口级的选项，分配一块标准的mbuf缓存来保存选项值，选项值通常是一个整数，所以将m\_len设成整数大小。相应的选项值通过switch语句复制到mbuf中。

918-925 如果执行的是switch中的default情况下的语句，则释放mbuf，并返回ENOPROTOOPT。否则，switch语句执行完成后，将指向mbuf的指针赋给\*mp。当函数返回后，getsockopt从该mbuf中将数据复制到进程提供的缓存，并释放mbuf。

图17-12说明对SO\_LINGER选项和作为布尔型标志实现的选项的处理。

872-877 SO\_LINGER选项请求返回两个值：一个是标志值，赋给l\_onoff；另一个是拖延时间，赋给l\_linger。

878-887 其余的选项作为布尔标志实现。将so\_options和optname执行逻辑与操作，如果选项被打开，则与操作的结果为非0值；反之则结果为0。注意：标志被打开并不表示返回值等于1。

sogetopt的下一部分代码(图17-13)将整型值选项的值复制到mbuf中。

888-906 将每一个选项作为一个整数复制到mbuf中。注意：有些选项在内核中是作为一个短整数存储的(如缓存高低水位标记)，但是作为整数返回。一旦将so\_error复制到mbuf中后，即清除so\_error，这是唯一的一次getsockopt调用修改插口状态。



```

872         case SO_LINGER:
873             m->m_len = sizeof(struct linger);
874             mtod(m, struct linger *)->l_onoff =
875                 so->so_options & SO_LINGER;
876             mtod(m, struct linger *)->l_linger = so->so_linger;
877             break;

878         case SO_USELOOPBACK:
879         case SO_DONTROUTE:
880         case SO_DEBUG:
881         case SO_KEEPAVAIL:
882         case SO_REUSEADDR:
883         case SO_REUSEPORT:
884         case SO_BROADCAST:
885         case SO_OOBINLINE:
886             *mtod(m, int *) = so->so_options & optname;
887             break;

```

uipc\_socket.c

图17-12 `sogetopt` 选项：SO\_LINGER 选项和布尔选项

```

888         case SO_TYPE:
889             *mtod(m, int *) = so->so_type;
890             break;

891         case SO_ERROR:
892             *mtod(m, int *) = so->so_error;
893             so->so_error = 0;
894             break;

895         case SO_SNDBUF:
896             *mtod(m, int *) = so->so_snd.sb_hiwat;
897             break;

898         case SO_RCVBUF:
899             *mtod(m, int *) = so->so_rcv.sb_hiwat;
900             break;
901         case SO_SNDLOWAT:
902             *mtod(m, int *) = so->so_snd.sb_lowat;
903             break;

904         case SO_RCVLOWAT:
905             *mtod(m, int *) = so->so_rcv.sb_lowat;
906             break;

```

uipc\_socket.c

图17-13 `sogetopt` 函数：整型值选项

```

907         case SO_SNDTIMEO:
908         case SO_RCVTIMEO:
909             {
910                 int    val = (optname == SO_SNDTIMEO ?
911                     so->so_snd.sb_timeo : so->so_rcv.sb_timeo);
912
913                 m->m_len = sizeof(struct timeval);
914                 mtod(m, struct timeval *)->tv_sec = val / hz;
915                 mtod(m, struct timeval *)->tv_usec =
916                     (val % hz) / tick;
917                 break;
918             }

```

uipc\_socket.c

图17-14 `sogetopt` 函数：超时选项

图17-14列出了`sogetopt`的第三和第四部分代码，它们的作用分别是处理 `SO_SNDTIMEO` 和 `SO_RCVTIMEO` 选项。

907-917 将发送或接收缓存中的 `sb_timeo` 值赋给 `var`。基于 `val` 中的时钟滴答数，在 `mbuf` 中构造一个 `timeval` 结构。

计算 `tv_usec` 的代码有一个差错。表达式应该为：`“(val % hz) * tick”`。

## 17.5 `fcntl` 和 `ioctl` 系统调用

因为历史的原因而非有意这么做，插口 API 的几个特点既能通过 `ioctl` 也能通过 `fcntl` 来访问。关于 `ioctl` 命令，我们已经讨论了很多。我们也几次提到 `fcntl`。

图17-15显示了本章描述的函数。

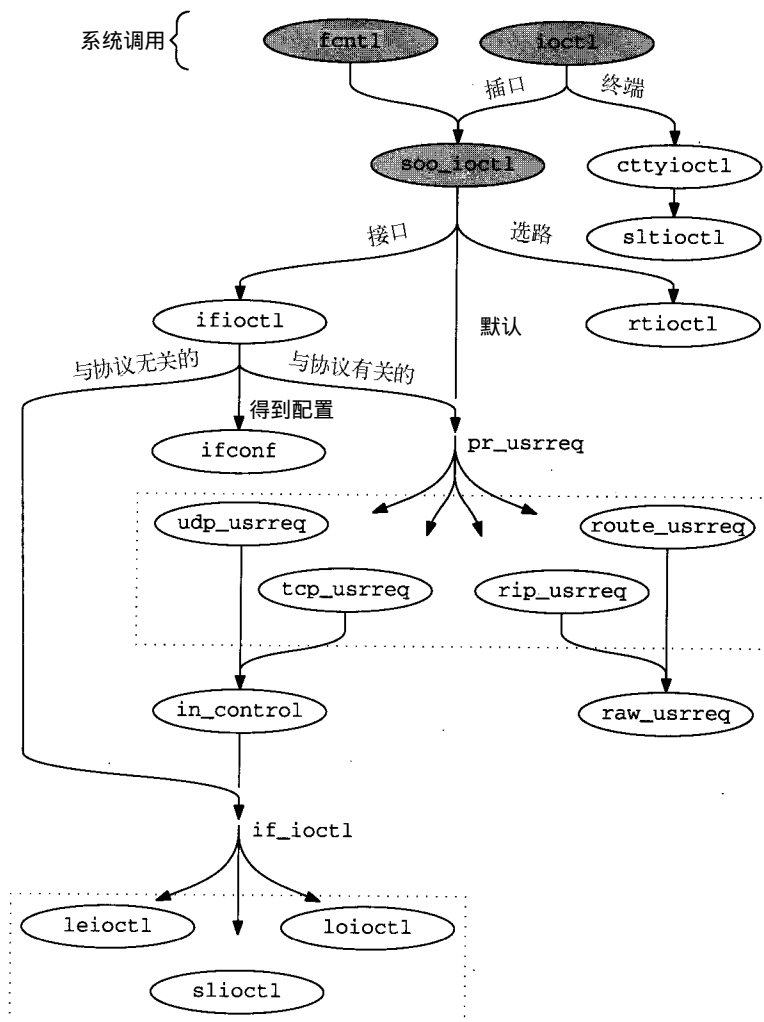


图17-15 `fcntl` 和 `ioctl` 函数

`ioctl` 和 `fcntl` 的原型分别为：

```
int ioctl(int fd, unsigned long result, char *argp);
int fcntl(int fd, int cmd, ... /* int arg */);
```

图17-16总结了这两个系统调用与插口有关的特点。我们在图 17-16中还列出了一些传统的常数，因为它们出现在代码中。考虑与 Posix的兼容性，可以用 O\_NONBLOCK来代替 FNONBLOCK，用O\_ASYNC来代替FASYNC。

描 述	fcntl	ioctl
通过打开或关闭 so_state中的SS_NBIO来使能或禁止非阻塞功能	FNONBLOCK文件状态标志	FIONBIO命令
通过打开或关闭 sb_flags中的 SB_ASYNC来使能或禁止异步通知功能	FASYNC文件状态标志	FIOASYNC命令
设置或得到 so_pgid，它是SIGIOG和SIGURG信号的目标进程或进程组	F_SETOWN或F_GETOWN	SIOCSPGRP或SIOCGPGRP命令
得到接收缓存中的字节数；返回 so_rcv.sb_cc		FIONREAD
返回OOB同步标记；即 so_state中的 SS_RCVATMARK标志		SIOCATMARK

图17-16 fcntl 和ioctl 命令

### 17.5.1 fcntl代码

图17-17列出了fcntl函数的部分代码。

kern\_descrip.c

```

133 struct fcntl_args {
134     int      fd;
135     int      cmd;
136     int      arg;
137 };
138 /* ARGSUSED */
139 fcntl(p, uap, retval)
140 struct proc *p;
141 struct fcntl_args *uap;
142 int      *retval;
143 {
144     struct filedesc *fdp = p->p_fdp;
145     struct file *fp;
146     struct vnode *vp;
147     int      i, tmp, error, flg = F_POSIX;
148     struct flock fl;
149     u_int     newmin;
150     if ((unsigned) uap->fd >= fdp->fd_nfiles ||
151         (fp = fdp->fd_ofiles[uap->fd]) == NULL)
152         return (EBADF);
153     switch (uap->cmd) {
154
155         /* command processing */
156
157     default:
158         return (EINVAL);
159     }
160     /* NOTREACHED */
161 }

```

kern\_descrip.c

图17-17 fcntl 系统调用：概况

133-153 验证完指向打开文件的描述符的正确性后，switch语句处理请求的命令。

253-257 对于不认识的命令，fcntl返回EINVAL。

图17-18仅显示fcntl中与插口有关的代码。

```

168     case F_GETFL:
169         *retval = OFLAGS(fp->f_flag);
170         return (0);
171     case F_SETFL:
172         fp->f_flag &= ~FCNTLFLAGS;
173         fp->f_flag |= FFLAGS(uap->arg) & FCNTLFLAGS;
174         tmp = fp->f_flag & FNONBLOCK;
175         error = (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) &tmp, p);
176         if (error)
177             return (error);
178         tmp = fp->f_flag & FASYNC;
179         error = (*fp->f_ops->fo_ioctl) (fp, FIOASYNC, (caddr_t) &tmp, p);
180         if (!error)
181             return (0);
182         fp->f_flag &= ~FNONBLOCK;
183         tmp = 0;
184         (void) (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) &tmp, p);
185         return (error);
186     case F_GETOWN:
187         if (fp->f_type == DTYPE_SOCKET) {
188             *retval = ((struct socket *) fp->f_data)->so_pgid;
189             return (0);
190         }
191         error = (*fp->f_ops->fo_ioctl)
192             (fp, (int) TIOCGPRG, (caddr_t) retval, p);
193         *retval = -*retval;
194         return (error);
195     case F_SETOWN:
196         if (fp->f_type == DTYPE_SOCKET) {
197             ((struct socket *) fp->f_data)->so_pgid = uap->arg;
198             return (0);
199         }
200         if (uap->arg <= 0) {
201             uap->arg = -uap->arg;
202         } else {
203             struct proc *p1 = pfind(uap->arg);
204             if (p1 == 0)
205                 return (ESRCH);
206             uap->arg = p1->p_pgrp->pg_id;
207         }
208         return ((*fp->f_ops->fo_ioctl)
209             (fp, (int) TIOCSGRP, (caddr_t) &uap->arg, p));

```

图17-18 fcntl 系统调用：插口处理

168-185 F\_GETFL返回与描述符相关的当前文件状态标志，F\_SETFL设置状态标志。通过调用fo\_ioctl将FNONBLOCK和FASYNC的新设置传递给对应的插口，而插口的新设置是通过图17-20中描述的soo\_ioctl函数来传递的。只有在第二个fo\_ioctl调用失败后，才第三次调用fo\_ioctl。该调用的功能是清除FNONBLOCK标志，但是应该改为将这个标志恢复

到原来的值。

186-194 `F_GETOWN`返回与插口相关联的进程或进程组的标识符，`so_pgid`。对于非插口描述符，将`TIOCGPGRP`命令传给对应的`fo_ioctl`函数。`F_SETOWN`的功能是给`so_pgid`赋一个新值。

### 17.5.2 ioctl代码

我们跳过`ioctl`系统调用本身而先从`soo_ioctl`开始讨论，如图17-20所示，因为`ioctl`的代码中的大部分是从图17-17所示的代码中复制的。我们已经说过，`soo_ioctl`函数将选路命令发送给`rtioctl`，接口命令发送给`ifioctl`，任何其他的命令发送给底层协议的`pr_usrreq`函数。

55-68 有几个命令是由`soo_ioctl`直接处理的。如果`*data`非空，则`FIONBIO`打开非阻塞方式，否则关闭非阻塞方式。正于我们已经了解的，这个标志将影响到`accept`、`connect`和`close`系统调用，也包括其他的读和写系统调用。

69-79 `FIOASYNC`使能或禁止异步I/O通知功能。如果设置了`SS_ASYNC`，则无论什么时候插口上有活动，就调用`sowakeup`，将信号`SIGIO`发送给相应的进程或进程组。

80-88 `FIONREAD`返回接收缓存中的可读字节数。`SIOCSPGRP`设置与插口相关的进程组，`SIOCGPGRP`则是得到它。`so_pgid`作为我们刚讨论过的`SIGIO`信号的目标进程或进程组，当有带外数据到达插口时，则作为`SIGURG`信号的目标进程或进程组。

89-92 如果插口正处于带外数据的同步标记，则`SIOCATMARK`返回真；否则返回假。

`ioctl`命令，`FIOxxx`和`SIOxxx`常量，有一个内部结构，如图17-19所示。

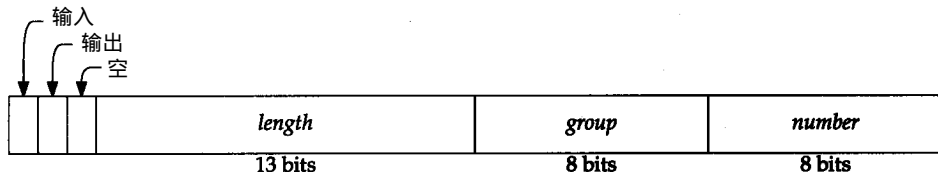


图17-19 ioctl 命令的内部结构

```

55 soo_ioctl(fp, cmd, data, p)                                     sys_socket.c
56 struct file *fp;
57 int cmd;
58 caddr_t data;
59 struct proc *p;
60 {
61     struct socket *so = (struct socket *) fp->f_data;
62     switch (cmd) {
63     case FIONBIO:
64         if (*(int *) data)
65             so->so_state |= SS_NBIO;
66         else
67             so->so_state &= ~SS_NBIO;
68         return (0);
69     case FIOASYNC:
70         if (*(int *) data) {
71             so->so_state |= SS_ASYNC;
72             so->so_rcv.sb_flags |= SB_ASYNC;

```

图17-20 soo\_ioctl 函数

```
73         so->so_snd.sb_flags |= SB_ASYNC;
74     } else {
75         so->so_state &= ~SS_ASYNC;
76         so->so_rcv.sb_flags &= ~SB_ASYNC;
77         so->so_snd.sb_flags &= ~SB_ASYNC;
78     }
79     return (0);
80 case FIONREAD:
81     *(int *) data = so->so_rcv.sb_cc;
82     return (0);
83 case SIOCSGRP:
84     so->so_pgid = *(int *) data;
85     return (0);
86 case SIOCGGRP:
87     *(int *) data = so->so_pgid;
88     return (0);
89 case SIOCATMARK:
90     *(int *) data = (so->so_state & SS_RCVATMARK) != 0;
91     return (0);
92 }
93 /*
94  * Interface/routing/protocol specific ioctls:
95  * interface and routing ioctls should have a
96  * different entry since a socket's unnecessary
97  */
98 if (IOCGROUP(cmd) == 'i')
99     return (ifioctl(so, cmd, data, p));
100 if (IOCGROUP(cmd) == 'r')
101     return (rtioctl(cmd, data, p));
102 return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
103     (struct mbuf *) cmd, (struct mbuf *) data, (struct mbuf *) 0));
104 }
```

sys\_socket.c

图17-20 (续)

如果将ioctl的第三个参数作为输入，则设置input。如果该参数作为输出，则 output被置位。如果不用该参数，则 void被置位。length是参数的大小(字节)。相关的命令在同一个group中但每一个命令在组中都有各自的number。图17-21中的宏用来解析 ioctl命令中的元素。

宏	描 述
IOCPARM_LEN(cmd)	返回cmd中的length
IOCBASECMD(cmd)	length设为0的命令
IOCGROUP(cmd)	返回cmd中的group

图17-21 ioctl 命令宏

93-104 宏IOCGROUP从命令中得到8 bit的group。接口命令由ifioctl处理。选路命令由rtioctl处理。通过PRU\_CONTROL请求将所有其他的命令传递给插口协议。

正如我们在第19章中描述的，Net/2定义了一个新的访问路由选择表接口，在该接口中，报文是通过一个在PF\_ROUTE域中产生的插口传递给路由选择子系统。用这种方法来代替这里讨论的ioctl。在不兼容的内核中，rtioctl总是返回ENOTSUPP。

17.6 getsockname系统调用

getsockname系统调用的原型是：

```
int getsockname(int fd, caddr_t asa, int * alen);
```

getsockname得到绑定在插口`fd`上的本地地址，并将它存入`asa`指向的缓存中。当在一个隐式的绑定中内核选择了一个地址，或在一个显式的`bind`调用中进程指定了一个通配符地址(2.2.5节)时，该函数就很有用。getsockname系统调用如图17-22所示。

```

682 struct getsockname_args {
683     int      fdes;
684     caddr_t  asa;
685     int      *alen;
686 };

687 getsockname(p, uap, retval)
688 struct proc *p;
689 struct getsockname_args *uap;
690 int      *retval;
691 {
692     struct file *fp;
693     struct socket *so;
694     struct mbuf *m;
695     int      len, error;

696     if (error = getsock(p->p_fd, uap->fdes, &fp))
697         return (error);
698     if (error = copyin((caddr_t) uap->alen, (caddr_t) &len, sizeof(len)))
699         return (error);
700     so = (struct socket *) fp->f_data;
701     m = m_getclr(M_WAIT, MT_SONAME);
702     if (m == NULL)
703         return (ENOBUFFS);
704     if (error = (*so->so_proto->pr_usrreq) (so, PRU_SOCKADDR, 0, m, 0))
705         goto bad;
706     if (len > m->m_len)
707         len = m->m_len;
708     error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len);
709     if (error == 0)
710         error = copyout((caddr_t) &len, (caddr_t) uap->alen,
711             sizeof(len));
712 bad:
713     m_freem(m);
714     return (error);
715 }

```

uipc\_syscalls.c

uipc\_syscalls.c

图17-22 getsockname 系统调用

682-715 getsock返回描述符的file结构。将进程指定的缓存的长度赋给`len`。这是我们第一次看到对`m_getclr`的调用：该函数分配一个标准的mbuf，并调用`bzero`清零。当协议收到`PRU_SOCKADDR`请求时，协议处理层负责将本地地址存入`m`。

如果地址长度大于进程提供的缓存的长度，则返回的地址将被截掉。`*alen`等于实际返回的字节数。最后，释放mbuf，并返回。

## 17.7 getpeername系统调用

getpeername系统调用的原型是：

```
int getpeername(int fd, caddr_t asa, int * alen);
```

getpeername系统调用返回指定插口上连接的远端地址。当一个调用 accept 的进程通过fork和exec启动一个服务器时(即,任何被inetd启动的服务器),经常要调用这个函数。服务器不能得到accept返回的远端地址,而只能调用getpeername。通常,要在应用的访问地址表查找返回地址,如果返回地址不在访问表中,则连接将被关闭。

某些协议,如TP4,利用这个函数来确定是否拒绝或证实一个进入的连接。在TP4中,accept返回的插口上的连接是不完整的,必须经证实之后才算连接成功。基于getpeername返回的地址,服务器能够关闭连接或通过发送或接收数据来间接证实连接。这一特点与TCP无关,因为TCP必须在三次握手完成之后,accept才能建立连接。图17-23列出了getpeername函数的代码。

```

719 struct getpeername_args {
720     int      fdes;
721     caddr_t  asa;
722     int      *alen;
723 };

724 getpeername(p, uap, retval)
725 struct proc *p;
726 struct getpeername_args *uap;
727 int      *retval;
728 {
729     struct file *fp;
730     struct socket *so;
731     struct mbuf *m;
732     int      len, error;

733     if (error = getsock(p->p_fd, uap->fdes, &fp))
734         return (error);
735     so = (struct socket *) fp->f_data;
736     if ((so->so_state & (SS_ISCONNECTED | SS_ISCONFIRMING)) == 0)
737         return (ENOTCONN);
738     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
739         return (error);
740     m = m_getclr(M_WAIT, MT_SONAME);
741     if (m == NULL)
742         return (ENOBUFS);
743     if (error = (*so->so_proto->pr_usrreq) (so, PRU_PEERADDR, 0, m, 0))
744         goto bad;
745     if (len > m->m_len)
746         len = m->m_len;
747     if (error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len))
748         goto bad;
749     error = copyout((caddr_t) & len, (caddr_t) uap->alen, sizeof(len));
750 bad:
751     m_freem(m);
752     return (error);
753 }

```

图17-23 getpeername 系统调用

719-753 图中列出的代码与getsockname的代码是一样的。getsock获取插口对应的file结构,如果插口还没有同对方建立连接或连接还没有证实(如,TP4),则返回ENOTCONN。如果已建立连接,则从进程那里得到缓存的大小,并分配一块 mbuf来存储地址。发送PRU\_PEERADDR请求给协议层来获取远端地址。将地址和地址的长度从内核的mbuf中复制到



进程提供的缓存中。释放 mbuf，并返回。

## 17.8 小结

本章中，我们讨论了六个修改插口功能的函数。插口选项由 `setsockopt` 和 `getsockopt` 函数处理。其他的选项，其中有些不仅仅用于插口，由 `fcntl` 和 `ioctl` 处理。最后，通过 `getsockname` 和 `getpeername` 来获取连接信息。

## 习题

- 17.1 为什么选项受标准 mbuf 大小 (MHLEN, 128 个字节) 的限制?
- 17.2 为什么图 17-7 中的最后一段代码能处理 `SO_LINGER` 选项?
- 17.3 图 17-9 中用来测试 `timeval` 结构的代码有些问题，因为 `tv->tv_sec * hz` 可能会溢出。请对这段代码作些修改来解决这个问题。