

第15章 插口层

15.1 引言

本书共有三章介绍 Net/3 的插口层代码，本章是第一章。插口概念最早出现于 1983 年的 4.2BSD 版本中，它的主要目的是提供一个统一的访问网络和进程间通信协议的接口。这里讨论的 Net/3 版基于 4.3BSD Reno 版，该版本与大多数 Unix 供应商使用的早期的 4.2 版有些细小的差别。

如第 1.7 节所介绍的，插口层的主要功能是将进程发送的与协议有关的请求映射到产生插口时指定的与协议有关的实现。

为了允许标准的 Unix I/O 系统调用，如 `read` 和 `write`，也能读写网络连接，在 BSD 版本中将文件系统和网络功能集成在系统调用级。与通过一个描述符访问一个打开的文件一样，进程也是通过一个描述符（一个小整数）来访问插口上的网络连接。这个特点使得标准的文件系统调用，如 `read` 和 `write`，以及与网络有关的系统调用，如 `sendmsg` 和 `recvmsg`，都能通过描述符来处理插口。

我们的重点是插口及相关的系统调用的实现而不是讨论如何使用插口层来实现网络应用。关于进程级的插口接口和如何编写网络应用的详细讨论，请参考 [Stevens 1990] 和 [Rago 1990]。

图 15-1 说明了进程中的插口接口与内核中的协议实现之间的层次关系。

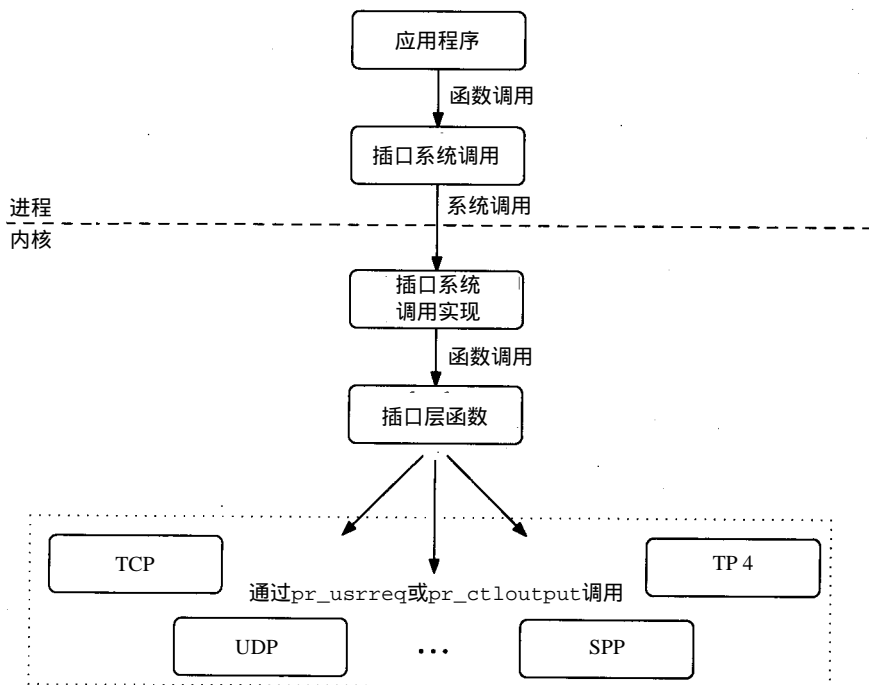


图 15-1 插口层将一般的请求转换为指定的协议操作

splnet处理

插口包含很多对 `splnet` 和 `splx` 的成对调用。正如第 1.12 节中介绍的，这些调用保护访问在插口层和协议处理层间共享的数据结构的代码。如果不使用 `splnet`，初始化协议处理和改变共享的数据结构的软件中断将使得插口层代码恢复执行时出现混乱。

我们假定读者理解了这些调用，因而在以后讨论中一般不再特别说明它们。

15.2 代码介绍

本章讨论涉及的两个文件在图 15-2 中列出。

文 件	描 述
<code>sys/socketvar.h</code>	socket 结构定义
<code>kern/uipc_syscalls.c</code> <code>kern/uipc_socket.c</code>	系统调用实现 插口层函数

图15-2 本章讨论涉及的源文件

全局变量

本章讨论涉及到的两个全局变量如图 15-3 所示。

变 量	数据类型	描 述
<code>socketps</code> <code>sysent</code>	<code>struct fileops</code> <code>struct sysent</code>	I/O 系统调用的 socket 实现 系统调用入口数组

图15-3 本章介绍的全局变量

15.3 socket 结构

插口代表一条通信链路的一端，存储或指向与链路有关的所有信息。这些信息包括：使用的协议、协议的状态信息（包括源和目的地址）、到达的连接队列、数据缓存和可选标志。图 15-5 中给出了插口和与插口相关的缓存的定义。

41-42 `so_type` 由产生插口的进程来指定，它指明插口和相关协议支持的通信语义。`so_type` 的值等于图 7-8 所示的 `pr_type`。对于 UDP，`so_type` 等于 `SOCK_DGRAM`，而对于 TCP，`so_type` 则等于 `SOCK_STREAM`。

43 `so_options` 是一组改变插口行为的标志。图 15-4 列出了这些标志。

通过 `getsockopt` 和 `setsockopt` 系统调用进程能修改除 `SO_ACCEPTCONN` 外所有的插口选项。当在插口上发送 `listen` 系统调用时，`SO_ACCEPTCONN` 被内核设置。

44 `so_linger` 等于当关闭一条连接时插口继续发送数据的时间间隔（单位为一个时钟滴答）（第 15.15 节）。

45 `so_state` 表示插口的内部状态和一些其他的特点。图 15-6 列出了 `so_state` 可能的取值。

so_options	仅用于内核	描 述
SO_ACCEPTCONN	•	插口接受进入的连接
SO_BROADCAST		插口能够发送广播报文
SO_DEBUG		插口记录排错信息
SO_DONTROUTE		输出操作旁路选路表
SO_KEEPAIVE		插口查询空闲的连接
SO_OOBINLINE		插口将带外数据同正常数据存放在一起
SO_REUSEADDR		插口能重新使用一个本地地址
SO_REUSEPORT		插口能重新使用一个本地地址和端口
SO_USELOOPBACK		仅针对选路域插口；发送进程收到它自己的选路请求

图15-4 so_options 的值

socketvar.h

```

41 struct socket {
42     short    so_type;           /* generic type, Figure 7.8 */
43     short    so_options;        /* from socket call, Figure 15.5 */
44     short    so_linger;         /* time to linger while closing */
45     short    so_state;          /* internal state flags, Figure 15.6 */
46     caddr_t  so_pcb;            /* protocol control block */
47     struct protosw *so_proto;    /* protocol handle */
48 /*
49  * Variables for connection queueing.
50  * Socket where accepts occur is so_head in all subsidiary sockets.
51  * If so_head is 0, socket is not related to an accept.
52  * For head socket so_q0 queues partially completed connections,
53  * while so_q is a queue of connections ready to be accepted.
54  * If a connection is aborted and it has so_head set, then
55  * it has to be pulled out of either so_q0 or so_q.
56  * We allow connections to queue up based on current queue lengths
57  * and limit on number of queued connections for this socket.
58  */
59     struct socket *so_head;      /* back pointer to accept socket */
60     struct socket *so_q0;        /* queue of partial connections */
61     struct socket *so_q;         /* queue of incoming connections */
62     short    so_q0len;          /* partials on so_q0 */
63     short    so_qlen;           /* number of connections on so_q */
64     short    so_qlimit;          /* max number queued connections */
65     short    so_timeo;           /* connection timeout */
66     u_short  so_error;           /* error affecting connection */
67     pid_t    so_pgid;           /* pgid for signals */
68     u_long   so_oobmark;        /* chars to oob mark */
69 /*
70  * Variables for socket buffering.
71  */
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;
83     caddr_t  so_tpcb;           /* Wisc. protocol control block XXX */

```

图15-5 struct socket定义

```

84 void      (*so_upcall) (struct socket * so, caddr_t arg, int waitf);
85 caddr_t so_upcallarg;      /* Arg for above */
86 };

```

socketvar.h

图15-5 (续)

so_state	仅用于内核	描 述
SS_ASYNC SS_NBIO		插口应该I/O事件的异步通知 插口操作不能阻塞进程
SS_CANTRCVMORE SS_CANTSENDMORE SS_ISCONFIRMING SS_ISCONNECTED SS_ISCONNECTING SS_ISDISCONNECTING SS_NOFDREF SS_PRIV SS_RCVATMARK	• • • • • • • • •	插口不能再从对方接收数据 插口不能再发送数据给对方 插口正在协商一个连接请求 插口被连接到外部插口 插口正在连接一个外部插口 插口正在同对方断连 插口没有同任何描述符相连 插口由拥有超级用户权限的进程所产生 在最近的带外数据到达之前, 插口已处理完所有收到的数据

图15-6 so_state 的值

从图 15-6 的第二列中可以看出, 进程可以通过 `fcntl` 和 `ioctl` 系统调用直接修改 `SS_ASYNC` 和 `SS_NBIO`。对于其他的标志, 进程只能在系统调用的执行过程中间接修改。例如, 如果进程调用 `connect`, 当连接被建立时, `SS_ISCONNECTED` 标志就会被内核设置。

SS_NBIO和SS_ASYNC标志

在默认情况下, 进程在发出 I/O 请求后会等待资源。例如, 对一个插口发 `read` 系统调用, 如果当前没有网络上来的数据, 则 `read` 系统调用就会被阻塞。同样, 当一个进程调用 `write` 系统调用时, 如果内核中没有缓存来存储发送的数据, 则内核将阻塞进程。如果设置了 `SS_NBIO`, 在对插口执行 I/O 操作且请求的资源不能得到时, 内核并不阻塞进程, 而是返回 `EWOULDBLOCK`。

如果设置了 `SS_ASYNC`, 当因为下列情况之一而使插口状态发生变化时, 内核发送 `SIGIO` 信号给 `so_pgid` 标识的进程或进程组:

- 连接请求已完成;
- 断连请求已被启动;
- 断连请求已完成;
- 连接的一个通道已被关闭;
- 插口上有数据到达;
- 数据已被发送(即, 输出缓存中有闲置空间); 或
- UDP或TCP插口上出现了一个异步差错。

46 `so_pcb` 指向协议控制块, 协议控制块包含与协议有关的状态信息和插口参数。每一种协议都定义了自己的控制块结构, 所以 `so_pcb` 被定义成一个通用的指针。图 15-7 列出了我们讨论的控制块结构。

`so_pcb` 从来不直接指向 `tcpcb` 结构; 参考图 22-1。

协 议	控 制 块	参考章节
UDP	struct inpcb	第22.3节
TCP	struct inpcb struct tcpcb	第22.3节 第24.5节
ICMP、IGMP和原始IP	struct inpcb	第22.3节
路由	struct rawcb	第20.3节

图15-7 协议控制块

47 `so_proto`指向进程在`socket`系统调用(第7.4节)中选择的协议的`protosw`结构。

48-64 设置了`SO_ACCEPTCONN`标志的插口维护两个连接队列。还没有完全建立的连接(如TCP的三次握手还没完成)被放在队列`so_q0`中。已经建立的连接或将被接受的连接(例如, TCP的三次握手已完成)被放入队列`so_q`中。队列的长度分别为`so_q0len`和`so_qlen`。每一个被排队的连接由它自己的插口来表示。在每一个被排队的插口中, `so_head`指向设置了`SO_ACCEPTCONN`的源插口。

插口上可排队的连接数通过`so_qlimit`来控制, 进程可以通过`listen`系统调用来设置`so_qlimit`。内核隐含设置的上限为5(`SOMAXCONN`, 图15-24)和下限为0。图15-29中显示的有点晦涩的公式使用`so_qlimit`来控制排队的连接数。

图15-8说明了有三个连接将被接受、一个连接已被建立的情况下的队列内容。

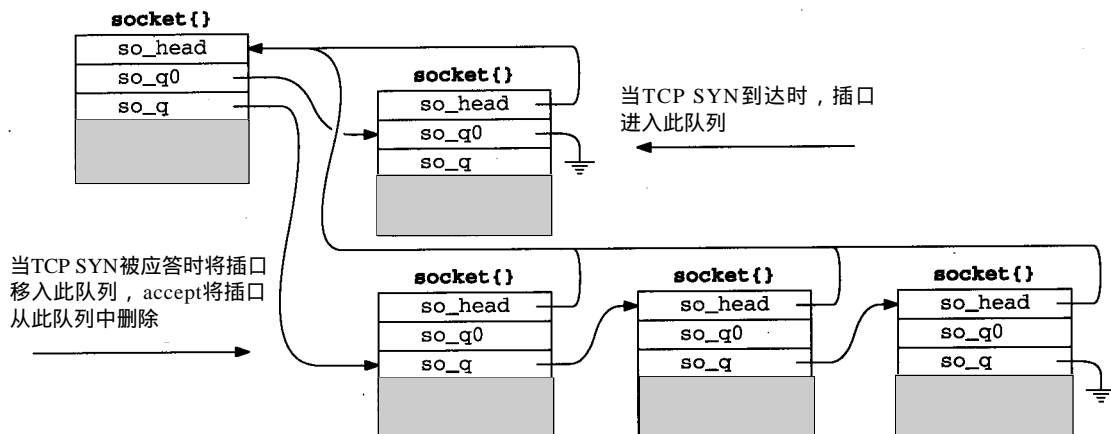


图15-8 插口连接队列

65 `so_timeo`用作`accept`、`connect`和`close`处理期间的等待通道(`wait channel`, 第15.10节)。

66 `so_error`保存差错代码, 直到在引用该插口的下一个系统调用期间差错代码能送给进程。

67 如果插口的`SS_ASYNC`被设置, 则`SIGIO`信号被发送给进程(如果`so_pgid`大于0)或进程组(如果`so_pgid`小于0)。可以通过`ioctl`的`SIOCSPGRP`和`SIOCGPGRP`命令来修改或检查`so_pgid`的值。关于进程组的更详细信息请参考[Stevens 1992]。

68 `so_oobmark`标识在输入数据流中最近收到的带外数据的开始点。第16.11节将讨论插口对带外数据的支持, 第29.7节将讨论TCP中的带外数据的语义。

69-82 每一个插口包括两个数据缓存，`so_rcv`和`so_snd`，分别用来缓存接收或发送的数据。`so_rcv`和`so_snd`是包含在插口结构中的结构而不是指向结构的指针。我们将在第16章中描述插口缓存的结构和使用。

83-86 在Net/3中不使用`so_tpcb`。`so_upcall`和`so_upcallarg`也仅用于Net/3中的NFS软件。

NFS与通常的软件不太一样。在很大程度上它是一个进程级的应用但却在内核中运行。当数据到达接收缓存时，通过`so_upcall`来触发NFS的输入处理。在这种情况下，`tsleep`和`wakeup`机制是不合适的，因为NFS协议是在内核中运行而不是作为一个普通进程。

文件`socketvar.h`和`uiipc_socket2.c`定义了几个简化插口层代码的宏和函数。图15-9对它们进行了描述。

名 称	描 述
<code>sosendallatonce</code>	<code>so</code> 中指定的协议要求每一个发送系统调用产生一个协议请求吗？ <code>int sosendallatonce(struct socket *so);</code>
<code>soisconnecting</code>	将插口状态设置为 <code>SO_ISCONNECTING</code> <code>int soisconnecting(struct socket *so);</code>
<code>soisconnected</code>	参考图15-30
<code>soreadable</code>	插口 <code>so</code> 上的读调用不阻塞就返回信息吗？ <code>int soreadable(struct socket *so);</code>
<code>sowriteable</code>	插口 <code>so</code> 上的写调用不阻塞就返回吗？ <code>int sowriteable(struct socket *so);</code>
<code>socantsendmore</code>	设置插口标志 <code>SO_CANTSENDMORE</code> 。唤醒所有等待在发送缓存上的进程 <code>int socantsendmore(struct socket *so);</code>
<code>socantrcvmore</code>	设置插口标志 <code>SO_CANTRCVMORE</code> 。唤醒所有等待在接收缓存上的进程 <code>int socantrcvmore(struct socket *so);</code>
<code>soisdisconnecting</code>	清除 <code>SS_ISCONNECTING</code> 标志。设置 <code>SS_ISDISCONG</code> 、 <code>SS_CANTRCVMORE</code> 和 <code>SS_CANTSENDMORE</code> 标志。唤醒所有等待在插口上的进程 <code>int soisdisconnecting(struct socket *so);</code>
<code>soisdisconnected</code>	清除 <code>SS_ISCONNECTING</code> 、 <code>SS_ISCONNECTED</code> 和 <code>SS_ISDISCONNECTING</code> 标志。设置 <code>SS_CANTRCVMORE</code> 和 <code>SS_CANTSENDMORE</code> 标志。唤醒所有等待在插口上的进程或等待 <code>close</code> 完成的进程 <code>int soisdisconnected(struct socket *so);</code>
<code>soqinsque</code>	将 <code>so</code> 插入 <code>head</code> 指向的队列中。如果 <code>q</code> 等于0，插口被插到存放未完成的连接的 <code>so_q0</code> 队列的后面。否则，插口被插到存放准备接受的连接的队列 <code>so_q</code> 的后面。Net/1错误地将插口插到队列的前面 <code>int soqinsque(struct socket * head, struct socket * so, int q);</code>
<code>soqremque</code>	从队列 <code>q</code> 中删除 <code>so</code> 。通过 <code>so->so_head</code> 来定位插口队列 <code>int soqremque(struct socket *so, int q);</code>

图15-9 插口的宏和函数

15.4 系统调用

进程同内核交互是通过一组定义好的函数来进行的，这些函数称为系统调用。在讨论支持网络的系统调用之前，我们先来看看系统调用机制的本身。

从进程到内核中的受保护的环境的转换是与机器和实现相关的。在下面的讨论中，我们使用Net/3在386上的实现来说明如何实现有关的操作。

在BSD内核中，每一个系统调用均被编号，当进程执行一个系统调用时，硬件被配置成仅传送控制给一个内核函数。将标识系统调用的整数作为参数传给该内核函数。在386实现中，这个内核函数为 `syscall`。利用系统调用的编号，`syscall`在表中找到请求的系统调用的 `sysent` 结构。表中的每一个单元均为一个 `sysent` 结构。

```
struct sysent {
    int sy_narg;           /* number of arguments */
    int (*sy_call) ();     /* implementing function */
};                          /* system call table entry */
```

表中有几个项是从 `sysent` 数组中来的，该数组是在 `kern/init_sysent.c` 中定义的。

```
struct sysent sysent[] = {
    /* ... */
    { 3, recvmsg },        /* 27 = recvmsg */
    { 3, sendmsg },        /* 28 = sendmsg */
    { 6, recvfrom },       /* 29 = recvfrom */
    { 3, accept },         /* 30 = accept */
    { 3, getpeername },    /* 31 = getpeername */
    { 3, getsockname },    /* 32 = getsockname */
    /* ... */
}
```

例如，`recvmsg` 系统调用在系统调用表中的第 27 个项，它有两个参数，利用内核中的 `recvmsg` 函数实现。

`syscall` 将参数从调用进程复制到内核中，并且分配一个数组来保存系统调用的结果。然后，当系统调用执行完成后，`syscall` 将结果返回给进程。`syscall` 将控制交给与系统调用相对应的内核函数。在 386 实现中，调用有点像：

```
struct sysent *callp;
error = (*callp->sy_call) (p, args, rval);
```

这里指针 `callp` 指向相关的 `sysent` 结构；指针 `p` 则指向调用系统调用的进程的进程表项；`args` 作为参数传给系统调用，它是一个 32 bit 长的数组；而 `rval` 则是一个用来保存系统调用的返回结果的数组，数组有两个元素，每个元素是一个 32 bit 长的字。当我们用“系统调用”这个词时，我们指的是被 `syscall` 调用的内核中的函数，而不是应用调用的进程中的函数。

`syscall` 期望系统调用函数（即 `sy_call` 指向的函数）在没有差错时返回 0，否则返回非 0 的差错代码。如果没有差错出现，内核将 `rval` 中的值作为系统调用（应用调用的）的返回值传送给进程。如果有差错，`syscall` 忽略 `rval` 中的值，并以与机器相关的方式返回差错代码给进程，使得进程能从外部变量 `errno` 中得到差错代码。应用调用的函数则返回 -1 或一个空指针表示应用应该查看 `errno` 获得差错信息。

在 386 上的实现，设置进位比特（carry bit）来表示 `syscall` 的返回值是一个差错代码。进程中的系统调用残桩将差错代码赋给 `errno`，并返回 -1 或空指针给应用。如果没有设置进位

比特, 则将 `syscall` 返回的值返回给进程中的系统调用的残桩。

总之, 实现系统调用的函数“返回”两个值: 一个给 `syscall` 函数; 在没有差错的情况下, `syscall` 将另一个(在 `rval` 中)返回给调用进程。

15.4.1 举例

`socket` 系统调用的原型是:

```
int socket(int domain, int type, int protocol);
```

实现 `socket` 系统调用的内核函数的原型是:

```
struct socket_args {  
    int domain;  
    int type;  
    int protocol;  
};  
socket(struct proc *p, struct socket_args *uap, int *retval);
```

当一个应用调用 `socket` 时, 进程用系统调用机制将三个独立的整数传给内核。 `syscall` 将参数复制到 32bit 值的数组中, 并将数组指针作为第二个参数传给 `socket` 的内核版。内核版的 `socket` 将第二个参数作为指向 `socket_args` 结构的指针。图 15-10 显示了上述过程。

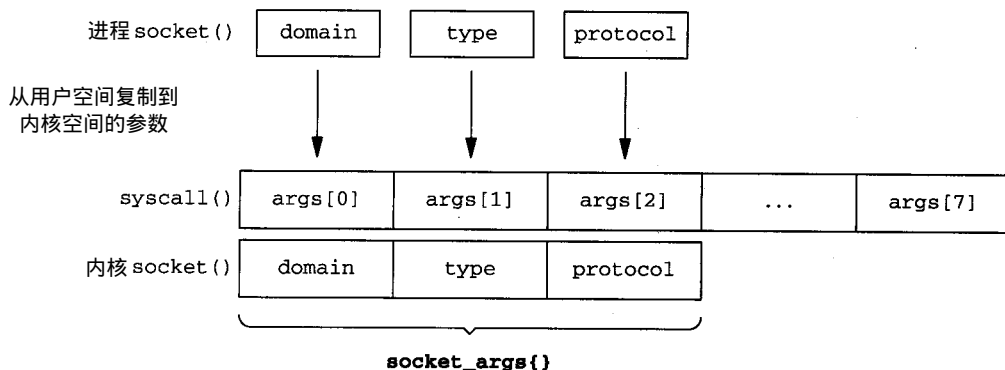


图15-10 socket 参数处理

同 `socket` 类似, 每一个实现系统调用的内核函数将 `args` 说明成一个与系统调用有关的结构指针, 而不是一个指向 32 bit 的字的数组的指针。

当原型无效时, 隐式的类型转换仅在传统的 K&R C 中或 ANSI C 中是合法的。如果原型是有效的, 则编译器将产生一个警告。

`syscall` 在执行内核系统调用函数之前将返回值置为 0。如果没有差错出现, 系统调用函数直接返回而不需清除 `*retval`, `syscall` 返回 0 给进程。

15.4.2 系统调用小结

图 15-11 对与网络有关的系统调用进行了小结。

我们将在本章中讨论建立、服务器、客户和终止类系统调用。输入、输出类系统调用将在第 16 章中介绍, 管理类系统调用将在第 17 章中介绍。

类 别	名 称	功 能
建 立	socket bind	在指明的通信域内产生一个未命名的插口 分配一个本地地址给插口
服务器	listen accept	使插口准备接收连接请求 等待并接受连接
客 户	connect	同外部插口建立连接
输 入	read readv recv recvfrom recvmsg	接收数据到一个缓存中 接收数据到多个缓存中 指明选项接收数据 接收数据和发送者的地址 接收数据到多个缓存中，接收控制信息和发送者地址；指明接收选项
输 出	write writev send sendto sendmsg	发送一个缓存中的数据 发送多个缓存中的数据 指明选项发送数据 发送数据到指明的地址 从多个缓存发送数据和控制信息到指明的地址；指明发送选项
I/O	select	等待I/O事件
终 止	shutdown close	终止一个或两个方向上的连接 终止连接并释放插口
管 理	fcntl ioctl setsockopt getsockopt getsockname getpeername	修改I/O语义 各类插口操作 设置插口或协议选项 得到插口或协议选项 得到分配给插口的本地地址 得到分配给插口的远端地址

图15-11 Net/3中的网络系统调用

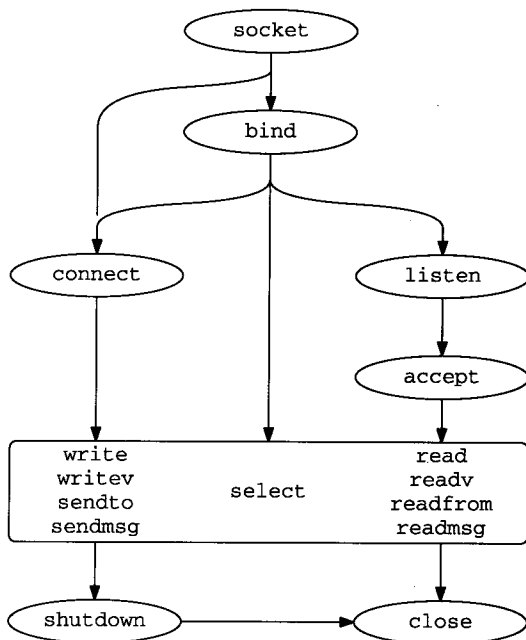


图15-12 网络系统调用流程图

图15-12画出了应用使用这些系统调用的顺序。大方块中的 I/O 系统调用可以在任何时候调用。该图不是一个完整的状态流程图，因为一些正确的转换在本图中没有画出；仅显示了一些常见的转换。

15.5 进程、描述符和插口

在描述插口系统调用之前，我们需要介绍将进程、描述符和插口联系在一起的数据结构。图15-13给出了这些结构以及我们的讨论有关的结构成员。关于文件结构的更复杂的解释请参考[Leffer et al. 1989]。

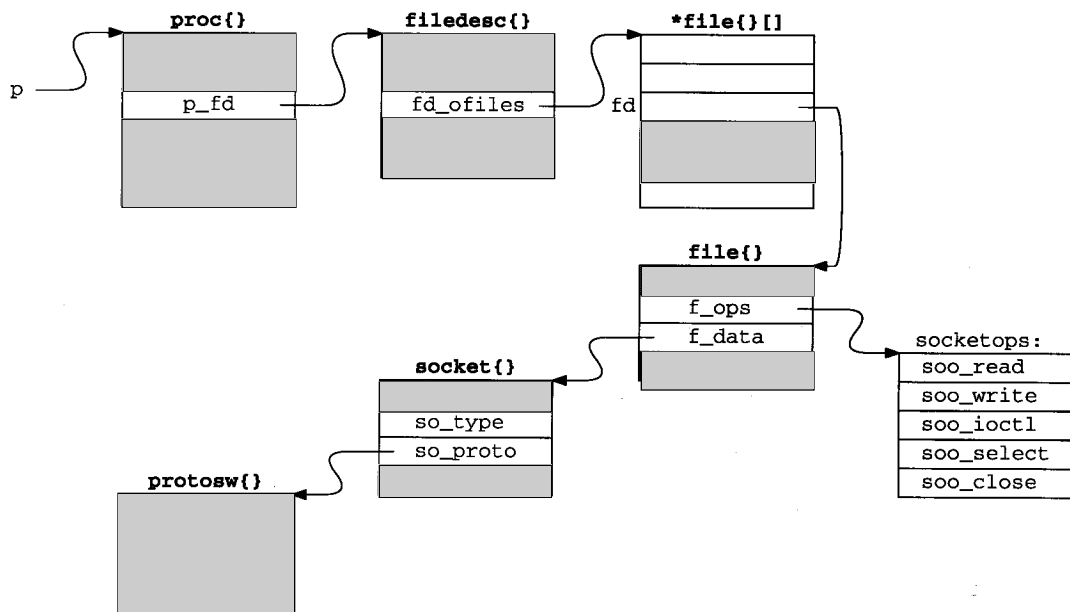


图15-13 进程、文件和插口结构

实现系统调用的函数的第一个参数总为 p ，即指向调用进程的 `proc` 结构的指针。内核利用 `proc` 结构记录进程的有关信息。在 `proc` 结构中，`p_fd` 指向 `filedesc` 结构，该结构的主要功能是管理 `fd_files` 指向的描述符表。描述符表的大小是动态变化的，由一个指向 `file` 结构的指针数组组成。每一个 `file` 结构描述一个打开的文件，该结构可被多个进程共享。

图15-13仅显示了一个 `file` 结构。通过 `p->p_fd->fd_files[fd]` 访问到该结构。在 `file` 结构中，有两个结构成员是我们感兴趣的：`f_ops` 和 `f_data`。I/O 系统调用(如 `read` 和 `write`)的实现因描述符中的 I/O 对象类型的不同而不同。`f_ops` 指向 `fileops` 结构，该结构包含一张实现 `read`、`write`、`ioctl`、`select` 和 `close` 系统调用的函数指针表。图 15-13 显示 `f_ops` 指向一个全局的 `fileops` 结构，即 `socketops`，该结构包含指向插口用的函数的指针。

`f_data` 指向相关 I/O 对象的专用数据。对于插口而言，`f_data` 指向与描述符相关的 `socket` 结构。最后，`socket` 结构中的 `so_proto` 指向产生插口时选中的协议的 `protosw` 结构。回想一下，每一个 `protosw` 结构是由与该协议关联的所有插口共享的。

下面我们开始讨论系统调用。

15.6 socket系统调用

socket系统调用产生一个新的插口，并将插口同进程在参数 domain、type和 protocol中指定的协议联系起来。该函数(如图15-14所示)分配一个新的描述符，用来在后续的系统调用中标识插口，并将描述符返回给进程。

```
42 struct socket_args {
43     int     domain;
44     int     type;
45     int     protocol;
46 };

47 socket(p, uap, retval)
48 struct proc *p;
49 struct socket_args *uap;
50 int     *retval;
51 {
52     struct filedesc *fdp = p->p_fdp;
53     struct socket *so;
54     struct file *fp;
55     int     fd, error;

56     if (error = falloc(p, &fp, &fd))
57         return (error);
58     fp->f_flag = FREAD | FWRITE;
59     fp->f_type = DTYPE_SOCKET;
60     fp->f_ops = &socketops;
61     if (error = socreate(uap->domain, &so, uap->type, uap->protocol)) {
62         fdp->fd_ofiles[fd] = 0;
63         ffree(fp);
64     } else {
65         fp->f_data = (caddr_t) so;
66         *retval = fd;
67     }
68     return (error);
69 }
```

uipc_syscalls.c

uipc_syscalls.c

图15-14 socket 系统调用

42-55 在每一个系统调用的前面，都定义了一个描述进程传递给内核的参数的结构。在这种情况下，参数是通过 socket_args传入的。所有插口层系统调用都有三个参数：p，指向调用进程的proc结构；uap，指向包含进程传送给系统调用的参数的结构；retval，用来接收系统调用的返回值。在通常情况下，忽略参数 p和retval，引用uap所指的结构中的内容。

56-60 falloc分配一个新的file结构和fd_ofiles数组(图15-13)中的一个元素。fp指向新分配的结构，fd则为结构在数组fd_ofiles中的索引。socket将file结构设置成可读、可写，并且作为一个插口。将所有插口共享的全局 fileops结构socketops连接到f_ops指向的file结构中。socketops变量在编译时被初始化，如图15-15所示。

60-69 socreate分配并初始化一个 socket结构。如果socreate执行失败，将差错代码赋给 error，释放file结

成 员	值
fo_read	soo_read
fo_write	soo_write
fo_ioctl	soo_ioctl
fo_select	soo_select
fo_close	soo_close

图15-15 socketops : 插口
用全局fileops 结构

构,清除存放描述符的数组元素。如果 `screate` 执行成功,将 `f_data` 指向 `socket` 结构,建立插口和描述符之间的联系。通过 `*retval` 将 `fd` 返回给进程。`socket` 返回0或返回由 `screate` 返回的差错代码。

15.6.1 `screate` 函数

大多数插口系统调用至少被分成两个函数,与 `socket` 和 `screate` 类似。第一个函数从进程那里获取需要的数据,调用第二个函数 `soxxx` 来完成功能处理,然后返回结果给进程。这种分成多个函数的做法是为了第二个函数能直接被基于内核的网络协议调用,如 NFS。`screate` 的代码如图 15-16 所示。

```

43 screate(dom, aso, type, proto)                                uipc_socket.c
44 int     dom;
45 struct socket **aso;
46 int     type;
47 int     proto;
48 {
49     struct proc *p = curproc;    /* XXX */
50     struct protosw *prp;
51     struct socket *so;
52     int     error;

53     if (proto)
54         prp = pffindproto(dom, proto, type);
55     else
56         prp = pffindtype(dom, type);
57     if (prp == 0 || prp->pr_usrreq == 0)
58         return (EPROTONOSUPPORT);
59     if (prp->pr_type != type)
60         return (EPROTOPTYPE);
61     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
62     bzero((caddr_t) so, sizeof(*so));
63     so->so_type = type;
64     if (p->p_ucred->cr_uid == 0)
65         so->so_state = SS_PRIV;
66     so->so_proto = prp;
67     error =
68         (*prp->pr_usrreq) (so, PRU_ATTACH,
69             (struct mbuf *) 0, (struct mbuf *) proto, (struct mbuf *) 0);
70     if (error) {
71         so->so_state |= SS_NOFDREF;
72         sofree(so);
73         return (error);
74     }
75     *aso = so;
76     return (0);
77 }

```

uipc_socket.c

图15-16 `screate` 函数

43-52 `screate` 共有四个参数：`dom`, 请求的协议域(如, `PF_INET`)；`aso`, 保存指向一个新的 `socket` 结构的指针；`type`, 请求的插口类型(如, `SOCK_STREAM`)；`proto`, 请求的协议。

1. 发现协议交换表

53-60 如果proto等于非0值，pffindproto查找进程请求的协议。如果 proto等于0，pffindtype用由type指定的语义在指定域中查找一种协议。这两个函数均返回一个指向匹配协议的protosw结构的指针或空指针(参考第7.6节)。

2. 分配并初始化socket结构

61-66 socreate分配一个新的socket结构，并将结构内容全清成0，记录下type。如果调用进程有超级用户权限，则设置插口结构中的SS_PRIV标志。

3. PRU_ATTACH请求

67-69 在与协议无关的插口层中发送与协议有关的请求的第一个例子出现在 socreate中。回想在第7.4节和图15-13中，so->so_proto->pr_usrreq是一个指向与插口so相关联的协议的用户请求函数指针。每一个协议均提供了一个这样的函数来处理从插口层来的通信请求。函数原型是：

```
int pr_usrreq(struct socket *so, int req, struct mbuf *m0, *m1, *m2);
```

第一个参数是一个指向相关插口的指针， req是一个标识请求的常数。后三个参数 (m0， m1， m2)因请求不同而异。它们总是被作为一个mbuf结构指针传递，即使它们是其他的类型。在必要的时候，进行类似转换以避免编译器的警告。

图15-17列出了pr_usrreq函数提供的通信请求。每一个请求的语义起决于服务请求的协议。

请 求	参 数			描 述
	m0	m1	m2	
PRU_ABORT				异常终止每一个存在的连接
PRU_ACCEPT		address		等待并接受连接
PRU_ATTACH		protocol		产生了一个新的插口
PRU_BIND		address		绑定地址到插口
PRU_CONNECT		address		同地址建立关联或连接
PRU_CONNECT2		socket2		将两个插口连在一起
PRU_DETACH				插口被关闭
PRU_DISCONNECT				切断插口和另一地址间的关联
PRU_LISTEN				开始监听连接请求
PRU_PEERADDR		buffer		返回与插口关联的对方地址
PRU_RCVD		flags		进程已收到一些数据
PRU_RCVOOB	buffer	flags		接收OOB数据
PRU_SEND	data	address	control	发送正常数据
PRU_SENDOOB	data	address	control	发送OOB数据
PRU_SHUTDOWN				结束同另一地址的通信
PRU_SOCKADDR		buffer		返回与插口相关联的本地地址

图15-17 pr_usrreq 函数

PRU_CONNECT2请求只用于 Unix域，它的功能是将两个本地插口连接起来。

Unix的管道(pipe)就是通过这种方式来实现的。

4. 退出处理

70-77 回到socreate，函数将协议交换表连接到插口，发送 PRU_ATTACH请求通知协议

已建立一个新的连接端点。该请求引起大多数协议，如 TCP和UDP，分配并初始化所有支持新的连接端点的数据结构。

15.6.2 超级用户特权

图15-18列出了要求超级用户权限的网络操作。

函 数	超级用户		描 述	参考图
	进程	插口		
in_control		•	分配接口地址、网络掩码、目的地址	图6-14
in_control		•	分配广播地址	图6-22
in_pcbbind	•		绑定到一个小于1024的Internet端口	图22-22
ifioctl	•		改变接口配置	图4-29
ifioctl	•		配置多播地址(见下面的说明)	图12-11
rip_usrreq	•		产生一个ICMP、IGMP或原始 IP插口	图32-10
slopen	•		将一个SLIP设备与一个tty设备联系起来	图5-9

图15-18 Net/3中的超级用户特权

当多播ioctl命令(SIOCADDMULTI和SIOCDELMULTI)是被IP_ADD_MEMBERSHIP和IP_DROP_MEMBERSHIP插口选项间接激活时，它可以被非超级用户访问。

在图15-18中，“进程”栏表示请求必须由超级用户进程来发起，“插口”栏表示请求是针对由超级用户产生的插口(也就是说，进程不需要超级用户权限，而只需有访问插口的权限，习题15.1)。在Net/3中，suser函数用来判断调用进程是否有超级用户权限，通过SS_PRIV标志来判断一个插口是否由超级用户进程产生。

因为rip_usrreq在用screate产生插口后立即检查SS_PRIV标志，所以我们认为只有超级用户进程才能访问这个函数。

15.7 getsock和sockargs函数

这两个函数重复出现在插口系统调用中。getsock的功能是将描述符映射到一个文件表项中，sockargs将进程传入的参数复制到内核中的一个新分配的mbuf中。这两个函数都要检查参数的正确性，如果参数不合法，则返回相应的非0差错代码。

图15-19列出了getsock函数的代码。

754-767 getsock函数利用fdp查找描述符fdes指定的文件表项，fdp是指向filedesc结构的指针。getsock将打开的文件结构指针赋给fpp，并返回，或者当出现下列情况时返回差错代码：描述符的值超过了范围而不是指向一个打开的文件；描述符没有同插口建立联系。

图15-20列出了sockargs函数的代码。

768-783 如图15-4中所描述的，sockargs将进程传给系统调用的参数的指针从进程复制到内核而不是复制指针指向的数据，这样做是因为每一个参数的语义只有相对应的系统调用才知道，而不是针对所有的系统调用。多个系统调用在调用sockargs复制参数指针后，将指针指向的数据从进程复制到内核中新分配的mbuf中。例如，sockargs将bind的第二个参

数指向的本地插口地址从进程复制到一个 mbuf 中。

```

754 getsock(fdp, fdes, fpp)
755 struct filedesc *fdp;
756 int fdes;
757 struct file **fpp;
758 {
759     struct file *fp;
760     if ((unsigned) fdes >= fdp->fd_nfiles ||
761         (fp = fdp->fd_ofiles[fdes]) == NULL)
762         return (EBADF);
763     if (fp->f_type != DTYPE_SOCKET)
764         return (ENOTSOCK);
765     *fpp = fp;
766     return (0);
767 }

```

— uipc_syscalls.c

图15-19 getsock 函数

```

768 sockargs(mp, buf, buflen, type)
769 struct mbuf **mp;
770 caddr_t buf;
771 int buflen, type;
772 {
773     struct sockaddr *sa;
774     struct mbuf *m;
775     int error;
776     if ((u_int) buflen > MLEN) {
777         return (EINVAL);
778     }
779     m = m_get(M_WAIT, type);
780     if (m == NULL)
781         return (ENOBUFS);
782     m->m_len = buflen;
783     error = copyin(buf, mtod(m, caddr_t), (u_int) buflen);
784     if (error)
785         (void) m_free(m);
786     else {
787         *mp = m;
788         if (type == MT_SONAME) {
789             sa = mtod(m, struct sockaddr *);
790             sa->sa_len = buflen;
791         }
792     }
793     return (error);
794 }

```

— uipc_syscalls.c

图15-20 sockargs 函数

如果数据不能存入一个 mbuf 中或无法分配 mbuf，则 sockargs 返回 EINVAL 或 ENOBUFS。注意，这里使用的是标准的 mbuf 而不是分组首部的 mbuf。copyin 的功能是将数据从进程复制到 mbuf 中。copyin 返回的最常见的差错是 EACCES，它表示进程提供的地址不正确。

784-785 当出现差错时，丢弃 mbuf，并返回差错代码。如果没有差错，通过 mp 返回指向 mbuf 的指针，sockargs 返回 0。

786-794 如果 type 等于 MT_SONAME，则进程传入的是一个 sockaddr 结构。sockargs

将刚复制的参数的长度赋给内部长度变量 `sa_len`。这一点确保即使进程没有正确地初始化结构，结构内的大小也是正确的。

Net/3确实包含了一段代码来支持在 pre-4.3BSD Reno 系统上编译的应用，这些应用的 `sockaddr` 结构中并没有 `sa_len` 字段，但是图 15-20 中没有显示这段代码。

15.8 bind系统调用

`bind` 系统调用将一个本地的网络运输层地址和插口联系起来。一般来说，作为客户的进程并不关心它的本地地址是什么。在这种情况下，进程在进行通信之前没有必要调用 `bind`；内核会自动为其选择一个本地地址。

服务器进程则总是需要绑定到一个已知的地址上。所以，进程在接受连接 (TCP) 或接收数据报 (UDP) 之前必须调用 `bind`，因为客户进程需要同已知的地址建立连接或发送数据报到已知的地址。

插口的外部地址由 `connect` 指定或由允许指定外部地址的写调用 (`sendto` 或 `sendmsg`) 指定。

图 15-21 列出了 `bind` 调用的代码。

```

70 struct bind_args {
71     int      s;
72     caddr_t  name;
73     int      namelen;
74 };
75 bind(p, uap, retval)
76 struct proc *p;
77 struct bind_args *uap;
78 int      *retval;
79 {
80     struct file *fp;
81     struct mbuf *nam;
82     int      error;
83     if (error = getsock(p->p_fd, uap->s, &fp))
84         return (error);
85     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
86         return (error);
87     error = sobind((struct socket *) fp->f_data, nam);
88     m_freem(nam);
89     return (error);
90 }

```

— *uipc_syscalls.c*

— *uipc_syscalls.c*

图 15-21 bind 函数

70-82 `bind` 调用的参数有 (在 `bind_args` 结构中)：`s`，插口描述符；`name`，包含传输地址 (如，`sockaddr_in` 结构) 的缓存指针；和 `namelen`，缓存大小。

83-90 `getsock` 返回描述符的 `file` 结构，`sockargs` 将本地地址复制到 `mbuf` 中，`sobind` 将进程指定的地址同插口联系起来。在 `bind` 返回 `sobind` 的结果之前，释放保存地址的 `mbuf`。

从技术上讲，一个描述符，如 `s`，标识一个同 `socket` 结构相关联的 `file` 结构，而它本身并不是一个 `socket` 结构。将这种描述符看作插口是为了简化我们的讨论。

我们将在下面的讨论中经常看到这种模式：进程指定的参数被复制到 `mbuf`，必要时还要

进行处理，然后在系统调用返回之前释放 mbuf。虽然 mbuf 是为方便处理网络数据分组而设计的，但是将它们用作一般的动态内存分配机制也是有效的。

bind 说明的另一种模式是：许多系统调用不使用 retval。在第 15.4 节中我们已提到过，在 syscall 将控制交给相应的系统调用之前总是将 retval 清 0。如果 0 不是合适的返回值，系统调用并不需要修改 retval。

sobind 函数

如图 15-22 所示，sobind 是一个封装器，它给与插口相关联的协议发送 PRU_BIND 请求。

```
78 sobind(so, nam)
79 struct socket *so;
80 struct mbuf *nam;
81 {
82     int      s = splnet();
83     int      error;
84
85     error =
86         (*so->so_proto->pr_usrreq) (so, PRU_BIND,
87                                     (struct mbuf *) 0, nam, (struct mbuf *) 0);
88     splx(s);
89     return (error);
90 }
```

uipc_socket.c

图 15-22 sobind 函数

78-89 sobind 发送 PRU_BIND 请求。如果请求成功，将本地地址 nam 同插口联系起来；否则，返回差错代码。

15.9 listen 系统调用

listen 系统调用的功能是通知协议进程准备接收插口上的连接请求，如图 15-23 所示。它同时也指定插口上可以排队等待的连接数的门限值。超过门限值时，插口层将拒绝进入的连接请求排队等待。当这种情况出现时，TCP 将忽略进入的连接请求。进程可以通过调用 accept (第 15.11 节) 来得到队列中的连接。

```
91 struct listen_args {
92     int      s;
93     int      backlog;
94 };
95
96 listen(p, uap, retval)
97 struct proc *p;
98 struct listen_args *uap;
99 int *retval;
100 {
101     struct file *fp;
102     int      error;
103
104     if (error = getsock(p->p_fd, uap->s, &fp))
105         return (error);
106     return (solisten((struct socket *) fp->f_data, uap->backlog));
107 }
```

uipc_syscalls.c

图 15-23 listen 系统调用

91-98 listen系统调用有两个参数：一个指定插口描述符；另一个指定连接队列门限值。

99-105 getsock返回描述符s的file结构，solisten将请求传递给协议层。

solisten函数

solisten函数发送PRU_LISTEN请求，并使插口准备接收连接，如图15-24所示。

90-109 在solisten发送PRU_LISTEN请求且pr_usrreq返回后，标识插口处于准备接收连接状态。如果当pr_usrreq返回时有连接正在连接队列中，则不设置SS_ACCEPTCONN标志。

计算存放在连接队列的最大值，并赋给so_qlimit。Net/3默认设置下限为0，上限为5(SOMAXCONN)条连接。

```

90 solisten(so, backlog)                                     uipc_socket.c
91 struct socket *so;
92 int backlog;
93 {
94     int s = splnet(), error;
95     error =
96         (*so->so_proto->pr_usrreq) (so, PRU_LISTEN,
97         (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
98     if (error) {
99         splx(s);
100         return (error);
101     }
102     if (so->so_q == 0)
103         so->so_options |= SO_ACCEPTCONN;
104     if (backlog < 0)
105         backlog = 0;
106     so->so_qlimit = min(backlog, SOMAXCONN);
107     splx(s);
108     return (0);
109 }

```

uipc_socket.c

图15-24 solisten 函数

15.10 tsleep和wakeup函数

当一个在内核中执行的进程因为得不到内核资源而不能继续执行时，它就调用 tsleep等待。tsleep的原型是：

```
int tsleep(caddr_t chan, int pri, char *mesg, int timeo);
```

tsleep的第一个参数chan，被称之为等待通道。它标志进程等待的特定资源或事件。许多进程能同时在同一个等待通道上睡眠。当资源可用或事件出现时，内核调用 wakeup，并将等待通道作为唯一的参数传入。wakeup的原型是：

```
void wakeup(caddr_t chan);
```

所有等待在该通道上的进程均被唤醒，并被设置成运行状态。当每一个进程均恢复执行时，内核安排tsleep返回。

当进程被唤醒时，tsleep的第二个参数pri指定被唤醒进程的优先级。pri中还包括几个用于tsleep的可选的控制标志。通过设置 pri中的PCATCH标志，当一个信号出现时，tsleep也返回。mesg是一个说明调用tsleep的字符串，它将被放在调用报文或ps的输出中。

*timeo*设置睡眠间隔的上限值，其单位是时钟滴答。

图15-25列出了`tsleep`的返回值。

因为所有等待在同一等待通道上的进程均被`wakeup`唤醒，所以我们总是看到在一个循环中调用`tsleep`。每一个被唤醒的进程在继续执行之前必须检查等待的资源是否可得到，因为另一个被唤醒的进程可能已经先一步得到了资源。如果仍然得不到资源，进程再调用`tsleep`等待。

tsleep()	描 述
0	进程被一个匹配的 <code>wakeup</code> 唤醒
EWOLDBLOCK	进程在睡眠 <code>timeo</code> 个时钟滴答后，在匹配的 <code>wakeup</code> 调用之前被唤醒
ERESTART	在睡眠期间信号被进程处理，应重新启动挂起的系统调用
EINTR	在睡眠期间信号被进程处理，挂起的系统调用失败

图15-25 `tsleep` 的返回值

多个进程在一个插口上睡眠等待的情况是不多见的，所以，通常情况下，一次调用`wakeup`只有一个进程被内核唤醒。

关于睡眠和唤醒机制的详细讨论请参考 [Leffler et al. 1989]。

举例

多个进程在同一个等待通道上睡眠的一个例子是：让多个服务器进程读同一个 UDP插口。每一个服务器都调用`recvfrom`，并且只要没有数据可读就在`tsleep`中等待。当一个数据报到达插口时，插口层调用`wakeup`，所有等待进程均被放入运行队列。第一个运行的服务器读取了数据报而其他的服务器则再次调用`tsleep`。在这种情况下，不需要每一个数据报启动一个新的进程，就可将进入的数据报分发到多个服务器。这种技术同样可以用来处理 TCP的连接请求，只需让多个进程在同一个插口上调用`accept`。这种技术在[Comer and Stevens 1993]中描述。

15.11 `accept`系统调用

调用`listen`后，进程调用`accept`等待连接请求。`accept`返回一个新的描述符，指向一个连接到客户的新的插口。原来的插口 `s` 仍然是未连接的，并准备接收下一个连接。如果 `name` 指向一个正确的缓存，`accept` 就会返回对方的地址。

处理连接的细节由与插口相关联的协议来完成。对于TCP而言，当一条连接已经被建立(即，三次握手已经完成)时，就通知插口层。对于其他的协议，如 OSI的TP4，只要一个连接请求到达，`tsleep`就返回。当进程通过在插口上发送或接收数据来显式证实连接后，连接则算完成。

图15-26说明`accept`的实现。

```
106 struct accept_args {
107     int    s;
108     caddr_t name;
109     int    *anamelen;
110 };
111
112 accept(p, uap, retval)
113 struct proc *p;
114 struct accept_args *uap;
115 int    *retval;
```

— *uipc_syscalls.c*

图15-26 `accept` 系统调用

```

115 {
116     struct file *fp;
117     struct mbuf *nam;
118     int     namelen, error, s;
119     struct socket *so;

120     if (uap->name && (error = copyin((caddr_t) uap->anamelen,
121                                     (caddr_t) & namelen, sizeof(namelen))))
122         return (error);
123     if (error = getsock(p->p_fd, uap->s, &fp))
124         return (error);
125     s = splnet();
126     so = (struct socket *) fp->f_data;
127     if ((so->so_options & SO_ACCEPTCONN) == 0) {
128         splx(s);
129         return (EINVAL);
130     }
131     if ((so->so_state & SS_NBIO) && so->so_qlen == 0) {
132         splx(s);
133         return (EWOULDBLOCK);
134     }
135     while (so->so_qlen == 0 && so->so_error == 0) {
136         if (so->so_state & SS_CANTRCVMORE) {
137             so->so_error = ECONNABORTED;
138             break;
139         }
140         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
141                             netcon, 0)) {
142             splx(s);
143             return (error);
144         }
145     }
146     if (so->so_error) {
147         error = so->so_error;
148         so->so_error = 0;
149         splx(s);
150         return (error);
151     }
152     if (error = falloc(p, &fp, retval)) {
153         splx(s);
154         return (error);
155     }
156     { struct socket *aso = so->so_q;
157       if (soqremque(aso, 1) == 0)
158         panic("accept");
159       so = aso;
160     }

161     fp->f_type = DTYPE_SOCKET;
162     fp->f_flag = FREAD | FWRITE;
163     fp->f_ops = &socketops;
164     fp->f_data = (caddr_t) so;
165     nam = m_get(M_WAIT, MT_SONAME);
166     (void) soaccept(so, nam);
167     if (uap->name) {
168         if (namelen > nam->m_len)
169             namelen = nam->m_len;
170         /* SHOULD COPY OUT A CHAIN HERE */
171         if ((error = copyout(mtod(nam, caddr_t), (caddr_t) uap->name,
172                                (u_int) namelen)) == 0)

```

图15-26 (续)

```
173         error = copyout((caddr_t) & namelen,  
174                         (caddr_t) uap->anamelen, sizeof(*uap->anamelen));  
175     }  
176     m_freem(nam);  
177     splx(s);  
178     return (error);  
179 }
```

uipc_syscalls.c

图15-26 (续)

106-114 `accept`有三个参数：`s`为插口描述符；`name`为缓存指针，`accept`将把外部主机的运输地址填入该缓存；`anamelen`是一个保存缓存大小的指针。

1. 验证参数

116-134 `accept`将缓存大小(`*anamelen`)赋给`namelen`，`getsock`返回插口的file结构。如果插口还没有准备好接收连接（即，还没有调用`listen`），或已经请求了非阻塞的I/O，且没有连接被送入队列，则分别返回`EINVAL`或`EWOULDBLOCK`。

2. 等待连接

135-145 当出现下列情况时，`while`循环退出：有一条连接到达；出现差错；或插口不能再接收数据。当信号被捕获之后（`tsleep`返回`EINTR`），`accept`并不自动重新启动。当协议层通过`sonewconn`将一条连接插入队列后，唤醒进程。

在循环内，进程在`tsleep`中等待，当有连接到达时，`tsleep`返回0。如果`tsleep`被信号中断或插口被设置成非阻塞，则`accept`返回`EINTR`或`EWOULDBLOCK`（图15-25）。

3. 异步差错

146-151 如果进程在睡眠期间出现差错，则将插口中的差错代码赋给`accept`中的返回码，清除插口中的差错码后，`accept`返回。

异步事件改变插口状态是比较常见的。协议处理层通过设置`so_error`或唤醒在插口上等待的所有进程来通知插口层插口状态的改变。因为这一点，插口层必须在每次被唤醒后检查`so_error`，查看是否在进程睡眠期间有差错出现。

4. 将插口同描述符相关联

152-164 `falloc`为新的连接分配一个描述符；调用`soqremque`将插口从接收队列中删除，放到描述符的file结构中。习题15.4讨论调用`panic`。

5. 协议处理

167-179 `accept`分配一个新的mbuf来保存外部地址，并调用`soaccept`来完成协议处理。在连接处理期间产生的新的插口的分配和排队在第15.12中描述。如果进程提供了一个缓存来接收外部地址，`copyout`将地址和地址长度分别从`nam`和`namelen`中复制给进程。如果有必要，`copyout`还可能将地址截掉，如果进程提供的缓存不够大。最后，释放mbuf，使能协议处理，`accept`返回。

因为仅仅分配了一个mbuf来存放外部地址，运输地址必须能放入一个mbuf中。因为Unix域地址是文件系统中的路径名（最长可达1023个字节），所以要受到这个限制，但这对Internet域中的16字节长的`sockaddr_in`地址没有影响。第170行的注释说明可以通过分配和复制一个mbuf链的方式来去掉这个限制。

soaccept函数

`soaccept`函数通过协议层获得新的连接的客户地址，如图15-27所示。

```

184 soaccept(so, nam)
185 struct socket *so;
186 struct mbuf *nam;
187 {
188     int s = splnet();
189     int error;

190     if ((so->so_state & SS_NOFDREF) == 0)
191         panic("soaccept: !NOFDREF");
192     so->so_state &= ~SS_NOFDREF;
193     error = (*so->so_proto->pr_usrreq) (so, PRU_ACCEPT,
194                                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
195     splx(s);
196     return (error);
197 }

```

uipc_socket.c

图15-27 soaccept 函数

184-197 soaccept 确保插口与一个描述符相连，并发送 PRU_ACCEPT 请求给协议。pr_usrreq 返回后，nam 中包含外部插口的名字。

15.12 sonewconn 和 soisconnected 函数

从图15-26中可以看出，accept 等待协议层处理进入的连接请求，并且将它们放入 so_q 中。图15-28利用TCP来说明这个过程。

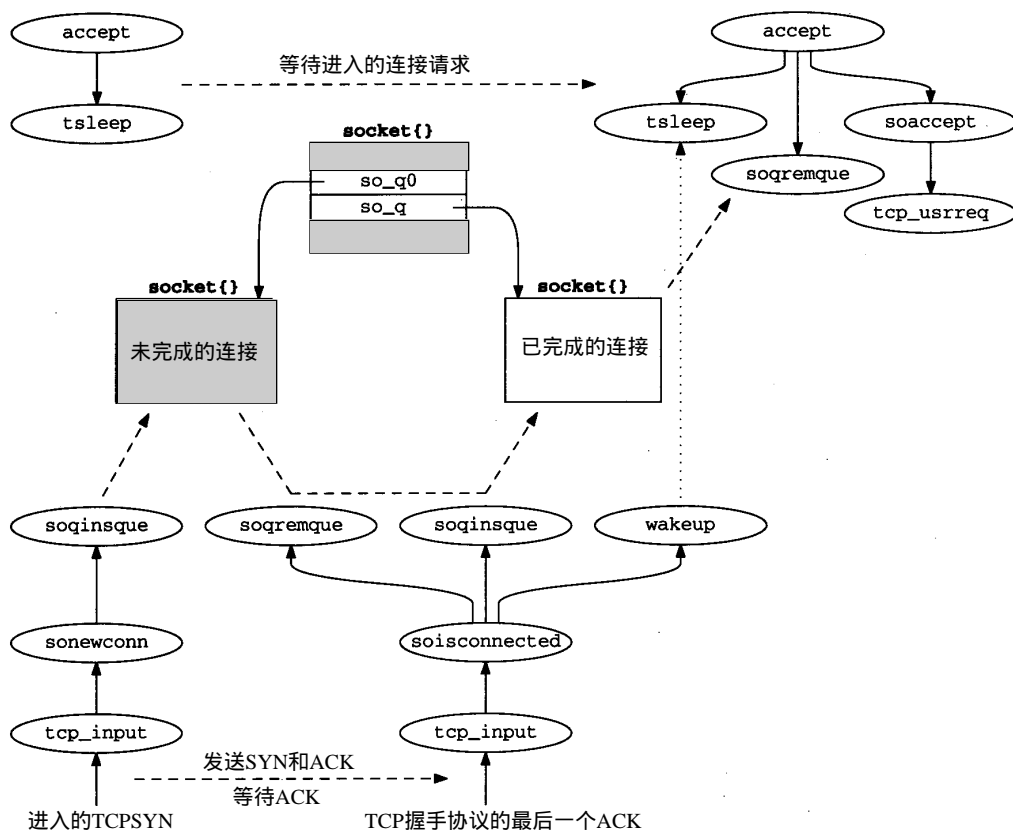


图15-28 处理进入的TCP连接

在图15-28的左上角，accept调用tsleep等待进入的连接。在左下角，tcp_input调用sonewconn为新的连接产生一个插口来处理进入的TCP SYN(图28-7)。sonewconn将产生的插口放入so_q0排队，因为三次握手还没有完成。

当TCP握手协议的最后一个ACK到达时，tcp_input调用soisconnected(图29-2)来更新产生的插口，并将它从so_q0中移到so_q中，唤醒所有调用accept等待进入的连接进程。

图的右上角说明我们在图15-26中描述的函数。当tsleep返回时，accept从so_q中得到连接，发送PRU_ATTACH请求。插口同一个新的文件描述符建立了联系，accept也返回到调用进程。

图15-29显示了sonewconn函数。

```

123 struct socket *
124 sonewconn(head, connstatus)
125 struct socket *head;
126 int connstatus;
127 {
128     struct socket *so;
129     int soqueue = connstatus ? 1 : 0;
130
131     if (head->so_qlen + head->so_q0len > 3 * head->so_qlimit / 2)
132         return ((struct socket *) 0);
133     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_DONTWAIT);
134     if (so == NULL)
135         return ((struct socket *) 0);
136     bzero((caddr_t) so, sizeof(*so));
137     so->so_type = head->so_type;
138     so->so_options = head->so_options & ~SO_ACCEPTCONN;
139     so->so_linger = head->so_linger;
140     so->so_state = head->so_state | SS_NOFDREF;
141     so->so_proto = head->so_proto;
142     so->so_timeo = head->so_timeo;
143     so->so_pgid = head->so_pgid;
144     (void) soreserve(so, head->so_snd.sb_hiwat, head->so_rcv.sb_hiwat);
145     soqinsque(head, so, soqueue);
146     if ((*so->so_proto->pr_usrreq) (so, PRU_ATTACH,
147         (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0)) {
148         (void) soqremque(so, soqueue);
149         (void) free((caddr_t) so, M_SOCKET);
150         return ((struct socket *) 0);
151     }
152     if (connstatus) {
153         sorwakeup(head);
154         wakeup((caddr_t) & head->so_timeo);
155         so->so_state |= connstatus;
156     }
157     return (so);
158 }

```

uipc_socket2.c

uipc_socket2.c

图15-29 sonewconn 函数

123-129 协议层将head(指向正在接收连接的插口的指针)和connstatus(指示新连接的状态的标志)传给sonewconn。对于TCP而言，connstatus总是等于0。

对于TP4，connstatus总是等于SS_ISCONFIRMING。当一个进程开始从插口上接收或发送数据时隐式证实连接。

1. 限制进入的连接

130-131 当下面的不等式成立时，`sonewconn`不再接收任何连接：

$$\text{so_qlen} + \text{so_q0len} > \frac{3 \times \text{so_qlimit}}{2}$$

这个不等式为一直没有完成的连接提供了一个令人费解的因子，且该不等式确保 `listen(fd, 0)` 允许一条连接。有关这个不等式的详细情况请参考卷 1 的图 18-23。

2. 分配一个新的插口

132-143 一个新的 `socket` 结构被分配和初始化。如果进程对处理接收连接状态的插口调用了 `setsockopt`，则新产生的 `socket` 继承好几个插口选项，因为 `so_options`、`so_linger`、`so_pgid` 和 `sb_hiwat` 的值被复制到新的 `socket` 结构中。

3. 排队连接

144 在第 129 行的代码中，根据 `connstatus` 的值设置 `soqueue`。如果 `soqueue` 为 0（如，TCP 连接），则将新的插口插入到 `so_q0` 中；若 `connstatus` 等于非 0 值，则将其插入到 `so_q` 中（如，TP4 连接）。

4. 协议处理

145-150 发送 `PRU_ATTACH` 请求，启动协议层对新的连接的处理。如果处理失败，则将插口从队列中删除并丢弃，然后 `sonewconn` 返回一个空指针。

5. 唤醒进程

151-157 如果 `connstatus` 等于非 0 值，所有在 `accept` 中睡眠或查询插口的可读性的进程均被唤醒。将 `connstatus` 对 `so_state` 执行或操作。TCP 协议从来不会执行这段代码，因为对 TCP 而言，`connstatus` 总是等于 0。

某些将进入的连接首先插入 `so_q0` 队列中的协议在连接建立阶段完成时调用 `soisconnected`，如 TCP。对于 TCP，当第二个 SYN 被应答时，就出现这种情况。

图 15-30 显示了 `soisconnected` 的代码。

```

78 soisconnected(so)
79 struct socket *so;
80 {
81     struct socket *head = so->so_head;
82     so->so_state &= ~(SS_ISCONNECTING | SS_ISDISCONNECTING | SS_ISCONFIRMING);
83     so->so_state |= SS_ISCONNECTED;
84     if (head && soqremque(so, 0)) {
85         soqinsque(head, so, 1);
86         sorwakeup(head);
87         wakeup((caddr_t) & head->so_timeo);
88     } else {
89         wakeup((caddr_t) & so->so_timeo);
90         sorwakeup(so);
91         sowwakeup(so);
92     }
93 }

```

—kern/uipc_socket2.c

—kern/uipc_socket2.c

图 15-30 `soisconnected` 函数

6. 排队未完成的连接

78-87 通过修改插口的状态来表明连接已经完成。当对进入的连接调用 `soisconnected`

(即，本地进程正在调用 `accept`) 时，`head` 为非空。

如果 `soqremque` 返回 1，就将插口放入 `so_q` 排队，`sorwakeup` 唤醒通过调用 `select` 测试插口的可读性来监控插口上连接到达的进程。如果进程在 `accept` 中因等待连接而阻塞，则 `wakeup` 使得相应的 `tsleep` 返回。

7. 唤醒等待新连接的进程

88-93 如果 `head` 为空，就不需要调用 `soqremque`，因为进程用 `connect` 系统调用初始化连接，且插口不在队列中。如果 `head` 非空，且 `soqremque` 返回 0，则插口已经在 `so_q` 队列中。在某些协议中，如 `TP4`，就出现这种情况，因为在 `TP4` 中，连接完成之前就已插入到 `so_q` 队列中。`wakeup` 唤醒所有阻塞在 `connect` 中的进程，`sorwakeup` 和 `sowakeup` 负责唤醒那些调用 `select` 等待连接完成的进程。

15.13 connect 系统调用

服务器进程调用 `listen` 和 `accept` 系统调用等待远程进程初始化连接。如果进程想自己初始化一条连接(即客户端)，则调用 `connect`。

对于面向连接的协议如 `TCP`，`connect` 建立一条与指定的外部地址的连接。如果进程没有调用 `bind` 来绑定地址，则内核选择并且隐式地绑定一个地址到插口。

对于无连接协议如 `UDP` 或 `ICMP`，`connect` 记录外部地址，以便发送数据报时使用。任何以前的外部地址均被新的地址所代替。

图15-31显示了 `UDP` 或 `TCP` 调用 `connect` 时涉及到的函数。

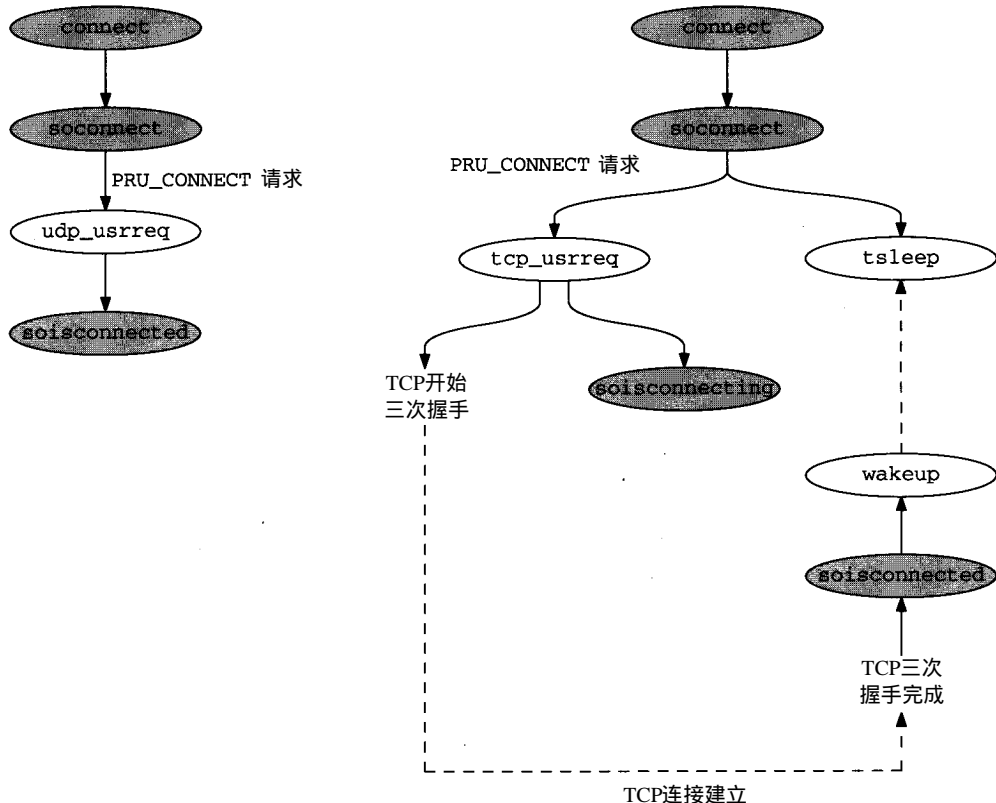


图15-31 connect 处理过程

图的左边说明 connect 如何处理无连接协议，如 UDP。在这种情况下，协议层调用 soisconnected 后 connect 系统调用立即返回。

图的右边说明 connect 如何处理面向连接的协议，如 TCP。在这种情况下，协议层开始建立连接，调用 soisconnecting 指示连接将在某个时候完成。如果插口是非阻塞的，soconnect 调用 tsleep 等待连接完成。对于 TCP，当三次握手完成时，协议层调用 soisconnected 将插口标识为已连接，然后调用 wakeup 唤醒等待的进程，从而完成 connect 系统调用。

图15-32列出了 connect 系统调用的代码。

uipc_syscalls.c

```

180 struct connect_args {
181     int      s;
182     caddr_t  name;
183     int      namelen;
184 };

185 connect(p, uap, retval)
186 struct proc *p;
187 struct connect_args *uap;
188 int      *retval;
189 {
190     struct file *fp;
191     struct socket *so;
192     struct mbuf *nam;
193     int      error, s;

194     if (error = getsock(p->p_fd, uap->s, &fp))
195         return (error);
196     so = (struct socket *) fp->f_data;
197     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING))
198         return (EALREADY);
199     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
200         return (error);

201     error = soconnect(so, nam);
202     if (error)
203         goto bad;
204     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING)) {
205         m_freem(nam);
206         return (EINPROGRESS);
207     }
208     s = splnet();
209     while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
210         if (error = tsleep((caddr_t) &so->so_timeo, PSOCK | PCATCH,
211                             netcon, 0))
212             break;
213     if (error == 0) {
214         error = so->so_error;
215         so->so_error = 0;
216     }
217     splx(s);
218 bad:
219     so->so_state &= ~SS_ISCONNECTING;
220     m_freem(nam);
221     if (error == ERESTART)
222         error = EINTR;
223     return (error);
224 }

```

uipc_syscalls.c

图15-32 connect 系统调用

180-188 connect的三个参数(在connect_args结构中)是：s为插口描述符；name是一个指针，指向存放外部地址的缓存；namelen为缓存的长度。

189-200 getsock获取插口描述符对应的file结构。可能已有连接请求在非阻塞的插口上，若出现这种情况，则返回EALREADY。函数sockargs将外部地址从进程复制到内核。

1. 开始连接处理

201-208 连接是从调用soconnect开始的。如果soconnect报告差错出现，connect跳转到bad。如果soconnect返回时连接还没有完成且使能了非阻塞的I/O，则立即返回EINPROGRESS以免等待连接完成。因为通常情况下，建立连接要涉及同远程系统交换几个分组，因而这个过程可能需要一些时间才能完成。如果连接没完成，则下次对connect调用就返回EALREADY。当连接完成时，soconnect返回EISCONN。

2. 等待连接建立

208-217 while循环直到连接已建立或出现差错时才退出。splnet防止connect在测试插口状态和调用tsleep之间错过wakeup。循环完成后，error包含0、tsleep中的差错代码或插口中的差错代码。

218-224 清除SS_ISCONNECTING标志，因为连接已完成或连接请求已失败。释放存储外部地址的mbuf，返回差错代码。

15.13.1 soconnect函数

soconnect函数确保插口处于正确的连接状态。如果插口没有连接或连接没有被挂起，则连接请求总是正确的。如果插口已经连接或连接正等待处理，则新的连接请求将被面向连接的协议(如TCP)拒绝。对于无连接协议，如UDP，多个连接是允许的，但是每一个新的请求

```

198 soconnect(so, nam)                                     — uipc_socket.c
199 struct socket *so;
200 struct mbuf *nam;
201 {
202     int      s;
203     int      error;
204     if (so->so_options & SO_ACCEPTCONN)
205         return (EOPNOTSUPP);
206     s = splnet();
207     /*
208      * If protocol is connection-based, can only connect once.
209      * Otherwise, if connected, try to disconnect first.
210      * This allows user to disconnect by connecting to, e.g.,
211      * a null address.
212      */
213     if (so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING) &&
214         ((so->so_proto->pr_flags & PR_CONNREQUIRED) ||
215          (error = sodisconnect(so))))
216         error = EISCONN;
217     else
218         error = (*so->so_proto->pr_usrreq) (so, PRU_CONNECT,
219                                           (struct mbuf *) 0, nam, (struct mbuf *) 0);
220     splx(s);
221     return (error);
222 }

```

— uipc_socket.c

图15-33 soconnect 函数

中的外部地址会取代原来的外部地址。

图15-33列出了 `soconnect` 函数的代码。

198-222 如果插口被标识准备接收连接, 则 `soconnect` 返回 `EOPNOTSUPP`, 因为如果已经对插口调用了 `listen`, 则进程不能再初始化连接。如果协议是面向连接的, 且一条连接已经被初始化, 则返回 `EISCONN`。对于无连接协议, 任何已有的同外部地址的联系都被 `sodisconnect` 切断。

`PRU_CONNECT` 请求启动相应的协议处理来建立连接或关联。

15.13.2 切断无连接插口和外部地址的关联

对于无连接协议, 可以通过调用 `connect`, 并传入一个不正确的 `name` 参数, 如指向内容为全0的结构指针或大小不对的结构, 来丢弃同插口相关联的外部地址。 `sodisconnect` 删除同插口相关联的外部地址, `PRU_CONNECT` 返回差错代码, 如 `EAFNOSUPPORT` 或 `EADDRNOTAVAIL`, 留下没有外部地址的插口。这种方式虽然有点晦涩, 但却是一种比较有用的断连方式, 在无连接插口和外部地址之间断连, 而不是替换。

15.14 shutdown系统调用

`shutdown` 系统调用关闭连接的读通道、写通道或读写通道, 如图 15-34所示。对于读通道, `shutdown` 丢弃所有进程还没有读走的数据以及调用 `shutdown` 之后到达的数据。对于写通道, `shutdown` 使协议作相应的处理。对于 TCP, 所有剩余的数据将被发送, 发送完成后发送 FIN。这就是 TCP 的半关闭特点(参考卷1的第18.5节)。

为了删除插口和释放描述符, 必须调用 `close`。可以在没有调用 `shutdown` 的情况下, 直接调用 `close`。同所有描述符一样, 当进程结束时, 内核将调用 `close`, 关闭所有还没有被关闭的插口。

```

550 struct shutdown_args {                                     uipc_syscalls.c
551     int      s;
552     int      how;
553 };

554 shutdown(p, uap, retval)
555 struct proc *p;
556 struct shutdown_args *uap;
557 int      *retval;
558 {
559     struct file *fp;
560     int      error;

561     if (error = getsock(p->p_fd, uap->s, &fp))
562         return (error);
563     return (sosshutdown((struct socket *) fp->f_data, uap->how));
564 }

```

uipc_syscalls.c

图15-34 `shutdown` 系统调用

550-557 在 `shutdown_args` 结构中, `s` 为插口描述符, `how` 指明关闭连接的方式。图 15-35列出了 `how` 和 `how++` (在图15-36中用到的) 的期望值。

how	how++	描 述
0	<i>FREAD</i>	关闭连接的读通道
1	<i>FWRITE</i>	关闭连接的写通道
2	<i>FREAD/FWRITE</i>	关闭连接的读写通道

图15-35 shutdown 系统调用选项

注意，在how和常数FREAD和FWRITE之间有一种隐含的数值关系。

558-564 shutdown是函数soshutdown的包装函数(wrapper function)。由getsock返回与描述符相关联的插口，调用soshutdown，并返回其值。

soshutdown和sorflush函数

关闭连接的读通道是由插口层调用 sorflush处理的，写通道的关闭是由协议层的PRU_SHUTDOWN请求处理的。soshutdown函数如图15-36所示。

```
720 soshutdown(so, how)                                     uipc_socket.c
721 struct socket *so;
722 int      how;
723 {
724     struct protosw *pr = so->so_proto;
725     how++;
726     if (how & FREAD)
727         sorflush(so);
728     if (how & FWRITE)
729         return ((*pr->pr_usrreq) (so, PRU_SHUTDOWN,
730             (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0));
731     return (0);
732 }
```

图15-36 soshutdown 函数

720-732 如果是关闭插口的读通道，则 sorflush丢弃插口接收缓存中的数据，禁止读连接(如图15-37所示)。如果是关闭插口的写通道，则给协议发送 PRU_SHUTDOWN请求。

733-747 进程等待给接收缓存加锁。因为 SB_NOINTR被设置，所以当中断出现时，sblock并不返回。在修改插口状态时，splimp阻塞网络中断和协议处理，因为协议层在接收到进入的分组时可能要访问接收缓存。

socantrcvmore标识插口拒绝接收进入的分组。将 sockbuf结构保存在asb中，当splx恢复中断后，要使用asb。调用bzero清除原始的sockbuf结构，使得接收队列为空。

释放控制mbuf

748-751 当shutdown被调用时，存储在接收队列中的控制信息可能引用了一些内核资源。通过sockbuf结构的副本中的sb_mb仍然可以访问mbuf链。

如果协议支持访问权限，且注册了一个dom_dispose函数，则调用该函数来释放这些资源。

在Unix域中，用控制报文在进程间传递描述符是可能的。这些报文包含一些引用计数的数据结构的指针。dom_dispose函数负责去掉这些引用，如果必要，还释放相关的数据缓存以避免产生一些未引用的结构和导致内存漏洞。有关在 Unix域内传递文件描述符的细节请参考 [Stevens 1990]和[Leffler et al. 1989]。

```

733 sorflush(so)                                     -----uipc_socket.c
734 struct socket *so;
735 {
736     struct sockbuf *sb = &so->so_rcv;
737     struct protosw *pr = so->so_proto;
738     int s;
739     struct sockbuf asb;

740     sb->sb_flags |= SB_NOINTR;
741     (void) sblock(sb, M_WAITOK);
742     s = splimp();
743     socanttrcmore(so);
744     sbunlock(sb);
745     asb = *sb;
746     bzero((caddr_t) sb, sizeof(*sb));
747     splx(s);

748     if (pr->pr_flags & PR_RIGHTS && pr->pr_domain->dom_dispose)
749         (*pr->pr_domain->dom_dispose) (asb.sb_mb);
750     sbrelease(&asb);
751 }
-----uipc_socket.c

```

图15-37 sorflush 函数

当sbrelease释放接收队列中的所有mbuf时，丢弃所有调用shutdown时还没有被处理的数据。

注意，连接的读通道的关闭完全由插口层来处理（习题15.6），连接的写通道的关闭通过发送PRU_SHUTDOWN请求交由协议处理。TCP协议收到PRU_SHUTDOWN请求后，发送所有排队的数据，然后发送一个FIN来关闭TCP连接的写通道。

15.15 close系统调用

close系统调用能用来关闭各类描述符。当fd是引用对象的最后的描述符时，与对象有关的close函数被调用：

```
error = (*fp->f_ops->fo_close)(fp,p);
```

如图15-13所示，插口的fp->f_ops->fo_close是soo_close函数。

15.15.1 soo_close函数

soo_close函数是soclose函数的封装器，如图15-38所示。

```

152 soo_close(fp, p)                                     -----sys_socket.c
153 struct file *fp;
154 struct proc *p;
155 {
156     int error = 0;

157     if (fp->f_data)
158         error = soclose((struct socket *) fp->f_data);
159     fp->f_data = 0;
160     return (error);
161 }
-----sys_socket.c

```

图15-38 soo_close 函数

152-161 如果socket结构与file相关联,则调用soclose,清除f_data,返回已出现的差错。

15.15.2 soclose函数

soclose函数取消插口上所有未完成的连接(即,还没有完全被进程接受的连接),等待数据被传输到外部系统,释放不需要的数据结构。

soclose函数的代码如图15-39所示。

```

130 struct socket *so;
131 {
132     int      s = splnet();          /* conservative */
133     int      error = 0;

134     if (so->so_options & SO_ACCEPTCONN) {
135         while (so->so_q0)
136             (void) soabort(so->so_q0);
137         while (so->so_q)
138             (void) soabort(so->so_q);
139     }
140     if (so->so_pcb == 0)
141         goto discard;
142     if (so->so_state & SS_ISCONNECTED) {
143         if ((so->so_state & SS_ISDISCONNECTING) == 0) {
144             error = sodisconnect(so);
145             if (error)
146                 goto drop;
147         }
148         if (so->so_options & SO_LINGER) {
149             if ((so->so_state & SS_ISDISCONNECTING) &&
150                 (so->so_state & SS_NBIOS))
151                 goto drop;
152             while (so->so_state & SS_ISCONNECTED)
153                 if (error = tsleep((caddr_t) & so->so_timeo,
154                                     PSOCK | PCATCH, netcls, so->so_linger))
155                     break;
156         }
157     }
158 drop:
159     if (so->so_pcb) {
160         int      error2 =
161             (*so->so_proto->pr_usrreq) (so, PRU_DETACH,
162             (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
163         if (error == 0)
164             error = error2;
165     }
166 discard:
167     if (so->so_state & SS_NOFDREF)
168         panic("soclose: NOFDREF");
169     so->so_state |= SS_NOFDREF;
170     sofree(so);
171     splx(s);
172     return (error);
173 }

```

uipc_socket.c

图15-39 soclose 函数

1. 丢弃未完成的连接

129-141 如果插口正在接收连接，`soclose`遍历两个连接队列，并且调用`soabort`取消每一个挂起的连接。如果协议控制块为空，则协议已同插口分离，`soclose`跳转到`discard`进行退出处理。

`soabort`发送`PRU_ABORT`请求给协议，并返回结果。本书中没有介绍`soabort`的代码。图23-38和图30-7讨论了UDP和TCP如何处理`PRU_ABORT`请求。

2. 断开已建立的连接或关联

142-157 如果插口没有同任何外部地址相连接，则跳转到`drop`处继续执行。否则，必须断开插口与对等地址之间的连接。如果断连没有开始，则`sodisconnect`启动断连进程。如果设置了`SO_LINGER`插口选项，`soclose`可能要等到断连完成后才返回。对于一个非阻塞的插口，从来不需要等待断连完成，所以在这种情况下，`soclose`立即跳转到`drop`。否则，连接终止正在进行且`SO_LINGER`选项指示`soclose`必须等待一段时间才能完成操作。直到出现下列情况时`while`才退出：断连完成；拖延时间(`so_linger`)到；或进程收到了一个信号。

如果滞留时间被设为0，`tsleep`仅当断连完成(也许因为一个差错)或收到一个信号时才返回。

3. 释放数据结构

158-173 如果插口仍然同协议相连，则发送`PRU_DETACH`请求断开插口与协议的联系。最后，插口被标记为同任何描述符没有关联，这意味着可以调用`sofree`释放插口。

`sofree`函数代码如图15-40所示。

```

110 sofree(so)
111 struct socket *so;
112 {
113     if (so->so_pcb || (so->so_state & SS_NOFDREF) == 0)
114         return;
115     if (so->so_head) {
116         if (!soqremque(so, 0) && !soqremque(so, 1))
117             panic("sofree dq");
118         so->so_head = 0;
119     }
120     sbrelease(&so->so_snd);
121     sorflush(so);
122     FREE(so, M_SOCKET);
123 }

```

— *uipc_socket.c*

— *uipc_socket.c*

图15-40 `sofree` 函数

4. 如果插口仍在用则返回

110-114 如果仍然有协议同插口相连，或如果插口仍然同描述符相连，则`sofree`立即返回。

5. 从连接队列中删除插口

115-119 如果插口仍在连接队列上(`so_head`非空)，则插口的队列应该为空。如果不空，则插口代码和内核`panic`中有差错。如果队列为空，清除`so_head`。

6. 释放发送和接收队列中的缓存

120-123 `sorelease`释放发送队列中的所有缓存, `sorflush`释放接收队列中的所有缓存。最后, 释放插口本身。

15.16 小结

本章中我们讨论了所有与网络操作有关的系统调用。描述了系统调用机制, 并且跟踪系统调用直到它们通过 `pr_usrreq` 函数进入协议处理层。

在讨论插口层时, 我们避免涉及地址格式、协议语义或协议实现等问题。在接下来的章节中, 我们将通过协议处理层中的 Internet 协议的实现将链路层处理和插口层处理联系在一起。

习题

- 15.1 一个没有超级用户权限的进程怎样才能获取对超级用户进程产生的插口的访问权?
- 15.2 一个进程怎样才能判断它提供给 `accept` 的 `sockaddr` 缓存是不是太小以至不能存放调用返回的外部地址?
- 15.3 IPv6 的插口有一个特点: 使 `accept` 和 `recvfrom` 返回一个 128 bit 的 IPv6 地址的数组作为源路由, 而不是仅返回一个对等地址。因为数组不能存放在一个 `mbuf` 中, 所以修改 `accept` 和 `recvfrom`, 使得它们能够处理协议层来的 `mbuf` 链而不是仅仅一个 `mbuf`。如果协议在 `mbuf` 簇中返回一个数组而不是一个 `mbuf` 链, 已有的代码仍然能正常工作吗?
- 15.4 为什么在图 15-26 中当 `soqremque` 返回一个空指针时要调用 `panic`?
- 15.5 为什么 `sorflush` 要复制接收缓存?
- 15.6 在 `sorflush` 将插口的接收缓存清 0 后, 如果还有数据到达会出现什么现象? 在做这个习题之前请阅读第 16 章的内容。