世行贷款

21世纪初高等教育教学改革项目

立项指南

教育部高等教育司

高等教育出版社

图书在版编目（CIP）数据

Linux 内核源代码情景分析. 下册 / 毛德操，胡希明著. 杭州：浙江大学出版社，2001.9
ISBN 7-308-02704-X

Ⅰ.L... Ⅱ.①毛...②胡... Ⅲ.Linux 操作系统—程序分析 Ⅳ.TP316.89

中国版本图书馆 CIP 数据核字（2001）第 027485 号

本书限中国大陆地区发行

责任编辑：张　明　樊晓燕
封面设计：俞亚彤
出版发行：浙江大学出版社
（杭州浙大路 38 号　邮政编码 310027）
（E-mail:zupress@mail.hz.zj.cn）
（网址：http://www.zjupress.com）
排　　版：浙江大学出版社电脑排版中心
印　　刷：浙江印刷集团公司
开　　本：889mm×1194mm　1/16
印　　张：47
字　　数：1423 千
版、印次：2001 年 9 月第 1 版　2001 年 11 月第 2 次印刷
印　　数：3001—6000
书　　号：ISBN 7-308-02704-X/TP・209
定　　价：70.00 元

第7章

基于 socket 的进程间通信

7.1 系统调用 socket( )

相对于传统的 Unix IPC，“插口”，即 socket（有些资料中也称“套接字”），是更为一般的进程间通信机制。它既适用于同一台计算机上的进程间通信，也适用于网络环境的进程间通信，并且是当今所有网络操作系统必不可少的基础功能。本章侧重从同一计算机上的进程间通信的角度介绍 socket 在 Linux 内核中的实现。至于它在网络环境中的推广，则因分量太重、篇幅太大，只好留作另一本书的内容。

在 Unix 的发展史中，AT&T 的贝尔实验室与加州大学伯克利分校的伯克利软件发布中心（BSD）可以说是两大主力。当 AT&T 致力于改进传统的 Unix 进程间通信功能，从而形成了一整套 SysV IPC 机制的同时，BSD 也在设法对其加以改进。与此同时，BSD 又最早将计算机网络的通信规程，特别是当时正在成形的 TCP/IP 规程，实现到 Unix 的内核中去。所以，对于当时的 BSD 来说，很自然地会把二者结合在一起考虑，把同一计算机（或曰网络“节点”）上的进程间通信纳入更广的、网络范围的进程间通信范畴，从而设计出一种更为一般化的进程间通信机制。这种努力的结果就是 socket 机制，这一机制其实是命名管道在计算机网络环境下的实现和推广。

如果比较一下 AT&T 和 BSD 各自在这方面的努力，就可以看出，AT&T 是有系统地、全面地对传统的 Unix IPC 加以改进。例如，针对在某些应用中（传统 Unix IPC 的）效率不够高的缺点，设计和开发了共享内存机制；针对缺乏进程间同步手段的问题，又设计和开发了用户空间的信号量机制。另外，管道要占用打开文件号，便引入了“键值”的概念，从而避开了使用打开文件号。但是，AT&T 的眼光始终只盯着单一计算机中的进程间通信。而 BSD 则正好相反，它对传统 Unix 进程间通信的改进并不是有系统的和全面的，可是它的眼光却跳出了单机的范围。虽然 socket 机制在单机范围内与 AT&T 的报文传递机制在概念上极为相似，但是却更为一般化，从而为后来网络技术的蓬勃发展做好了技术准备。可以说，SysV IPC 和 socket 是互相补充而不是各搞一套。

至于 Linux，则很自然地兼收并蓄，把二者都继承下来了。

顾名思义，一个 socket 就好像一个通信线的插口。只要通信的双方都有插口，并且两个插口之间有通信线路相连接，就可以互相通信了。从概念上说，socket 与管道其实并无多大区别，如果把两个

插口之间的“连线”比喻成“管道”，那么插口就相当于管道两端的“水龙头”，而且二者都表现为已打开文件。不同的是管道两端只能在同一台计算机上，而通过虚拟的“通信线路”相连接的两个插口却可以分别存在于计算机网络中的不同节点上。当然，尽管二者在概念上相似，在具体的实现和使用上却有着很大的不同。而且，管道所传递的是无结构的字节流，而通过 socket 传递的则是有结构的报文。

一个插口在逻辑上有三个特征，或者说三个要素，那就是网域、类型以及规程。

首先是网域，它表明一个插口是用于哪一种网络，或者说哪一族网络规程的。由于各种网络对节点地址的命名方法不同，所以又称为“地址族”（address family）或“规程族”（protocol family）。例如，常数 AF_INET 表示互联网（英特网）插口，所以各节点都使用 IP 地址；而 AF_IPX 则为 Novell 的 IPX 网插口，AF_X25 为 X.25 网插口，等等。其中有个特例，那就是什么网也不是，只是在一台计算机上用于进程间通讯，BSD 为这种特例定义的域名为 AF_UNIX。后来，在 POSIX 标准里又定义了一个 AF_LOCAL，以示对别的操作系统也一视同仁。

其次为“类型”，它表明在网络中通信所遵循的模式。网络通信有两种主要模式，一种称为“有连接”或“面向连接”（connection oriented）的通信；另一种则称为“无连接”（connectionless）通信。“有连接”模式常常又称为“虚电路”（virtual circuit）模式。在这种模式中，通信的双方要先通过一定的步骤在互相之间建立起一种虚拟的连接，或者说虚拟的线路，然后再通过虚拟的连接线路进行通信。在通信的过程中，所有报文传递都保持着原来的次序，报文在网络中传输的过程中受到的不均匀延迟会在接收端得到补偿。所以所有报文之间都是有关联的，每个报文都不是孤立的。更重要的是，在这种模式中所有报文的传递都是“可靠”的，由网络中物理通信线路引入的差错会由通信规程中的应答和重发机制加以克服。同时，在这种模式中还提供了“流量控制”的手段。从用户的角度来看，“有连接”类型的插口对报文的传递作出了承诺。如果一个进程通过系统调用在一个“有连接”插口上发送一个报文，那么，只要进程从这次系统调用正常返回，就说明该报文已经被递交到了接收方的插口（但接收方进程未必已经读取这个报文）。“无连接”模式就不同了。“无连接”模式常常又称为“数据包”（datagram）模式，也称“面向报文”的通信模式（message oriented）。在“无连接”模式中并不需要事先在双方之间建立起“虚电路”，而直接就可以发送或接收报文，但是每个报文都是孤立的，其正确性也没有保证，甚至可能丢失。如果两个报文穿越网络时走过了不同的路径，或者甚至相同的路径，但是受到了不一致的延迟，那么它们在接收端的次序就可能与发送端的次序不同。所以，由“无连接”模式的插口所提供的报文传递是不可靠的，它对用户作出的承诺只是“尽力传递”，但却是没有保证的。此外，在“无连接”模式中也没有“流量控制”手段。如果一个进程通过系统调用在一个“无连接”插口上发送一个报文，那么当进程从这次系统调用正常返回时只是说明该插口将会例行公事地将报文传递到接收方，但并不表示这报文已经到达了接收方的插口。从另一个角度，还可以说，“有连接”插口的报文传递是同步的，而“无连接”插口的报文传递则是异步的。

最后是“规程”，它表明具体的网络规程。一般来说，网域和类型结合在一起大体上就确定了适用的规程。例如，要是网域为 AF_INET，而类型为“无连接”，则规程基本上就是 UDP 了。但是，在有些情况下还可能会有别的选择，此时就由它来进一步明确具体的规程。

其实，插口的这三个特征是互相联系的，归根结底就是反映了一个插口所运行的（网络）规程。由于在每个插口的后面都隐藏着网络规程，对插口的比较详尽的讨论就势必要涉及到计算机网络这个课题。要在本书中谈论计算机网络，并进而分析 Linux 内核中有关网络规程的代码是不现实的，因为那本身就已经足够构成另一本书的材料了。所以，我们在本书中将只讨论 Unix 域的插口，也就是用于同一计算机中进程间通信的插口，隐藏在这种插口后面的规程是“没有规程”。

Unix 域的插口同样也分两种模式，但是二者都提供可靠的报文传递。在网络环境下，不可靠性是由网络的基础设施（如物理线路）引起的，两种不同模式实际上反映了对待物理介质的不可靠性的不同态度和策略。“有连接”模式采取了“大包大揽”的态度，企图在不可靠物理介质的基础上构筑起一层可靠的报文传递机制。而“无连接”模式则采取了“矛盾上交”的态度，说“既然物理介质不可靠，那我也没有办法，只能有什么给你什么”，让用户自己去设法克服或避免由此而引起的问题。可是，Unix 域的插口既然只提供同一台计算机上的进程间通信，而根本就不涉及网络设施，其物理介质实际上就是内存，那就根本不存在由物理介质引入的不可靠性。不过，尽管 Unix 域插口的两种模式都提供可靠的报文传递，二者还是有区别的，读完本节以后就会清楚这些区别。

像对 Sys V IPC 操作一样，Linux 内核为所有与 socket 有关的操作提供一个统一的系统调用入口，但是在用户程序界面上则通过 C 语言程序库 c.lib 提供诸多库函数，看起来就好像都是独立的系统调用一样。内核中为 socket 设置的总入口为 sys_socketcall( )，其代码在 net/socket.c 中：

```
1512    /*
1513     *  System call vectors.
1514     *
1515     *  Argument checking cleaned up. Saved 20% in size.
1516     *  This function doesn't need to set the kernel lock because
1517     *  it is set by the callees.
1518     */
1519
1520    asmlinkage long sys_socketcall(int call, unsigned long *args)
1521    {
1522        unsigned long a[6];
1523        unsigned long a0,a1;
1524        int err;
1525
1526        if(call<1||call>SYS_RECVMSG)
1527            return -EINVAL;
1528
1529        /* copy_from_user should be SMP safe. */
1530        if (copy_from_user(a, args, nargs[call]))
1531            return -EFAULT;
1532
1533        a0=a[0];
1534        a1=a[1];
1535
1536        switch(call)
1537        {
1538            case SYS_SOCKET:
1539                err = sys_socket(a0,a1,a[2]);
1540                break;
1541            case SYS_BIND:
1542                err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
1543                break;
1544            case SYS_CONNECT:
```

```
1545            err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
1546            break;
1547        case SYS_LISTEN:
1548            err = sys_listen(a0,a1);
1549            break;
1550        case SYS_ACCEPT:
1551            err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
1552            break;
1553        case SYS_GETSOCKNAME:
1554            err = sys_getsockname(a0,(struct sockaddr *)a1, (int *)a[2]);
1555            break;
1556        case SYS_GETPEERNAME:
1557            err = sys_getpeername(a0, (struct sockaddr *)a1, (int *)a[2]);
1558            break;
1559        case SYS_SOCKETPAIR:
1560            err = sys_socketpair(a0,a1, a[2], (int *)a[3]);
1561            break;
1562        case SYS_SEND:
1563            err = sys_send(a0, (void *)a1, a[2], a[3]);
1564            break;
1565        case SYS_SENDTO:
1566            err = sys_sendto(a0,(void *)a1, a[2], a[3],
1567                    (struct sockaddr *)a[4], a[5]);
1568            break;
1569        case SYS_RECV:
1570            err = sys_recv(a0, (void *)a1, a[2], a[3]);
1571            break;
1572        case SYS_RECVFROM:
1573            err = sys_recvfrom(a0, (void *)a1, a[2], a[3],
1574                        (struct sockaddr *)a[4], (int *)a[5]);
1575            break;
1576        case SYS_SHUTDOWN:
1577            err = sys_shutdown(a0,a1);
1578            break;
1579        case SYS_SETSOCKOPT:
1580            err = sys_setsockopt(a0, a1, a[2], (char *)a[3], a[4]);
1581            break;
1582        case SYS_GETSOCKOPT:
1583            err = sys_getsockopt(a0, a1, a[2], (char *)a[3], (int *)a[4]);
1584            break;
1585        case SYS_SENDMSG:
1586            err = sys_sendmsg(a0, (struct msghdr *) a1, a[2]);
1587            break;
1588        case SYS_RECVMSG:
1589            err = sys_recvmsg(a0, (struct msghdr *) a1, a[2]);
1590            break;
1591        default:
1592            err = -EINVAL;
```

```
1593                break;
1594        }
1595        return err;
1596    }
```

函数的第一个参数 call 即为具体的操作码，而参数 args 为指向一个数组的指针。根据具体操作码的不同，需要从用户空间复制参数的数量也不同。为了根据操作码确定需要从用户空间复制的字节数，代码 socket.c 中定义了一个数组 nargs[ ]：

```
1505    /* Argument list sizes for sys_socketcall */
1506    #define AL(x) ((x) * sizeof(unsigned long))
1507    static unsigned char nargs[18]={AL(0),AL(3),AL(3),AL(3),AL(2),AL(3),
1508                    AL(3),AL(3),AL(4),AL(4),AL(4),AL(6),
1509                    AL(6),AL(2),AL(5),AL(5),AL(3),AL(3)};
1510    #undef AL
```

至于操作码，则是在 include/linux/net.h 中定义的：

```
30    #define SYS_SOCKET      1       /* sys_socket(2)        */
31    #define SYS_BIND        2       /* sys_bind(2)          */
32    #define SYS_CONNECT     3       /* sys_connect(2)       */
33    #define SYS_LISTEN      4       /* sys_listen(2)        */
34    #define SYS_ACCEPT      5       /* sys_accept(2)        */
35    #define SYS_GETSOCKNAME 6       /* sys_getsockname(2)   */
36    #define SYS_GETPEERNAME 7       /* sys_getpeername(2)   */
37    #define SYS_SOCKETPAIR  8       /* sys_socketpair(2)    */
38    #define SYS_SEND        9       /* sys_send(2)          */
39    #define SYS_RECV        10      /* sys_recv(2)          */
40    #define SYS_SENDTO      11      /* sys_sendto(2)        */
41    #define SYS_RECVFROM    12      /* sys_recvfrom(2)      */
42    #define SYS_SHUTDOWN    13      /* sys_shutdown(2)      */
43    #define SYS_SETSOCKOPT  14      /* sys_setsockopt(2)    */
44    #define SYS_GETSOCKOPT  15      /* sys_getsockopt(2)    */
45    #define SYS_SENDMSG     16      /* sys_sendmsg(2)       */
46    #define SYS_RECVMSG     17      /* sys_recvmsg(2)       */
```

注释中括号里的“2”表示所述的函数为系统调用。

我们先把这些操作码（以及相应的函数）分一下类，然后说明它们的用途，最后再来看它们是怎样实现的。

这些操作码（函数）大体上可分为五类。

7.1.1　插口的创建与撤除

属于这一类的操作码有：

- SYS_SOCKET：创建一个插口，其用户程序界面（由 libc 提供，下同）为：

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

这里的三个参数即为前面所述插口的三个要素。不过，通常第三个参数 protocol 为 0，因为一般来说前两个参数确定以后，具体的规程也就定了，所以用 0 表示默认由系统根据前两个参数确定的规程，只有在比较特殊的应用中才需要指定具体的规程。插口创建成功以后返回一个正整数，实际上是一个打开文件号。但这个打开文件号是与一个代表插口的数据结构相联系的，并不是与磁盘上的某个文件相联系。

- SYS_BIND：将代表着一个插口的打开文件号与某一个域中的可寻址实体或“插口地址”相结合。例如，在互联网中的可寻址实体为网中以 IP 地址区分的节点，而在同一节点上又有若干不同的“传输层”收发口（逻辑意义上），所以此时的结合就是与 IP 地址加收发口逻辑编号的结合。在 Unix 域中，此种可寻址实体是一个文件名，所以就成了与文件名的结合。用户程序界面为：

```
int bind(int sockfd , struct sockaddr *my_addr, socklen_t addrlen);
```

这里的 sockfd 即为 socket( )所返回的打开文件号。在 bind 到某个地址或文件名之前，虽然插口已经建立但却无从寻访。如果说 socket( )好像是安装了一个电话机的话，那么 bind( )就好像为它指定了一个电话号码。

- SYS_SOCKETPAIR：创建一对已经互相连接的无名插口，概念上类似于 pipe( )。用户程序界面为：

```
int socketpair (int domain, int type , int protocal , int sv[2]);
```

前面三个参数与 socket( )相同，数组 sv[ ]则用来返回创建后的一对打开文件号。

- SYS_ SHUTDOWN：部分或完全关闭一个插口，用户程序界面为：

```
int shutdown (int s , int how);
```

参数 s 为插口的打开文件号，参数 how 为 0 表示不再允许接收，1 表示不再允许发送，2 则表示接收和发送都不再允许。由于插口体现为特殊的打开文件，所以也可以用 close( )来关闭。

7.1.2 插口间连接的建立

“有连接”模式通信的双方是不对称的，可以说天生就是 client/server 的关系。而连接的建立则必须经过一定的步骤：

- SYS_LISTEN：作为 server 的一方首先要通过 listen( )向内核“挂号”。只有挂了号的插口才会被内核视为 server 一侧的插口并允许它接受来自 client 一侧的连接请求。用户程序界面为：

```
int listen(int s, int backlog);
```

参数 s 即为插口的打开文件号。当连接请求到来，而一时得不到接受时，就暂时进入一个队列，在那里等待 server 方的接受。这里的参数 backlog 就规定了这个队列的最大长度。

- SYS_CONNECT：在“有连接”模式中，client 一方要通过 connect( )向已经挂了号的 server 方插口请求连接。用户程序界面为：

```
int connect (int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

参数 sockfd 为 client 一侧插口的打开文件号，serv_addr 则指向一个表示 server 方插口地址的数据结构。当然，这个地址必须已经由 server 方通过 bind( )指定给 server 方的插口。由于有些网域中的地址可能不是固定长度的，所以还要有个参数 addrlen 来指明其长度。

对 connect( )的调用一般要到 server 方接受了连接或者出错时才返回。

- SYS_ACCEPT：server 方通过 accept( )接受或等待接受到来的（或已经在队列中的）第一个连接请求。用户程序界面为：

```
int  accept(int s, struct sockaddr  *addr , socklen_t  *addrlen);
```

参数 s 为一个已经挂了号的 server 方插口（打开文件号），参数 addr 与 addrlen 则用来返回连接请求的来源，也就是 client 方插口的地址。对 accept( )调用要到有连接请求到来而建立起连接，或者发生了出错时才返回。

特别要说明的是，这个函数返回一个新的打开文件号。这个新的打开文件号就代表着已经建立起连接的 server 方插口。而原来的插口 s，则保持原样不变，又可以用来接受新的（其他的）连接请求。这个插口就好像是下蛋的鸡，每接受一个连接请求就生下一个“蛋”，那就是已经建立起连接的插口。在典型的应用中，server 进程在通过 accept( )接受了一个连接请求，从而与一个 client 进程建立起一个连接以后，就会通过 fork( )创建出一个子进程。然后，子进程将作为“种籽”的 server 方插口关闭，而使用新的插口与 client 进程通信并为之提供服务。而父进程则把新的插口关闭，并且再一次调用 accept( )，通过“种籽”插口来接受新的连接请求。这就是典型的 client/server 运行模式。

7.1.3 “有连接”模式的报文发送与接收

“有连接”模式的插口一定要在 client 和 server 双方建立起连接以后才能用于通信。由于插口在用户界面上表现为已打开文件，并且“有连接”模式的通信既是可靠的又保持原有的次序，所以可以把传递的信息看作有序的字节流，从而可以用普通的 read( )和 write( )系统调用，像读/写普通文件一样地来接收和发送信息。除此之外，也可以用专门为插口而设的三对库函数之一来接收和发送，即 recv( )/send( )、recvfrom( )/send_to( )以及 recvmsg( )/sendmsg( )。不过，这些库函数主要用于“无连接”模式，所以我们将它们与“无连接”模式的发送与接收放在一起介绍。这里还要指出，无论是“有连接”还是“无连接”模式，插口都是双向的，并且就发送和接收而言双方是对称的。

7.1.4 “无连接”模式的报文发送与接收

顾名思义，“无连接”模式的插口不需要事先建立连接，插口一经创建马上就可以发送或接收报文。另一方面，由于“无连接”模式的通信既不是可靠的，也不一定保持原有的次序，所以不宜用 read( )和 write( )像对待一个有序字节流那样使用“无连接”模式的插口，而要用专门设置的三组库函数来进行。与文件操作 read( )、write( )相比，这三组函数的特点是它们都保留报文的边界，有关这一点读者在后面看了源代码以后就清楚了。

- SYS_SENDTO：如前所述，“无连接”模式的插口一经建立（并且 bind 到一个插口地址）以后就可以进行通信，而无需先建立连接。当然，此时对所发送的每个报文都要提供对方的地址（在“无连接”模式中，每个报文都是独立的），所以，sendto( )的用户程序界面为：

```
int  sendto( int  s ,  const void * msg ,  size_t  len,  int  flags,
                const struct sockaddr  *to,  socklen_t  tolen);
```

显然，这个函数是专为“无连接”模式的报文发送而设置的，参数 to 指向对方的地址即 sockaddr 数据结构，而 tolen 则为地址的长度。虽然这个函数的界面是专为“无连接”模式设计的，但是也可以用这个函数在“有连接”模式的插口上发送报文，不过此时应将参数 to 设置成 NULL，而将 tolen 设置成 0。

在内核中，操作 SYS_SENDTO 是由 sys_sendto( )实现的。

- SYS_SEND：从用户程序界面来看，send( )似乎主要是为“有连接”模式设计的：

```
int  send(int  s,  const void  *msg ,  size_t  len ,  int  flags);
```

与 sendto( )的界面相比，这里没有提供对方的地址。在“有连接”模式中，连接已经事先建立好，当然不需要每次都提供对方地址了。但是，即使在“无连接”模式中，当准备接连向同一目标发送很多个报文时，每次都要提供对方的地址。这样做既麻烦又降低了效率（每次都要从用户空间把地址复制到内核中）。是不是可以简化一下呢？例如，是否可以先“预设”一个双方地址，随后就采用 send( )来发送，而不必每次都重复地提供相同的地址。事实正是这样，对于“无连接”模式的插口，可以用 connect( )先设置一个对方地址，然后再用 send( )发送报文，而实际上每次都使用预先设置好的对方地址。但是要注意，在“无连接”模式中使用 connect( )与在“有连接”模式中使用 connect( )有本质的区别。在“无连接”模式中，connect( )的作用只是让内核为“本地”插口记下预设的对方地址，而并不涉及与对方之间控制报文的往返。以后则在发送的每个报文头部附上这个地址，以指明报文的目的地。至于在“有连接”模式中的 connect( )，则实际向对方发送一个请求连接的控制报文（指在网络环境下），并等待对方的响应。连接建立了以后，随同每个报文发送的可以只是一个连接号，而不一定要包括对方地址。所以，虽然在形式上两种模式都可以使用 connect( )，并且有时把经过 connect( )预设好对方地址的“无连接”插口也说成是处于“已连接状态”，但实质上是完全不同的。读者在资料中碰到此类词句时要注意根据上下文加以区分。

在内核中，SYS_SEND 是由 sys_send( )实现的，而 sys_send( )又是由 sys_sendto( )实现的，代码见 net/socket.c：

[sys_socketcall( ) > sys_send( )]

```
1207    asmlinkage long sys_send(int fd, void * buff, size_t len, unsigned flags)
1208    {
1209        return sys_sendto(fd, buff, len, flags, NULL, 0);
1210    }
```

- SYS_SENDMSG：库函数 sendmsg( )是前两个函数的推广与加强，这是三个函数中最复杂的，其用户程序界面为：

```
int  sendmsg(int  s ,  const struct msghdr  *msg ,  int  flags);
```

参数 msg 指向一个 msghdr 数据结构，定义于 socket.h：

```
33    struct msghdr {
34        void        *msg_name;     /* Socket name          */
35        int         msg_namelen;   /* Length of name       */
36        struct iovec    *msg_iov;     /* Data blocks          */
37        __kernel_size_t     msg_iovlen; /* Number of blocks     */
```

```
38        void      *msg_control;    /*Per protocol magic(eg BSD file
                                                descriptor passing)*/
39        __kernel_size_t     msg_controllen; /* Length of cmsg list */
40        unsigned    msg_flags;
41    };
```

每个 msghdr 数据结构都代表着一个报文。在 msghdr 结构中，msg_name 为对方的插口地址（或者也可以称作插口名）。而 msg_iov 则指向一个结构数组，该数组中的每一个元素都是一块数据，这样一个报文的内容就可以分散在若干个互不连续的缓冲区中，而在逻辑上却连在一起，在网络环境下这是很有好处的。还有，msg_control 和 msg_controllen 的作用是传递控制信息，在 Unix 域中用来在进程间传递访问权限，还可以用来传递“打开文件描述体”。这些以后再介绍。

由于每个报文都有个对方地址，这显然适合于“无连接”模式。但是，像 sendto( )一样，它也可以用于“有连接”模式中，不过要将 msg_name 设成 NULL，msg_namelen 设成 0。

- SYS_RECV、SYS_RECVFROM、SYS_RECVMSG：这三个操作都是用来从某个插口 s 接收报文的，并且分别与前列用来发送报文的操作相对应，所以我们把它们合并在一起叙述。完成这些操作的库函数为：

```
int  recv(int  s,  void  *buf ,  size_t  len ,  int  flags);
int  recvfrom(int  s,  void  *buf ,  size_t  len ,  int  flags,
                struct sockaddr  *from ,  socklen_t  *fromlen);
int  recvmsg (int  s ,  struct msghdr  *msg ,  int  flags);
```

与发送报文的库函数类似，recv( )通常用于“有连接”模式的插口，或者虽然是“无连接”模式的插口，但却已经用 connect( )预设了对方地址。函数 recvfrom( )的接收是全方位的，参数 from 并不是用来指定接收来自哪一个远方插口的报文，而是用来在接收到一个报文时指明报文的来源。所以 from 所指向的数据结构只是一个用于返回报文来源的缓冲区，而 fromlen 所指则为该缓冲区的大小。在内核中，SYS_RECV 由 sys_recv( )实现，SYS_RECVFROM 则由 sys_recvfrom( )实现。但是，就像 sys_send( )与 sys_sendto( )之间的关系一样，sys_recv( )也是通过 sys_recvfrom( )实现的，代码见 net/socket.c：

[sys_socketcall( ) > sys_recv( )]

```
1258   asmlinkage long sys_recv(int fd, void * ubuf, size_t size, unsigned flags)
1259   {
1260       return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
1261   }
```

也就是说，如果把 recvfrom( )的参数 from 和 fromlen 设置成 NULL，就跟 recv( )一样了。函数 recvmsg( )则与 sendmsg( )相对应，也具有将一个报文的内容分散在若干个缓冲区内的功能，并且具有接收控制信息的功能。

7.1.5 控制以及辅助性的操作

- SYS_GETSOCKNAME：库函数 getsockname( )用来获取为一个插口 s 指定（bind）的地址或名字：

```
int getsockname(int s, struct sockaddr *name , socklen_t *namelen);
```

- SYS_GETPEERNAME：“有连接”模式的插口，不管是 client 方还是 server 方，一旦建立起连接以后就有了一个“对方”。库函数 getpeername( )用来获取插口 s 的对方插口的地址（或名字）：

```
int getpeername(int s, struct sockaddr * name , socklen_t * namelen);
```

- SYS_SETSOCKOPT、SYS_GETSOCKOPT：库函数 setsockopt( )和 getsockopt( )用来设置和读取一个插口 s 所实行规程中的一些可选项，由于可选项都是在网络环境下运用的，而本书只讲述 Unix 域的插口，所以我们在这里并不关心这两个函数。

有了上面这些预备知识以后，我们就可以从下一节开始介绍 socket 通信机制在 Unix 域中的实现了。为了往后阅读的方便，此处先给出利用插口实现进程间通信的流程示意图（图 7.1）。

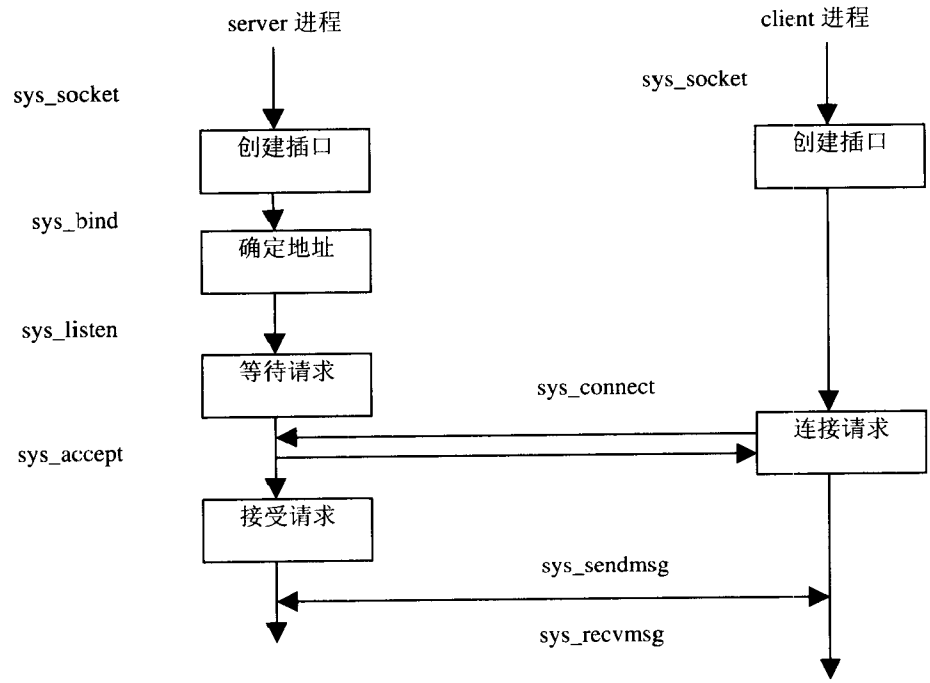

图 7.1 插口通信流程示意图

7.2 函数 sys_socket( )——创建插口

操作 SYS_SOCKET 是由 sys_socket( )实现的，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_socket( )]

```
889    asmlinkage long sys_socket(int family, int type, int protocol)
890    {
891        int retval;
892        struct socket *sock;
893
894        retval = sock_create(family, type, protocol, &sock);
895        if (retval < 0)
896            goto out;
897
898        retval = sock_map_fd(sock);
899        if (retval < 0)
900            goto out_release;
901
902    out:
903        /* It may be already another descriptor 8) Not kernel problem. */
904        return retval;
905
906    out_release:
907        sock_release(sock);
908        return retval;
909    }
```

前面说过，插口对于用户程序而言就是特殊的已打开文件。内核中为插口定义了一种特殊的文件类型，形成一种特殊的文件系统 sockfs，定义于 net/socket.c 中：

```
301    static struct vfsmount *sock_mnt;
302    static DECLARE_FSTYPE(sock_fs_type, "sockfs", sockfs_read_super,
303        FS_NOMOUNT|FS_SINGLE);
```

在系统初始化时，要通过 kern_mount( )安装这个文件系统。安装时有个作为连接件的 vfsmount 数据结构，这个结构的地址就保存在一个全局的指针 sock_mnt 中。所谓创建一个插口，就是在 sockfs 文件系统中创建一个特殊文件，或者说一个节点，并建立起为实现插口功能所需的一整套数据结构。所以首先是建立一个 socket 数据结构，然后将其“映射”到一个已打开文件中。函数 sock_create( )的代码在同一文件（socket.c）中。这段代码由于比较长，我们分段来看：

[sys_socketcall( ) > sys_socket( ) > sock_create( )]

```
814    int sock_create(int family, int type, int protocol, struct socket **res)
815    {
816        int i;
817        struct socket *sock;
818
819        /*
820         *  Check protocol is in range
```

```
821         */
822        if(family<0 || family>=NPROTO)
823            return -EAFNOSUPPORT;
824
825        /* Compatibility.
826
827           This uglymoron is moved from INET layer to here to avoid
828           deadlock in module load.
829         */
830        if (family == PF_INET && type == SOCK_PACKET) {
831            static int warned;
832            if (!warned) {
833                warned = 1;
834                printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n",
                                                current->comm);
835            }
836            family = PF_PACKET;
837        }
838
839    #if defined(CONFIG_KMOD) && defined(CONFIG_NET)
840        /* Attempt to load a protocol module if the find failed.
841         *
842         * 12/09/1996 Marcin: But! this makes REALLY only sense, if the user
843         * requested real, full-featured networking support upon configuration.
844         * Otherwise module support will break!
845         */
846        if (net_families[family]==NULL)
847        {
848            char module_name[30];
849            sprintf(module_name,"net-pf-%d",family);
850            request_module(module_name);
851        }
852    #endif
853
854        net_family_read_lock( );
855        if (net_families[family] == NULL) {
856            i = -EAFNOSUPPORT;
857            goto out;
858        }
859
```

这一段代码的开始部分是检查和处理参数的范围。由于我们在这里只关心 Unix 域，也就是当 family 为 AF_UNIX 时的情景，所以这段代码对我们不起什么作用。接下来是一段条件编译，如果系统配置了可动态安装内核模块的功能，并且网络驱动程序是动态安装的，就要检查一下由参数 family 所指定网域的驱动程序是否已经安装，尚未安装的话就要调用 request_module( )把它安装起来。至于 Unix 域插口的驱动程序，那是内核所固有的，并非动态安装的模块。此外，就像每种文件系统都有个 file_operations 数据结构一样，每种网域，包括 Unix 域，也都有个 net_proto_family 数据结构。在系统

初始化或安装模块时，要将指向相应网域的这个数据结构的指针填入一个数组 net_families[ ]中，否则就说明系统尚不支持给定的网域。Unix 域的 net_proto_family 数据结构为 unix_family_ops，定义于 net/unix/af_unix.c 中：

```
1844    struct net_proto_family unix_family_ops = {
1845        PF_UNIX,
1846        unix_create
1847    };
```

就是说，结构中只有一个代表着 Unix 域的常数 PF_UNIX 和一个函数指针 unix_create，而 PF_UNIX 决定了指向这个数据结构的指针在 net_families[ ]中的位置。

回到 sys_socket( )中继续往下看（socket.c）：

[sys_socketcall( ) > sys_socket( ) > sock_create( )]

```
862    /*
863     *  Allocate the socket and allow the family to set things up. if
864     *  the protocol is 0, the family is instructed to select an appropriate
865     *  default.
866     */
867
868        if (!(sock = sock_alloc( )))
869        {
870            printk(KERN_WARNING "socket: no more sockets\n");
871            i = -ENFILE;        /* Not exactly a match, but its the
872                                   closest posix thing */
873            goto out;
874        }
875
876        sock->type  = type;
877
878        if ((i = net_families[family]->create(sock, protocol)) < 0)
879        {
880            sock_release(sock);
881            goto out;
882        }
883
884        *res = sock;
885
886    out:
887        net_family_read_unlock( );
888        return i;
889        }
```

插口是由 socket 数据结构代表的，这种数据结构定义于 include/linux/net.h 中：

```
65    struct socket
66    {
67        socket_state        state;
68
69        unsigned long       flags;
70        struct proto_ops    *ops;
71        struct inode        *inode;
72        struct fasync_struct *fasync_list; /* Asynchronous wake up list */
73        struct file     *file;      /* File back pointer for gc */
74        struct sock     *sk;
75        wait_queue_head_t   wait;
76
77        short           type;
78        unsigned char       passcred;
79    };
```

结构中各个成分的用途随着代码的阅读会变得清楚起来，这里暂且只要知道有这些成分就可以了。不过我们建议读者在搞清了这些成分的用途以后再回过头来自己加上注释。

函数 sock_alloc( )分配一个 socket 数据结构并进行一些初始化，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > sock_alloc( )]

```
427   /**
428    *  sock_alloc  -   allocate a socket
429    *
430    *  Allocate a new inode and socket object. The two are bound together
431    *  and initialised. The socket is then returned. If we are out of inodes
432    *  NULL is returned.
433    */
434
435   struct socket *sock_alloc(void)
436   {
437       struct inode * inode;
438       struct socket * sock;
439
440       inode = get_empty_inode( );
441       if (!inode)
442           return NULL;
443
444       inode->i_sb = sock_mnt->mnt_sb;
445       sock = socki_lookup(inode);
446
447       inode->i_mode = S_IFSOCK|S_IRWXUGO;
448       inode->i_sock = 1;
449       inode->i_uid = current->fsuid;
450       inode->i_gid = current->fsgid;
451
```

```
452         sock->inode = inode;
453         init_waitqueue_head(&sock->wait);
454         sock->fasync_list = NULL;
455         sock->state = SS_UNCONNECTED;
456         sock->flags = 0;
457         sock->ops = NULL;
458         sock->sk = NULL;
459         sock->file = NULL;
460
461         sockets_in_use[smp_processor_id( )].counter++;
462         return sock;
463     }
```

可见，取得一个 inode 结构是取得一个 socket 结构的必要条件。不仅如此，socket 结构其实只是 inode 结构中的一部分！读者在“文件系统”一章中看到过有关 inode 数据结构的说明，在 inode 结构中有一个关键性的成分 u。这是一个 union，要按具体的文件类型和格式而解释成不同的数据结构。目前 Linux 支持 20 多种不同的文件系统，因此对这个 union 有 20 多种不同的解释，而 socket 结构正是其中之一。代码中 445 行的 socki_lookup( )只是将 inode 结构中的这个 union 解释为 socket 结构而已，见 net/socket.c 中的代码：

[sys_socketcall( ) > sys_socket( ) > sock_alloc( ) > socki_lookup( )]

```
377     extern __inline__ struct socket *socki_lookup(struct inode *inode)
378     {
379         return &inode->u.socket_i;
380     }
```

同时，在 inode 结构中还要将 i_mode 里的 S_IFSOCK 标志位设成 1，并将 i_sock 也设成 1，以示这个 inode 结构所代表的并不是磁盘文件，而是一个插口。

分配了一个 socket 结构，并且设置好插口的类型以后，就通过由 unix_family_ops 提供的函数指针 create 调用 Unix 域的插口创建程序 unix_create( )，其代码在 af_unix.c 中：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > unix_create( )]

```
498     static int unix_create(struct socket *sock, int protocol)
499     {
500         if (protocol && protocol != PF_UNIX)
501             return -EPROTONOSUPPORT;
502
503         sock->state = SS_UNCONNECTED;
504
505         switch (sock->type) {
506         case SOCK_STREAM:
507             sock->ops = &unix_stream_ops;
508             break;
509             /*
```