第15章　高级进程间通信

15.1　引言

上一章说明了各种UNIX系统提供的IPC经典方法，包括:管道、FIFO、消息队列、信号量和共享存储。本章介绍某些高级的IPC以及它们的应用方法，包括:流管道和命名流管道。使用这些机制，可以在进程间传送打开文件描述符。在分别为每一个客户进程提供一个通道的系统中，这些通信机制使客户进程能与精灵服务进程会合。 4.2BSD和SVR3.2最早提供这些高级形式的IPC，但是至今尚未广泛使用，也缺少参考文献。本章中很多思想来自 Pressotto和Ritchie〔1990〕的论文。

15.2　流管道

流管道是一个双向（全双工）管道。单个流管道就能向父、子进程提供双向的数据流。图15-1显示了观察流管道的两种方式。它与图 14-1的唯一区别是双向箭头连线，因为流管道是全双工的。

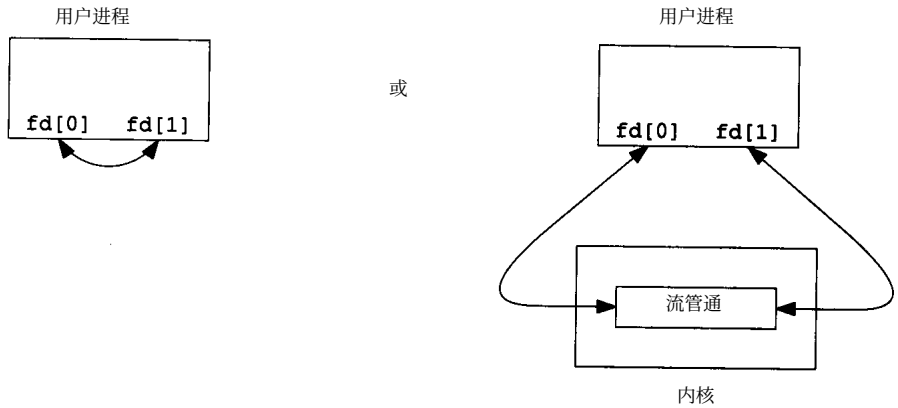

图15-1　观察流管道的两种方式

实例

下面用一个流管道再次实现了程序 14-9的协作进程实例。程序15-1是新的main函数。add2协作进程与程序14-8中的相同。程序15-1调用了创建一个流管道的新函数s_pipe。（下面将说明该函数的SVR4和4.3+BSD版本。）

程序15-1　用流管道驱动add2过滤进程的程序

```
#include    <signal.h>
#include    "ourhdr.h"
```

```
static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (s_pipe(fd) < 0)         /* only need a single stream pipe */
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) {                         /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ( (n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);

    } else {                                    /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO) {
            if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
        }
        if (fd[1] != STDOUT_FILENO) {
            if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
        }
        if (execl("./add2", "add2", NULL) < 0)
            err_sys("execl error");
    }
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

父程序只使用fd[0]，子程序只使用fd[1] 。因为流管道的每一端都是全双工的，所以父进程读、写fd[0]，而子程序将fd[1]复制到标准输入和标准输出。图15-2显示了由此构成的描述符。

s_pipe函数定义为与标准pipe函数类似。它的调用参数与pipe相同，但返回的描述符以读-写方式打开。

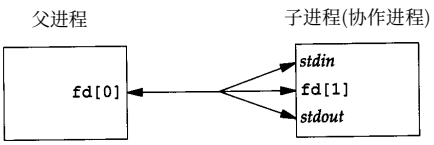

图15-2 为协作进程安排的描述符

实例——SVR4下的s_pipe函数

程序15-2是s_pipe函数的SVR4版本。它只是调用创建全双工管道的标准pipe函数。

程序15-2 s_pipe函数的SVR4版本

```
#include    "ourhdr.h"

int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( pipe(fd) );
}
```

在系统V的早期版本中也可以创建流管道，但要进行的处理较多。有关在SVR3.2下创建流管道的详细情况，请参阅Stevens〔1990〕。

图15-3显示了SVR4之下管道的基本结构。它主要是两个相互连接的流首。

因为管道是一种流设备，故可将处理模块压入管道的任一一端。15.5.1节将用此技术提供一个可以装配的命名管道。

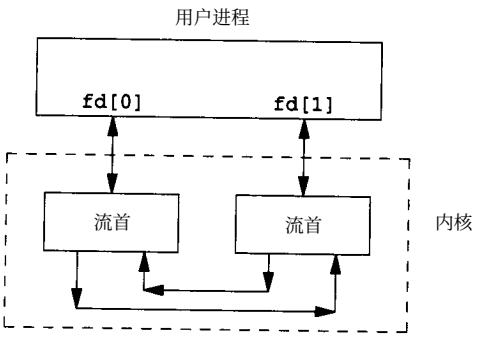

图15-3 SVR4之下的管道

实例——4.3+BSD之下的s_pipe函数

程序15-3是s_pipe函数的BSD版本。此函数在4.2BSD及以后的各版本中起作用。它创建一对互连的UNIX域流套接口。

自4.2BSD开始，常规的管道已用此方式实现。但是，当调用pipe时，第一个描述符的写端和第二个描述符的读端都被关闭。为获得全双工管道，必须直接调用socketpair。

程序15-3 s_pipe函数的BSD版本

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    "ourhdr.h"

int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( socketpair(AF_UNIX, SOCK_STREAM, 0, fd) );
}
```

15.3 传送文件描述符

在进程间传送打开文件描述符的能力非常有用。用此可以对客户机 /服务器应用进行不同的设计。它允许一个进程（一般是服务器）处理与打开一个文件有关的所有操作（涉及的细节可能是：将网络名翻译为网络地址、拨号调制解调器、协商文件锁等。）以及向调用进程返回一描述符，该描述符可被用于以后的所有 I/O函数。打开文件或设备的所有细节对客户而言都是透明的。

> 4.2BSD支持传送打开描述符，但其实施中有些错误。4.3BSD排除了这些错误。SVR3.2及以上版本都支持传送打开描述符。

下面进一步说明“ 从一个进程向另一个进程传送一打开文件描述符 ”的含义。回忆图 3-2，其中显示了两个进程，它们打开了同一文件。虽然它们共享同一 v节点表，但每个进程都有它自己的文件表项。

当从一个进程向另一个进程传送一打开文件描述符时，我们想要发送进程和接收进程共享同一文件表项。图15-4显示了所希望的安排。在技术上，发送进程实际上向接受进程传送一个指向一打开文件表项的指针。该指针被分配存放在接收进程的第一个可用描述符项中。（注意，不要得到错觉以为发送进程和接收进程中的描述符编号是相同的，通常它们是不同的。）这种情况与在fork之后，父、子进程完全共享一个打开文件表项相同（见图 8-1）。

当发送进程将描述符传送给接收进程后，通常它关闭该描述符。发送进程关闭该描述符并不造成关闭该文件或设备，其原因是该描述符对应的文件仍需为接收进程打开（即使接收进程尚未接收到该描述符）。

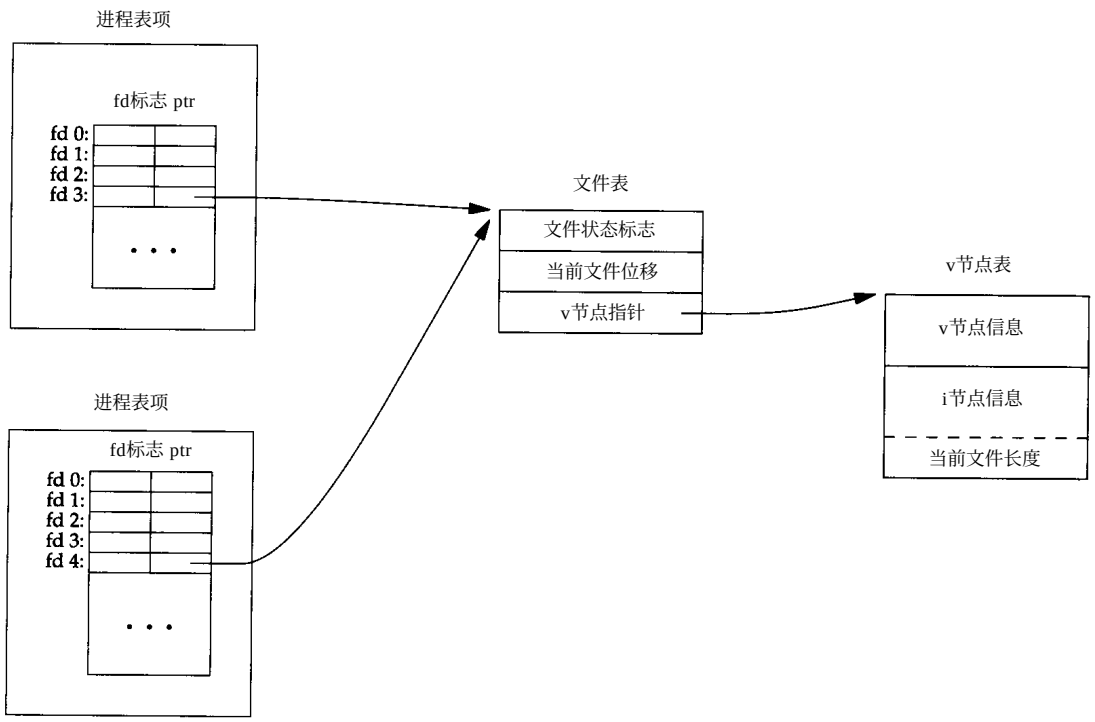

图15-4 从上一进程传送一个打开文件至下一进程

下面定义本章使用的三个函数（第18章也将使用）以发送和接收文件描述符。本节将会给出对于SVR4和4.3+BSD的这三个函数的不同实现。

```
#include "ourhdr.h"

int send_fd(int spipefd, int filedes);

int send_err(int spipefd, int status, const char *errmsg);
```

两个函数返回：若成功则为0，若出错则为-1

```
int recv_fd(int spipefd, ssize_t (*userfunc)(int, const void *, size_t));
```

返回：若成功则为文件描述符，若出错则<0

当一个进程（通常是服务器）希望将一个描述符传送给另一个进程时，它调用 send_fd或send_err。等待接收描述符的进程（客户机）调用recv_fd。

send_fd经由流管道*spipefd*发送描述符filedes。send_err 经由流管道*spipefd*发送*errmsg*和*status*字节。*status*的值应在－1~－255之间

客户机调用recv_fd接收一描述符。如果一切正常（发送者调用了send_fd），则作为函数值返回非负描述符。否则，返回值是由send_err发送的*status*（－1~－255之间的一个值）。另外，如果服务器发送了一条出错消息，则客户机调用它自己的*userfunc*处理该消息。*userfunc*的第一个参数是常数STDERR_FILENO，然后是指向出错消息的指针及其长度。客户机常将 *userfunc*指定为UNIX的write函数。

我们实现了用于这三个函数的我们自己制定的协议。为发送一描述符， send_fd先发送两个0字节，然后是实际描述符。为了发送一条出错消息， send_err 发送*errmsg*，然后是1个0字节，最后是*status*字节的绝对值（1~255）。recv_fd读流管道中所有字节直至null字符。null字符之前的所有字符都送给调用者的*userfunc*。recv_fd读到的下一个字节是*status*字节。若*status*字节为0，那么一个描述符已传送，否则表示没有接收到描述符.

send_err函数在将出错消息写到流管道后，即调用send_fd函数。这示于程序15-4中。

程序15-4 send_err函数

```
#include    "ourhdr.h"

/* Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol. */

int
send_err(int clifd, int errcode, const char *msg)
{
    int     n;

    if ( (n = strlen(msg)) > 0)
        if (writen(clifd, msg, n) != n) /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(clifd, errcode) < 0)
        return(-1);

    return(0);
}
```

以下三节介绍了在SVR4、4.3BSD和4.3+BSD下，两个函数send_fd和recv_fd的实际实现。

15.3.1 SVR4

在SVR4之下，文件描述符用两个ioctl命令在一流管道中交换，这两个命令是：I_SENDFD和I_RECVFD。为了发送一描述符，将ioctl的第三个参数设置为实际描述符。这示于程序15-5中。

程序15-5 SVR4的send_fd函数

```
#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    char    buf[2];     /* send_fd()/recv_fd() 2-byte protocol */

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (fd < 0) {
        buf[1] = -fd;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;     /* zero status means OK */
    }

    if (write(clifd, buf, 2) != 2)
        return(-1);

    if (fd >= 0)
        if (ioctl(clifd, I_SENDFD, fd) < 0)
            return(-1);
    return(0);
}
```

当接收一描述符时，ioctl的第三个参数是一指向strrecvfd结构的指针。

```
struct strrecvfd {
  int     fd;       /* new descriptor */
  uid_t   uid;      /* effective user ID of sender */
  gid_t   gid;      /* effective group ID of sender */
  char    fill[8];
};
```

recv_fd读流管道直到接收到双字节协议的第一个字节（null字节）．当发出带I_RECVFD命令的ioctl时，在流读首处的第一条消息应当是一个描述符，它是由 I_SENDFD发来的，或者得到一条出错消息。这示于程序15-6中。

程序15-6 SVR4的recv_fd函数

```
#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
```

```
     * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
     * 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int                 newfd, nread, flag, status;
    char                *ptr, buf[MAXLINE];
    struct strbuf       dat;
    struct strrecvfd    recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(servfd, NULL, &dat, &flag) < 0)
            err_sys("getmsg error");
        nread = dat.len;
        if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
            /* See if this is the final data with null & status.
               Null must be next to last byte of buffer, status
               byte is last byte.  Zero status means there must
               be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (ioctl(servfd, I_RECVFD, &recvfd) < 0)
                        return(-1);
                    newfd = recvfd.fd;  /* new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);

        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

15.3.2 4.3BSD

不幸的是，对于4.3BSD以及在其基础上构造的SunOS和Ultrix，以及从4.3BSD Reno开始的后续版本必须提供不同的实现。

为了交换文件描述符，调用sendmsg(2)和recvmsg(2)函数。这两个函数的参数中都有一个指向msghdr的指针，该结构包含了所有关于要发送和接收消息的信息。该结构定义在<sys/socket.h>头文件中，在BSD4.3之下，其样式是：

```
struct msghdr {
  caddr_t       msg_name;     /* optional address */
```

```
    int             msg_namelen; /* size of address */
    struct iovec *msg_iov;      /* scatter/gather array */
    int             msg_iovlen;  /* # elements in msg_iov array */
    caddr_t         msg_accrights;    /* access rights sent/received */
    int             msg_accrightslen; /* size of access rights buffer */
};
```

头两个元素通常用于在网络连接上发送数据报文，在这里，目的地址可以由每个数据报文指定。下面两个元素使我们可以指定缓存的数组（散布读和聚集写），这如同对readv和writev函数（见12.7节）的说明一样。最后两个元素处理存取权的传送和接收。当前唯一定义的存取权是文件描述符。存取权仅可跨越一个UNIX域套接口传送（即在4.3BSD之下作为流管道所使用的）。为了发送或接收一文件描述符，将msg_accrights设置为指向该整型描述符，将msg_accrightslen设置为描述符的长度（即整型的长度）。仅当此长度非0时，才传送或接收描述符。

程序15-7是4.3BSD的send_fd函数。

程序15-7　4.3BDS的send_fd函数

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <errno.h>
#include    <stddef.h>
#include    "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;

    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd < 0) {
        msg.msg_accrights    = NULL;
        msg.msg_accrightslen = 0;
        buf[1] = -fd;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        msg.msg_accrights    = (caddr_t) &fd;   /* addr of descriptor */
        msg.msg_accrightslen = sizeof(int);     /* pass 1 descriptor */
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_fd() */

    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);

    return(0);
}
```

在sendmsg调用中，发送双字节协议数据（null和status字节）和描述符。

为了接收一文件描述符，从流管道读，直至读到 null字节，它位于最后的status字节之前。null字节之前是一条出错消息，它来自发送者。这示于程序15-8。

程序15-8 4.3BSD的recv_fd函数

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <stddef.h>
#include    "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nread, status;
    char            *ptr, buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

status = -1;
for ( ; ; ) {
    iov[0].iov_base = buf;
    iov[0].iov_len  = sizeof(buf);
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_accrights = (caddr_t) &newfd;/* addr of descriptor */
    msg.msg_accrightslen = sizeof(int);  /* receive 1 descriptor */

    if ( (nread = recvmsg(servfd, &msg, 0)) < 0)
        err_sys("recvmsg error");
    else if (nread == 0) {
        err_ret("connection closed by server");
        return(-1);
    }

        /* See if this is the final data with null & status.
           Null must be next to last byte of buffer, status
           byte is last byte.  Zero status means there must
           be a file descriptor to receive. */
    for (ptr = buf; ptr < &buf[nread]; ) {
        if (*ptr++ == 0) {
            if (ptr != &buf[nread-1])
                err_dump("message format error");
            status = *ptr & 255;
            if (status == 0) {
                if (msg.msg_accrightslen != sizeof(int))
                    err_dump("status = 0 but no fd");
                /* newfd = the new descriptor */
            } else
                newfd = -status;
            nread -= 2;
        }
    }
    if (nread > 0)
        if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
```

```
                return(-1);

        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

注意，该程序总是准备接收一描述符（在每次调用 recvmsg之前，设置msg_accrights和msg_accrightslen），但是仅当在返回时msg_accrightslen非0，才确实接收到一描述符。

15.3.3 4.3+BSD

从4.3BSD Reno开始，更改了msghdr结构的定义。在以前版本中被称之为“存取权”的最后两个元素改称为“辅助数据”。另外，在该结构结束处增加了一个新成员msg_flags。

```
struct msghdr {
  caddr_t       msg_name;       /* optional address */
  int           msg_namelen;    /* size of address */
  struct iovec *msg_iov;        /* scatter/gather array */
  int           msg_iovlen;     /* # elements in msg_iov array */
  caddr_t       msg_control;    /* ancillary data */
  u_int         msg_controllen; /* size of ancillary data */
  int           msg_flags;      /* flags on received message */
};
```

现在，msg_control字段指向一个cmsghdr（控制消息头）结构。

```
struct cmsghdr {
  u_int  cmsg_len;   /* data byte count, including header */
  int    cmsg_level; /* originating protocol */
  int    cmsg_type;  /* protocol-specific type */
  /* followed by the actual control message data */
};
```

为了发送一文件描述符，将cmsg_len设置为cmsghdr结构长度加一个整型（描述符）的长度。将cmsg_level设置为SOL_SOCKET，cmsg_type设置为SCM_RIGHTS，这表明正在传送的是存取权（SCM表示套接口级控制消息）。实际描述符的存放位置紧随在cmsy_type字段之后，使用CMSG_DATA宏以获得指向该整型数的指针。程序15-9示出了4.3BSD Reno之下的send_fd函数。

程序15-9 4.3BSD的send_fd函数

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <errno.h>
#include    <stddef.h>
#include    "ourhdr.h"

static struct cmsghdr   *cmptr = NULL;  /* buffer is malloc'ed first time */
#define CONTROLLEN  (sizeof(struct cmsghdr) + sizeof(int))
        /* size of control buffer to send/recv one file descriptor */

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
```

```
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (fd < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = (caddr_t) cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd;      /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_fd() */

    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

为了接收一描述符（见程序15-10），我们为cmsghdr结构和一描述符分配了足够的存储区，设置msg_control使其指向所分配到的存储区，然后调用recvmsg。

程序15-10 4.3BSD Reno的recv_fd函数

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <stddef.h>
#include    "ourhdr.h"

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */
#define CONTROLLEN  (sizeof(struct cmsghdr) + sizeof(int))
        /* size of control buffer to send/recv one file descriptor */

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd(). */
int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))

{
    int             newfd, nread, status;
    char            *ptr, buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

    status = -1;
    for ( ; ; ) {
```

```
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = (caddr_t) cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ( (nread = recvmsg(servfd, &msg, 0)) < 0)
            err_sys("recvmsg error");
        else if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
            /* See if this is the final data with null & status.
               Null must be next to last byte of buffer, status
               byte is last byte.  Zero status means there must
               be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr); /* new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

15.4 open服务器第1版

目前，使用文件描述符传送技术开发了一个open服务器：它是一个可执行程序，由一个进程执行以打开一个或多个文件。该服务器不是将文件送回调用进程，而是送回一个打开文件描述符。这使该服务器对任何类型的文件（例如调制解调器线或网络连接）而不单是普通文件都能起作用。这也意味着，用IPC交换最小量的信息——从客户机到服务器传送文件名和打开方式，而从服务器到客户机返回描述符。文件内容则不需用IPC传送。

将服务器设计成一个单独的可执行程序有很多优点：

(1) 任一客户机都易于和服务器联系，这类似于客户机调用一库函数。不需要将一特定服务编码在应用程序中，而是设计一种可供重用的设施。

(2) 如若需要更改服务器，那么也只影响一个程序。相反，更新一库函数可能要更改调用此库函数的所有程序（用连编程序重新连接）。共享库函数可以简化这种更新。

(3) 服务器可以是设置-用户-ID程序，于是使其具有客户机没有的附加许可权。注意，一

个库函数（或共享库函数）不能提供这种能力。

客户机创建一流管道，然后调用 fork和exec以调用服务器。客户机经流管道发送请求，服务器经管道回送响应。定义客户机和服务器间的协议如下：

(1) 客户机经流管道向服务器发送下列形式的请求：

```
open <pathname> <openmode>\0
```

<*openmode*>是open函数的第二个参数，以十进制表示。该请求字符串以null字节结尾。

(2) 服务器调用send_fd 或send_err回送一打开描述符或一条出错消息。

这是一个进程向其父进程发送一打开描述符的实例。 15.6节将修改此实例，其中使用了一个精灵服务器，它将一个描述符发送给完全无关的进程。

程序15-11是头文件open.h，它包括标准系统头文件，并且定义了各个函数原型。

程序15-11 open.h头文件

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CL_OPEN "open"          /* client's request for server */

                /* our function prototypes */
int     csopen(char *, int);
```

程序15-12是main函数，其中包含一个循环，它先从标准输入读一个路径名，然后将该文件复制至标准输出。它调用函数csopen以与open服务器联系，从其返回一打开描述符。

程序15-12 main函数

```
#include    "open.h"
#include    <fcntl.h>

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE], line[MAXLINE];

                /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        line[strlen(line) - 1] = 0; /* replace newline with null */

                /* open the file */
        if ( (fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

                /* and cat to stdout */
        while ( (n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

程序15-13是函数csopen，它先创建一流管道，然后进行服务器的fork和exec操作。

程序15-13 csopen函数

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (s_pipe(fd) < 0)
            err_sys("s_pipe error");
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) {        /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO) {
                if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("dup2 error to stdin");
            }
            if (fd[1] != STDOUT_FILENO) {
                if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                    err_sys("dup2 error to stdout");
            }
            if (execl("./opend", "opend", NULL) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len)
        err_sys("writev error");

            /* read descriptor, returned errors handled by write() */
    return( recv_fd(fd[0], write) );
}
```

子进程关闭管道的一端，父进程关闭另一端。子进程也为它所执行的服务器将管道的一端复制到其标准输入和标准输出0（另一种可选择的方案是将描述符fd[1]的ASCII 表示形式作为一个参数传送给服务器。）

父进程将请求发送给服务器，请求中包含路径名和打开方式。最后，父进程调用 recv_fd以返回描述符或错误消息。如果服务器返回一错误消息则调用 write，向标准出错输出该消息。

现在，观察 open服务器。其程序是 opend，它由子进程执行（见程序 15-13）。先观察

opend.h头文件（见程序15-14），它包括了系统头文件，并且说明了全局变量和函数原型。

程序15-14 opend.h头文件

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CL_OPEN "open"              /* client's request for server */

            /* declare global variables */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open() flag: O_xxx ... */
extern char *pathname;  /* of file to open() for client */

            /* function prototypes */
int      cli_args(int, char **);
void     request(char *, int, int);
```

main函数（见程序15-15）经流管道（它的标准输入）读来自客户机的请求，然后调用函数request。

程序15-15 main函数

```
#include    "opend.h"

            /* define global variables */
char     errmsg[MAXLINE];
int      oflag;
char    *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ( (nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */

        request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

程序15-16中的request 函数承担全部工作。它调用函数 buf_args将客户机请求分解成标准argv型的参数表，然后调用函数cli_args处理客户机的参数。如果一切正常，则调用 open打开相应文件，接着调用 send_fd，经由流管道（它的标准输出）将描述符回送给客户机。如果出错则调用send_err回送一则出错消息，其中使用了前面说明的客户机 -服务器协议。

客户机请求是一个空的中断的字符串，其参数由空格分隔。程序 15-17中的buf_args函数将字符串分解成标准argv型参数表，并调用用户函数处理参数。本节稍后及第18章将用到该函数。我们使用ANSI C函数strtok将字符串分割成参数。

程序15-16 request函数

```
#include    "opend.h"
#include    <fcntl.h>
```

```
void
request(char *buf, int nread, int fd)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request not null terminated: %*.*s\n",
                                        nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }

            /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(fd, -1, errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                pathname, strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }

            /* send the descriptor */
    if (send_fd(fd, newfd) < 0)
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

程序15-17　buf_args函数

```
#include    "ourhdr.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/* buf[] contains white-space separated arguments.  We convert it
 * to an argv[] style array of pointers, and call the user's
 * function (*optfunc)() to process the argv[] array.
 * We return -1 to the caller if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's
 * buf[] array is modified (nulls placed after each token). */

int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;

    while ( (ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    return( (*optfunc)(argc, argv) );
            /* Since argv[] pointers point into the user's buf[],
```

```
                user's function can just copy the pointers, even
                though argv[] array will disappear on return. */
}
```

buf_args调用的服务器函数是cli_args（见程序15-18）。它验证客户机发送的参数是否正确，然后将路径名和打开方式存放在全局变量中。

这样也就完成了open服务器，它由客户机执行fork和exec而调用。在fork之前创建了一个流管道，然后客户机和服务器用其进行通信。在这种安排下，每个客户机都有一服务器。

在下一节观察了客户机-服务器连接后，我们将在15.6节重新实现一个open服务器，其中用一个精灵进程作为服务器，所有客户机都与其进行联系。

程序15-18 cli_args函数

```
#include    "opend.h"

/* This function is called by buf_args(), which is called by
 * request().  buf_args() has broken up the client's buffer
 * into an argv[] style array, which we now process. */

int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }

    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

15.5 客户机-服务器连接函数

对于相关进程（例如，父进程和子进程）之间的IPC，流管道非常有用。前节所述的open服务器使用未命名的流管道能从子进程向父进程传送文件描述符。但是当处理无关进程时（例如，若服务器是一精灵进程），则需要使用有名的流管道。

可以先构造一未命名流管道（用s_pipe函数），然后对每一端加上一文件系统路径名。精灵进程服务器将只创建流管道的一端，并对该端加上一名字。这样，无关的客户机可以向服务者的流管道端发送消息，从而与精灵进程会聚。这类似于图14-11中所示的情况，在该图中客户机使用FIFO发送它们的请求。

一种更好的方法是：服务器创建一名字公开的流管道的一端，然后客户机连接至该端。另外，每次一个新客户机连至服务器的命名流管道时，就在客户机和服务器之间创建一条全新的流管道。这样，每次一个新客户机连接至服务器，以及客户机终止时，服务器都会得到通知。SVR4和4.3+BSD都支持这种形式的IPC。本节将开发三个函数，客户机-服务器可以使用这些函数以建立上述针对每个客户机的连接。

```
#include "ourhdr.h"

int serv_listen(const char *name);
```

返回：若成功则返回为文件描述符，若出错则<0

首先，一个服务器应当宣布，它愿意听取客户机在一个众所周知名字上的连接，该名字是在文件系统中的一个路径名。为此调用 serv_listen，其参数*name*是服务器的众所周知名字。客户机希望与服务器连接时使用此名字。该函数的返回值是命名流管道服务器端的文件描述符。

一旦服务器已调用serv_listen，它将调用serv_accept等待客户连接到达。

```
#include "ourhdr.h"

int serv_accept(int listenfd, uid_t *uidptr);
```

返回：若成功则返回为文件描述符，若出错则 <0

*listenfd*是serv_listen返回的描述符。在客户机连接到服务器众所周知的名字上之前，此函数并不返回。当客户机连接至服务器时，自动创建一条全新的流管道，其新描述符作为该函数的值返回。另外，客户机的有效用户ID通过指针uidptr存储。

客户机为与服务器连接只需调用cli_conn函数。

```
#include "ourhdr.h"

int cli_conn(const char *name);
```

返回：若成功则返回为文件描述符，若出错则 <0

客户指定的*name*应当与服务器调用serv_listen时宣布的相同。返回的描述符引用连接至服务器的流管道

使用上述三个函数，就可编写服务器精灵进程，它可以管理任一数量的客户机。唯一的限制是单个进程可用的描述符数，服务器对于每一个客户机连接都需要一个描述符。因为这些函数处理的都是普通文件描述符，所以服务器使用 select或poll就可在所有客户机之间多路转接I/O请求。最后，因为客户机-服务器连接都是流管道，所以可以经由连接传送打开描述符。

下面两节将说明在SVR4和4.3+BSD之下这三个函数的实现。第 18章开发一个通用的连接服务器时，也将使用这三个函数。

15.5.1 SVR4

SVR4提供装配的流以及一个名为 connld的流处理模块，用其可以提供与服务器有唯一连接的命名流管道。

> 装配流和connld模块是由Presotto和Ritchie[1990]为Research UNIX系统开发的，后来由SVR4采用。

首先，服务器创建一未命名流管道，并将流处理模块connld压入一端。图15-5显示了这一处理结果。

然后，使压入 connld的一端具有一路径名。SVR4提供fattach函数实现这一点。任一进程（例如客户机）打开此路径名就引用该管道的命名端。

程序15-19使用了20余行代码实现serv_listen函数。

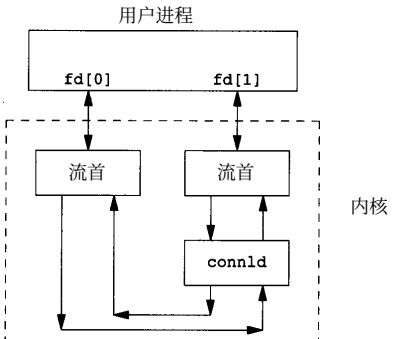

图15-5 在一端压入connld模块后的流管道

程序15-19 SVR4的serv_listen函数

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

#define FIFO_MODE   (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
                    /* user rw, group rw, others rw */

int         /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int     tempfd, fd[2], len;

                    /* create a file: mount point for fattach() */
    unlink(name);
    if ( (tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);

    if (pipe(fd) < 0)
        return(-3);
                            /* push connld & fattach() on fd[1] */
    if (ioctl(fd[1], I_PUSH, "connld") < 0)
        return(-4);
    if (fattach(fd[1], name) < 0)
        return(-5);

    return(fd[0]);  /* fd[0] is where client connections arrive */
}
```

当另一进程对管道的命名端（connld模块压入端）调用open时，发生下列处理过程：

(1) 创建一个新管道。

(2) 该新管道的一个描述符作为open的返回值回送给客户机。

(3) 另一个描述符在命名管道的另一端（亦即不是压入 connld的端）传送给服务器。服务器以带I_RECVFD命令的ioctl接受该新描述符。

假定服务器用fattach函数加到其管道的众所周知的名字是/tmp/serv1。图15-6显示了客户机调用：

```
fd=open("/tmp/serv1", O_RDWR);
```

并返回后产生的结果。

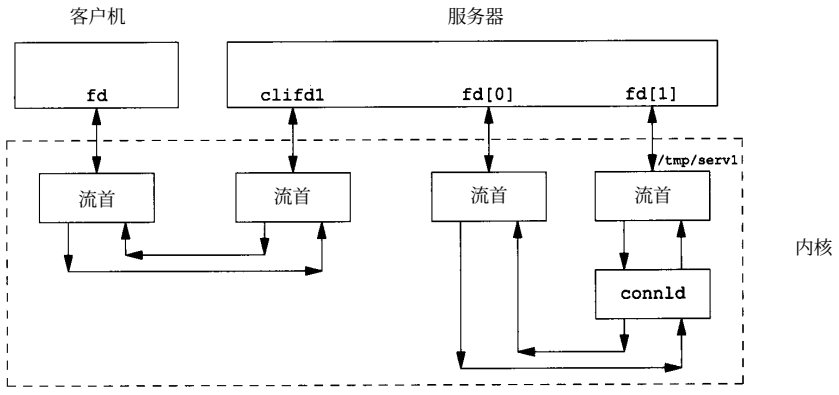

图15-6 客户机-服务器在命名管道上的连接

客户机和服务器之间的管道是open创建的，被打开的路径名实际上是一命名管道，其中压入了connld模块。客户机得到由open返回的文件描述符fd。服务器处的新文件描述符是clifdl，它是由服务器在描述符fd[0]上以I_RECVFD命令调用ioctl而接收到的。一旦服务器在fd[1]上压入了connld模块，并对fd[1]附接上一个名字，它就不再使用fd[1]。

服务器调用程序15-20中的serv_accept函数等待客户机连接到达。

程序15-20　SVR4的serv_accept函数

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID. */

int         /* returns new fd if all OK, -1 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd    recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);     /* could be EINTR if signal caught */

    if (uidptr != NULL)
        *uidptr = recvfd.uid;   /* effective uid of caller */

    return(recvfd.fd);  /* return the new descriptor */
}
```

在图15-6中，serv_accept的第一个参数应当是描述符fd[0]，serv_accept的返回值是描述符clifdl。

客户机调用程序15-21中的cli_conn函数起动对服务器的连接。

程序15-21　SVR4的cli_conn函数

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

/* Create a client endpoint and connect to a server. */

int         /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int     fd;

                /* open the mounted stream */
    if ( (fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0)
        return(-2);

    return(fd);
}
```

我们对返回的描述符是否引用一个流设备进行了两次检查，以便处理服务器没有起动，但该路径名却存在于文件系统中的情况。（在SVR4下，几乎没有什么理由去调用cli_conn，而不

是直接调用 open。下一节将看到，在 BSD系统之下， cli_conn函数要复杂得多，因此编写cli_conn函数就很必要。）

15.5.2 4.3+BSD

在4.3+BSD之下，为了用UNIX域套接口连接客户机和服务器，需要有一套不同的操作函数。因为应用 socket、bind、listen、accept和connect函数的大部分细节与其他网络协议有关（参见Stevens〔1990〕），所以此处不详细展开。

> 因为SVR4也支持UNIX域套接口，所以本节所示代码同样可在SVR4之下工作。

程序15-22包含了serv_listen函数。它是服务器调用的第一个函数。

程序15-22 4.3+BSD的serv_listen函数

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/un.h>
#include    "ourhdr.h"

/* Create a server endpoint of a connection. */

int         /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int                 fd, len;
    struct sockaddr_un  unix_addr;

                    /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    unlink(name);   /* in case it already exists */

                    /* fill in socket address structure */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif

                    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);

    if (listen(fd, 5) < 0)  /* tell kernel we're a server */
        return(-3);

    return(fd);
}
```

首先，调用socket函数创建一个UNIX域套接口。然后，填充sockeraddr_un结构，将一个众所周知的路径名赋与该套接口。该结构是调用bind函数的一个参数。然后调用listen以通知内核：本服务器正等待来自客户机的连接。（listen的第二个参数是5，它是最大的未决连接请求数，

内核将这些请求对该描述符进行排队。大多数实现强制该值的上限为5。）

客户机调用cli_conn函数（见程序15-23）起动与服务器的连接。

程序15-23　4.3+BSD的cli_conn函数

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/stat.h>
#include    <sys/un.h>
#include    "ourhdr.h"

/* Create a client endpoint and connect to a server. */

#define CLI_PATH    "/var/tmp/"     /* +5 for pid = 14 chars */
#define CLI_PERM    S_IRWXU         /* rwx for user only */

int         /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int                 fd, len;
    struct sockaddr_un  unix_addr;

                /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

                /* fill socket address structure w/our address */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    sprintf(unix_addr.sun_path, "%s%05d", CLI_PATH, getpid());
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
    if (len != 16)
        err_quit("length != 16");   /* hack */
#endif

    unlink(unix_addr.sun_path);     /* in case it already exists */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);
    if (chmod(unix_addr.sun_path, CLI_PERM) < 0)
        return(-3);

                /* fill socket address structure w/server's addr */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif

    if (connect(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-4);

    return(fd);
}
```

调用socket函数以创建客户端的UNIX域套接口，然后客户机专用的名字填入socketaddr_un结构。该路径名的最后5个字符是客户机的进程ID（我们可以查证此结构的长度是14个字符，以避免UNIX域套接口早期实现的某些错误）。在路径名已经存在的情况下调用unlink，然后再调用bind将一名字赋与客户机的套接口，这就创建了文件系统中的路径名，该文件的类型是套接口。接着调用chmod，它关闭除user_read，user_write和user_execute以外的存取权。在serv_accept中，服务器检查该套接口的这些许可权和用户ID，以验证用户的身份。

然后，以服务器众所周知的路径名填充另一个socketaddr_un结构。最后，connect函数起动与服务器的连接。

创建每个客户机与服务器的唯一连接是通过在serv_accept函数中调用accept函数实现的（见程序15-24）。

程序15-24 4.3+BSD的serv_accept函数

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/stat.h>
#include    <sys/un.h>
#include    <stddef.h>
#include    <time.h>

#include    "ourhdr.h"

#define STALE    30  /* client's name can't be older than this (sec) */

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us. */

int         /* returns new fd if all OK, <0 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, len;
    time_t              staletime;
    struct sockaddr_un  unix_addr;
    struct stat         statbuf;

    len = sizeof(unix_addr);
    if ( (clifd = accept(listenfd, (struct sockaddr *) &unix_addr, &len)) < 0)
        return(-1);     /* often errno=EINTR, if signal caught */

            /* obtain the client's uid from its calling address */
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len -= sizeof(unix_addr.sun_len) - sizeof(unix_addr.sun_family);
#else               /* vanilla 4.3BSD */
    len -= sizeof(unix_addr.sun_family);    /* len of pathname */
#endif
    unix_addr.sun_path[len] = 0;            /* null terminate */

    if (stat(unix_addr.sun_path, &statbuf) < 0)
        return(-2);
#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0)
        return(-3);     /* not a socket */
#endif
    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU)
          return(-4);   /* is not rwx------ */

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
```

```
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime)
          return(-5);    /* i-node is too old */

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;    /* return uid of caller */

    unlink(unix_addr.sun_path);      /* we're done with pathname now */

    return(clifd);
}
```

服务器在调用accept中堵塞以等待客户机调用cli_conn。当accept返回时，其返回值是连向客户机的全新的描述符（这类似于 SVR4中connld模块所做的）。另外，accept也通过其第二个参数（指向socketaddr_un结构的指针）返回客户机赋与其套接口的路径名（它包含客户机的进程ID）。用null字节结束此路径名，然后调用stat。这使我们可以验证此路径名确实是一个套接口，其许可权user_read，user_write和user_execute。我们也验证与该套接口相关的三个时间不超过30秒。（time函数返回自UNIX纪元经过的时间和日期，它们都以秒计。）如果所有这些检查都通过，则认为该客户机的身份（其有效用户ID）是该套接口的所有者。虽然这种检查并不完善，但却是现有系统所能做得最好的。（如果内核能像SVR4 I_RECVFD做的那样，将有效用户ID返回给accept，那就更好一些。）

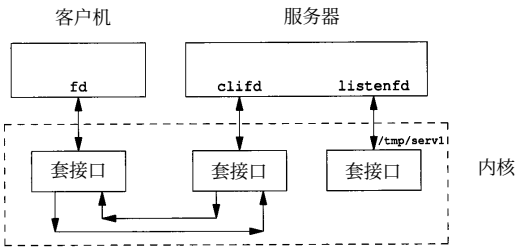

图15-7 UNIX域套接口上客户机-服务器连接

图15-7显示了cli_conn调用返回后的这种连接，假定服务器众所周知的名字是 /tmp/serv1。请将此图与图15-6相比较。

15.6 open服务器第2版

在15.4节中，客户机调用fork和exec构造了一个open服务器，它说明了如何从子程序向父程序传送文件描述符。本节将开发一个精灵进程样式的open服务器。一个服务器处理所有客户机的请求。由于避免使用了fork和exec，我们期望这一设计会更有效的。在客户机和服务器之间仍将使用上一节说明的三个函数：serv_listen、serv_accept和cli_conn。这一服务器将表明：一个服务器可以处理多个客户机，为此使用的技术是12.5节中说明的select和poll函数。

本节所述的客户机类似于15.4节中的客户机。确实，文件main.c是完全相同的（见程序15-12）。在open.h头文件（见程序15-11）中则加了下面1行：

```
#define CS_OPEN "/home/stevens/open" /* server's well-known name */
```

因为在这里调用的是cli_conn而非fork和exec，所以文件open.c与程序15-13完全不同。这示于程序15-25中。

程序15-25 csopen函数

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int
```

```
csopen(char *name, int oflag)
{
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ( (csfd = cli_conn(CS_OPEN)) < 0)
            err_sys("cli_conn error");
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;
                        /* null at end of buf always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len)
        err_sys("writev error");

                    /* read back descriptor */
                    /* returned errors handled by write() */
    return( recv_fd(csfd, write) );
}
```

客户机与服务器之间使用的协议仍然相同。

让我们先查看服务器。头文件opend.h（见程序15-26）包括了标准头文件，并且说明了全局变量和函数原型。

程序15-26 open.h头文件

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CS_OPEN "/home/stevens/opend" /* well-known name */
#define CL_OPEN "open"                /* client's request for server */

            /* declare global variables */
extern int   debug;     /* nonzero if interactive (not daemon) */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open flag: O_xxx ... */
extern char *pathname;  /* of file to open for client */

typedef struct {    /* one Client struct per connected client */
  int   fd;     /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */
                    /* (both manipulated by client_XXX() functions) */

            /* function prototypes */
int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     request(char *, int, int, uid_t);
```

因为此服务器处理所有客户机，所以它必须保存每个客户机连接的状态。这是用定义在opend.h头文件中的client数组实现的。程序15-27定义了三个处理此数组的函数。

程序15-27 处理client数组的三个函数

```
#include    "opend.h"

#define NALLOC  10      /* #Client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size + NALLOC) * sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

            /* have to initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}
/* Called by loop() when connection request from a new client arrives */

int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }
            /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/* Called by loop() when we're done with a client */

void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

第一次调用client_add时，它调用client_alloc、client_alloc又调用malloc为该数组的10个登记项分配空间。在这10个登记项全部用完后，再调用client_add，使realloc分配附加空间。依靠这种动态空间分配，我们无需在编译时限制client数组的长度。

如果出错，那么因为假定服务器是精灵进程，所以这些函数调用log_函数（见附录B）。

main函数（见程序15-28）定义全局变量，处理命令行选择项，然后调用loop函数。如果以-d选择项调用服务器，则它以交互方式运行而非精灵进程。当测试些服务器时，使用交互运行方式。

程序15-28 main函数

```
#include    "opend.h"
#include    <syslog.h>

            /* define global variables */
int       debug;
char      errmsg[MAXLINE];
int       oflag;
char     *pathname;
Client   *client = NULL;
int       client_size;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

    loop();     /* never returns */
}
```

loop函数是服务器的无限循环。我们将给出该函数的两种版本。程序15-29是使用select的一种版本。（在4.3+BSD和SVR4之下工作），程序15-30是使用poll（用于SVR4）的另一种版本。

程序15-29 使用select的loop函数

```
#include    "opend.h"
#include    <sys/time.h>

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

FD_ZERO(&allset);

            /* obtain fd to listen for client requests on */
```

```
        if ( (listenfd = serv_listen(CS_OPEN)) < 0)
            log_sys("serv_listen error");
        FD_SET(listenfd, &allset);
        maxfd = listenfd;
        maxi = -1;

        for ( ; ; ) {
            rset = allset;      /* rset gets modified each time around */
            if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
                log_sys("select error");

            if (FD_ISSET(listenfd, &rset)) {
                        /* accept new client request */
                if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                    log_sys("serv_accept error: %d", clifd);
                i = client_add(clifd, uid);
                FD_SET(clifd, &allset);
                if (clifd > maxfd)
                    maxfd = clifd;  /* max fd for select() */
                if (i > maxi)
                    maxi = i;       /* max index in client[] array */
                log_msg("new connection: uid %d, fd %d", uid, clifd);
                continue;
            }

            for (i = 0; i <= maxi; i++) {   /* go through client[] array */
                if ( (clifd = client[i].fd) < 0)
                    continue;
                if (FD_ISSET(clifd, &rset)) {
                            /* read argument buffer from client */
                    if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                        log_sys("read error on fd %d", clifd);
                    else if (nread == 0) {
                        log_msg("closed: uid %d, fd %d",
                                        client[i].uid, clifd);
                        client_del(clifd);  /* client has closed conn */
                        FD_CLR(clifd, &allset);
                        close(clifd);
                    } else          /* process client's rquest */
                        request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
```

此函数调用serv_listen以创建服务器对于客户机连接的端点。此函数的其余部分是一个循环，它以select调用开始。在select返回后，两个条件可能为真：

(1) 描述符listenfd可能准备好读，这意味着新客户机已调用了cli_conn。为了处理这种情况。我们将调用serv_accept，然后更新client数组以及与该新客户机相关的簿记消息。（跟踪作为select第一个参数的最高描述符编号。也跟踪使用中的client数组的最高下标。）

(2) 一个现存的客户机的连接可能准备好读。这意味这下列两事件之一：(a) 该客户机已经终止，或（b）该客户机已发送一新请求。如果 read返回0（文件结束），则可认为一客户机终止。如果读返回值大于0则可判定有一新请求需处理，调用request处理此新的客户机请求。

用allset描述符集跟踪当前使用的描述符。当新客户机连至服务器时，此描述符集的适当位被打开。当该客户机终止时，适当位就被关闭。

因为客户机的所有描述符都由内核自动关闭（包括与服务器的连接），所以我们知道什么时候一客户机终止，该终止是否自愿。这与系统V IPC机构不同。

使用poll函数的loop函数示于程序15-30中。

程序15-30 使用poll的loop函数

```
#include    "opend.h"
#include    <poll.h>
#include    <stropts.h>

void
loop(void)
{
    int             i, maxi, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;

    if ( (pollfd = malloc(open_max() * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");

                /* obtain fd to listen for client requests on */
    if ( (listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);     /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;
    pollfd[0].events = POLLIN;
    maxi = 0;
     for ( ; ; ) {
         if (poll(pollfd, maxi + 1, INFTIM) < 0)
             log_sys("poll error");

         if (pollfd[0].revents & POLLIN) {
                     /* accept new client request */
             if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                 log_sys("serv_accept error: %d", clifd);
             i = client_add(clifd, uid);
             pollfd[i].fd = clifd;
             pollfd[i].events = POLLIN;
             if (i > maxi)
                 maxi = i;
             log_msg("new connection: uid %d, fd %d", uid, clifd);
         }

         for (i = 1; i <= maxi; i++) {
             if ( (clifd = client[i].fd) < 0)
                 continue;
             if (pollfd[i].revents & POLLHUP)
                 goto hungup;
             else if (pollfd[i].revents & POLLIN) {
                         /* read argument buffer from client */
                 if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                     log_sys("read error on fd %d", clifd);
                 else if (nread == 0) {
    hungup:
                     log_msg("closed: uid %d, fd %d",
                                         client[i].uid, clifd);
                     client_del(clifd);  /* client has closed conn */
                     pollfd[i].fd = -1;
                     close(clifd);
                 } else          /* process client's rquest */
                     request(buf, nread, clifd, client[i].uid);
             }
         }
     }
}
```

为使打开描述符的数量能与客户机数量相当，我们动态地为 pollfd结构分配空间（函数open_max见程序2-3）。

client数组的第0个登记项用于listenfd描述符。于是，client数组中的客户机下标号与pollfd数组中所用的下标号相同。新客户机连接的到达由listenfd描述符中的POLLIN指示。如同前述，调用serv_accept以接收该连接。

对于一个现存的客户机，应当处理来自poll的两个不同事件：客户机终止由POLLHUP指示，以及来自现存客户的一个新要求由 POLLIN指示。回忆习题14.7，在还有数据在流首可读时，挂起消息能到达流首。对于管道，在处理挂起前，我们希望先读所有数据。但是对于服务器，当从客户机接收到挂起消息时，能关闭该流连接，于是也就丢弃了仍在流上的所有数据。因为已经不能回送任何响应，所以也就没有理由再去处理仍在流上的任何请求。

如同本函数的select版本，调用request函数（见程序15-31）处理来自客户机的新请求。此函数类似于其早期版本（见程序15-16）。它调用同一函数buf_args（见程序15-17），buf_args又调用cli_args（见程序15-18）。

程序15-31 request函数

```
#include    "opend.h"
#include    <fcntl.h>

void
request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request from uid %d not null terminated: %*.*s\n",
                                    uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);

            /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

            /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

这样就完成了open服务器，它使用一个精灵进程处理所有的客户机请求。

15.7 小结

本章集中讨论了如何在进程间传送文件描述符，及服务器如何接受来自客户机的连接，并演示了SVR4和4.3+BSD中的实现过程。目前大多数UNIX系统都提供这些高级IPC功能。第18章还将再次用到本章所述的函数。

本章给出了open服务器的两个版本。一个版本由客户机用fork和exec直接调用，另一版本为处理所有客户机请求的精灵进程服务器。这两个版本均采用了15.3节所述的文件描述符传送和接收函数。第二个版本还采用了15.5节所述的客户机-服务器连接函数以及12.5节所述的I/O多路转接函数。

习题

15.1 改写程序15-1，要求是：对于流管道使用标准I/O库函数代替read和write。

15.2 使用本章说明的文件描述符传送函数以及8.8节中说明的父-子进程同步例程，编写具有下列功能的程序：该程序调用fork，然后子进程打开一现存文件并将打开的描述符传给父进程。父进程读该文件的当前位移量，并打印它以便验证。若此文件如上述从子进程传递到父进程，则父、子进程应共享同一文件表项，所以当子进程每次更改该文件当前位移量，那么这种更改同样影响到父进程的描述符。使子进程将该文件定位至一个不同位移量，并通知父进程。

15.3 程序15-14和15.15分别定义和说明了全局变量，两者的区别是什么？

15.4 改写bug_args函数（见程序15-17），删除其中对argv数组长度的编译时间限制。请用动态存储分配。

15.5 说明优化程序15-29和程序15-30中loop函数的方法，并实现之。