第19章 伪 终 端

19.1 引言

第9章介绍进行终端登录时，需要通过一个终端设备自动提供终端的语义。在终端和运行程序之间有一个终端行规程（见图 11-2），通过这个规程我们能够在终端上设置特殊字符（退格、行删除、中断等）。但是，当一个登录请求到达网络连接时，终端行规程并不是自动被加载到网络连接和登录程序 shell之间的。图9-5显示了一个伪终端设备驱动程序被用来提供终端语义。

除了用于网络登录，伪终端还被用在其他方面，本章将对此进行介绍。我们将首先提供SVR4和4.3+BSD系统下用于创建伪终端的函数，然后使用这些函数编写一个程序用来调用 pty。我们将看到这个程序的各种使用：在输入字符和终端显示之间进行转换（ BSD的码转换程序）和运行协同进程来避免程序14-10中遇到的缓存问题。

19.2 概述

伪终端（pseudo terminal）这个名词暗示了与一个应用程序相比，它更加像一个终端。但事实上，伪终端并不是一个真正的终端。图19-1显示了使用伪终端的进程的典型结构。其中关键点如下：

(1) 通常一个进程打开伪终端主设备然后调用fork。子进程建立了一个新的对话，打开一个相应的伪终端从设备，将它复制成标准输入、标准输出和标准出错，然后调用 exec。伪终端从设备成为子进程的控制终端。

(2) 对于伪终端从设备之上的用户进程来说，其标准输入、标准输出和标准出错都能当作终端设备使用。用户进程能够调用第 11章中讲到的所有输入/输出函数。但是因为在伪终端从设备之下并没有真正的设备，无意义的函数调用（改变波特率、发送中断符、设置奇偶校验等）将被忽略。

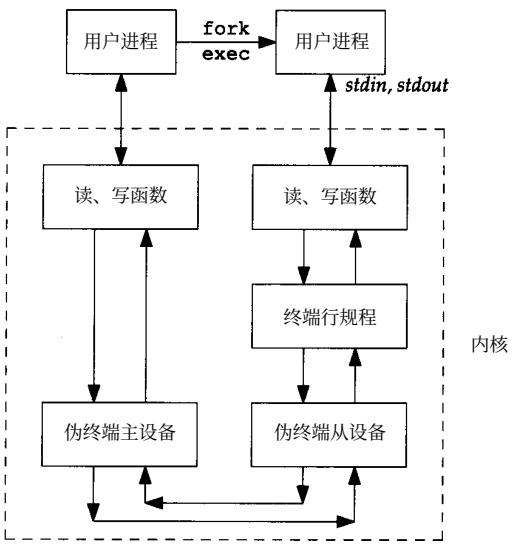

图19-1 典型的伪终端进程结构

(3) 任何写到伪终端主设备的输入都会作为从设备端的输入，反之亦然。事实上所有从设备端的输入都来自于主设备上的用户进程。这看起来就像一个流管道（见图 15-3），但从设备上的终端行规程使我们拥有普通管道之外的其他处理能力。

图19-1显示了BSD系统中的伪终端结构。19.3.2节将介绍如何打开这些设备。在SVR4系统中伪终端是使用流系统来创建的（见 12.4节）。图19-2详细描述了SVR4系统中各个伪终端模块

之间的关系。虚线框中的两个流模块是可选的。请注意从设备上的三个流模块同程序 12-10（网络登录）的输出是一样的。19.3.1节将介绍如何组织这些流模块。

从现在开始将简化以上图示，首先不再画出图 19-1的“ 读、写功能 ”或图 19-2的流首。使用缩写“ pty ”表示伪终端，并将图 19-2中所有伪终端从设备之上的流模块集合表示为“ 终端行规程 ”模块。

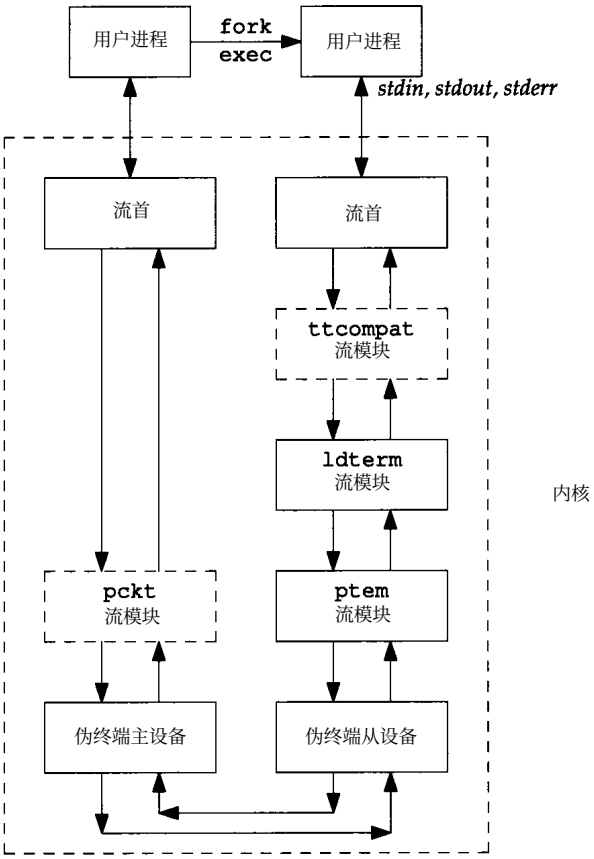

图19-2 SVR4下的伪终端结构

19.2.1 网络登录服务器

伪终端用于构造网络登录服务器。典型的例子是 telnetd和rlogind服务器。Stevens〔1990〕第15章详细讨论了提供rlogin服务的步骤。一旦登录shell运行在远端主机上，即可得到如图 19-3的结构。同样的结构也用于telnetd服务器。

在rlogind服务器和登录shell之间有两个exec调用，这是因为login程序通常是在两个exec之间检验用户是否合法的。

本图的一个关键点是驱动伪终端主设备的进程通常同时在读写另一个 I/O流。本例中另一个I/O流是TCP/IP。这表示该进程必然使用了某种形式的如 select或poll那样的I/O多路转接（见12.5节），或被分成两个进程。回忆18.7节讨论过的一个进程和两个进程的比较。

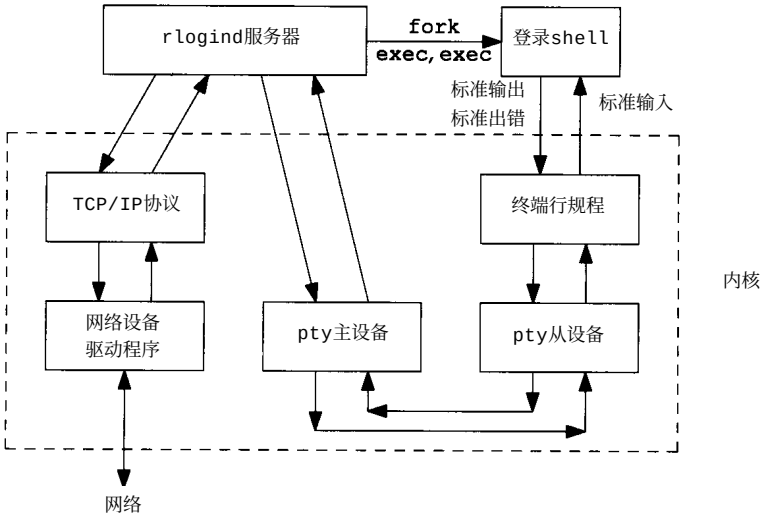

图19-3 rlogind服务器的进程组织结构

19.2.2 script程序

script程序是随SVR4和4.3+BSD提供的，该程序将终端对话期间所有的输入和输出信息在一个文件中做一个拷贝。它通过将自己置于终端和登录 shell的一个新的调用之间来完成这个工作。图19-4详细描述了script程序相关的交互。这里特别指出 script程序通常是从登录shell起动的，该shell然后等待程序的结束。

script程序运行时，在伪终端从设备之上终端行规程的所有输出都被复制到一个 script文件中（通常叫做typescript)。因为击键通常被行规程的模块回显，该 script文件也包括了输入的内容。但是，因为口令不被回显，该script文件不会包含口令。

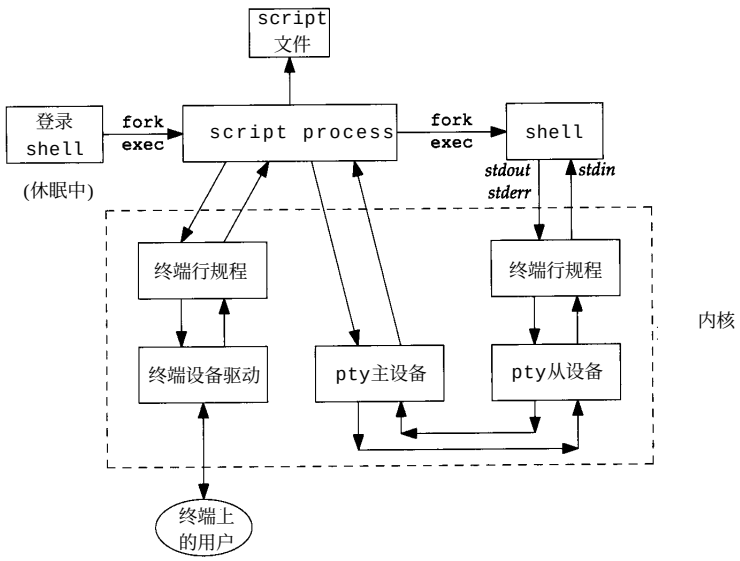

图19-4 script程序

本书中所有运行程序并显示其输出的实例都是由 script程序实现的，这样避免了手动拷贝程序输出可能带来的错误。

在19.5节开发一个通用的pty程序后，我们将看到一个巧妙的shell脚本能够将它转化成一个script程序。

19.2.3 expect程序

伪终端可以用来使交互式的程序运行在非交互的状态中。许多程序需要一个终端来运行，18.7节中的call进程就是一个例子。它假定标准输入是一个终端并在起动时将其设置为初始模式（见程序18-20）。该程序不能被shell脚本用来运行以自动拨号到远程系统，登录，取出信息和注销。

同修改所有交互式程序来支持批处理模式的操作比较，一个更好的解决方法是通过一个script来驱动交互式程序。expect程序[Libes 1990;1991]提供了这样的方法。类似于19.5节的pty程序，它使用伪终端来运行其他程序。并且，expect还提供了一种编程语言用于检查程序的输出，以确定用什么作为输入发送给该程序。当一个交互式的程序开始从一个脚本运行时，不能仅仅是将脚本中的所有内容输入到程序中去。相应的，要通过检查程序的输出来决定下一步输入的内容。

19.2.4 运行协同进程

在程序14-10所示的例子中，不能调用使用标准I/O库进行输入、输出的协同进程，这是因为当通过管道与协同进程进行通讯时，标准 I/O库会将标准输入和输出的内容放到缓存中，从而引起死锁。如果协同进程是一个已经编译的程序而我们又没有源程序，则无法在源程序中加入fflush语句来解决这个问题。图8显示了一个进程驱动协同进程的情况。我们只需将一个伪终端放到两个进程之间，见图19-5。

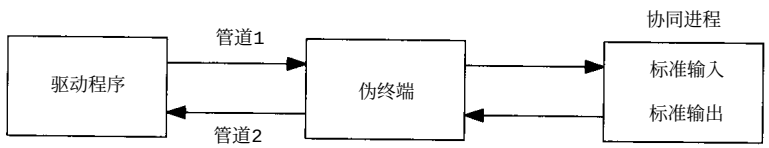

图19-5 用伪终端驱动一个协同进程

现在协同进程的标准输入和标准输出就像终端设备一样，所以标准 I/O库会将这两个流设置为行缓存。

父进程有两种不同的方法在自身和协同进程之间获得伪终端（这种情况下的父进程可以类似程序14-9，使用两个管道和协同进程进行通讯；或者像程序 15-1那样，使用一个流管道）。一个方法是父进程直接调用 pty_fork函数（见19.4节）而不是fork。另一种方法是exec该pty程序，将协同进程作为参数（见19.5节）。我们将在说明pty程序后介绍这两种方法。

19.2.5 观看长时间运行程序的输出

使用任一个标准shell，都可以将一个需要长时间运行的程序放到后台运行。但是如果将该程序的标准输出重定向到一个文件，并且如果它产生的输出不多，我们就不能方便地监控程序

的进展，这是因为标准I/O库会将标准输出放在缓存中保存。我们看到的将只是成块的输出结果，有时甚至可能是8192字节一块。

如果有源程序，则可以加入fflush调用。另一种方法是，可以在pty程序下运行该程序，让标准I/O库认为输出是终端。图19-6说明了这个结构，我们将这个缓慢输出的程序称为slowout。从登录shell到pty进程的fort/exec箭头用虚线表示，以强调pty进程是作为后台任务运行的。

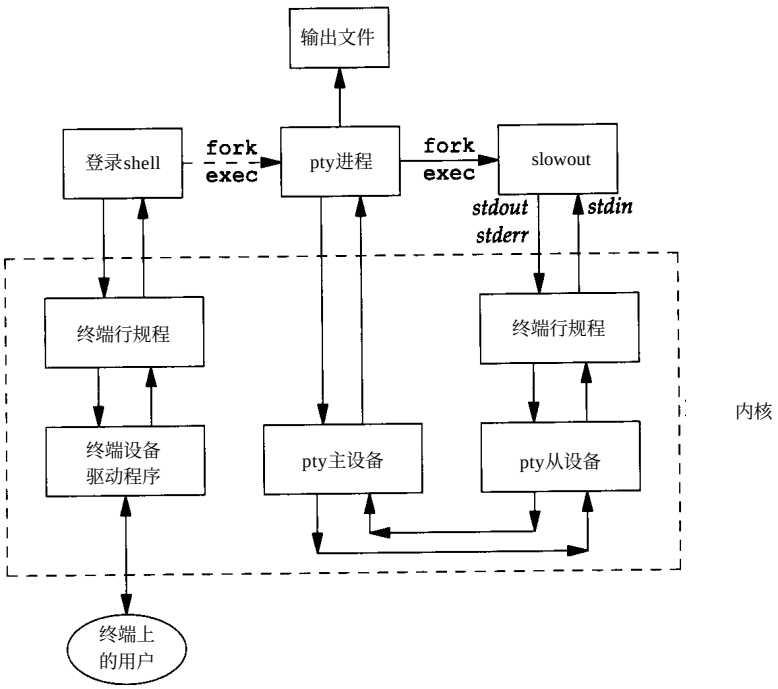

图19-6 使用伪终端运行一个缓慢输出的程序

19.3 打开伪终端设备

在SVR4和4.3+BSD系统中打开伪终端设备的方法有所不同。我们提供两个函数来处理所有细节：ptym_open用来打开下一个有效的伪终端主设备，ptys_open用来打开相应的从设备。

```
#include "ourhdr.h"

int ptym_open(char *pts_name);
                                返回：若成功则为伪终端主设备文件描述符，否则为 -1

int ptys_open(int fdm, char *pts_name);
                                返回：若成功则为伪终端从设备文件描述符，否则为 -1
```

通常我们不直接调用这两个函数——函数pty_fork（见19.4节）调用它们并fork出一个子进程。

ptym_open决定下一个有效的伪终端主设备并打开该设备。这个调用必须分配一个数组来存放主设备或从设备的名称，并且如果调用成功，相应的主设备或从设备的名称会通过pts_name返回。这个名称和ptym_open返回的文件描述符将传给ptys_open，该函数用来打开一个从设备。

在讲解pty_fork函数之后，使用两个函数来打开这两个设备的原因将会很明显。通常，一个进程调用ptym_open来打开一个主设备并且得到从设备的名称。该进程然后 fork子进程，子进程在调用setsid建立新的对话后调用ptys_open来打开从设备。这就是从设备如何成为子进程的控制终端的过程。

19.3.1 SVR4

SVR4系统下所有伪终端的流实现细节在 AT&T〔1990d〕第12章中有所说明，还描述了以下三个函数：grantpt(3)，unlockpt(3)，和ptsname(3)。

伪终端主设备是/dev/ptmx。这是一个流的增殖设备（clone device）。这意味着当我们打开该增殖设备，其open例程自动决定第一个未被使用的伪终端主设备并打开这个设备。（下一节将看到在伯克利系统中，我们必须自己找到第一个未被使用的伪终端主设备。）

程序19-1 SVR4的伪终端打开函数

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

extern char *ptsname(int);  /* prototype not in any system header */

int
ptym_open(char *pts_name)
{
    char    *ptr;
    int     fdm;

    strcpy(pts_name, "/dev/ptmx");  /* in case open fails */
    if ( (fdm = open(pts_name, O_RDWR)) < 0)
        return(-1);

    if (grantpt(fdm) < 0) {     /* grant access to slave */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) {    /* clear slave's lock flag */
        close(fdm);
        return(-3);
    }
    if ( (ptr = ptsname(fdm)) == NULL) {    /* get slave's name */
        close(fdm);
        return(-4);
    }

    strcpy(pts_name, ptr);  /* return name of slave */
    return(fdm);            /* return fd of master */
}

int
ptys_open(int fdm, char *pts_name)
{
    int     fds;

            /* following should allocate controlling terminal */
    if ( (fds = open(pts_name, O_RDWR)) < 0) {
        close(fdm);
```

```
        return(-5);
    }
    if (ioctl(fds, I_PUSH, "ptem") < 0) {
        close(fdm);
        close(fds);
        return(-6);
    }
    if (ioctl(fds, I_PUSH, "ldterm") < 0) {
        close(fdm);
        close(fds);
        return(-7);
    }
    if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
        close(fdm);
        close(fds);
        return(-8);
    }

    return(fds);
}
```

首先打开设备/dev/ptmx并得到伪终端主设备的文件描述符。打开这个主设备自动锁定了对应的从设备。

然后调用grantpt来改变从设备的许可权。执行如下操作：(a) 将从设备的所有权改为有效用户ID；(b) 将组所有权改为组tty；(c) 将许可权改为只允许用户-读，用户-写和组-写。将组所有权设置为tty并允许组-写许可权是因为程序wall(1)和write(1)的设置-用户-ID为组tty。调用函数grantpt执行/usr/lib/pt_chmod。该程序的设置-用户-ID为root，因此它能够修改从设备的所有者和许可权。

函数unlockpt用来清除从设备的内部锁。在打开从设备前必须做这件事情。并且必须调用ptsname来得到从设备的名称。这个名称的格式是/dev/pts/*NNN*。

文件中接下来的函数是ptys_open，该函数真正被用来打开一个从设备。在SVR4系统中，如果调用者是一个还没有控制终端的对话期首进程，open就会分配一个从设备作为控制终端。如果不希望函数自动做这件事，可以在调用时指明O_NOCTTY标志。

打开从设备后，将三个流模块放在从设备的流上。ptem是伪终端虚拟模块，ldterm是终端行规程模块。这两个模块合在一起像一个真正的终端模块一样工作。ttcompat提供了向老系统如V7、4BSD和Xenix的ioctl调用的兼容性。这是一个可选的模块，但是因为它自动尝试控制台登录和网络登录（见程序12-10的输出），我们将其加到从设备的流中。

调用这两个函数的结果是得到：伪终端主设备的文件描述符和从设备的文件描述符。

19.3.2 4.3+BSD

在4.3+BSD系统中必须自己来确定第一个可用的伪终端主设备。为达到这个目的，从/dev/ptyp0开始并不断尝试，直到成功打开一个可用的伪终端主设备或试完所有设备。在打开设备时，将看到两种可能的错误：EIO指设备已经被使用；ENOENT表示设备不存在。对于后一种情况，可以停止搜索，因为所有的伪终端设备都在被使用中。一旦成功地打开一个例如名为/dev/pty*MN*的伪终端主设备，那么对应的从设备的名称为/dev/tty*MN*。

程序19-2中的函数ptys_open打开该从设备。我们在该函数中调用chown和chmod，必须意识到调用这两个函数的进程必须有超级用户许可权。如果必须改变所有权，那么这两个函数调用必须放在一个设置-用户-ID的root用户的可执行函数中，这类似于4.3+BSD系统下的grantpt函数。

在4.3+BSD系统之下打开pty从设备不具有像分配作为控制终端的设备那样的副作用。下一节将探讨如何在4.3+BSD系统下分配控制终端。

这个函数尝试16种不同的伪终端主设备：从/dev/ptyp0到/dev/ptyTf。具体有效的pty设备号取决于两个因素：(a) 内核中配置的号码；(b) /dev目录下的特殊文件号。对于任何程序来说，有效的号码是(a)和(b)中较小的一个。并且，即使(a)和(b)中小的值大于64，许多现有的BSD应用(如telnetd，rlogind等等)会搜索程序19-2中第一个for循环中的pqrs。

程序19-2 4.3+BSD的伪终端open函数

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

int
ptym_open(char *pts_name)
{
    int     fdm;
    char    *ptr1, *ptr2;

    strcpy(pts_name, "/dev/ptyXY");
      /* array index: 0123456789 (for references in following code) */

    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        pts_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            pts_name[9] = *ptr2;

                    /* try to open master */
            if ( (fdm = open(pts_name, O_RDWR)) < 0) {
                if (errno == ENOENT)    /* different from EIO */
                    return(-1);         /* out of pty devices */
                else
                    continue;           /* try next pty device */
            }

            pts_name[5] = 't';  /* change "pty" to "tty" */
            return(fdm);        /* got it, return fd of master */
        }
    }
    return(-1);     /* out of pty devices */
}

int
ptys_open(int fdm, char *pts_name)
{
    struct group    *grptr;
    int             gid, fds;

    if ( (grptr = getgrnam("tty")) != NULL)
        gid = grptr->gr_gid;
    else
        gid = -1;       /* group tty is not in the group file */

        /* following two functions don't work unless we're root */
    chown(pts_name, getuid(), gid);
```

```
    chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP);

    if ( (fds = open(pts_name, O_RDWR)) < 0) {
        close(fdm);
        return(-1);
    }
    return(fds);
}
```

19.4 pty_fork函数

现在使用上一节介绍的两个函数： ptym_open和ptys_open，编写我们称之为pty_fork的函数。这个新函数具有如下功能：打开主设备和从设备，建立作为对话期管理者的子进程并使其具有控制终端。

```
#include <sys/types.h>
#include <termios.h>
#include <sys/ioctl.h>    /* 4.3+BSD defines struct winsize here */
#include "ourhdr.h"

pid_t pty_fork(int *ptrfdm, char *slave_name,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

返回：子进程中为0，父进程中为子进程的进程ID，若错误则为-1

pty主设备的文件描述符通过*ptrfdm*指针返回。

如果*slave_name*不为空，从设备的名称被存放在该指针指向的存储区中。调用者必须为该存储区分配空间。

如果指针*slave_termios*不为空，该指针所引用的结构将初始化从设备的终端行规程。如果该指针为空，系统将从设备的termios结构初始化为一个由具体应用定义的初始状态。类似的，如果*slave_winsize*指针不为空，该指针所引用的结构将初始化从设备的窗口大小。如果该指针为空，winsize结构通常被初始化为0。

程序19-3显示了这个程序的代码。调用相应的 ptym_open和ptys_open函数，这个函数在SVR4和4.3+BSD系统下都可以使用。

在打开伪终端主设备后， fork将被调用。正如前面提到的，要等到调用 setid建立新的对话期后才调用ptys_open。当调用setsid时，子进程还不是一个进程组的首进程（想一想为什么？）因此9.5节列出的三个操作被使用：（a）子进程作为对话的管理者创建一个新的对话期；（ b）子进程创建一个新的进程组；（c）子进程没有控制终端。在SVR4系统中，当调用ptys_open时，从设备成为了控制终端。在 4.3+BSD系统中，必须调用 ioctl并使用参数TIOCSCTTY来分配一个控制终端。然后termios和winsize这两个结构在子进程中被初始化。最后从设备的文件描述符被复制到子进程的标准输入、标准输出和标准出错中。这表示由子进程所 exec的进程都会将上述三个句柄同伪终端从设备联系起来。

在调用fork后，父进程返回伪终端主设备的描述符并返回。下一节将在 pty程序中使用pty_fork。

程序19-3 pty_fork函数

```
#include    <sys/types.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>   /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

pid_t
pty_fork(int *ptrfdm, char *slave_name,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int     fdm, fds;
    pid_t   pid;
    char    pts_name[20];

    if ( (fdm = ptym_open(pts_name)) < 0)
        err_sys("can't open master pty: %s", pts_name);

    if (slave_name != NULL)
        strcpy(slave_name, pts_name);   /* return name of slave */

    if ( (pid = fork()) < 0)
        return(-1);

    else if (pid == 0) {        /* child */
        if (setsid() < 0)
            err_sys("setsid error");

                /* SVR4 acquires controlling terminal on open() */
        if ( (fds = ptys_open(fdm, pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm);     /* all done with master in child */

#if defined(TIOCSCTTY) && !defined(CIBAUD)
                /* 4.3+BSD way to acquire controlling terminal */
                /* !CIBAUD to avoid doing this under SunOS */
        if (ioctl(fds, TIOCSCTTY, (char *) 0) < 0)
            err_sys("TIOCSCTTY error");
#endif
                /* set slave's termios and window size */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }
                /* slave becomes stdin/stdout/stderr of child */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds > STDERR_FILENO)
            close(fds);
        return(0);      /* child returns 0 just like fork() */

    } else {                    /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

19.5 pty程序

编写pty程序的目的是为了用键入：

```
pty prog arg1 arg2
```

来代替：

```
prog arg1 arg2
```

这样使我们可以用pty来执行另一个程序，该程序在一个自己的对话期中执行，并和一个伪终端连接。

看一下pty程序的源码。程序19-4包含main函数。它调用上一节的pty_fork函数。

程序19-4 pty程序的main函数

```
#include    <sys/types.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>   /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

static void set_noecho(int);    /* at the end of this file */
void        do_driver(char *);  /* in the file driver.c */
void        loop(int, int);     /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int             fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t           pid;
    char            *driver, slave_name[20];
    struct termios  orig_termios;
    struct winsize  size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d:einv")) != EOF) {
        switch (c) {
        case 'd':       /* driver for stdin/stdout */
            driver = optarg;
            break;

        case 'e':       /* noecho for slave pty's line discipline */
            noecho = 1;
            break;

        case 'i':       /* ignore EOF on standard input */
            ignoreeof = 1;
            break;

        case 'n':       /* not interactive */
            interactive = 0;
            break;

        case 'v':       /* verbose */
            verbose = 1;
            break;

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

```
            }
        }
        if (optind >= argc)
            err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");

        if (interactive) {  /* fetch current termios and window size */
            if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
                err_sys("tcgetattr error on stdin");
            if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
                err_sys("TIOCGWINSZ error");
            pid = pty_fork(&fdm, slave_name, &orig_termios, &size);

        } else
            pid = pty_fork(&fdm, slave_name, NULL, NULL);

        if (pid < 0)
            err_sys("fork error");

        else if (pid == 0) {        /* child */
            if (noecho)
                set_noecho(STDIN_FILENO);   /* stdin is slave pty */

            if (execvp(argv[optind], &argv[optind]) < 0)
                err_sys("can't execute: %s", argv[optind]);
        }

        if (verbose) {
            fprintf(stderr, "slave name = %s\n", slave_name);
            if (driver != NULL)
                fprintf(stderr, "driver = %s\n", driver);
        }

        if (interactive && driver == NULL) {
            if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
                err_sys("tty_raw error");
            if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
                err_sys("atexit error");
        }

        if (driver)
            do_driver(driver);  /* changes our stdin/stdout */

        loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

        exit(0);
    }

    static void
    set_noecho(int fd)      /* turn off echo (for slave pty) */
    {
        struct termios  stermios;

        if (tcgetattr(fd, &stermios) < 0)
            err_sys("tcgetattr error");

        stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
        stermios.c_oflag &= ~(ONLCR);
                /* also turn off NL to CR/NL mapping on output */

        if (tcsetattr(fd, TCSANOW, &stermios) < 0)
            err_sys("tcsetattr error");
    }
```

下一节检测pty程序的不同使用时，将会探讨多种的行命令选择项。

在调用pty_fork前，我们取得了termios和winsize结构的值，将其传递给pty_fork。通过这种方法，伪终端从设备具有和现在的终端相同的初始状态。

从pty_fork返回后，子进程关闭了伪终端从设备的回显，并调用 execvp来执行命令行指定的程序。所有的命令行参数将成为程序的参数。

父进程在调用exit时执行原先设置的退出处理程序，它复原终端状态，将用户终端设置为初始模式（可选）。下一节将讨论do_driver函数。

接下来父进程调用函数loop（见程序19-5）。该函数仅仅是将所有标准输入拷贝到伪终端主设备，并将伪终端主设备接收到的所有内容拷贝到标准输出。同 18.7节一样，我们有两个选择——一个进程还是两个进程？为了有所区别，这里使用两个进程，尽管使用 select或poll的单进程也是可行的。

程序19-5 loop函数

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

#define BUFFSIZE    512

static void sig_term(int);
static volatile sig_atomic_t    sigcaught;  /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t   child;
    int     nread;
    char    buff[BUFFSIZE];
    if ( (child = fork()) < 0) {
        err_sys("fork error");

    } else if (child == 0) {    /* child copies stdin to ptym */
        for ( ; ; ) {
            if ( (nread = read(STDIN_FILENO, buff, BUFFSIZE)) < 0)
                err_sys("read error from stdin");
            else if (nread == 0)
                break;      /* EOF on stdin means we're done */

            if (writen(ptym, buff, nread) != nread)
                err_sys("writen error to master pty");
        }

            /* We always terminate when we encounter an EOF on stdin,
               but we only notify the parent if ignoreeof is 0. */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM);   /* notify parent */
        exit(0);    /* and terminate; child can't return */
    }

        /* parent copies ptym to stdout */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ( (nread = read(ptym, buff, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */

        if (writen(STDOUT_FILENO, buff, nread) != nread)
            err_sys("writen error to stdout");
    }

    /* There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error. */

    if (sigcaught == 0) /* tell child if it didn't send us the signal */
```

```
        kill(child, SIGTERM);
    return;     /* parent returns to caller */
}

/* The child sends us a SIGTERM when it receives an EOF on
 * the pty slave or encounters a read() error. */

static void
sig_term(int signo)
{
    sigcaught = 1;      /* just set flag and return */
    return;             /* probably interrupts read() of ptym */
}
```

注意，当使用两个进程时，如果一个终止，那么它必须通知另一个。我们用 SIGTERM进行这种通知。

19.6 使用pty程序

接下来看一下pty程序的不同例子，了解一下使用不同命令行选择项的必要性。

如果使用KornShell，我们执行：

pty ksh

得到一个运行在一个伪终端下的新的shell。

如果文件ttyname同程序11-7相同，可按如下方式执行pty程序：

```
$ who
stevens  console  Feb 6 10:43
stevens  ttyp0    Feb 6 15:00
stevens  ttyp1    Feb 6 15:00
stevens  ttyp2    Feb 6 15:00
stevens  ttyp3    Feb 6 15:48
stevens  ttyp4    Feb 7 14:28          ttyp4是正在使用的最高终端设备
$ pty ttyname                          在pty上运行程序11-7
fd 0: /dev/ttyp5                       ttyp5是下一个有效的pty设备号
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

19.6.1 utmp文件

6.7节讨论了记录当前UNIX系统登录用户的utmp文件。那么在伪终端上运行程序的用户是否被认为登录了呢？如果是远程登录，telnetd和rlogind，显然伪终端上的用户应该在utmp中拥有相应条目。但是，从窗口系统或运行 script程序，在伪终端上运行 shell的用户是否应该在utmp中拥有相应条目呢？这个问题一直没有一个统一的认识。有的系统有记录，有的没有。如果没有记录的话，who(1)程序一般不会显示正在被使用的伪终端。

除非utmp允许其他用户的写许可权，否则一般的程序将不能对其进行写操作。某些系统提供这个写许可权。

19.6.2 作业控制交互

当在pty上运行作业控制shell时，它能够正常地运行。例如，

```
pty ksh
```

在pty上运行KornShell。我们能够在这个新shell下运行程序和使用作业控制，如同在登录 shell中一样。但如果在pty下运行一个交互式程序而不是作业控制shell，比如：

```
pty cat
```

一切正常直到键入作业控制的暂停字符。在 SVR4和4.3+BSD系统中作业控制暂停字符将会被显示为^Z而被忽略。在SunOS4.1.2中，cat进程终止，pty进程终止，回到初始登录shell。

为了明白其中的原因，我们需要检查所有相关的进程、这些进程所属的进程组和对话期。图19-7显示了pty cat运行的结构图。

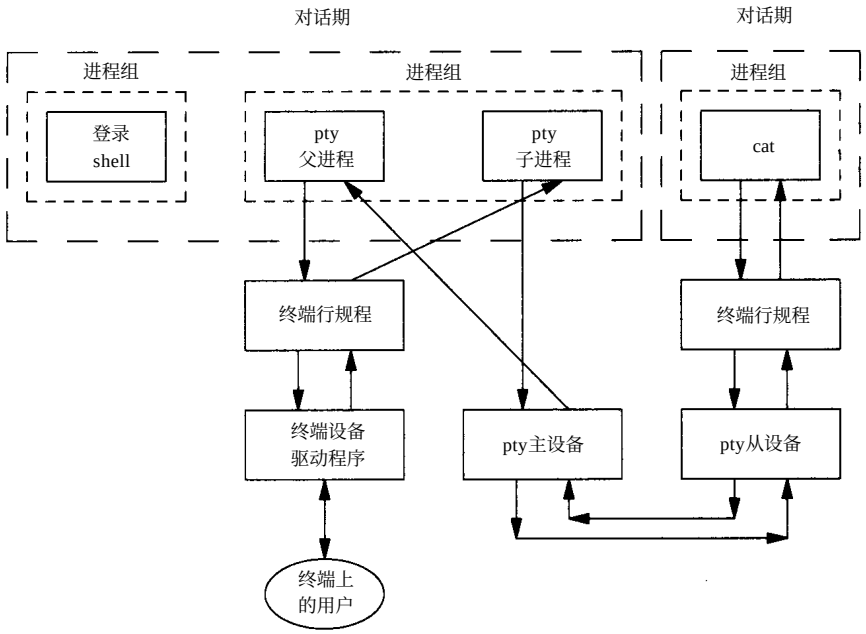

图19-7 pty cat的进程组和对话期

当键入暂停字符（Ctrl-Z），它将被cat进程下的行规程模块所识别，这是因为 pty将终端（在pty父进程之下）设置为初始模式。但内核不会终止 cat进程，这是因为它属于一个孤儿进程组（见9.10节）。cat的父进程是pty的父进程，属于另一个对话期。

不同的系统处理这种情况的方法也不同。在POSIX.1中这个SIGTSTP信号不被发送给进程。早期的伯克利系统递送一个进程甚至不能捕获的 SIGKILL。这就是我们在SunOS4.1.2中看到的。（POSIX.1 Rationale建议用SIGHUP作为更好的替代，因为进程能够捕获它。）用程序8-17查看cat进程的终止状态，可以发现该进程确实由 SIGKILL信号终止。

在SVR4和4.3+BSD系统中，我们修改了程序 10-22来观察结果。修改后的程序能够在捕获SIGTSTP后进行打印，并在捕获 SIGCONT信号后再打印一次并继续执行。这说明 SIGTSTP被进程捕获，但是当进程试图发送信号给自己来暂停本进程的时候，内核立即发送 SIGCONT信号使之继续执行。内核将不会让进程被作业控制停止。 SVR4和4.3+BSD系统的这种处理方法比起发送一个SIGKILL的方法来显得不那么激烈。

当使用pty来运行作业控制shell时，被这个新shell调用的作业将不是任何孤儿进程组的成员，这是因为作业控制 shell总是属于同一个对话期。在这种情况下， Ctrl-Z被发送到被shell调用的进程，而不是shell本身。

让被pty调用的进程能够处理作业控制信号的唯一的方法是：另外增加一个命令行标志让pty子进程能够自己认识作业暂停字符，而不是让该字符通过其他行规程模块。

19.6.3 检查长时间运行程序的输出

另一个使用pty进行作业控制交互的例子见图19-6。如果运行一个程序：

```
pty slowout > file.out &
```

当子进程试图从标准输入（终端）读入数据时， pty进程立刻停止运行。这是因为该作业是一个后台作业并且当它试图访问终端时会使作业控制停止。如果将标准输入重定向使得 pty不从终端读取数据，如：

```
pty slowout < /dev/null > file.out &
```

那么pty程序立即终止，这是因为它从标准输入读取到一个文件结束符。解决这个问题的方法是使用-i选择项。这个选择项的含义是忽略来自标准输入的文件结束符：

```
pty -i slowout < /dev/null > file.out &
```

这个标志导致在遇到文件结束符时，程序 19-5的子进程终止，但子进程不会使父进程也终止。相反的，父进程一直将伪终端从设备的输出拷贝到标准输出（本例中的 file.out）。

19.6.4 script程序

使用pty程序，可以用下面的方式实现BSD系统中的script（1）程序。

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

一旦执行这个script程序，即可以运行ps来观察进程之间的关系。图19-8显示了这些关系。

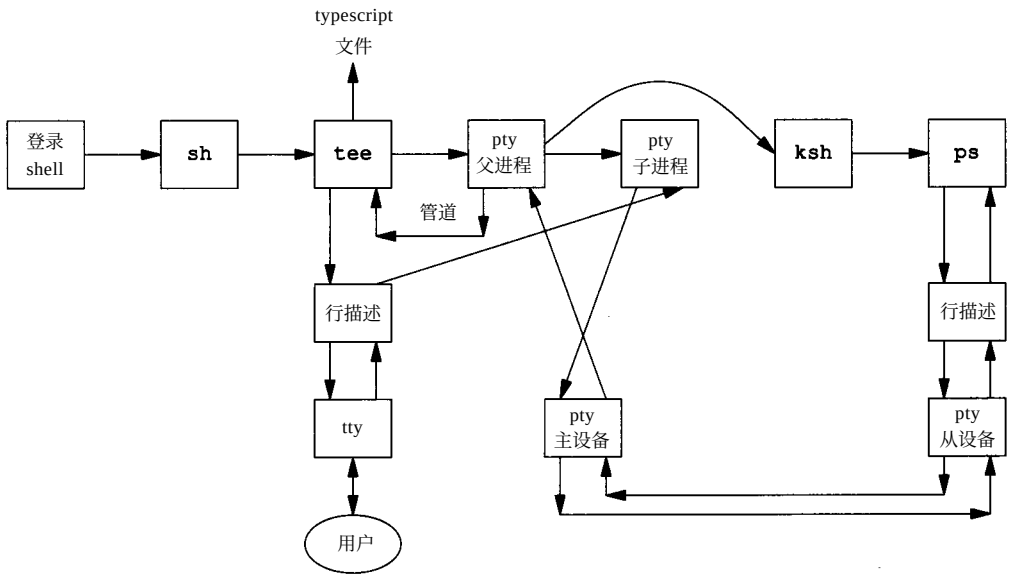

图19-8 script shell脚本

在这个例子中，假设SHELL变量是KornShell（可能是/din/ksh）。如前面所述， script仅仅是将新的shell（和它调用的所有的子进程）的输出拷贝出来，但是因为伪终端从设备上的行规程模块通常允许回显，故绝大多数键入都被写到typescript文件中去。

19.6.5 运行协同进程

在程序14-9中，我们不能让协同进程使用标准I/O函数，其原因是标准输入和输出不是终端，其输入和输出将被放到缓存中。如果用

```
if (execl("./pty", "pty", "-e", "add2", (char *) 0) < 0)
```

替代：

```
if (execl("./add2", "add2", (char *) 0) < 0)
```

在pty下运行协同进程，该程序即使使用了标准I/O仍然可以正确运行。

图19-9显示了在使用伪终端作为协同进程的输入和输出的情况。框中的“驱动程序”是前面提到过的改变了execl的程序14-9。这是图19-5的一个扩充，它显示了所有的进程间联系和数据流。

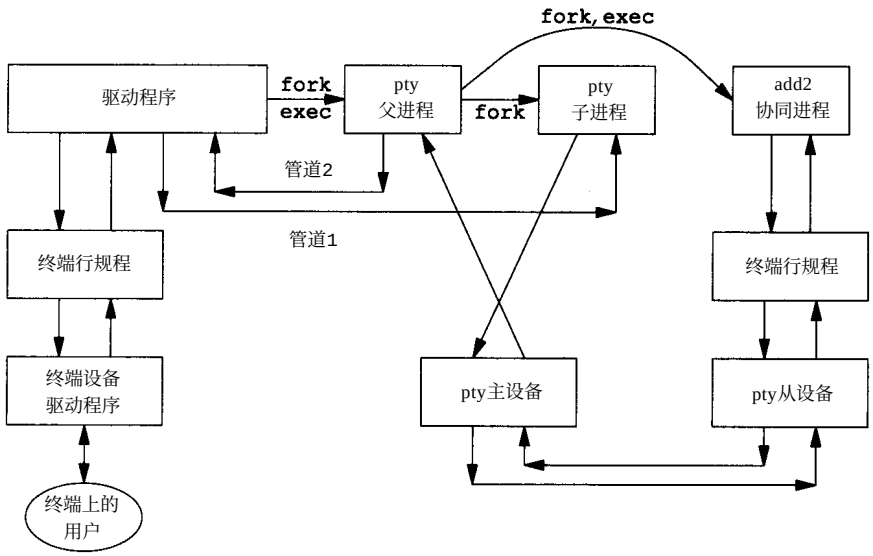

图19-9 运行一协同进程，以pty作为其输入和输出

这个例子显示了对于pty程序-e（不回显）选择项的重要性。pty不以交互方式运行，这是因为它的标准输入不是一个终端。在程序19-4中interactive标志默认为false，这是因为对isatty调用的返回结果是false。这意味着在真正的终端之上的行规程保持在典型模式下并允许回显。指定-e选择项后，关掉了伪终端从设备上的行规程模块的回显。如果不这样做，则键入的每一个字符都将被两个行规程模块显示两次。

我们还要用-e选择项关闭termios结构的ONLCR标志，防止所有的协同进程的输出被回车和换行符终止。

在不同的系统上测试这个例子会遇到12.8节描述readn和writen函数时提到的问题。当描述符不是引用普通的磁盘文件时，从read返回的读取数据量可能因实现不同而有所区别。协同进程使用pty时，如果调用通过管道的read而返回结果不到一行，将输出不可预测的结果。解决的方法不是使用程序14-9而是使用修改过的使用标准I/O库的习题14.5的程序，将两个管道都设置为行缓存。这样fgets函数将会读完一个整行。程序14-9的while循环假设送到协同进程的每一行都会带来一行的返回结果。

19.6.6 用非交互模式驱动交互式程序

虽然让pty运行所有的协同进程是非常诱人的想法，但如果协同进程是交互式的，就不能

正常工作。问题在于pty只是将其标准输入复制到pty，并将来自pty的复制到其标准输出。而并不关心具体得到什么数据。

举个例子，我们可以在pty直接与调制解调器对话之下运行18.7节的call客户机：

```
pty call t2500
```

这样做不比直接键入call t2500有什么优点，但我们可能希望用一个shell命令运行call程序来取得调制解调器的一些内部寄存器的内容。如果t2500.cmd包括两行：

```
aatn?
~.
```

第一行打印出调制解调器的寄存器值，第二行终止call程序。但是如果运行：

```
pty -1 < t2500.cmd call t2500
```

结果就不是我们希望得到的。事实上文件t2500.cmd的内容首先被送到了调制解调器。当交互运行call程序时我们等待调制解调器显示“ Connected ”，但是pty程序不知道这样做。这就是为什么需要比pty更巧妙的程序，如expect，自脚本文件运行交互式程序。

在pty上即使运行程序14-9也不能正常工作，这是因为程序14-9认为它在一个管道写入的每一行都会在另一个管道产生一行。而且， 14-9程序总是先发送一行到系统进程，然后再读取一行。在上面的例子中，我们需要先收到行“ Connected ”，然后再发送数据。

这里有一些使用shell命令驱动交互式程序的方法。可以在 pty上增加一种命令语言和一个解释器。但是一个适当的命令语言可能十倍于 pty程序的大小。另一种方法是使用命令语言并用pty_fork函数来调用交互式程序，这正是expect程序所做的。

我们将采用一种不同的方法，使用选择项-d让pty程序同一个管理输入和输出的驱动进程连接起来。该驱动进程的标准输出是pty的标准输入，反之亦然。这有点像协同进程，只是在 pty的“另一边”。此种进程结构与图19-9中所示的几乎相同，但是在这种情况下由pty来完成驱动进程的fork和exec。而且我们将在pty和驱动进程之间使用一个单独的流管道，而不是两个半双工管道。

程序19-6是do_driver函数的源码，该函数被pty（见程序19-4）的main函数在使用-d选项时调用。

程序19-6 pty程序的do_driver函数

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

void
do_driver(char *driver)
{
    pid_t   child;
    int     pipe[2];

        /* create a stream pipe to communicate with the driver */
    if (s_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ( (child = fork()) < 0)
        err_sys("fork error");

    else if (child == 0) {          /* child */
        close(pipe[1]);

                /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
```

```
            err_sys("dup2 error to stdin");

                /* stdout for driver */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        close(pipe[0]);

                /* leave stderr for driver alone */

        execlp(driver, driver, (char *) 0);
        err_sys("execlp error for: %s", driver);
    }

    close(pipe[0]);     /* parent */

    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");

    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    close(pipe[1]);

    /* Parent returns, but with stdin and stdout connected
       to the driver. */
}
```

通过编写自己的驱动程序的方法，可以随意地驱动交互式程序。即使驱动程序有和 pty连接在一起的标准输入和标准输出，它仍然可以通过 /dev/tty同用户交互。这个解决方法仍不如expect程序通用，但是它提供了一种不到50行代码的选择方案。

19.7 其他特性

伪终端还有其他特性，我们在这里简略提一下。 AT&T[1990d]和4.3+BSD系统的操作手册有更详细的内容。

19.7.1 打包模式

打包模式能够使伪终端主设备了解到伪终端从设备的状态变化。在 SVR4系统中可以将流模块pckt压入主设备端来设置这种模式。图 19-2显示了这种可选模式。在 4.3+BSD系统中可以通过TIOCPKT的ioctl来设置这种模式。

SVR4和4.3+BSD系统中具体的打包模式有所不同。在 SVR4系统中，读取伪终端主设备的进程必须调用 getmsg从流中取得数据，这是因为 pckt模块将一些事件转化为无数据的流消息。在4.3+BSD系统中每一次从伪终端主设备的读操作都会在可选数据之后返回状态字节。

无论实现的方法是什么样的，打包模式的目的是，当伪终端从设备之上的行规程模块出现以下事件时，通知进程从伪终端主设备读取数据：读入队列被刷新；写出队列被刷新；输出被停止（如：Ctrl-S）；输出重新开始；XON/XOFF流开关被关闭后重新打开；XON/XOFF流开关被打开后重新关闭。这些事件被rlogin客户机和rlogid服务器等使用。

19.7.2 远程模式

伪终端主设备可以用 TIOCREMOTE的ioctl将伪终端从设备设置成远程模式。虽然 SVR4和4.3+BSD系统使用同样的命令来打开或关闭这个特性，但是在 SVR4系统中ioctl的第三个参数是一个整型数，而 4.3+BSD中是一个指向整型数的指针。

当伪终端主设备将伪终端从设备设置成这种模式时，它通知伪终端从设备之上的行规程模块对从主设备收到的任何数据都不要进行处理，无论它是不是从设备的termios结构的规范或非规范标志。远程模式适用于窗口管理器这种进行自己的行编辑的应用模式。

19.7.3 窗口大小变化

伪终端主设备上的进程可以用TIOCSWINSZ的ioctl来设置从设备的窗口大小。如果新的大小和老的不同，一个SIGWINCH信号将被发送到伪终端从设备的前台进程组。

19.7.4 信号发生

读写伪终端主设备的进程可以向伪终端从设备的进程组发送信号。在SVR4系统中，可以通过TIOCSIGNAL的ioctl完成这个功能，第三个参数就是信号的数值。在4.3+BSD中通过TIOCSIG的ioctl来完成，第三个参数就是指向信号编号值的指针。

19.8 小结

本章首先说明了在SVR4和4.3+BSD系统中打开伪终端的代码。然后用此代码提供了用于多种不同应用的通用的pty_fork函数。这个函数是小程序（pty）的基础。并且讨论了许多伪终端的属性。

伪终端在大多数UNIX系统中每天都被用来进行网络登录。我们检查了伪终端的许多其他用途，从script程序到使用批处理脚本来驱动交互式程序。

习题

19.1 当用telnet或rlogin远程登录到一个BSD系统上时，像我们在19.3.2节讨论过的那样，伪终端从设备的所有权和许可权被设置。该过程是如何发生的？

19.2 修改4.3+BSD系统中的ptys_open函数，使之调用一个设置-用户-ID程序来改变伪终端从设备的所有权和许可权（像SVR4系统中的grantpt函数所做的）。

19.3 使用pty程序来决定你的系统初始化termios结构和winsize结构的值。

19.4 重写loop函数（见程序19-5），使之成为一个使用select或poll的单个进程。

19.5 在子进程中，pty_fork返回后，标准输入、标准输出和标准出错都以读写方式打开。能够将标准输入变成只读，另两个变成只写吗？

19.6 在图19-7中，指出哪个进程组是前台的，哪个进程组是后台的，并指出对话期管理者。

19.7 在图19-7中，当键入文件终止符时，进程终止的顺序是什么？如果可能的话，用进程计数来修改之。

19.8 script(1)程序通常开始时在输出文件头增加一行，结束时在输出文件末尾增加一行。将这个特性加到本章简单的shell脚本中。

19.9 解释为什么在下面的例子中，文件data的内容被输出到终端上，而程序ttyname只产生输出而从不读取输入。

```
$ cat data                      一个有两行的文件
hello,
world
$ pty -i < data ttyname         -i 忽略stdin的文件结束标志
hello,                          这两行来自何处？
```

```
world                                    我们期望ttyname输出这三行
fd 0:/dev/ttyp5
fd 1:/dev/ttyp5
fd 2:/dev/ttyp5
```

19.10　写一个调用pty_fork的程序，该程序有一个子进程，该子进程exec另一个要求你编写的程序。子进程调用的新的程序能够捕获SIGTERM和SIGWINCH。当捕获到消息时，该程序要打印出来，并且对于后一种消息，还要打印终端窗口大小。然后让父进程向19.7节描述过的，有ioctl的伪终端从设备的进程组发送SIGTERM消息。从伪终端从设备读回消息验证捕获的消息。接下来用父进程设置伪终端从设备窗口的大小，并读回伪终端从设备的输出。让父进程退出并确定是否要伪终端从设备进程也终止，如果要终止，应如何终止？