第18章　调制解调器拨号器

18.1　引言

与调制解调器相关的程序要处理如此种类繁多的调制解调器很困难。在大多数 UNIX系统中总有两个程序来处理调制解调器。第一个是远程登录程序，它允许我们拨通另外的计算机、登录和使用远程系统。在系统V中这个程序叫做cu，而BSD则称它为tip。它们完成类似的工作，而且都可以处理很多不同类型的调制解调器。另一个使用调制解调器的程序是 uucico，它是UUCP包的一部分。问题是不同种类调制解调器的具体特性一般都包含在这些程序的内部，所以，如果想写其他使用调制解调器的程序，就不得不做与这些程序类似的工作。同样，如果想要改变这些程序，使其不通过调制解调器，而利用其他介质通信（例如网络连接），那么也要做很大的改动。

本章开发了一个程序来处理调制解调器所有需要处理的细节。我们把所有这些的细节都集中到这个程序中，而不是分散在多个程序里（这个程序的构思来自于 Presotto 和Ritchie [1990]所描述的连接服务器）。为了使用这个程序，必须能如 15.3节所说明的那样调用它，并使它传回文件描述符。然后，用这个程序来开发远程登录程序（类似 cu和tip ）。

18.2　历史

cu(1) 命令（意思是call UNIX）是在V7中出现的。但它只能处理一种特殊的自动拨号单元（ACU）。伯克利的Bill Shannon修改了cu，并把它实现在4.2BSD的tip(1)中。这之间的最大改变是使用了一个文本文件/etc/remote来存放所有的系统信息（电话号码、优先的拨号器、波特率、奇偶校验、流控制等）。这个版本的tip支持六种不同的拨号单元和调制解调器，如要支持其他种类的调制解调器则要修改源码。

与cu和tip一样，UUCP系统也可以使用调制解调器和自动拨号单元。 UUCP对不同的调制解调器进行加锁，因此多个 UUCP的实例可以同时运行。这样， tip和cu程序就不得不遵循UUCP协议，避免与UUCP冲突。在BSD系统中，UUCP使用了它自己的拨号函数。这些函数被连接到UUCP的可执行程序中，这样增加新的调制解调器也需要修改源码。

SVR2提供了一个dial(3)函数来将调制解调器拨号的一致特性归纳到一个库函数中。这个函数由cu使用，但UUCP不使用。这是一个标准的C库函数，所以可以被一般程序使用。

Honey DanBer UUCP系统是将调制解调器命令从 C源程序中抽取出来，将它们放在一个Dialers文件中。这就允许不修改源码就可以加入新类型的调制解调器。但是 cu和UUCP所使用的访问Dialers文件的函数不是很通用。这说明cu和UUCP可以不重新开发代码去处理Dialers文件中的拨号信息，但除了cu和UUCP以外的程序并不能使用这个文件。

在所有这些版本的cu、tip和UUCP中，加锁保证了在同一时间只有一个程序使用某一设备。因为这些程序工作在不同系统中，早期的版本不提供记录锁，而使用一个早期形式的文件加锁，这会导致当一个程序崩溃时，该锁文件仍旧保留，所以又开发特殊的技术来处理这种情况。（对特殊设备文件不能使用记录锁，所以记录锁也不能完全解决问题）。

18.3 程序设计

我们来分析一下调制解调器拨号器所应该具有的特性：

(1) 它必须在不改动源码的情况下支持新增加的调制解调器类型。

为了达到这个目标，我们使用了Honey DanBer的Dialers文件。我们将所有使用这个文件来拨号调制解调器的代码都放到一个精灵进程服务器中，这样任何程序都可以使用 15.5节中的客户机-服务器函数来访问它。

(2) 一定要使用一些特定形式的锁，以保证当那些持有锁的程序在非正常结束时能自动释放它的锁。以前那些专门的技术，如那些在大多数 cu和UUCP版本中仍然使用的技术，都不应再使用。

我们用一个服务器精灵进程来处理所有的设备加锁。因为 15.5节中的客户机-服务器函数会在客户机终止时自动通知服务器，所以这个精灵进程能释放进程所持有的任何加锁。

(3) 新的程序一定要能够使用我们所开发的所有特性。开发一个新的处理调制解调器的程序不应当什么都要自己实现，它拨任何类型的调制解调器应该就像函数调用一样简单方便。为此，我们让中央服务器精灵进程处理所有与拨号有关的操作，并返回一个文件描述符。

(4) 客户机程序，例如cu和tip，不应当需要特别权限。这些程序不应当是设置 -用户-ID程序。但是要给予服务器精灵进程特殊权限，允许客户机程序运行时无需特权。

显然我们不能改动已有的cu、tip和UUCP程序，但应该让其他程序在我们工作的基础上实现起来更加简单。当然，我们也一定要充分吸取已有的 UNIX 拨号程序的优点。

图18-1描述了客户机-服务器工作模式的结构。

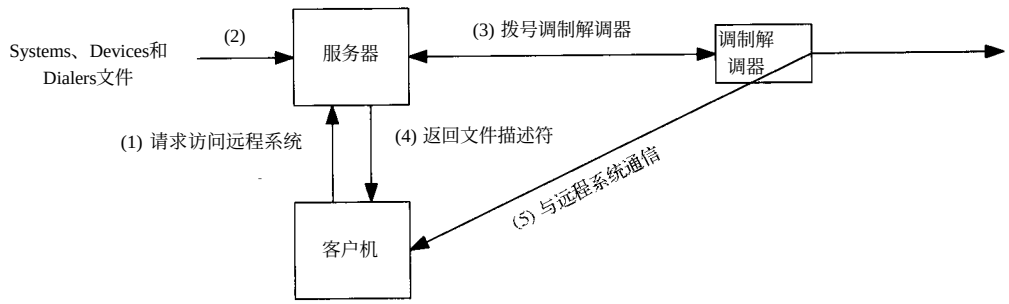

图18-1 客户机-服务器工作模式示意图

建立与远程系统的通信过程如下：

(1) 起动服务器。

(2) 客户机起动，使用cli_conn函数（见15.5节）建立与服务器的连接。客户机向服务器发出一个请求，请求拨号远程系统。

(3) 服务器读取Systems、Devices和Dialers配置文件来决定如何拨号远程系统（下一节将讲述这些文件）。如果正使用一个调制解调器，在对应的 Dialers配置文件中就包含了这个特定调制解调器的所有命令。

(4) 服务器打开该调制解调器设备并拨号该调制解调器。这需要一些时间（一般为15~30秒）。服务器处理所有对该调制解调器的加锁，以避免各用户间的冲突。

(5) 如果拨号成功，服务器返回一个该调制解调器设备的文件描述符给客户机。在 15.3节中的函数可以发送和接受这个描述符。

(6) 客户机直接与远程系统通信。服务器不再参与这个过程。客户机读写上一步返回的文件描述符即可。

客户机与服务器的通信过程第(2)~(5)步是通过一个流管道进行的。当客户机完成与远程系统的通信时，客户机关闭该流管道。服务器发现该管道关闭后，释放对调制解调器设备的加锁。

18.4 数据文件

这一节将讲述Honey DanBer UUCP系统所使用的三个文件：Systems、 Devices和Dialers。在这些文件中有很多 UUCP所使用的域。本节不详细讲述这些域和 UUCP系统本身。参见Redman [1989]可得到更详细的信息。

表18-1分栏列出了Systems 文件中的六个域。

表18-1 Systems 文件

name	time	type	class	phone	login
host1	任意	ACU	19200	5551234	不使用
host1	任意	ACU	9600	5552345	不使用
host1	任意	ACU	2400	5556789	不使用
modem	任意	modem	19200	—	不使用
laser	任意	laser	19200	—	不使用

*name*域是远程系统的名字。例如，可以使用 cu host1这种形式的命令。这里要注意的是，我们可以对同一远程系统建立多个项。系统按顺序尝试拨号这些项。表 18-1中名为modem和laser的项对应于与调制解调器和激光打印机的直接连接。并不需要拨号来连接这些设备，但对它们仍需要打开合适的终端连线，并处理好加锁问题。

*Time*域指定了拨号的星期和时间。这是一个 UUCP的域。*Type*域指定了对这个特定的*name*使用Devices文件中的哪一项。*Class*域是指线路速率（波特率）。*Phone*域指定那些type为ACU项的电话号码，而其他项的*phone*域是一个连字符。最后一个域*login*，是一个字符串。它是在UUCP中远程登录时所使用的，这里不使用这个域。

Devices文件包含了调制解调器和那些直接连接的主机的信息。表18-2列出了这个文件中的五个域。*type*域与Systems文件中*type*域对应。*class*域也一定要与Systems文件中对应的class域一致，它通常指定了线路速率。

表18-2 Devices 文件

type	line	line	class	dialer
ACU	cua0	—	19200	tbfast
ACU	cua0	—	9600	tb9600
ACU	cua0	—	2400	tb2400
ACU	cua0	—	1200	tb1200
modem	ttya	—	19200	direct
laser	ttyb	—	19200	direct

设备的实际名称是对*line*字段加前缀/dev/。在这个例子中，实际设备是/dev/cua0，/dev/ttya和/dev/ttyb。另外一个域*line2*，没有被使用。

最后一个域*dialer*，与Dialers文件中对应项一致。对于直接相连的项则为direct。

表18-3显示了Dialers文件的格式。这个文件包含了所有调制解调器的拨号命令。

表18-3 Dialers文件

dialer	*sub*	*handshake*
tb9600	=W-,	"" \dA\pA\pA\pTQ0S2=255S12=255s50=6s58=2s68=255\r\c OK\r \EATDT\T\r\c CONNECT\s9600 \r\c ""
tbfast	=W-,	"" \dA\pA\pA\pTQ0S2=255S12=255s50=255s58=2s68=255s110=1s111=30\r\c OK\r \EATDT\T\r\c CONNECT\sFAST

表中只列出两项，没有列出 Devices中的tb1200和tb2400项。*handshake*域本应该写在同一行中，因为版面的限制，我们把它放在两行上。

*dialer*域与Devices文件中的行相对应。*sub*域则指定了电话号码中等号和减号的替代字符。在表18-3中，这个域表明了用w代替等号，逗号代替减号。这样就允许 Systems文件中的电话号码中含有等号（意思是等待拨号音）和减号（意思是暂停）。在不同的调制解调器上，这两个字符的含义不同，将它们替换成何种字符，需在 Dialers文件中指定。

最后一个域*handshake*，包含了实际的拨号指令。它是一连串以空格分开的字符串，称为期望-发送串。我们期望（即一直读取，直到得到匹配字符串）得到第一个字符串，然后发送（写入）第二个字符串。作为一个例子，让我们来查看 tbfast项。这个项是用于PEP (Packetized Ensemble Protocol) 模式的Telebit Trailblazer 调制解调器。

(1) 第一个期望字符串是空，意思是“ 期望空 ”。这总是成功的。

(2) 发送第二个字符串，这个字符串以\d开头，\d表示暂停两秒。然后发送A。再暂停半秒（\p），发送另外一个A，暂停，再发送一个A，再暂停。接着，发送余下的以 T开头的字符串。这些都是设置调制解调器的命令。\r发送一个回车，\c表明在发送字符串结尾不要开始新行。

(3) 从调制解调器读取，直到得到字符串OK\r（\r表示回车）。

(4) 下一个发送串以\E开头，这允许进行回应检查：每次发送给调制解调器一个字符，就一直读取直到有回应。然后发送四个字符 ATDT。下一个特殊字符\T，是指使用替代的电话号码。然后是一个回车符，然后是\c是指在发送字符串后不要开始新行。

(5) 最后的期望字符串是等待调制解调器返回 CONNECT FAST。（\s意思是单个空格。）

当收到最后的期望字符串后，拨号就完成了。（当然，在*handshake*字符串中可能出现其他更多的特殊字符序列，这里就不详细说明了。）

现在来总结一下，对这三个文件的操作。

(1) 使用远程系统的名称，在Systems文件中找到相同*name*的第一项。

(2) 在Devices文件中找到对应的项，其*type*域和*class*域与Systems文件中项的相应域匹配。

(3) 在Dilaler文件中找到与Devices文件*dialer*域对应的项。

(4) 拨号。

这个过程如果失败，有两个原因：（1）对应于Devices文件中*line*域的设备已经被其他人所使用，（2）拨号不成功。（例如，远程系统电话占线，或者远程系统关机不响应电话等）。第二种情况一般可以通过对调制解调器读写超时来确定。（见习题18.10）。不管出现哪一种情况，都要回到拨号的第(1)步，然后选择Systems文件中同一远程系统的下一项。如同我们在表 18-1中看到的，一个特定的主机可以有多个项，每个主机可以有多个电话号码（同一个电话号码也可以对应多个设备）。

在Honey DanBer 系统中还有其他我们没有用到的文件。如Dialcodes指定了Systems文件中电话号码的缩写，而Sysfiles文件允许指定Systems、Devices、Dialers文件的替代文件。

18.5 服务器设计

现在我们开始描述一下服务器。有两个因素影响服务器的设计。

(1) 拨号过程可能会延续一段时间（15~30秒），所以服务器一定要创建一个子进程来处理实际的拨号。

(2) 服务器的精灵进程（父进程）一定要管理所有的加锁。

图18-2显示了这个过程。

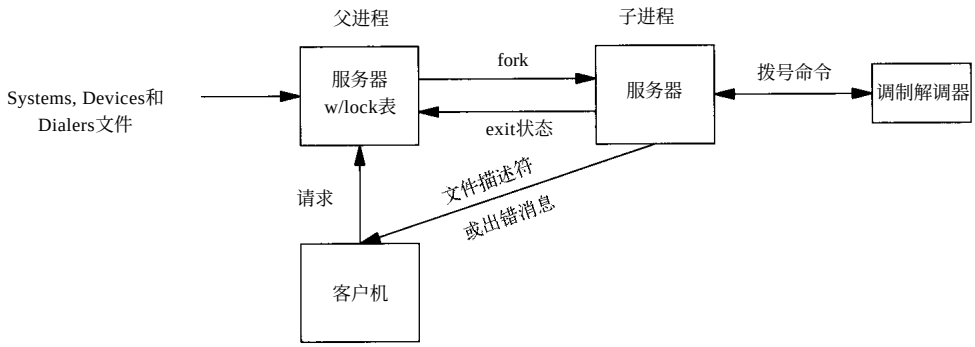

图18-2 调制解调器拨号器的工作过程

服务器的工作过程如下：

(1) 父进程在服务器的众所周知名字处接收从客户机发来的请求。如 15.5节所述，这在客户机-服务器之间生成了一个流管道。父进程就像15.6节中的open服务器一样，要同时处理多个客户机。

(2) 基于客户机要联系的远程系统的名字，父进程查询Systems文件和Devices文件找到匹配的项。父进程同时也维护一个加锁表，记录哪些设备在被使用，这样它就不查询那些被使用的项了。

(3) 如果发现匹配项，则frok一个出子进程来进行实际的拨号。（父进程这时可以处理其他客户机请求）。如果成功，子进程就在客户机指定的流管道上将调制解调器的文件描述符传给客户机（这个管道在fork时也被复制了），并调用exit(0)。如果发生了错误（例如，电话线占线、没有响应等），子进程调用exit(1)。

(4) 子进程结束时，会发送信号 SIGCHLD通知父进程。父进程就得到子进程的结束状态（waitpid）。

如果子进程成功，父进程就不用再做其他事情。在客户机结束使用调制解调器之前，必须一直对调制解调器加锁。客户机指定的客户机 -父进程之间的流管道就一直打开着。这样，当客户机终止时，父进程得到通知，然后释放对设备的加锁。

如果子进程不成功，父进程就从 Systems文件中尝试找下一个匹配项。如果找到了对远程系统的另一项，父进程返回上一步，创建一个新的子进程来拨号。如果没有找到新的匹配项，父进程调用send_err（见程序15-4）后关闭与客户机的流管道。

与每一个客户机有一个连接使子进程在必要时能将调试输出发回给客户机。发生问题时，客户机常常想要看到整个实际拨号过程。

18.6 服务器源码

服务器包括17个源文件。表18-4详细说明了父进程和子进程所使用的文件，以及这些文件

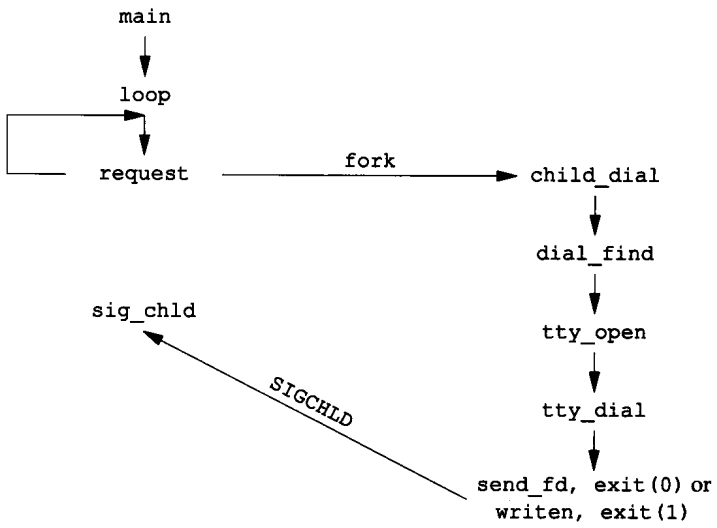

图18-3 服务器的函数调用过程

中所包含的函数。图18-3描述了不同函数的调用过程。

表18-4 服务器源文件

源文件	父进程/	子进程	函 数
childdial.c		C	child_dial
cliargs.c	P		cli_args
client.c	P		client_alloc, client_add, client_del, client_sigchld
ctlstr.c		C	ctl_str
debug.c		C	DEBUG, DEBUG_NONL
devfile.c	P		dev_next, dev_rew, dev_find
dialfile.c		C	dial_next, dial_rew, dial_find
expectstr.c		C	expect_str, exp_read, sig_alrm
lock.c	P		find_line, lock_set, lock_rel, is_locked
loop.c	P		loop, cli_done, child_done
main.c	P		main
request.c	P		request
sendstr.c		C	send_str
sigchld.c	P		sig_chld
sysfile.c	P		sys_next, sys_rew, sys_posn
ttydial.c		C	tty_dial
ttyopen.c		C	tty_open

程序18-1是calld.h 头文件，它被包含在所有这些源文件中。 calld.h 包含几个系统头文件，定义了一些基本的常量，声明了全局变量。

程序18-1 calld.h头文件

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

#define CS_CALL "/home/stevens/calld"    /* well-known name */
#define CL_CALL "call"
#define MAXSYSNAME  256
#define MAXSPEEDSTR 256
```

```
#define NALLOC  10              /* #structs to alloc/realloc for */
            /* Client structs (client.c), Lock structs (lock.c) */
#define WHITE       " \t\n"             /* for separating tokens */
#define SYSTEMS     "./Systems"         /* my own copies for now */
#define DEVICES     "./Devices"
#define DIALERS     "./Dialers"

            /* declare global variables */
extern int       clifd;
extern int       debug;     /* nonzero if interactive (not daemon) */
extern int       Debug;     /* nonzero for dialing debug output */
extern char      errmsg[];  /* error message string to return to client */
extern char     *speed;     /* speed (actually "class") to use */
extern char     *sysname;   /* name of system to call */
extern uid_t     uid;       /* client's uid */
extern volatile sig_atomic_t        chld_flag;  /* when SIGCHLD occurs */
extern enum parity { NONE, EVEN, ODD } parity;  /* specified by client */

typedef struct {        /* one Client struct per connected client */
  int   fd;          /* fd, or -1 if available */
  pid_t pid;         /* child pid while dialing */
  uid_t uid;         /* client's user ID */
  int   childdone;   /* nonzero when SIGCHLD from dialing child recvd:
                        1 means exit(0), 2 means exit(1) */
  long  sysftell;    /* next line to read in Systems file */
  long  foundone;    /* true if we find a matching sysfile entry */
  int   Debug;               /* option from client */
  enum parity parity;        /* option from client */
  char  speed[MAXSPEEDSTR]; /* option from client */
  char  sysname[MAXSYSNAME];/* option from client */
} Client;

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

typedef struct {        /* everything for one entry in Systems file */
  char  *name;       /* system name */
  char  *time;       /* (e.g., "Any") time to call (ignored) */
  char  *type;       /* (e.g., "ACU") or system name if direct connect */
  char  *class;      /* (e.g., "9600") speed */
  char  *phone;      /* phone number or "-" if direct connect */
  char  *login;      /* uucp login chat (ignored) */
} Systems;

typedef struct {        /* everything for one entry in Devices file */
  char  *type;       /* (e.g., "ACU") matched by type in Systems */
  char  *line;       /* (e.g., "cua0") without preceding "/dev/" */
  char  *line2;      /* (ignored) */
  char  *class;      /* matched by class in Systems */
  char  *dialer;     /* name of dialer in Dialers */
} Devices;

typedef struct {        /* everything for one entry in Dialers file */
  char  *dialer;     /* matched by dialer in Devices */
  char  *sub;        /* phone number substitution string (ignored) */
  char  *expsend;    /* expect/send chat */
} Dialers;

extern Systems  systems;    /* filled in by sys_next() */
extern Devices  devices;    /* filled in by dev_next() */
extern Dialers  dialers;    /* filled in by dial_next() */

               /* our function prototypes */
void     child_dial(Client *);                  /* childdial.c */
```

```
int     cli_args(int, char **);                 /* cliargs.c */

int     client_add(int, uid_t);                 /* client.c */
void    client_del(int);
void    client_sigchld(pid_t, int);

void    loop(void);                             /* loop.c */

char   *ctl_str(char);                          /* ctlstr.c */

int     dev_find(Devices *, const Systems *);   /* devfile.c */
int     dev_next(Devices *);
void    dev_rew(void);

int     dial_find(Dialers *, const Devices *); /* dialfile.c */
int     dial_next(Dialers *);
void    dial_rew(void);

int     expect_str(int, char *);                /* expectstr.c */

int     request(Client *);                      /* request.c */

int     send_str(int, char *, char *, int);     /* sendstr.c */

void    sig_chld(int);                          /* sigchld.c */

long    sys_next(Systems *);                    /* sysfile.c */
void    sys_posn(long);
void    sys_rew(void);

int     tty_open(char *, char *, enum parity, int);    /* ttyopen.c */

int     tty_dial(int, char *, char *, char *, char *); /* ttydial.c */

pid_t   is_locked(char *);                      /* lock.c */
void    lock_set(char *, pid_t);
void    lock_rel(pid_t);

void    DEBUG(char *, ...);                     /* debug.c */
void    DEBUG_NONL(char *, ...);
```

我们定义了一个Client结构，它包含了每一客户机的所有信息。这是一个对程序 15-26中类似结构的扩展。在创建一个子进程为客户机拨号和子进程终止之间，可以处理任一多的其他客户机。这个结构同时包含了我们所需要的其他信息，如尝试找到 Systems文件中的其他项，重新拨号等。

我们同样为Systems、Devices、Dialers文件中每一项定义了一个结构。

程序18-2是服务器的main函数。因为这个程序一般是作为精灵进程运行，故提供了一个 -d 的命令行选择项，允许交互式运行。

程序18-2　main函数

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

            /* define global variables */
int         clifd;
int         debug;  /* daemon's command line flag */
int         Debug;  /* Debug controlled by client, not cmd line */
char        errmsg[MAXLINE];
char        *speed;
char        *sysname;
uid_t       uid;
Client      *client = NULL;
```

```
int         client_size;
Systems     systems;
Devices     devices;
Dialers     dialers;
volatile sig_atomic_t chld_flag;
enum parity parity = NONE;

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

    log_open("calld", 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 '?':
            log_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

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

当使用-d选择项后，所有对log_XXX函数的调用(见附录B)都送到标准错误。否则它们就会被syslog记录下来。

函数loop是服务器程序的主循环（见程序18-3）。它使用了select函数来复制不同的描述符。

程序18-3　loop.c文件

```
#include    "calld.h"
#include    <sys/time.h>
#include    <errno.h>

static void cli_done(int);
static void child_done(int);

static fd_set   allset; /* one bit per client conn, plus one for listenfd */
                        /* modified by loop() and cli_done() */

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

    if (signal_intr(SIGCHLD, sig_chld) == SIG_ERR)
        log_sys("signal error");

                /* obtain descriptor to listen for client requests on */
    if ( (listenfd = serv_listen(CS_CALL)) < 0)
        log_sys("serv_listen error");
```

```
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
maxfd = listenfd;
maxi = -1;

for ( ; ; ) {
    if (chld_flag)
        child_done(maxi);
    rset = allset;      /* rset gets modified each time around */
    if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0) {
        if (errno == EINTR) {
                /* caught SIGCHLD, find entry with childdone set */
            child_done(maxi);
            continue;       /* issue the select again */
        } else
            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;
    }

        /* Go through client[] array.
           Read any client data that has arrived. */

    for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
        if ( (clifd = cliptr->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) {
                /* The client has terminated or closed the stream
                   pipe.  Now we can release its device lock. */

                log_msg("closed: uid %d, fd %d",
                                    cliptr->uid, clifd);
                lock_rel(cliptr->pid);
                cli_done(clifd);
                continue;
            }
            /* Data has arrived from the client.  Process the
               client's request. */

            if (buf[nread-1] != 0) {
                log_quit("request from uid %d not null terminated:"
                         " %*.*s", uid, nread, nread, buf);
                cli_done(clifd);
                continue;
            }
            log_msg("starting: %s, from uid %d", buf, uid);
```

```
                        /* Parse the arguments, set options.  Since
                           we may need to try calling again for this
                           client, save options in client[] array. */
                    if (buf_args(buf, cli_args) < 0)
                        log_quit("command line error: %s", buf);
                    cliptr->Debug = Debug;
                    cliptr->parity = parity;
                    strcpy(cliptr->sysname, sysname);
                    strcpy(cliptr->speed, (speed == NULL) ? "" : speed);
                    cliptr->childdone = 0;
                    cliptr->sysftell = 0;
                    cliptr->foundone = 0;

                    if (request(cliptr) < 0) {
                            /* system not found, or unable to connect */
                        if (send_err(cliptr->fd, -1, errmsg) < 0)
                            log_sys("send_err error");
                        cli_done(clifd);
                        continue;
                    }
                    /* At this point request() has forked a child that is
                       trying to dial the remote system.  We'll find
                       out the child's status when it terminates. */
                }
            }
        }
    }

/* Go through the client[] array looking for clients whose dialing
   children have terminated.  This function is called by loop() when
   chld_flag (the flag set by the SIGCHLD handler) is nonzero. */

static void
child_done(int maxi)
{
    Client  *cliptr;

again:
    chld_flag = 0;  /* to check when done with loop for more SIGCHLDs */

for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
    if ( (clifd = cliptr->fd) < 0)
        continue;
    if (cliptr->childdone) {
        log_msg("child done: pid %d, status %d",
                        cliptr->pid, cliptr->childdone-1);

        /* If the child was successful (exit(0)), just clear
           the flag.  When the client terminates, we'll read
           the EOF on the stream pipe above and release
           the device lock. */

        if (cliptr->childdone == 1) {    /* child did exit(0) */
            cliptr->childdone = 0;
            continue;
        }

        /* Unsuccessful: child did exit(1).  Release the device
           lock and try again from where we left off. */

        cliptr->childdone = 0;
        lock_rel(cliptr->pid);  /* unlock the device entry */
        if (request(cliptr) < 0) {
                    /* still unable, time to give up */
            if (send_err(cliptr->fd, -1, errmsg) < 0)
```

```
                log_sys("send_err error");
            cli_done(clifd);
            continue;
        }
        /* request() has forked another child for this client */
    }
}
if (chld_flag)  /* additional SIGCHLDs have been caught */
    goto again; /* need to check all childdone flags again */
}

/* Clean up when we're done with a client. */

static void
cli_done(int clifd)
{
    client_del(clifd);      /* delete entry in client[] array */
    FD_CLR(clifd, &allset); /* turn off bit in select() set */
    close(clifd);           /* close our end of stream pipe */
}
```

loop函数初始化client数组，建立一个对SIGCHLD信号的处理器。我们不调用signal，而调用了signal_intr，这样当有信号返回时，每一个慢速的系统调用都可以被中断。然后 loop函数调用serv_listen（见程序15-19和程序15-22）。loop函数的其他部分是一个基于select函数的无限循环，它检查如下两种情况：

(1) 如果传来一个新的客户机连接请求，则调用 serv_accept（见程序15-20和15-24）。函数client_add为这个新的客户机在client数组中增加一项。

(2) 浏览整个client数组，看是否有客户机终止或者新的客户机请求到达。

当一个客户机终止拨号时（不管是否自愿），它的客户机专用的通向服务器的流管道被关闭，我们从管道上得到一个文件终止符。这时，可以释放这个客户机所拥有的全部加锁，并删除client数组中的项。

当从客户机收到请求时，则调用 request函数（函数buf_args见程序15-17）。如果客户机调用的远程系统的名字有效，而且找到相对应的 Devices项，request函数就会创建一个子进程，然后返回。

在loop函数运行中，子进程终止这一外部事件随时都可能发生。如果在 select函数中阻塞，则select函数返回一个EINTR错误。因为在loop函数的其他地方也可能出现子进程终止信号，所以在这个循环中每次调用 select之前都检测标志chld_flag，如果有，则调用child_done来处理子进程终止。

loop函数浏览整个client数组，检查每一项的childdone标志。如果子进程成功，就不需要做其他事情。但如果子进程以状态1终止时（exit(1)），则调用request函数来尝试使用Systems文件中的下一项。

程序18-4 是cli_args，这个函数在客户机请求到达时被loop函数中的buf_args所调用。它处理客户机送来的命令行参数。注意这个函数根据命令行参数设置全局变量，然后 loop函数将这些全局变量复制到client数组的相应的项中，这些选择项只影响一个客户机请求。

程序18-5是client.c，它定义了处理client数组的函数组。程序18-5和程序15-27的区别在于在这里一定要根据进程的ID（见函数client_sigchld）来查询所需要的项。

程序18-6是文件lock.c。其中的函数管理父进程中的 lock数组。同上面的client数组一样，调用realloc来给lock数组动态分配空间，以避免编译时间限制。

程序18-4 cli_args函数

```
#include    "calld.h"

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

int
cli_args(int argc, char **argv)
{
    int     c;

    if (argc < 2 || strcmp(argv[0], CL_CALL) != 0) {
        strcpy(errmsg, "usage: call <options> <hostname>");
        return(-1);
    }
    Debug = 0;      /* option defaults */
    parity = NONE;
    speed = NULL;
    opterr = 0;     /* don't want getopt() writing to stderr */
    optind = 1;     /* since we call getopt() multiple times */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
        case 'd':
            Debug = 1;  /* client wants DEBUG() output */
            break;

        case 'e':       /* even parity */
            parity = EVEN;
            break;

        case 'o':       /* odd parity */
            parity = ODD;
            break;

        case 's':       /* speed */
            speed = optarg;
            break;

        case '?':
            sprintf(errmsg, "unrecognized option: -%c\n", optopt);
            return(-1);
        }
    }
    if (optind < argc)
        sysname = argv[optind];     /* name of host to call */
    else {
        sprintf(errmsg, "missing <hostname> to call\n");
        return(-1);
    }
    return(0);
}
```

程序18-5 client.c程序

```
#include    "calld.h"

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);
}

/* Find the client entry corresponding to a process ID.
 * This function is called by the sig_chld() signal
 * handler only after a child has terminated. */

void
client_sigchld(pid_t pid, int stat)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].pid == pid) {
            client[i].childdone = stat; /* child's exit() status +1 */
            return;
        }
    }
    log_quit("can't find client entry for pid %d", pid);
}
```

程序18-6 管理设备锁的函数组

```
#include    "calld.h"

typedef struct {
  char  *line;  /* points to malloc()ed area */
                /* we lock by line (device name) */
  pid_t pid;    /* but unlock by process ID */
                /* pid of 0 means available */
} Lock;
static Lock *lock = NULL;   /* the malloc'ed/realloc'ed array */
static int   lock_size;     /* #entries in lock[] */
static int   nlocks;        /* #entries currently used in lock[] */

/* Find the entry in lock[] for the specified device (line).
 * If we don't find it, create a new entry at the end of the
 * lock[] array for the new device.  This is how all the possible
 * devices get added to the lock[] array over time. */

static Lock *
find_line(char *line)
{
    int     i;
    Lock    *lptr;

    for (i = 0; i < nlocks; i++) {
        if (strcmp(line, lock[i].line) == 0)
            return(&lock[i]);   /* found entry for device */
    }

    /* Entry not found.  This device has never been locked before.
       Add a new entry to lock[] array. */

    if (nlocks >= lock_size) {  /* lock[] array is full */
        if (lock == NULL)       /* first time through */
            lock = malloc(NALLOC * sizeof(Lock));
        else
            lock = realloc(lock, (lock_size + NALLOC) * sizeof(Lock));
        if (lock == NULL)
            err_sys("can't alloc for lock array");

        lock_size += NALLOC;
    }

    lptr = &lock[nlocks++];
    if ( (lptr->line = malloc(strlen(line) + 1)) == NULL)
        log_sys("malloc error");
    strcpy(lptr->line, line);   /* copy caller's line name */
    lptr->pid  = 0;
    return(lptr);
}

void
lock_set(char *line, pid_t pid)
{
    Lock    *lptr;

    log_msg("locking %s for pid %d", line, pid);
    lptr = find_line(line);
    lptr->pid  = pid;
}

void
lock_rel(pid_t pid)
{
    Lock    *lptr;
```

```
    for (lptr = &lock[0]; lptr < &lock[nlocks]; lptr++) {
        if (lptr->pid == pid) {
            log_msg("unlocking %s for pid %d", lptr->line, pid);
            lptr->pid  = 0;
            return;
        }
    }
    log_msg("can't find lock for pid = %d", pid);
}

pid_t
is_locked(char *line)
{
    return( find_line(line)->pid ); /* nonzero pid means locked */
}
```

lock数组中的每一项都与一个*line*（Devices文件的第二个域）相关联。因为这些加锁函数不知道该数据文件中所有的*line*值，所以每当一个新*line*被第一次加锁时，就在lock数组中增加一项。函数find_line处理加锁。

下面的三个源文件处理三个数据文件： Systems, Devices和Dialers。每个文件有个XXX_next函数，它读取文件中的下一行，调用 ANSI C函数strtok把这一行分成多个域。程序18-7处理Systems文件。

程序18-7 读取Systems文件的函数

```
#include    "calld.h"

static FILE *fpsys = NULL;
static int   syslineno;          /* for error messages */
static char  sysline[MAXLINE];
        /* can't be automatic; sys_next() returns pointers into here */

/* Read and break apart a line in the Systems file. */

long                        /* return >0 if OK, -1 on EOF */
sys_next(Systems *sysptr)   /* structure is filled in with pointers */
{
    if (fpsys == NULL) {
        if ( (fpsys = fopen(SYSTEMS, "r")) == NULL)
            log_sys("can't open %s", SYSTEMS);
        syslineno = 0;
    }

again:
    if (fgets(sysline, MAXLINE, fpsys) == NULL)
        return(-1);     /* EOF */
    syslineno++;

    if ( (sysptr->name = strtok(sysline, WHITE)) == NULL) {
        if (sysline[0] == '\n')
            goto again;     /* ignore empty line */
        log_quit("missing 'name' in Systems file, line %d", syslineno);
    }
    if (sysptr->name[0] == '#')
        goto again;         /* ignore comment line */

    if ( (sysptr->time = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'time' in Systems file, line %d", syslineno);

    if ( (sysptr->type = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'type' in Systems file, line %d", syslineno);
```

```
    if ( (sysptr->class = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'class' in Systems file, line %d", syslineno);

    if ( (sysptr->phone = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'phone' in Systems file, line %d", syslineno);

    if ( (sysptr->login = strtok(NULL, "\n")) == NULL)
        log_quit("missing 'login' in Systems file, line %d", syslineno);
        return(ftell(fpsys));   /* return the position in Systems file */
    }

    void
    sys_rew(void)
    {
        if (fpsys != NULL)
            rewind(fpsys);
        syslineno = 0;
    }

    void
    sys_posn(long posn)      /* position Systems file */
    {
        if (posn == 0)
            sys_rew();
        else if (fseek(fpsys, posn, SEEK_SET) != 0)
            log_sys("fseek error");
    }
```

函数sys_next由request调用，其功能是读取Systems文件的下一项。

对于每一个客户机，必须记住当前位置（Client结构的sysftell成员变量）。这样如果一个子进程拨号远程系统没有成功，则可以知道是 Systems文件中的哪一项失败，然后尝试其他项。这个位置可通过调用标准的I/O函数ftell得到，可以用fseek函数复位。

程序18-8包含了读取Devices文件的函数。

程序18-8　读取Devices文件的函数

```
    #include    "calld.h"

    static FILE *fpdev = NULL;
    static int  devlineno;          /* for error messages */
    static char devline[MAXLINE];
            /* can't be automatic; dev_next() returns pointers into here */

    /* Read and break apart a line in the Devices file. */

    int
    dev_next(Devices *devptr)       /* pointers in structure are filled in */
    {
        if (fpdev == NULL) {
            if ( (fpdev = fopen(DEVICES, "r")) == NULL)
                log_sys("can't open %s", DEVICES);
            devlineno = 0;
        }

    again:
        if (fgets(devline, MAXLINE, fpdev) == NULL)
            return(-1);     /* EOF */
        devlineno++;

    if ( (devptr->type = strtok(devline, WHITE)) == NULL) {
        if (devline[0] == '\n')
            goto again;     /* ignore empty line */
        log_quit("missing 'type' in Devices file, line %d", devlineno);
```

```
    }
    if (devptr->type[0] == '#')
        goto again;          /* ignore comment line */

    if ( (devptr->line = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line' in Devices file, line %d", devlineno);

    if ( (devptr->line2 = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line2' in Devices file, line %d", devlineno);

    if ( (devptr->class = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'class' in Devices file, line %d", devlineno);

    if ( (devptr->dialer = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'dialer' in Devices file, line %d", devlineno);

    return(0);
}

void
dev_rew(void)
{
    if (fpdev != NULL)
        rewind(fpdev);
    devlineno = 0;
}

/* Find a match of type and class */

int
dev_find(Devices *devptr, const Systems *sysptr)
{
    dev_rew();
    while (dev_next(devptr) >= 0) {
        if (strcmp(sysptr->type, devptr->type) == 0 &&
            strcmp(sysptr->class, devptr->class) == 0)
                return(0);      /* found a device match */
    }
    sprintf(errmsg, "device '%s'/'%s' not found\n",
                            sysptr->type, sysptr->class);
    return(-1);
}
```

可以看到，request函数调用dev_find函数来确定type域、class域与Systems文件中对应域相同的项。

程序18-9是读取Dialers文件的函数。

程序18-9　读取Dialers文件的函数

```
#include    "calld.h"

static FILE *fpdial = NULL;
static int  diallineno;         /* for error messages */
static char dialline[MAXLINE];
        /* can't be automatic; dial_next() returns pointers into here */

/* Read and break apart a line in the Dialers file. */

int
dial_next(Dialers *dialptr) /* pointers in structure are filled in */
{
    if (fpdial == NULL) {
        if ( (fpdial = fopen(DIALERS, "r")) == NULL)
            log_sys("can't open %s", DIALERS);
```

```
            diallineno = 0;
        }

    again:
        if (fgets(dialline, MAXLINE, fpdial) == NULL)
            return(-1);      /* EOF */
        diallineno++;

        if ( (dialptr->dialer = strtok(dialline, WHITE)) == NULL) {
            if (dialline[0] == '\n')
                goto again;      /* ignore empty line */
            log_quit("missing 'dialer' in Dialers file, line %d", diallineno)
        }
        if (dialptr->dialer[0] == '#')
            goto again;          /* ignore comment line */

        if ( (dialptr->sub = strtok(NULL, WHITE)) == NULL)
            log_quit("missing 'sub' in Dialers file, line %d",  diallineno);

        if ( (dialptr->expsend = strtok(NULL, "\n")) == NULL)
            log_quit("missing 'expsend' in Dialers file, line %d",diallineno);

        return(0);
    }

    void
    dial_rew(void)
    {
        if (fpdial != NULL)
            rewind(fpdial);
        diallineno = 0;
    }

    /* Find a dialer match */

    int
    dial_find(Dialers *dialptr, const Devices *devptr)
    {
        dial_rew();
        while (dial_next(dialptr) >= 0) {
            if (strcmp(dialptr->dialer, devptr->dialer) == 0)
                return(0);       /* found a dialer match */
        }
        sprintf(errmsg, "dialer '%s' not found\n", dialptr->dialer);
        return(-1);
    }
```

可以看到函数child_dial调用dial_find函数来查找*dialer*域与一个特定设备匹配的项。

从表18-4可以发现Systems和Devices文件由父进程处理，而子进程处理 Dialers文件。这也是整个设计的目的——父进程发现一个匹配的未被加锁的设备，然后创建一个子进程进行实际的拨号工作。

现在看一下程序18-10中的request函数。loop函数调用此函数，来确定对应一个特定的远程主机的而且未被加锁的拨号设备。为了达到这个目的， loop函数查找Systems和Devices文件。如果发现匹配项，则创建一个子进程。除了远程主机名字，还允许客户机指定拨号速度。例如，对于表18-1中的Systems文件，客户机的拨号请求可能是这样的：

```
call -s 9600 host1
```

这样就忽略了表18-1中速率不是9600的host1项。

需要注意的是，除非知道了子进程的 ID，否则不能使用lock_set函数来记录设备锁，但是在创建子进程之前又必须检查设备是否被加锁。因为总是要先加锁，然后才起动子进程拨号，

因此使用TELL_WAIT函数（见程序10-17）来同步父进程和子进程。同样需要注意：虽然检查函数is_locked和实际的加锁函数set_lock是两个分立的操作（也就是说，不是一个原子操作），但是这并没有竞争的问题，这是因为request函数只能被一个服务器精灵进程所调用——它不能被多个进程调用。

如果request返回0，则创建一个子进程来开始拨号。如果request返回1时，则说明或者远程系统的名称无效，或者所有可能的设备都已经被加锁。

程序18-10 request函数

```
#include    "calld.h"

int                         /* return 0 if OK, -1 on error */
request(Client *cliptr)
{
    pid_t   pid;

    errmsg[0] = 0;
        /* position where this client left off last (or rewind) */
    sys_posn(cliptr->sysftell);

    while ( (cliptr->sysftell = sys_next(&systems)) >= 0) {
        if (strcmp(cliptr->sysname, systems.name) == 0) {
                    /* system match */
                /* if client specified a speed, it must match too */
            if (cliptr->speed[0] != 0 &&
                strcmp(cliptr->speed, systems.class) != 0)
                    continue;   /* speeds don't match */

            DEBUG("trying sys: %s, %s, %s, %s", systems.name,
                    systems.type, systems.class, systems.phone);
            cliptr->foundone++;

            if (dev_find(&devices, &systems) < 0)
                break;
            DEBUG("trying dev: %s, %s, %s, %s", devices.type,
                    devices.line, devices.class, devices.dialer);
            if ( (pid = is_locked(devices.line)) != 0) {
                sprintf(errmsg, "device '%s' already locked by pid %d\n",
                                                    devices.line, pid);
                continue;   /* look for another entry in Systems file */
            }

                /* We've found a device that's not locked.
                   fork() a child to to the actual dialing. */
            TELL_WAIT();
            if ( (cliptr->pid = fork()) < 0)
                log_sys("fork error");
            else if (cliptr->pid == 0) {    /* child */
                WAIT_PARENT();      /* let parent set lock */
                child_dial(cliptr); /* never returns */
            }
            /* parent */
            lock_set(devices.line, cliptr->pid);
                    /* let child resume, now that lock is set */
            TELL_CHILD(cliptr->pid);
            return(0);  /* we've started a child */
        }
    }
    /* reached EOF on Systems file */
    if (cliptr->foundone == 0)
        sprintf(errmsg, "system '%s' not found\n", cliptr->sysname);
```

```
        else if (errmsg[0] == 0)
            sprintf(errmsg, "unable to connect to system `%s'\n",
                                                    cliptr->sysname);
        return(-1);     /* also, cliptr->sysftell is -1 */
    }
```

父进程专用函数的最后一个是sig_chld，它是SIGCHLD信号的处理程序。这个函数在程序18-11中。

程序18-11 sig_chld信号处理程序

```
#include    "calld.h"
#include    <sys/wait.h>

/* SIGCHLD handler, invoked when a child terminates. */

void
sig_chld(int signo)
{
    int     stat, errno_save;
    pid_t   pid;

    errno_save = errno;     /* log_msg() might change errno */
    chld_flag = 1;
    if ( (pid = waitpid(-1, &stat, 0)) <= 0)
        log_sys("waitpid error");

    if (WIFEXITED(stat) != 0)
                /* set client's childdone status for loop() */
        client_sigchld(pid, WEXITSTATUS(stat)+1);
    else
        log_msg("child %d terminated abnormally: %04x", pid, stat);

    errno = errno_save;
    return;     /* probably interrupts accept() in serv_accept() */
}
```

当一个子进程终止时，必须在client数组的适当位置上记录下它的终止状态和进程 ID。调用函数client_sigchld（见程序18-5）来完成这个工作。

需要注意，这里违反了第10章中的原则 —— 一个信号处理程序只处理一个全局变量，而不做其他。这里调用了 waitpid和函数client_sigchld（见程序18-5），后者对信号而言是安全的，它所做的只是在client数组的对应项中做记录。它并不创建或删除项（那样的话将是不可重入的），它也不调用任何系统函数。

POSIX.1定义的waitpid对信号而言是安全的（见表 10-3）。如果不从信号处理程序中调用waitpid，那么当chld_flag标志非0时父进程必须调用 waitpid。但是因为在主循环查询 chld_flag之前子进程有可能终止，则要或者在每个子进程终止时对 chld_flag加1（这样主循环就知道要调用多少次waitpid），或者在循环内使用WNOHANG标志，并调用waitpid（见表8-3）。最简单的方法还是从信号量处理程序中调用 waitpid，在client数组中记录下信息。

现在讨论客户机在拨号远程系统时需调用的一些函数。当 request调用child_dial来创建子进程（见程序18-12）时，会用到这些函数。

程序18-12 child_dial 函数

```
#include    "calld.h"
```

```
/* The child does the actual dialing and sends the fd back to
 * the client.  This function can't return to caller, must exit.
 * If successful, exit(0), else exit(1).
 * The child uses the following global variables, which are just
 * in the copy of the data space from the parent:
 *      cliptr->fd (to send DEBUG() output and fd back to client),
 *      cliptr->Debug (for all DEBUG() output), cliptr->parity,
 *      systems, devices, dialers. */

void
child_dial(Client *cliptr)
{
    int     fd, n;

    Debug = cliptr->Debug;
    DEBUG("child, pid %d", getpid());

    if (strcmp(devices.dialer, "direct") == 0) { /* direct tty line */
        fd = tty_open(systems.class, devices.line, cliptr->parity, 0);
        if (fd < 0)
            goto die;
    } else {                    /* else assume dialing is needed */
        if (dial_find(&dialers, &devices) < 0)
            goto die;
        fd = tty_open(systems.class, devices.line, cliptr->parity, 1);
        if (fd < 0)
            goto die;
        if (tty_dial(fd, systems.phone, dialers.dialer,
                             dialers.sub, dialers.expsend) < 0)
            goto die;
    }

    DEBUG("done");
            /* send the open descriptor to client */
    if (send_fd(cliptr->fd, fd) < 0)
        log_sys("send_fd error");
    exit(0);    /* parent will see this */

die:
    /* The child can't call send_err() as that would send the final
       2-byte protocol to the client.  We just send our error message
       back to the client.  If the parent finally gives up, it'll
       call send_err(). */

    n = strlen(errmsg);
    if (writen(cliptr->fd, errmsg, n) != n) /* send error to client */
        log_sys("writen error");
    exit(1);    /* parent will see this, release lock, and try again */
}
```

如果所使用的设备是直接相连的，那么只要调用tty_open来打开终端设备并设置终端设备参数。但如果这个设备是调制解调器，那么就要使用三个函数： dial_find（确定Dialers文件中的符合要求的项）、tty_open和tty_dial（进行实际的拨号）。

如果child_dial成功了，它调用send_fd（见程序15-5和程序15-9）把文件描述符传给客户机，并调用exit(0)。否则，它将出错消息传给客户机，并调用 exit(1)。客户机专用的流管道在创建时被复制，所以子进程可以直接将文件描述符或出错消息传给客户机。

客户机可以在发送给服务器的命令行中使用 -d选择项，与此对应，设置客户机专用标志变量Debug。程序18-13的DEBUG和DEBUG_NONL函数使用了此标志，将调试信息传回客户机。当拨号遇到问题时，这个调试信息就有用了。这两个函数主要被子进程所调用，父进程也会在request函数中调用它们（见程序18-10）。

程序18-13 调试函数

```
#include    "calld.h"
#include    <stdarg.h>

/* Note that all debug output goes back to the client. */

void
DEBUG(char *fmt, ...)       /* debug output, newline at end */
{
    va_list args;
    char    line[MAXLINE];
    int     n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    strcat(line, "\n");
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}

void
DEBUG_NONL(char *fmt, ...)  /* debug output, NO newline at end */
{
    va_list args;
    char    line[MAXLINE];
    int     n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}
```

程序18-14是tty_open函数。这个函数用来打开设备并设置设备工作模式，它适用于调制解调器设备或直接相连的设备。Systems文件中的*class*域和Devices文件指定了线路速率，客户机还可以指定校验方式。

程序18-14 tty_open函数

```
#include    "calld.h"
#include    <fcntl.h>
#include    <termios.h>

/* Open the terminal line */

int
tty_open(char *class, char *line, enum parity parity, int modem)
{
    int             fd, baud;
    char            devname[100];
    struct termios  term;

            /* first open the device */
    strcpy(devname, "/dev/");
```

```
    strcat(devname, line);
    if ( (fd = open(devname, O_RDWR | O_NONBLOCK)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                devname, strerror(errno));
        return(-1);
    }
    if (isatty(fd) == 0) {
        sprintf(errmsg, "%s is not a tty\n", devname);
        return(-1);
    }

            /* fetch then set modem's terminal status */
    if (tcgetattr(fd, &term) < 0)
        log_sys("tcgetattr error");

    if (parity == NONE)
        term.c_cflag = CS8;
    else if (parity == EVEN)
        term.c_cflag = CS7 | PARENB;
    else if (parity == ODD)
        term.c_cflag = CS7 | PARENB | PARODD;
    else
        log_quit("unknown parity");
    term.c_cflag |= CREAD |          /* enable receiver */
                    HUPCL;           /* lower modem lines on last close */
                                     /* 1 stop bit (since CSTOPB off) */

    if (modem == 0)
        term.c_cflag |= CLOCAL;      /* ignore modem status lines */

    term.c_oflag  = 0;               /* turn off all output processing */
    term.c_iflag  = IXON | IXOFF |   /* Xon/Xoff flow control (default) */
                    IGNBRK |         /* ignore breaks */
                    ISTRIP |         /* strip input to 7 bits */
                    IGNPAR;          /* ignore input parity errors */
    term.c_lflag  = 0;       /* everything off in local flag:
                                disables canonical mode, disables
                                signal generation, disables echo */
    term.c_cc[VMIN]  = 1;    /* 1 byte at a time, no timer */
    term.c_cc[VTIME] = 0;    /* (See Figure 18.10) */

    if      (strcmp(class, "38400") == 0)   baud = B38400;
    else if (strcmp(class, "19200") == 0)   baud = B19200;
    else if (strcmp(class, "9600") == 0)    baud = B9600;
    else if (strcmp(class, "4800") == 0)    baud = B4800;
    else if (strcmp(class, "2400") == 0)    baud = B2400;
    else if (strcmp(class, "1800") == 0)    baud = B1800;
    else if (strcmp(class, "1200") == 0)    baud = B1200;
    else if (strcmp(class, "600") == 0)     baud = B600;
    else if (strcmp(class, "300") == 0)     baud = B300;
    else if (strcmp(class, "200") == 0)     baud = B200;
    else if (strcmp(class, "150") == 0)     baud = B150;
    else if (strcmp(class, "134") == 0)     baud = B134;
    else if (strcmp(class, "110") == 0)     baud = B110;
    else if (strcmp(class, "75") == 0)      baud = B75;
    else if (strcmp(class, "50") == 0)      baud = B50;
    else {
        sprintf(errmsg, "invalid baud rate: %s\n", class);
        return(-1);
    }
    cfsetispeed(&term, baud);
    cfsetospeed(&term, baud);

    if (tcsetattr(fd, TCSANOW, &term) < 0)  /* set attributes */
        log_sys("tcsetattr error");

    DEBUG("tty open");
```

```
        clr_fl(fd, O_NONBLOCK);      /* turn off nonblocking */
        return(fd);
    }
```

因为有时要打开连接在调制解调器上的终端必须先检测到调制解调器的载波才行，所以我们以非阻塞方式打开终端设备。我们是在向外拨号，而不是向内拨号，所以不愿意等待。在这个函数的结尾处调用clr_fl函数来清除这种非阻塞方式。在函数tty_open中，调制解调器和直接相连的线路的唯一区别就是对直接相连的线路要设置CLOCAL位。

详细的拨号过程在函数tty_dial（见程序18-15）中实现。这个函数只在调制解调器拨号时被调用，打开直接相连设备时不调用。

程序18-15　tty_dial 函数

```
#include    "calld.h"

int
tty_dial(int fd, char *phone, char *dialer, char *sub, char *expsend)
{
    char    *ptr;

    ptr = strtok(expsend, WHITE);   /* first expect string */
    for ( ; ; ) {
        DEBUG_NONL("expect = %s\nread: ", ptr);
        if (expect_str(fd, ptr) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);      /* at the end of the expect/send */
        DEBUG_NONL("send = %s\nwrite: ", ptr);
        if (send_str(fd, ptr, phone, 0) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);      /* at the end of the expect/send */
    }
}
```

这个函数只是调用一个函数来处理期望得到的字符串，调用另外一个函数来处理发送的字符串。当没有发送或期望字符串时工作就完成了（这里并不处理表18-3中的*sub*域）。

程序18-16是输出发送字符串的send_str函数。为了不使得这个程序太过臃肿，我们没有实现每一个转义序列 —— 我们只保证这个程序能处理表18-3中的Dialers文件。

程序18-16　send_str函数

```
#include    "calld.h"

int
send_str(int fd, char *ptr, char *phone, int echocheck)
{
    char    c, tempc;

        /* go though send string, converting escape sequences on the fly */
    while ( (c = *ptr++) != 0) {
        if (c == '\\') {
            if (*ptr == 0) {
                sprintf(errmsg, "backslash at end of send string\n");
                return(-1);
            }
```

```
        c = *ptr++;     /* char following backslash */

        switch (c) {
        case 'c':       /* no CR, if at end of string */
            if (*ptr == 0)
                goto returnok;
            continue;   /* ignore if not at end of string */

        case 'd':       /* 2 second delay */
            DEBUG_NONL("<delay>");
            sleep(2);
            continue;

        case 'p':       /* 0.25 second pause */
            DEBUG_NONL("<pause>");
            sleep_us(250000);   /* Exercise 12.6 */
            continue;

        case 'e':
            DEBUG_NONL("<echo check off>");
            echocheck = 0;
            continue;

        case 'E':
            DEBUG_NONL("<echo check on>");
            echocheck = 1;
            continue;

        case 'T':       /* output phone number */
            send_str(fd, phone, phone, echocheck);  /* recursive */
            continue;

        case 'r':
            c = '\r';
            break;

        case 's':
            c = ' ';
            break;


            /* room for lots more case statements ... */


        default:
            sprintf(errmsg, "unknown send escape char: \\%s\n",
                                                    ctl_str(c));
            return(-1);
        }
    }

    DEBUG_NONL("%s", ctl_str(c));
    if (write(fd, &c, 1) != 1)
        log_sys("write error");
        if (echocheck) {        /* wait for char to be echoed */
            do {
                if (read(fd, &tempc, 1) != 1)
                    log_sys("read error");
                DEBUG_NONL("{%s}", ctl_str(tempc));
            } while (tempc != c);
        }
    }
    c = '\r';   /* if no \c at end of string, CR written at end */
    DEBUG_NONL("%s", ctl_str(c));
    if (write(fd, &c, 1) != 1)
        log_sys("write error");
```

```
returnok:
    DEBUG("");
    return(0);
}
```

send_str调用函数ctl_str函数将ASCII控制字符转换成可以打印的形式。程序 18-17是ctl_str函数。

程序18-17 ctl_str函数

```
#include    "calld.h"

/* Make a printable string of the character "c", which may be a
 * control character.  Works only with ASCII. */

char *
ctl_str(char c)
{
    static char tempstr[6];     /* biggest is "\177" + null */

    c &= 255;
    if (c == 0)
        return("\\0");          /* really shouldn't see a null */
    else if (c < 040)
        sprintf(tempstr, "^%c", c + 'A' - 1);
    else if (c == 0177)
        return("DEL");
    else if (c > 0177)
        sprintf(tempstr, "\\%03o", c);
    else
        sprintf(tempstr, "%c", c);
    return(tempstr);
}
```

在整个拨号过程中最困难的是识别期望字符串。程序 18-18就是来完成这项工作的函数expect_str。(对于发送字符串，我们只实现了Dialers文件所提供的特性的一个子集。)

程序18-18 读取和识别期望字符串的函数组

```
#include    "calld.h"

#define EXPALRM     45          /* alarm time to read expect string */

static int      expalarm = EXPALRM;
static void     sig_alrm(int);
static volatile sig_atomic_t    caught_alrm;

static size_t   exp_read(int, char *);

int                     /* return 0 if got it, -1 if not */
expect_str(int fd, char *ptr)
{
    char    expstr[MAXLINE], inbuf[MAXLINE];
    char    c, *src, *dst, *inptr, *cmpptr;
    int     i, matchlen;

    if (strcmp(ptr, "\"\"") == 0)
        goto returnok;      /* special case of "" (expect nothing) */

            /* copy expect string, converting escape sequences */
    for (src = ptr, dst = expstr; (c = *src++) != 0; ) {
```

```
        if (c == '\\') {
            if (*src == 0) {
                sprintf(errmsg, "invalid expect string: %s\n", ptr);
                return(-1);
            }
            c = *src++;     /* char following backslash */
            switch (c) {
            case 'r':   c = '\r'; break;
            case 's':   c = ' '; break;
                /* room for lots more case statements ... */
            default:
                sprintf(errmsg, "unknown expect escape char: \\%s\n",
                                                    ctl_str(c));
                return(-1);
            }
        }
        *dst++ = c;
    }
    *dst = 0;
    matchlen = strlen(expstr);

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        log_quit("signal error");
    caught_alrm = 0;
    alarm(expalarm);

    do {
        if (exp_read(fd, &c) < 0)
            return(-1);
    } while (c != expstr[0]);   /* skip until first chars equal */

        cmpptr = inptr = inbuf;
        *inptr = c;

        for (i = 1; i < matchlen; i++) {    /* read matchlen chars */
            inptr++;
            if (exp_read(fd, inptr) < 0)
                return(-1);
        }

        for ( ; ; ) {       /* keep reading until we have a match */
            if (strncmp(cmpptr, expstr, matchlen) == 0)
                break;      /* have a match */
            inptr++;
            if (exp_read(fd, inptr) < 0)
                return(-1);
            cmpptr++;
        }
    returnok:
        alarm(0);
        DEBUG("\nexpect: got it");
        return(0);
    }

    size_t          /* read one byte, handle timeout errors & DEBUG */
    exp_read(int fd, char *buf)
    {
        if (caught_alrm) {  /* test flag before blocking in read */
            DEBUG("\nread timeout");
            return(-1);
        }
        if (read(fd, buf, 1) == 1) {
            DEBUG_NONL("%s", ctl_str(*buf));
            return(1);
        }
```

```
        if (errno == EINTR && caught_alrm) {
            DEBUG("\nread timeout");
            return(-1);
        }
        log_sys("read error");
    }

    static void
    sig_alrm(int signo)
    {
        caught_alrm = 1;
        return;
    }
```

我们首先复制期望字符串，转换特殊字符。匹配方法是从调制解调器读取字符直到该字符与期望字符串的第一个字符匹配，然后再读取与期望字符串同样多的字符。从这开始，从调制解调器连续地读取字符到缓存中，把它们与期望字符串比较，直至得到整个匹配串（当然还有更好的匹配算法，我们所选的这个算法只是为了简化编程。从调制解调器读取的字符一般 50个一组读入，期望字符串的长度一般是10~20个字符）。

每当尝试匹配一个期望串时，都必须设置一个警告标志，警告是我们可以确定没有收到匹配字符串的唯一方式。

现在介绍完了服务器精灵进程。这个精灵进程所做的其实就是打开一个终端设备和使用调制解调器拨号。至于打开终端设备以后的工作取决于客户机。下面将看到一个提供了类似于 cu和tip界面的客户机，它允许我们对远程系统拨号并登录。

18.7 客户机设计

客户机与服务器之间的界面只是若干行代码。客户机生成一个命令行，发送到服务器，然后收到一个文件描述符或者一个错误消息。客户机的设计着重于客户机如何处理返回的文件描述符。这一节描述了一个类似于 cu和tip程序的call客户机程序。这个程序允许我们对远程系统拨号，并登录。远程系统并不一定是一个 UNIX系统。我们可以使用这个程序来同那些与本机通过RS-232串口连接的系统或设备进行通信。

18.7.1 终端行规程

图12-7和12-8给出了一个调制解调器拨号器的概况。图 18-4则是图12-7的扩充。这里要注意的是，在用户和调制解调器之间有两个行规程，并假设我们使用这个程序来拨号一个远程UNIX系统。（回忆程序12-10的输出，与一个基于流的终端系统相比，图 18-4只是一个简化。事实上可能有多个流组成这个行规程，可能有多个模块构成终端设备驱动程序。此外没有显式地表明流首。）

图18-4本地系统中调制解调器上方的两个虚线框中的过程是由服务器的 tty_open函数（见程序18-14）建立的。该函数设置虚线框中的终端行规程为非规范模式。本地系统中的调制解调器被服务器函数tty_dial所拨号（见程序18-15）。终端行规程的虚线框和call进程之间的两个箭头对应于服务器端返回的文件描述符。（这里把一个描述符显示为两个箭头，是为了重申它是一个全双工的描述符）。

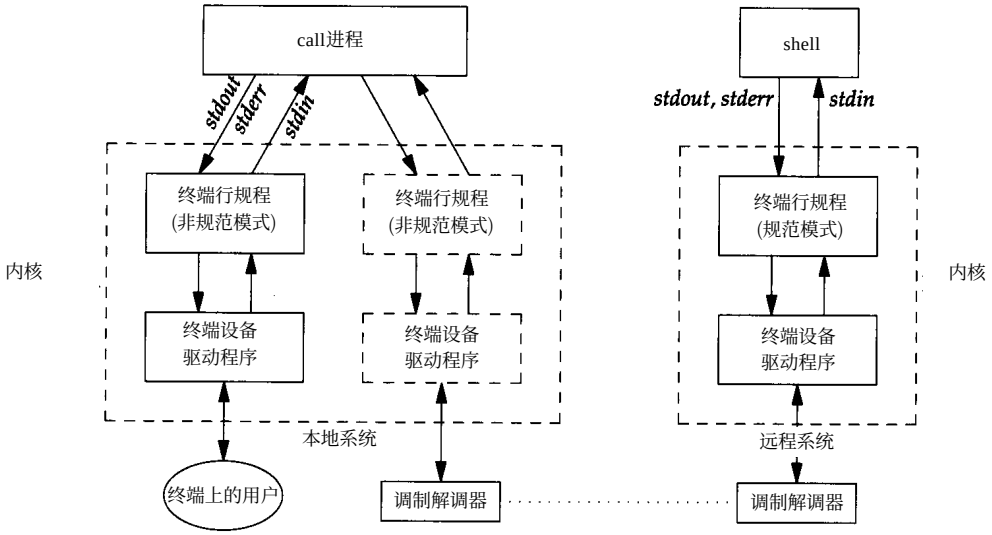

图18-4 调制解调器拨号器登录远程UNIX主机过程的示意图

远程系统shell下方的行规程框被登录进程设置为规范模式。当拨通远程系统时，我们希望输入的特殊字符（如文件结束、删除行等）被远程主机上的行规程模块所认识。这样，就必须将终端（标准输入、标准输出、call进程的标准出错）上方的行规程模块设置为非规范模式。

18.7.2 一个进程还是两个进程

在图18-4中，我们看到的call程序只有一个进程。因为要读取两个描述符、写两个描述符，所以要求支持像select或者poll一样的I/O多路转接函数。我们同样也可以把客户机程序设计为如图12-8中的两个进程：一个父进程、一个子进程。图 18-5就显示了这两个进程以及它们下方的行规程。从历史上看， cu和tip就像图18-5中所示，一直是两个进程。这是因为早期的 UNIX系统不支持I/O多路转接函数。

我们基于以下两个原因采用一个进程：

(1) 采用两个进程会使得客户机的终止变得复杂。如果在一行的开始处输入 ~.（一个波浪符号加上一个点）来终止连接，子进程认识了这个符号并终止，父进程却必须捕捉到SIGCHLD信号才终止。

如果这个连接是由远程系统终止的，或者线路断掉了，父进程从调制解调器描述符读取到文件终止符号而检测到这情况。然后，父进程必须通知子进程，这样子进程才会终止。

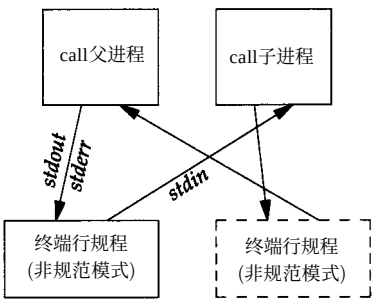

图18-5 两个进程的call程序

使用一个进程则避免了终止时的进程间通信。

(2) 我们要在客户机实现一个文件传送函数，类似于 cu和tip程序中的put和take命令。我们在标准输入中输入这些命令，在一行开始处输入一个波浪号（缺省的转义字符）。如果采用两个进程，这些命令被子进程所识别（见图 18-5）。但是客户机接收到的文件（使用 take命令），会使用调制解调器描述符，而这时这个描述符正在被父进程读取。这样的话，为了实现 take命令，子进程就必须通知父进程，让父进程停止从调制解调器读取。父进程可能会在读取描述符

时被阻塞，这样还需要一个信号来中断父进程的读取。当子进程结束后，还需要另外通知父进程继续读取调制解调器。这样经常会使情况变得混乱。

一个单一的进程会简化整个客户机。但是，若只使用一个进程，则无法只是停止子进程的作业，而BSD的tip程序则支持这种特性。它允许停止子进程而父进程仍然继续运行。这意味着所有的终端输入被重定向到shell，而不是子进程中，这使我们可以工作在本地系统，仍然可以看到远程系统产生的输出。当在远程系统上运行一个很长时间的作业，又希望在本地系统中看到它产生的输出的情况，这个功能就很方便。

现在来看一下实现客户机的源码。

18.8 客户机源码

因为客户机并不处理与远程系统相连接的细节，与服务器比较，客户机比较小些。客户机程序中大约有一半用于处理那些类似于take和put的命令。

程序18-19是call.h头文件，它被包含在所有的源文件中。

程序18-19 call.h 头文件

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

#define CS_CALL "/home/stevens/calld"   /* well-known server name */
#define CL_CALL "call"                  /* command for server */

                /* declare global variables */
extern char  escapec;   /* tilde for local commands */
extern char *src;       /* for take and put commands */
extern char *dst;       /* for take and put commands */

                /* function prototypes */
int     call(const char *);
int     doescape(int);
void    loop(int);
int     prompt_read(char *, int (*)(int, char **));
void    put(int);
void    take(int);
int     take_put_args(int, char **);
```

发送给服务器的命令和服务器的众所周知名字必须与程序18-1中的值保持一致。

程序18-20是客户机的main函数。

程序18-20 main函数

```
#include    "call.h"

            /* define global variables */
char     escapec = '~';
char    *src;
char    *dst;

static void usage(char *);

int
main(int argc, char *argv[])
```

```
{
    int         c, remfd, debug;
    char        args[MAXLINE];

    args[0] = 0;    /* build arg list for conn server here */
    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = 1;
            strcat(args, "-d ");
            break;
        case 'e':       /* even parity */
            strcat(args, "-e ");
            break;

        case 'o':       /* odd parity */
            strcat(args, "-o ");
            break;

        case 's':       /* speed */
            strcat(args, "-s ");
            strcat(args, optarg);
            strcat(args, " ");
            break;

        case '?':
            usage("unrecognized option");
        }
    }
    if (optind < argc)
        strcat(args, argv[optind]); /* name of host to call */
    else
        usage("missing <hostname> to call");

    if ( (remfd = call(args)) < 0)  /* place the call */
        exit(1);    /* call() prints reason for failure */
    printf("Connected\n");

    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");

    loop(remfd);                    /* and do it */

    printf("Disconnected\n\r");
    exit(0);
}

static void
usage(char *msg)
{
    err_quit("%s\nusage: call -d -e -o -s<speed> <hostname>", msg);
}
```

main函数处理命令行参数，把它们保存在args数组中，再把它们发送给服务器。call函数与服务器联系并返回远程系统的文件描述符。

图18-4终端上方的行规程模块被tty_raw函数（见程序11-10）设置为非规范模式。为了在完成工作后重置终端，我们将tty_atexit函数设置为终止处理程序。

调用函数loop把输入到调制解调器的所有东西和从调制解调器读取的东西复制到终端上。

程序18-21中的call函数联系服务器以得到一个调制解调器的文件描述符。如前所述，它只

用了若干行代码就完成了这项工作。

程序18-21　call函数

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

/* Place the call by sending the "args" to the calling server,
 * and reading a file descriptor back. */

int
call(const char *args)
{
    int             csfd, len;
    struct iovec    iov[2];

                    /* create connection to conn server */
    if ( (csfd = cli_conn(CS_CALL)) < 0)
        err_sys("cli_conn error");

    iov[0].iov_base = CL_CALL " ";
    iov[0].iov_len  = strlen(CL_CALL) + 1;
    iov[1].iov_base = (char *) args;
    iov[1].iov_len  = strlen(args) + 1;
                            /* null at end of args always sent */
    len = iov[0].iov_len + iov[1].iov_len;
    if (writev(csfd, &iov[0], 2) != len)
        err_sys("writev error");

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

loop函数处理两个输入流和两个输出流之间的多路转接。可以使用 poll或者select，这取决于本地系统提供什么函数。程序18-22使用poll来实现。

程序18-22　使用poll函数的loop函数

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

/* Copy everything from stdin to "remfd",
 * and everything from "remfd" to stdout. */

#define BUFFSIZE    512

void
loop(int remfd)

{
    int             bol, n, nread;
    char            c, buff[BUFFSIZE];
    struct pollfd   fds[2];

    setbuf(stdout, NULL);       /* set stdout unbuffered */
                                /* (for printfs in take() and put() */
    fds[0].fd = STDIN_FILENO;   /* user's terminal input */
    fds[0].events = POLLIN;
    fds[1].fd = remfd;          /* input from remote (modem) */
    fds[1].events = POLLIN;

    for ( ; ; ) {
```

```
        if (poll(fds, 2, INFTIM) <= 0)
            err_sys("poll error");

        if (fds[0].revents & POLLIN) {  /* data to read on stdin */
            if (read(STDIN_FILENO, &c, 1) != 1)
                err_sys("read error from stdin");

            if (c == escapec && bol) {
                if ( (n = doescape(remfd)) < 0)
                    break;      /* user wants to terminate */
                else if (n == 0)
                    continue;   /* escape seq has been processed */

                /* else, char following escape was not special,
                   so it's returned and echoed below */
                c = n;
            }
            if (c == '\r' || c == '\n')
                bol = 1;
            else
                bol = 0;

            if (write(remfd, &c, 1) != 1)
                err_sys("write error");
        }
        if (fds[0].revents & POLLHUP)
            break;      /* stdin hangup -> done */

        if (fds[1].revents & POLLIN) {  /* data to read from remote */
            if ( (nread = read(remfd, buff, BUFFSIZE)) <= 0)
                break;      /* error or EOF, terminate */

            if (writen(STDOUT_FILENO, buff, nread) != nread)
                err_sys("writen error to stdout");
        }
        if (fds[1].revents & POLLHUP)
            break;      /* modem hangup -> done */
    }
}
```

loop函数的基本循环只是在等待从调制解调器或终端来的数据。当从终端读取到数据时，就把它拷贝到调制解调器，反过来也一样。稍微复杂一点的是，要识别一行的第一个转义字符（波浪线）。

要注意，从终端（标准输入）每次读取一个字符，但从调制解调器每次读取一个缓存的内容。每次从终端读取一个字符的原因之一是，我们必须在新行开始处注意每一个字符以识别可能的特殊字符。虽然每次I/O读取一个字符浪费了CPU时间（见表3-1），但是一般来说从终端的输入要比从远程系统得到的内容要少的多。（在使用本程序的一个定量测试中，本地每一个输入，远程系统就有大约100个输出。）

当检测到转义字符时，就调用doescape函数来处理这个输入的命令（见程序18-23）。我们这里只支持5个命令。简单的命令就在这个函数内部实现，复杂的命令，如take和put，则通过各自的函数实现。

• 一个句号终止客户机。对于某些设备，如激光打印机，这是终止客户机的唯一方法。当登录到远程系统后（见图18-4），退出远程系统会引起调制解调器掉线，loop函数就会从调制解调器得到挂断信号。

• 如果系统支持作业控制，则识别作业控制挂起字符，并挂起客户机。注意直接识别这个字符要比使用行规程识别它而产生SIGSTOP信号要简单些（见程序10-22）。那样，在停止之前不得不重置终端模式，而继续执行时又要重置。

• 调制解调器描述符中英镑符号产生BREAK。使用POSIX.1 的tcsendbreak函数来实现这个BREAK（见11.8节）。这个BREAK经常会引起远程系统的getty或者ttymon程序改变线路速率。（见9.2节）

• take和put命令需要分别调用不同的函数。区分这两个命令的方法是记住它们说明了客户机在本地系统要做的操作：从远程系统取（take）一个文件或者送（put）一个文件给远程系统。

程序18-24是实现take命令的代码。take函数先是调用prompt_read（见程序18-25），它显示~[take]. 提示来响应~t命令。prompt_read函数然后从终端读取输入的一行，包含源路径（远程系统上的文件）和目标路径（本机上的文件）。把读取的结果储存在全局变量src和dst中。

程序18-23 escape函数

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

/* Called when first character of a line is the escape character
 * (tilde).  Read the next character and process.  Return -1
 * if next character is "terminate" character, 0 if next character
 * is valid command character (that's been processed), or next
 * character itself (if the next character is not special). */

int
doescape(int remfd)
{
    char    c;

    if (read(STDIN_FILENO, &c, 1) != 1)     /* next input char */
        err_sys("read error from stdin");

    if (c == escapec)       /* two in a row -> process as one */
        return(escapec);

    else if (c == '.') {    /* terminate */
        write(STDOUT_FILENO, "~.\n\r", 4);
        return(-1);

#ifdef  VSUSP
    } else if (c == tty_termios()->c_cc[VSUSP]) { /* suspend client */
        tty_reset(STDIN_FILENO);    /* restore tty mode */
        kill(getpid(), SIGTSTP);    /* suspend ourself */

        tty_raw(STDIN_FILENO);      /* and reset tty to raw */
        return(0);
#endif

    } else if (c == '#') {  /* generate break */
        tcsendbreak(remfd, 0);
        return(0);

    } else if (c == 't') {  /* take a file from remote host */
        take(remfd);
        return(0);

    } else if (c == 'p') {  /* put a file to remote host */
        put(remfd);
        return(0);
    }

    return(c);          /* not a special character */
}
```

程序18-24 处理take命令

```
#include    "call.h"

#define CTRLA   001     /* eof designator for take */

static int       rem_read(int);
static char      rem_buf[MAXLINE];
static char     *rem_ptr;
static int       rem_cnt = 0;

/* Copy a file from remote to local. */

void
take(int remfd)
{
    int     n, linecnt;
    char    c, cmd[MAXLINE];
    FILE    *fpout;

    if (prompt_read("~[take] ", take_put_args) < 0) {
        printf("usage: [take] <sourcefile> <destfile>\n\r");
        return;
    }

            /* open local output file */
    if ( (fpout = fopen(dst, "w")) == NULL) {
        err_ret("can't open %s for writing", dst);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

            /* send cat/echo command to remote host */
    sprintf(cmd, "cat %s; echo %c\r", src, CTRLA);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");

            /* read echo of cat/echo command line from remote host */
    rem_cnt = 0;        /* initialize rem_read() */
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            return;     /* line has dropped */
        if (c == '\n')
            break;      /* end of echo line */
    }

            /* read file from remote host */
    linecnt = 0;
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            break;          /* line has dropped */
        if (c == CTRLA)
            break;          /* all done */
        if (c == '\r')
            continue;       /* ignore returns */
        if (c == '\n')      /* but newlines are written to file */
            printf("\r%d", ++linecnt);
        if (putc(c, fpout) == EOF)
            break;          /* output error */
    }
    if (ferror(fpout) || fclose(fpout) == EOF) {
        err_msg("output error to local file");
        putc('\r', stderr);
        fflush(stderr);
```

```
        }
        c = '\n';
        write(remfd, &c, 1);
    }

    /* Read from remote.  Read up to MAXLINE, but parcel out one
     * character at a time. */

    int
    rem_read(int remfd)
    {
        if (rem_cnt <= 0) {
            if ( (rem_cnt = read(remfd, rem_buf, MAXLINE)) < 0)
                err_sys("read error");
            else if (rem_cnt == 0)
                return(0);
            rem_ptr = rem_buf;
        }
        rem_cnt--;
        return(*rem_ptr++ & 0177);
    }
```

在take函数为写而打开本地文件后，发送以下命令到远程主机：

```
cat sourcefile; echo ^A
```

这使远程主机执行cat命令，然后回显Ctrl-A的ASCII字符。我们从远程主机的返回信息中查看Ctrl-A字符，当发现后，就知道文件传送已经结束了。注意同样必须读取发送的命令行的回应，只有确定了命令的回应后，才能开始接受cat 命令的输出。

当从远程文件中读取时，查找新行的标志，记录下新行的数目。在左边界显示这个行号，覆盖原有的行号（在printf中使用回车终止一行，而没有使用换行符）。这就在终端上提供了一个可视的文件传送进度指示，最后在结尾处显示最终的行号。

客户机源文件中也包含了rem_read函数，这个函数被用来从远程主机读取每个字符。每次读取一个缓存的大小，但每次只返回拨号者一个字符。

最初，take命令的程序每次读取一个字符，就像以前的cu和tip程序一样。十年以前，那时1200波特率的调制解调器还被认为是高速，这样做是可以的。但是使用现在快速的调制解调器，它们以9600或以上的波特率将字符传送给终端设备，那么字符就会丢失。作者使用的是一个PEP模式Telebit T2500的调制解调器，即使在本地主机和远程主机都使用了流量控制情况下，用cu和tip命令时还是会遇到这个问题，当传送一个大的文本文件（大约75 000字节）时，大约在一半的时候字符丢失了，只得重新传送。

解决方法是对rem_read函数重新编码，每次读取一个缓存的字符。这样做会将 CPU的时间减少到大约1/3左右（传送这个75 000字节的文件，从16秒减少到5秒），而且每次都提供了可信的文件传送。在rem_read函数中临时加入一个计数器，可以看到每次调用时读取到多少字符。表18-5显示了一个结果。

表18-5 文件传送时读取的字节数

字节数	次数	字节数	次数	字节数	次数	字节数	次数
1	1	28	2	39	1	55	1
13	1	29	1	40	1	56	9

(续)

字节数	次数	字节数	次数	字节数	次数	字节数	次数
16	1	32	1	46	1	57	751
17	1	33	1	48	2	58	530
22	1	34	1	51	2	59	2
24	1	35	1	52	2	114	1
25	4	37	1	53	1	115	1
26	3	38	1	54	1	194	1

在这个结果中，只有一次是返回一个字节， 99%次是返回57或者58个字节。这个改动把读取的次数从75 000次减少到1 329次。

注意表18-5中每次返回的字节数，这时行规程模块通过tty_open函数（见程序18-14）将它的MIN设置为1且TIME设置为0。这就是11.11节中的实例B。在这里要注意MIN仅仅只是最小值，如果要求的数量比最小值多而且远程系统已经准备好，则可以读取到更多的字节。当 MIN被设置为1时，我们并没有被限制为每次读取一个字节。

程序18-25是两个辅助的函数take_put_args和prompt_read。其中prompt_read会被take和put两函数调用，并以take_put_args作为它的一个参数，然后，由buf_args函数调用（见程序15-17）。

程序18-25　take_put_args和prompt_read函数

```
#include    "call.h"

/* Process the argv-style arguments for take or put commands. */

int
take_put_args(int argc, char **argv)
{
    if (argc == 1) {
        src = dst = argv[0];
        return(0);
    } else if (argc == 2) {
        src = argv[0];
        dst = argv[1];
        return(0);
    }
    return(-1);
}

static char cmdargs[MAXLINE];
            /* can't be automatic; src/dst point into here */

/* Read a line from the user.  Call our buf_args() function to
 * break it into an argv-style array, and call userfunc() to
 * process the arguments. */

int
prompt_read(char *prompt, int (*userfunc)(int, char **))
{
    int     n;
    char    c, *ptr;

    tty_reset(STDIN_FILENO);    /* allow user's editing chars */

    n = strlen(prompt);
```

```
        if (write(STDOUT_FILENO, prompt, n) != n)
            err_sys("write error");

        ptr = cmdargs;
        for ( ; ; ) {
            if ( (n = read(STDIN_FILENO, &c, 1)) < 0)
                err_sys("read error");
            else if (n == 0)
                break;
            if (c == '\n')
                break;
            if (ptr < &cmdargs[MAXLINE-2])
                *ptr++ = c;
        }
        *ptr = 0;       /* null terminate */

        tty_raw(STDIN_FILENO);      /* reset tty mode to raw */

        return( buf_args(cmdargs, userfunc) );
                    /* return whatever userfunc() returns */
    }
```

函数prompt_read从终端读入一行，然后调用buf_args将这一行分解为标准的参数列表，然后用take_put_args来处理这些参数。需要注意的是终端被重置到规范模式来读取参数，当输入命令行时允许使用标准的编辑字符。

最后的客户机函数是put，见程序18-26。这个函数被调用来拷贝一个本地文件到远程主机。

程序18-26 put 函数

```
#include    "call.h"

/* Copy a file from local to remote. */

void
put(int remfd)
{
    int     i, n, linecnt;
    char    c, cmd[MAXLINE];
    FILE    *fpin;

    if (prompt_read("~[put] ", take_put_args) < 0) {
        printf("usage: [put] <sourcefile> <destfile>\n\r");
        return;
    }

            /* open local input file */
    if ( (fpin = fopen(src, "r")) == NULL) {
        err_ret("can't open %s for reading", src);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

            /* send stty/cat/stty command to remote host */
    sprintf(cmd, "stty -echo; cat >%s; stty echo\r", dst);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");
    tcdrain(remfd);     /* wait for our output to be sent */
    sleep(4);           /* and let stty take effect */
```

```
            /* send file to remote host */
        linecnt = 0;
        for ( ; ; ) {
            if ( (i = getc(fpin)) == EOF)
                break;          /* all done */
            c = i;
            if (write(remfd, &c, 1) != 1)
                break;          /* line has probably dropped */
            if (c == '\n')      /* increment and display line counter */
                printf("\r%d", ++linecnt);
        }
            /* send EOF to remote, to terminate cat */
        c = tty_termios()->c_cc[VEOF];
        write(remfd, &c, 1);
        tcdrain(remfd);             /* wait for our output to be sent */
        sleep(2);
        tcflush(remfd, TCIOFLUSH);  /* flush echo of stty/cat/stty */
        c = '\n';
        write(remfd, &c, 1);

        if (ferror(fpin)) {
            err_msg("read error of local file");
            putc('\r', stderr);
            fflush(stderr);
        }
        fclose(fpin);
    }
```

就像take命令一样，我们发送一个命令字符串给远程系统。这次命令是：

```
stty -echo; cat destfile; stty echo
```

我们必须关闭回送（echo），否则整个文件将回送给我们。为了终止cat命令，发送文件终止符（一般采用Ctrl-D）。这要求本机和远程主机使用相同的文件终止符。另外，文件中不能含有远程系统中的ERASE和KILL特殊字符。

18.9 小结

本章讨论了两个不同的程序：一个是服务器的精灵进程用来拨号远程系统，另一个是远程登录程序，它使用服务器连接远程系统。服务器也可以被其他需要与远程系统连接的程序或其他与主机通过异步终端端口连接的硬件设备所使用。

服务器的设计类似与15.6节中的open服务器，需要使用流管道，每个客户机要连接到服务器上，并传递文件描述符。这些高级的进程间通信特性允许我们建立具有我们所需特定功能（见18.3节）的客户机-服务器应用。

客户机类似UNIX系统中的cu和tip程序，但在本章的例子中并不需要关心拨号、与 UUCP锁定的文件是否冲突或者如何建立调制解调器行规程模块等细节。服务器将处理这些细节。它使我们集中在客户机所关心的事情上，例如如何提供一个可信的文件传输机制等。

习题

18.1 怎样才能避免18.3节中的第(1)步（手工起动服务器）？

18.2 在程序18-4中，如果不将optind设置为1会出现什么情况？

18.3 如果在request函数（见程序18-10）中创建一个子进程后，在子进程终止前修改了Systems文件会出现什么情况？

18.4 7.8节提到在可重新分配的存储区域中要谨慎使用指针，因为这区域可以移动。但在18.3节中，为什么能在可重新分配的client数组上使用cliptr指针呢？

18.5 如果take或put命令的路径参数中含有一个分号，会出现什么情况呢？

18.6 修改服务器，使得它在起动时一次读取三个数据文件并保存在存储区域中。那么，如果数据文件被修改了，服务器应当怎么处理呢？

18.7 程序18-21中，为什么要在填充writev函数中的结构时，发送一个参数args？

18.8 用select函数代替poll函数来实现程序18-22。

18.9 如何确定用put命令传送的文件不包含可以被远程系统上行规程解释的字符？

18.10 dialing 函数越早发现拨号失败，才能越早尝试Systems文件中的下一项。如果可以确定远程系统的电话占线，而不是等到计时器 expect_str超时才确定拨号失败，就可以节约15~20秒时间。为了处理这些错误，在4.3BSD UUCP的期望-发送字符串中增加了期望字符串ABORT。它加上一个说明字符串，就可以放弃当前的拨号。例如，在表 18-3最后期望字符串CONNECT\Sfast的前面，可以加上ABORT BUSY来实现这个功能。