The Lost Chapter

A Modem Dialer

1 Introduction

Programs that deal with modems have always had a hard time coping with the wide variety of modems that are available. On most UNIX systems there are two programs that handle modems. The first is a remote login program that lets us dial some other computer, log in, and use that system. In the System V world this program is called `cu`, while Berkeley systems call it `tip`. Both programs do similar things, and both have knowledge of many different types of modems. The other program that uses a modem is `uucico`, part of the UUCP package. The problem is that knowledge that a modem is being used is often built into these programs, and if we want to write some other program that needs a modem, we have to perform many of the same tasks. Also, if we want to change these programs to use some form of communication instead of a modem (such as a network connection), major changes are often required.

In this chapter we develop a separate program that handles all the details of modem handling. This lets us isolate all these details into a single program, instead of having it spread through multiple programs. (This program was motivated by the connection server described in Presotto and Ritchie [1990].) To use this program we have to be able to invoke it and have it pass back a file descriptor, as we described in Section 17.4. We then use this program in developing a remote login program (similar to `cu` and `tip`).

2 History

The `cu`(1) command (which stands for "call UNIX") appeared in Version 7. But it handled only one particular ACU (automatic call unit). At Berkeley, Bill Shannon modified `cu`, and it appeared in 4.2BSD as the `tip`(1) program. The biggest change was the use of a text file (`/etc/remote`) to contain all the information for various systems

(phone number, preferred dialer, baud rate, parity, flow control, etc.). This version of `tip` supported about six different call units and modems, but to add support for some other type of modem required source code changes.

Along with `cu` and `tip`, the UUCP system also accessed modems and automatic call units. UUCP managed locks on different modems, so that multiple instances of UUCP could be running at the same time. The `tip` and `cu` programs had to honor the UUCP locking protocol, to avoid interfering with UUCP. On the BSD systems, UUCP developed its own set of dialer functions. These functions were link edited into the UUCP executable, which meant the addition of a new modem type required source code changes.

SVR2 provided a `dial`(3) function that attempted to isolate the unique features of modem dialing into a single library function. It was used by `cu`, but not by UUCP. This function was in the standard C library, so it was available to any program.

The Honey DanBer UUCP system [Redman 1989] took the modem commands out of the C source files and put them into a `Dialers` file. This allowed the addition of new modem types without having to modify the source code. But the functions used by `cu` and UUCP to access the `Dialers` file were not generally available. This means that without redeveloping all the code to process the dialing information in the `Dialers` file, programs other than `cu` and UUCP couldn't use this file.

Throughout all these versions of `cu`, `tip`, and UUCP, locking was required to assure only a single program accessed a single device at a time. Since all these programs worked across many different systems, earlier versions of which provided no record locking, a rudimentary form of file locking was used. This could lead to lock files being left around after a program crashed, and ad hoc techniques were developed to handle this. (We can't use record locking on special device files, so record locking by itself doesn't provide a complete solution.)

3   Program Design

Let's detail the features that we want the modem dialer to have.

1. It must be possible to add new modem types without requiring source code changes.

   To obtain this feature, we'll use the Honey DanBer `Dialers` file. We'll put all the code that uses this file to dial the modem into a daemon server, so any program can access it using the client–server functions from Section 17.2.2.

2. Some form of locking must be used so that the abnormal termination of a program holding a lock automatically releases the lock. Ad hoc techniques, such as those still used by most versions of `cu` and UUCP, should finally be discarded, since better methods exist.

   We'll let the server daemon handle all the device locking. Since the client–server functions from Section 17.2.2 automatically notify the server when a client terminates, the daemon can release any locks that the process had.

3. New programs must be able to use all the features that we develop. A new program that deals with a modem should not have to reinvent the wheel. Dialing any type of modem should be as simple as a function call.

   For this feature, we'll let the central server daemon do all the dialing, passing back a file descriptor.

4. Client programs, such as `cu` and `tip`, shouldn't need special privileges. They should not be set-user-ID programs.

   We'll give the special privileges to the server daemon, allowing its clients to run without any special privileges.

Obviously we can't change the existing `cu`, `tip`, and UUCP programs, but we should make it easier for others to build on this work. Also, we should take the best features of the existing UNIX dialing programs.

Figure 1 shows the arrangement of the client and server.
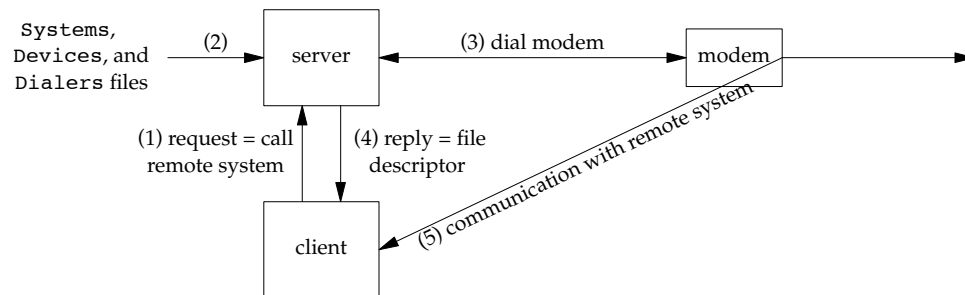
Figure 1   Overview of client and server.

The steps involved in establishing communications with a remote system are as follows:

0. The server is started.

1. The client is started and opens a connection to the server, using the `cli_conn` function (Section 17.2.2). The client sends a request for the server to call the remote system.

2. The server reads the `Systems`, `Devices`, and `Dialers` files to determine how to call the remote system. (We describe these files in the next section.) If a modem is being used, the `Dialers` file contains all the modem-specific commands to dial the modem.

3. The server `opens` the modem device and dials the modem. This can take a while (typically around 15−30 seconds). The server handles all locking of this device, to avoid interfering with other users of this device.

4. If the dialing was successful, the server passes back the open file descriptor for the modem device to the client. We use our functions from Section 17.4 to send and receive the descriptor.

5. The client communicates directly with the remote system. The server is not involved in this communication—the client `reads` and `writes` the file descriptor returned in step 4.

The communication between the client and server (steps 1 and 4) is across a full-duplex pipe (an *s-pipe*, a term we introduced in Section 17.4 to represent either a STREAMS pipe or a UNIX domain socket being used as a pipe). When the client is finished communicating with the remote system it closes this s-pipe (normally just by terminating). The server notices this close and releases the lock on the modem device.

4   Data Files

In this section we describe the three files used by the Honey DanBer UUCP system: `Systems`, `Devices`, and `Dialers`. There are many fields in these files that are used by the UUCP system. We don't describe these additional fields (or the UUCP system) in detail. Refer to Redman [1989] for additional details.

Figure 2 shows the six fields from the `Systems` file in a tabular format.

name	*time*	*type*	*class*	*phone*	*login*
host1	Any	ACU	19200	5551234	(not used)
host1	Any	ACU	9600	5552345	(not used)
host1	Any	ACU	2400	5556789	(not used)
modem	Any	modem	19200	–	(not used)
laser	Any	laser	19200	–	(not used)

Figure 2   The `Systems` file.

The *name* is the name of the remote system. We use this in commands of the form `cu host1`, for example. Note that we can have multiple entries for the same remote system. These entries are tried in the order in which they appear in the file. The entries named `modem` and `laser` are for connecting directly to a serial modem and a serial laser printer. We don't need to dial a modem to connect to these devices, but we still need to open the appropriate terminal line, and handle the appropriate locks.

time specifies the time-of-day and days of the week to call this host. This is a UUCP field. The *type* field specifies which entry in the `Devices` file is to be used for this *name*. The *class* field is really the line speed to be used (baud rate). *phone* specifies the phone number for entries with a *type* of ACU. For other entries the `phone` field is just a hyphen. The final field, *login*, is the remainder of the line. It is a series of strings used by UUCP to log in to the remote system. We don't need this field.

The `Devices` file contains information on the modems and directly connected hosts. Figure 3 shows the five fields in this file. The *type* field matches an entry in the `Systems` file with an entry in the `Devices` file. The *class* field must also match the corresponding field in the `Systems` file. It normally specifies the line speed.

The actual name of a device is obtained by prefixing the *line* field with `/dev/`. In this example the actual devices are `/dev/cua0`, `/dev/ttya`, and `/dev/ttyb`. The next field, *line2*, is not used.

type	*line*	*line2*	*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

Figure 3 The `Devices` file.

The final field, *dialer*, matches the corresponding entry in the `Dialers` file. For the directly connected entries this field is `direct`.

Figure 4 shows the format of the `Dialers` file. This is the file that contains all the modem-specific dialing commands.

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`

Figure 4 The `Dialers` file.

We show only two entries for this file—we don't show the entries for `tb1200` and `tb2400` that were referenced in the `Devices` file. The *handshake* field is contained on a single line. We have broken it into two lines to fit on the page.

The *dialer* field is used to locate the matching entry from the `Devices` file. The *sub* field specifies substitutions to be performed for an equals sign and a minus sign that appear in a phone number. In the two entries in Figure 4 this field says to substitute a `W` for an equals sign, and a comma for a minus sign. This allows the phone numbers in the `Systems` file to contain an equals sign (meaning "wait for dialtone") and a minus sign (meaning "pause"). The translation of these two characters to whatever each particular modem requires is specified by the `Dialers` file.

The final field, *handshake*, contains the actual dialing instructions. It is a sequence of blank-separated strings called expect–send strings. We expect (i.e., `read` until we match) the first string and then send (i.e., `write`) the next string. Let's look at the `tbfast` entry as an example. This entry is for a Telebit Trailblazer modem in its PEP (packetized ensemble protocol) mode.

1. The first expect string is empty, meaning "expect nothing." We always successfully match this empty string.

2. We send the next string. Special send sequences are specified with the backslash character. `\d` causes a delay for 2 seconds. We then send an `A`. We pause for one-half second (`\p`), send another `A`, pause, send another `A`, and pause again. We then send the remaining characters in the string, starting with `T`. These

commands all set parameters in the modem. The \r sends a carriage return and the final \c says not to write the normal newline at the end of the send string.

3. We read from the modem until we receive the string OK\r. (Again, the sequence \r means a carriage return.)

4. The next send string begins with \E. This enables echo checking: each time we send a character to the modem, we read back until the character is echoed. We then send the four characters ATDT. The next special character, \T, causes the phone number to be substituted. This is followed by a carriage return; the \c prevents the normal newline at the end of the send string from being sent.

5. The final expect string waits for CONNECT FAST to be returned by the modem. (The sequence \s means a single space.)

When this final expect string is received, the dialing is complete. (There are many more special sequences that can appear in the *handshake* string that we don't cover.)

Let's summarize the actions that we have to perform with these three files.

1. Using the name of the remote system, find the first entry in the Systems file with the same *name*.

2. Find the matching entry in the Devices file with a *type* and *class* that match the corresponding entries in the Systems file entry.

3. Find the entry in the Dialers file that matches the *dialer* field in the Devices file.

4. Dial the modem.

There are two reasons why this can fail: (1) the device corresponding to the *line* field in the Devices file is already in use by someone else or (2) the dialing is unsuccessful (e.g., the phone on the remote system is busy, or the remote system is down and is not answering the phone). The second case is often detected by a timeout occurring when we're reading from the modem, trying to match an expect string (see Exercise 10). In either case, we want to go back to step 1 and search for the next entry for the same remote system. As we saw in Figure 2, a given host can have multiple entries, each with a different phone number (and each phone number could correspond to a different device).

There are other files in the Honey DanBer system that we don't use in the example in this chapter. The file Dialcodes specifies dialcode abbreviations for phone numbers in the Systems file. The file Sysfiles allows the specification of alternate copies of the three files Systems, Devices, and Dialers.

5   Server Design

We'll start with a description of the server. Two factors affect the design of the server:

1. Dialing can take a while (15–30 seconds), so the server has to fork a child process to do the actual dialing.

2. The daemon server (the parent) has to be the one process that manages all the locks.
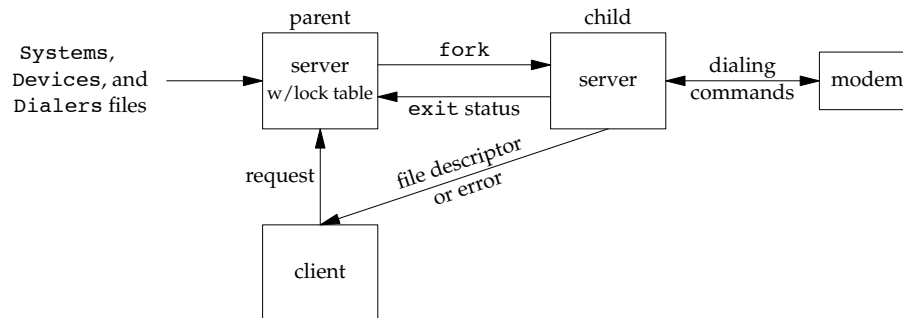
Figure 5 shows the arrangement of the processes.

Figure 5 Arrangement of processes in modem dialer.

The steps performed by the server are the following:

1. The parent receives the request from the client at the server's well-known name. As we described in Sections 17.2.2 and 17.3.2, this creates a unique s-pipe between the client and server. This parent process has to handle multiple clients at the same time, like the open server in Section 17.6.

2. Based on the name of the remote system that the client wants to contact, the parent goes through the `Systems` file and `Devices` file to find a match. The parent also keeps a lock table of which devices are currently in use, so it can skip those entries in the `Devices` file that are in use.

3. If a match is found, a child is `fork`ed to do the actual dialing. (The parent can handle other clients at this point.) If successful, the child sends the file descriptor for the modem back to the client on the client-specific s-pipe (which got duplicated across the `fork`) and calls `exit(0)`. If an error occurs (phone line busy, no answer, etc.), the child calls `exit(1)`.

4. The parent is notified of the child termination by `SIGCHLD` and fetches its termination status (`waitpid`).

   If the child was successful there is nothing more for the parent to do. The lock must be held until the client is finished with the modem device. The client-specific s-pipe between the client and parent is left open. This way, when the client does terminate, the parent is notified, and the parent releases the lock.

   If the child was not successful, the parent picks up in the `Systems` file where it left off for this client and tries to find another match. If another entry is found for the remote system, the parent goes back to step 3 and `fork`s a new child to do the actual dialing. If no more entries exist for the remote system, the parent calls `send_err` (Figure 17.19) and closes the client-specific s-pipe.

Having a unique connection to each client allows the child to send debug output back to the client, if desired. Often the client wants to see the progress of the actual dialing, if problems occur. Even though the dialing is being done by the child of an unrelated server, the unique connection allows the child to send output directly back to its client.

6   Server Source Code

We have 17 source files that constitute the server. Figure 6 lists the files containing the various functions and specifies which are used by the parent and the child. Figure 7 provides an overview of how the various functions are called.

| Source file	Parent/Child		Functions
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

Figure 6 Source files for server.
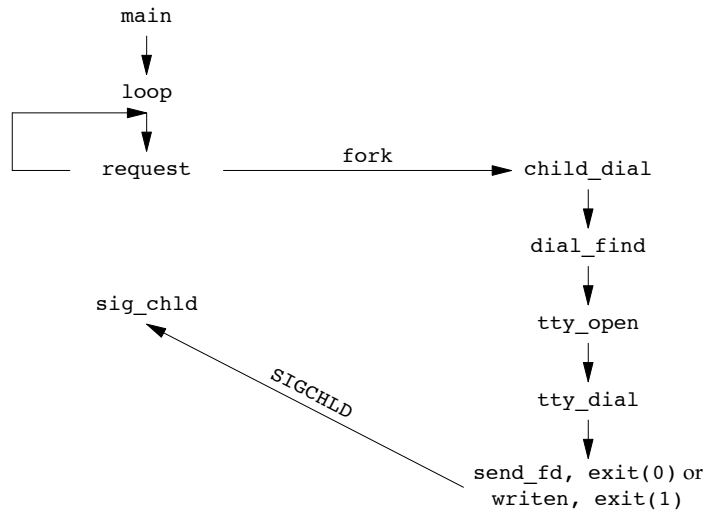
Figure 7 Overview of function calling in server.

Figure 8 shows the `calld.h` header, which is included by all the server source files. It includes the standard system headers, defines some basic constants, and declares the global variables.

```c
#include "apue.h"
#include <errno.h>

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

/*
 * Number of structures to alloc/realloc for Client structs
 * (client.c) and Lock structs (lock.c).
 */
#define NALLOC  10

#define WHITE       " \t\n"             /* for separating tokens */
#define SYSTEMS     "./Systems"         /* my own copies for now */
#define DEVICES     "./Devices"
#define DIALERS     "./Dialers"

extern int    clifd;
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;

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

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() */

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(unsigned 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 *, ...);
```

Figure 8 The calld.h header

We define a `Client` structure that contains all the information for each client. This is an expansion of the similar structure in Figure 17.35. In the time between `forking` a child to dial for a client and that child terminating, we can handle any number of other clients. This structure contains all the information that we need to try to find another `Systems` file entry for that client, and try dialing again.

We also define one structure for all the information for a single entry in the `Systems`, `Devices`, and `Dialers` files.

Figure 9 shows the `main` function for the server. Since this program is normally run as a daemon server, we provide a `-d` command line option that lets us run the program interactively.

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

int         clifd, log_to_stderr, client_size;
int         Debug;  /* Debug controlled by client, not cmd line */
char        errmsg[MAXLINE];
char        *speed, *sysname;
uid_t       uid;
Client      *client = NULL;
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 */
            log_to_stderr = 1;
            break;

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

    if (log_to_stderr == 0)
        daemonize("calld");

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

Figure 9 The `main` function

When the **-d** option is set, all the calls to the `log_XXX` functions (Appendix B) are sent to standard error. Otherwise they are logged using `syslog`.

The function `loop` is the main loop of the server (Figure 10). It multiplexes the various descriptors with the `select` function.

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

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

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

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 */
}
```

Figure 10 The `loop.c` file

The `loop` function initializes the `client` array and establishes a signal handler for `SIGCHLD`. We call `signal_intr` instead of `signal` so that any slow system call is interrupted when our signal handler returns. The `loop` function then calls `serv_listen` (Figures 17.10 and 17.15). The rest of the function is an infinite loop based on the `select` function, that tests for the following two conditions:

1. If a new client connection arrives, we call `serv_accept` (Figures 17.11 and 17.16). The function `client_add` creates an entry in the `client` array for the new client.

2. We then go through the `client` array, to see if (a) any client has terminated, or (b) any client requests have arrived.

   When a client terminates (whether voluntarily or not) its client-specific s-pipe to the server is closed, and we read an end of file from our end of the s-pipe. At this point we can release any device locks that the client owned and release the entry in the `client` array.

   When a request arrives from a client, we set things up and call `request`. (We showed the function `buf_args` in Figure 17.32.) If the name of the remote system is valid and if an available device entry is located, `request` forks a child process and returns.

The termination of a child is one external event that can happen at any time in this function. If we're blocked in the `select` function, it returns an error of `EINTR` in this case. Since the signal can also happen at other points in the `loop` function, we test the flag `chld_flag` each time through the loop before calling `select`. If the signal has occurred, we call the function `child_done` to process the termination.

This function goes through the `client` array, examining the `childdone` flag for each valid entry. If the child was successful, there's nothing else to do at this point. But if the child terminated with an `exit` status of 1, we call `request` to try to find another `Systems` file entry for this client.

Figure 11 shows the function `cli_args`, which is called by `buf_args` in the `loop` function when a client request arrives. It processes the command-line arguments from the client.

```
#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) {
            sprintf(errmsg, "missing <hostname> to call\n");
            return(-1);
        }
        sysname = argv[optind];       /* name of host to call */
        return(0);
}
```

Figure 11   The `cli_args` function

Note that this function sets global variables based on the command-line arguments, which `loop` then copies into the appropriate entry in the `client` array, since these options affect only a single client's request.

Figure 12 shows the file `client.c`, which defines the functions that manipulate the `client` array. The only difference between Figure 12 and Figure 17.36 is that we now have to look up an entry based on the process ID (in the function `client_sigchld`).

```
#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);
}
```

Figure 12 The `client.c` file

Figure 13 shows the file `lock.c`. The functions in this file manage the `lock` array for the parent. As with the `client` functions, we call `realloc` to allocate space dynamically for the `lock` array to avoid compile-time limits.

```
#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 */
}
```

Figure 13 Functions for managing client device locks

Each entry in the lock array is associated with a single *line* (the second field in the
Devices file). Since these locking functions don't know all the different *line* values in
this data file, new entries in the lock array are created whenever a new *line* is locked
the first time. The function find_line handles this.

The next three source files handle the three data files: Systems, Devices, and
Dialers. Each file has a XXX_next function that reads the next line of the file and
breaks it up into fields. The ISO C function strtok is called to break the lines into
fields. The functions in Figure 14 handle the Systems file.

```
#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");
}
```

Figure 14   Functions to read `Systems` file

The function `sys_next` is called by `request` to read the next entry in the file.

We have to remember our position in this file for each client (the `sysftell` member of the `Client` structure). This is so that if a child fails to dial the remote system, we can pick up where we left off in the `Systems` file (for that client), to try to find another entry for the remote system. The position is obtained by calling the standard I/O function `ftell` and reset using `fseek`.

Figure 15 contains the functions for reading the `Devices` file.

```
#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);
}
```

Figure 15 Functions for reading `Devices` file

We'll see that the `request` function calls `dev_find` to locate an entry with *type* and *class* fields that match an entry in the `Systems` file.

Figure 16 contains the functions for reading the `Dialers` file.

```
#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);
}
```

Figure 16 Functions for reading `Dialers` file

We'll see that the `child_dial` function calls `dial_find` to find an entry with a *dialer* field that matches a particular device.

Note from Figure 6 that the `Systems` and `Devices` files are handled by the parent, while the `Dialers` file is handled by the child. This was one of the design goals—the parent finds a matching device that is not locked and `fork`s a child to do the actual dialing.

We look at the `request` function in Figure 17. It was called by the `loop` function to try to locate an unlocked device for the specified remote host. To do this it goes through the `Systems` file, then the `Devices` file. If a match is found, a child is `fork`ed. We allow the client to specify a speed, in addition to the name of the remote system. For example, with the `Systems` file in Figure 2, the client's request can look like

```
call -s 9600 host1
```

which causes us to ignore the other two entries for `host1` in Figure 2.

Note that we can't record the device lock using `lock_set` until we know the process ID of the child (i.e., after the `fork`), but we have to test whether the device is locked before the `fork`. Since we don't want the child starting until we have set the lock, we use the `TELL_WAIT` functions (Figure 10.24) to synchronize the parent and child. Also note that although the test `is_locked` and the actual setting of the lock by `set_lock` are two separate operations (i.e., not a single atomic operation), we do not have a race condition. This is because `request` is called only by the single parent server daemon—it is not called by multiple processes.

If `request` returns 0, a child was `forked` to start the dial; otherwise it returns –1 to indicate that either the name of the remote system wasn't valid or all the possible devices for the remote system were locked.

```
#include    "calld.h"

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

    /*
     * Position where this client left off last (or rewind).
     */
    errmsg[0] = 0;
    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 */
}
```

Figure 17 The request function

The last of the parent-specific functions is `sig_chld`, the signal handler for the SIGCHLD signal. This is shown in Figure 18.

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

/*
 * SIGCHLD handler, invoked when a child terminates.
 * Probably interrupts accept() in serv_accept().
 */
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 childdone status for loop */
        client_sigchld(pid, WEXITSTATUS(stat)+1);
    else
        log_msg("child %d terminated abnormally: %04x", pid, stat);

    errno = errno_save;
}
```

Figure 18 The sig_chld signal handler

When a child terminates we must record its termination status and process ID in the appropriate entry in the `client` array. We call the function `client_sigchld` (Figure 12) to do this.

Note that we are violating one of our earlier rules from Chapter 10—a signal handler should only set a global variable and nothing else. Here we call `waitpid` and the function `client_sigchld` (Figure 12). This latter function is signal safe. All it does is record information in an entry in the `client` array—it doesn't create or delete entries (which would be nonreentrant) and it doesn't call any system functions.

`waitpid` is defined by POSIX.1 to be signal safe (Figure 10.4). If we didn't call `waitpid` from the signal handler, the parent would have to call it when the flag `chld_flag` was nonzero. But since numerous children can terminate before the main loop gets a chance to look at `chld_flag`, we would either need to increment `chld_flag` each time a child terminated (so the main loop would know how many times to call `waitpid`) or call `waitpid` in a loop, with the `WNOHANG` flag (Figure 8.7). The simplest solution is to call `waitpid` from the signal handler, and record the information in the `client` array.

We now proceed to the functions that are called by the child as part of its attempt to dial the remote system. Everything starts for the child after the `fork` when `request` calls `child_dial` (Figure 19).

```
#include    "calld.h"

/*
 * The child does the actual dialing and sends the fd back to
 * the client.  This function can't return to caller; it 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 */
}
```

Figure 19   The child_dial function

If the device being used is directly connected, just the function tty_open is called to
open the terminal device and set all the appropriate terminal parameters. But if the
device is a modem, three functions are called: dial_find (to locate the appropriate
entry in the Dialers file), tty_open, and tty_dial (to do the actual dialing).

   If child_dial is successful, it returns the file descriptor to the client by calling
send_fd (Figures 17.20 and 17.22) and calls exit(0). Otherwise it sends an error
message back to the client across the s-pipe and calls exit(1). The client-specific
s-pipe is duplicated across the fork, so the child can send either the descriptor or error
message directly back to the client.

```
#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");
}
```

Figure 20 Debugging functions

The client can send a −d option in its command to the server, and this sets the client-specific variable Debug. This flag is used in Figure 20 by the two functions DEBUG and DEBUG_NONL to send debugging information back to the client. This information is useful when dialing problems are encountered for a particular system. These two functions are called predominantly by the child, although the parent also calls them from the request function (Figure 17).

Figure 21 shows the tty_open function. It is called for both modem devices and direct connect devices to open the terminal and set its modes. The *class* field of the Systems and Devices file specified the line speed, and the client can specify the parity.

```
#include    "calld.h"
#include    <fcntl.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;

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);
        }
```

Figure 21   The tty_open function

We open the terminal device with the nonblocking flag, as sometimes the open of a terminal connected to a modem doesn't return until the modem's carrier is present. Since we are dialing out and not dialing in, we don't want to wait. At the end of the function we call the clr_fl function to clear the nonblocking mode. The only difference between a modem and a direct connect line in the tty_open function is that we set the CLOCAL bit for a direct connect line.

The details of dialing a modem takes place in the tty_dial function (Figure 22). This function is only called for modem lines, not for direct connect lines.

```
#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 */
    }
}
```

Figure 22 The `tty_dial` function

The `tty_dial` function just calls one function to handle the expect string and another
to handle the send string. We are done when there are no more send or expect strings.
(Note that we do not handle the *sub* string from Figure 4.)

   Figure 23 shows the function `send_str` that outputs the send strings. To keep the
size of this example manageable, we have not implemented every special escape
sequence—we have implemented enough to use the program with the `Dialers` file
shown in Figure 4.

```
#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 14.6 */
            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);
}
```

Figure 23 The `send_str` function

`send_str` calls the function `ctl_str` to convert ASCII control characters into a printable version. Figure 24 shows the `ctl_str` function.

```
#include    "calld.h"

/*
 * Make a printable string of the character "c", which may be a
 * control character.  Works only with ASCII.
 */
char *
ctl_str(unsigned 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);
}
```

Figure 24 The `ctl_str` function

The hardest part of dialing the modem is recognizing the expect strings. Figure 25 shows the function `expect_str` that does this. (As with the send strings, we have implemented only a subset of all the possible features provided by the `Dialers` file.)

```
#include    "calld.h"

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

static int      expalarm = EXPALRM;
static volatile sig_atomic_t    caught_alrm;

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");
    else
        log_sys("read error");
    return(-1);
}

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

int                     /* return 0 if got it, -1 if not */
expect_str(int fd, char *ptr)
{
    char    expstr[MAXLINE], inbuf[MAXLINE];
    char    c;
    char    *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);
}
```

Figure 25 Functions to read and recognize expect strings

We first copy the expect string, converting the special characters. Our matching technique is to read characters from the modem until the character matches the first character of the expect string. We then read enough characters to equal the number of characters in the expect string. From that point we continually read characters from the modem into the buffer, comparing them against the expect string, until we have a match or until the alarm goes off. (There are better algorithms for string matching—ours was chosen to simplify the coding. The number of characters returned by the modem that are compared to the expect string is usually on the order of 50, and the size of the expect string is often around 10−20 characters.)

Note that we have to set an alarm each time we try to match an expect string, as the alarm is the only way we can determine that we didn't receive what we were waiting for.

This completes the server daemon. All it does is open a terminal device and dial a modem. What happens with the terminal device after it is opened depends on the client. We'll now examine a client that provides an interface similar to cu and tip, allowing us to dial a remote system and log in.

7   Client Design

The interface between the client and server is only about a dozen lines of code. The client formats a command line, sends it to the server, and receives back either a file descriptor or an error indication. The rest of the client design depends on what the client wants to do with the returned descriptor. In this section we'll outline the design of the `call` client that works like the familiar cu and `tip` programs. It allows us to call a remote system and log in to it. The remote system need not be a UNIX system—we can use it to communicate with any system or device that's connected to the host with an RS-232 serial connection.

Terminal Line Disciplines

In Figures 14.20 and 14.21 we gave an overview of the `telnet` program, which has a similar structure to the modem dialer. Figure 26 is an expansion of Figure 14.20, recognizing that there are two line disciplines between the user and the modem and assuming that we're using the program to dial into a remote UNIX host. (Recall from the output of Figure 14.18 that for a STREAMS-based terminal system, Figure 26 is a simplification. There may be multiple STREAMS modules making up the line discipline and multiple modules making up the terminal device driver. We also don't explicitly show the stream head.)

The two dotted boxes in Figure 26 above the modem on the local system were established by the server's `tty_open` function (Figure 21). That function set the dotted terminal line discipline module to noncanonical (i.e., raw) mode. The modem on the local system was dialed by the server's `tty_dial` function (Figure 22). The two arrows between the dotted terminal line discipline box and the `call` process correspond to the descriptor returned by the server. (We show the single descriptor as two arrows, to reiterate the fact that it's a full-duplex descriptor.)
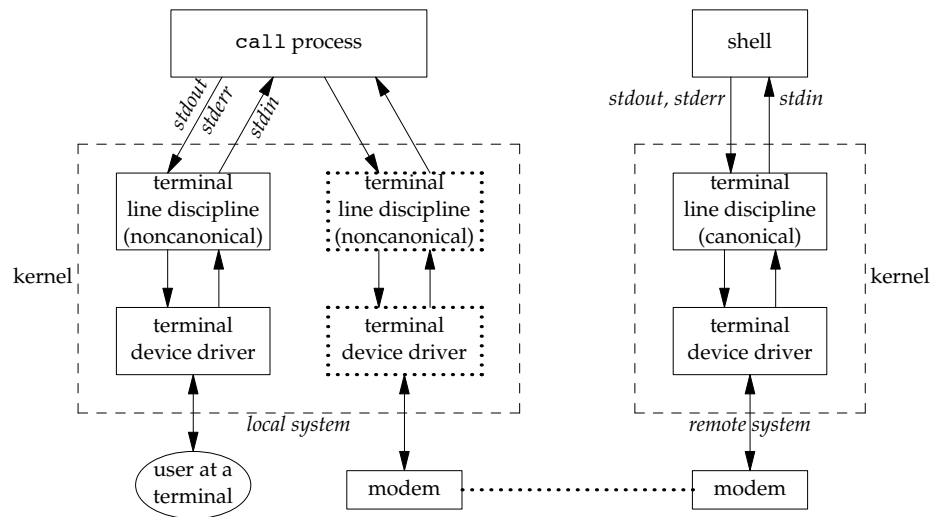
Figure 26 Overview of modem dialer process to log in to remote UNIX host.

The line discipline box beneath the shell on the remote system is set by the login process on that system to be in the canonical mode. After we have dialed the remote system we want the special terminal input characters (end of file, erase line, etc. from Section 18.3) recognized by the line discipline module on the remote host. That means we have to set the mode of the line discipline module above the terminal (standard input, standard output, and standard error of the `call` process) to noncanonical mode.

One Process or Two?

In Figure 26 we show the `call` process as a single process. Doing so requires support for an I/O multiplexing function such as `select` or `poll`, since two descriptors are being read from and two descriptors are being written to. We could also design the client as two processes, a parent and a child, as we showed in Figure 14.21. Figure 27 shows only these two processes and the line disciplines beneath them. Historically, `cu` and `tip` have always been two processes, as in Figure 27. This is because early UNIX systems didn't support an I/O multiplexing function.

We choose to use a single process for the following two reasons.

1. Having two processes complicates the termination of the client. If we terminate the connection by entering `˜.` (a tilde followed by a period) at the beginning of a line, the child recognizes this and terminates. The parent then has to catch the `SIGCHLD` signal so that the parent can terminate too.

   If the connection is terminated by the remote system or if the line is dropped, the parent will detect this by reading an end of file from the modem descriptor. The parent then has to notify the child, so that the child can also terminate.
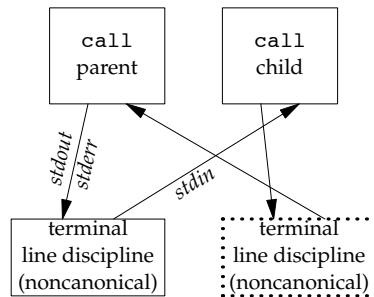
Figure 27 The `call` client as two processes.

Using a single process obviates the need for one process notifying the other when it terminates.

2. We are going to implement a file transfer function in the client, similar to the `put` and `take` commands of `cu` and `tip`. We enter these commands on the standard input, on a line that begins with a tilde (the default escape character). These commands are recognized by the child if two processes are being used (Figure 27). But the file that's received by the client, in the case of a `take` command, comes across the modem descriptor, which is being `read` by the parent. This means that to implement the `take` command, the child has to notify the parent so that the parent stops reading from the modem. The parent is probably blocked in a `read` on this descriptor, so a signal is required to interrupt the parent's `read`. When the child is done, another notification is required to tell the parent to resume reading from the modem. While possible, this scenario quickly becomes messy.

A single process simplifies the entire client. By using a single process, however, we lose the ability to job-control stop only the child. The BSD `tip` program supports this feature. It allows us to stop the child while the parent continues running. This means all the terminal input is directed back to our shell instead of the child, letting us work on the local system, but we'll still see any output generated by the remote system. This is handy if we start a long running job on the remote system and want to see any output that it generates, while working on the local system.

We now look at the source code to implement the client.

8   Client Source Code

The client is smaller than the server, since the client doesn't handle all the details of connecting the remote system—the server from Section 6 handles this. About one half of the client is needed to handle commands such as `take` and `put`.

Figure 28 shows the `call.h` header that is included by all the client source files.

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

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

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

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 **);
```

Figure 28 The `call.h` header

The command for the server and the server's well-known name have to correspond to the values in Figure 8.

Figure 29 shows the `main` function.

```
#include    "call.h"

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);
}
```

Figure 29   The `main` function

The `main` function processes the command-line arguments, saving them in the array `args`, which is sent to the server. The function `call` contacts the server and returns the file descriptor to the remote system.

The line discipline module above the terminal (Figure 26) is set to noncanonical mode using the `tty_raw` function (Figure 18.20). To reset the terminal when we're done we establish the function `tty_atexit` as an exit handler.

The function `loop` is then called to copy everything that we type at the terminal to the modem and everything from the modem to the terminal.

The `call` function in Figure 30 contacts the server to obtain a file descriptor for the modem. As we said earlier, it takes only a dozen lines of code to contact the server to obtain the descriptor.

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

    /*
     * Read back the descriptor.  Returned errors are handled
     * by write().
     */
    return(recv_fd(csfd, write));
}
```

Figure 30 The `call` function

The function `loop` handles the I/O multiplexing between the two input streams and the two output streams. We can use either `poll` or `select`, depending what the local system provides. Figure 31 shows an implementation using `poll`.

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

#define BUFFSIZE    512

/*
 * Copy everything from stdin to "remfd",
 * and everything from "remfd" to stdout.
 */
void
loop(int remfd)
```

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

    /*
     * Set stdout unbuffered for printfs in take() and put().
     */
    setbuf(stdout, NULL);
    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, -1) <= 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 wasn't special, */
                /* so return and echo it 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 */
    }
}
```

Figure 31 The loop function using the poll function

The basic loop of this function just waits for data to appear from either the terminal or the modem. When data is `read` from the terminal, it's just copied to the modem and vice versa. The only complication is to recognize the escape character (the tilde) as the first character of a line.

Note that we `read` one character at a time from the terminal (standard input), but up to one buffer at a time from the modem. One reason for the single character at a time from the terminal is because we have to look at every character to know when a new line begins, to recognize the special commands. Although this character-at-a-time I/O is expensive in terms of CPU time (recall Figure 3.5), there is usually far less input from the terminal than from the remote system. (In remote login sessions using this program measured by the author, there are around 100 characters output by the remote host for every character input.)

When the escape character is seen, `doescape` is called to process the command (Figure 32). We support only five commands. Simple commands are handled directly in this function, while the more complicated `take` and `put` commands are handled by separate functions (`take` and `put`).

- A period terminates the client. For some devices, such as a laser printer, this is the only way to terminate the client. When we're logged into a remote system, such as in Figure 26, logging out from that system usually causes the remote modem to drop the phone line, causing a hangup to be received on the modem descriptor by the `loop` function.

- If the system supports job control we recognize the job-control suspend character and suspend the client. Note that it is simpler for us to recognize this character directly and stop ourselves than to have the line discipline recognize the character and generate the `SIGSTOP` signal (compare with Figure 10.30). We have to reset the terminal mode before stopping ourselves, and reset it when we're continued.

- A pound sign generates a BREAK on the modem descriptor. We use the POSIX.1 `tcsendbreak` function to do this (Section 18.8). The BREAK condition often causes the remote system's `getty` or `ttymon` program to switch line speeds (Section 9.2).

- The `take` and `put` commands require separate functions to be called. The way to distinguish between the two commands is to remember that the command describes what the client is doing on the local system: taking a file from the remote system or putting a file to the remote system.

```
#include    "call.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 */
}
```

Figure 32   The doescape function

Figure 33 shows the code required to handle the `take` command. The function
`take` first calls `prompt_read` (which we show in Figure 34) to echo `˜[take]` in
response to the `˜t` command. The `prompt_read` function then reads a line of input
containing the source pathname (the file on the remote host) and the destination
pathname (the file on the local host) from the terminal. The results are stored in the
global variables `src` and `dst`.

```
#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;
    char    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);
    }
```

Figure 33   Processing the `take` command

After the `take` function opens the local file for writing it sends the following command to the remote host:

cat *sourcefile* ; echo ˆA

This causes the remote host to execute the `cat` command, followed by an echo of the ASCII Control-A character. We look for this Control-A in all the characters that are returned by the remote host, and when we encounter it, we know the file transfer is complete. Note that we also have to read back the echo of the command line that we send to the remote host. Only after that echo do we start receiving the output of the `cat` command.

   While we're reading the remote file we look for newline characters and count the lines returned. We display these at the left margin, overwriting each line number with the next (since we terminate the line in the `printf` with a carriage return only and not a newline). This provides a visual display on the terminal of the progress of the file transfer and a final line count at the end.

   This source file also contains the function `rem_read`, which is called to read each character from the remote host. We `read` up to one buffer at a time, but return only one character at a time to the caller.

   Originally the `take` command was written to `read` one character at a time, similar to what `cu` and `tip` have historically done. This was OK 23 years ago, when 1200 baud modems were considered fast. But as modems became faster, delivering characters to the terminal device driver at 38400 baud and above, characters could get lost.

   The solution to this problem is to code the `rem_read` function to read up to a buffer at a time. Doing this reduces the system CPU time and provides more reliable transfer.

   Note that the number of bytes returned by `read` can be greater than 1 even though the line discipline module for the modem has its MIN set to 1 and TIME set to 0 by the `tty_open` function (Figure 21). This is case B from Section 18.11. This illustrates that MIN is only a minimum. If we ask for more than the minimum, and the bytes are ready to be read, they're returned. We are not restricted to character-at-a-time input when we set MIN to 1.

   Figure 34 shows the two ancillary functions `take_put_args` and `prompt_read`. The latter is called from both the `take` and `put` functions, with the former as an argument (that is then called by the `buf_args` function, Figure 17.32).

```
#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);
}

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

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

Figure 34 The `take_put_args` and `prompt_read` functions

The function `prompt_read` reads a line of input from the terminal, and then calls `buf_args` to split the line into a standard argument list that is processed by `take_put_args`. Note that the terminal is reset to canonical mode to read the arguments, allowing the use of the standard editing characters while entering the line.

The final client function is `put`, shown in Figure 35. It is called to copy a local file to the remote host.

```
#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);
}
```

Figure 35   The put function

As with the `take` command, we send a command string to the remote system. This time the command is

        stty -echo; cat > *destfile* ; stty echo

We have to turn echo off, otherwise the entire file would also be sent back to us. To terminate the `cat` command we send the end-of-file character (often Control-D). This requires that the same end-of-file character be used on both the local system and the remote system. Additionally, the file cannot contain the ERASE or KILL characters in use on the remote system.

9   Summary

In this chapter we've looked at two different programs: a daemon server that dials a modem and a remote login program that uses the server to contact a remote system that's connected through a terminal port. The server can be used by other programs that need to contact remote systems or hardware devices connected through asynchronous terminal ports.

   The design of the server was similar to the open server in Section 17.6 and required the use of full-duplex pipes, unique per-client connections to the server, and the passing of file descriptors. These advanced IPC features allow us to build client–server applications with many desirable features, as described in Section 3.

   The client is similar to the `cu` and `tip` programs provided by many UNIX systems, but in our example we didn't have to worry about dialing a modem, interfering with UUCP lock files, setting the characteristics of the modem's line discipline module, and the like. The server handles all these details. It let us concentrate on the real issues of the client, such as providing a reliable file transfer mechanism.

10   References

Presotto, D. L., andRitchie, D. M. 1990. "Interprocess Communication in the Ninth Edition UNIX System," *Software Practice and Experience*, vol. 20, no. S1, pp. S1/3-S1/17 (June).

Redman, B. E. 1989. "UUCP UNIX-to-UNIX Copy," *Unix Networking*, ed. S. G. Kochan, andP. H. Wood, pp. 5-48. Howard W. Sams and Company, Indianapolis, IN.

Exercises

1   How can we avoid step 0 (starting the server by hand) in Section 3?

2   What happens if we don't set `optind` to 1 in Figure 11?

3   What happens if someone edits the `Systems` file between the time `request` (Figure 17) `forks` a child and the time the child terminates with a status of 1?

4   In Section 7.8 we said to be careful any time we use pointers into a region that gets `realloced`, since the region can move around in memory on each call to `realloc`. Why can we use the pointer `cliptr` in Figure 10 when the `client` array is manipulated by `realloc`?

5   What happens if either of the pathname arguments to the `take` and `put` commands contain a semicolon?

6   Modify the server to read its three data files once when it starts, storing them in memory. If the files are modified, how should the server handle this?

7   In Figure 30 why do we cast the argument `args` when filling in the structure for `writev`?

8   Implement Figure 31 using `select` instead of `poll`.

9   How can you verify that the file being sent with the `put` command does not contain characters that will be interpreted by the line discipline on the remote system?

10 The faster the dialing function recognizes that a dial has failed, the faster it can proceed to the next possible entry in the `Systems` file. For example, if we can determine that the remote phone is busy and terminate before the timer in `expect_str` expires, we can save 15 or 20 seconds. To handle these types of errors, the 4.3BSD UUCP expect–send strings allow an expect string of `ABORT`, followed by a string that if matched, aborts the current dial. For example, right before the final expect string `CONNECT\sFAST` in Figure 4 we would like to add

```
ABORT BUSY
```

Implement this feature.