13.4 daemon_init Function

Figure 13.4 shows a function named daemon_init that we can call (normally from a server) to daemonize the process. This function should be suitable for use on all variants of Unix, but some offer a C library function called daemon that provides similar features. BSD offers the daemon function, as does Linux.

Figure 13.4 daemon_init function: daemonizes the process.

daemon_init.c

```
 1 #include    "unp.h"
 2 #include    <syslog.h>

 3 #define MAXFD   64

 4 extern int daemon_proc;         /* defined in error.c */

 5 int
 6 daemon_init(const char *pname, int facility)
 7 {
 8     int     i;
 9     pid_t     pid;

10     if ( (pid = Fork()) < 0)
11         return (-1);
12     else if (pid)
13         _exit(0);               /* parent terminates */

14     /* child 1 continues... */

15     if (setsid() < 0)           /* become session leader */
16         return (-1);

17     Signal(SIGHUP, SIG_IGN);
18     if ( (pid = Fork()) < 0)
19         return (-1);
20     else if (pid)
21         _exit(0);               /* child 1 terminates */

22     /* child 2 continues... */

23     daemon_proc = 1;            /* for err_XXX() functions */

24     chdir("/");                 /* change working directory */

25     /* close off file descriptors */
26     for (i = 0; i < MAXFD; i++)
27         close(i);

28     /* redirect stdin, stdout, and stderr to /dev/null */
29     open("/dev/null", O_RDONLY);
30     open("/dev/null", O_RDWR);
31     open("/dev/null", O_RDWR);

32     openlog(pname, LOG_PID, facility);

33     return (0);                 /* success */
34 }
```

fork

10–13 We first call fork and then the parent terminates, and the child continues. If the process was started as a shell command in the foreground, when the parent terminates, the shell thinks the command is done. This automatically runs the child process in the background. Also, the child inherits the process group ID from the parent but gets its own process ID. This guarantees that the child is not a process group leader, which is required for the next call to setsid.

setsid

15–16 setsid is a POSIX function that creates a new session. (Chapter 9 of APUE talks about process relationships and sessions in detail.) The process becomes the session leader of the new session, becomes the process group leader of a new process group, and has no

controlling terminal.

Ignore `SIGHUP` and `Fork` Again

17–21 We ignore `SIGHUP` and call `fork` again. When this function returns, the parent is really the first child and it terminates, leaving the second child running. The purpose of this second `fork` is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. When a session leader without a controlling terminal opens a terminal device (that is not currently some other session's controlling terminal), the terminal becomes the controlling terminal of the session leader. But by calling `fork` a second time, we guarantee that the second child is no longer a session leader, so it cannot acquire a controlling terminal. We must ignore `SIGHUP` because when the session leader terminates (the first child), all processes in the session (our second child) receive the `SIGHUP` signal.

Set Flag for Error Functions

23 We set the global `daemon_proc` to nonzero. This external is defined by our `err_XXX` functions (Section D.3), and when its value is nonzero, this tells them to call `syslog` instead of doing an `fprintf` to standard error. This saves us from having to go through all our code and call one of our error functions if the server is not being run as a daemon (i.e., when we are testing the server), but call `syslog` if it is being run as a daemon.

Change Working Directory

24 We change the working directory to the root directory, although some daemons might have a reason to change to some other directory. For example, a printer daemon might change to the printer's spool directory, where it does all its work. Should the daemon ever generate a `core` file, that file is generated in the current working directory. Another reason to change the working directory is that the daemon could have been started in any filesystem, and if it remains there, that filesystem cannot be unmounted (at least not without using some potentially destructive, forceful measures).

Close any open descriptors

25–27 We close any open descriptors that are inherited from the process that executed the daemon (normally a shell). The problem is determining the highest descriptor in use: There is no Unix function that provides this value. There are ways to determine the maximum number of descriptors that the process can open, but even this gets complicated (see p. 43 of APUE) because the limit can be infinite. Our solution is to close the first 64 descriptors, even though most of these are probably not open.

> Solaris provides a function called `closefrom` for use by daemons to solve this problem.

Redirect `stdin`, `stdout`, and `stderr` to `/dev/null`

29–31 We `open` `/dev/null` for standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (EOF), and the kernel just discards anything written to them. The reason for opening these descriptors is so that any library function called by the daemon that assumes it can read from standard input or write to either standard output or standard error will not fail. Such a failure is potentially dangerous. If the daemon ends up opening a socket to a client, that socket descriptor ends up as `stdout` or `stderr` and some erroneous call to something like `perror` then sends unexpected data to a client.

Use `syslogd` for Errors

32 `Openlog` is called. The first argument is from the caller and is normally the name of the program (e.g., `argv[0]`). We specify that the process ID should be added to each log message. The *facility* is also specified by the caller, as one of the values from Figure 13.2 or 0 if the default of `LOG_USER` is acceptable.

We note that since a daemon runs without a controlling terminal, it should never receive the `SIGHUP` signal from the kernel. Therefore, many daemons use this signal as a notification from the administrator that the daemon's configuration file has changed, and the daemon should reread the file. Two other signals that a daemon should never receive are `SIGINT` and `SIGWINCH`, so daemons can safely use these signals as another way for administrators to indicate some change that the daemon should react to.

Example: Daytime Server as a Daemon

Figure 13.5 is a modification of our protocol-independent daytime server from Figure 11.14 that calls our `daemon_init` function to run as daemons.

There are only two changes: We call our `daemon_init` function as soon as the program starts, and we call our `err_msg` function, instead of `printf`, to print the client's IP address and port. Indeed, if we want our programs to be able to run as a daemon, we must avoid calling the `printf` and `fprintf` functions and use our `err_msg` function instead.

Note how we check `argc` and issue the appropriate usage message *before* calling `daemon_init`. This allows the user starting the daemon to get immediate feedback if the command has the incorrect number of arguments. After calling `daemon_init`, all subsequent error messages go to syslog.

If we run this program on our Linux host `linux` and then check the `/var/log/messages` file (where we send all `LOG_USER` messages) after connecting from the same machine (e.g., localhost), we have

```
Jun 10 09:54:37 linux daytimetcpsrv2[24288]:
connection from 127.0.0.1.55862
```

(We have wrapped the one long line.) The date, time, and hostname are prefixed automatically by the `syslogd` daemon.

Figure 13.5 Protocol-independent daytime server that runs as a daemon.

inetd/daytimetcpsrv2.c

```
 1 #include     "unp.h"
 2 #include     <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t addrlen, len;
 8     struct sockaddr *cliaddr;
 9     char     buff[MAXLINE];
10     time_t ticks;

11     if (argc < 2 || argc > 3)
12         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

13     daemon_init(argv[0], 0);

14     if (argc == 2)
15         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
16     else
17         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);

18     cliaddr = Malloc(addrlen);

19     for ( ; ; ) {
20         len = addrlen;
21         connfd = Accept(listenfd, cliaddr, &len);
22         err_msg("connection from %s", Sock_ntop(cliaddr, len));

23         ticks = time(NULL);
24         snprintf(buff, sizeof(buff), "%.24s/r/n", ctime(&ticks));
25         Write(connfd, buff, strlen(buff));

26         Close(connfd);
27     }
28 }
```