13.1 Introduction

A *daemon* is a process that runs in the background and is not associated with a controlling terminal. Unix systems typically have many processes that are daemons (on the order of 20 to 50), running in the background, performing different administrative tasks.

The lack of a controlling terminal is typically a side effect of being started by a system initialization script (e.g., at boot-time). But if a daemon is started by a user typing to a shell prompt, it is important for the daemon to disassociate itself from the controlling terminal to avoid any unwanted interraction with job control, terminal session management, or simply to avoid unexpected output to the terminal from the daemon as it runs in the background.

There are numerous ways to start a daemon:

1. During system startup, many daemons are started by the system initialization scripts. These scripts are often in the directory `/etc` or in a directory whose name begins with `/etc/rc`, but their location and contents are implementation-dependent. Daemons started by these scripts begin with superuser privileges.

   A few network servers are often started from these scripts: the `inetd` superserver (covered later in this chapter), a Web server, and a mail server (often `sendmail`). The `syslogd` daemon that we will describe in Section 13.2 is normally started by one of these scripts.

2. Many network servers are started by the `inetd` superserver. `inetd` itself is started from one of the scripts in Step 1. `inetd` listens for network requests (Telnet, FTP, etc.), and when a request arrives, it invokes the actual server (Telnet server, FTP server, etc.).

3. The execution of programs on a regular basis is performed by the `cron` daemon, and programs that it invokes run as daemons. The `cron` daemon itself is started in Step 1 during system startup.

4. The execution of a program at one time in the future is specified by the `at` command. The `cron` daemon normally initiates these programs when their time arrives, so these programs run as daemons.

5. Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.

Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages or emergency messages that need to be handled by an administrator. The `syslog` function is the standard way to output these messages, and it sends the messages to the `syslogd` daemon.

13.2 syslogd Daemon

Unix systems normally start a daemon named `syslogd` from one of the system initializations scripts, and it runs as long as the system is up. Berkeley-derived implementations of `syslogd` perform the following actions on startup:

1. The configuration file, normally `/etc/syslog.conf`, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file `/dev/console`, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the `syslogd` daemon on another host.
2. A Unix domain socket is created and bound to the pathname `/var/run/log` (`/dev/log` on some systems).
3. A UDP socket is created and bound to port 514 (the `syslog` service).
4. The pathname `/dev/klog` is opened. Any error messages from within the kernel appear as input on this device.

The `syslogd` daemon runs in an infinite loop that calls `select`, waiting for any one of its three descriptors (from Steps 2, 3, and 4) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the `SIGHUP` signal, it rereads its configuration file.

We could send log messages to the `syslogd` daemon from our daemons by creating a Unix domain datagram socket and sending our messages to the pathname that the daemon has bound, but an easier interface is the `syslog` function that we will describe in the next section. Alternately, we could create a UDP socket and send our log messages to the loopback address and port 514.

> Newer implementations disable the creation of the UDP socket, unless specified by the administrator, as allowing anyone to send UDP datagrams to this port opens the system up to denial-of-service attacks, where someone could fill up the filesystem (e.g., by filling up log files) or cause log messages to be dropped (e.g., by overflowing `syslog's` socket receive buffer).
>
> Differences exist between the various implementations of `syslogd`. For example, Unix domain sockets are used by Berkeley-derived implementations, but System V implementations use a `STREAMS` log driver. Different Berkeley-derived implementations use different pathnames for the Unix domain socket. We can ignore all these details if we use the `syslog` function.

13.3 `syslog` Function

Since a daemon does not have a controlling terminal, it cannot just `fprintf` to `stderr`. The common technique for logging messages from a daemon is to call the `syslog` function.

```
#include <syslog.h>

void syslog(int priority, const char *message, ... );
```

Although this function was originally developed for BSD systems, it is provided by virtually all Unix vendors today. The description of `syslog` in the POSIX specification is consistent with what we describe here. RFC 3164 provides documentation of the BSD syslog protocol.

The *priority* argument is a combination of a *level* and a *facility*, which we show in Figures 13.1 and 13.2. Additional detail on the *priority* may be found in RFC 3164. The *message* is like a format string to `printf`, with the addition of a `%m` specification, which is replaced with the error message corresponding to the current value of `errno`. A newline can appear at the end of the *message*, but is not mandatory.

Figure 13.1. *level* of log messages.

level	Value	Description
LOG_EMERG	0	System is unusable (highest priority)
LOG_ALERT	1	Action must be taken immediately
LOG_CRIT	2	Critical conditions
LOG_ERR	3	Error conditions
LOG_WARNING	4	Warning conditions
LOG_NOTICE	5	Normal but significant condition (default)
LOG_INFO	6	Informational
LOG_DEBUG	7	Debug-level messages (lowest priority)

Figure 13.2. *facility* of log messages.

facility	Description
LOG_AUTH	Security/authorization messages
LOG_AUTHPRIV	Security/authorization messages (private)
LOG_CRON	`cron` daemon
LOG_DAEMON	System daemons
LOG_FTP	FTP daemon
LOG_KERN	Kernel messages
LOG_LOCAL0	Local use
LOG_LOCAL1	Local use
LOG_LOCAL2	Local use
LOG_LOCAL3	Local use
LOG_LOCAL4	Local use
LOG_LOCAL5	Local use
LOG_LOCAL6	Local use
LOG_LOCAL7	Local use
LOG_LPR	Line printer system
LOG_MAIL	Mail system
LOG_NEWS	Network news system
LOG_SYSLOG	Messages generated internally by `syslogd`
LOG_USER	Random user-level messages (default)
LOG_UUCP	UUCP system

Log messages have a *level* between 0 and 7, which we show in Figure 13.1. These are ordered values. If no *level* is specified by the sender, `LOG_NOTICE` is the default.

Log messages also contain a *facility* to identify the type of process sending the message. We show the different values in Figure 13.2. If no *facility* is specified, `LOG_USER` is the default.

For example, the following call could be issued by a daemon when a call to the `rename` function unexpectedly fails:

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

The purpose of *facility* and *level* is to allow all messages from a given facility to be handled the same in the `/etc/syslog.conf` file, or to allow all messages of a given level to be handled the same. For example, the configuration file could contain the lines

```
kern.*          /dev/console
local7.debug    /var/log/cisco.log
```

to specify that all kernel messages get logged to the console and all `debug` messages from the `local7` facility get appended to the file `/var/log/cisco.log`.

When the application calls `syslog` the first time, it creates a Unix domain datagram socket and then calls `connect` to the well-known pathname of the socket created by the `syslogd` daemon (e.g., `/var/run/log`). This socket remains open until the process terminates. Alternately, the process can call `openlog` and `closelog`.

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);

void closelog(void);
```

`openlog` can be called before the first call to `syslog` and `closelog` can be called when the application is finished sending log messages.

ident is a string that will be prepended to each log message by `syslog`. Often this is the program name.

The *options* argument is formed as the logical OR of one or more of the constants in Figure 13.3.

Figure 13.3. *options* for `openlog`.

options	Description
LOG_CONS	Log to console if cannot send to syslogd daemon
LOG_NDELAY	Do not delay open, create socket now
LOG_PERROR	Log to standard error as well as sending to syslogd daemon
LOG_PID	log the Process ID with each message

Normally the Unix domain socket is not created when `openlog` is called. Instead, it is opened during the first call to `syslog`. The `LOG_NDELAY` option causes the socket to be created when `openlog` is called.

The *facility* argument to `openlog` specifies a default facility for any subsequent calls to `syslog` that do not specify a facility. Some daemons call `openlog` and specify the facility (which normally does not change for a given daemon). They then specify only the *level* in each call to `syslog` (since the *level* can change depending on the error).

Log messages can also be generated by the `logger` command. This can be used from within shell scripts, for example, to send messages to `syslogd`.

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 }
```

13.5 inetd Daemon

On a typical Unix system, there could be many servers in existence, just waiting for a client request to arrive. Examples are FTP, Telnet, Rlogin, TFTP, and so on. With systems before 4.3BSD, each of these services had a process associated with it. This process was started at boot-time from the file `/etc/rc`, and each process did nearly identical startup tasks: create a socket, `bind` the server's well-known port to the socket, wait for a connection (if TCP) or a datagram (if UDP), and then `fork`. The child process serviced the client and the parent waited for the next client request. There are two problems with this model:

1. All these daemons contained nearly identical startup code, first with respect to socket creation, and also with respect to becoming a daemon process (similar to our `daemon_init` function).
2. Each daemon took a slot in the process table, but each daemon was asleep most of the time.

The 4.3BSD release simplified this by providing an Internet *superserver:* the `inetd` daemon. This daemon can be used by servers that use either TCP or UDP. It does not handle other protocols, such as Unix domain sockets. This daemon fixes the two problems just mentioned:

1. It simplifies writing daemon processes since most of the startup details are handled by `inetd`. This obviates the need for each server to call our `daemon_init` function.
2. It allows a single process (`inetd`) to be waiting for incoming client requests for multiple services, instead of one process for each service. This reduces the total number of processes in the system.

The `inetd` process establishes itself as a daemon using the techniques that we described with our `daemon_init` function. It then reads and processes its configuration file, typically `/etc/inetd.conf`. This file specifies the services that the superserver is to handle, and what to do when a service request arrives. Each line contains the fields shown in Figure 13.6.

Figure 13.6. Fields in `inetd.conf` file.

Field	Description
service-name	Must be in `/etc/services`
socket-type	`stream` (TCP) or `dgram` (UDP)
protocol	Must be in `/etc/protocols`: either `tcp` or `udp`
wait-flag	Typically `nowait` for TCP or `wait` for UDP
login-name	From `/etc/passwd`: typically `root`
server-program	Full pathname to `exec`
server-program-arguments	Arguments for `exec`

Some sample lines are

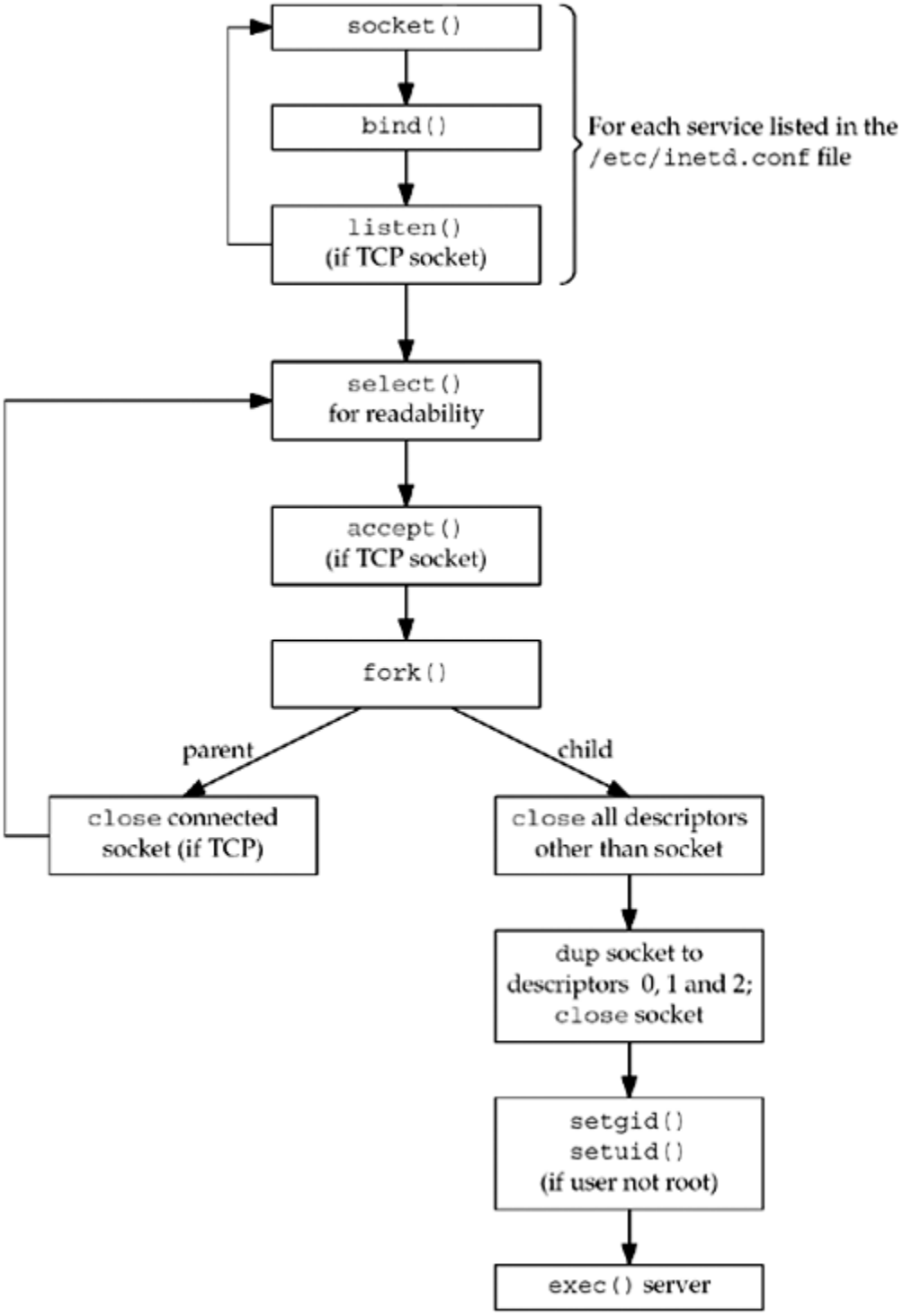
```
ftp     stream  tcp  nowait  root     /usr/bin/ftpd      ftpd -l
telnet  stream  tcp  nowait  root     /usr/bin/telnetd   telnetd
login   stream  tcp  nowait  root     /usr/bin/rlogind   rlogind -s
tftp    dgram   udp  wait    nobody   /usr/bin/tftpd     tftpd -s /tftpboot
```

The actual name of the server is always passed as the first argument to a program when it is `exec`ed.

> This figure and the sample lines are just examples. Most vendors have added their own features to `inetd`. Examples are the ability to handle RPC servers, in addition to TCP and UDP servers, and the ability to handle protocols other than TCP and UDP. Also, the pathname to `exec` and the command-line arguments to the server obviously depend on the implementation.
>
> The *wait-flag* field can be a bit confusing. In general, it specifies whether the daemon started by `inetd` intends to take over the listening socket associated with the service. UDP services don't have separate listening and accepting sockets, and are virtually always configured as `wait`. TCP services could be handled either way, at the discretion of the person writing the daemon, but `nowait` is most common.
>
> The interaction of IPv6 with `/etc/inetd.conf` depends on the vendor and special attention to detail is required to get what you want. Some use a *protocol* of `tcp6` or `udp6` to indicate that an IPv6 socket should be created for a service. Some allow *protocol* values of `tcp46` or `udp46` indicate the daemon wants sockets that allow both IPv6 and IPv4 connections. These special protocol names do not typically appear in the `/etc/protocols` file.

A picture of what the `inetd` daemon does is shown in Figure 13.7.

Figure 13.7. Steps performed by `inetd`.



1. On startup, it reads the `/etc/inetd.conf` file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. The maximum number of servers that `inetd` can handle depends on the maximum number of descriptors that `inetd` can create. Each new socket is added to a descriptor set that will be used in a call to `select`.

2. `bind` is called for the socket, specifying the port for the server and the wildcard IP address. This TCP or UDP port number is obtained by calling `getservbyname` with the *service-name* and *protocol* fields from the configuration file as arguments.

3. For TCP sockets, `listen` is called so that incoming connection requests are accepted. This step is not done for datagram sockets.

4. After all the sockets are created, `select` is called to wait for any of the sockets to become readable. Recall from Section 6.3 that a listening TCP socket becomes readable when a new connection is ready to be `accepted` and a UDP socket becomes readable when a datagram arrives. `inetd` spends most of its time blocked in this call to `select`, waiting for a socket to be readable.

5. When `select` returns that a socket is readable, if the socket is a TCP socket and the `nowait` flag is given, `accept` is called to accept the new connection.

6. The `inetd` daemon `forks` and the child process handles the service request. This is similar to a standard concurrent server (Section 4.8).

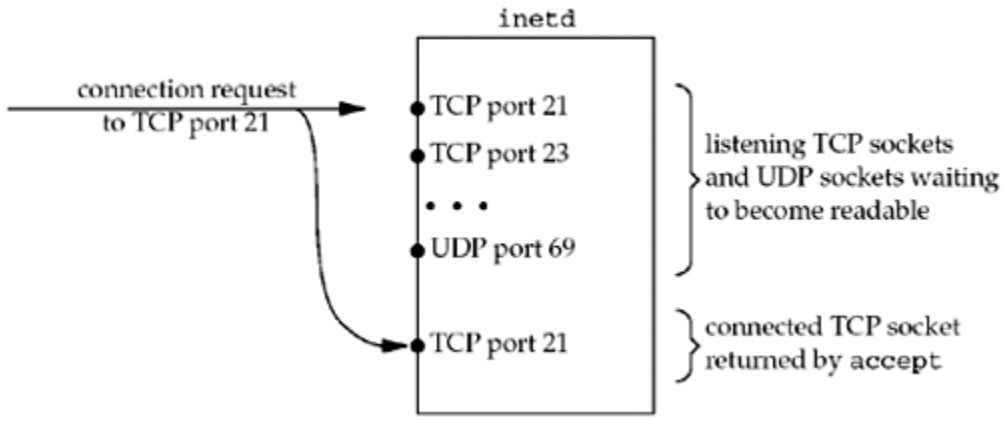
   The child closes all descriptors except the socket descriptor it is handling: the new connected socket returned by `accept` for a TCP server or the original UDP socket. The child calls `dup2` three times, duplicating the socket onto descriptors 0, 1, and 2 (standard input, standard output, and standard error). The original socket descriptor is then closed. By doing this, the only descriptors that are open in the child are 0, 1, and 2. If the child reads from standard input, it is reading from the socket and anything it writes to standard output or standard error is written to the socket. The child calls `getpwnam` to get the password file entry for the *login-name* specified in the configuration file. If this field is not `root`, then the child becomes the specified user by executing the `setgid` and `setuid` function calls. (Since the `inetd` process is executing with a user ID of 0, the child process inherits this user ID across the `fork`, and is able to become any user that it chooses.)

   The child process now does an `exec` to execute the appropriate *server-program* to handle the request, passing the arguments specified in the configuration file.

7. If the socket is a stream socket, the parent process must close the connected socket (like our standard concurrent server). The parent calls `select` again, waiting for the next socket to become readable.

If we look in more detail at the descriptor handling that is taking place, Figure 13.8 shows the descriptors in `inetd` when a new connection request arrives from an FTP client.

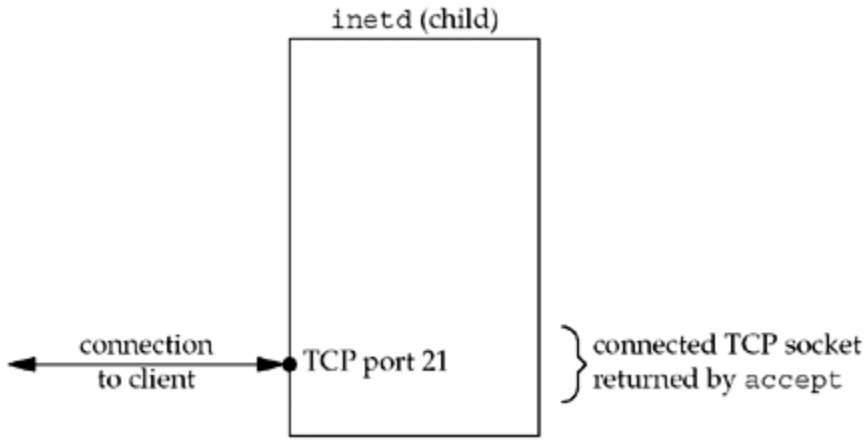
Figure 13.8. `inetd` descriptors when connection request arrives for TCP port 21.



The connection request is directed to TCP port 21, but a new connected socket is created by `accept`.

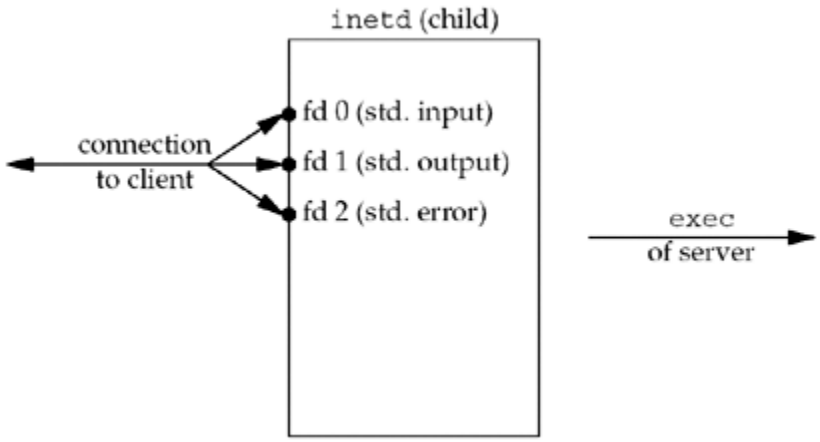
Figure 13.9 shows the descriptors in the child, after the call to `fork`, after the child has closed all the descriptors except the connected socket.

Figure 13.9. `inetd` descriptors in child.



The next step is for the child to duplicate the connected socket to descriptors 0, 1, and 2 and then close the connected socket. This gives us the descriptors shown in Figure 13.10.

Figure 13.10. `inetd` descriptors after `dup2`.



The child then calls `exec`. Recall from Section 4.7 that all descriptors normally remain open across an `exec`, so the real server that is `exec`ed uses any of the descriptors, 0, 1, or 2, to communicate with the client. These should be the only descriptors open in the server.

The scenario we have described handles the case where the configuration file specifies `nowait` for the server. This is typical for all TCP services and it means that `inetd` need not wait for its child to terminate before accepting another connection for that service. If another connection request arrives for the same service, it is returned to the parent process as soon as it calls `select` again. Steps 4, 5, and 6 listed earlier are executed again, and another child process handles this new request.

Specifying the `wait` flag for a datagram service changes the steps done by the parent process. This flag says that `inetd` must wait for its child to terminate before selecting on this socket again. The following changes occur:

1. When `fork` returns in the parent, the parent saves the process ID of the child. This allows the parent to know when this specific child process terminates, by looking at the value returned by `waitpid`.
2. The parent disables the socket from future `selects` by using the `FD_CLR` macro to turn off the bit in its descriptor set. This means that the child process takes over the socket until it terminates.
3. When the child terminates, the parent is notified by a `SIGCHLD` signal, and the parent's signal handler obtains the process ID of the terminating child. It reenables `select` for the corresponding socket by turning on the bit in its descriptor set for this socket.

The reason that a datagram server must take over the socket until it terminates, preventing `inetd` from `selecting` on that socket for readability (awaiting another client datagram), is because there is only one socket for a datagram server, unlike a TCP server that has a listening socket and one connected socket per client. If `inetd` did not turn off readability for the datagram socket, and if the parent (`inetd`) executed before the child, then the datagram from the client would still be in the socket receive buffer, causing `select` to return readable again, causing `inetd` to `fork` another (unneeded) child. `inetd` must ignore the datagram socket until it knows that the child has read the datagram from the socket receive queue. The way that `inetd` knows when that child is finished with the socket is by receiving `SIGCHLD`, indicating that the child has terminated. We will show an example of this in Section 22.7.

The five standard Internet services that we described in Figure 2.18 are handled internally by `inetd` (see Exercise 13.2).

Since `inetd` is the process that calls `accept` for a TCP server, the actual server that is invoked by `inetd` normally calls `getpeername` to obtain the IP address and port number of the client. Recall Figure 4.18 where we showed that after a `fork` and an `exec` (which is what `inetd` does), the only way for the actual server to obtain the identify of the client is to call `getpeername`.

`inetd` is normally not used for high-volume servers, notably mail and Web servers. `sendmail`, for example, is normally run as a standard concurrent server, as we described in Section 4.8. In this mode, the process control cost for each client connection is just a `fork`, while the cost for a TCP server invoked by `inetd` is a `fork` and an `exec`. Web servers use a variety of techniques to minimize the process control overhead for each client connection, as we will discuss in Chapter 30.

> It is now common to find an extended Internet services daemon, called `xinetd`, on Linux and other systems. `xinetd` provides the same basic function as `inetd`, but also includes a long list of other interesting features. Those features include options for logging, accepting or rejecting connections based on the client's address, configuring services one-per-file instead of a single monolithic configuration, and many more. It is not described further here since the basic *superserver* idea behind them both is the same.

13.6 daemon_inetd Function

Figure 13.11 shows a function named daemon_inetd that we can call from a server we know is invoked by inetd.

Figure 13.11 daemon_inetd function: daemonizes process run by inetd.

daemon_inetd.c

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

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

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;            /* for our err_XXX() functions */
8     openlog(pname, LOG_PID, facility);
9 }
```

This function is trivial compared to daemon_init, because all of the daemonization steps are performed by inetd when it starts. All that we do is set the daemon_proc flag for our error functions (Figure D.3) and call openlog with the same arguments as the call in Figure 13.4.

Example: Daytime Server as a Daemon Invoked by inetd

Figure 13.12 is a modification of our daytime server from Figure 13.5 that can be invoked by inetd.

Figure 13.12 Protocol-independent daytime server that can be invoked by inetd.

inetd/daytimetcpsrv3.c

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

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

10     daemon_inetd(argv[0], 0);

11     cliaddr = Malloc(sizeof(struct sockaddr_storage));
12     len = sizeof(struct sockaddr_storage);
13     Getpeername(0, cliaddr, &len);
14     err_msg("connection from %s", Sock_ntop(cliaddr, len));

15     ticks = time(NULL);
16     snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17     Write(0, buff, strlen(buff));

18     Close(0);                   /* close TCP connection */
19     exit(0);
20 }
```

There are two major changes in this program. First, all the socket creation code is gone: the calls to tcp_listen and to accept. Those steps are done by inetd and we reference the TCP connection using descriptor 0 (standard input). Second, the infinite for loop is gone because we are invoked once per client connection. After servicing this client, we terminate.

Call getpeername

11–14 Since we do not call tcp_listen, we do not know the size of the socket address structure it returns, and since we do not call accept, we do not know the client's protocol address. Therefore, we allocate a buffer for the socket address structure using sizeof(struct sockaddr_storage) and call getpeername with descriptor 0 as the first argument.

To run this example on our Solaris system, we first assign the service a name and port, adding the following line to `/etc/services`:

```
mydaytime      9999/tcp
```

We then add the following line to `/etc/inetd.conf`:

```
mydaytime  stream  tcp  nowait  andy
      /home/andy/daytimetcpsrv3  daytimetcpsrv3
```

(We have wrapped the long line.) We place the executable in the specified location and send the `SIGHUP` signal to `inetd`, telling it to reread its configuration file. The next step is to execute `netstat` to verify that a listening socket has been created on TCP port 9999.

```
solaris % netstat -na | grep 9999
      *.9999               *.*               0      0  49152     0  LISTEN
```

We then invoke the server from another host.

```
linux % telnet solaris 9999
Trying 192.168.1.20...
Connected to solaris.
Escape character is '^]'.
Tue Jun 10 11:04:02 2003
Connection closed by foreign host.
```

The `/var/adm/messages` file (where we have directed the `LOG_USER` facility messages to be logged in our `/etc/syslog.conf` file) contains the following entry:

```
Jun 10 11:04:02 solaris daytimetcpsrv3[28724]: connection from 192.168.1.10.58145
```

13.7 Summary

Daemons are processes that run in the background independent of control from all terminals. Many network servers run as daemons. All output from a daemon is normally sent to the `syslogd` daemon by calling the `syslog` function. The administrator then has complete control over what happens to these messages, based on the daemon that sent the message and the severity of the message.

To start an arbitrary program and have it run as a daemon requires a few steps: Call `fork` to run in the background, call `setsid` to create a new POSIX session and become the session leader, `fork` again to avoid obtaining a new controlling terminal, change the working directory and the file mode creation mask, and close all unneeded files. Our `daemon_init` function handles all these details.

Many Unix servers are started by the `inetd` daemon. It handles all the required daemonization steps, and when the actual server is started, the socket is open on standard input, standard output, and standard error. This lets us omit calls to `socket`, `bind`, `listen`, and `accept`, since all these steps are handled by `inetd`.