1.5 A Simple Daytime Server

We can write a simple version of a TCP daytime server, which will work with the client from Section 1.2. We use the wrapper functions that we described in the previous section and show this server in Figure 1.9.

Figure 1.9 TCP daytime server.

intro/daytimetcpsrv.c

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

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int      listenfd, connfd;
 7     struct sockaddr_in servaddr;
 8     char     buff[MAXLINE];
 9     time_t ticks;

10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11     bzeros(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13); /* daytime server */

15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16     Listen(listenfd, LISTENQ);

17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);

19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));

22         Close(connfd);
23     }
24 }
```

Create a TCP socket

10 The creation of the TCP socket is identical to the client code.

Bind server's well-known port to socket

11–15 The server's well-known port (13 for the daytime service) is bound to the socket by filling in an Internet socket address structure and calling `bind`. We specify the IP address as `INADDR_ANY`, which allows the server to accept a client connection on any interface, in case the server host has multiple interfaces. Later we will see how we can restrict the server to accepting a client connection on just a single interface.

Convert socket to listening socket

16 By calling `listen`, the socket is converted into a listening socket, on which incoming connections from clients will be accepted by the kernel. These three steps, `socket`, `bind`, and `listen`, are the normal steps for any TCP server to prepare what we call the *listening descriptor* (`listenfd` in this example).

The constant `LISTENQ` is from our `unp.h` header. It specifies the maximum number of client connections that the kernel will queue for this listening descriptor. We say much more about this queueing in Section 4.5.

Accept client connection, send reply

17–21 Normally, the server process is put to sleep in the call to `accept`, waiting for a client connection to arrive and be accepted. A TCP connection uses what is called a *three-way handshake* to establish a connection. When this handshake completes, `accept` returns, and the return value from the function is a new descriptor (`connfd`) that is called the *connected descriptor*. This new descriptor is used for communication with the new client. A new descriptor is returned by `accept` for each client that connects to our server.

> The style used throughout the book for an infinite loop is

```
for ( ; ; ) {
    . . .
}
```

The current time and date are returned by the library function `time`, which returns the number of seconds since the Unix Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). The next library function, `ctime`, converts this integer value into a human-readable string such as

```
Mon May 26 20:58:40 2003
```

A carriage return and linefeed are appended to the string by `snprintf`, and the result is written to the client by `write`.

> If you're not already in the habit of using `snprintf` instead of the older `sprintf`, now's the time to learn. Calls to `sprintf` cannot check for overflow of the destination buffer. `snprintf`, on the other hand, requires that the second argument be the size of the destination buffer, and this buffer will not overflow.
>
> `snprintf` was a relatively late addition to the ANSI C standard, introduced in the version referred to as *ISO C99*. Virtually all vendors provide it as part of the standard C library, and many freely available versions are also available. We use `snprintf` throughout the text, and we recommend using it instead of `sprintf` in all your programs for reliability.
>
> It is remarkable how many network break-ins have occurred by a hacker sending data to cause a server's call to `sprintf` to overflow its buffer. Other functions that we should be careful with are `gets`, `strcat`, and `strcpy`, normally calling `fgets`, `strncat`, and `strncpy` instead. Even better are the more recently available functions `strlcat` and `strlcpy`, which ensure the result is a properly terminated string. Additional tips on writing secure network programs are found in Chapter 23 of [Garfinkel, Schwartz, and Spafford 2003].

Terminate connection

22 The server closes its connection with the client by calling `close`. This initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end. We will say much more about TCP's three-way handshake and the four TCP packets used to terminate a TCP connection in Section 2.6.

As with the client in the previous section, we have only examined this server briefly, saving all the details for later in the book. Note the following points:

- As with the client, the server is protocol-dependent on IPv4. We will show a protocol-independent version that uses the `getaddrinfo` function in Figure 11.13.
- Our server handles only one client at a time. If multiple client connections arrive at about the same time, the kernel queues them, up to some limit, and returns them to `accept` one at a time. This daytime server, which requires calling two library functions, `time` and `ctime`, is quite fast. But if the server took more time to service each client (say a few seconds or a minute), we would need some way to overlap the service of one client with another client.
- The server that we show in Figure 1.9 is called an *iterative server* because it iterates through each client, one at a time. There are numerous techniques for writing a *concurrent server*, one that handles multiple clients at the same time. The simplest technique for a concurrent server is to call the Unix `fork` function (Section 4.7), creating one child process for each client. Other techniques are to use threads instead of `fork` (Section 26.4), or to pre-`fork` a fixed number of children when the server starts (Section 30.6).
- If we start a server like this from a shell command line, we might want the server to run for a long time, since servers often run for as long as the system is up. This requires that we add code to the server to run correctly as a Unix *daemon*: a process that can run in the background, unattached to a terminal. We will cover this in Section 13.4.