4.6 `accept` Function

`accept` is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.7). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, -1 on error

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument (Section 3.3): Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*; on return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If `accept` is successful, its return value is a brand-new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing `accept`, we call the first argument to `accept` the *listening socket* (the descriptor created by `socket` and then used as the first argument to both `bind` and `listen`), and we call the return value from `accept` the *connected socket*. It is important to differentiate between these two sockets. A given server normally creates only one listening socket, which then exists for the lifetime of the server. The kernel creates one connected socket for each client connection that is `accept`ed (i.e., for which the TCP three-way handshake completes). When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the *cliaddr* pointer), and the size of this address (through the *addrlen* pointer). If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers.

Figure 1.9 shows these points. The connected socket is closed each time through the loop, but the listening socket remains open for the life of the server. We also see that the second and third arguments to `accept` are null pointers, since we were not interested in the identity of the client.

Example: Value-Result Arguments

We will now show how to handle the value-result argument to `accept` by modifying the code from Figure 1.9 to print the IP address and port of the client. We show this in Figure 4.11.

Figure 4.11 Daytime server that prints client IP address and port

intro/daytimetcpsrv1.c

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

int
main(int argc, char **argv)
{
    int     listenfd, connfd;
    socklen_t len;
    struct sockaddr_in servaddr, cliaddr;
    char    buff[MAXLINE];
    time_t  ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

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

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        len = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &len);
        printf("connection from %s, port %d\n",
               Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
               ntohs(cliaddr.sin_port));
24          ticks = time(NULL);
25          snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26          Write(connfd, buff, strlen(buff));

27          Close(connfd);
28      }
29 }
```

New declarations

7-8 We define two new variables: `len`, which will be a value-result variable, and `cliaddr`, which will contain the client's protocol address.

Accept connection and print client's address

19-23 We initialize `len` to the size of the socket address structure and pass a pointer to the `cliaddr` structure and a pointer to `len` as the second and third arguments to `accept`. We call `inet_ntop` (Section 3.7) to convert the 32-bit IP address in the socket address structure to a dotted-decimal ASCII string and call `ntohs` (Section 3.4) to convert the 16-bit port number from network byte order to host byte order.

> Calling `sock_ntop` instead of `inet_ntop` would make our server more protocol-independent, but this server is already dependent on IPv4. We will show a protocol-independent version of this server in Figure 11.13.

If we run our new server and then run our client on the same host, connecting to our server twice in a row, we have the following output from the client:

```
solaris % daytimetcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
solaris % daytimetcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

We first specify the server's IP address as the loopback address (127.0.0.1) and then as its own IP address (192.168.1.20). Here is the corresponding server output:

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

Notice what happens with the client's IP address. Since our daytime client (Figure 1.5) does not call `bind`, we said in Section 4.4 that the kernel chooses the source IP address based on the outgoing interface that is used. In the first case, the kernel sets the source IP address to the loopback address; in the second case, it sets the address to the IP address of the Ethernet interface. We can also see in this example that the ephemeral port chosen by the Solaris kernel is 43388, and then 43389 (recall Figure 2.10).

As a final point, our shell prompt for the server script changes to the pound sign (#), the commonly used prompt for the superuser. Our server must run with superuser privileges to `bind` the reserved port of 13. If we do not have superuser privileges, the call to `bind` will fail:

```
solaris % daytimetcpsrv1
bind error: Permission denied
```

[Team LiB]