Socket Tutorial

This section covers the basic methodologies of using sockets.

Socket Creation

The **`socket()`** call creates a socket in the specified domain and of the specified type.

```
s = socket(domain, type, protocol);
```

If the protocol is unspecified (a value of `0`), the system selects a protocol that supports the requested socket type. The socket handle (a file descriptor) is returned.

The domain is specified by one of the constants defined in `sys/socket.h`. Constants named `AF_`*suite* specify the address format to use in interpreting names as shown in Table 2-1.

Table 2-1 Protocol Family

`AF_APPLETALK`	Apple Computer Inc. Appletalk network
`AF_INET`	Internet domain
`AF_PUP`	Xerox Corporation PUP internet
`AF_UNIX`	Unix file system

Socket types are defined in `sys/socket.h`. These types--`SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`--are supported by `AF_INET` and `AF_UNIX`. The following creates a stream socket in the Internet domain:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket with the TCP protocol providing the underlying communication. The following creates a datagram socket for intramachine use:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Use the default protocol (the *protocol* argument is `0`) in most situations. You can specify a protocol other than the default, as described in "Advanced Topics".

Binding Local Names

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to it. Communicating processes are connected through addresses. In the Internet domain, a connection is composed of local and remote addresses, and local and remote ports. In the UNIX domain, a connection is composed of (usually) one or two path names. In most domains, connections must be unique.

In the Internet domain, there can never be duplicate ordered sets, such as: `protocol`, `local address`, `local port`, `foreign address`, `foreign port`. UNIX domain sockets need not always be bound to a name, but, when bound, there can never be duplicate ordered sets such as: `local pathname` or `foreign pathname`. The path names cannot refer to existing files.

The **`bind()`** call allows a process to specify the local address of the socket. This forms the set `local address`, `local port` (or `local pathname`) while **`connect()`** and **`accept()`** complete a socket's association by fixing the remote half of the address tuple. The **`bind()`** function call is used as follows:

```
bind (s, name, namelen);
```

The socket handle is *s*. The bound name is a byte string that is interpreted by the supporting protocol(s). Internet domain names contain an Internet address and port number. UNIX domain names contain a path name and a family. Example 2-1 shows binding the name `/tmp/foo` to a UNIX domain socket.

Example 2-1 Bind Name to Socket

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
               strlen(addr.sun_path) + sizeof (addr.sun_family));
```

When determining the size of an `AF_UNIX` socket address, null bytes are not counted, which is why **`strlen()`** use is fine.

The file name referred to in `addr.sun_path` is created as a socket in the system file name space. The caller must have write permission in the directory where `addr.sun_path` is created. The file should be deleted by the caller when it is no longer needed. `AF_UNIX` sockets can be deleted with **`unlink()`**.

Binding an Internet address is more complicated but the call is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
                s = socket(AF_INET, SOCK_STREAM, 0);
                sin.sin_family = AF_INET;
                sin.sin_addr.s_addr = htonl(INADDR_ANY);
                sin.sin_port = htons(MYPORT);
                bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The content of the address `sin` is described in "Address Binding", where Internet address bindings are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the **`connect()`** call initiates a connection. In the UNIX domain, this might appear as:

```
struct sockaddr_un server;
server.sun.family = AF_UNIX;
 ...
connect(s, (struct sockaddr *)&server,
                                strlen(server.sun_path) + sizeof (server.sun_family));
```

In the Internet domain it might appear as:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. See "Signals and Process Group ID". This is the usual way that local addresses are bound to a socket on the client side.

In the examples that follow, only `AF_INET` sockets are described.

To receive a client's connection, a server must perform two steps after binding its socket. The first is to indicate how many connection requests can be queued. The second step is to accept a connection:

```
struct sockaddr_in from;
 ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

The socket handle *s* is the socket bound to the address to which the connection request is sent. The second parameter of **`listen()`** specifies the maximum number of outstanding connections that might be queued. `from` is a structure that is filled with the address of the client. A `NULL` pointer might be passed. *fromlen* is the length of the structure. (In the UNIX domain, `from` is declared a `struct sockaddr_un`.)

`accept()` normally blocks. **`accept()`** returns a new socket descriptor that is connected to the requesting client. The value of *fromlen* is changed to the actual size of the address.

A server cannot indicate that it accepts connections only from specific addresses. The server can check the from address returned by **`accept()`** and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the accept call. These techniques are presented in "Advanced Topics".

Connection Errors

An error is returned if the connection is unsuccessful (however, an address bound by the system remains). Otherwise, the socket is associated with the server and data transfer can begin.

Table 2-2 lists some of the more common errors returned when a connection attempt fails.

Table 2-2 Socket Connection Errors

Socket Errors	**Error Description**
`ENOBUFS`	Lack of memory available to support the call.
`EPROTONOSUPPORT`	Request for an unknown protocol.
`EPROTOTYPE`	Request for an unsupported type of socket.
`ETIMEDOUT`	No connection established in specified time. This happens when the destination host is down or when problems in the network result in lost transmissions.

Socket Errors	Error Description
`ECONNREFUSED`	The host refused service. This happens when a server process is not present at the requested address.
`ENETDOWN` or `EHOSTDOWN`	These errors are caused by status information delivered by the underlying communication interface.
`ENETUNREACH` or `EHOSTUNREACH`	These operational errors can occur either because there is no route to the network or host, or because of status information returned by intermediate gateways or switching nodes. The status returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the functions to send and receive data. You can send or receive a message with the normal **`read()`** and **`write()`** function calls:

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

Or the calls **`send()`** and **`recv()`** can be used:

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send()` and **`recv()`** are very similar to **`read()`** and **`write()`**, but the `flags` argument is important. The flags, defined in `sys/socket.h`, can be specified as a nonzero value if one or more of the following is required:

`MSG_OOB`

Send and receive out-of-band data

`MSG_PEEK`

Look at data without reading

`MSG_DONTROUTE`

Send data without routing packets

Out-of-band data is specific to stream sockets. When `MSG_PEEK` is specified with a **`recv()`** call, any data present is returned to the user but treated as still unread. The next **`read()`** or **`recv()`** call on the socket returns the same data. The option to send data without routing packets applied to the outgoing packets is currently used only by the routing table management process and is unlikely to be interesting to most users.

Closing Sockets

A `SOCK_STREAM` socket can be discarded by a **`close()`** function call. If data is queued to a socket that promises reliable delivery after a **`close()`**, the protocol continues to try to transfer the data. If the data is still undelivered after an arbitrary period, it is discarded.

A **`shutdown()`** closes `SOCK_STREAM` sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

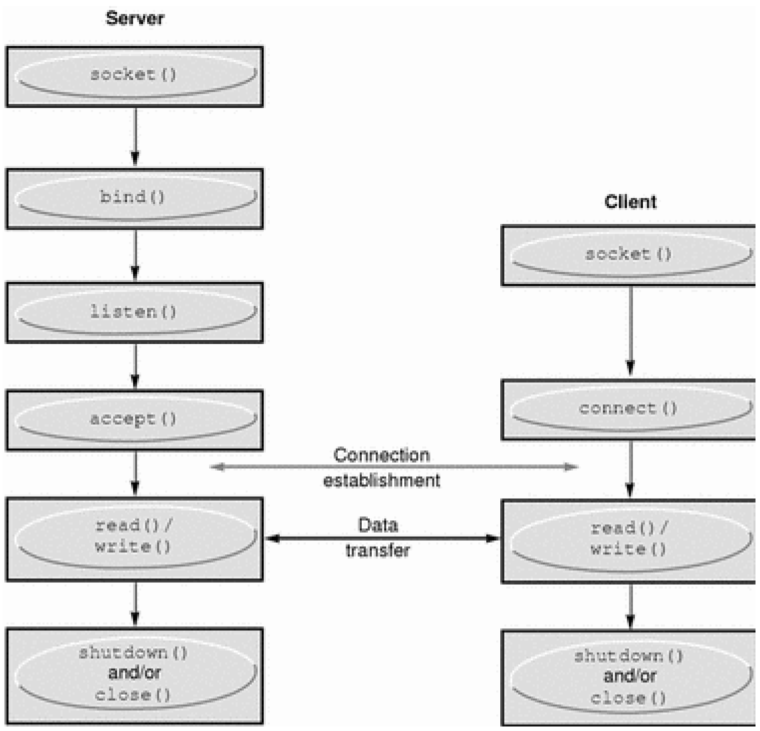
Where `how` is defined as:

0	Disallows further receives
1	Disallows further sends
2	Disallows both further sends and receives

Connecting Stream Sockets

Figure 2-1 and the next two examples illustrate initiating and accepting an Internet domain stream connection.

Figure 2-1 Connection-Oriented Communication Using Stream Sockets



The program in Example 2-2 is a server. It creates a socket and binds a name to it, then displays the port number. The program calls **`listen()`** to mark the socket ready to accept connection requests and initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes it. The use of `INADDR_ANY` is explained in "Address Binding".

Example 2-2 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
* This program creates a socket and then begins an infinite loop.
* Each time through the loop it accepts a connection and prints
* data from it. When the connection breaks, or the client closes
* the connection, the program accepts a new connection.
*/

main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;

   /* Create socket. */
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock == -1) {
      perror("opening stream socket");
      exit(1);
   }
   /* Bind socket using wildcards.*/
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = 0;
   if (bind(sock, (struct sockaddr *) &server, sizeof server)
         == -1) {
      perror("binding stream socket");
      exit(1);
   }
   /* Find out assigned port number and print it out. */
   length = sizeof server;
   if (getsockname(sock,(struct sockaddr *) &server,&length)
         == -1) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(server.sin_port));
   /* Start accepting connections. */
   listen(sock, 5);
   do {
      msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
      if (msgsock == -1
         perror("accept");
      else do {
         memset(buf, 0, sizeof buf);
         if ((rval = read(msgsock,buf, 1024)) == -1)
            perror("reading stream message");
         if (rval == 0)
            printf("Ending connection\n");
         else
            /* assumes the data is printable */
            printf("-->%s\n", buf);
      } while (rval > 0);
      close(msgsock);
   } while(TRUE);
   /*
    * Since this program has an infinite loop, the socket "sock" is
    * never explicitly closed. However, all sockets will be closed
    * automatically when a process is killed or terminates normally.
    */
   exit(0);
}
```

To initiate a connection, the client program in Example 2-3 creates a stream socket and calls **connect()**, specifying the address of the socket for connection. If the target socket exists and the request is accepted, the connection is complete and the program can send data. Data are delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines, such as **ntohl()**, **ntohs()**, **htons()**, and **htonl()**, in this program, see the byteorder(3N) man page.

Example 2-3 Internet Domain Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(argc, argv)
                int argc;
                char *argv[];
{
                int sock;
                        struct sockaddr_in server;
                        struct hostent *hp, *gethostbyname();
                        char buf[1024];

                        /* Create socket. */
                        sock = socket( AF_INET, SOCK_STREAM, 0);
                        if (sock == -1) {
                                perror("opening stream socket");
                                exit(1);
                        }
                        /* Connect socket using name specified by command line. */
                        server.sin_family = AF_INET;
                        hp = gethostbyname(argv[1] );
/*
 * gethostbyname returns a structure including the network address
 * of the specified host.
 */
                if (hp == (struct hostent *) 0) {
                        fprintf(stderr, "%s: unknown host\n", argv[1]);
                        exit(2);
                }
                memcpy((char *) &server.sin_addr, (char *) hp->h_addr,
                        hp->h_length);
                server.sin_port = htons(atoi(argv[2]));
                if (connect(sock, (struct sockaddr *) &server, sizeof server)
                                == -1) {
                        perror("connecting stream socket");
                        exit(1);
                }
                if (write( sock, DATA, sizeof DATA) == -1)
                        perror("writing on stream socket");
                close(sock);
                exit(0);
}
```

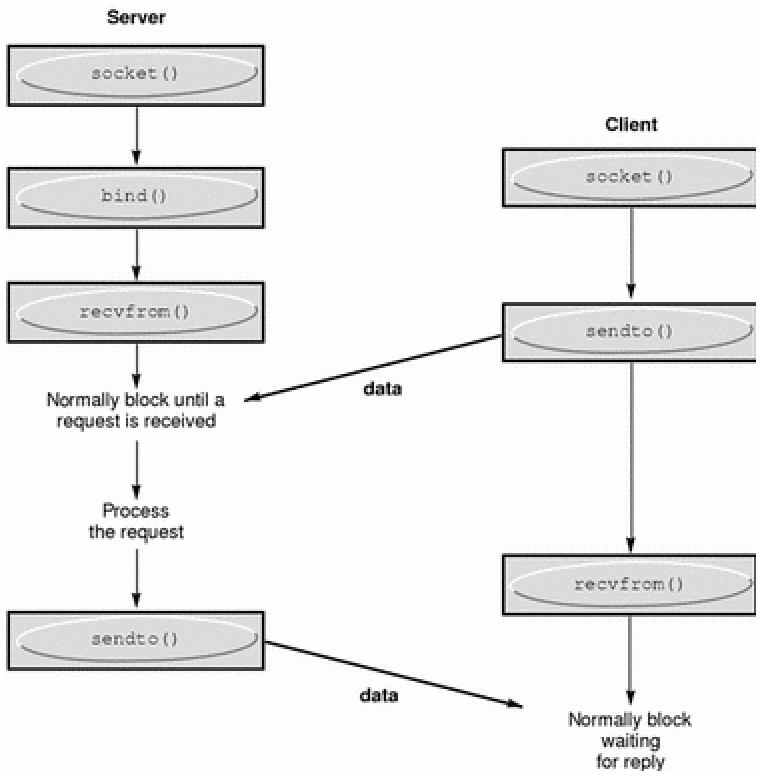
Datagram Sockets

A datagram socket provides a symmetric data exchange interface. There is no requirement for connection establishment. Each message carries the destination address. Figure 2-2 shows the flow of communication between server and client.

Note -

The **`bind()`** step shown below for the server is optional.

Figure 2-2 Connectionless Communication Using Datagram Sockets



Datagram sockets are created as described in "Socket Creation". If a particular local address is needed, the **`bind()`** operation must precede the first data transmission. Otherwise, the system sets the local address and/or port when data is first sent. To send data, the **`sendto()`** call is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are the same as in connection-oriented sockets. The *to* and *tolen* values indicate the address of the intended recipient of the message. A locally detected error condition (such as an unreachable network) causes a return of `-1` and *errno* to be set to the error number.

To receive messages on a datagram socket, the **`recvfrom()`** call is used:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

Before the call, *fromlen* is set to the size of the *from* buffer. On return, it is set to the size of the address from which the datagram was received.

Datagram sockets can also use the **`connect()`** call to associate a socket with a specific destination address. It can then use the **`send()`** call. Any data sent on the socket without explicitly specifying a destination address is addressed to the connected peer, and only data received from that peer is delivered. Only one connected address is permitted for one socket at a time. A second **`connect()`** call changes the destination address. Connect requests on datagram sockets return immediately. The system records the peer's address. **`accept()`**, and **`listen()`** are not used with datagram sockets.

While a datagram socket is connected, errors from previous **`send()`** calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket, or an option of **`getsockopt()`**, `SO_ERROR`, can be used to interrogate the error status.

Example 2-4 shows how to send an Internet call by creating a socket, binding a name to the socket, and sending the message to the socket.

Example 2-4 Sending an Internet Domain Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_in name;
   struct hostent *hp, *gethostbyname();

   /* Create socket on which to send. */
   sock = socket(AF_INET,SOCK_DGRAM, 0);
   if (sock == -1) {
      perror("opening datagram socket");
      exit(1);
   }
   /*
    * Construct name, with no wildcards, of the socket to ``send''
    * to. gethostbyname returns a structure including the network
    * address of the specified host. The port number is taken from
    * the command line.
    */
   hp = gethostbyname(argv[1]);
   if (hp == (struct hostent *) 0) {
      fprintf(stderr, "%s: unknown host\n", argv[1]);
      exit(2);
   }
   memcpy((char *) &name.sin_addr, (char *) hp->h_addr,
      hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));
   /* Send message. */
   if (sendto(sock,DATA, sizeof DATA ,0,
      (struct sockaddr *) &name,sizeof name) == -1)
      perror("sending datagram message");
   close(sock);
   exit(0);
}
```

Example 2-5 shows how to read an Internet call by creating a socket, binding a name to the socket, and then reading from the socket.

Example 2-5 Reading Internet Domain Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
* The include file <netinet/in.h> defines sockaddr_in as:
 *      struct sockaddr_in {
 *              short                   sin_family;
 *              u_short                 sin_port;
 *              struct                  in_addr sin_addr;
 *              char                    sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */

main()
{
   int sock, length;
   struct sockaddr_in name;
   char buf[1024];

   /* Create socket from which to read. */
   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock == -1) {
      perror("opening datagram socket");
      exit(1);
   }
   /* Create name with wildcards. */
   name.sin_family = AF_INET;
   name.sin_addr.s_addr = INADDR_ANY;
   name.sin_port = 0;
   if (bind(sock,(struct sockaddr *)&name, sizeof name) == -1) {
      perror("binding datagram socket");
      exit(1);
   }
   /* Find assigned port value and print it out. */
   length = sizeof(name);
   if (getsockname(sock,(struct sockaddr *) &name, &length)
         == -1)         {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(name.sin_port));
   /* Read from the socket. */
   if (read(sock, buf, 1024) == -1 )
      perror("receiving datagram packet");
   /* Assumes the data is printable */
   printf("-->%s\n", buf);
   close(sock);
   exit(0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or files. Use the **`select()`** call to do this:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of **`select()`** is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of **`select()`** are pointers to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. Any of these pointers can be a properly cast null. Each set is a structure containing an array of long integer bit masks. The size of the array is set by `FD_SETSIZE` (defined in `select.h`). The array is long enough to hold one bit for each `FD_SETSIZE` file descriptor.

The macros `FD_SET`(*fd*, *&mask*) and `FD_CLR`(*fd*, *&mask*) add and delete, respectively, the file descriptor *fd* in the set `mask`. The set should be zeroed before use, and the macro `FD_ZERO`(*&mask*) clears the set `mask`.

The fifth argument of **`select()`** allows a time-out value to be specified. If the `timeout` pointer is `NULL`, **`select()`** blocks until a descriptor is selectable, or until a signal is received. If the fields in `timeout` are set to `0`, **`select()`** polls and returns immediately.

`select()` normally returns the number of file descriptors selected. **`select()`** returns a `0` if the time-out has expired. **`select()`** returns `-1` for an error or interrupt with the error number in *errno* and the file descriptor masks unchanged. For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

You should test the status of a file descriptor in a select mask with the `FD_ISSET`(*fd*, *&mask*) macro. It returns a nonzero value if *fd* is in the set `mask`, and `0` if it is not. Use **`select()`** followed by a `FD_ISSET`(*fd*, *&mask*) macro on the read set to check for queued connect requests on a socket.

Example 2-6 shows how to select on a "listening" socket for readability to determine when a new connection can be picked up with a call to **`accept()`**. The program accepts connection requests, reads data, and disconnects on a single socket.

Example 2-6 Using `select()` to Check for Pending Connections

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */

main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;
   fd_set ready;
   struct timeval to;

   /* Open a socket and bind it as in previous examples. */

   /* Start accepting connections. */
   listen(sock, 5);
   do {
      FD_ZERO(&ready);
      FD_SET(sock, &ready);
      to.tv_sec = 5;
      to.tv_usec = 0;
      if (select(sock + 1, &ready, (fd_set *)0, (fd_set *)0, &to) == -1) {
         perror("select");
         continue;
      }
      if (FD_ISSET(sock, &ready)) {
         msgsock = accept(sock, (struct sockaddr *)0,
            (int *)0);
         if (msgsock == -1)
            perror("accept");
         else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock, buf, 1024)) == -1)
               perror("reading stream message");
            else if (rval == 0)
               printf("Ending connection\n");
            else
               printf("-->%s\n", buf);
         } while (rval > 0);
         close(msgsock);
      } else
         printf("Do something else\n");
   } while (TRUE);
           exit(0);
}
```

In previous versions of the **`select()`** routine, its arguments were pointers to integers instead of pointers to `fd_sets`. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

`select()` provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals described in "Advanced Topics" provide asynchronous notification of output completion, input availability, and exceptional conditions.

- *Previous*: What Are Sockets?
- *Next*: Standard Routines