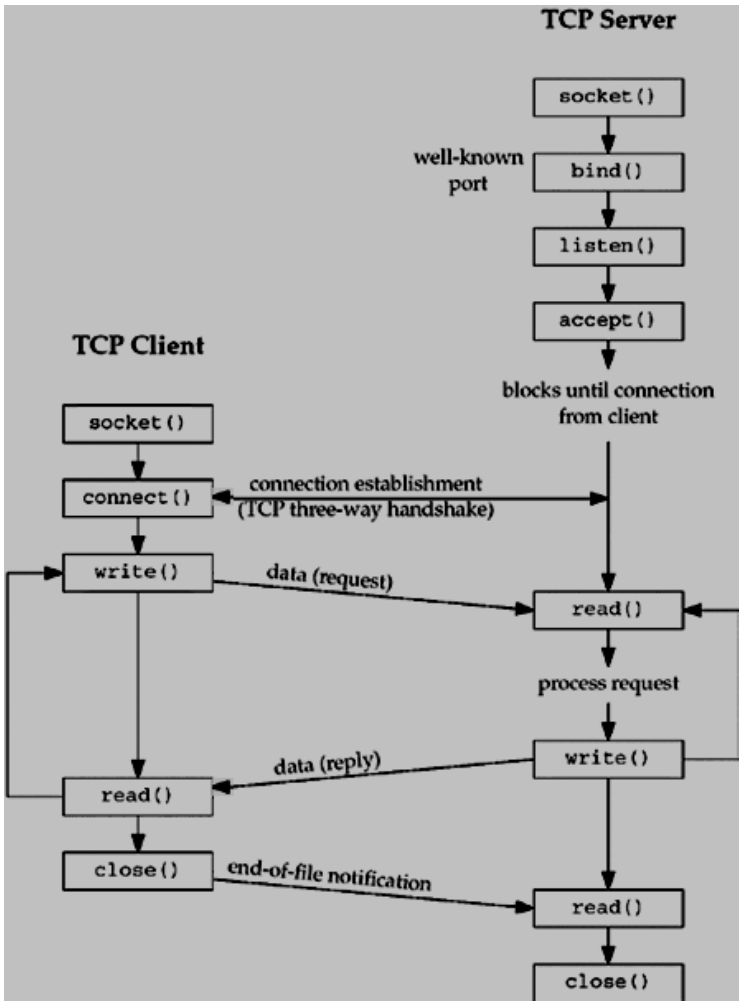
Unix Socket - Core Functions

Previous Page Next Page

This chapter describes the core socket functions required to write a complete TCP client and server.

The following diagram shows the complete Client and Server interaction −



The socket Function

To perform network I/O, the first thing a process must do is, call the socket function, specifying the type communication protocol desired and protocol family, etc.

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

int socket (int family, int type, int protocol);
```

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

family − It specifies the protocol family and is one of the constants shown below −

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

This chapter does not cover other protocols except IPv4.

type − It specifies the kind of socket you want. It can take one of the following values −

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol − The argument should be set to the specific protocol type given below, or 0 to select the system's defa for the given combination of family and type −

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

The *connect* Function

The *connect* function is used by a TCP client to establish a connection with a TCP server.

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

serv_addr − It is a pointer to struct sockaddr that contains destination IP address and port.

addrlen − Set it to sizeof(struct sockaddr).

The *bind* Function

The *bind* function assigns a local protocol address to a socket. With the Internet protocols, the protocol address the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP p number. This function is called by TCP server only.

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

int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

my_addr − It is a pointer to struct sockaddr that contains the local IP address and port.

addrlen − Set it to sizeof(struct sockaddr).

You can put your IP address and your port automatically

A 0 value for port number means that the system will choose a random port, and *INADDR_ANY* value for IP addre means the server's IP address will be assigned automatically.

```
server.sin_port = 0;
server.sin_addr.s_addr = INADDR_ANY;
```

NOTE − All ports below 1024 are reserved. You can set a port above 1024 and below 65535 unless they are t ones being used by other programs.

The *listen* Function

The *listen* function is called only by a TCP server and it performs two actions −

The listen function converts an unconnected socket into a passive socket, indicating that the kernel shou accept incoming connection requests directed to this socket.

The second argument to this function specifies the maximum number of connections the kernel shou queue for this socket.

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

```
int listen(int sockfd,int backlog);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

backlog − It is the number of allowed connections.

The *accept* Function

The *accept* function is called by a TCP server to return the next completed connection from the front of t completed connection queue. The signature of the call is as follows −

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

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

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communica with the client.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

cliaddr − It is a pointer to struct sockaddr that contains client IP address and port.

addrlen − Set it to sizeof(struct sockaddr).

The *send* Function

The *send* function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to se data over UNCONNECTED datagram sockets, you must use sendto() function.

You can use *write()* system call to send data. Its signature is as follows −

```
int send(int sockfd, const void *msg, int len, int flags);
```

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

msg − It is a pointer to the data you want to send.

len − It is the length of the data you want to send (in bytes).

flags − It is set to 0.

The *recv* Function

The *recv* function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want receive data over UNCONNECTED datagram sockets you must use recvfrom().

You can use *read()* system call to read the data. This call is explained in helper functions chapter.

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

buf − It is the buffer to read the information into.

len − It is the maximum length of the buffer.

flags − It is set to 0.

The *sendto* Function

The *sendto* function is used to send data over UNCONNECTED datagram sockets. Its signature is as follows −

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen
```

This call returns the number of bytes sent, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

msg − It is a pointer to the data you want to send.

len − It is the length of the data you want to send (in bytes).

flags − It is set to 0.

to − It is a pointer to struct sockaddr for the host where data has to be sent.

tolen − It is set it to sizeof(struct sockaddr).

The *recvfrom* Function

The *recvfrom* function is used to receive data from UNCONNECTED datagram sockets.

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);
```

This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

buf − It is the buffer to read the information into.

len − It is the maximum length of the buffer.

flags − It is set to 0.

from − It is a pointer to struct sockaddr for the host where data has to be read.

fromlen − It is set it to sizeof(struct sockaddr).

The *close* Function

The *close* function is used to close the communication between the client and the server. Its syntax is as follows −

```
int close( int sockfd );
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

The *shutdown* Function

The *shutdown* function is used to gracefully close the communication between the client and the server. This functi gives more control in comparison to the *close* function. Given below is the syntax of *shutdown* −

```
int shutdown(int sockfd, int how);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

sockfd − It is a socket descriptor returned by the socket function.

how − Put one of the numbers −

0 − indicates that receiving is not allowed,

1 − indicates that sending is not allowed, and

2 − indicates that both sending and receiving are not allowed. When *how* is set to 2, it's the sar thing as close().

The *select* Function

The *select* function indicates which of the specified file descriptors is ready for reading, ready for writing, or has error condition pending.

When an application calls *recv or recvfrom*, it is blocked until data arrives for that socket. An application could

doing other useful processing while the incoming data stream is empty. Another situation is when an applicati receives data from multiple sockets.

Calling *recv or recvfrom* on a socket that has no data in its input queue prevents immediate reception of data fr other sockets. The select function call solves this problem by allowing the program to poll all the socket handles see if they are available for non-blocking reading and writing operations.

Given below is the syntax of *select* −

```
int select(int  nfds, fd_set  *readfds, fd_set  *writefds, fd_set *errorfds, struct timeval *timeout);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

nfds − It specifies the range of file descriptors to be tested. The select() function tests file descriptors in t range of 0 to nfds-1

readfds − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked being ready to read, and on output, indicates which file descriptors are ready to read. It can be NULL indicate an empty set.

writefds − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked being ready to write, and on output, indicates which file descriptors are ready to write. It can be NULL indicate an empty set.

exceptfds − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked error conditions pending, and on output indicates, which file descriptors have error conditions pending can be NULL to indicate an empty set.

timeout − It points to a timeval struct that specifies how long the select call should poll the descriptors for available I/O operation. If the timeout value is 0, then select will return immediately. If the timeout argume is NULL, then select will block until at least one file/socket handle is ready for an available I/O operatic Otherwise *select* will return after the amount of time in the timeout has elapsed OR when at least o file/socket descriptor is ready for an I/O operation.

The return value from select is the number of handles specified in the file descriptor sets that are ready for I/O. If t time limit specified by the timeout field is reached, select return 0. The following macros exist for manipulating a f descriptor set −

FD_CLR(fd, &fdset) − Clears the bit for the file descriptor fd in the file descriptor set *fdset.*

FD_ISSET(fd, &fdset) − Returns a non-zero value if the bit for the file descriptor *fd* is set in the f descriptor set pointed to by *fdset*, and 0 otherwise.

FD_SET(fd, &fdset) − Sets the bit for the file descriptor fd in the file descriptor set fdset.

FD_ZERO(&fdset) − Initializes the file descriptor set fdset to have zero bits for all file descriptors.

The behavior of these macros is undefined if the fd argument is less than 0 or greater than or equal to FD_SETSIZ

Example

```
fd_set fds;

struct timeval tv;

/* do socket initialization etc.
tv.tv_sec = 1;
tv.tv_usec = 500000;

/* tv now represents 1.5 seconds */
FD_ZERO(&fds);

/* adds sock to the file descriptor set */
FD_SET(sock, &fds);

/* wait 1.5 seconds for any data to be read from any single socket */
select(sock+1, &fds, NULL, NULL, &tv);

if (FD_ISSET(sock, &fds)) {
   recvfrom(s, buffer, buffer_len, 0, &sa, &sa_len);
   /* do something */
}
else {
   /* do something else */
}
```

Previous Page Next Page