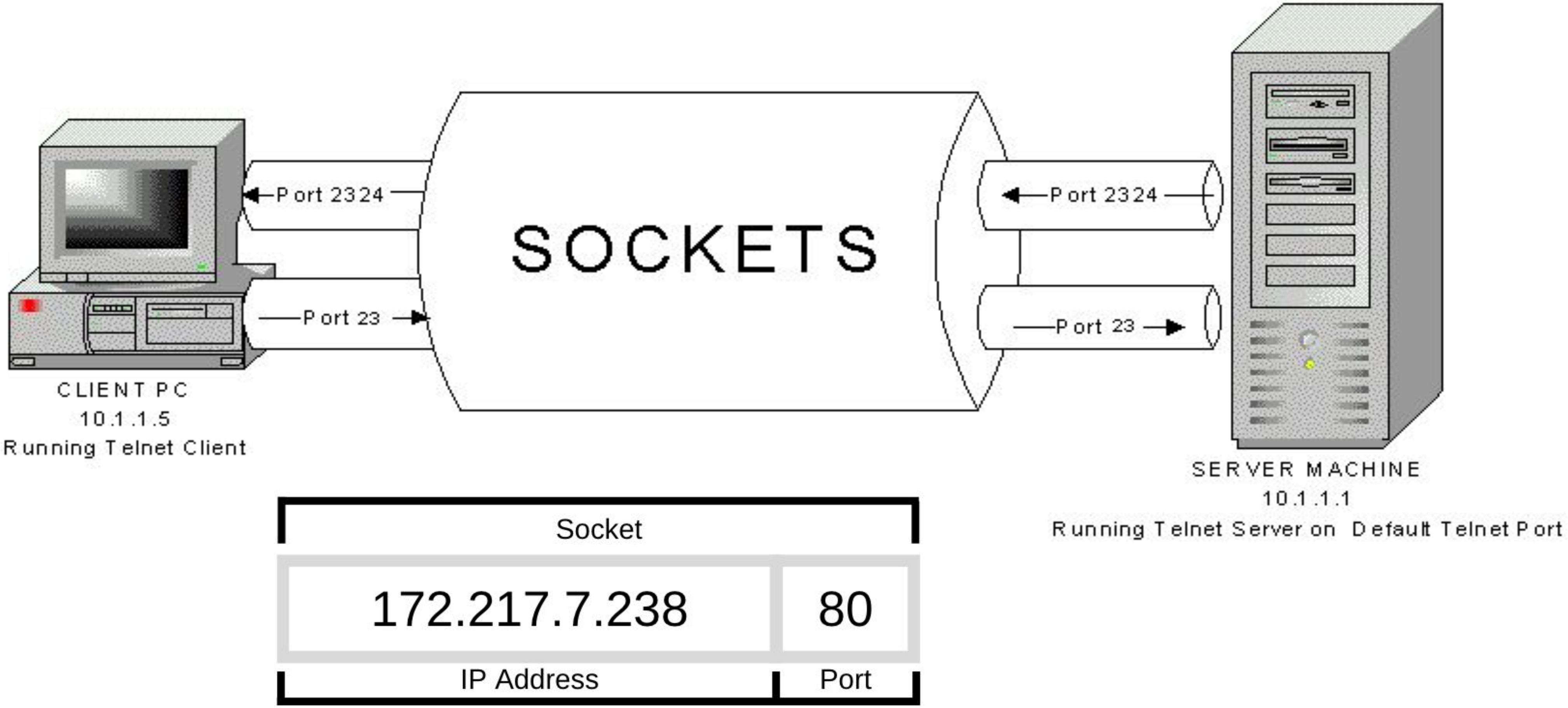
# Unit 2

- **Sockets Introduction**: Introduction, Socket Address Structures, Value-Result Arguments, Byte Ordering and Manipulation Functions.

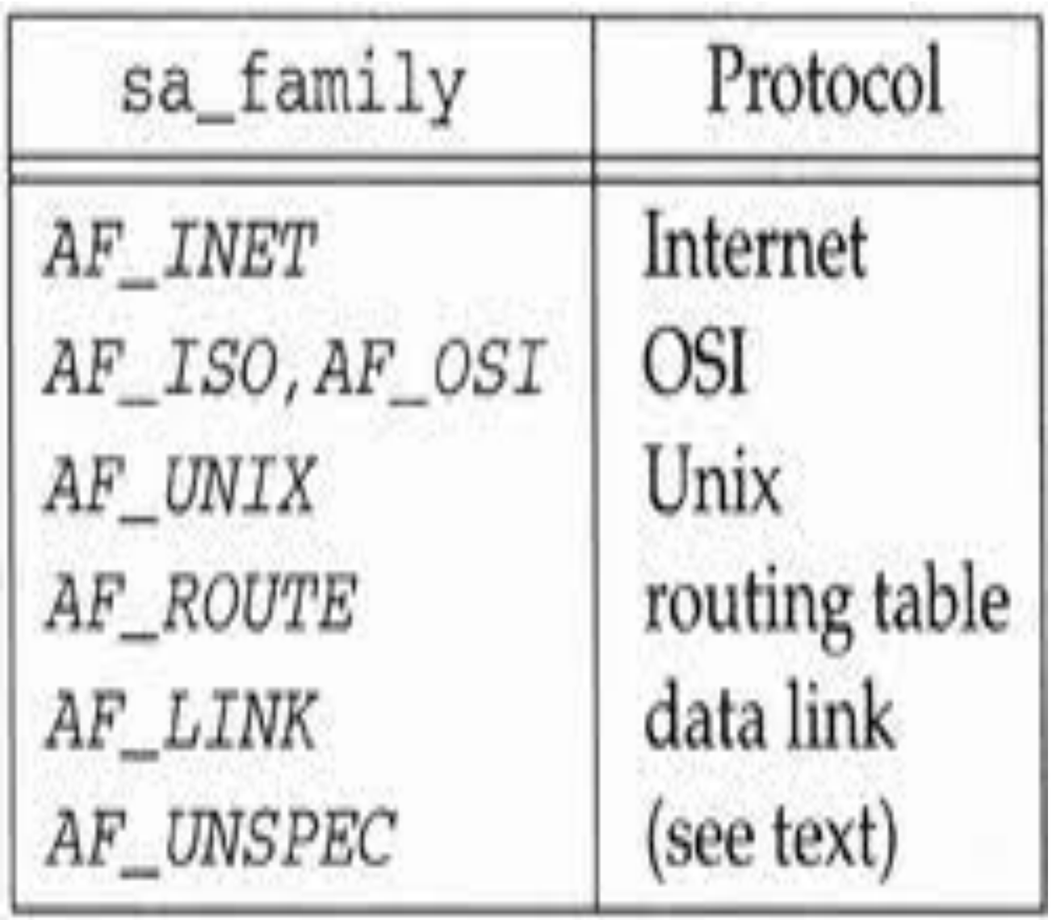
- **Elementary TCP Sockets**: socket, connect, bind, listen, accept, fork and exec, Concurrent Server design, getcsockname and getpeername functions.

- **Self learning topics:** TCP Echo Client/Server Functions.

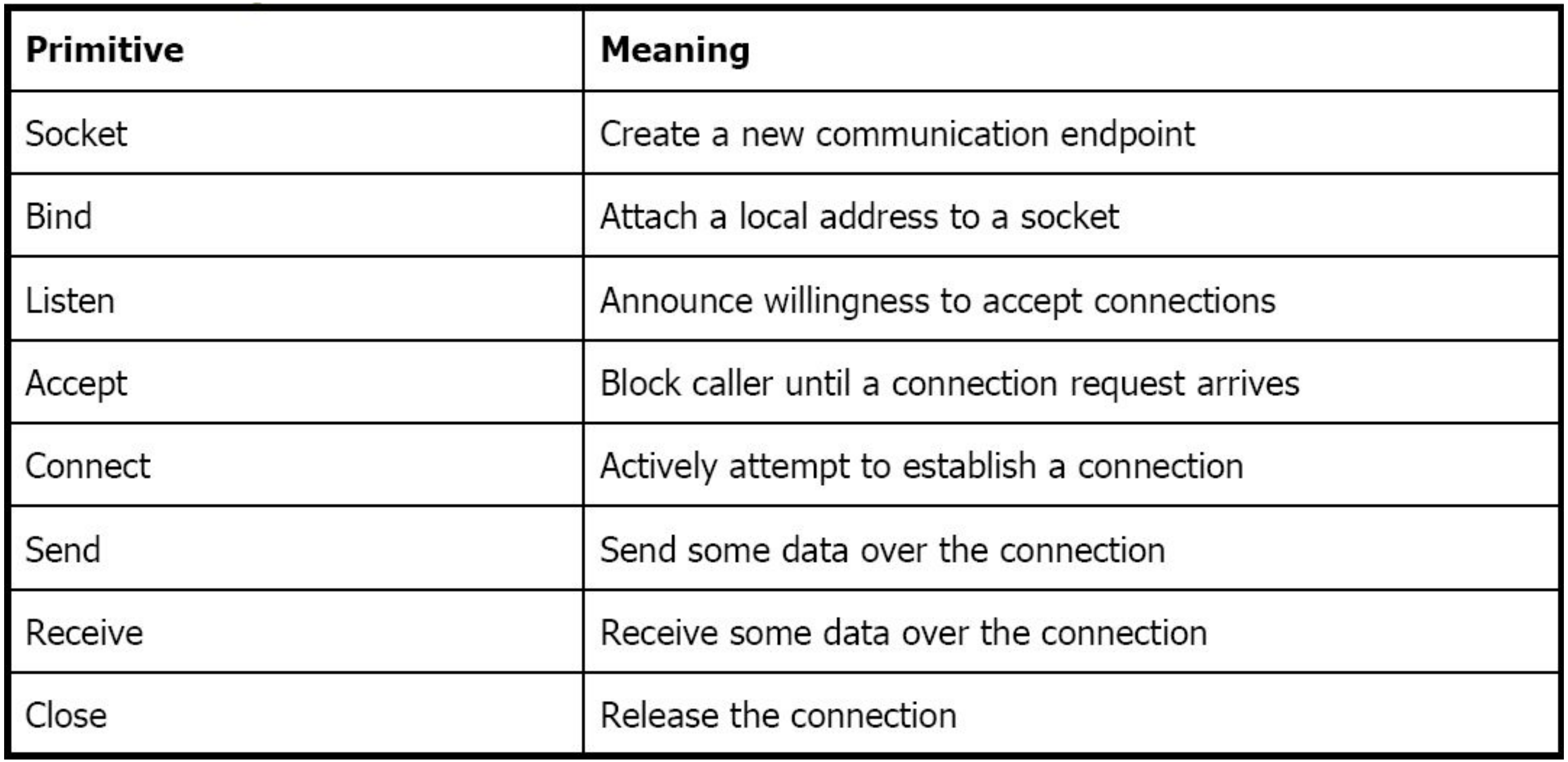
# Sockets : An end point for communication between processes across the network



# Socket Address Families



| sa_family | Protocol |
|---|---|
| AF_INET | Internet |
| AF_ISO, AF_OSI | OSI |
| AF_UNIX | Unix |
| AF_ROUTE | routing table |
| AF_LINK | data link |
| AF_UNSPEC | (see text) |

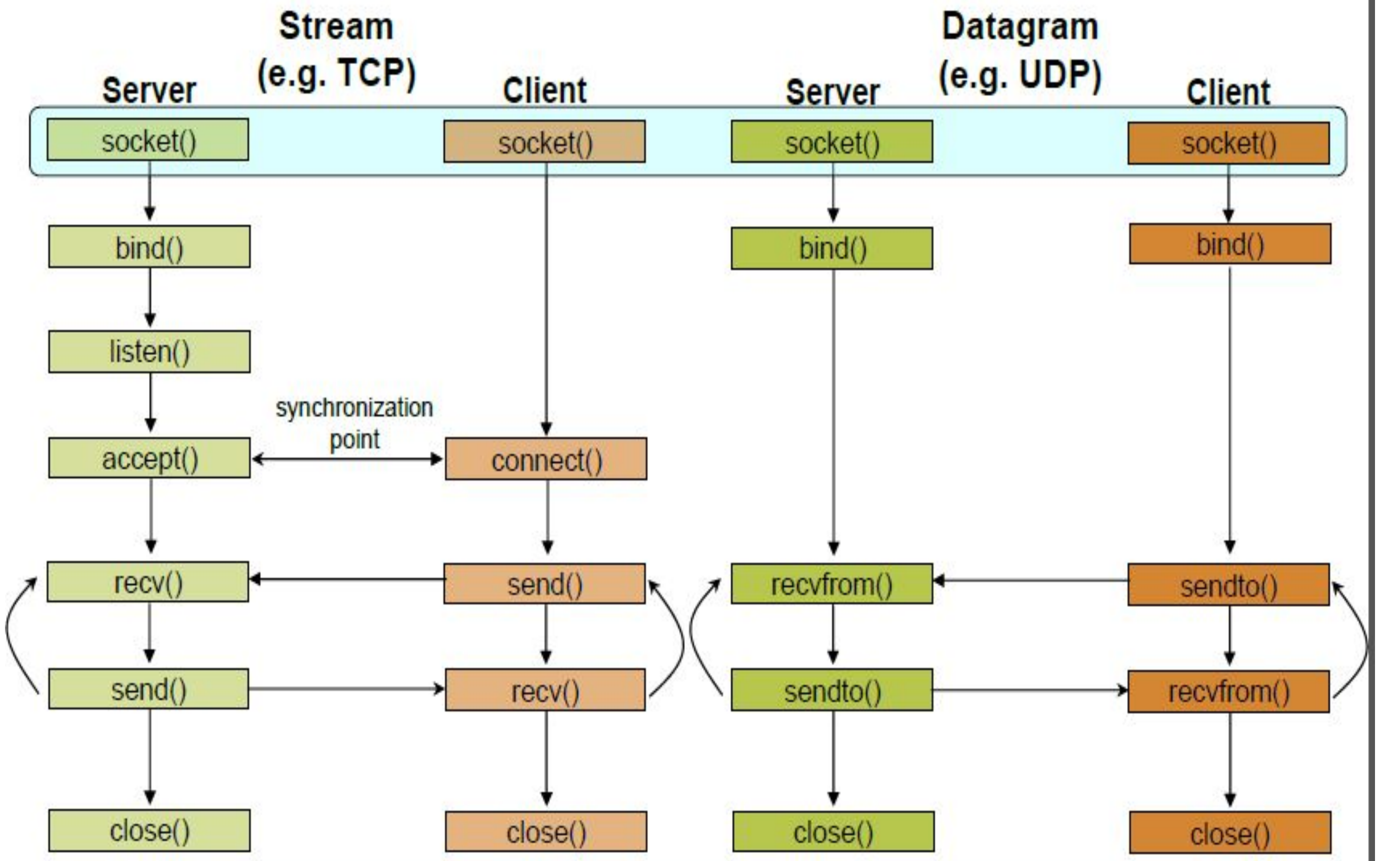


| Primitive | Meaning |
|---|---|
| Socket | Create a new communication endpoint |
| Bind | Attach a local address to a socket |
| Listen | Announce willingness to accept connections |
| Accept | Block caller until a connection request arrives |
| Connect | Actively attempt to establish a connection |
| Send | Send some data over the connection |
| Receive | Receive some data over the connection |
| Close | Release the connection |

# Client-Server communication

- Server
  - passively waits for and responds to clients
  - passive socket
- Client
  - initiates the communication
  - must know the address and the port of the server
  - active socket

# Client - Server Communication - Unix



# Socket creation in C: `socket()`

- `int sockid = socket(family, type, protocol);`
  - sockid: socket descriptor, an integer (like a file-handle)
  - family: integer, communication domain, e.g.,
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - type: communication type
    - SOCK_STREAM - reliable, 2-way, connection-based service
    - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
  - protocol: specifies protocol
    - IPPROTO_TCP IPPROTO_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1

☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

# Socket close in C: `close()`

- When finished using a socket, the socket should be closed

-
  ```
  status = close(sockid);
  ```
  - sockid: the file descriptor (socket being closed)
  - status: 0 if successful, -1 if error

- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

# Specifying Addresses

- Socket API defines a **generic** data type for addresses:

```
struct sockaddr {
   unsigned short sa_family; /* Address family (e.g. AF_INET) */
   char sa_data[14];         /* Family-specific address information */
}
```

- Particular form of the sockaddr used for **TCP/IP** addresses:

```
struct in_addr {
   unsigned long s_addr;           /* Internet address (32 bits) */
}
struct sockaddr_in {
   unsigned short sin_family;      /* Internet protocol (AF_INET) */
   unsigned short sin_port;        /* Address port (16 bits) */
   struct in_addr sin_addr;      /* Internet address (32 bits) */
   char sin_zero[8];               /* Not used */
}
```

☞ **Important**: sockaddr_in can be casted to a sockaddr

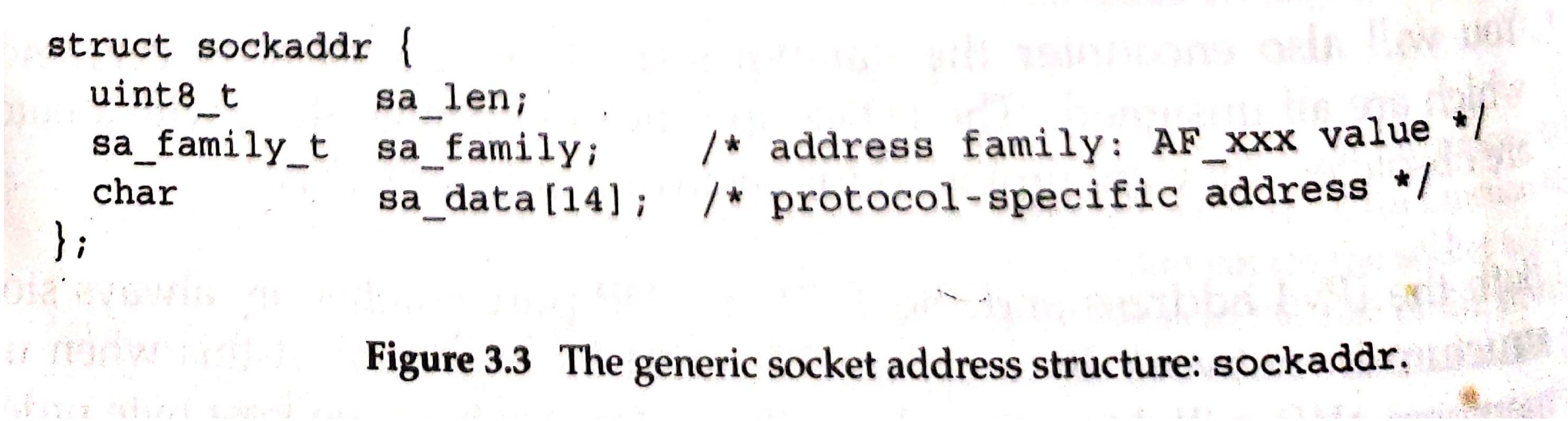
# Socket Address Structures

- Most socket functions require a **pointer** to a socket address structure as an **argument.**

- Each supported protocol suite defines **its own** socket address structure.

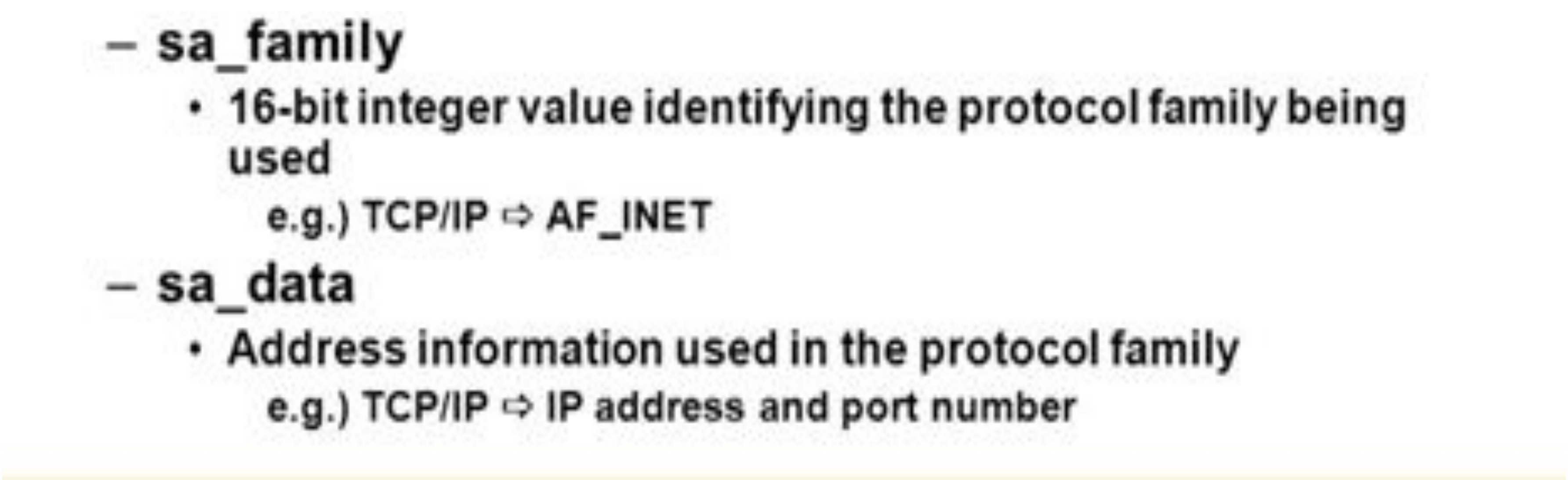
- The names of these structures **begin** with **sockaddr_** and end with a unique suffix for each protocol suite.

# Generic Socket Address Structure cont..



```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;    /* address family: AF_xxx value */
  char         sa_data[14];  /* protocol-specific address */
};
```

**Figure 3.3** The generic socket address structure: sockaddr.



- **sa_family**
  - 16-bit integer value identifying the protocol family being used

    e.g.) TCP/IP ⇨ AF_INET
- **sa_data**
  - Address information used in the protocol family

    e.g.) TCP/IP ⇨ IP address and port number

- A socket address structures is *always passed by **reference** when passed as an* **argument** to any **socket functions.**

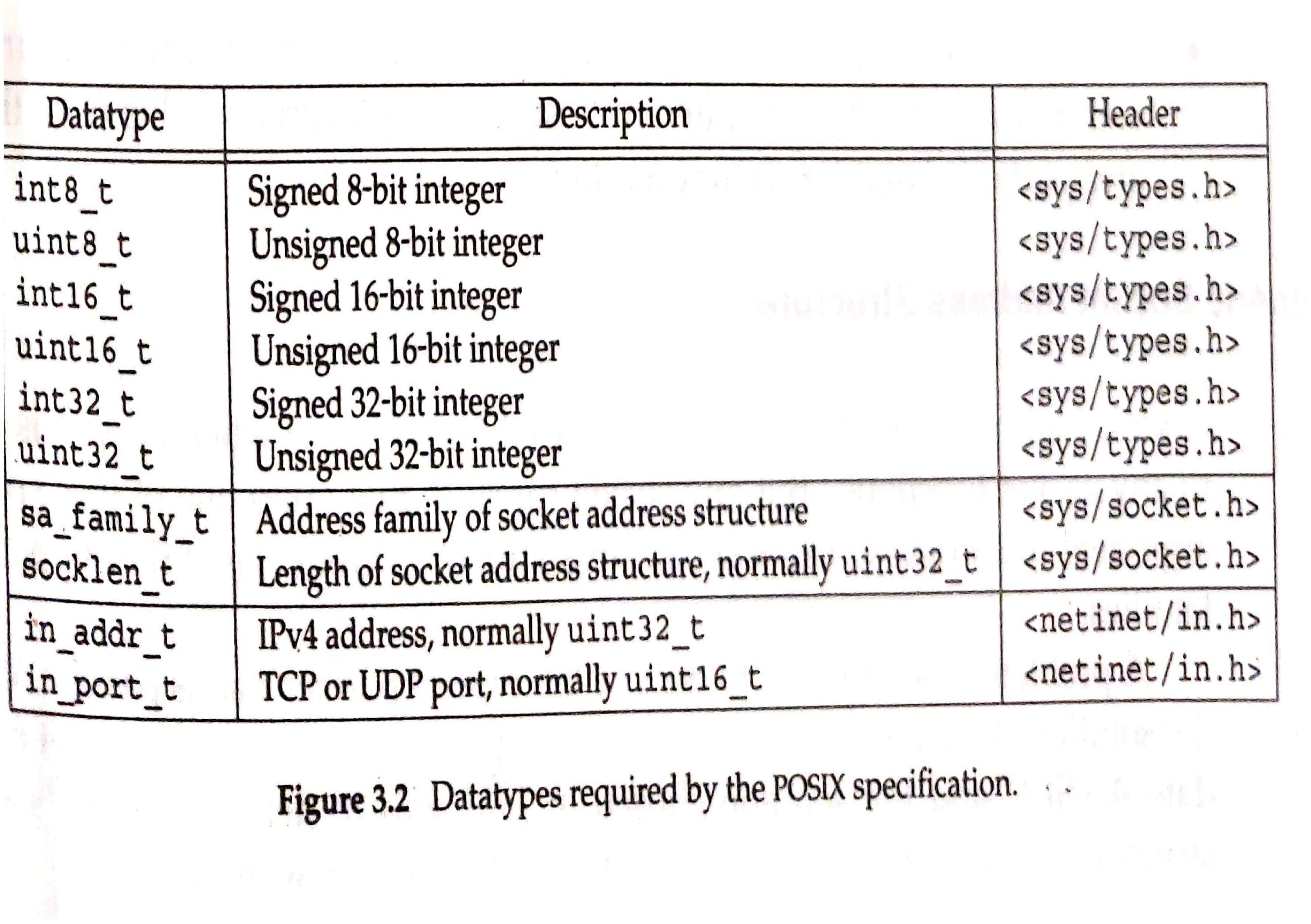
- The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function:

```
int bind(int,  struct sockaddr *,  socklen_t);
```

- This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure.

- For example,

```
struct sockaddr_in  serv;    /* IPv4 socket address structure */
/* fill in serv{} */
bind(sockfd,    (struct sockaddr *) &serv,    sizeof(serv) );
```



| Datatype | Description | Header |
|---|---|---|
| int8_t | Signed 8-bit integer | <sys/types.h> |
| uint8_t | Unsigned 8-bit integer | <sys/types.h> |
| int16_t | Signed 16-bit integer | <sys/types.h> |
| uint16_t | Unsigned 16-bit integer | <sys/types.h> |
| int32_t | Signed 32-bit integer | <sys/types.h> |
| uint32_t | Unsigned 32-bit integer | <sys/types.h> |
| sa_family_t | Address family of socket address structure | <sys/socket.h> |
| socklen_t | Length of socket address structure, normally uint32_t | <sys/socket.h> |
| in_addr_t | IPv4 address, normally uint32_t | <netinet/in.h> |
| in_port_t | TCP or UDP port, normally uint16_t | <netinet/in.h> |

**Figure 3.2** Datatypes required by the POSIX specification.

## IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address stru ture," is named sockaddr_in and is defined by including the <netinet/in.: header. Figure 3.1 shows the POSIX definition.

```
struct in_addr {
  in_addr_t   s_addr;              /* 32-bit IPv4 address */
                                   /* network byte ordered */

};

struct sockaddr_in {
  uint8_t          sin_len;       /* length of structure (16) */
  sa_family_t      sin_family;    /* AF_INET */
  in_port_t        sin_port;      /* 16-bit TCP or UDP port number */
                                  /* network byte ordered */
  struct in_addr   sin_addr;      /* 32-bit IPv4 address */
                                  /* network byte ordered */
  char             sin_zero[8];   /* unused */
};
```

Figure 3.1 The Internet (IPv4) socket address structure: sockaddr_in.

## IPv6 Socket Address Structure

The IPv6 socket address is defined by including the <netinet/in.h> header, and we show it in Figure 3.4.

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t          sin6_len;      /* length of this struct (28) */
  sa_family_t      sin6_family;   /* AF_INET6 */
  in_port_t        sin6_port;     /* transport layer port# */
                                  /* network byte ordered */

  uint32_t         sin6_flowinfo; /* flow information, undefined */
  struct in6_addr  sin6_addr;     /* IPv6 address */
                                  /* network byte ordered */

  uint32_t         sin6_scope_id; /* set of interfaces for a scope */
};
```

**Figure 3.4** IPv6 socket address structure: sockaddr_in6.

# New Generic Socket Address Structure

- A new generic socket address structure was defined as part of the IPv6 sockets API, **to overcome some of the shortcomings of the existing struct sockaddr**.

- Unlike the struct sockaddr, the **new struct sockaddr_storage** is ***large enough to hold any socket address type supported by the system.***

- The sockaddr_storage structure is defined by including the **<netinet/in.h> header**

# New Generic Socket Address Structure (cont..)

```
struct sockaddr_storage {

   uint8_t       ss_len;   /* length of this struct (implementation dependent) */

   sa_family_t  ss_family; /* address family: AF_xxx value */

   /* implementation-dependent elements to provide:

      * a) alignment sufficient to fulfill the alignment requirements of  all socket
   address types  that the system   supports.
      * b) enough storage to hold any type of socket address that the  system
   supports.
*/
};
```

- The sockaddr_storage type provides a generic socket address structure that is **different from struct sockaddr in two ways**:

a.) If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.

b.)  The sockaddr_storage is large enough to contain any socket address structure that the system supports.

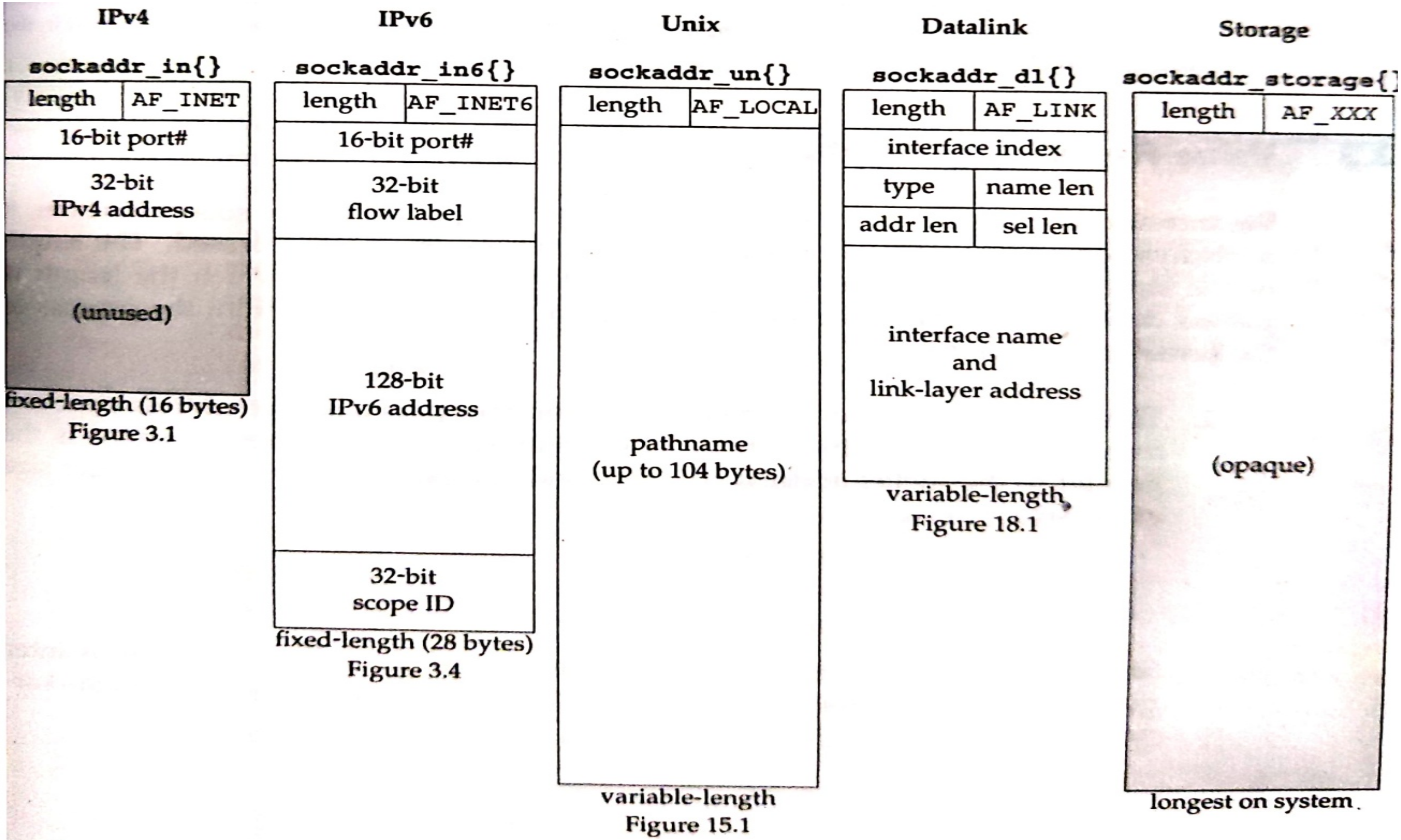
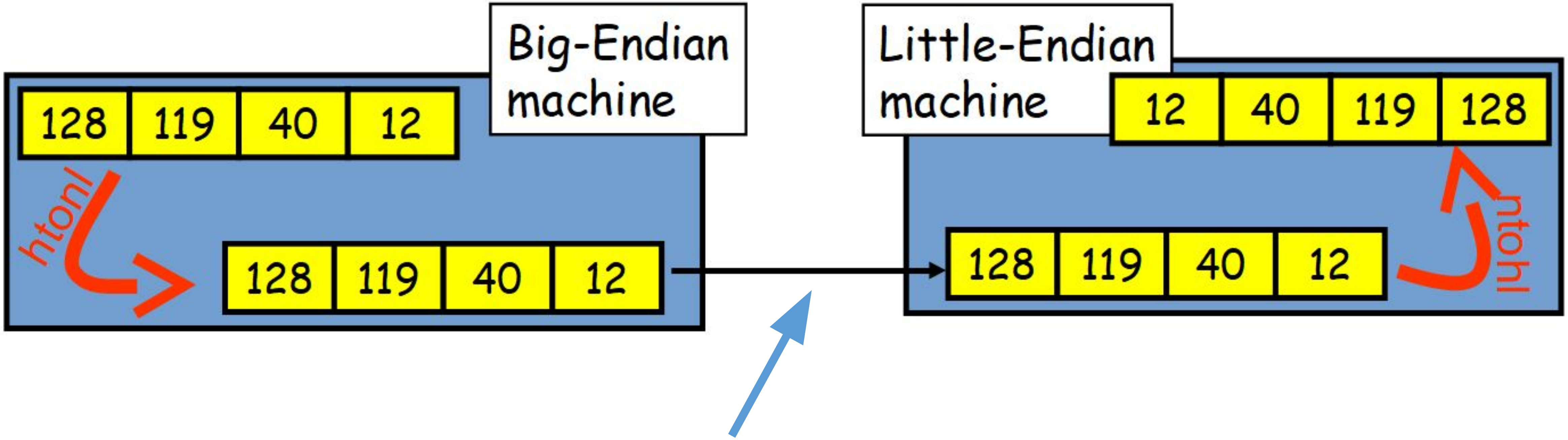


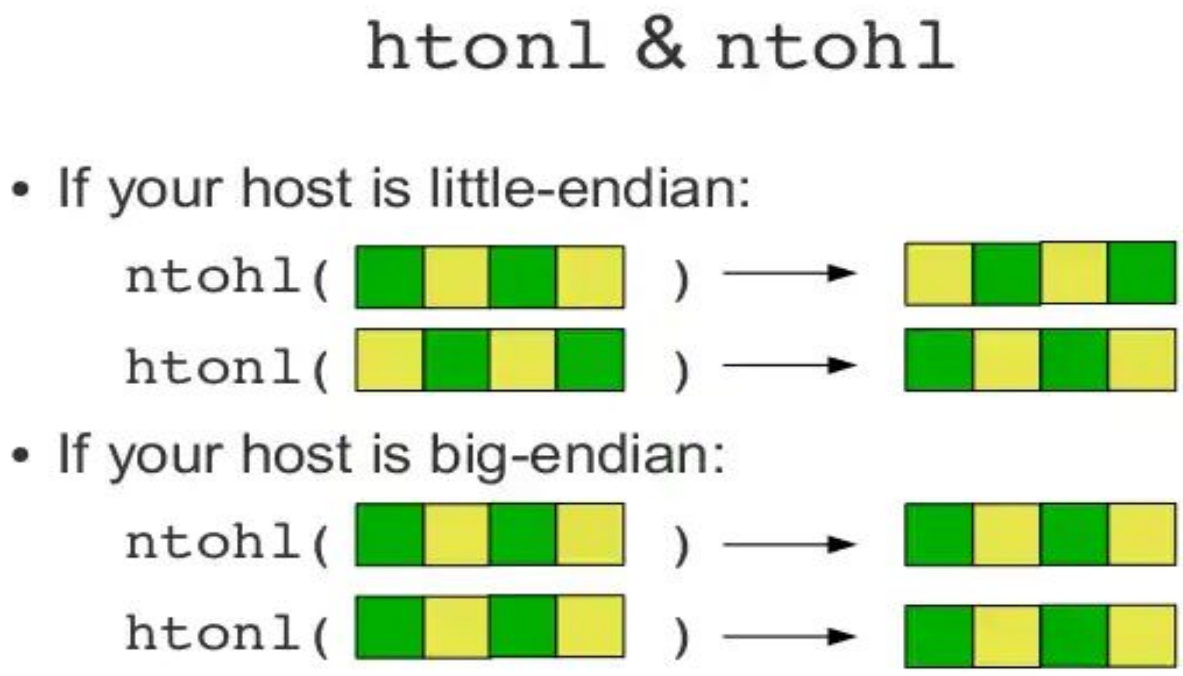
**Figure 3.6** Comparison of various socket address structures.

# Network and Host Byte Ordering



Network Byte Order – Big-Endian

# Byte Order conversion functions



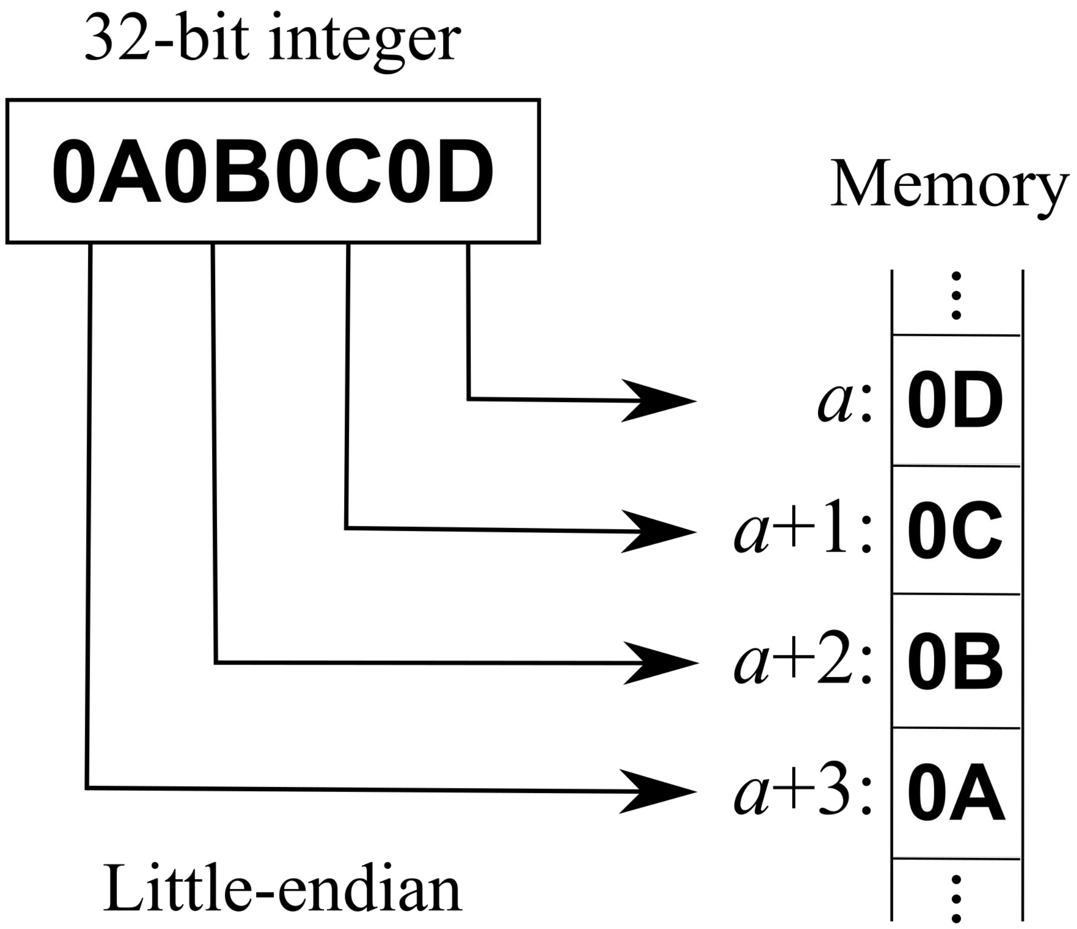
htonl & ntohl

- If your host is little-endian:

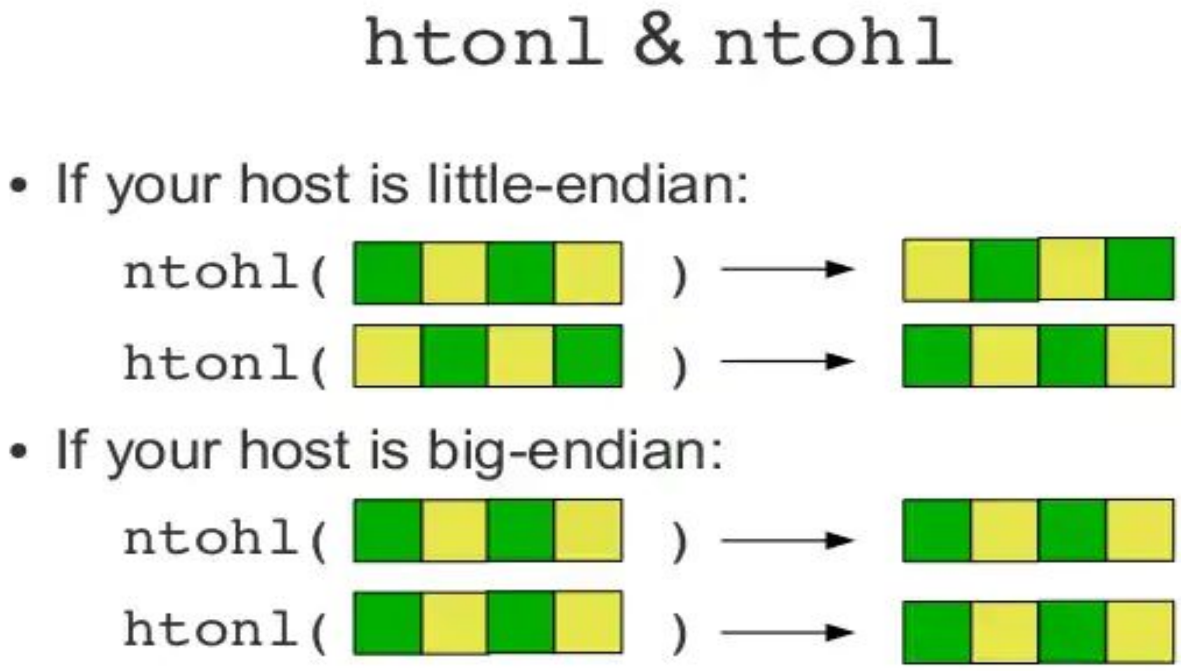
```
ntohl( 🟩🟨🟩🟨 ) → 🟨🟩🟨🟩
htonl( 🟨🟩🟨🟩 ) → 🟩🟨🟩🟨
```

- If your host is big-endian:

```
ntohl( 🟩🟨🟩🟨 ) → 🟩🟨🟩🟨
htonl( 🟩🟨🟩🟨 ) → 🟩🟨🟩🟨
```



# Byte Order conversion functions



## htonl & ntohl

- If your host is little-endian:

ntohl( 🟩🟨🟩🟨 ) → 🟨🟩🟨🟩

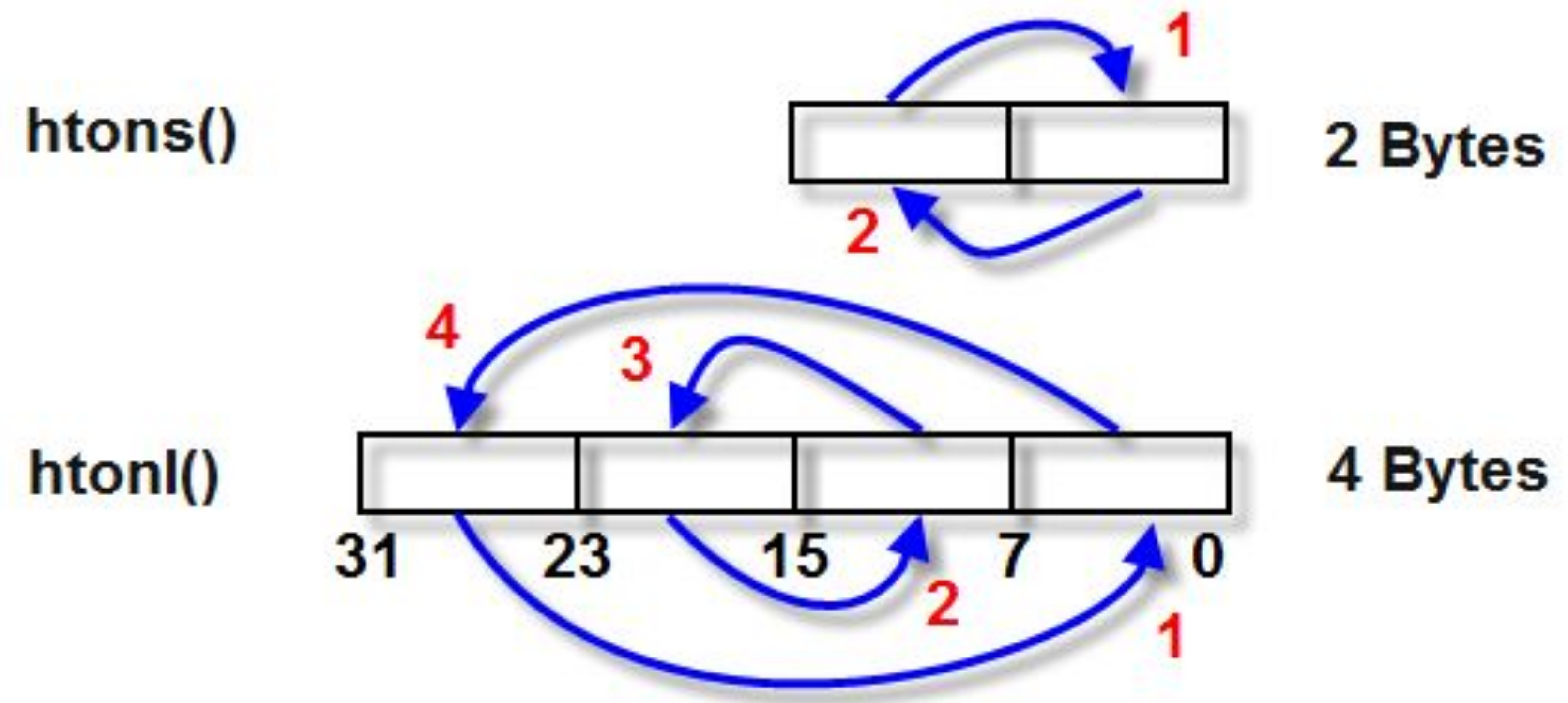
htonl( 🟨🟩🟨🟩 ) → 🟩🟨🟩🟨

- If your host is big-endian:

ntohl( 🟩🟨🟩🟨 ) → 🟩🟨🟩🟨

htonl( 🟩🟨🟩🟨 ) → 🟩🟨🟩🟨

# Byte Order conversion functions



# Value-Result Arguments

- when a socket address structure is passed to any socket function, it is always passed by reference.
- That is, a pointer to the structure is passed.

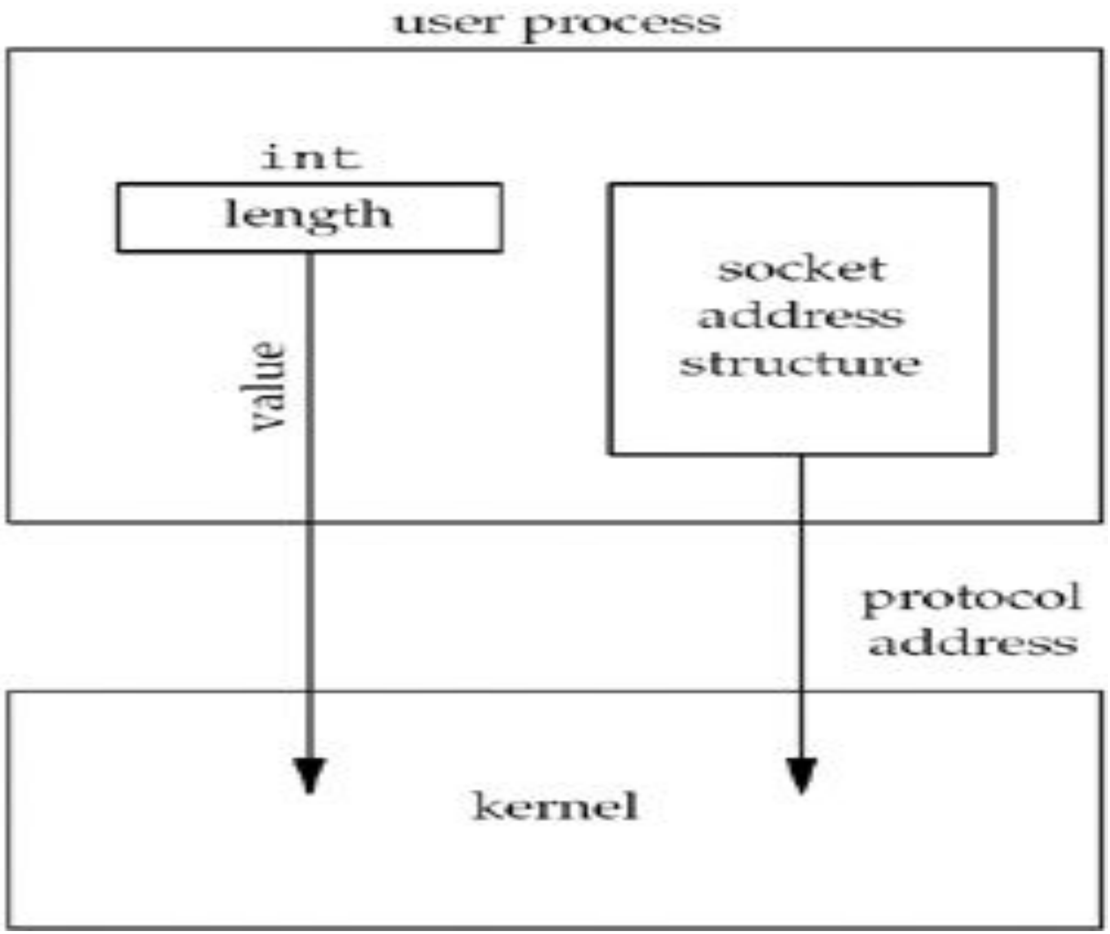
- **The length of the structure is also passed as an argument.** But the way in which the length is passed depends on **which direction the structure is being passed**: **from the process to the kernel, or vice versa.**

1. Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel.

   **One argument** to these three functions is the **pointer to the socket address structure** and **another argument** is the **integer size of the structure**, as in

```
struct sockaddr_in  serv;

/* fill in serv{} */

connect ( sockfd,      (SA *) &serv,      sizeof(serv)  );
```

- Since the kernel is passed both the **pointer** and the **size of what the pointer points to**, it knows *exactly how much data to copy from the process into the kernel*. Figure 3.7 shows this scenario.



- Four functions, **accept, recvfrom, getsockname, and getpeername,** pass a *socket address structure* from the **kernel** to the **process**, the reverse direction from the previous scenario.

- Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

```
struct  sockaddr_un  cli;            /* Unix domain */
socklen_t  len;

len = sizeof(cli);                /* len is a value */
getpeername( unixfd,     (SA *) &cli,    &len );
/* len may have changed */
```

- The reason that the size changes from an **integer to be a pointer** to an **integer** is because the size is both a ***value*** *when the function is called (****it tells the kernel*** **the size of the structure so that the kernel does not write past the end of the structure when filling it in**) and

- a ***result*** *when the* ***function returns*** *(it* ***tells the*** **process how much information the kernel actually stored in the structure**).

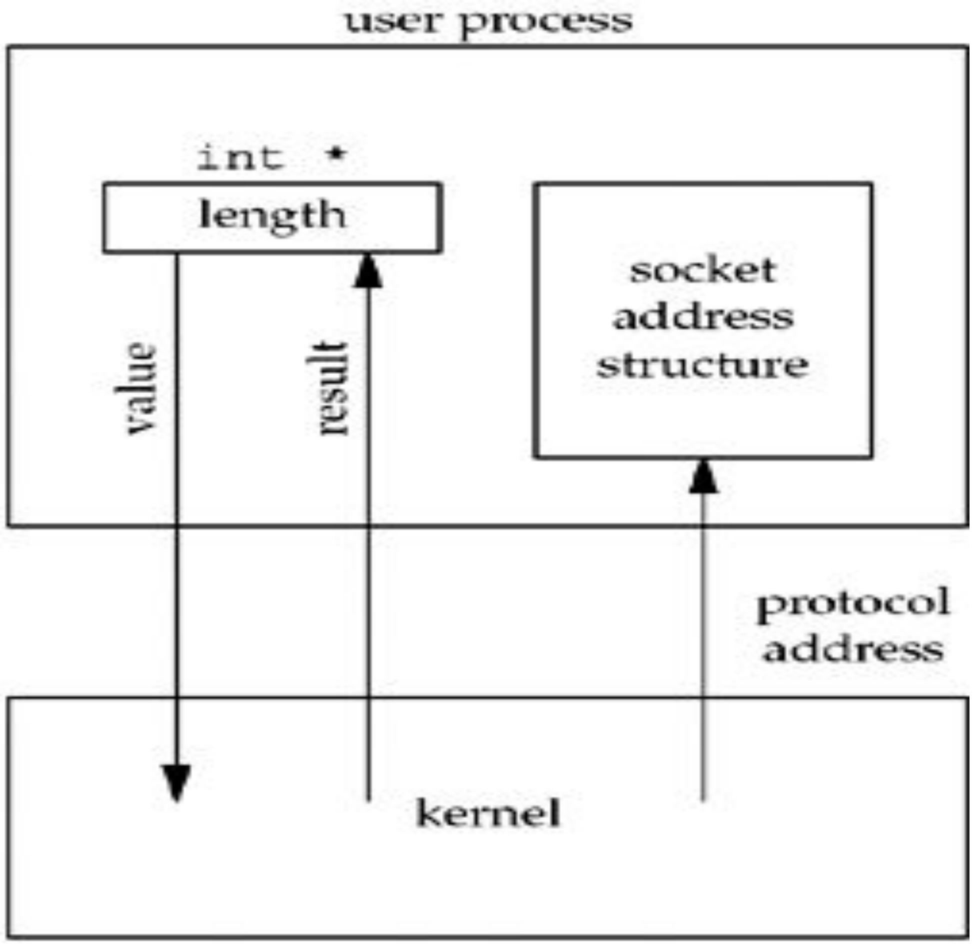
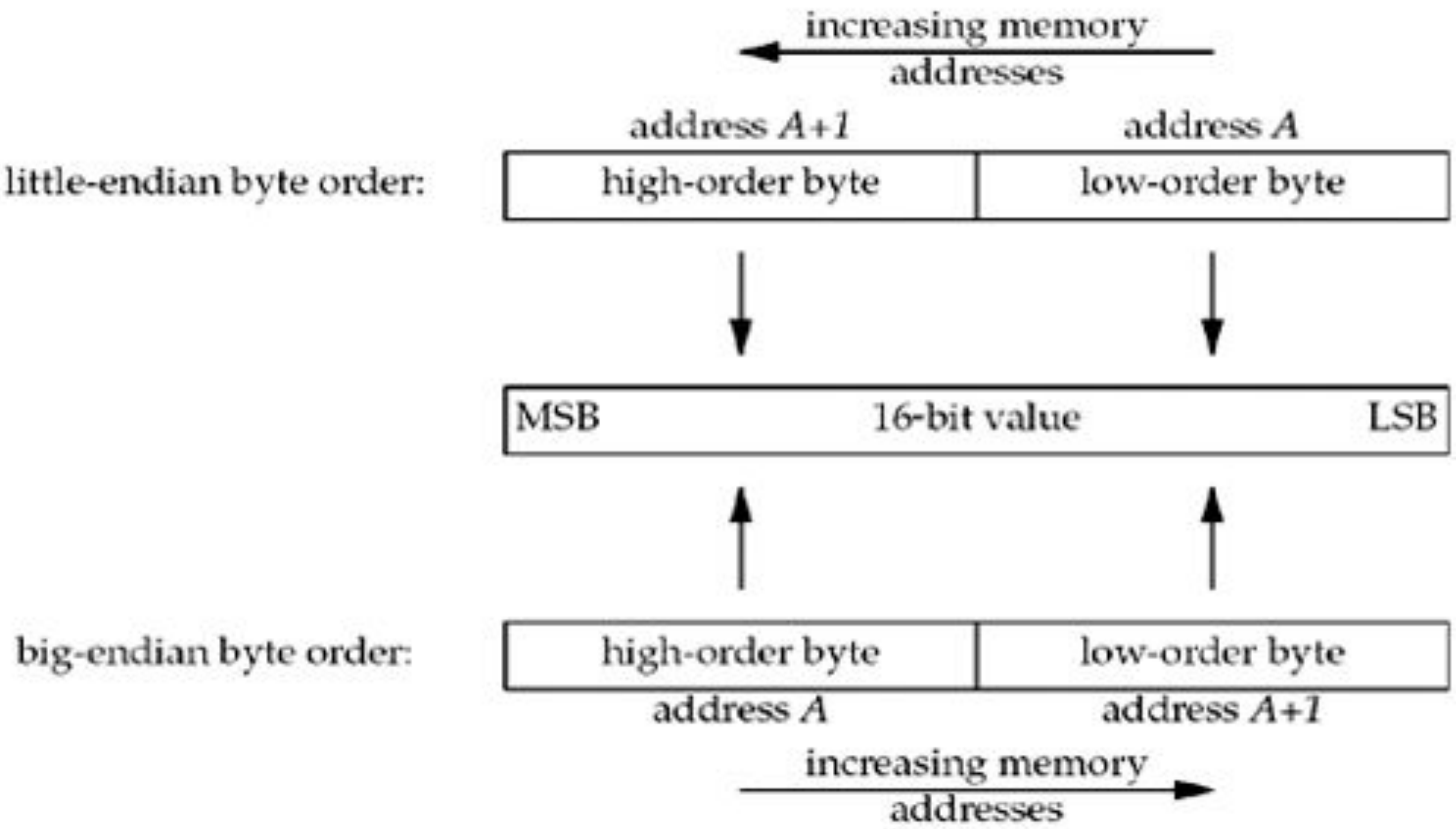


**Figure 3.8. Socket address structure passed from kernel to process.**

# Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes.
- There are two ways to store the two bytes in memory:
-  with the **low-order byte** at the **starting address**, known as ***little-endian byte order***
- *with the **high-order byte** at the **starting address,** known as **big-endian byte order**.*



- We must deal with these byte ordering differences as network programmers because networking protocols must specify a ***network byte order***.

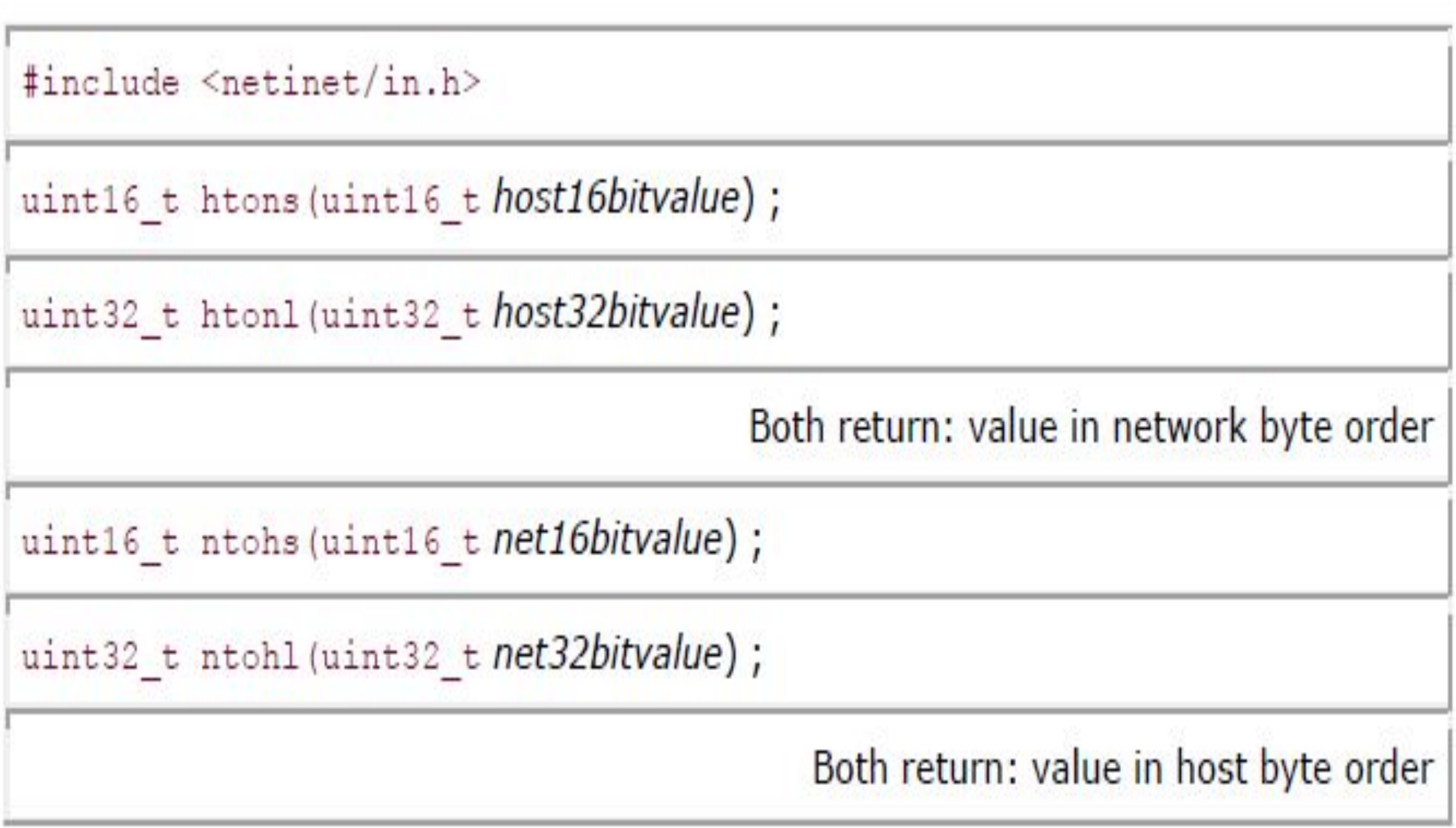
- *For example, in a TCP* segment, there is a **16-bit port number** and a **32-bit IPv4 address.**
- The **sending protocol stack** and the **receiving protocol stack** must **agree** on the **order** in which the bytes of these multibyte fields will be transmitted.

- The **Internet protocols** use **big-endian byte ordering** for these multibyte integers.

- In theory, an implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail.
- But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in **network byte order.**

- Our concern is therefore **converting between host byte order and network byte order.** We use the following **four functions to convert between these two byte orders**.

**Converting between host byte order and network byte order.** We use the following **four functions to convert between these two byte orders**



```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue) ;

uint32_t htonl(uint32_t host32bitvalue) ;

                                   Both return: value in network byte order

uint16_t ntohs(uint16_t net16bitvalue) ;

uint32_t ntohl(uint32_t net32bitvalue) ;

                                      Both return: value in host byte order
```

# Byte Manipulation Functions

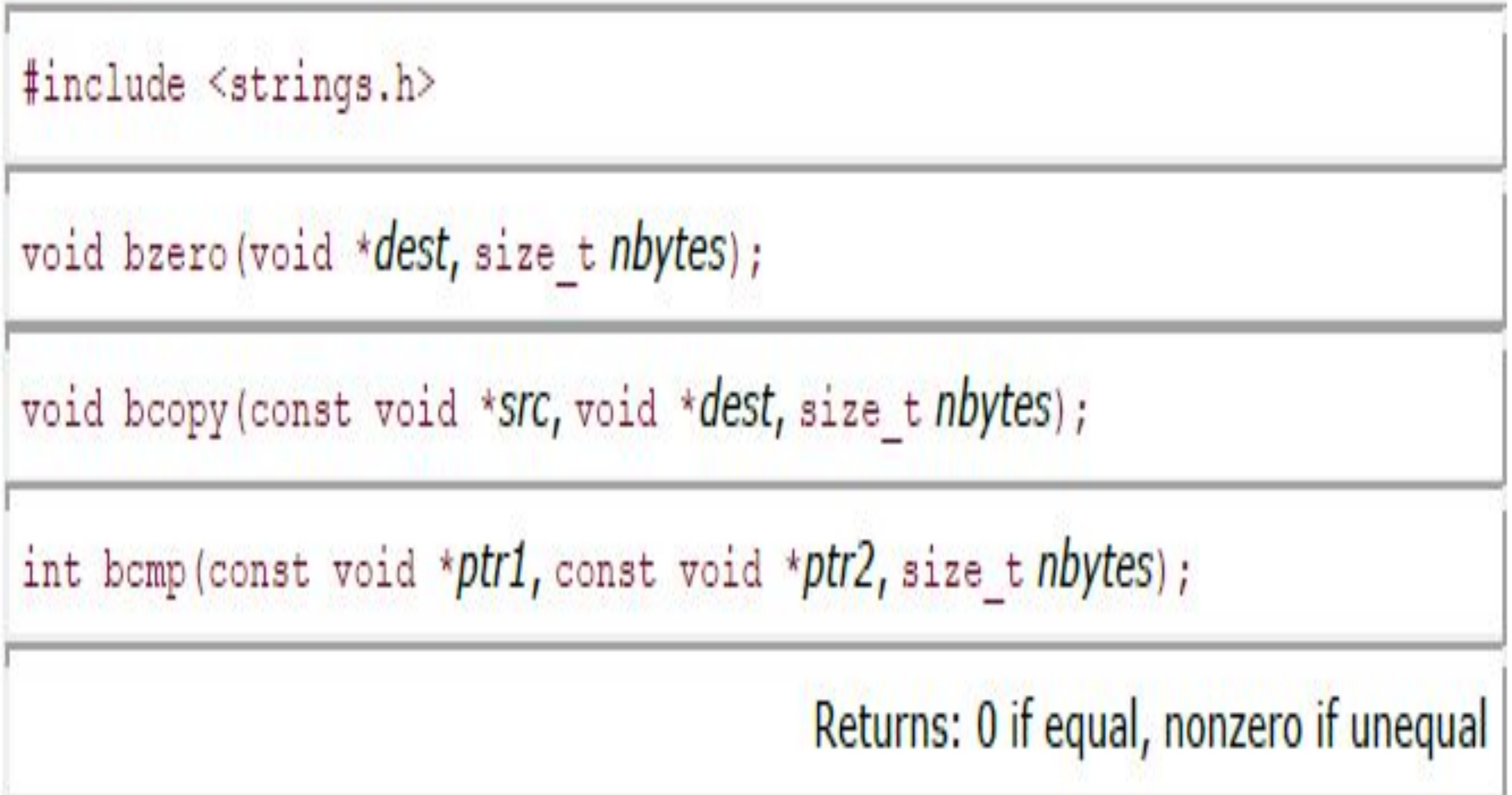
- There are **two groups** of functions that operate on **multibyte fields**, *without interpreting the data*, and *without assuming that the data is a null-terminated C string.*

- We need these types of functions when dealing with **socket address structures** because we need to ***manipulate fields such as IP addresses***, which can contain ***bytes of 0***, but ***are not C character strings***.

- The functions beginning with **str (for string),** defined by including the **<string.h>** header, deal with ***null-terminated C character strings.***

- The first group of functions, whose names begin with **b (for byte),** are from *4.2BSD* and are still provided by almost any system that **supports the socket functions.**

- The second group of functions, whose names begin with **mem** (**for memory**), are from the ***ANSI C standard*** and are provided with any ***system that supports an*** **ANSI C library.**

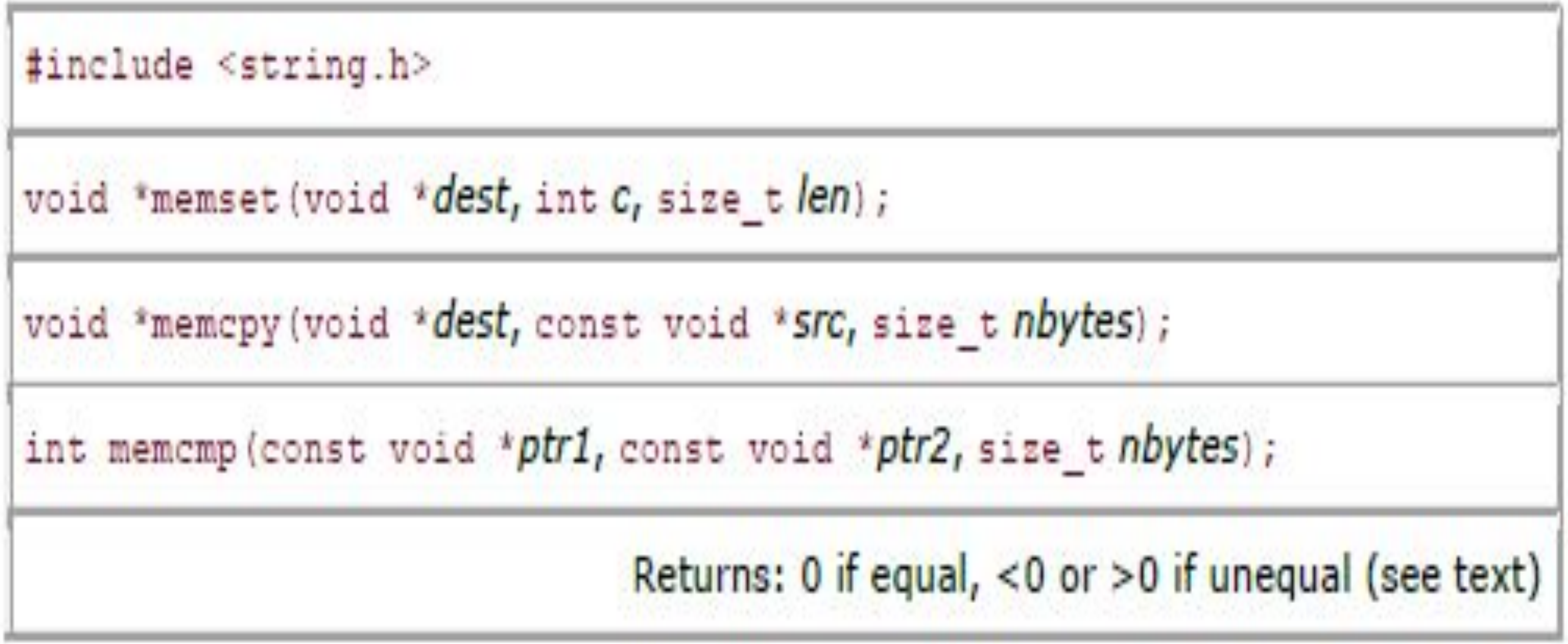


```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                                        Returns: 0 if equal, nonzero if unequal
```

- **bzero** sets the **specified number of bytes to 0 in the destination**. We often use this function to **initialize a socket address structure to 0**.

- **bcopy** ***moves the specified number of bytes from the source to the destination***.

- **bcmp** ***compares two arbitrary byte strings***. The return value is **zero** if the **two byte strings are identical**; otherwise, it is **nonzero.**



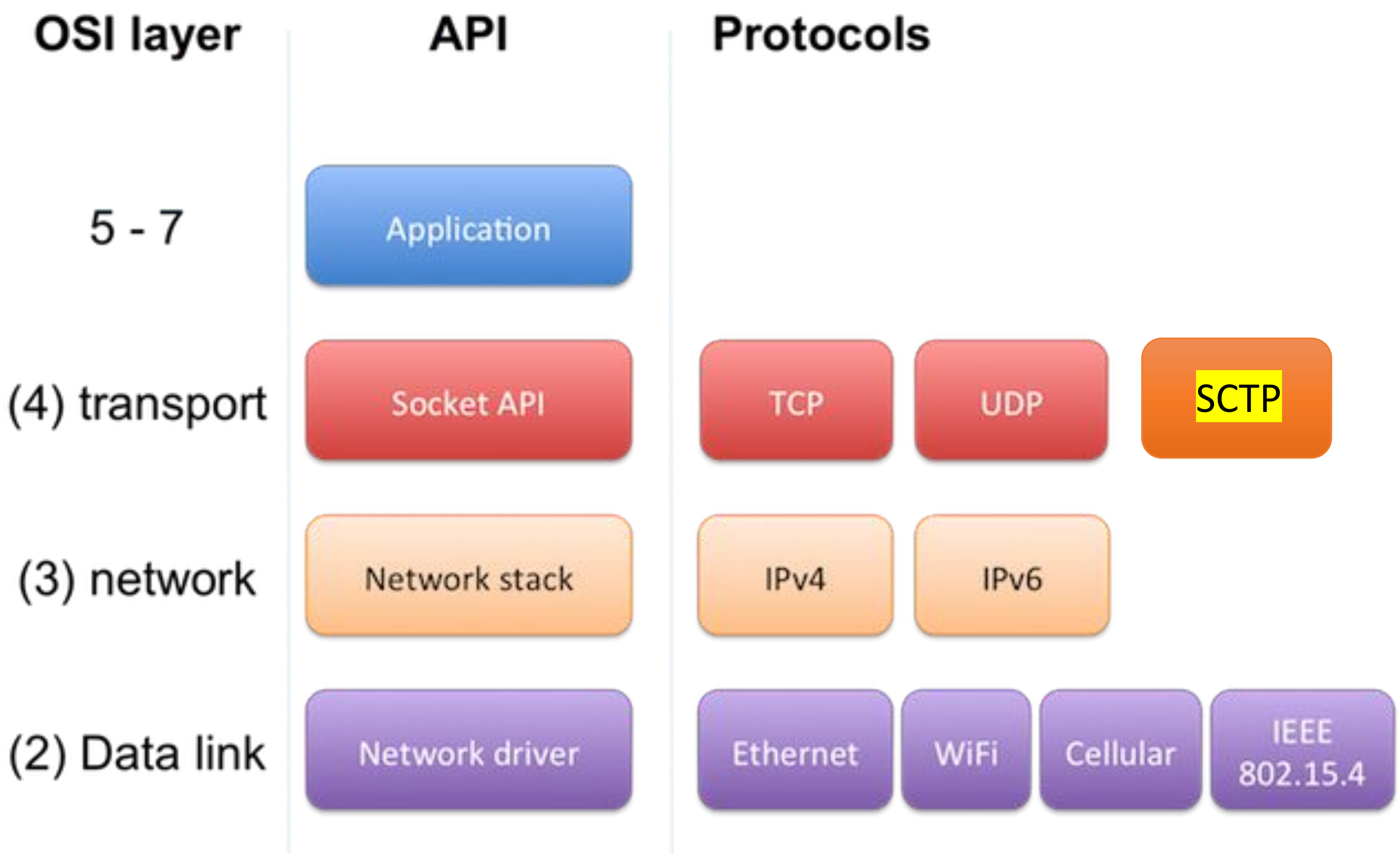
```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                                   Returns: 0 if equal, <0 or >0 if unequal (see text)
```

**memse**t sets the specified number of bytes to the **value *c*** *in the destination.*

***memcpy*** *is* similar to bcopy, but the order of the two pointer arguments is swapped. Bcopy correctly handles overlapping fields, while the behavior of memcpy is **undefined if the source and destination overlap**.

***memcmp compares two arbitrary byte strings and returns 0 if they are identical***. If not identical, the return value is either greater than 0 or less than 0,

# IPV4 and IPV6



| OSI layer | API | Protocols | | | |
|---|---|---|---|---|---|
| 5 - 7 | Application | | | | |
| (4) transport | Socket API | TCP | UDP | SCTP | |
| (3) network | Network stack | IPv4 | IPv6 | | |
| (2) Data link | Network driver | Ethernet | WiFi | Cellular | IEEE 802.15.4 |

# APIs – For concurrent Server

Socket

Bind

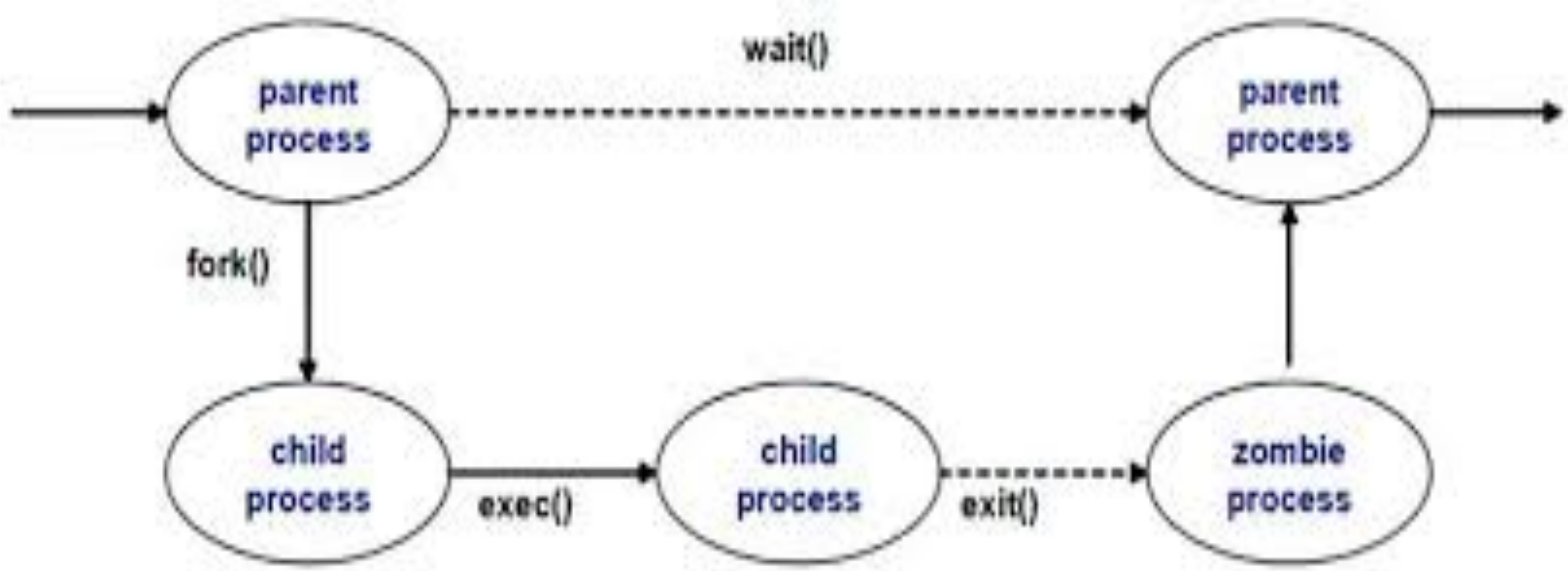
Listen

Accept

fork and exec

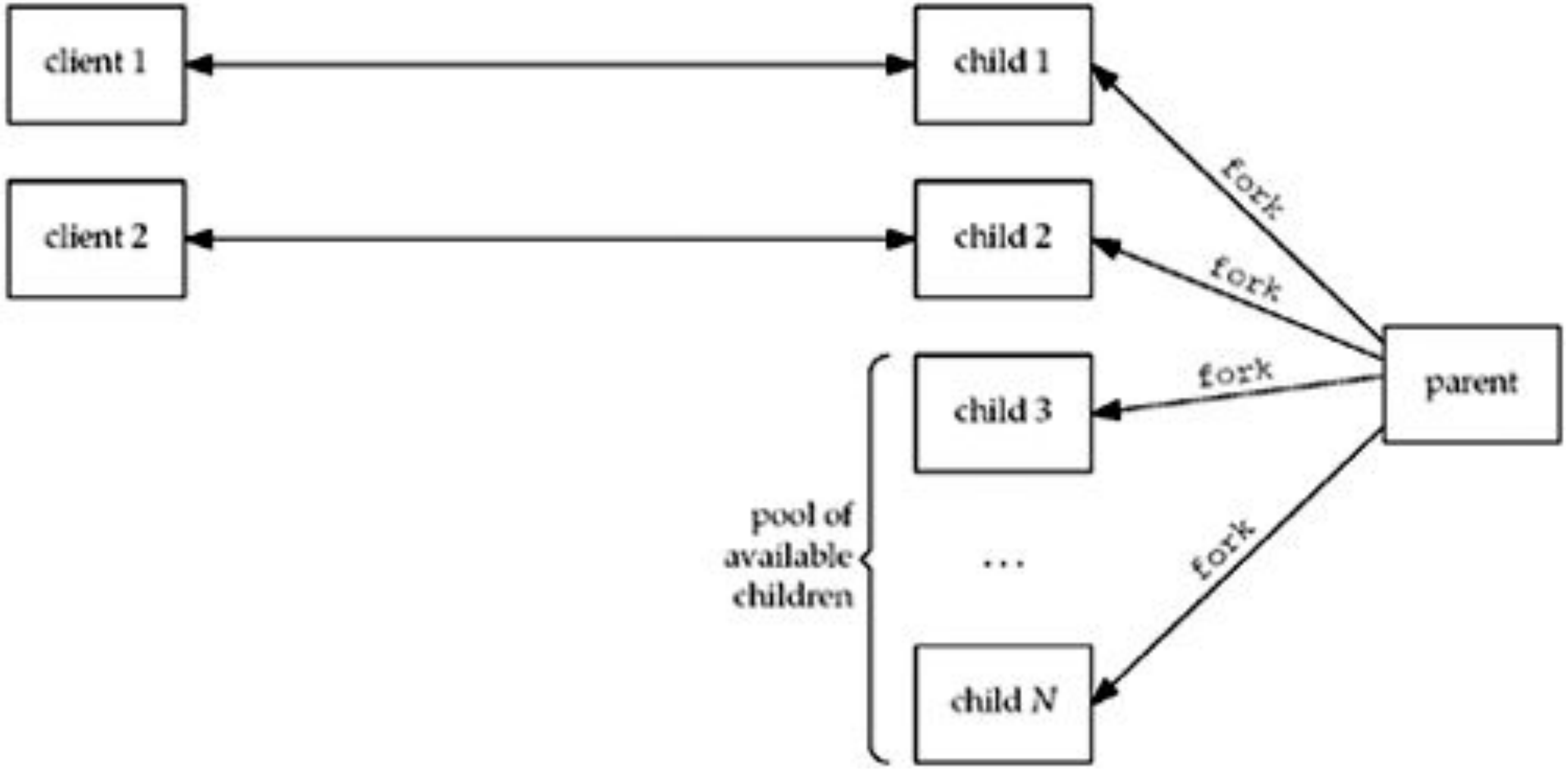
# APIs – For concurrent Server

## fork and exec

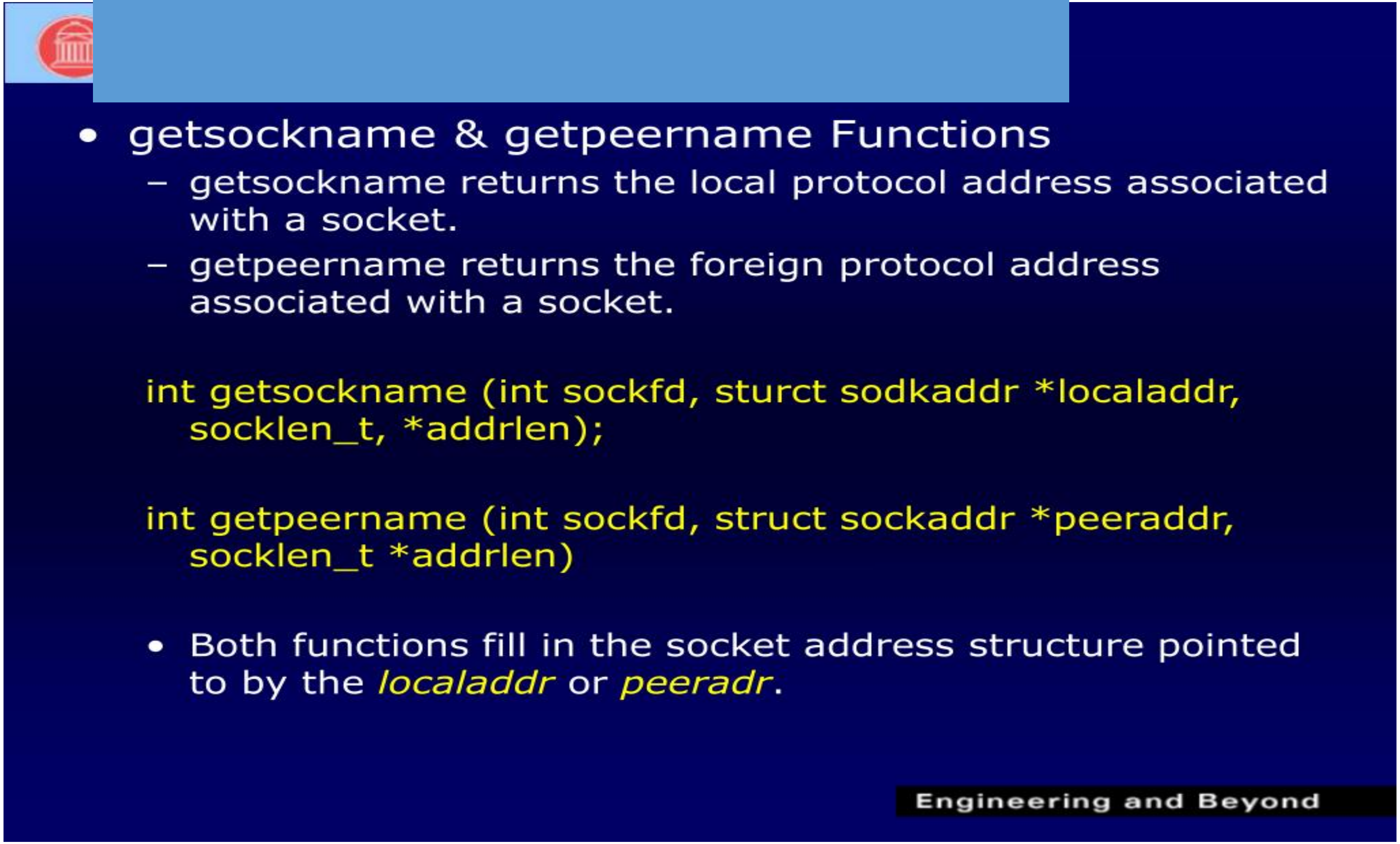


# APIs – For concurrent Server

fork and exec



# APIs – getsockname and getpeername



- getsockname & getpeername Functions
  - getsockname returns the local protocol address associated with a socket.
  - getpeername returns the foreign protocol address associated with a socket.

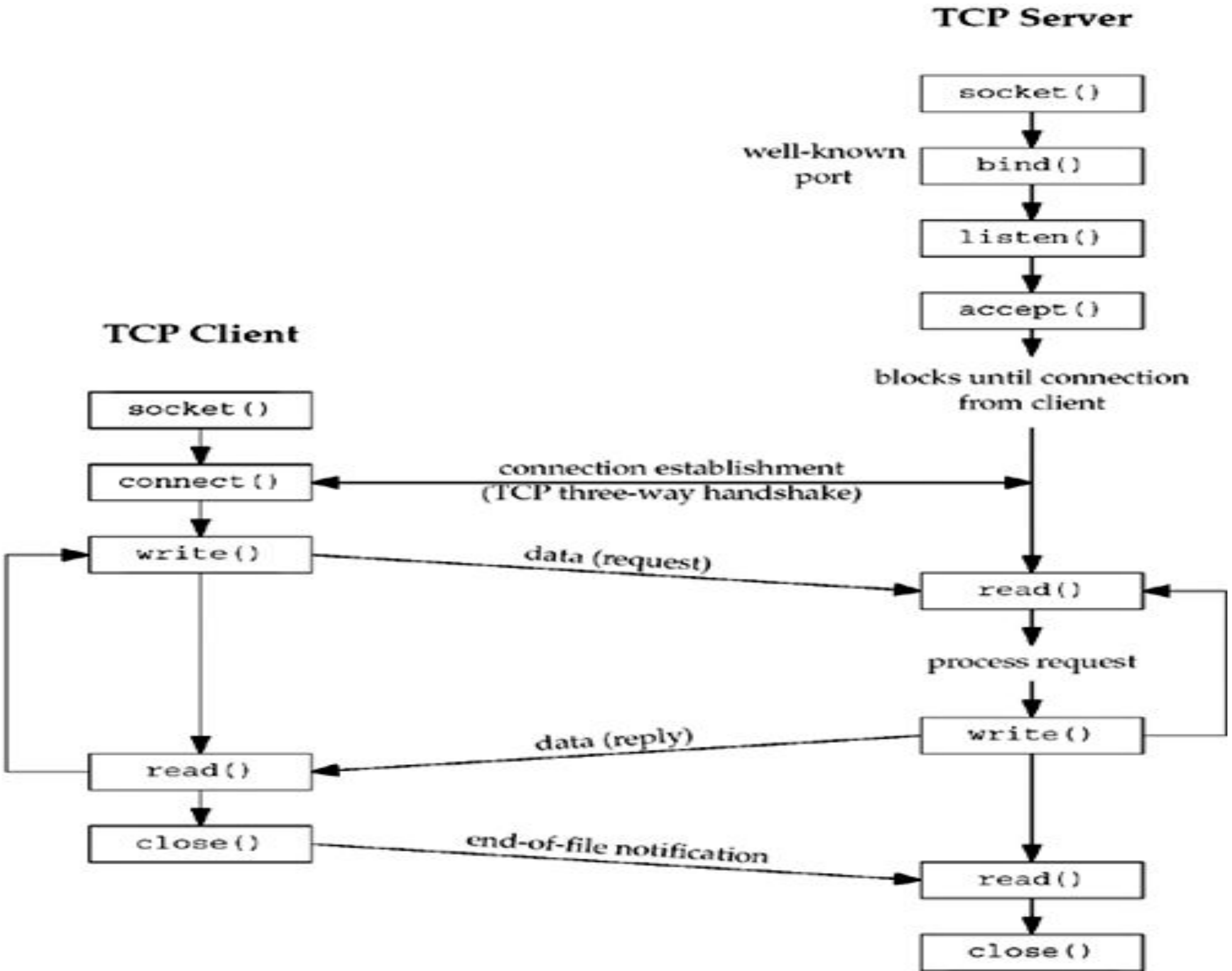
```
int getsockname (int sockfd, sturct sodkaddr *localaddr,
   socklen_t, *addrlen);

int getpeername (int sockfd, struct sockaddr *peeraddr,
   socklen_t *addrlen)
```

- Both functions fill in the socket address structure pointed to by the *localaddr* or *peeradr*.

Engineering and Beyond

# socket' Function

- To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (**TCP using IPv4**, **UDP using IPv6**, **Unix domain stream protocol,** etc.).


TCP Server
socket()
well-known port
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
connection establishment (TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
            Returns: non-negative descriptor if OK, -1 on error
```

***family*** *specifies the* ***protocol family*** *and is one of the constants shown in Figure 4.2.*
This argument is often referred to as ***domain instead of family.***

*The* ***socket type*** *is one* of the **constants** shown in Figure 4.3

. The *protocol argument to the socket function* should be set to the specific protocol type found in Figure 4.4, or 0 to select the system's default for the given combination of ***family and type***.

# Figure 4.2. Protocol *family constants for socket function.*

| Family | Description |
|---|---|
| AF_INET | IPV4 Protocols |
| AF_INET6 | IPV6 Protocols |
| AF_LOCAL | Unix domain protocols |
| AF_ROUTE | Routing Sockets |
| AF_KEY | Key Socket |

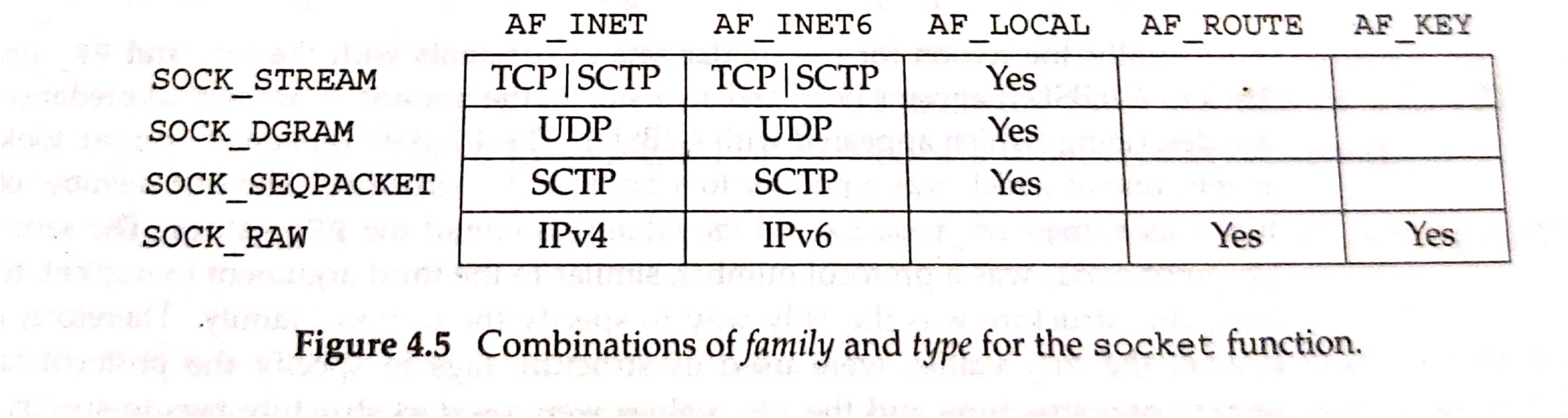
# Figure 4.3. *type of socket for socket function.*

| type | Description |
|---|---|
| SOCK_STREAM | Stream socket |
| SOCK_DGRAM | Datagram socket |
| SOCK_SEQPACKET | Sequenced packet socket |
| SOCK_RAW | Raw socket |

# Fig 4.4 protocol of sockets for AF_INET or AF_INET6

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

# Figure 4.5. Combinations of *family and type for the socket function.*



| | AF_INET | AF_INET6 | AF_LOCAL | AF_ROUTE | AF_KEY |
|---|---|---|---|---|---|
| SOCK_STREAM | TCP\|SCTP | TCP\|SCTP | Yes | | |
| SOCK_DGRAM | UDP | UDP | Yes | | |
| SOCK_SEQPACKET | SCTP | SCTP | Yes | | |
| SOCK_RAW | IPv4 | IPv6 | | Yes | Yes |

**Figure 4.5** Combinations of *family* and *type* for the socket function.

# 'connect' Function

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

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

**int connect(int *sockfd, const struct sockaddr *servaddr, socklen_t addrlen);***

**Returns: 0 if OK, -1 on error**

- *sockfd is a socket descriptor returned by the socket function.*
- *The second and third* arguments are a pointer to a socket address structure and its size
- The socket address structure must contain the IP address and port number of the server.
- The client does not have to call bind before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary

- In the case of a **TCP socket,** the **connect function initiates TCP's three-way handshake**
- The function **returns** only when the **connection is established or an error occurs.**

- There are several different error returns possible.

1. **If the client TCP receives no response to its SYN segment**, **ETIMEDOUT** is returned. 4.4BSD, for example, sends one SYN when connect is called, another 6 seconds later, and another 24 seconds later (p. 828 of TCPv2). If no response is received after a total of 75 seconds, the error is returned.

2. **If the server's response to the client's SYN is a reset (RST),** this indicates that no process is waiting for connections on the server host at the port specified (i.e., the server process is probably not running). This is a *hard error and the* error **ECONNREFUSED** is returned to the client as soon as the RST is received.

3. If the client's SYN elicits an **ICMP "destination unreachable"** from some intermediate router, this is considered a ***soft error.***

- *The client kernel saves the* message but keeps sending SYNs with the same time between each SYN as in the first scenario.
- If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either **EHOSTUNREACH or ENETUNREACH**.
- It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the connect call returns without waiting at all.

# 'bind' Function

- The bind function **assigns** a **local protocol address** to a **socket.**

- With the Internet protocols, the **protocol address** is the **combination** of either a **32-bit IPv4 address or a 128-bit IPv6 address**, along with a **16-bit TCP or UDP port number**

#include <sys/socket.h>

int bind (int *sockfd, const struct sockaddr *myaddr, socklen_t addrlen);*

Returns: 0 if OK,-1 on error

The second argument is a pointer to a protocol-specific address

- the third argument is the size of this address structure.

- **Servers bind their well-known port when they start**
- If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connect or listen is called.

- It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port
- But
- it is rare for a TCP server to let the kernel choose an ephemeral port, **since servers are known by their well-known port**.

- A process can bind a specific IP address to its socket.
- The IP address must belong to an interface on the host.
- For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket.
- For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

# 'listen' Function

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

1. When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect.

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

   In terms of the **TCP state transition diagram** the **call to listen** moves the socket from the **CLOSED state** to the **LISTEN state.**

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

#include <sys/socket.h>

#int listen (int *sockfd, int backlog);*

Returns: 0 if OK, -1 on error

- This function is normally **called after both the socket and bind functions** and must be called **before calling the accept function**.

- To understand the *backlog argument, we must realize that* ***for a given listening socket****,* the **kernel maintains two queues:**

1. An *incomplete connection queue*, *which contains an entry for each SYN that* has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the **SYN_RCVD state**

2. A *completed connection queue, which contains an entry for each client with* whom the TCP three-way handshake has completed. These sockets are in the **ESTABLISHED state**

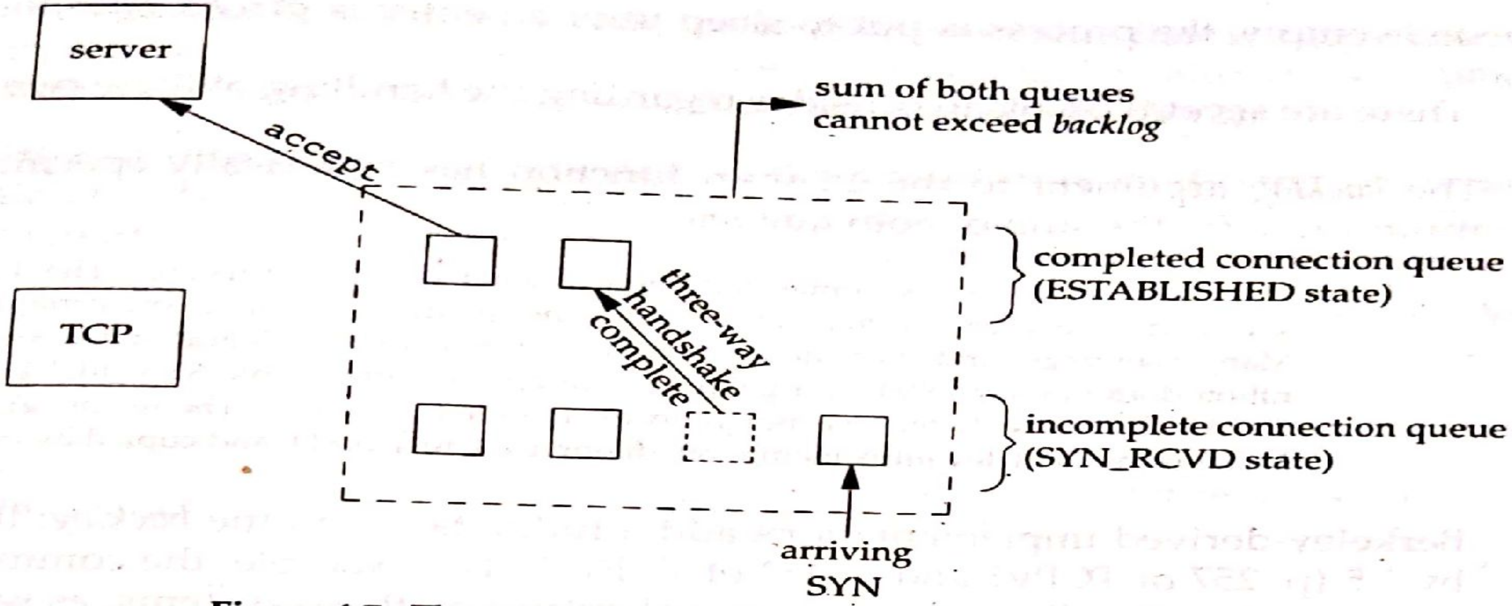

server
accept
sum of both queues cannot exceed *backlog*
TCP
three-way handshake complete
completed connection queue (ESTABLISHED state)
incomplete connection queue (SYN_RCVD state)
arriving SYN

**Figure 4.7** The two queues maintained by TCP for a listening socket.

When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved. Figure 4.8 depicts the packets exchanged during the connection establishment with these two queues.

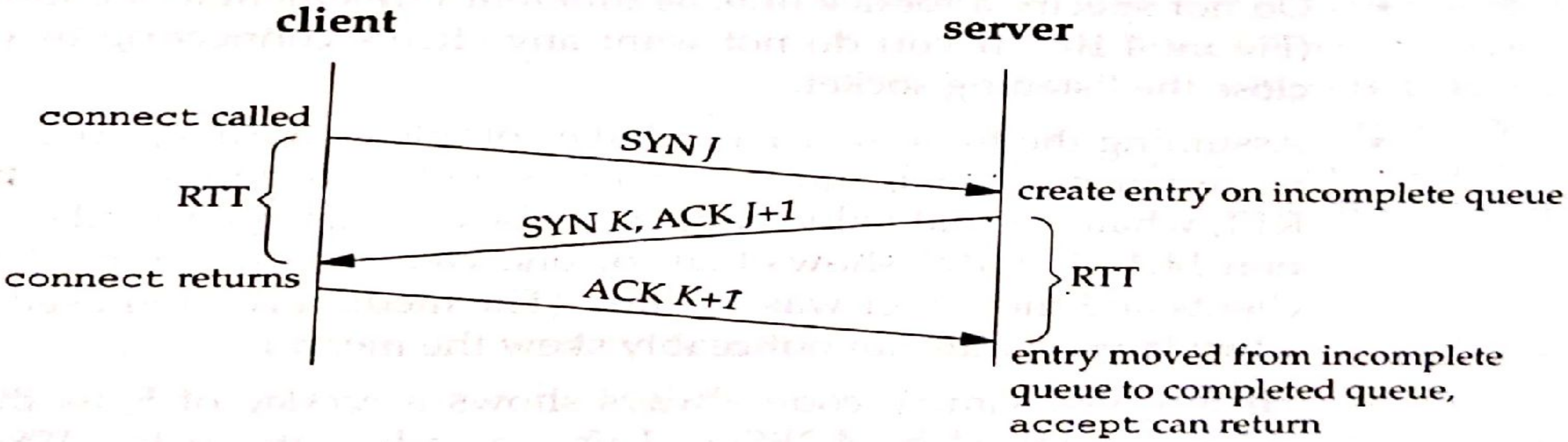

client
server
connect called
RTT
connect returns
SYN *J*
SYN *K*, ACK *J*+1
ACK *K*+1
create entry on incomplete queue
RTT
entry moved from incomplete queue to completed queue, accept can return

**Figure 4.8** TCP three-way handshake and the two queues for a listening socket.

- Figure  shows Wrapper function for listen that allows an environment variable to specify *backlog.*

***lib/wrapsock.c***

```
Void
Listen (int fd, int backlog)
{
char *ptr;
        /* can override 2nd argument with environment variable */

    if ( (ptr = getenv("LISTENQ")) != NULL)
    backlog = atoi (ptr);

    if (listen (fd, backlog) < 0)
    err_sys ("listen error");
}
```

# 'accept' Function

- accept is called by a TCP server to return the next completed connection from the front of the completed connection queue.

- If the completed connection queue is empty, the process is put to sleep .

**#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*

- 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.*

# 'fork' and 'exec' Functions

- The only way in Unix to create a new process.

**#include <unistd.h>**

**pid_t fork(void);**

Returns: **0** in **child**, **process ID of child** in **parent**, -**1** on **error**

- Fork is called *once but it returns twice.*

- It returns once in the calling process (called the **parent**) with a return value that is the **process ID of the newly created process** (the child).

- It also returns once in the child, with a return value of 0
- Hence, **the return value tells the process** whether it is the **parent** or the **child.**

- The **reason** fork returns 0 in the child, **instead of the parent's process ID**, is because a child has only one parent and it can always obtain the parent's process ID by calling getppid.

- A parent, on the other hand, can have any number of children, and there is no way to obtain the process IDs of its children. If a parent wants to keep track of the process IDs of all its children, it must record the return values from fork.

- All descriptors open in the parent before the call to fork are shared with the child after fork returns.

- **The parent calls accept and then calls fork.**

- **The connected socket is then shared between the parent and child**

- Normally, the child then reads and writes the connected socket and the parent closes the connected socket.

- **There are two typical uses of fork:**

1. **A process makes a copy of itself** so that **one copy can handle one operation** while the **other copy does another task**. This is typical for network servers.

2. **A process wants to execute another program**. Since the only way to create a new process is by **calling fork,** the process first calls fork to make a copy of itself, and then **one of the copies (typically the child process) calls exec to replace itself with the new program**. This is typical for programs such as shells.

# Exec Function

- The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the **six exec functions.**

- **exec replaces the current process image with the new program file**, and this new program normally starts at the main function. The process ID does not change.

- We refer to the **process that calls exec** as the ***calling process*** *and the* ***newly executed* program** as the *new program.*

- The differences in the six exec functions are:

(a) whether the program file to execute is specified by a *filename or a pathname;*

*(b) whether the arguments to the new* program are listed one by one or referenced through an array of pointers; and

(c) whether the environment of the calling process is passed to the new program or whether a new environment is specified.

- #include <unistd.h>
- int exec**l** (const char *_pathname, const char *arg0, ... /* (char *) 0 */ );_

- int exec**v** (const char *_pathname, char *const argv[]);_

- int exec**le** (const char *_pathname, const char *arg0, ... /*_ (char *) 0, char *const _envp[] */ );_

- int exec**ve** (const char *_pathname, char *const argv[], char *const envp[]);_
- int exec**lp** (const char *_filename, const char *arg0, ... /* (char *) 0 */ );_
- int exec**vp** (const char *_filename, char *const argv[]);_

All six return: -1 on error, no return on success

- Only exec**ve** is a **system call within the kernel** and the **other five** are **library functions** that call exec**ve**

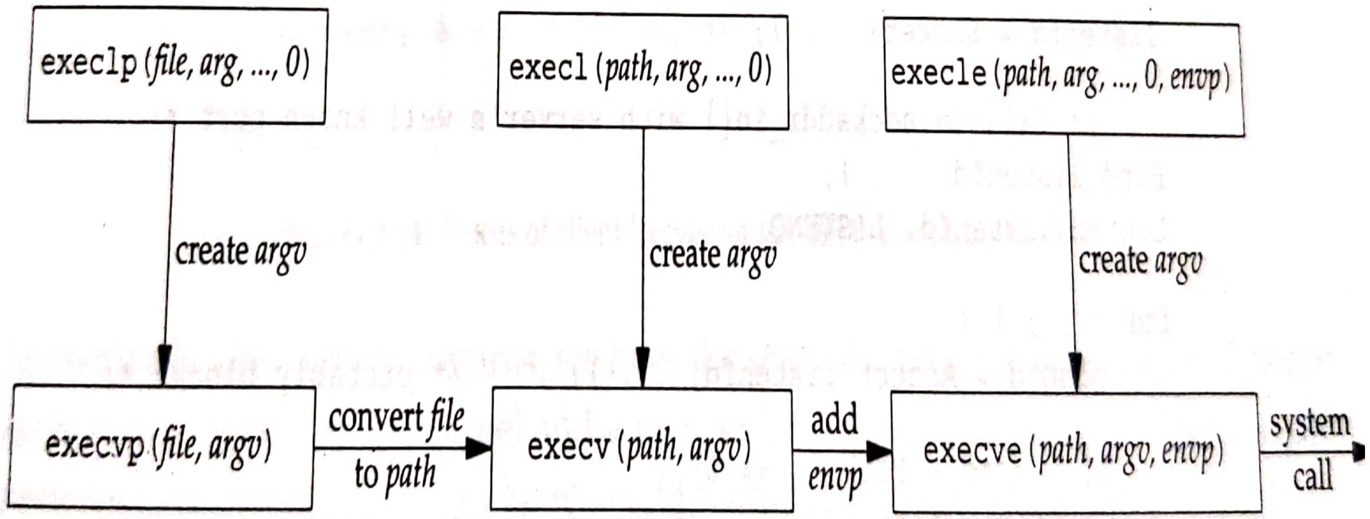


**Figure 4.12** Relationship among the six exec functions.

# Concurrent Servers

- When a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time.

- The simplest way to **write a *concurrent server under Unix* is to *fork a child* process to handle each client**. Figure 4.13 shows the outline for a typical concurrent server.

```
pid_t  pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
  connfd = Accept (listenfd, ... ); /* probably blocks */
  if( (pid = Fork()) == 0) {
    Close(listenfd); /* child closes listening socket */
    doit(connfd); /* process the request */
    Close(connfd); /* done with this client */
    exit(0); /* child terminates */
}
Close(connfd); /* parent closes connected socket */
}
```

- When a **connection is established**, **accept returns**, the **server calls fork**, and the **child process services the client (on connfd, the connected socket**) and the **parent process waits for another connection (on listenfd, the listening socket).**

- The **parent closes the connected socket since the child handles the new client.**

- In Figure 4.13, we assume that the function doit **does whatever is required to service the client.**

- When this **function returns**, we **explicitly close the connected socket in the child**. This is not required since the next statement **calls exit,** and **part of process termination is to close all open descriptors by the kernel**.

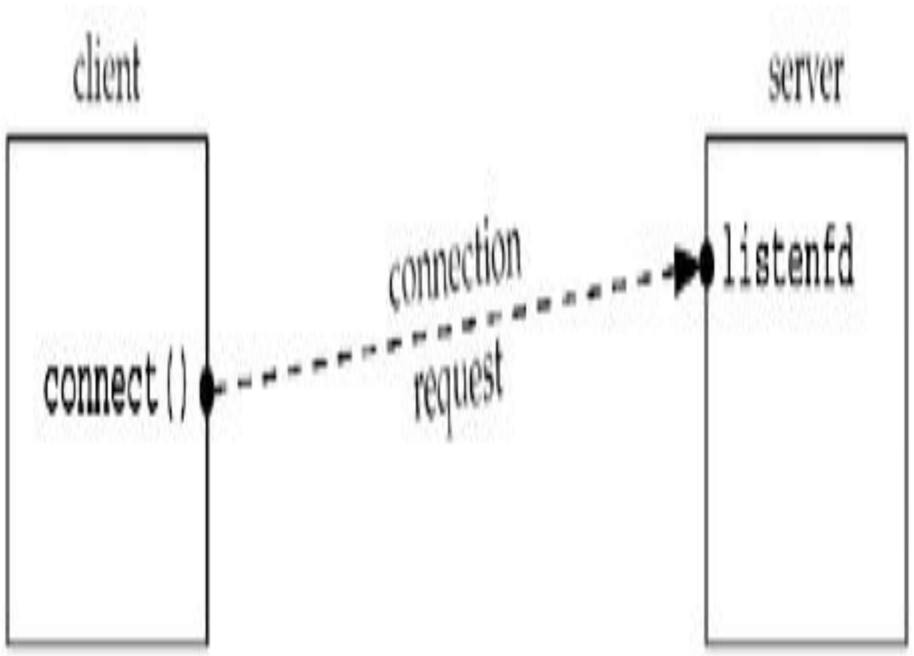
- Every file or socket has a **reference count.** The reference count is maintained in the **file table entry**

- This is a count of the number of descriptors that are currently open that refer to this file or socket.

- In Figure 4.13, after socket returns, the file table entry associated with listenfd has a reference count of 1.

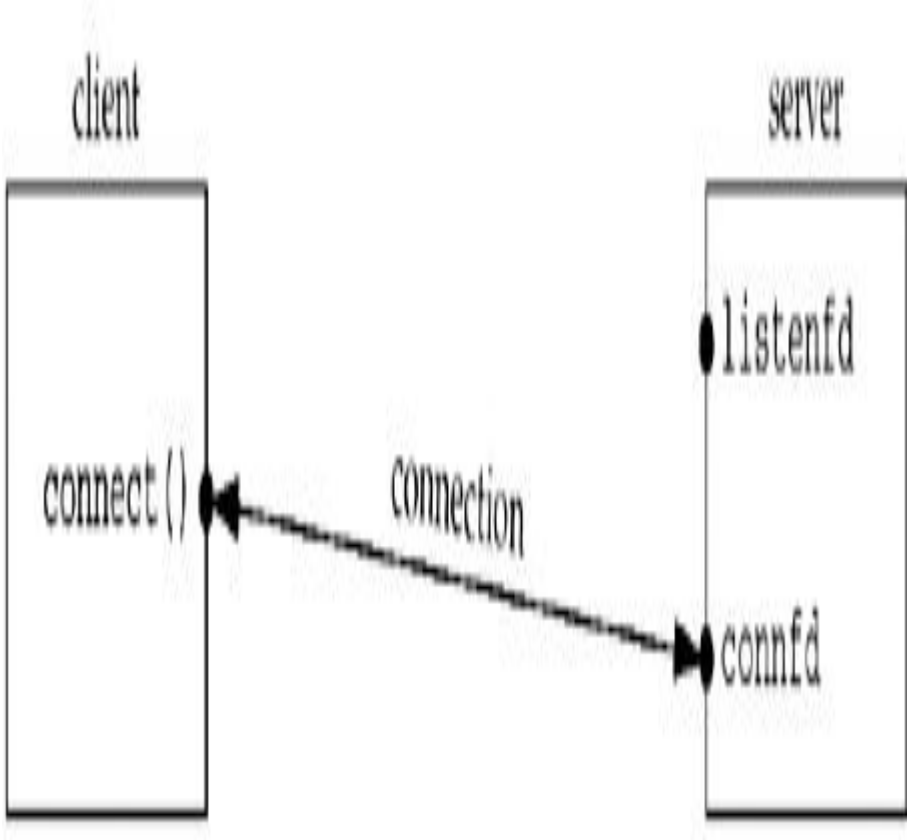
- **After accept returns, the file table entry associated with connfd has a reference count of 1**.
- But, **after fork returns**, *both descriptors are shared (i.e., duplicated) between the parent and child,* so the file table entries associated with both sockets now have a reference count of **2**.
- Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0.
- This will occur at **some time later** when the **child closes connfd.**

- We can also visualize the sockets and connection that occur in Figure 4.13 as follows.
- First, Figure 4.14 shows the status of the client and server while the **server is blocked in the call to accept** and **the connection request arrives from**
- **Figure 4.14. Status of client/server before call to accept returns. the client.**

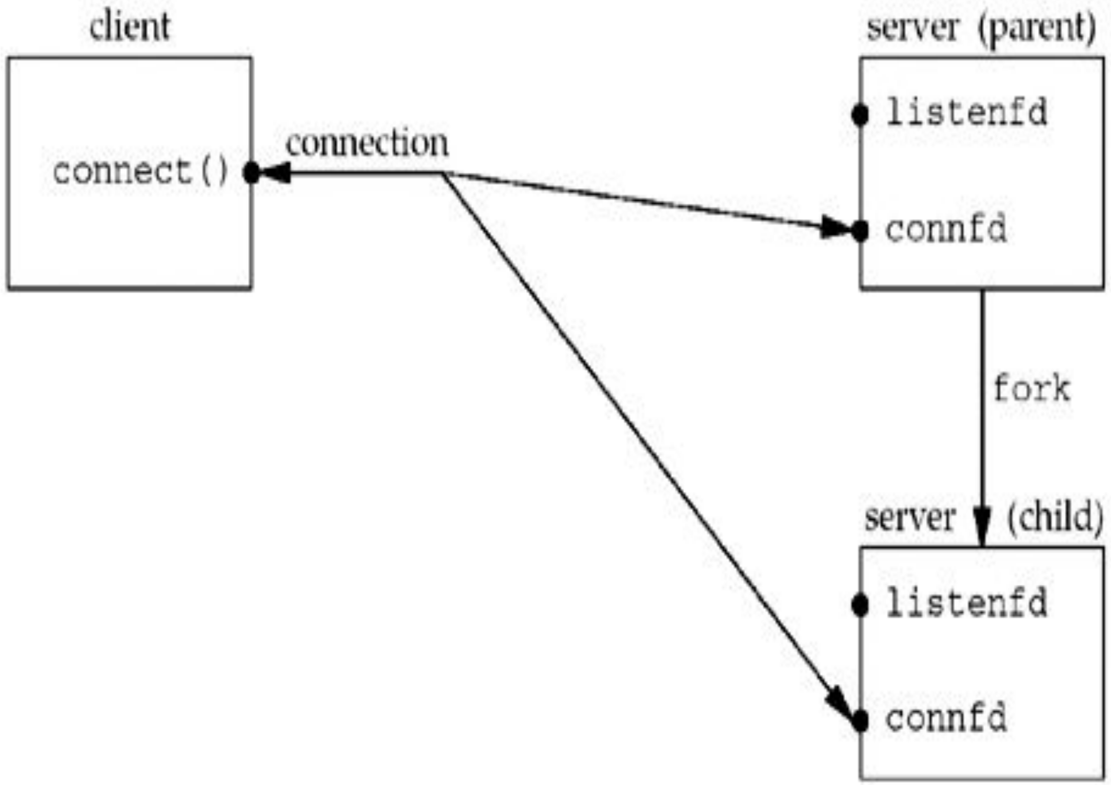


- Immediately **after accept returns**, we have the scenario shown in Figure 4.15.
- The connection is accepted by the kernel and a **new socket**, **connfd**, is **created.**
- This is a **connected socket** and **data can now be read and written across the connection**

- **Figure 4.15. Status of client/server after return from accept.**

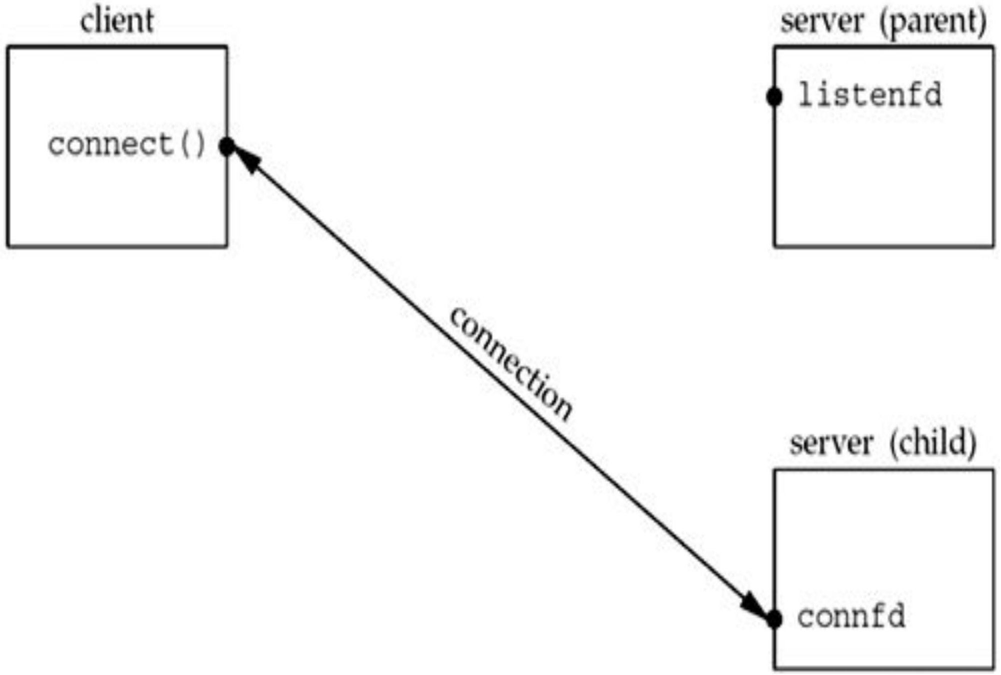


- The next step in the concurrent server is to call fork. Figure 4.16 shows the status after fork returns.
- **Figure 4.16. Status of client/server after fork returns.**



**Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.**

- The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.
- **Figure 4.17. Status of client/server after parent and child close appropriate sockets.**



This is the desired final state of the sockets. **The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.**

# 'close' Function

- The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
 int close (int sockfd);
                                    Returns: 0 if OK, -1 on error
```

- The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.

- The socket descriptor is no longer usable by the process: It cannot be used as an argument to read or write.

- But, TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place

# 'getsockname' and 'getpeername' Functions

- These two functions return either the **local protocol address** associated with a socket **(getsockname)** or the **foreign protocol address** associated with a socket **(getpeername).**

#include <sys/socket.h>

int getsockname(int *sockfd, struct sockaddr *localaddr, socklen_t *addrlen);*

int getpeername(int *sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);*

Both return: 0 if OK, -1 on error

- **NOTE:** the final argument for both functions is a value-result argument. That is, both functions fill in the socket address structure pointed to by *localaddr or peeraddr.*

# Review Questions

1. What are Sockets? Explain Socket address structures
2. Explain Generic socket address structure and its fields
3. Explain IPV4 socket address structure and its fields
4. Explain IPV6 address structure and its fields
5. Explain New Generic socket address structure and its fields
6. Explain Value-Result Arguments.
7. Explain Byte Ordering Functions
8. Explain Byte Manipulation Functions.
9. What are Socket Functions? With neat figure explain Socket functions for elementary TCP client/Server

10. Discuss the following Socket functions
    - Connect Function
    - bind Function
    - listen Function
    - Accept Function
    - close Function
11. Explain fork and exec Functions
12. Explain in detail the socket Function. Explain the two queues maintained by TCP for a listening socket . How can you increase the maximum queue size to hold more number of connections.
13. What is the difference between Iterative and Concurrent Servers
14. What are Concurrent Servers? Explain how does Concurrent Servers handle multiple clients at the same time. (explain with the necessary code and the diagrams)
15. What are getsockname and getpeername functions