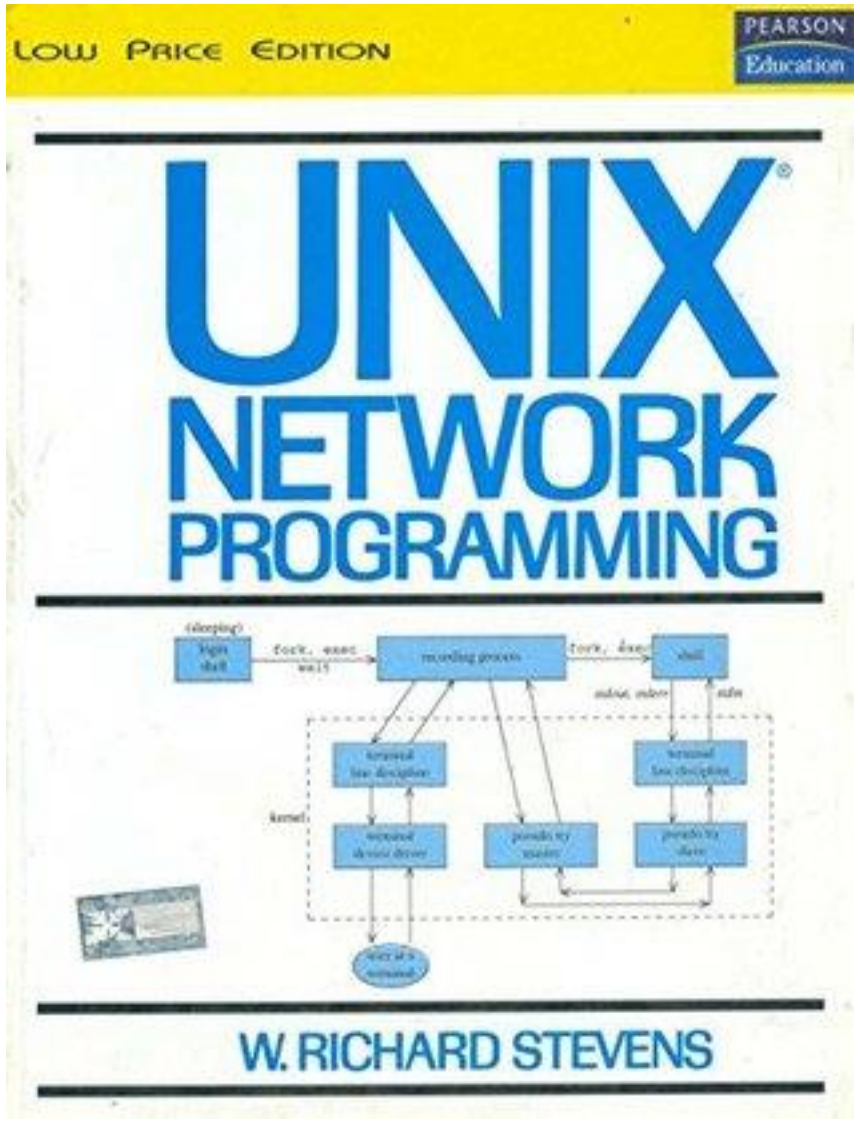
Network Programming



Unit – II 8 Hours

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.

Introduction:

- Elementary socket functions are required to write a complete TCP client and server, along with concurrent servers.
- **Sockets Introduction**: 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.
- Socket address structures, can be passed in two directions:
- Process to the kernel, example : Bind, connect, sendto and sendmsg.
- Kernel to the process, example : accept, recvfrom, recvmsg, getpeername, getsockname.
- All set the sin_len member before returning to the process.

Socket Address Structures: IPv4, Generic, IPv6

```
struct in_addr {
  in_addr_t         s_addr;           /* 32-bit IPv4 address, network byte order */      };
struct sockaddr_in {
  uint8_t           sin_len;          /* length of structure */
  sa_family_t       sin_family;       /* AF_INET */
  in_port_t         sin_port;         /* 16-bit port#, network byte order */
  struct in_addr    sin_addr;         /* 32-bit IPv4 address, network byte order */
  char              sin_zero[8];      /* unused */                                        };
struct sockaddr {                     /* only used to cast pointers */
  uint8_t           sa_len;
  sa_family         sa_family;        /* address family: AF_xxx value */
  char              sa_data[14];      /* protocol-specific address */                     };
struct in6_addr {
  unit8_t           s6_addr[16];      /* 128-bit IPv6 address, network byte order */ };
struct sockaddr_in6 {
  uint8_t           sin6_len;         /* length of this struct [24] */
  sa_family         sin6_family;      /* AF_INET6 */
  in_port_t         sin6_port;        /* port#, network byte order */
  uint32_t          sin6_flowinfo;    /* flow label and priority */
  struct in6_addr   sin6_addr;        /* IPv6 adress, network byte order */    };
```

Datatypes Required by Posix.1g

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>
sa_family_t	address family of socket addr struct	<sys/types.h>
socklen_t	length of socket addr struct, uint32_t	<sys/types.h>
in_addr_t	IPv4 address, normally uint32_t	<sys/types.h>
in_port_t	TCP or UDP port, normally uint16_t	<sys/types.h>

POSIX specifications requires only three members in the structure:
- sin_family,
- sin_addr, and
- sin_port.
- Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.
- in_addr_t datatype must be an unsigned integer type of at least 32 bits.
- in_port_t must be an unsigned integer type of at least 16 bits, and
- sa_family_t can be any unsigned integer type. (8-bit unsigned integer or an unsigned 16-bit integer).
- The sin_zero member is unused.

Generic Socket Address Structure

A socket address structures is always passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

use of these generic socket address structures :

1. **cast pointers to protocol-specific structures**. (Application programmer’s view)
2. **kernel must take the caller's pointer, cast it to a struct sockaddr *, and then look at the value of sa_family to determine the type of the structure(Kernel;s view)**

A generic socket address structure in the <sys/socket.h> header:

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

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);
```

Example:

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

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the <netinet/in.h> header:

- The SIN6_LEN constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET
- The members in this structure are ordered so that if the sockaddr_in6 structure is 64-bit aligned, so is the 128-bit sin6_addr member.
- The sin6_flowinfo member is divided into two fields:
    - The low-order 20 bits are the flow label
    - The high-order 12 bits are reserved
- The sin6_scope_id identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address

we define SA to be the string struct sockaddr, just to shorten the code that we must write to cast these pointers.

•From an application programmer ’s point of view, the only use of these generic socket address structures is to cast pointers to protocol-specific structures.

•From the kernel’s perspective, another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller’s pointer, cast it to a struct sockaddr *, and then look at the value of sa_family to determine the type of the structure.

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:

The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:
1.If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.
2.The sockaddr_storage is large enough to contain any socket address structure that the system supports.
The fields of the sockaddr_storage structure are opaque to the user, except for ss_family and ss_len (if present). The sockaddr_storage must be cast or copied to the appropriate socket address structure for the address given in ss_family to access any other fields.

Value-Result Arguments

- When a socket address structure is passed to any socket function, it is always passed by reference (a pointer to the structure is passed). The length of the structure is also passed as an argument.
- The way in which the length is passed depends on which direction the structure is being passed:
- From the **process to the kernel**
- From the **kernel to the process**

From process to kernel

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

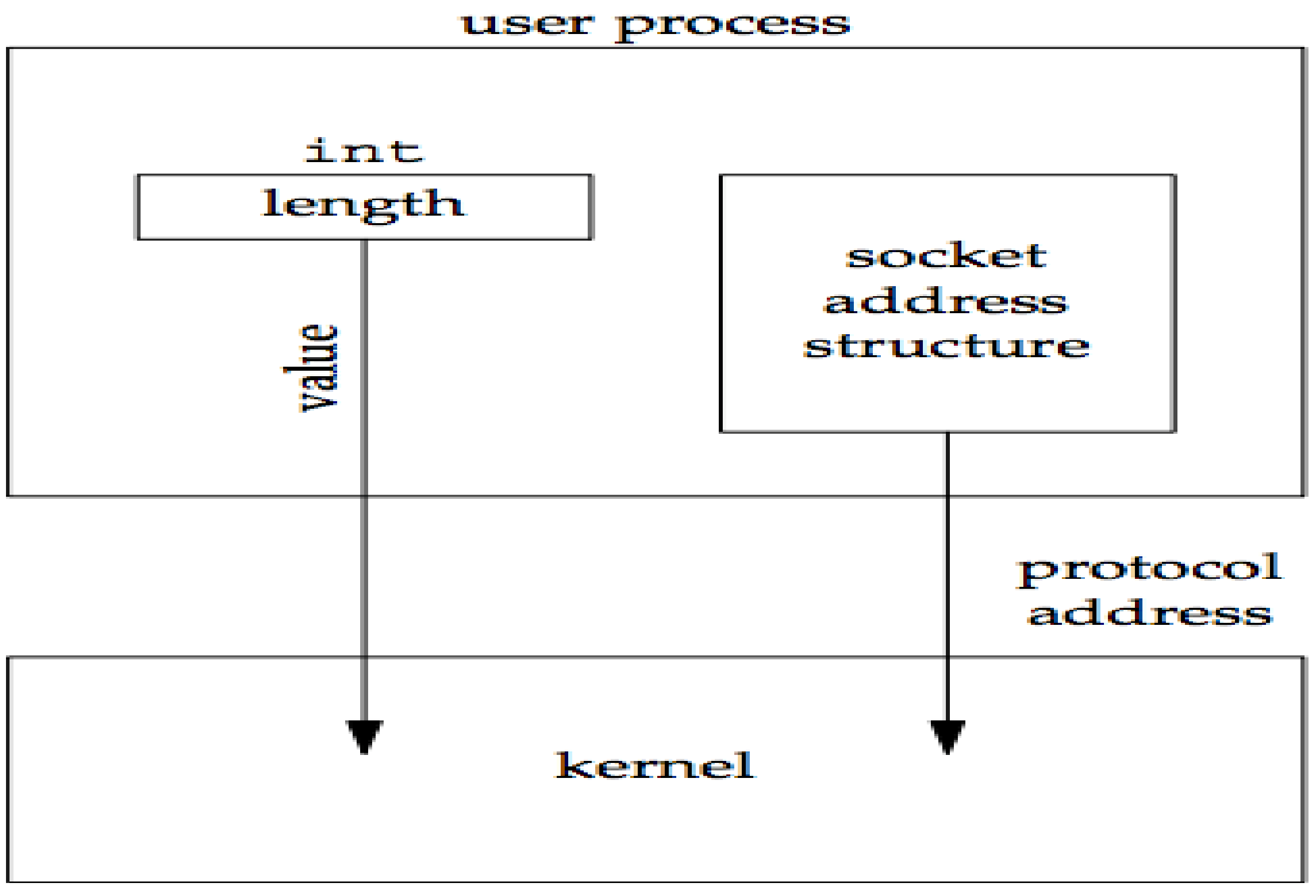
Arguments to these functions:

•The pointer to the socket address structure

•The integer size of the structure

```
struct sockaddr_in serv; /* fill in serv{} */ connect (sockfd, (SA *) &serv, sizeof(serv));
```

From the **process to the kernel**



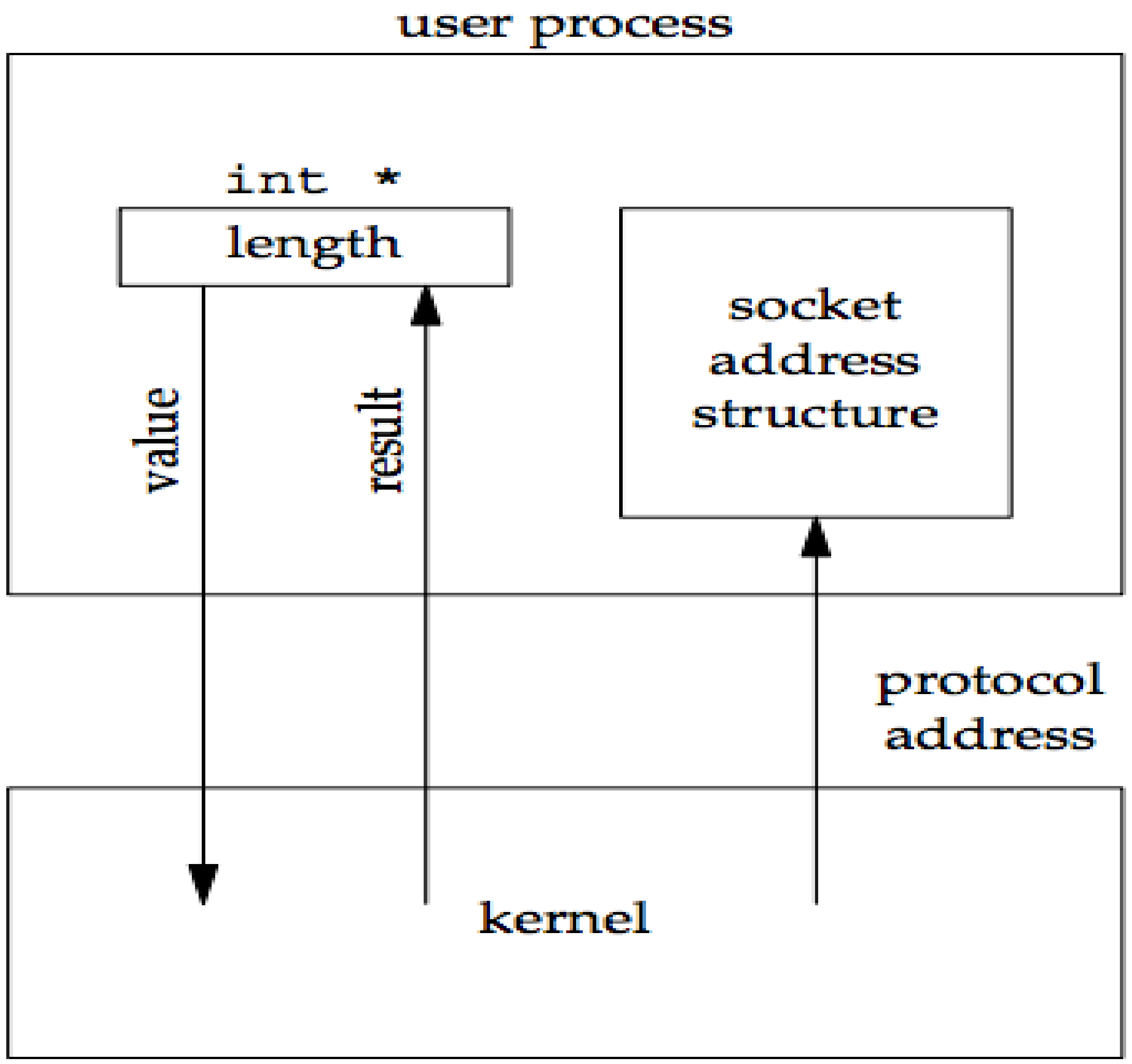
The datatype for the size of a socket address structure is actually socklen_t and not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.

From kernel to process

accept, recvfrom, getsockname, and getpeername functions pass a socket address structure from the kernel to the process.
Arguments to these functions:
•The pointer to the socket address structure
•The pointer to an integer containing the size of the structure.

```
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 */
```

From kernel to process



Value-result argument

The size changes from an integer to be a pointer to an integer because the size is both a value when the function is called and a result when the function returns.

•As a **value**: 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.

•As a **result**: it tells the process how much information the kernel actually stored in the structure

For two other functions that pass socket address structures, recvmsg and sendmsg, the length field is not a function argument but a structure member.

If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: 16 for an IPv4 sockaddr_in and 28 for an IPv6 sockaddr_in6. But with a variable-length socket address structure (e.g., a Unix domain sockaddr_un), the value returned can be less than the maximum size of the structure.

Though the most common example of a value-result argument is the length of a returned socket address structure, we encounter other value-result arguments :

•The middle three arguments for the select function.

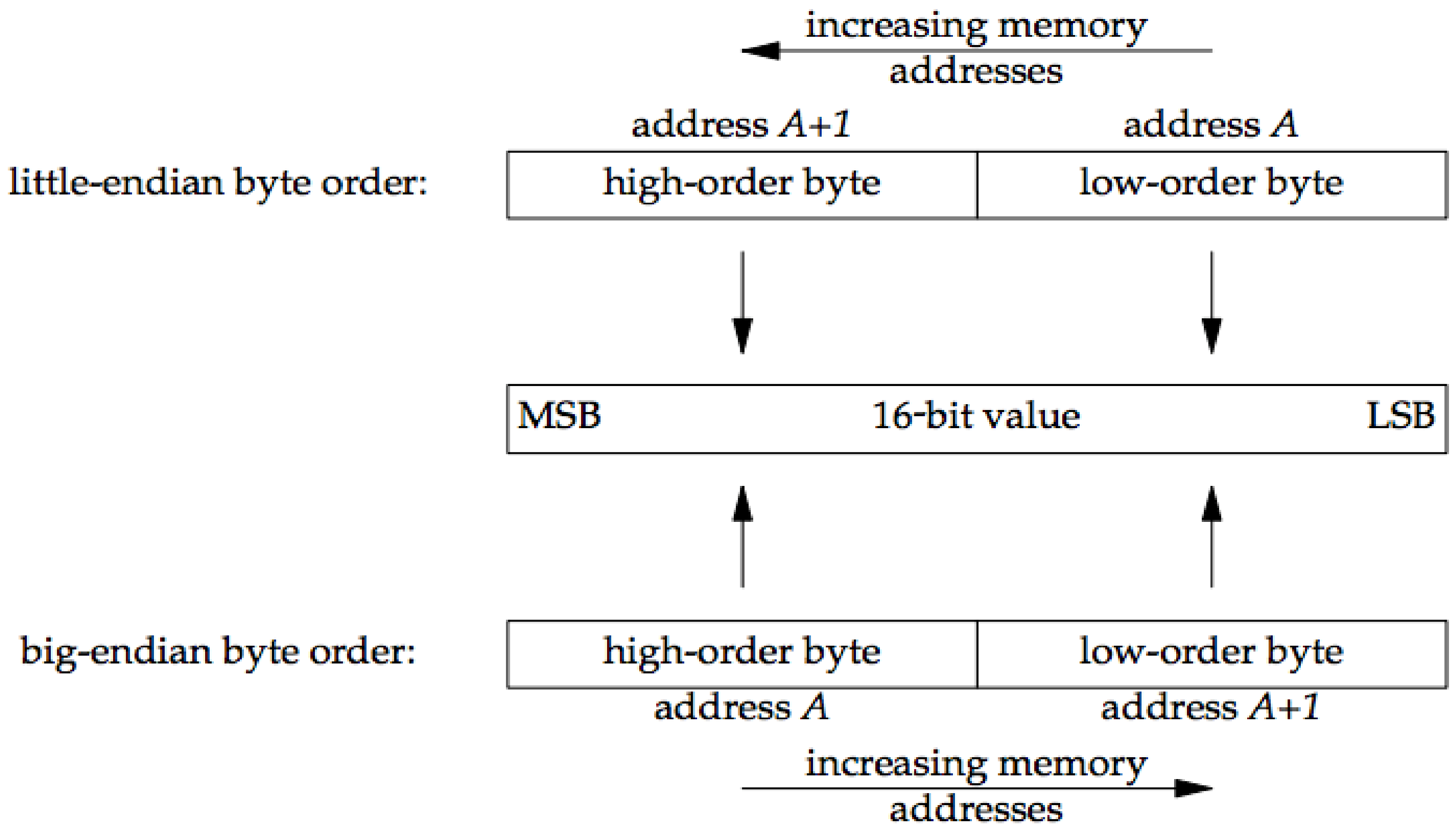
•The length argument for the getsockopt function.

•The msg_namelen and msg_controllen members of the msghdr structure, when used with recvmsg.

•The ifc_len member of the ifconf structure.

•The first of the two length arguments for the sysctl function.

For a 16-bit integer that is made up of 2 bytes, there are two ways to store the two bytes in memory:
Little-endian order: low-order byte is at the starting address.
Big-endian order: high-order byte is at the starting address.



Host byte order refer to the byte ordering used by a given system. The program below prints the host byte order:

```
#include "unp.h"

int
main(int argc, char **argv)
{
union {
short s;
char c[sizeof(short)];
} un;

un.s = 0x0102;
printf("%s: ", CPU_VENDOR_OS);
if (sizeof(short) == 2) {
if (un.c[0] == 1 && un.c[1] == 2)
printf("big-endian\n");
else if (un.c[0] == 2 && un.c[1] == 1)
printf("little-endian\n");
else
printf("unknown\n");
} else
exit(0);  }
```

We store the two-byte value 0x0102 in the short integer and then look at the two consecutive bytes, c[0] (the address *A*) and c[1] (the address *A+1*) to determine the byte order.

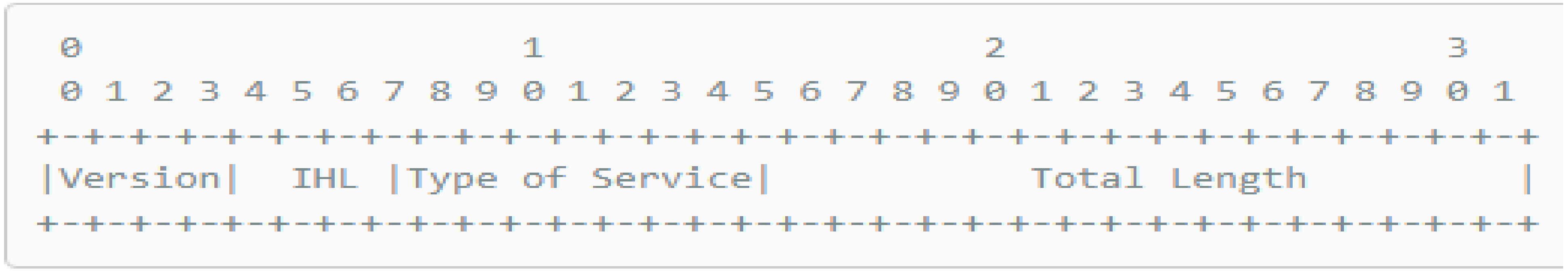
The string CPU_VENDOR_OS is determined by the GNU autoconf program.

- Networking protocols must specify a **network byte order**. 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.
- But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in 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 */
```

- h stands for *host*
- n stands for *network*
- s stands for *short* (16-bit value, e.g. TCP or UDP port number)
- l stands for *long* (32-bit value, e.g. IPv4 address)

- When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros.
- We use the term "byte" to mean an 8-bit quantity since almost all current computer systems use 8-bit bytes. Most Internet standards use the term **octet** instead of byte to mean an 8-bit quantity.
- Bit ordering is an important convention in Internet standards, such as the the first 32 bits of the IPv4 header from RFC 791:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- This represents four bytes in the order in which they appear on the wire; the leftmost bit is the most significant. However, the numbering starts with zero assigned to the most significant bit

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 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.
- We first show the Berkeley-derived functions, although the only one we use in this text is bzero. (We use it because it has only two arguments and is easier to remember than the three-argument memset function, as explained on p. 8.) You may encounter the other two functions, bcopy and bcmp, in existing applications.
- unp_bzero.h

```
#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 */
```

The memory pointed to by the const pointer is read but not modified by the function.
•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) */
```

- memset 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. The ANSI C memmove function must be used when the fields overlap. One way to remember the order of the two pointers for memcpy is to remember that they are written in the same left-to-right order as an assignment statement in C: dest = src; One way to remember the order of the final two arguments to memset is to realize that all of the ANSI C memXXX functions require a length argument, and it is always the final argument.
- 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, depending on whether the first unequal byte pointed to by ptr1 is greater than or less than the corresponding byte pointed to by ptr2. The comparison is done assuming the two unequal bytes are unsigned chars.

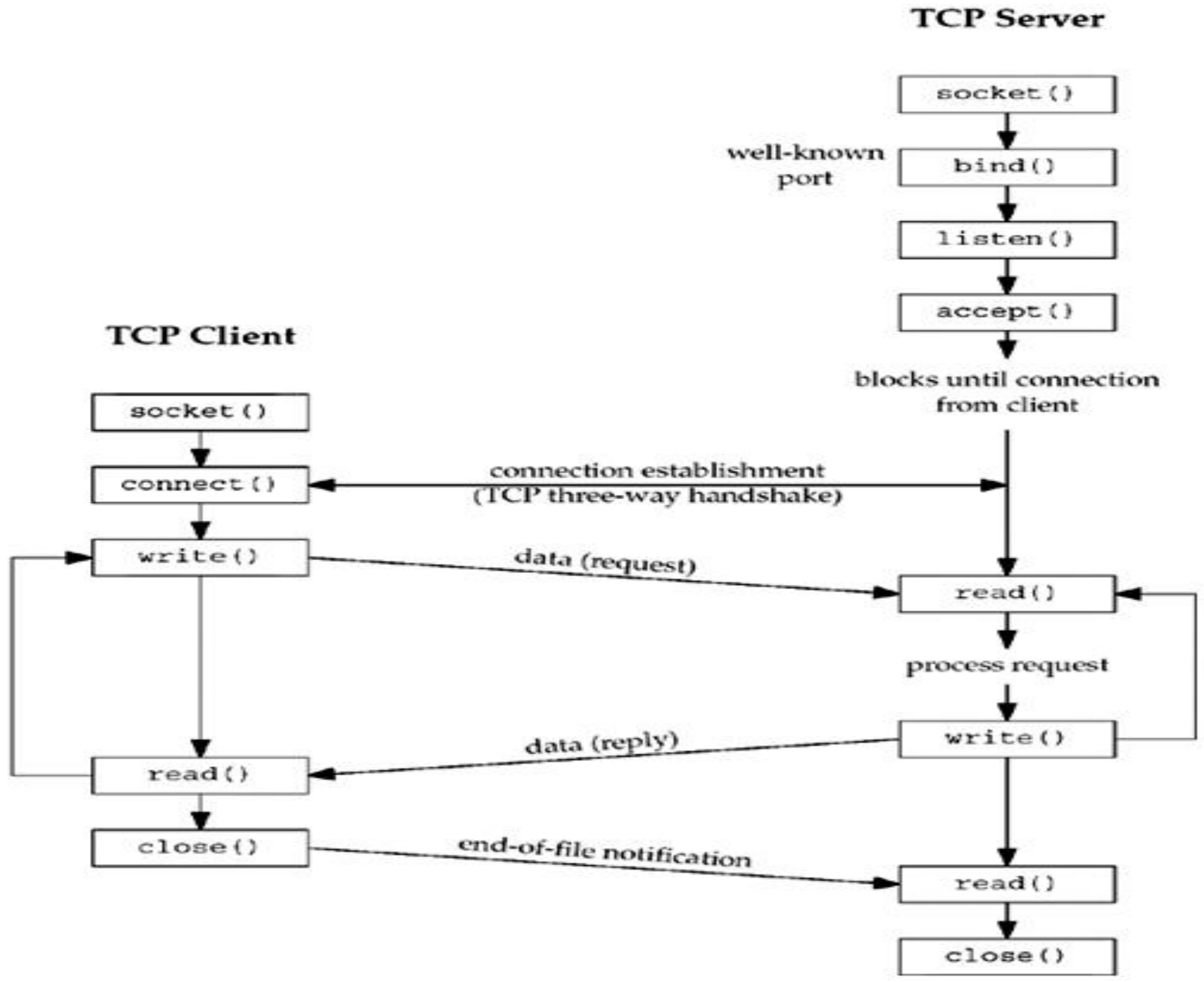
Elementary TCP Sockets

- Introduction

This chapter describes the elementary socket functions required to write a complete TCP client and server, along with concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to fork a new process just for that client. In this chapter, we consider only the one-process-per-client model using fork.

The figure below shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

Socket functions for elementary TCP client/server.



Socket Functions

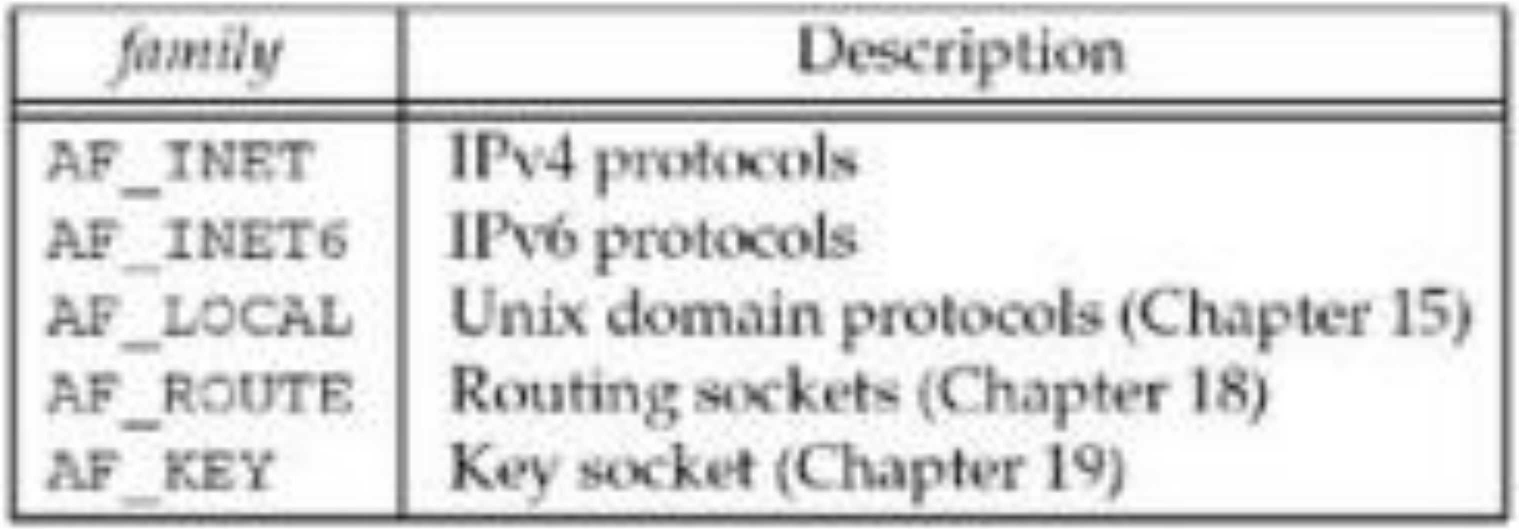
To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

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

Arguments:

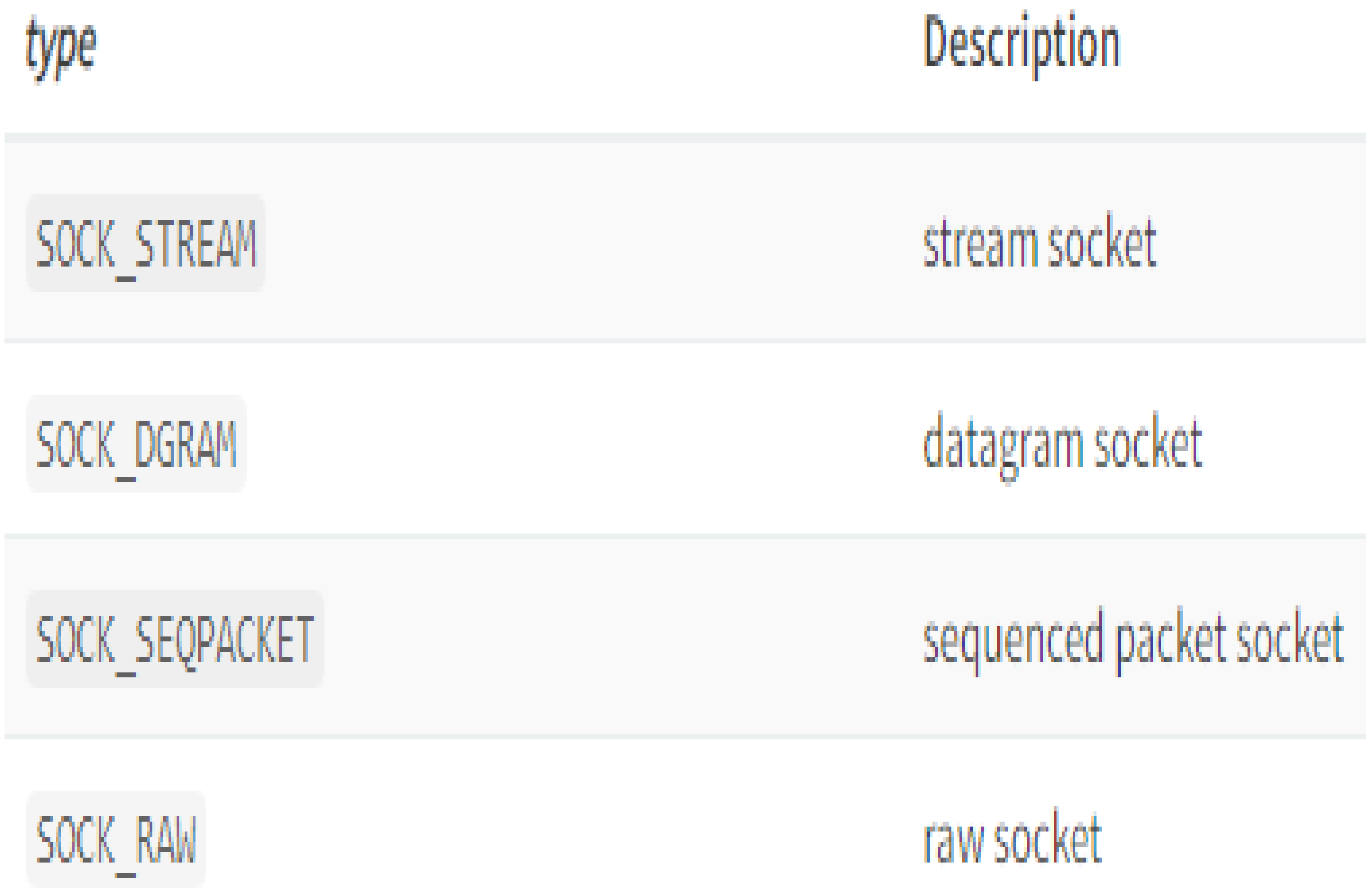
• *family* specifies the protocol family and is one of the constants in the table below. This argument is often referred to as *domain* instead of *family*.

Protocol family constants for socket function.



family	Description
`AF_INET`	IPv4 protocols
`AF_INET6`	IPv6 protocols
`AF_LOCAL`	Unix domain protocols (Chapter 15)
`AF_ROUTE`	Routing sockets (Chapter 18)
`AF_KEY`	Key socket (Chapter 19)

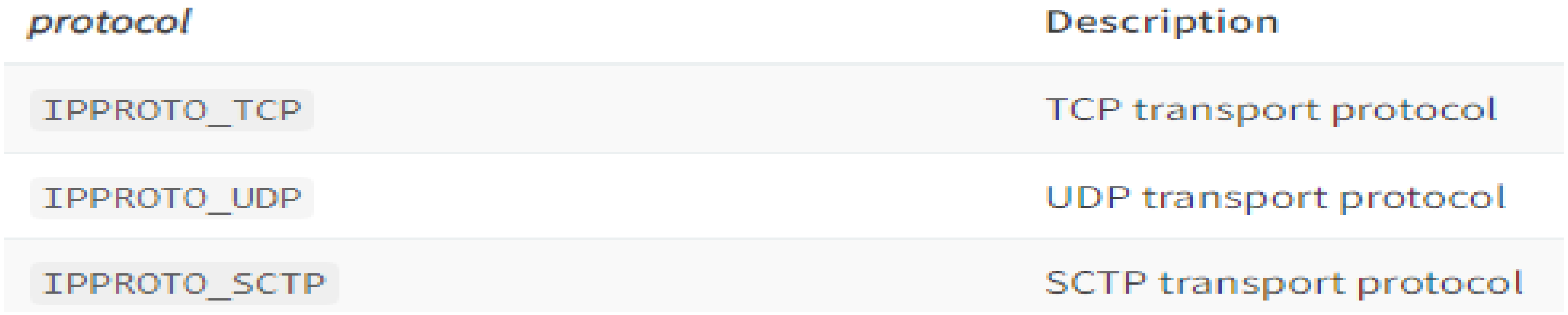
Type of socket fot socket functions



type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

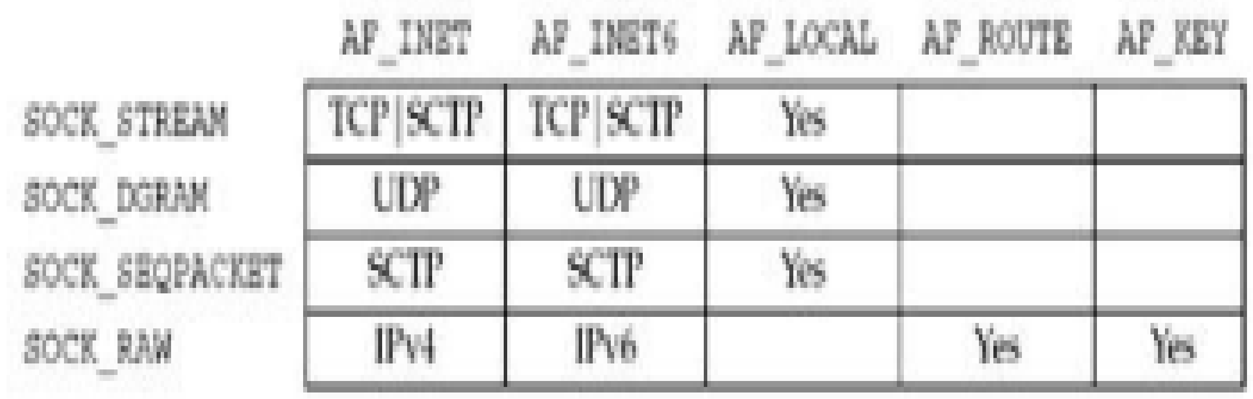
Protocol of sockets for AF_INET or AF_INET6

The *protocol* argument to the socket function should be set to the specific protocol type found in the table below, or 0 to select the system's default 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

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

On success, the socket function returns a small non-negative integer value, similar to a file descriptor. We call this a **socket descriptor**, or a *sockfd*. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw). We have not yet specified either the local protocol address or the foreign protocol address.

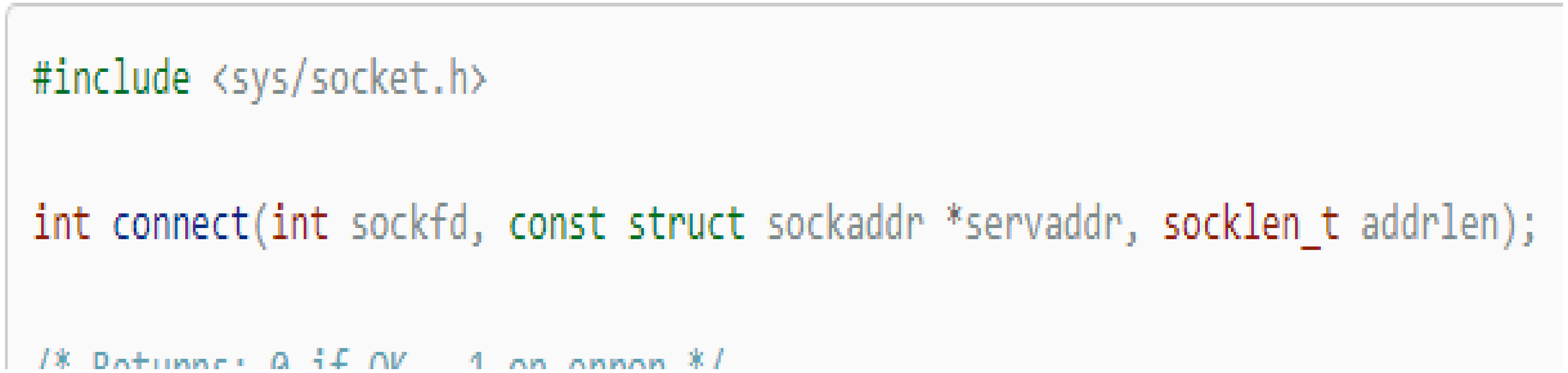
AF_*xxx* Versus PF_*xxx*

The "AF_" prefix stands for "address family" and the "PF_" prefix stands for "protocol family." Historically, the intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break.

To conform to existing coding practice, we use only the AF_ constants in this text, although you may encounter the PF_ value, mainly in calls to socket.

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 *servaddr* and *addrlen* arguments are a pointer to a socket address structure (which contains the IP address and port number of the server) and its size.
- 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 (Section 2.6). The function returns only when the connection is established or an error occurs. There are several different error returns possible.

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 (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. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:

- When a SYN arrives for a port that has no listening server.
- When TCP wants to abort an existing connection.
- When TCP receives a segment for a connection that does not exist.

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. Note that network unreachables are considered obsolete, and applications should just treat ENETUNREACH and EHOSTUNREACH as the same error.

Example: nonexistent host on the local subnet *

We run the client daytimetcpcli and specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent. When the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 192.168.1.100 connect error: Connection timed out
```

We only get the error after the connect times out. Notice that our err_sys function prints the human-readable string associated with the ETIMEDOUT error.

Example: no server process running *

We specify a host (a local router) that is not running a daytime server:

```
solaris % daytimetcpcli 192.168.1.5 connect error: Connection refused
```

The server responds immediately with an RST.

Example: destination not reachable on the Internet *

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets with `tcpdump`, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

As with the `ETIMEDOUT` error, `connect` returns the `EHOSTUNREACH` error only after waiting its specified amount of time.

In terms of the TCP state transition diagram (Figure 2.4):

- `connect` moves from the CLOSED state (the state in which a socket begins when it is created by the `socket` function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
- If `connect` fails, the socket is no longer usable and must be closed. We cannot call `connect` again on the socket.

In Figure 11.10, we will see that when we call `connect` in a loop, trying each IP address for a given host until one works, each time `connect` fails, we must close the socket descriptor and call `socket` again.

bind Function

The bind function assigns a local protocol address to a socket. 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 *myaddr* is a pointer to a protocol-specific addres
- The third argument *addrlen* 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.

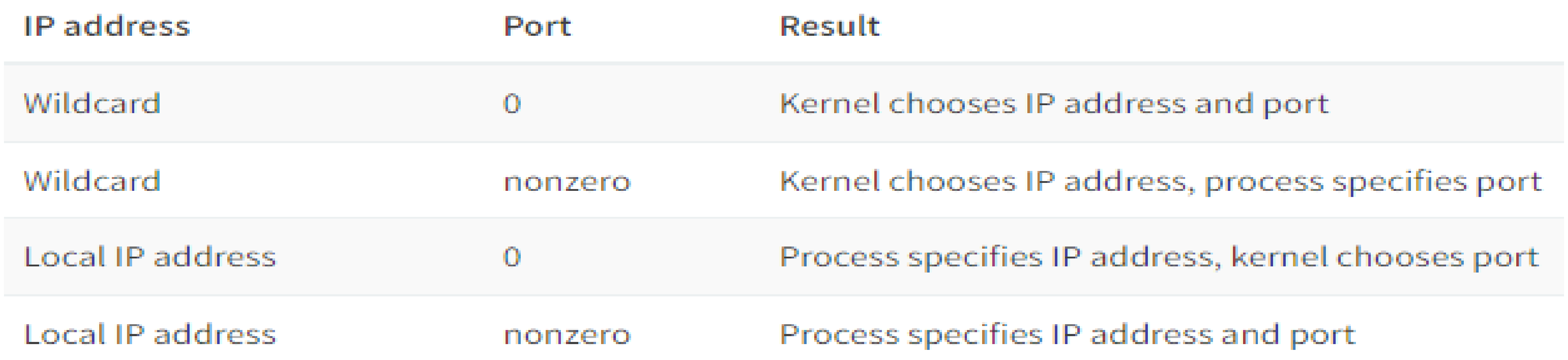
- It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port (Figure 2.10)
- However, it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

Exceptions to this rule are Remote Procedure Call (RPC) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can connect to the server. This also applies to RPC servers using UDP.

•**A process can bind a specific IP address to its socket.** The IP address must belong to an interface on the host.

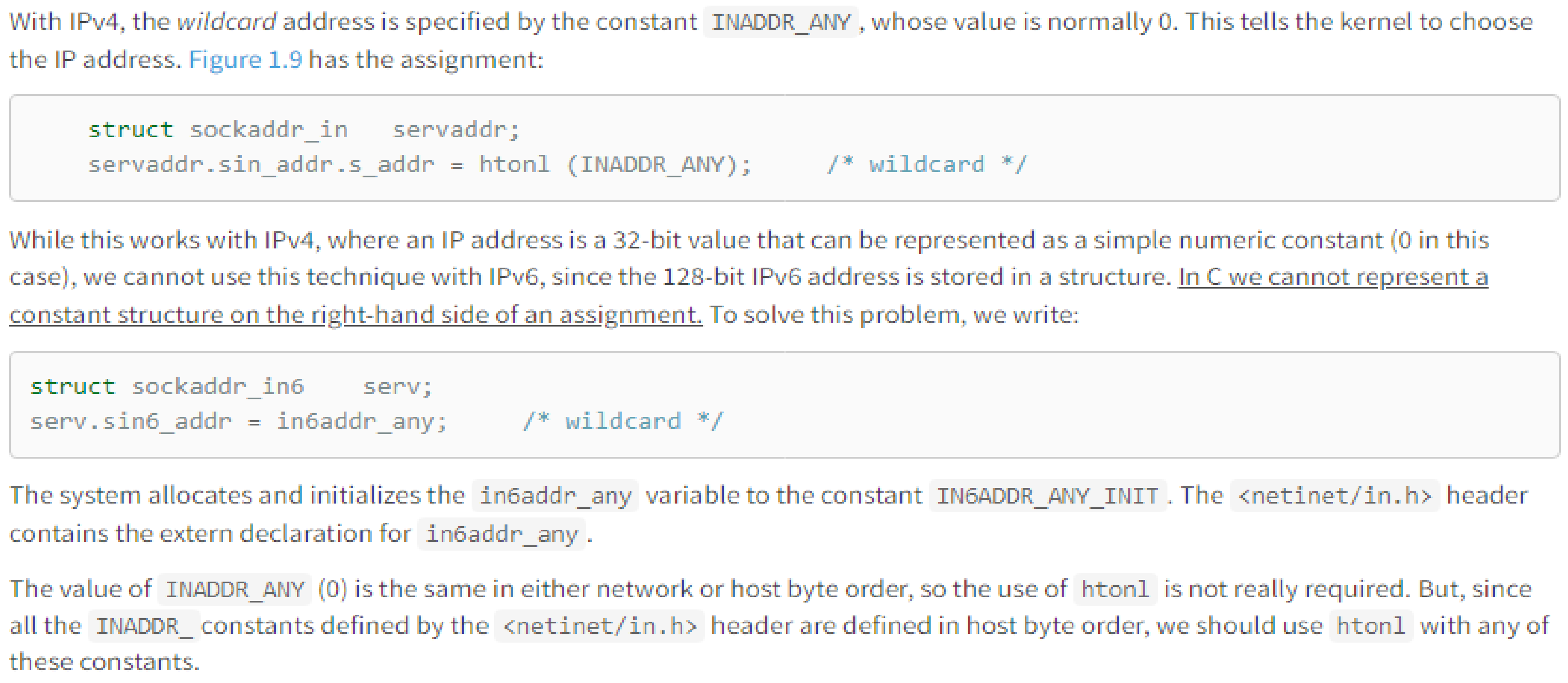
- For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server
- For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address.

Calling bind lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set sin_addr and sin_port, or sin6_addr and sin6_port, depending on the desired result.



IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called. But if we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).



With IPv4, the *wildcard* address is specified by the constant `INADDR_ANY`, whose value is normally 0. This tells the kernel to choose the IP address. Figure 1.9 has the assignment:

```
    struct sockaddr_in   servaddr;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);     /* wildcard */
```

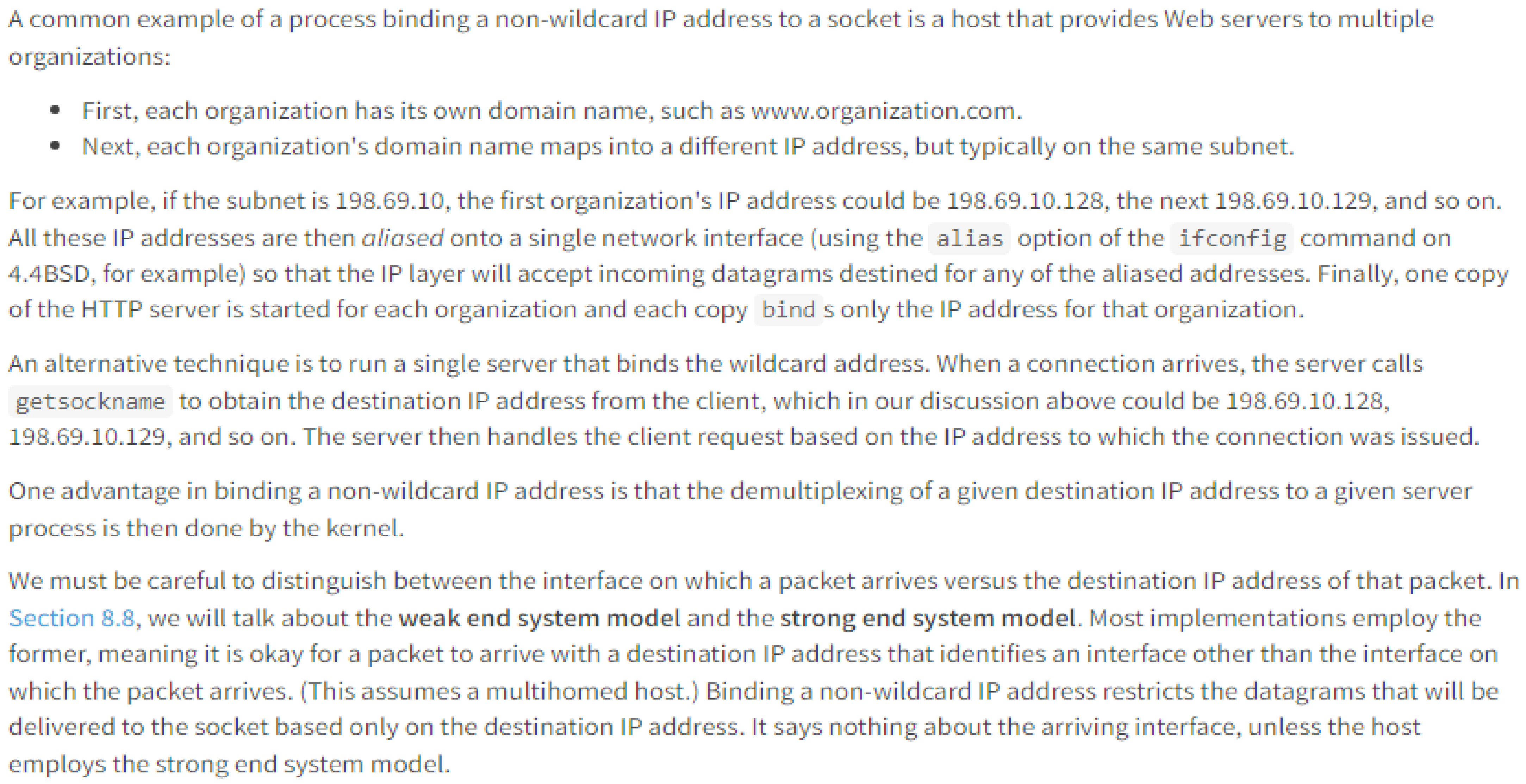
While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure. In C we cannot represent a constant structure on the right-hand side of an assignment. To solve this problem, we write:

```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;     /* wildcard */
```

The system allocates and initializes the `in6addr_any` variable to the constant `IN6ADDR_ANY_INIT`. The `<netinet/in.h>` header contains the extern declaration for `in6addr_any`.

The value of `INADDR_ANY` (0) is the same in either network or host byte order, so the use of `htonl` is not really required. But, since all the `INADDR_` constants defined by the `<netinet/in.h>` header are defined in host byte order, we should use `htonl` with any of these constants.

Binding a non-wildcard IP address



A common example of a process binding a non-wildcard IP address to a socket is a host that provides Web servers to multiple organizations:

- First, each organization has its own domain name, such as www.organization.com.
- Next, each organization's domain name maps into a different IP address, but typically on the same subnet.

For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All these IP addresses are then *aliased* onto a single network interface (using the `alias` option of the `ifconfig` command on 4.4BSD, for example) so that the IP layer will accept incoming datagrams destined for any of the aliased addresses. Finally, one copy of the HTTP server is started for each organization and each copy `bind`s only the IP address for that organization.

An alternative technique is to run a single server that binds the wildcard address. When a connection arrives, the server calls `getsockname` to obtain the destination IP address from the client, which in our discussion above could be 198.69.10.128, 198.69.10.129, and so on. The server then handles the client request based on the IP address to which the connection was issued.

One advantage in binding a non-wildcard IP address is that the demultiplexing of a given destination IP address to a given server process is then done by the kernel.

We must be careful to distinguish between the interface on which a packet arrives versus the destination IP address of that packet. In Section 8.8, we will talk about the **weak end system model** and the **strong end system model**. Most implementations employ the former, meaning it is okay for a packet to arrive with a destination IP address that identifies an interface other than the interface on which the packet arrives. (This assumes a multihomed host.) Binding a non-wildcard IP address restricts the datagrams that will be delivered to the socket based only on the destination IP address. It says nothing about the arriving interface, unless the host employs the strong end system model.

listen Function

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

1. 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 (Figure 2.4), the call to `listen` moves the socket from the CLOSED state to the LISTEN state.
    - When a socket is created by the `socket` function (and before calling `listen`), it is assumed to be an active socket, that is, a client socket that will issue a `connect`.
2. The second argument *backlog* to this function specifies the maximum number of connections the kernel should queue for this socket.

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

Connection queues *

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 (Figure 2.4).
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 (Figure 2.4).

The *backlog* argument *

Several points to consider when handling the two queues:

- **Sum of both queues**. The *backlog* argument to the `listen` function has historically specified the maximum value for the sum of both queues.
- **Multiplied by 1.5**. Berkeley-derived implementations add a fudge factor to the *backlog*: It is multiplied by 1.5.
    - If the *backlog* specifies the maximum number of completed connections the kernel will queue for a socket, then the reason for the fudge factor is to take into account incomplete connections on the queue.
- **Do not specify value of 0** for *backlog*, as different implementations interpret this differently (Figure 4.10). If you do not want any clients connecting to your listening socket, close the listening socket.
- **One RTT**. If the three-way handshake completes normally (no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT.
- **Configurable maximum value**. Many current systems allow the administrator to modify the maximum value for the *backlog*. Historically, sample code always shows a *backlog* of 5 (which is adequate today).
- **What value should the application specify for the *backlog*** (5 is often inadequate)? There is no easy answer to this.
    - HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server.
    - Another method is to allow a command-line option or an environment variable to override the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error. The following example is the wrapper function for `listen` which allows the environment variable `LISTENQ` to override the value specified by the caller:

lib/wrapsock.c#L166

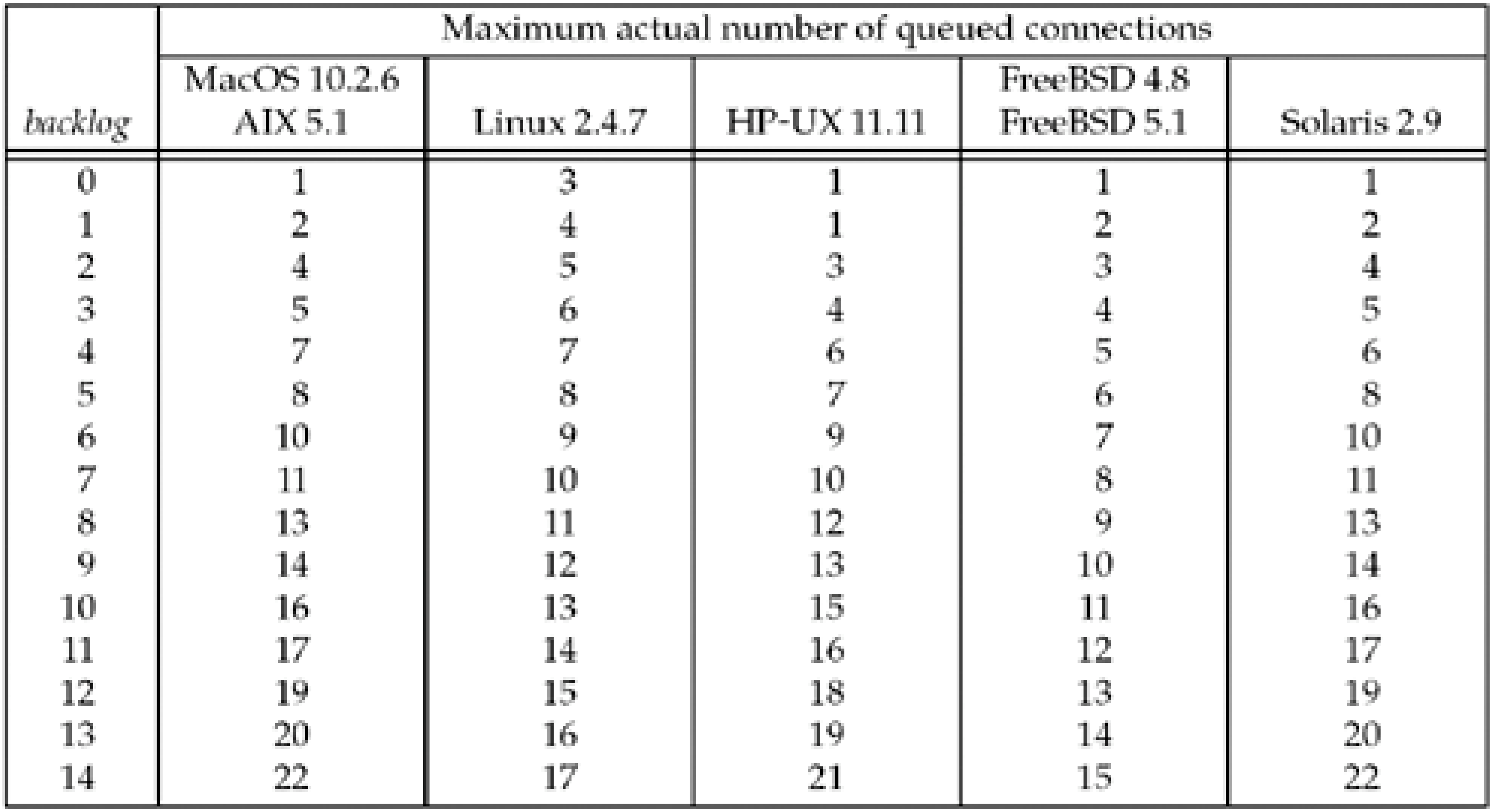
```c
void
Listen(int fd, int backlog)
{
    char    *ptr;

        /* can override 2nd argument with environment variable */
    if ( (ptr = getenv("LISTENQ")) != NULL)
        backlog = atoi(ptr);

    if (listen(fd, backlog) < 0)
        err_sys("listen error");
}
/* end Listen */
```

•**Fixed number of connections**. Historically the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to accept. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy Web servers have shown that this is false. The reason for specifying a large *backlog* is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.

•**No RST sent if queues are full**. If the queues are full when a client SYN arrives, TCP ignores the arriving SYN; it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP immediately responded with an RST, the client's connect would return an error, forcing the application to handle this condition instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."

•**Data queued in the socket's receive buffer**. Data that arrives after the three-way handshake completes, but before the server calls accept, should be queued by the server TCP, up to the size of the connected socket's receive buffer.

The following figure shows actual number of queued connections for values of *backlog*:



	Maximum actual number of queued connections				
backlog	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

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 successful, accept returns a new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client.

- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is 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 accepted (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),
- 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. See intro/daytimetcpsrv.c.

fork and exec Functions

- Before describing how to write a concurrent server in the next section, we must describe the Unix **fork** function.
- This function (including the variants of it provided by some systems) is 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 it returns values 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.

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. We will see many examples of this later in the text.
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` (described next) to replace itself with the new program. This is typical for programs such as shells.

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. (We will often refer generically to "the `exec` function" when it does not matter which of the six is called.) `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*.

> Older manuals and books incorrectly refer to the new program as the *new process*, which is wrong, because a new process is not created.

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 execl (const char *pathname, const char *arg0, ... /* (char *) 0 */);

int execv (const char *pathname, char *constargv[]);

int execle (const char *pathname, const char *arg0, ... /* (char *) 0, char *constenvp[] */);

int execve (const char *pathname, char *constargv[], char *constenvp[]); int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */);

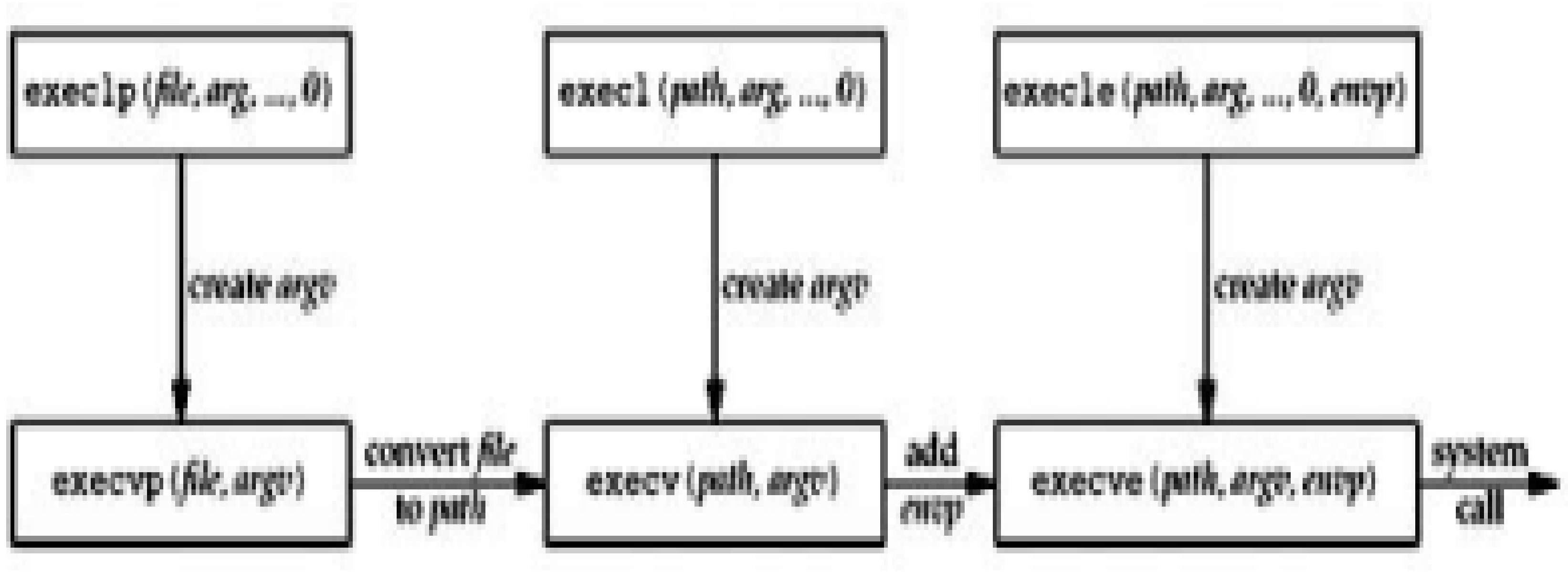
int execvp (const char *filename, char *constargv[]);

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

These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the main function.

- The relationship among these six functions is shown in Figure 4.12. Normally, only execve is a system call within the kernel and the other five are library functions that call execve.

Figure 4.12. Relationship among the six exec functions.



Note the following differences among these six functions:

1. The three functions in the top row specify each argument string as a separate argument to the `exec` function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an *argv* array, containing pointers to the argument strings. This *argv* array must contain a null pointer to specify its end, since a count is not specified.

2. The two functions in the left column specify a *filename* argument. This is converted into a *pathname* using the current `PATH` environment variable. If the *filename* argument to `execlp` or `execvp` contains a slash (/) anywhere in the string, the `PATH` variable is not used. The four functions in the right two columns specify a fully qualified *pathname* argument.

3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable `environ` is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The *envp* array of pointers must be terminated by a null pointer.

- Concurrent Servers

The server described in intro/daytimetcpsrv1.c is an **iterative server**. But 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.

The following code shows the outline for a typical concurrent server:

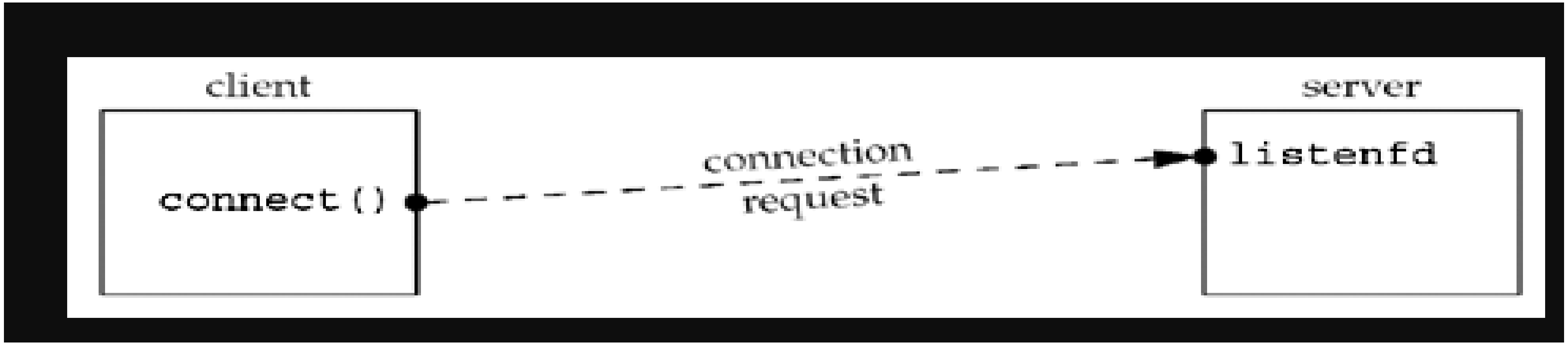
```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
 /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; )
{ connfd = Accept (listenfd, ... ); /* probably blocks */
if( (pid = Fork()) == 0)
{ Close(listenfd); /* child closes 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.

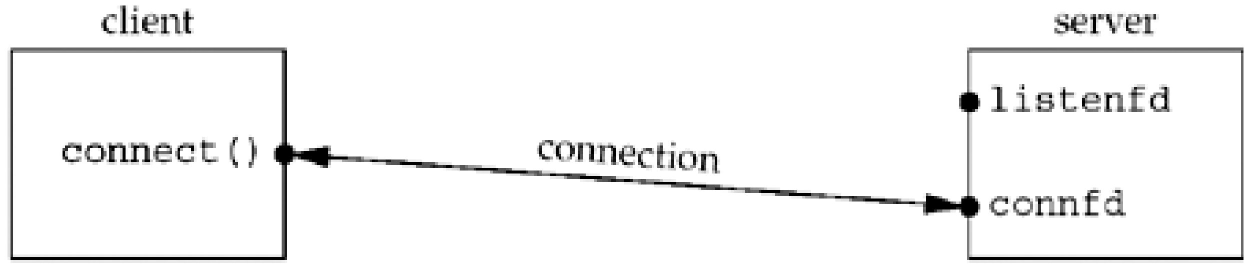
•We assume that the function doit does whatever is required to service the client. When this function returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel. Whether to include this explicit call to close or not is a matter of personal programming taste.

Visualizing the sockets and connection

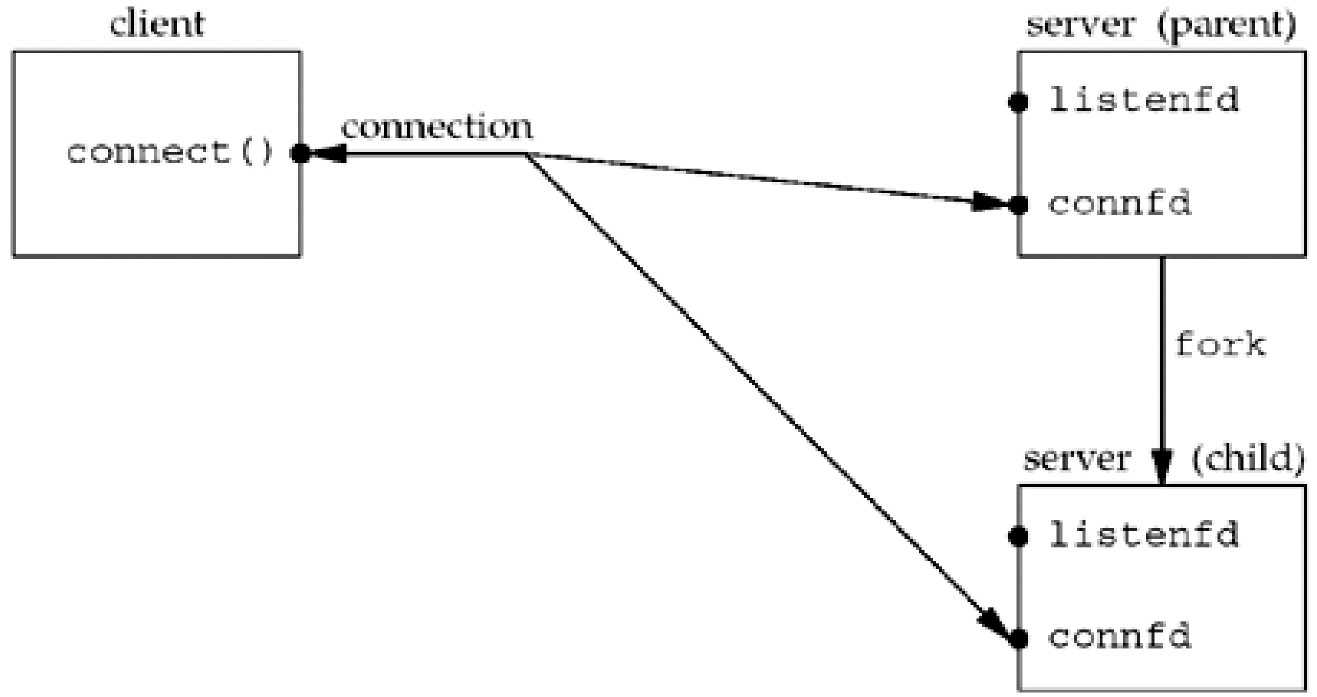
The following figures visualize the sockets and connection in the code above:
Before call to accept returns, the server is blocked in the call to accept and the connection request arrives from the client:



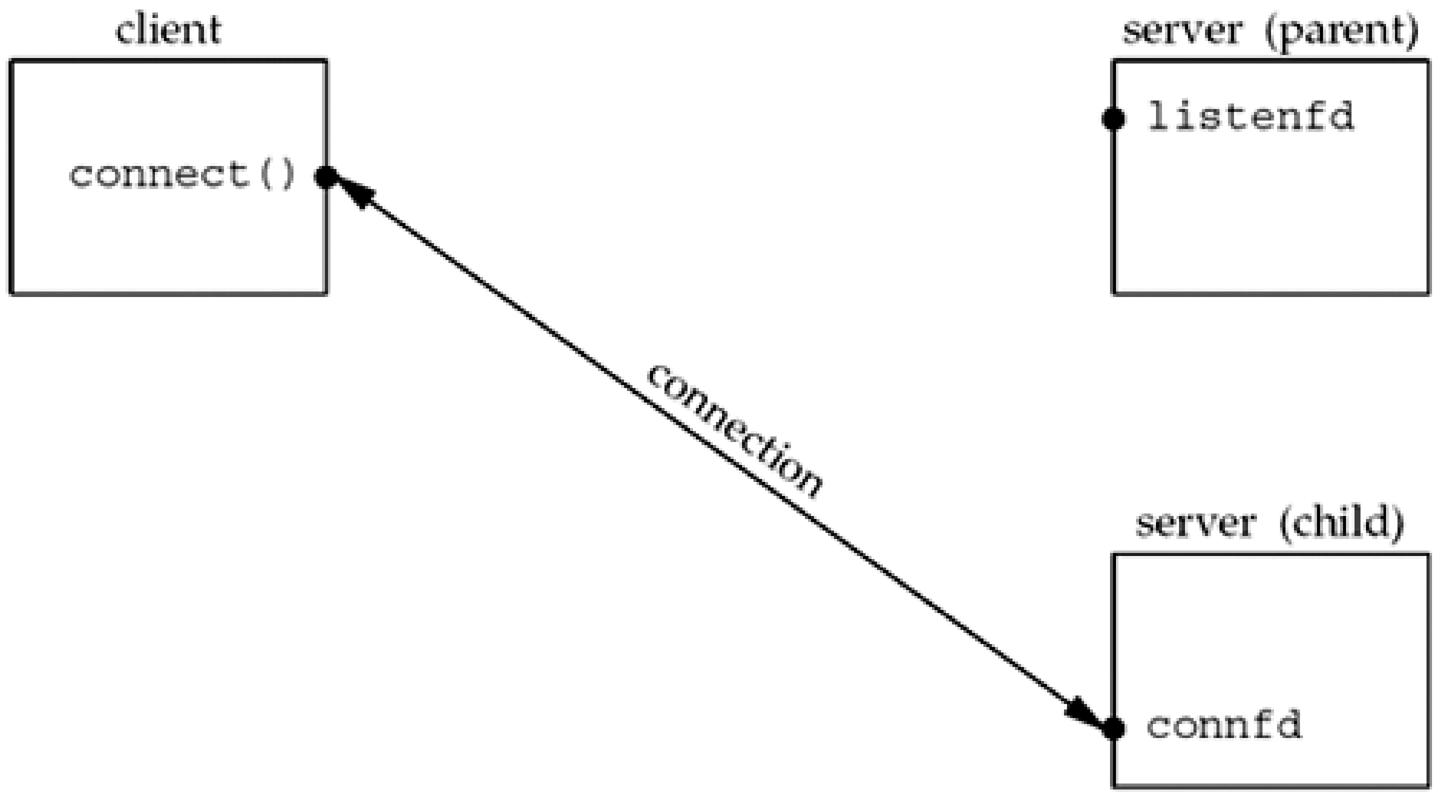
Ater return from `accept`, the connection is accepted by the kernel and a new socket `connfd` is created (this is a connected socket and data can now be read and written across the connection):



After `fork` returns, both descriptors, `listenfd` and `connfd`, are shared (duplicated) between the parent and child:



After the parent closes the connected socket and the child closes the listening socket:



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.

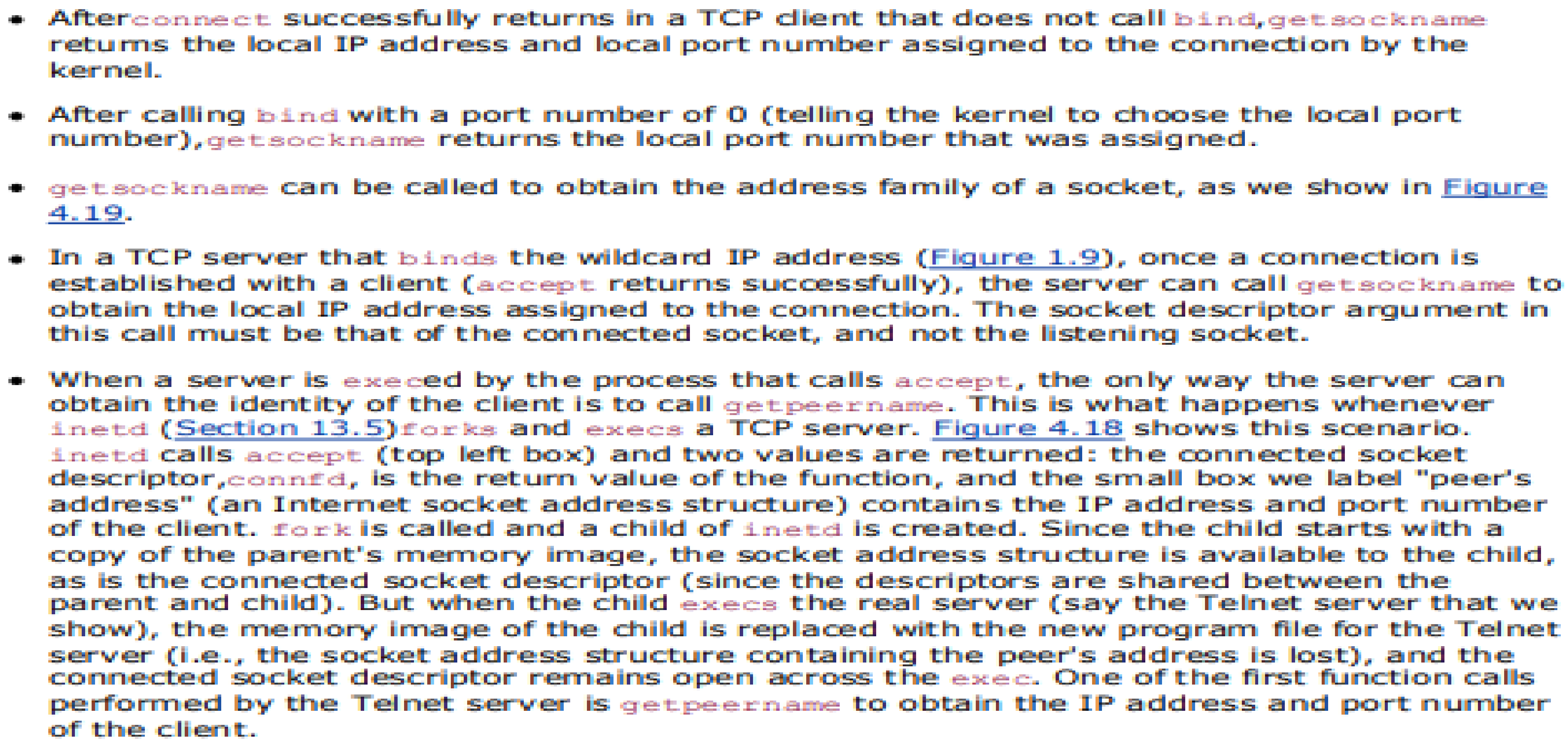
getsockname and getpeername Functions

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

```
#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 */
```

The *addrlen* argument for both functions is value-result argument: both functions fill in the socket address structure pointed to by localaddr or peeraddr.
The term "name" in the function name is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPV4 and IPV6 is the combination of an IP address and port number. These functions have nothing to do with domain names.
These two functions are required for the following reasons:

- These two functions are required for the following reasons:



- After`connect` successfully returns in a TCP client that does not call `bind`,`getsockname` returns the local IP address and local port number assigned to the connection by the kernel.
- After calling `bind` with a port number of 0 (telling the kernel to choose the local port number),`getsockname` returns the local port number that was assigned.
- `getsockname` can be called to obtain the address family of a socket, as we show in Figure 4.19.
- In a TCP server that `bind`s the wildcard IP address (Figure 1.9), once a connection is established with a client (`accept` returns successfully), the server can call `getsockname` to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is `exec`ed by the process that calls `accept`, the only way the server can obtain the identity of the client is to call `getpeername`. This is what happens whenever `inetd` (Section 13.5)`fork`s and `exec`s a TCP server. Figure 4.18 shows this scenario. `inetd` calls `accept` (top left box) and two values are returned: the connected socket descriptor,`connfd`, is the return value of the function, and the small box we label "peer's address" (an Internet socket address structure) contains the IP address and port number of the client. `fork` is called and a child of `inetd` is created. Since the child starts with a copy of the parent's memory image, the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child). But when the child `exec`s the real server (say the Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (i.e., the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the `exec`. One of the first function calls performed by the Telnet server is `getpeername` to obtain the IP address and port number of the client.