SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC302J- Computer Networks

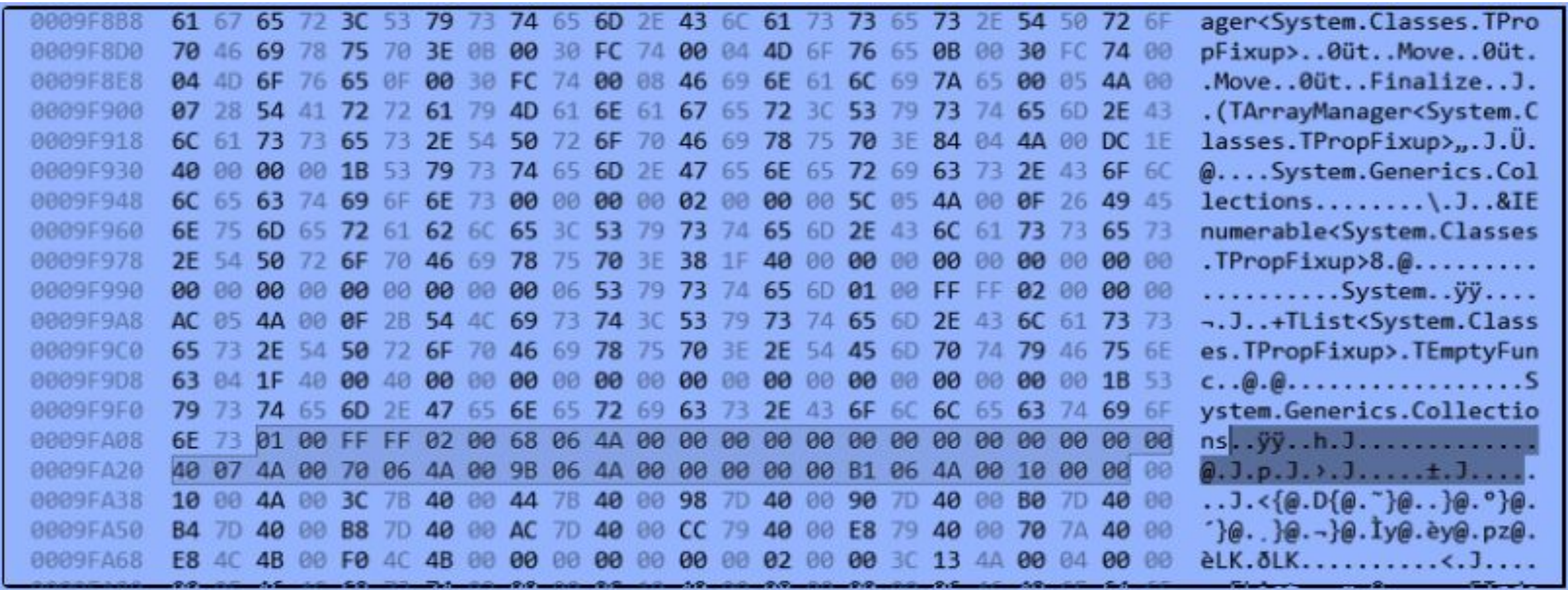
Unit II

Byte ordering, Byte ordering conversion functions, System calls, Sockets, System calls used with Sockets, Iterative and concurrent server, Socket Interface, Structure and Functions of Socket

Syllabus - Unit II

- **Byte ordering**
- **Byte ordering conversion functions**
- **System calls**
- **Sockets**
- **System calls used with Sockets**
- **Iterative and concurrent server**
- **Socket Interface**
- **Structure and Functions of Socket**
- Remote Procedure Call
- RPC Model, Features
- TCP Client Server Program
- Input, Output Processing Module
- UDP Client Server Program
- UDP Control block table & Module
- UDP Input & Output Module
- SCTP Sockets
- SCTP Services and Features, Packet Format
- SCTP Client/Server

1. Byte Ordering



```
0009F8B8  61 67 65 72 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73 65 73 2E 54 50 72 6F  ager<System.Classes.TPro
0009F8D0  70 46 69 78 75 70 3E 0B 00 30 FC 74 00 04 4D 6F 76 65 0B 00 30 FC 74 00  pFixup>..0üt..Move..0üt.
0009F8E8  04 4D 6F 76 65 0F 00 30 FC 74 00 08 46 69 6E 61 6C 69 7A 65 00 05 4A 00  .Move..0üt..Finalize..J.
0009F900  07 28 54 41 72 72 61 79 4D 61 6E 61 67 65 72 3C 53 79 73 74 65 6D 2E 43  .(TArrayManager<System.C
0009F918  6C 61 73 73 65 73 2E 54 50 72 6F 70 46 69 78 75 70 3E 84 04 4A 00 DC 1E  lasses.TPropFixup>„.J.Ü.
0009F930  40 00 00 00 1B 53 79 73 74 65 6D 2E 47 65 6E 65 72 69 63 73 2E 43 6F 6C  @....System.Generics.Col
0009F948  6C 65 63 74 69 6F 6E 73 00 00 00 00 02 00 00 00 5C 05 4A 00 0F 26 49 45  lections........\.J..&IE
0009F960  6E 75 6D 65 72 61 62 6C 65 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73 65 73  numerable<System.Classes
0009F978  2E 54 50 72 6F 70 46 69 78 75 70 3E 38 1F 40 00 00 00 00 00 00 00 00 00  .TPropFixup>8.@.........
0009F990  00 00 00 00 00 00 00 00 00 06 53 79 73 74 65 6D 01 00 FF FF 02 00 00 00  ..........System..ÿÿ....
0009F9A8  AC 05 4A 00 0F 2B 54 4C 69 73 74 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73  ¬.J..+TList<System.Class
0009F9C0  65 73 2E 54 50 72 6F 70 46 69 78 75 70 3E 2E 54 45 6D 70 74 79 46 75 6E  es.TPropFixup>.TEmptyFun
0009F9D8  63 04 1F 40 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1B 53  c..@.@.................S
0009F9F0  79 73 74 65 6D 2E 47 65 6E 65 72 69 63 73 2E 43 6F 6C 6C 65 63 74 69 6F  ystem.Generics.Collectio
0009FA08  6E 73 01 00 FF FF 02 00 68 06 4A 00 00 00 00 00 00 00 00 00 00 00 00 00  ns..ÿÿ..h.J.............
0009FA20  40 07 4A 00 70 06 4A 00 9B 06 4A 00 00 00 00 00 B1 06 4A 00 10 00 00 00  @.J.p.J.›.J.....±.J.....
0009FA38  10 00 4A 00 3C 7B 40 00 44 7B 40 00 98 7D 40 00 90 7D 40 00 B0 7D 40 00  ..J.<{@.D{@.˜}@..}@.°}@.
0009FA50  B4 7D 40 00 B8 7D 40 00 AC 7D 40 00 CC 79 40 00 E8 79 40 00 70 7A 40 00  ´}@..}@.¬}@.Ìy@.èy@.pz@.
0009FA68  E8 4C 4B 00 F0 4C 4B 00 00 00 00 00 00 00 02 00 00 3C 13 4A 00 04 00 00  èLK.ðLK..........<.J....
```

Byte Ordering

- An **arrangement of bytes** when data is transmitted over the network is called byte ordering.
- Different computers will use different byte ordering.
- When communication taking place between two machines byte ordering should not make discomfort.
- Generally an Internet protocol will specify a common form to allow different machines byte ordering. TCP/IP is the Internet Protocol in use.

Byte Ordering functions

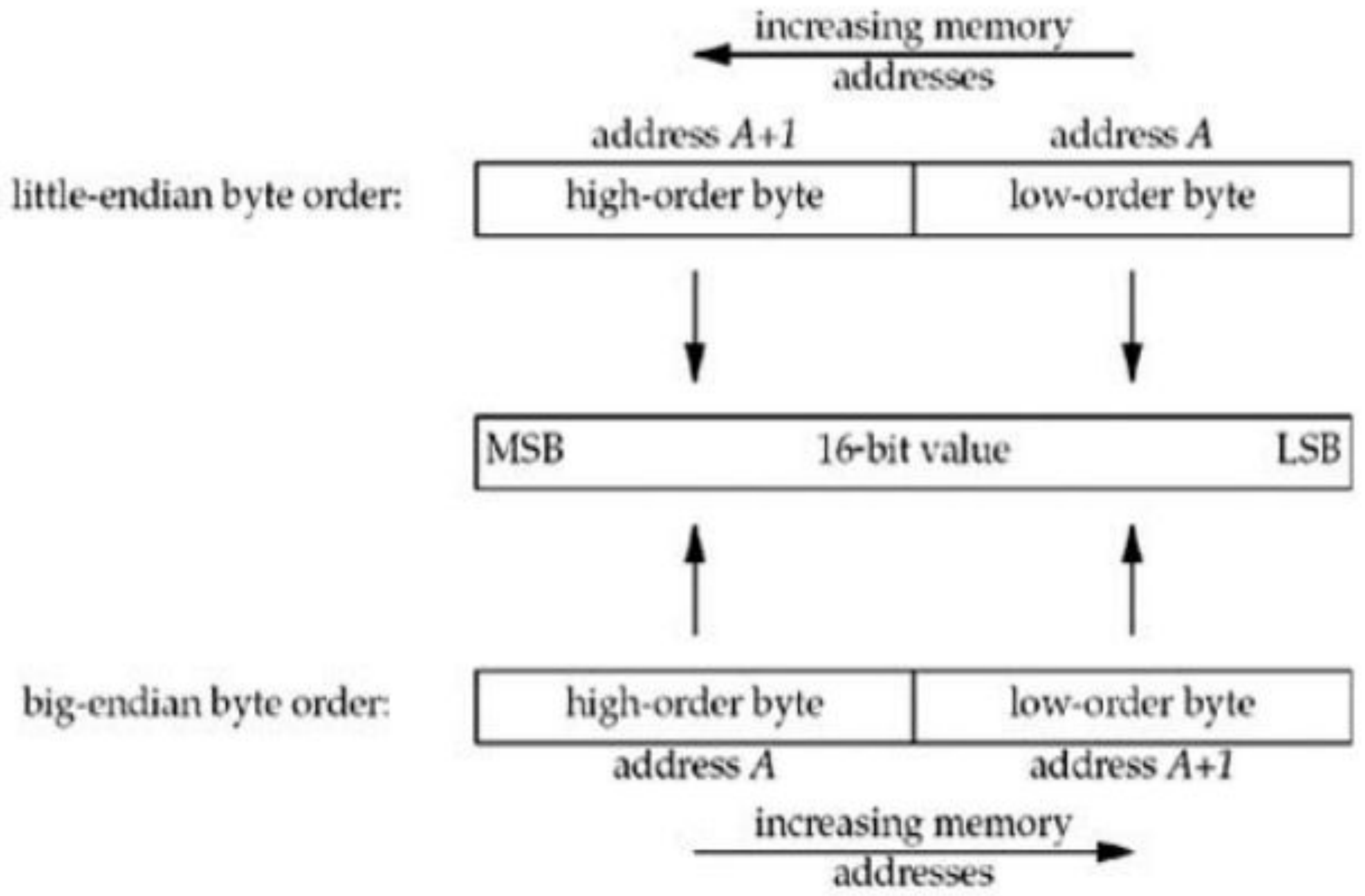
- Two ways to store bytes : Big endian and Little endian
    - Big-endian – High order byte is stored on starting address and low order byte is stored on next address
    - Little-endian – Low order byte is stored on starting address and high order byte is stored on next address

Byte ordering functions (contd.)

Example

- Consider a 16-bit integer that is made up of 2 bytes.
- There are two ways to store the two bytes in memory:
  - ✔ Big-endian byte order.
  - ✔ Little-endian byte order
- There is no standard between these two byte orderings and therefore systems encounter that use both formats.
- The byte ordering used by a given system is called as the **host byte order**.

*Example :*Little-endian byte order and big-endian byte order for a 16-bit integer.

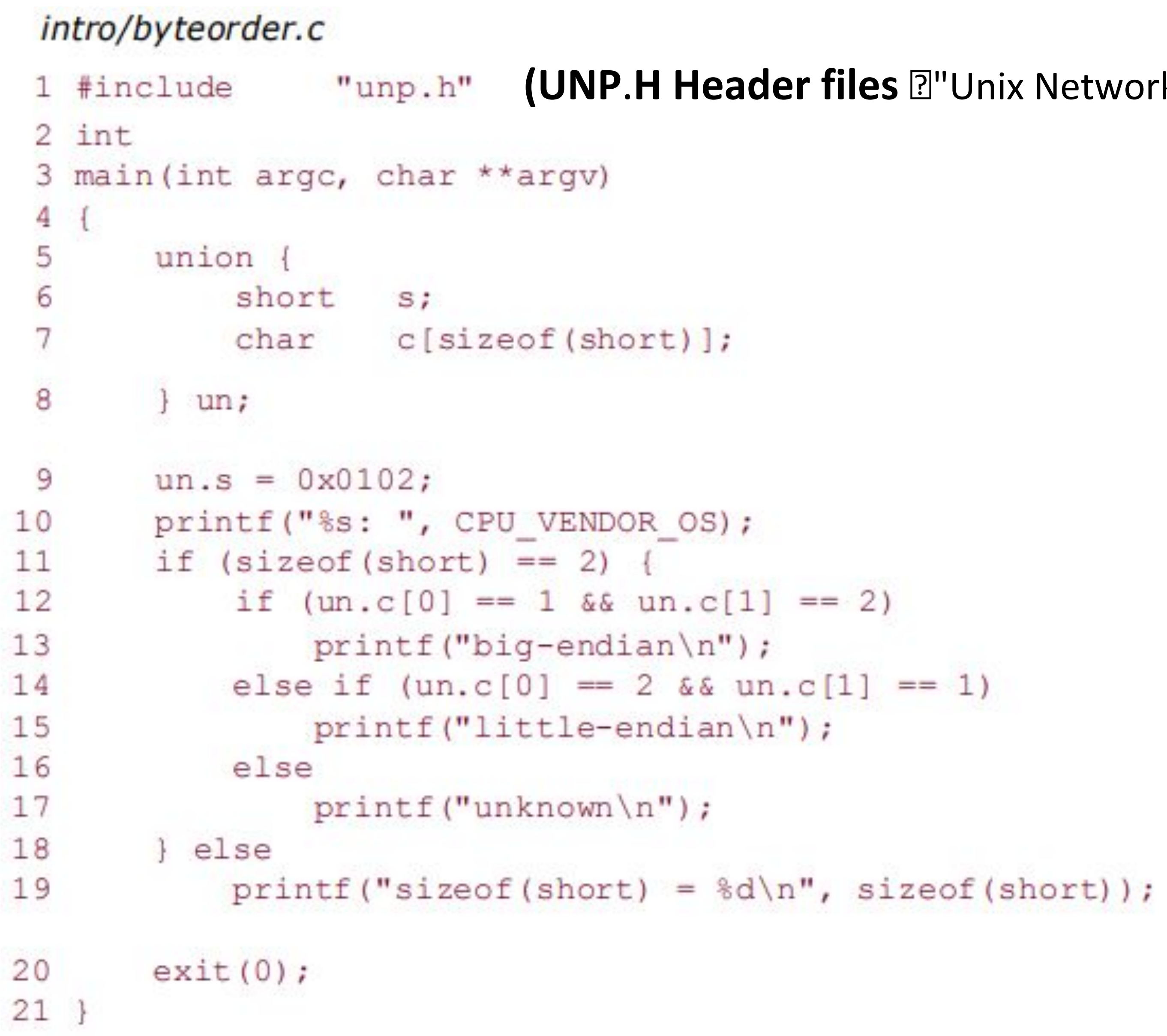


Byte ordering functions (contd.)

- Increasing memory addresses going from right to left in the top, and from left to right in the bottom.
- The most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.
  - ✔ The terms "little-endian" and "big-endian" indicate which end of the multibyte value
  - ✔ The little end or the big end, is stored at the starting address of the value
- There is no standard between these two byte orderings and systems encounter that use both formats.
- "Unix Network Programming" **UNP.H Header files**

Byte ordering functions (contd.)

Program to determine host byte order



intro/byteorder.c

```
#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
        printf("sizeof(short) = %d\n", sizeof(short));

    exit(0);
}
```

(**UNP.H Header files** □"Unix Network Programming“)

Byte ordering functions (contd.)

Output

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

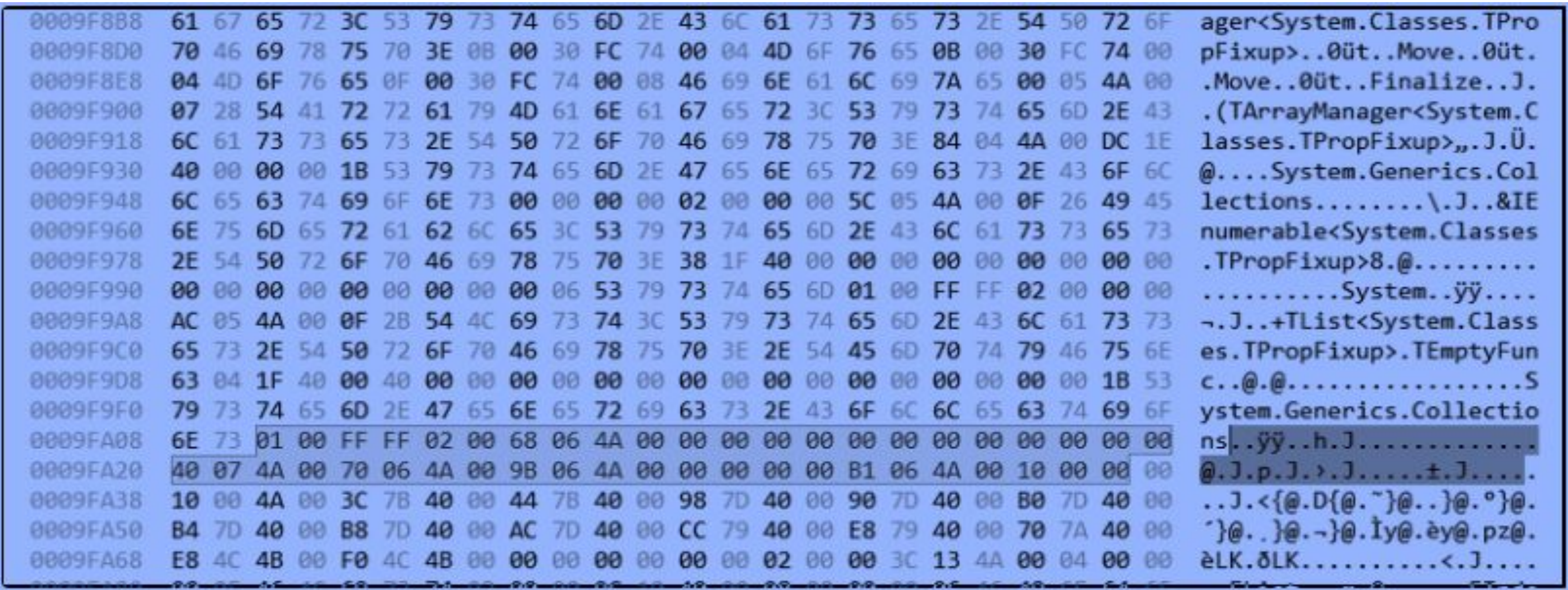
Byte ordering functions (contd.)

- The arrangement is same for a 32-bit integer.
- Currently a variety of systems that can change between little-endian and big endian byte ordering, sometimes at system reset, sometimes at run-time.
- Byte ordering differences as among networking protocols.
- 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.**

Byte ordering functions (contd.)

- In general, 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, certain fields in the socket address structures must be maintained in network byte order.
- Therefore converting between host byte order and network byte order is necessary.
- The following four functions are used to convert between these two byte orders.

2. Byte Ordering conversion functions

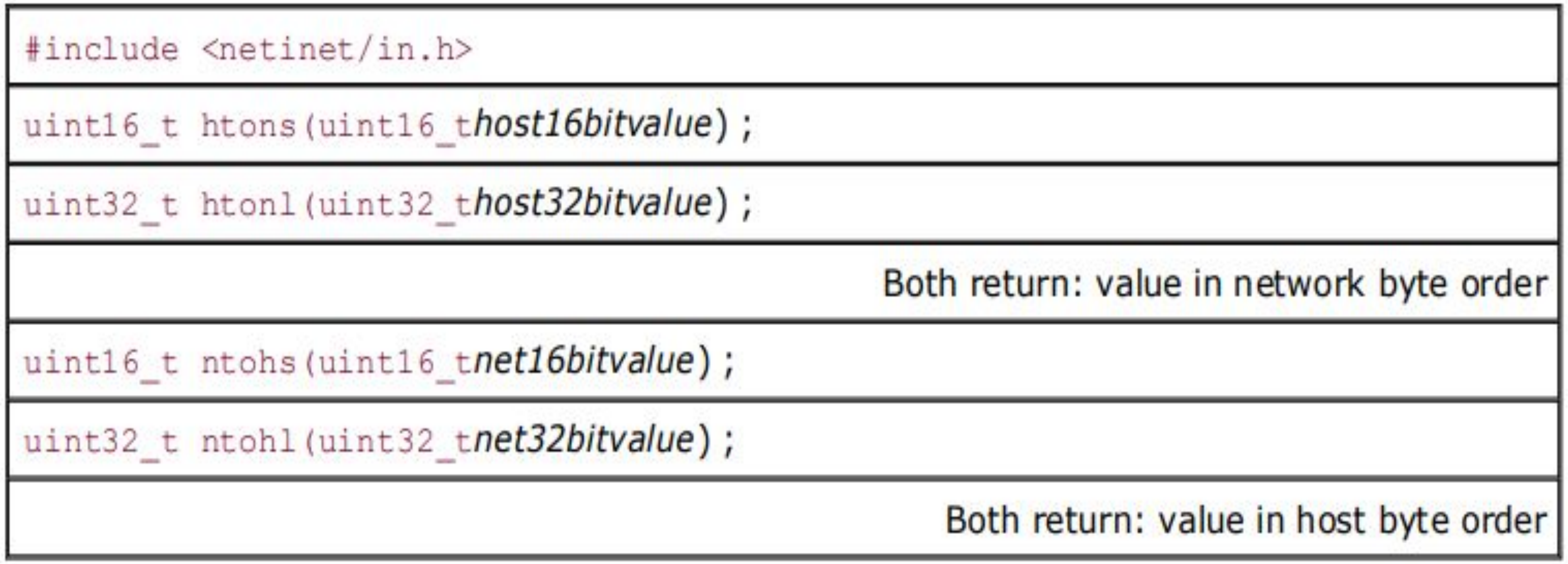


```
0009F8B8  61 67 65 72 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73 65 73 2E 54 50 72 6F  ager<System.Classes.TPro
0009F8D0  70 46 69 78 75 70 3E 0B 00 30 FC 74 00 04 4D 6F 76 65 0B 00 30 FC 74 00  pFixup>..0üt..Move..0üt.
0009F8E8  04 4D 6F 76 65 0F 00 30 FC 74 00 08 46 69 6E 61 6C 69 7A 65 00 05 4A 00  .Move..0üt..Finalize..J.
0009F900  07 28 54 41 72 72 61 79 4D 61 6E 61 67 65 72 3C 53 79 73 74 65 6D 2E 43  .(TArrayManager<System.C
0009F918  6C 61 73 73 65 73 2E 54 50 72 6F 70 46 69 78 75 70 3E 84 04 4A 00 DC 1E  lasses.TPropFixup>„.J.Ü.
0009F930  40 00 00 00 1B 53 79 73 74 65 6D 2E 47 65 6E 65 72 69 63 73 2E 43 6F 6C  @....System.Generics.Col
0009F948  6C 65 63 74 69 6F 6E 73 00 00 00 00 02 00 00 00 5C 05 4A 00 0F 26 49 45  lections........\.J..&IE
0009F960  6E 75 6D 65 72 61 62 6C 65 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73 65 73  numerable<System.Classes
0009F978  2E 54 50 72 6F 70 46 69 78 75 70 3E 38 1F 40 00 00 00 00 00 00 00 00 00  .TPropFixup>8.@.........
0009F990  00 00 00 00 00 00 00 00 00 06 53 79 73 74 65 6D 01 00 FF FF 02 00 00 00  ..........System..ÿÿ....
0009F9A8  AC 05 4A 00 0F 2B 54 4C 69 73 74 3C 53 79 73 74 65 6D 2E 43 6C 61 73 73  ¬.J..+TList<System.Class
0009F9C0  65 73 2E 54 50 72 6F 70 46 69 78 75 70 3E 2E 54 45 6D 70 74 79 46 75 6E  es.TPropFixup>.TEmptyFun
0009F9D8  63 04 1F 40 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1B 53  c..@.@.................S
0009F9F0  79 73 74 65 6D 2E 47 65 6E 65 72 69 63 73 2E 43 6F 6C 6C 65 63 74 69 6F  ystem.Generics.Collectio
0009FA08  6E 73 01 00 FF FF 02 00 68 06 4A 00 00 00 00 00 00 00 00 00 00 00 00 00  ns..ÿÿ..h.J.............
0009FA20  40 07 4A 00 70 06 4A 00 9B 06 4A 00 00 00 00 00 B1 06 4A 00 10 00 00 00  @.J.p.J.›.J.....±.J.....
0009FA38  10 00 4A 00 3C 7B 40 00 44 7B 40 00 98 7D 40 00 90 7D 40 00 B0 7D 40 00  ..J.<{@.D{@.˜}@..}@.°}@.
0009FA50  B4 7D 40 00 B8 7D 40 00 AC 7D 40 00 CC 79 40 00 E8 79 40 00 70 7A 40 00  ´}@.¸}@.¬}@.Ìy@.èy@.pz@.
0009FA68  E8 4C 4B 00 F0 4C 4B 00 00 00 00 00 00 00 02 00 00 3C 13 4A 00 04 00 00  èLK.ðLK.........<.J....
```

Conversion functions

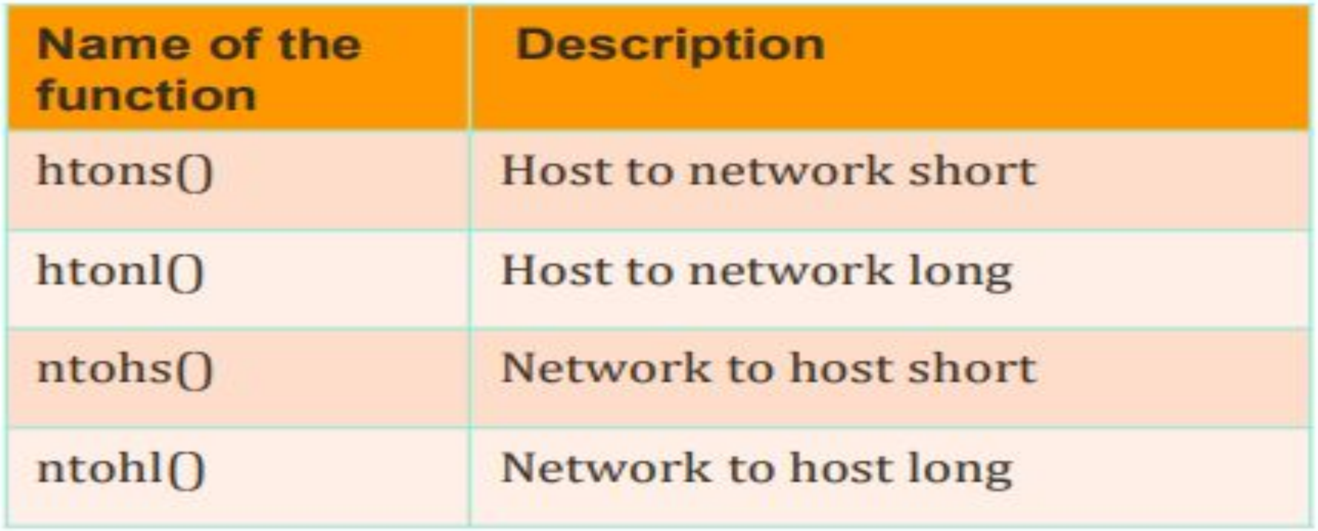
- In general, an implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail.
- But, certain fields in the socket address structures must be maintained in network byte order.
- Therefore converting between host byte order and network byte order is necessary.
- The following four functions are used to convert between these two byte orders.

Conversion functions (cont.)

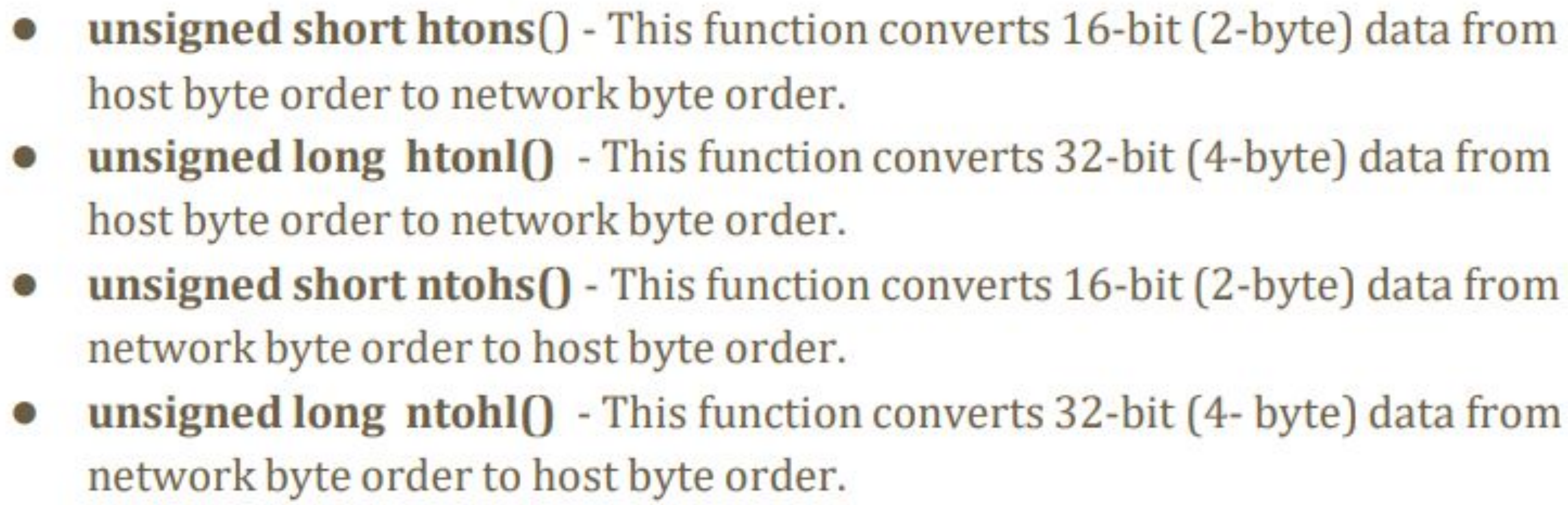


```
#include <netinet/in.h>

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
```

Conversion functions (cont.)



Name of the function	Description
htons()	Host to network short
htonl()	Host to network long
ntohs()	Network to host short
ntohl()	Network to host long



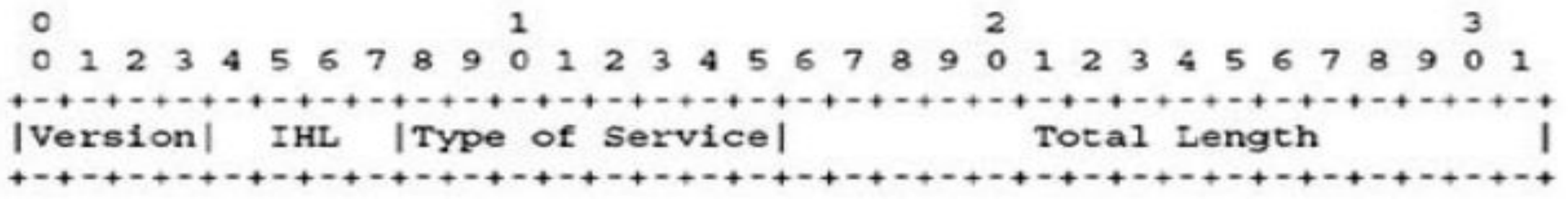
- **unsigned short htons**() - This function converts 16-bit (2-byte) data from host byte order to network byte order.
- **unsigned long htonl()** - This function converts 32-bit (4-byte) data from host byte order to network byte order.
- **unsigned short ntohs()** - This function converts 16-bit (2-byte) data from network byte order to host byte order.
- **unsigned long ntohl()** - This function converts 32-bit (4- byte) data from network byte order to host byte order.

Conversion functions (cont.)

- From these functions, **h** stands for host, **n** stands for network, **s** stands for short, and **l** stands for long.
- In general **s** as a 16-bit value (such as a TCP or UDP port number) and **l** as a 32-bit value (such as an IPv4 address); The htonl and ntohl functions operate on 32-bit values
- 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

Conversion functions (cont.)

- The term **"byte"** means an 8-bit quantity since almost all current computer systems use 8-bit bytes.
- But most Internet standards use the term **octet** instead of byte to mean an 8-bit quantity.
- This started in the early days of TCP/IP because much of the early work was done on systems such as the DEC-10 (Digital Equipment Corporation), which did not use 8-bit bytes.
- Another important convention in Internet standards is bit ordering.
- Exa **all)**



```
 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         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

3. System Calls

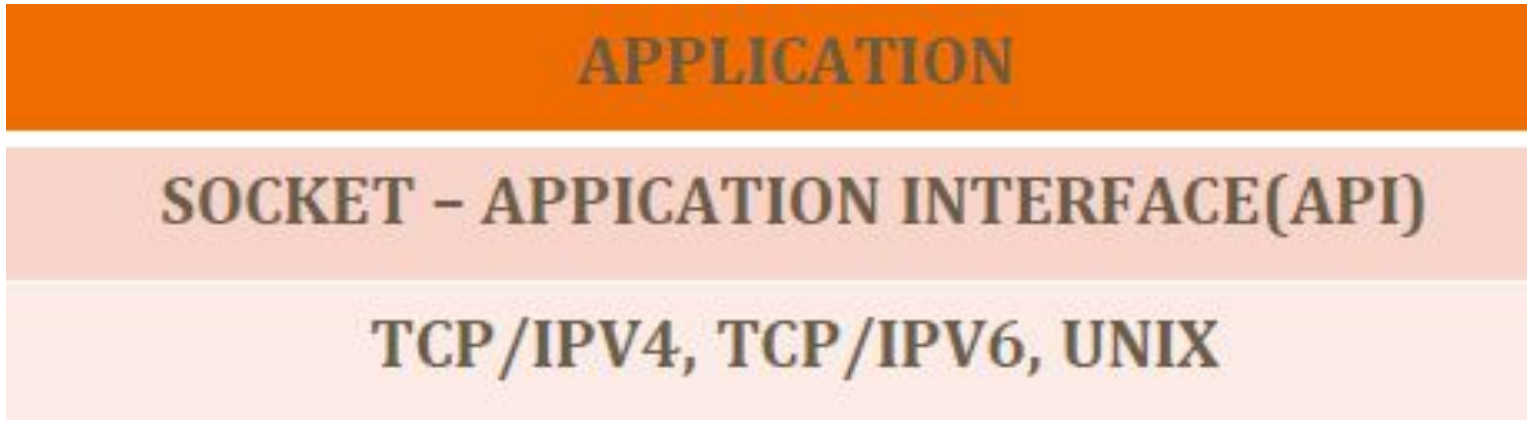
System calls

- An interface between process and operating systems
- It provides
  - ✔ the services of the OS to the user programs via Application Program Interface (API)
  - ✔ An interface to allow user level processes to request services of the OS
  - ✔ System calls are the only entry points into the kernel system

4. Sockets

Sockets

- A *socket* is one endpoint of a two-way communication link between two programs running on the network.
- A Socket is an interface between applications and the network services provided by operating systems
- Applications use sockets to send and receive the data
- Socket provides IP address and port address

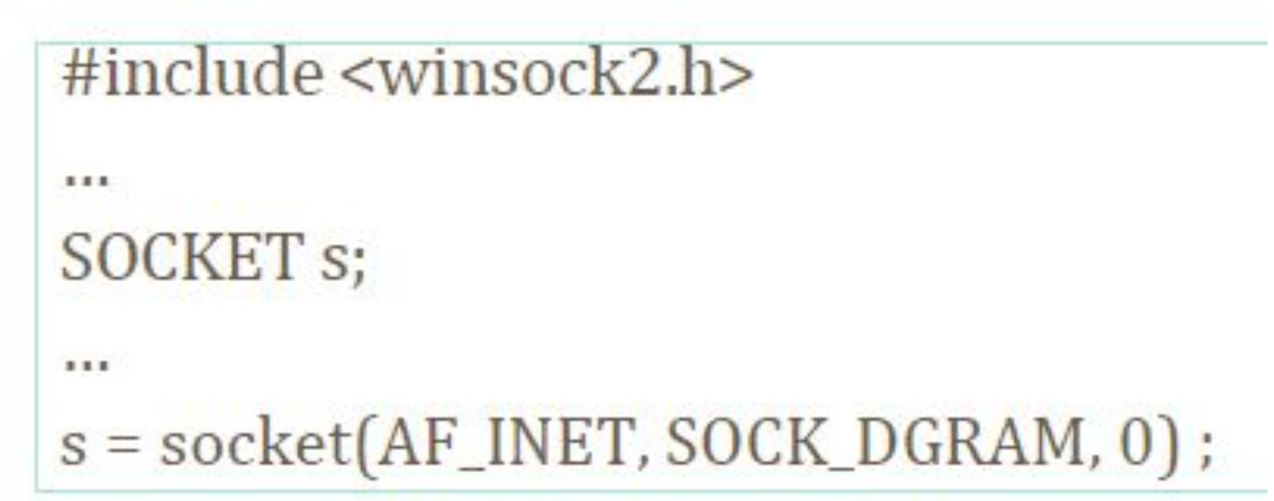


Sockets Descriptors

- To perform file I/O, file descriptor is used
- To perform network I/O, socket descriptor is used
- Each active socket is identified by its socket descriptor
- The data type of a socket descriptor is ***SOCKET***
- Hack into the header file ***winsock2.h***
- Type ***def u_int SOCKET***; **u_int** is defined as unsigned int
- In UNIX systems, socket is just a special file, and socket descriptors are kept in the file descriptor table
- The Windows operating system keeps a separate table of socket descriptors (named socket descriptor table or SDT) for each process

Sockets Creation

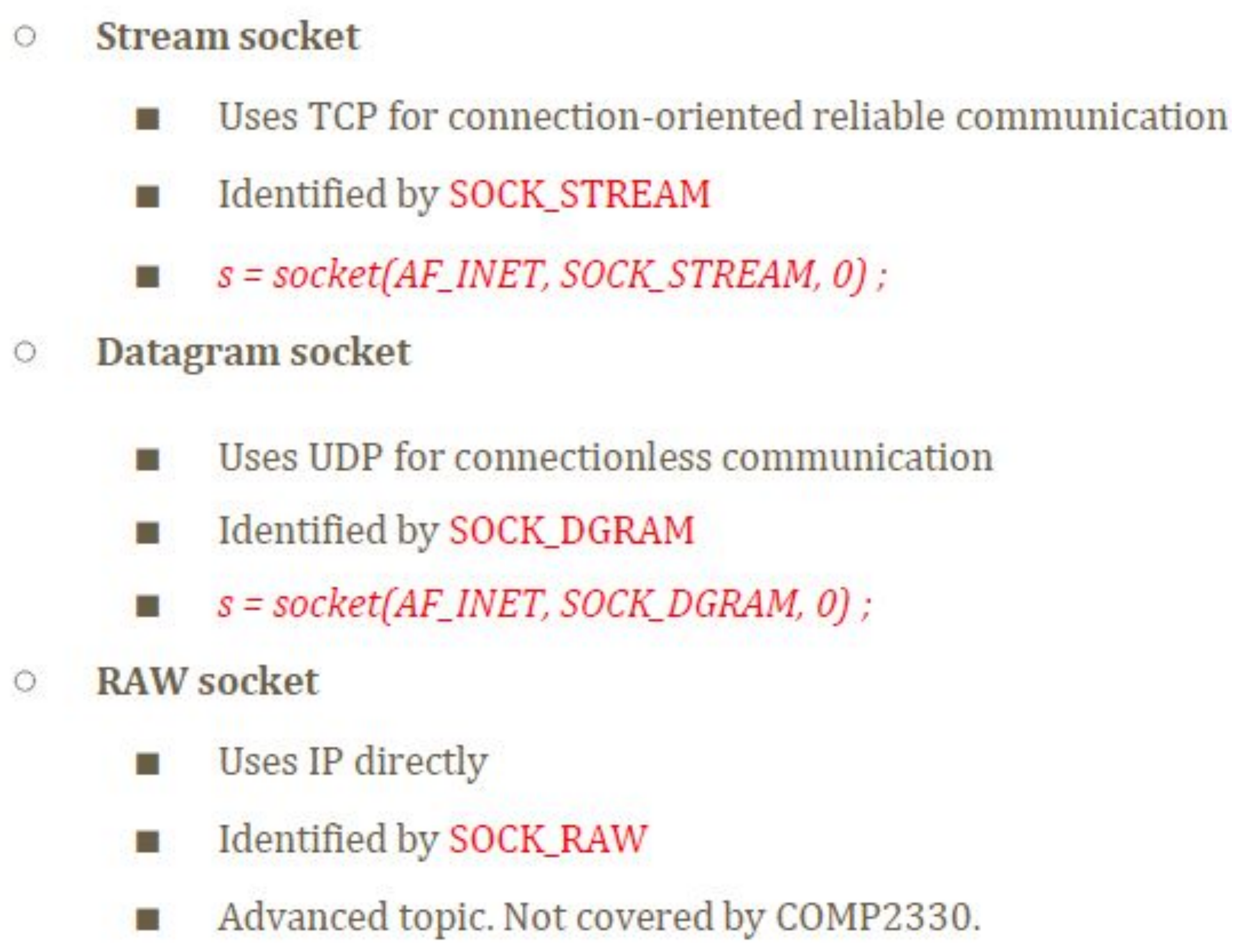
- The socket API contains a function socket()that can be called to create a socket. e.g.:



```
#include <winsock2.h>
...
SOCKET s;
...
s = socket(AF_INET, SOCK_DGRAM, 0) ;
```

Types of socket

- Under protocol family AF_INET



- Stream socket
  - Uses TCP for connection-oriented reliable communication
  - Identified by SOCK_STREAM
  - *s = socket(AF_INET, SOCK_STREAM, 0) ;*
- Datagram socket
  - Uses UDP for connectionless communication
  - Identified by SOCK_DGRAM
  - *s = socket(AF_INET, SOCK_DGRAM, 0) ;*
- RAW socket
  - Uses IP directly
  - Identified by SOCK_RAW
  - Advanced topic. Not covered by COMP2330.

System Data structures for sockets

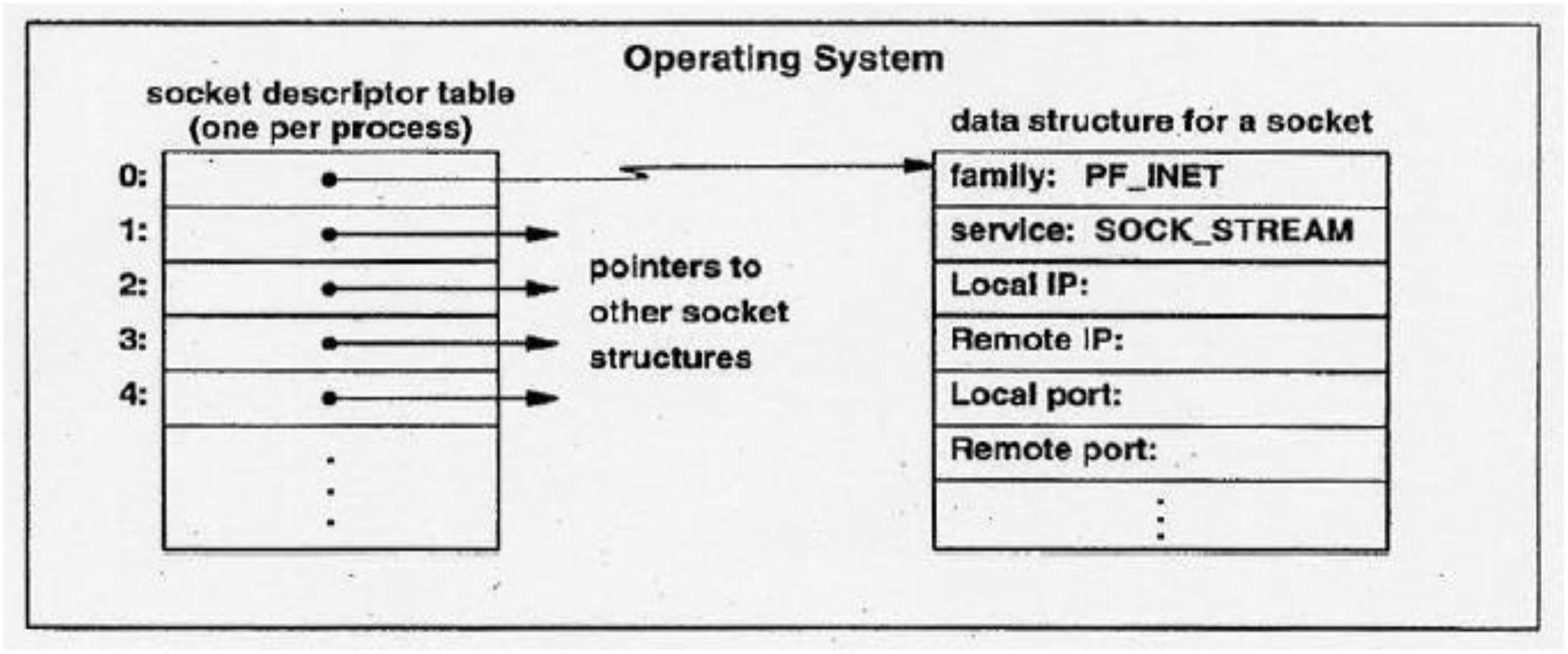
- When an application process calls socket(), the operating system allocates a new data structure to hold the information needed for communication, and fills in a new entry in the process's socket descriptor table **(SDT)** with a pointer to the data structure**.**
  - A process may use multiple sockets at the same time. The socket descriptor table is used to manage the sockets for this process.
  - Different processes use different SDTs.

System Data structures for sockets (cont.)

- The internal data structure for a socket contains many fields, but the system leaves most of them unfilled. The application must make additional procedure calls to fill in the socket data structure before the socket can be used.
- The socket is used for data communication between two processes (which may locate at different machines). So the socket data structure should at least contain the address information, e.g., **IP addresses, port numbers, etc.**

System Data structures for sockets (cont.)

- Conceptual operating system (Windows) data structures after five callsto *socket() by a process. The system keeps a separate socket descriptor table for each process; threads in the process share the table.*



5. System calls used with sockets

Elementary socket system calls

- To do network I/O, the first thing a process must do is to call the **socket** system call, specifying the type of communication protocol desired.

  #include <sys/types.h>

  #include <sys/socket.h>

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

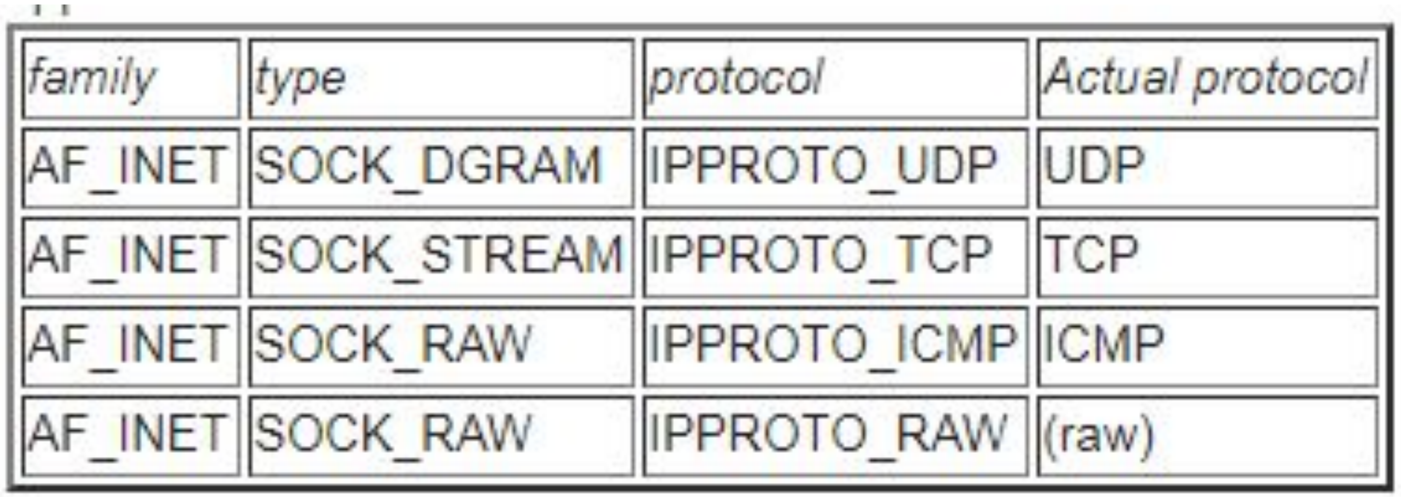
- The *family* is one of (The AF_ prefix stands for "address family.“)
  - AF_UNIX -- Unix internal protocols
  - **AF_INET -- Internet protocols**
  - AF_NS -- Xerox NS Protocols
  - AF_IMPLINK -- IMP link layer

Elementary socket system calls

- The socket *type* is one of the following:
  - SOCK_STREAM stream socket
  - SOCK_DGRAM datagram socket
  - SOCK_RAW raw socket
  - SOCK_SEQPACKET sequenced packet socket
  - SOCK_RDM reliably delivered message socket (not implemented yet)

Elementary socket system calls

- The *protocol* argument to the socket system call is typically set to 0 for most user applications. The valid combinations are shown as follows.



family	*type*	*protocol*	*Actual protocol*
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(raw)

Bind

- Bind socket to the local address and port
  - The **bind** system call assigns a name to an unnamed socket.

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

#include <sys/socket.h>

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

  - The first argument is the socket descriptor returned from **socket** system call.
  - The second argument is a pointer to a protocol-specific address
  - The third argument is the size of this address.
  - Returns 0 on success, and -1 if an error occurs

Bind uses

- Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.
- A client can register a specific address for itself.
- A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we sent the letter to.

Connect

- A client process **connect**s a socket descriptor following the **socket** system call to establish a connection with a server.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

- The *sockfd* is a socket descriptor that was returned by the **socket** system call.
- The second and third arguments are a pointer to a socket address, and its size

Connect uses

- For most connection-oriented protocols (TCP, for example), the **connect** system call results in the actual establishment of a connection between the local system and the foreign system.
- The **connect** system call does not return until the connection is established, or an error is returned to the process.
- The client does not have to **bind** a local address before calling **connect**.
- The connection typically causes these four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*,*foreign-addr*, and *foreign-process*.
- In all the connection-oriented clients, we will let **connect** assign the local address.

Listen

- This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- It is usually executed after both the **socket** and **bind** system calls, and immediately before the **accept** system call.
- The *backlog* argument specifies how many connection requests can be queued by the system while it waits for the server to execute the **accept** system call.
- This argument is usually specified as 5, the maximum value currently allowed.

Accept

- After a connection-oriented server executes the **listen** system call described above, an actual connection from some client process is waited for by having the server execute the **accept** system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

- **accept** takes the first connection request on the queue and creates another socket with the same properties as *sockfd*.
- If there are no connection requests pending, this call blocks the caller until one arrives.

Accept

- The *peer* and *addrlen* arguments are used to return the address of the connected peer process (the client).
- *addrlen* is called a value-result argument: the caller sets its value before the system call, and the system call stores a result in the variable.
- For this system call the caller sets *addrlen* to the size of the **sockaddr** structure whose address is passed as the *peer* argument.

Send, sendto,recv, recvfrom

- These system calls are similar to the standard **read** and **write** system calls, but additional arguments are required.

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, char *buff, int nbytes, int flags);
int sendto(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *to, int
    addrlen);
int recv(int sockfd, char *buff, int nbytes, int flags);
int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int
    *addrlen);
```

Send, sendto,recv, recvfrom

- The first three arguments, *sockfd*, *buff*, and *nbytes*, to the four system calls are similar to the first three arguments for **read** and **write**.
- The *flags* argument can be safely set to zero ignoring the details for it.
- The *to* argument for **sendto** specifies the protocol-specific address of where the data is to be sent.
- Since this address is protocol-specific, its length must be specified by *addrlen*.
- The **recvfrom** system call fills in the protocol-specific address of who sent the data into *from*. The length of this address is also returned to the caller in *addrlen*.
- The final argument to **sendto** is an integer value, while the final argument to **recvfrom** is a pointer to an integer value.

Close

- The normal Unix **close** system call is also used to close a socket.

  int close(int sock*fd*);

- If the socket being closed is associated with a protocol that promises reliable delivery (e.g., TCP or SPP), the system must assure that any data within the kernel that still has to be transmitted or acknowledged, is sent.
- Normally, the system returns from the **close** immediately, but the kernel still tries to send any data already queued.

6. Iterative and concurrent server

Overview

- The server *iterates* through each client, one at a time is called iterative server
- There are numerous techniques for writing a *concurrent server, one that handles multiple clients at the same time.*
  - *The simplest* technique a concurrent server is to call *fork* function, creating one child process for each client.
  - Other techniques are to use *threads* instead of fork or to pre-fork a fixed number of children when the server starts.
- 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 is to fork a child process to handle each client.*
- Most TCP servers are concurrent, with the server calling fork for every client connection that it handles. While most UDP servers are iterative.

Concurrency

a. Concurrency in Clients

- Clients can run on a machine either iteratively or concurrently.
- Running clients *iteratively* means running them one by one; one client must start, run, and terminate before the machine can start another client.
- Most computers today allow *concurrent clients*; that is, two or more clients can run at the same time.

Concurrency (cont.)

b. Concurrency in Servers

- An *iterative server* can process only one request at a time; it receives a request, processes it, and sends the response to the requestor before it handles another request.
- A *concurrent server*, on the other hand, can process many requests at the same time and thus can share its time between many requests.

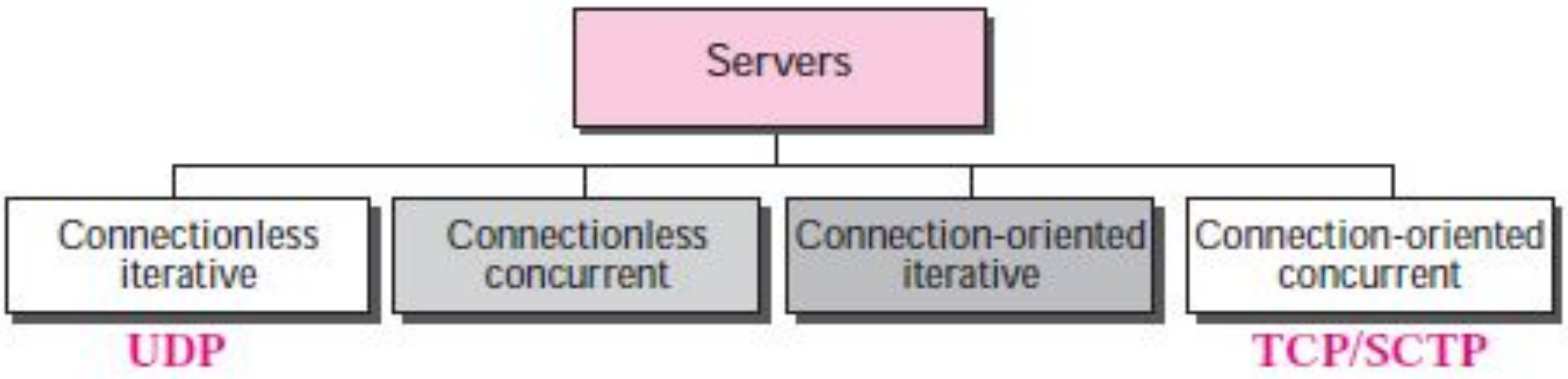
Concurrency (cont.)

b. Concurrency in Servers (Cont.)

- The servers use either UDP, a connectionless transport layer protocol, or TCP/SCTP, a connection-oriented transport layer protocol.
- Server operation, depends on two factors:
  - the transport layer protocol and
  - the service method.
- Four types of servers: connectionless iterative, connectionless concurrent, connection-oriented iterative, and connection-oriented concurrent

Concurrency (cont.)

Server types



Servers
Connectionless iterative
Connectionless concurrent
Connection-oriented iterative
Connection-oriented concurrent
UDP
TCP/SCTP

Concurrency-Server types (cont.)

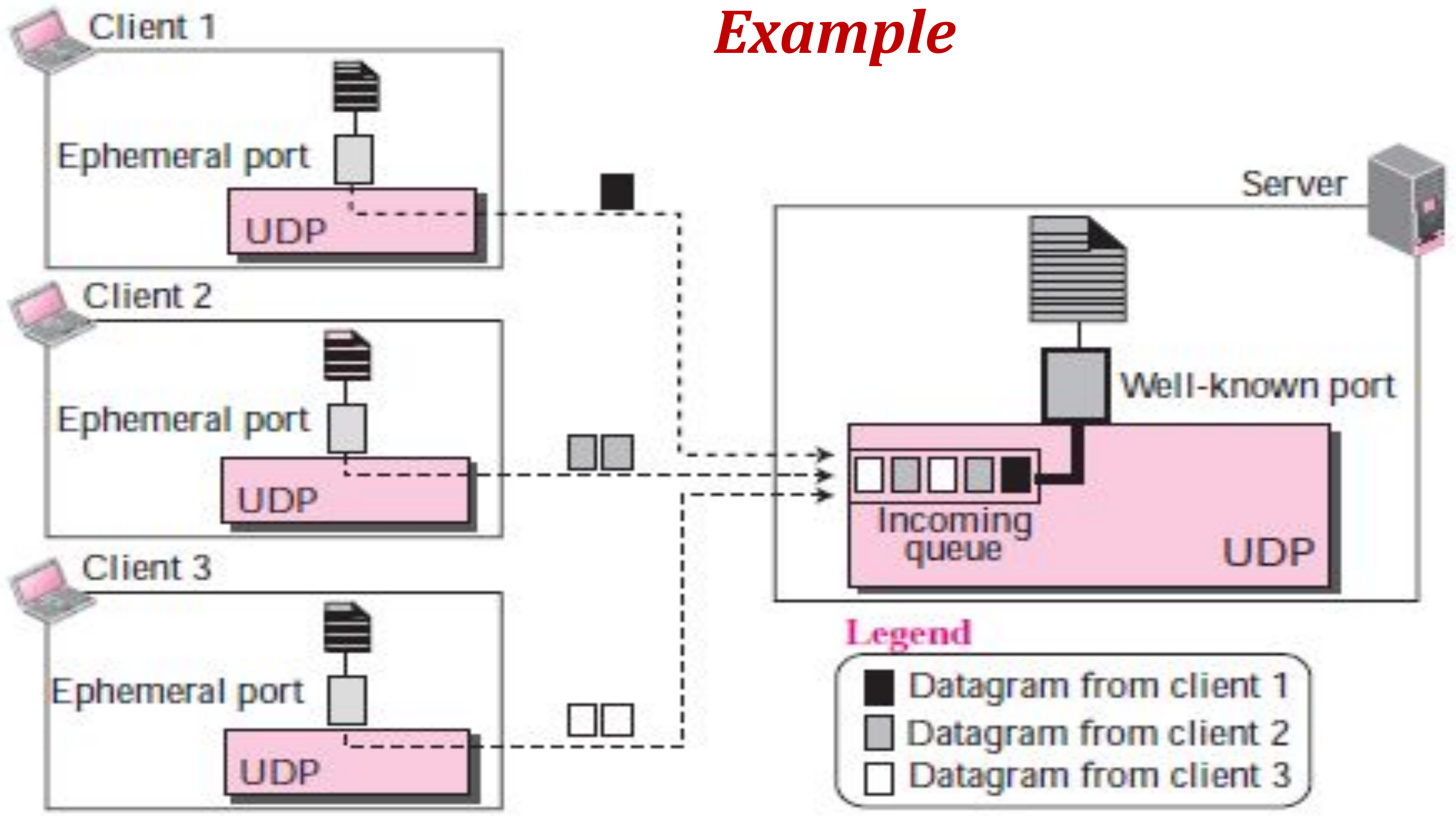
i. Connectionless Iterative Server

- The servers that use UDP are normally iterative; i.e., the server processes one request at a time.
- A server gets the request received in a datagram from UDP, processes the request, gives the response to UDP to send to client.
- The server pays no attention to the other datagrams.
- These datagrams are stored in a queue, waiting for service.
- They could all be from one client or from many clients and processed one by one in order of arrival.
- The server uses one single port i.e., the well-known port.
- All the datagrams arriving at this port wait in line to be served

Concurrency-Server types (cont.)

i. Connectionless Iterative Server (Cont.)

Example



Concurrency-Server types (cont.)

ii. Connection-Oriented Concurrent Server

- The servers that use TCP (or SCTP) are normally concurrent.
- Here the server can serve many clients at the same time.
- Communication is connection-oriented, which means that a request is a stream of bytes that can arrive in several segments and the response can occupy several segments.
- A connection is established between the server and each client, and the connection remains open until the entire stream is processed and the connection is terminated.
- This type of server cannot use only one port because each connection needs a port and many connections may be open at the same time.

Concurrency-Server types (cont.)

ii. Connection-Oriented Concurrent Server (Cont.)

- Many ports are needed, but a server can use only one well-known port.
- The solution is to have one well-known port and many ephemeral ports.
- The server accepts connection requests at the well-known port.
- A client can make its initial approach to this port to make the connection.
- After the connection is made, the server assigns a temporary port to this connection to free the well-known port.
- Data transfer can now take place between these two temporary ports, one at the client site and the other at the server site.

Concurrency-Server types (cont.)

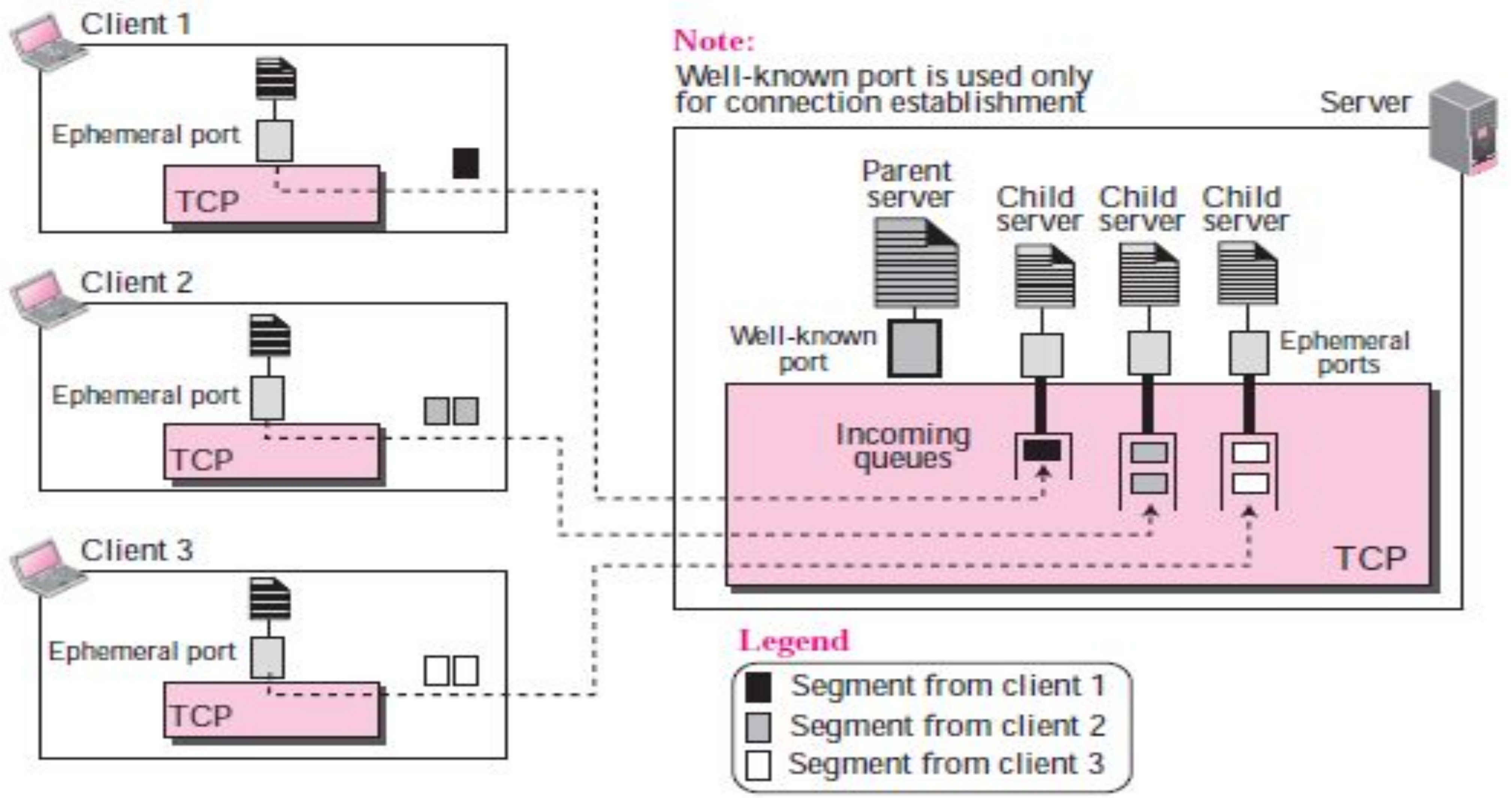
ii. Connection-Oriented Concurrent Server (Cont.)

- The well-known port is now free for another client to make the connection.
- To serve several clients at the same time, a server creates child processes, which are copies of the original process (parent process).
- The server must also have one queue for each connection.
- The segments come from the client, are stored in the appropriate queue, and will be served concurrently by the server.

Concurrency-Server types (cont.)

ii. Connection-Oriented Concurrent Server (Cont.)

Example



7. Socket Interface

Structure and Functions of socket

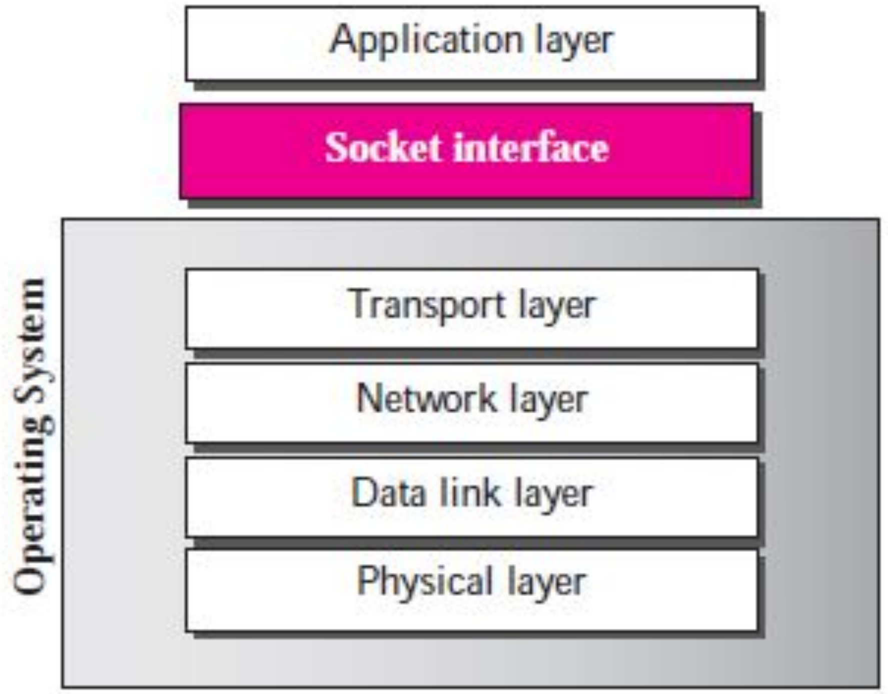
Socket Interface

- How can a client process communicate with a server process?
- A computer program is a set of predefined instructions that tells the computer what to do.
- A computer program has a set of instructions for mathematical operations, another set of instructions for string manipulation, still another set of instructions for input/output access.
- If we need a program to be able to communicate with another program running on another machine, we need a new set of instructions to tell the transport layer to open the connection, send data to and receive data from the other end, and close the connection.
- A set of instructions of this kind is normally referred to as an **interface.**
- **An interface is a set of instructions designed for interaction between two entities.** i.e., (operating system and the application programs)

Socket Interface (cont.)

- Several interfaces have been designed for communication. Among them three are common: **socket interface, transport layer interface (TLI), and STREAM.**
- Socket interface started in early 1980s at the University of Berkeley as part of a UNIX environment.
- The socket interface, as a set of instructions, is located between the operating system and the application programs.
- To access the services provided by the TCP/IP protocol suite, an application needs to use the instructions defined in the socket interface.
- **Example:** *file interface*

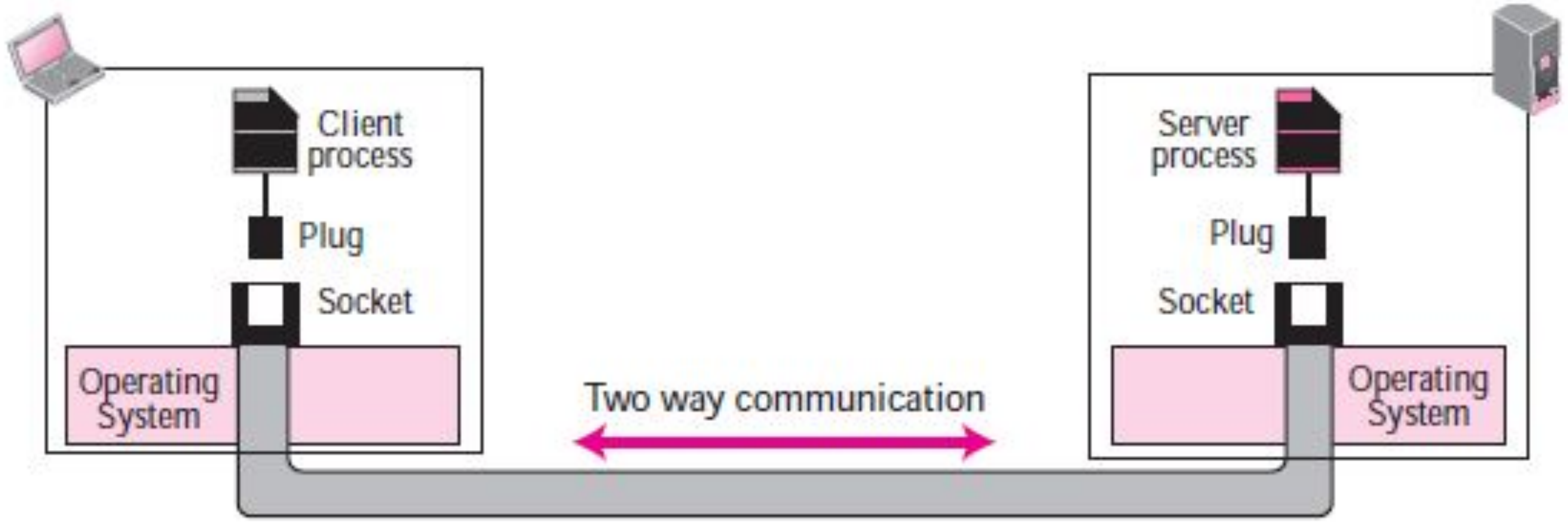
Relation between the operating system and the TCP/IP suite



Socket

- A **socket is a software abstract simulating a hardware socket.**
- To use the communication channel, an application program (client or server) needs to request the operating system to create a socket.
- The application program then can *plug* into the socket to send and receive data.
- For data communication to occur, a pair of sockets, each at one end of communication, is needed.
- Example: a telephone, in the Internet a socket is a software data

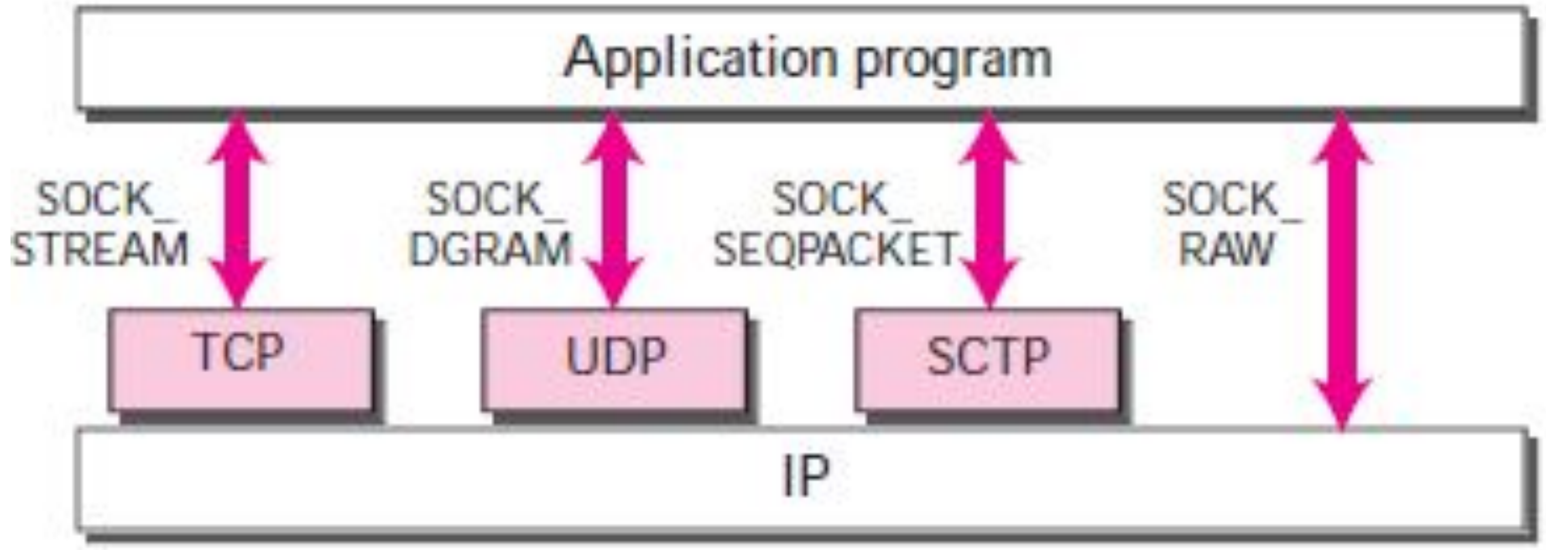
Concept of sockets



Data Structure

- The format of data structure to define a socket depends on the underlying language used by the processes.
- For discussion assume that the processes are written in C language. In C language, a socket is defined as a five-field structure (struct or record)
- The programmer should not redefine this structure; it is already defined.
- The programmer needs only to use the header file that includes this definition

Socket types

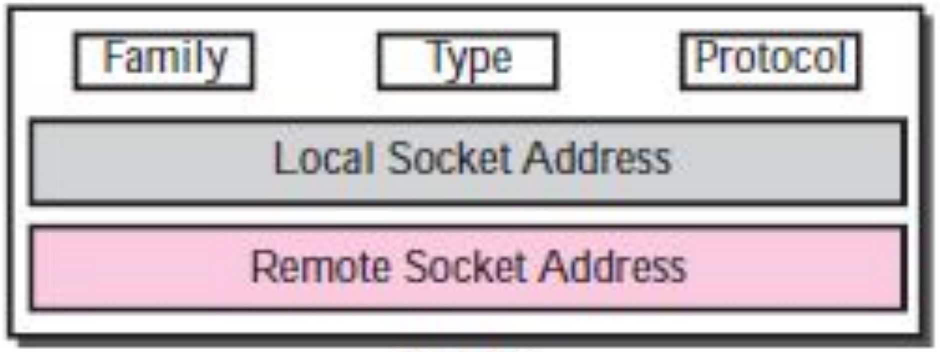


Data Structure (cont.)

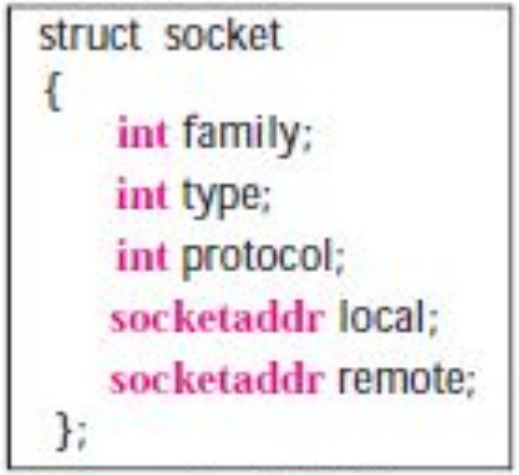
- The field used in this structure are

- ❑ **Family:**
  - o This field defines the protocol group: IPv4, IPv6, UNIX domain protocols, and so on.
  - o The family type we use in TCP/IP is defined by the constant IF_INET for IPv4 protocol and IF_INET6 for IPv6 protocol.
- ❑ **Type:**
  - o This field defines four types of sockets: SOCK_STREAM (for TCP), SOCK_DGRAM (for UDP), SOCK_SEQPACKET (for SCTP), and SOCK_RAW (for applications that directly use the services of IP).
- ❑ **Protocol:** This field defines the protocol that uses the interface. It is set to 0 for TCP/IP protocol suite.
- ❑ **Local socket address:** This field defines the local socket address; i.e., A combination of an IP address and a port number.
- ❑ **Remote socket address:** This field defines the remote socket address.

Socket Data Structure



Fields



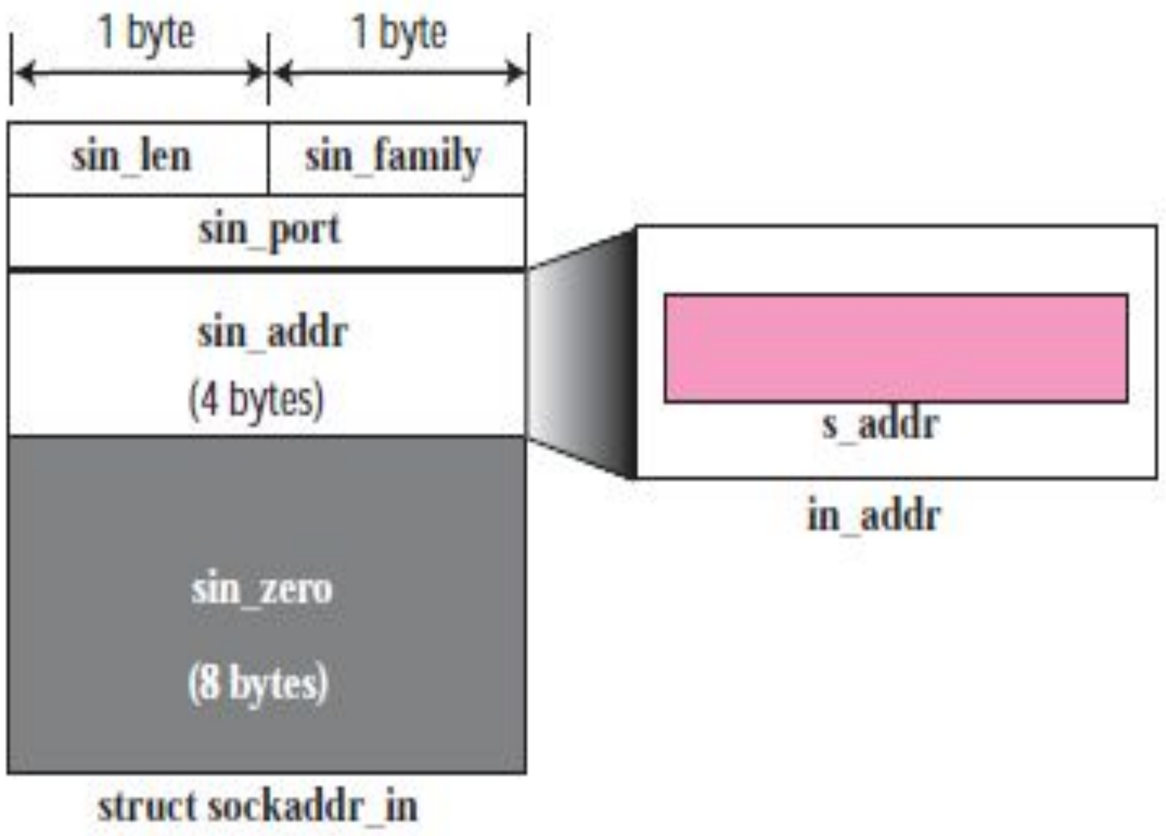
```
struct  socket
{
        int family;
        int type;
        int protocol;
        socketaddr local;
        socketaddr remote;
 };
```

Generic definition

Structure of Socket Address

- We need to understand the structure of a socket address, a combination of IP address and port number.
- Several types of socket addresses have been defined in network programming, we define only the one used for IPv4. In this version a socket address is a complex data structure (struct in C)
- The *struct sockaddr_in* has five fields, but the *sin_addr* field is itself a *struct* of type ***in_addr*** with a one single field *s_addr*.

IPv4 socket address



```
struct  in_addr
{
    in_addr_t    s_addr;    // A 32-bit IPv4 address
}
```

```
struct  sockaddr_in
{
    uint8_t           sin_len;        // length of structure (16 bytes)
    sa_family_t       sin_family;     // set to AF_INET
    in_port_t         sin_port;       // A 16-bit port number
    struct in_addr    sin_addr;       // A 32-bit IPv4 address
    char              sin_zero[8];    // unused
}
```

Socket Functions

- The interaction between a process and the operating system is done through a list of predefined functions
  - The *socket* Function
  - The *bind* Function
  - The *Connect* Function
  - The *listen* Function
  - The *accept* Function
  - The *fork* function
  - The *send and recv* Functions
  - The *sendto and recvfrom* Functions
  - The *close* Function

Socket Functions (cont.) *1. The socket Function*

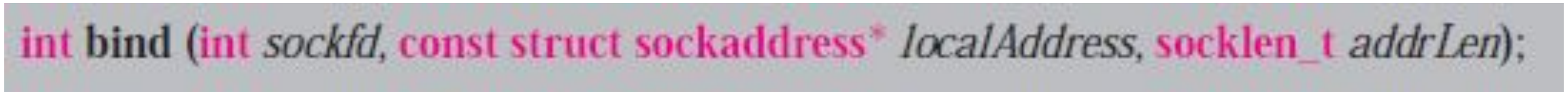
- The operating system (OS) defines the socket structure
- The OS, does not create a socket until instructed by the process.
- The process needs to use the *socket function call to create a socket.*
- *The prototype for* this function is shown here:

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

- If the call is successful, the function returns a unique socket descriptor *sockfd (a non-negative integer) that can be used to refer to* the socket in other calls;
- if the call is not successful, the OS returns −1.

Socket Functions (cont.) *2. The bind Function*

- To bind the socket to the local computer and local port, the *bind function needs to be called.*
- *The bind function,* fills the value for the local socket address (local IP address and local port number). It returns −1 if the binding fails.

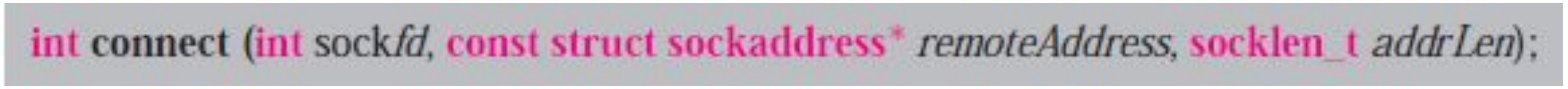


```
int bind (int sockfd, const struct sockaddress* localAddress, socklen_t addrLen);
```

- *sockfd value of the socket descriptor returned from the* socket function call
- *localAddress pointer to a socket address that needs to have* been defined (by the system or the programmer),
- *addrLen length of* the socket address

Socket Functions (cont.) *3. The connect Function*

- The connect function is used to add the remote socket address to the socket structure. It returns −1 if the connection fails.

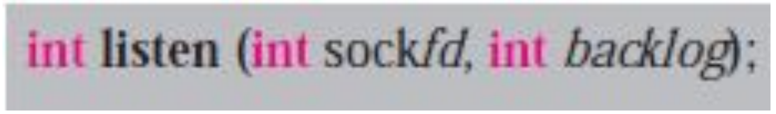


```
int connect (int sockfd, const struct sockaddress* remoteAddress, socklen_t addrLen);
```

- *sockfd value of the socket descriptor returned from the* socket function call
- Second and third argument defines the remote address instead of the local

Socket Functions (cont.) *4. The listen Function*

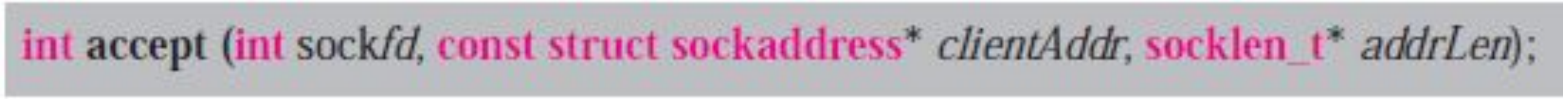
- The listen function is called only by the TCP server.
- After TCP has created and bound a socket, it must inform the operating system that a socket is ready for receiving client requests.
- This is done by calling the *listen function.*
- *The backlog is the maximum number* of connection requests. The function returns −1 if it fails.



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

Socket Functions (cont.) *5. The accept Function*

- The accept function is used by a server to inform TCP that it is ready to receive connections from clients. This function returns −1 if it fails.



```
int accept (int sockfd, const struct sockaddress* clientAddr, socklen_t* addrLen);
```

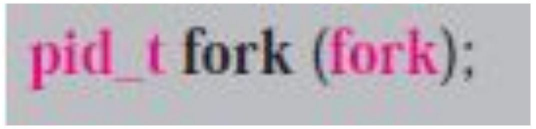
- The last two arguments are pointers to address and to length.
- The accept function is a blocking function that, when called, blocks itself until a connection is made by a client.
- The accept function then gets the client socket address and the address length and passes it to the server process to be used to access the client.

Socket Functions (cont.) *5. The accept Function* (cont.)

a. Accept function makes the process check if there is any client connection request in the waiting buffer. If not, the accept makes the process to sleep. The process wakes up when the queue has at least one request.
b. After a successful call to the accept, a new socket is created and the communication is established between the client socket and the new socket of the server.
c. The address received from the *accept function fills the remote socket address in* the new socket.
d. The address of the client is returned via a pointer. If the programmer does not need this address, it can be replaced by NULL.
e. The length of address to be returned is passed to the function and also returned via a pointer. If this length is not needed, it can be replaced by NULL.

Socket Functions (cont.) *6. The fork function*

- The *fork function is used by a process to duplicate a process.*
- *The process that calls* the *fork function is referred to as the parent process; the process that is created, the* duplicate, is called the child process
- The fork process is called once, but it returns twice.



```
pid_t fork (fork);
```

- In the parent process, the return value is a positive integer (the process ID of the parent process that called it).
- In the child process, return value is 0. If error, the fork function returns −1.
- After the fork, two processes are running concurrently; CPU gives running time to each process alternately.

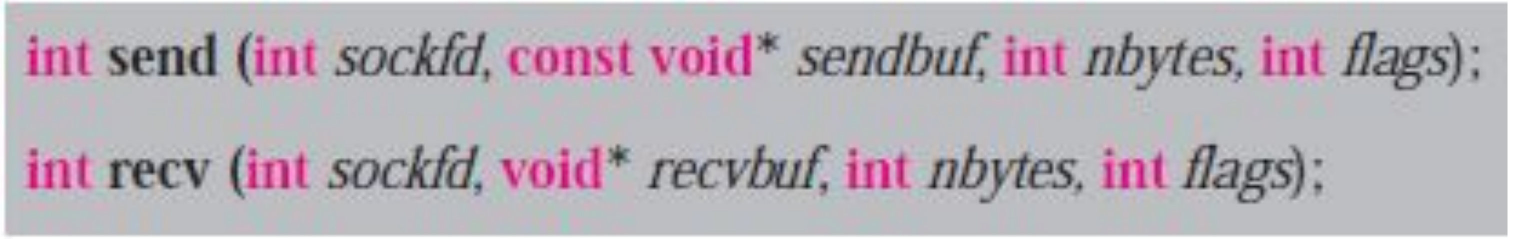
Socket Functions (cont.) *7. The send* and *recv Function*

- The *send function* is used by a process to send data to another process running on a remote machine.
- The *recv* function is used by a process to receive data from another process running on a remote machine.
- These functions assume that there is already an open connection between two machines; therefore, it can only be used by TCP (or SCTP).
- These functions returns the number of bytes send or receive.

Socket Functions (cont.)

7. The *send* and *recv* Function

- Here *sockfd socket descriptor;*
- *sendbuf pointer to the buffer where data to* be sent have been stored;
- *recvbuf pointer to the buffer where the data received* is to be stored.
- *nbytes size of data to be sent or received.*
- *This function* returns the number of actual bytes sent or received if successful and –1 if there is an error.



```
int send (int sockfd, const void* sendbuf, int nbytes, int flags);
int recv (int sockfd, void* recvbuf, int nbytes, int flags);
```

Socket Functions (cont.)

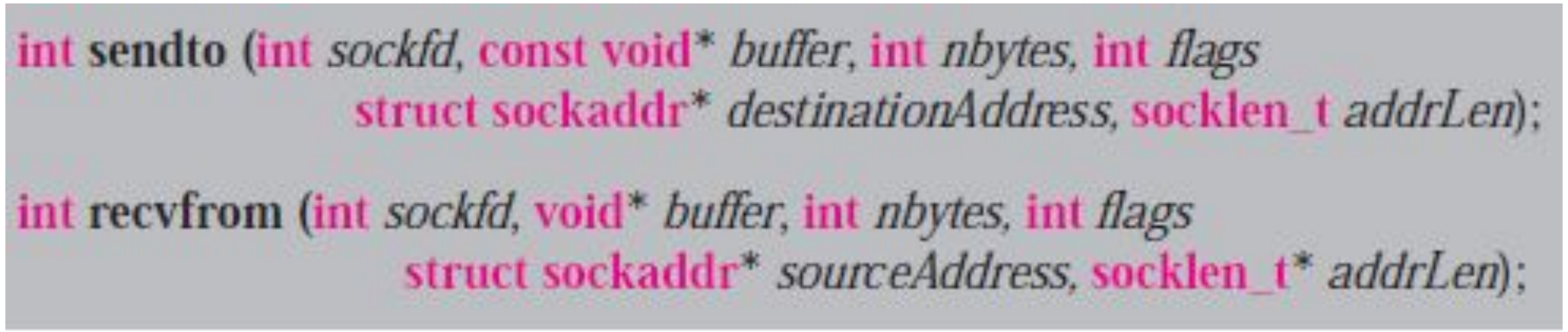
8. The sendto and *recvfrom Function*

- The *sendto function is used by a process to send data to a remote process using the* services of UDP.
- The *recvfrom function is used by a process to receive data from a* remote process using the services of UDP.
- Since UDP is a connectionless protocol, one of the arguments defines the remote socket address (destination or source).

Socket Functions (cont.)

8. The *sendto* and *recvfrom* Function

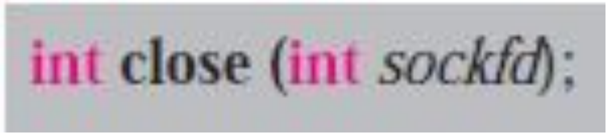
- Here *sockfd socket descriptor,*
- *buffer pointer to the buffer where data to* be sent or to be received is stored,
- *buflen length of the buffer.*
- *T*he value of the flag can be nonzero, (Let it be 0 for our simple programs)
- These functions return the number of bytes sent or received if successful and –1 if there is an error.



```
int sendto (int sockfd, const void* buffer, int nbytes, int flags
                    struct sockaddr* destinationAddress, socklen_t addrLen);

int recvfrom (int sockfd, void* buffer, int nbytes, int flags
                    struct sockaddr* sourceAddress, socklen_t* addrLen);
```

Socket Functions (cont.) *9. The close Function*

- The close function is used by a process to close a socket.
- The *sockfd is not valid after calling this function.*



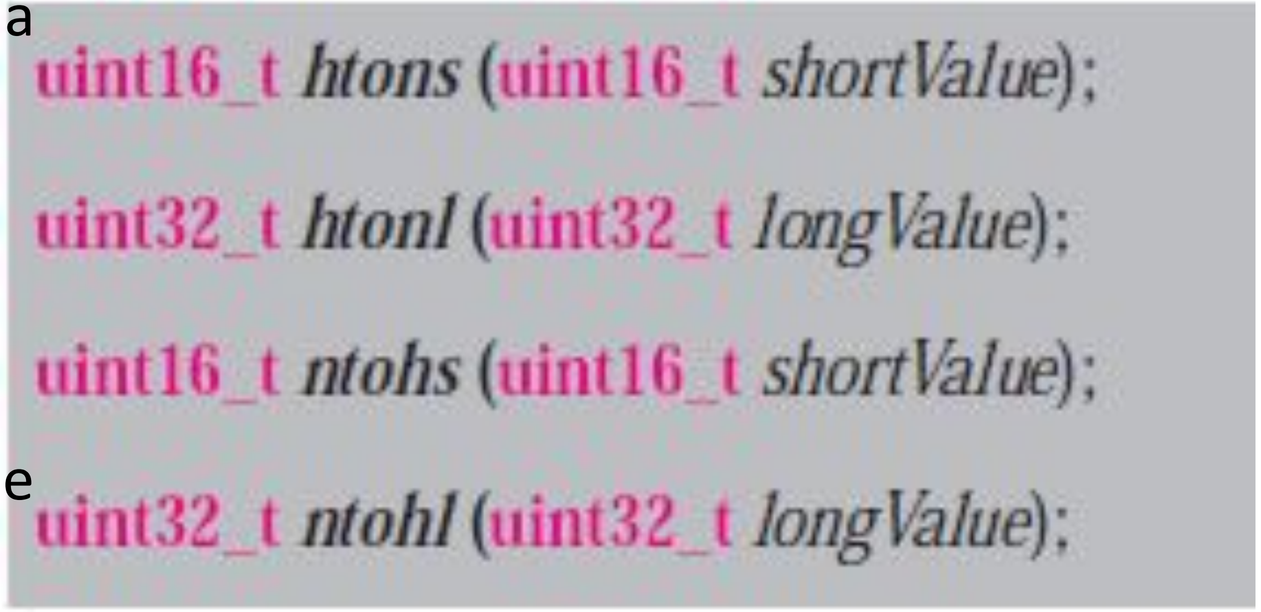
```
int close (int sockfd);
```

- *The socket returns an integer, 0* for success and –1 for error.

Socket Functions (cont.)

10. Byte Ordering Functions

- **htons (host to network short),** which changes a short (16-bit) value to a network byte order,
- **htonl (host to network long),** which does the same for a long (32-bit) value.
- There are also two functions that do exactly the opposite: **ntohs and ntohl.**

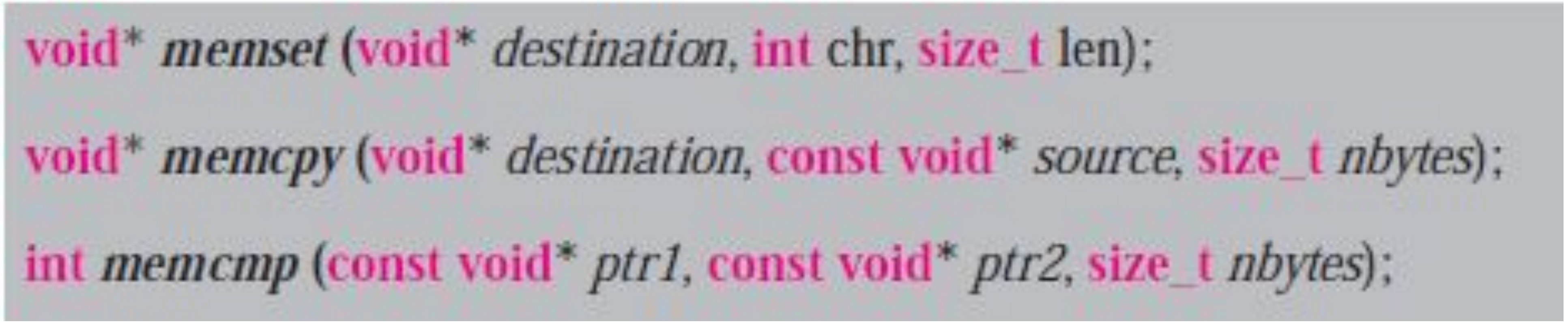


```
uint16_t htons (uint16_t shortValue);
uint32_t htonl (uint32_t longValue);
uint16_t ntohs (uint16_t shortValue);
uint32_t ntohl (uint32_t longValue);
```

Socket Functions (cont.)

11. Memory Management Functions

- To manage values stored in the memory three common memory functions are used
  - *memset* (memory set) is used to set (store) a specified number of bytes (value of len) in the memory defined by the destination pointer (starting address).
  - *memcpy* (memory copy) is used to copy a specified number of bytes (value of nbytes) from part of a memory (source) to another part of memory (destination).
  - *Memcmp* (memory compare), is used to compare two sets of bytes (nbytes) starting from ptr1 and ptr2.

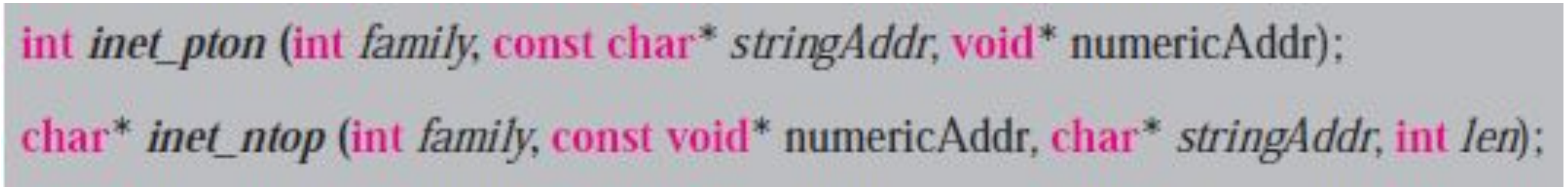


```
void* memset (void* destination, int chr, size_t len);
void* memcpy (void* destination, const void* source, size_t nbytes);
int memcmp (const void* ptr1, const void* ptr2, size_t nbytes);
```

Socket Functions (cont.)

12. Address Conversion Functions

- When we want to store the address in a socket, we need to change it to a number.
- Two functions are used to convert an address from a presentation to a number and vice versa: ***inet_pton*** (presentation to number) ***and inet_ntop*** (number to presentation).



```
int inet_pton (int family, const char* stringAddr, void* numericAddr);
char* inet_ntop (int family, const void* numericAddr, char* stringAddr, int len);
```

Thank
You