Sockets

- A communication structure
- Acts as an end point
- Two processes need a socket at each end to communicate with each other
- A socket is defined in the operating system as a structure

Data Types Defined

u_char	: unsigned 8 bit character
u_short	: unsigned 16 bit integer
u_long	: unsigned 32 bit integer

Socket Address Structures

- Socket function 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 structure begin with sockaddr_ and end with a unique suffix for each protocol suite.

Socket System Calls

- **socket(): Used to create a socket**

```
int socket(int family,    int type,       int protocol)
                 ↓              ↓                 ↓
              AF_INET       SOCK_DGRAM            0
                            SOCK_STREAM
```

Returns socket descriptor on success, -1 on error

Example

```
if ( (sockfd = socket( AF_INET, SOCK_DGRAM, 0 )) < 0 ) {
            cout<<"Socket creation error";
            exit(-1);
}
```

Socket Structure Fields

- Family : Defines protocol group-IPv4, IPv6 etc. **AF_INET**
- Type : Defines type of a socket- stream socket, datagram socket etc. **SOCK_STREAM** , **SOCK_DGRAM**
- Protocol : Set to **0** for TCP and UDP
- Local Socket Address: This field defines the local socket address, a structure of type **sockaddr_in.**
- Remote Socket Address: This field defines the remote socket address, a structure of type **sockaddr_in.**

Internet Address Structure

```
struct in_addr
{
        u_long s_addr;    /*32 bit IPV4 address
};
```

Internet Socket Address Structure

```
struct sockaddr_in
{
        u_short   sin_family;      /* AF_INET*/
        u_short   sin_port;        /*16 bit TCP or UDP port number*/
        struct in_addr   sin_addr; /* 32 bit IPv4 address*/
};
```

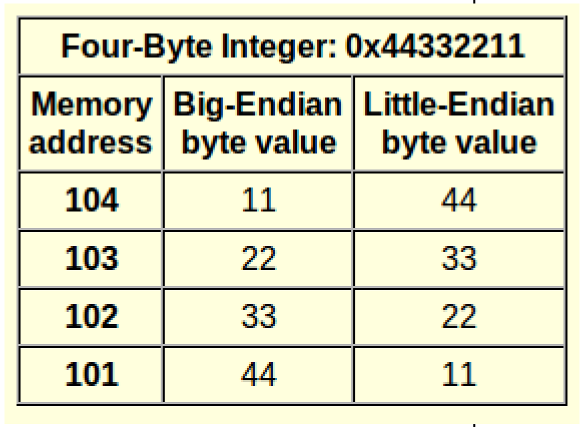
Example: ***struct*** sockaddr_in **serv_addr;**

```
portno = 52632;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

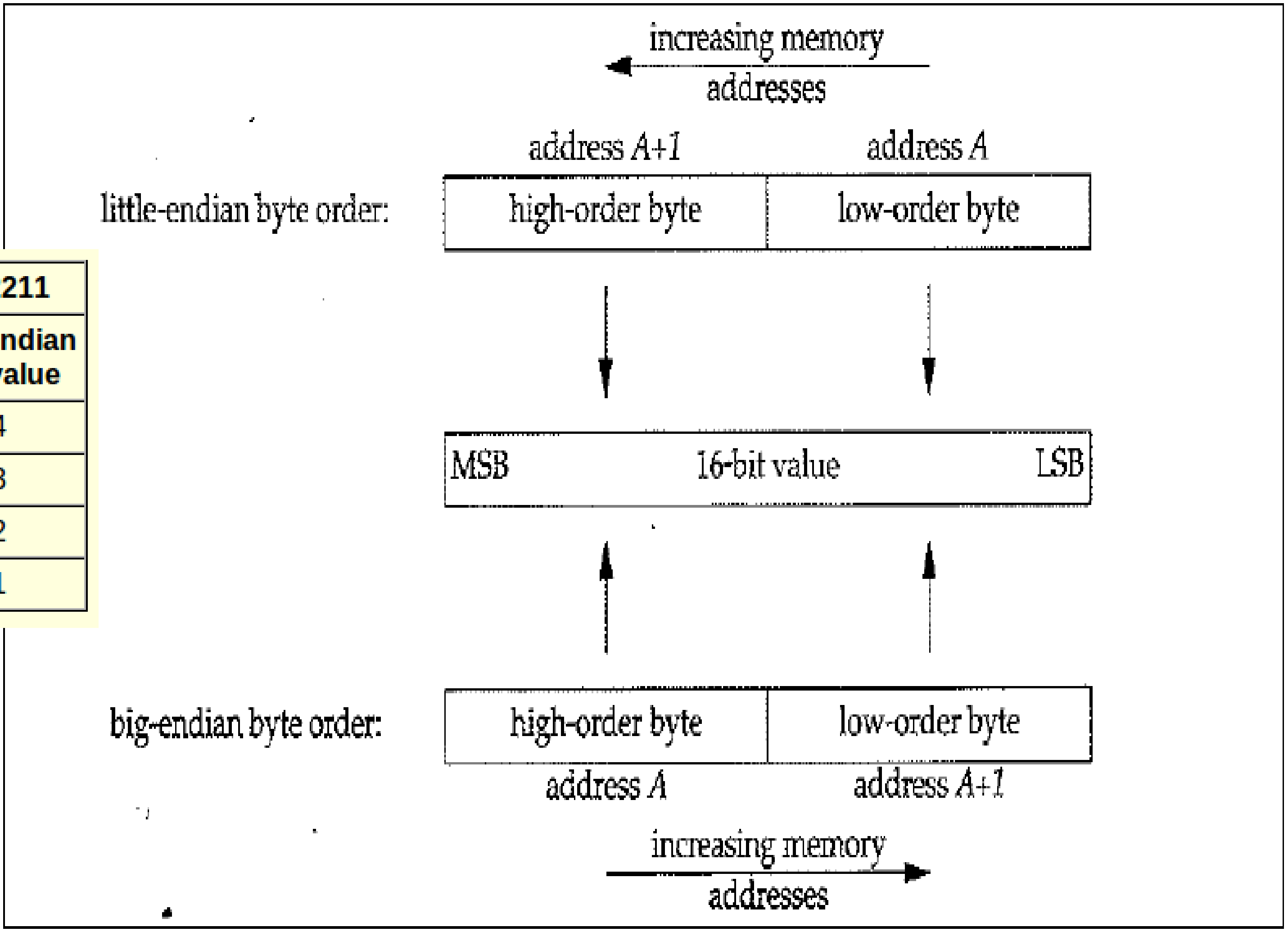
Byte Order

- Network byte order
  - Big Endian
- Host byte order
  - Big Endian *or* Little Endian
- Functions to deal with this
  - htons() & htonl() (host to network short and long)
  - ntohs() & ntohl() (network to host short and long)
- When to worry?
  - putting data onto the wire
  - pulling data off the wire

Byte Ordering



Four-Byte Integer: 0x44332211		
Memory address	**Big-Endian byte value**	**Little-Endian byte value**
104	11	44
103	22	33
102	33	22
101	44	11



Byte Ordering

- Big Endian   : IBM, Motorola
- Little Endian   : Intel based Pcs

Network byte order is Big Endian

Byte order transformations

- u_short htons(u_short host_short)
- u_short ntohs(u_short network_short)
- u_long htonl(u_long host_long)
- u_long ntohl(u_long network_long)

Socket Programming

- **bind(): Adds local socket address to an already created socket**

```
int bind(int sockfd, const struct sockaddr *localaddr, int localaddrlen)
```

Returns 0 on success, -1 on error

“**bind**” System Call

*Why use **bind**?*

- Bind helps the servers to register themselves with the system. It tells the system that “any messages received at the particular IP address and the specified port be directed to the me”. This is required in case of both connection-oriented (like TCP) and connectionless (like UDP) servers.
- 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 is not required in the case of connection-oriented client as we shall see shortly.

“listen” System Call
Definition and Parameters

- This system call is used by a connection-oriented server (i.e., **SOCK_STREAM** socket) to indicate that it is willing to receive connections from clients.
- The function prototype is as follows –

  int listen(int sockfd, int backlog);

- The ***sockfd*** field is nothing but the socket descriptor obtained after executing the socket system call.
- The ***backlog*** parameter defines the maximum length of the queue of pending connections may grow to. This argument is usually specified as 5 which the maximum value currently allowed.

Socket Programming

- **listen(): Informs OS that the server is ready to accept connection**

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

Returns 0 on success, -1 on error

Socket Programming

- **accept(): called by a TCP server to remove the first connection from the corresponding queue**

int **accept**(int sockfd, const struct sockaddr *clientaddr, int * clientaddrelen)

Returns socket descriptor on success,   -1 on error

- **connect(): Used by a process (usually a client) to establish an active connection to a remote process (normally a server)**

```
int connect(int sockfd, const struct sockaddr *serveraddr, int serveraddrlen)
```

Returns 0 on success, -1 on error

Socket Programming

- **send(): used by a process to send data to another process running on a remote machine**

```
int send(int sockfd, const void *buf, size_t len, int flags)
```

Returns number of bytes sent on success, -1 on error

"**send**" System Call
Definition and Parameters

- A connection-oriented client/server can use the send system call to exchange messages between each other. The function prototype is as follows –

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

- ***sockfd*** is the socket descriptor returned by the **socket** system call **in the case of the client** and the **new socket descriptor** returned by the **accept** system call in the case of the server.
- ***msg*** is the message that has to be sent and ***len*** is the length of the message being sent.
- The ***flags*** parameter is usually set to 0.

- **recv(): used by a process to receive data from another process running on a remote machine**

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

Returns number of bytes received on success, -1 on error

"**recv**" System Call

Definition and Parameters

- To receive messages from a connected socket a client/server can use the **recv** system call as follows –

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

- ***sockfd*** and ***flags*** have the same meaning as in the **send** system call.
- Here ***buf*** is nothing but the message that will be received by the client/server.
- The ***len*** parameter specifies the number of characters (in bytes) that can be read at once.
- The **recv** system call blocks until a message is received.

Socket Programming

- **close(): used by a process to close a socket and terminate a TCP connection**

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

Returns 0 on success,   -1 on error

"**close**" System Call
Definition and Parameters

- The Unix **close** system call is also used to close a socket.
- close() is used to completely terminate the socket connection and release all associated resources.
- The function prototype is as follows –

  int close(int sockfd);
- Here ***fd*** refers to the socket descriptor.
- The **close** system call does not close the socket at once. Before closing it tries to send any queued data or any queued data to be sent is flushed.
- **The close** system call returns 0 on success and -1 on error.

shutdown()

Communication on a socket is bidirectional. We can disable I/O on a socket with the shutdown function.

shutdown() is used to selectively stop data flow in one direction of the socket connection

#include <sys/socket.h>

int shutdown (int sockfd, int how);

Returns: *0 if OK, 1 on error*

sockfd **–** is the socket descriptor returned by *socket( )* command.

how- is **SHUT_RD**, then **reading** from the socket is **disabled**. If how is **SHUT_WR**, then we **can't** use the socket for **transmitting data**. We can use **SHUT_RDWR** to **disable both data transmission and reception.**

"sendto" System Call
Definition and Parameters

- A connectionless client/server can use sendto to exchange messages between each other.
- Note that **send** may be used only when the socket is in a connected state. But, **sendto** can be used at any time.
- The function prototype is as follows –

  int sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
- Here ***sockfd***, ***buf***, ***len*** and ***flags*** have the same meaning as described before in the send system call.
- The ***to*** argument for sendto points to the protocol-specific address structure containing the address where the data is to be sent.

“sendto” System Call

Some more information…

- ***tolen*** is the length of the protocol-specific address structure pointed to by ***to***.
- Both send and sendto system calls return the number of characters (in bytes) sent if the call succeeds or else -1 is returned if an error occurred.

"recvfrom" System Call

Definition and Parameters

- A connectionless client/server can use the recvfrom system call to receive messages as follows –

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

- Here ***sockfd***, ***buf***, ***len*** and ***flags*** have the same meaning as in the recv system call.
- ***from*** points to the protocol-specific address structure which will be filled in with the source address (i.e., address of the host that sent the message).

"recvfrom" System Call
Parameters and more…

- Initially ***fromlen*** is initialized to the length of the address structure pointed to by ***from***, on return, ***fromlen*** is modified to indicate the actual size of the address stored.
- Like **recv**, **recvfrom** blocks until it receives a message.
- Both **recv** and **recvfrom** system calls return the number of bytes received, or -1 is returned if an error occurred while receiving an message.

A Simple TCP client program

```
#include<string.h>
#include<arpa/inet.h>
#include<iostream.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<fcntl.h>
#include<sys/stat.h>

#define MAXSIZE 50
```

Socket Programming

```
main() {
        int sockfd,retval;
        int recedbytes,sentbytes;
        struct sockaddr_in serveraddr;
        char buff[MAXSIZE];
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        if(sockfd==-1) {
                cout<<"\nSocket creation error";
        exit(0);
        }
```

Socket Programming

```
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(3387);
serveraddr.sin_addr.s_addr=inet_addr("127.0.0.1");
\*The inet_addr() function converts the Internet host address cp from numbers-and-dots notation
into binary data in network byte order. */
retval=connect(sockfd,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
if(retval==-1) {
            cout<<"connection error";
            exit(-1);
 }
```

Socket Programming

```
gets(buff);
sentbytes=send(sockfd,buff,sizeof(buff),0);
if(sentbytes==-1) {
        close(sockfd);
        exit(0); }
recedbytes=recv(sockfd,buff,sizeof(buff),0);
puts(buff);
cout<<"\n";
close(sockfd); }
```

A simple TCP server program

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#define MAXSIZE 50
```

Socket Programming

```
main()
{
        int sockfd,newsockfd,retval;
        socklen_t actuallen;
        int recedbytes,sentbytes;
        struct sockaddr_in serveraddr,clientaddr;
        char buff[MAXSIZE];
        int a=0;
```

Socket Programming

```
sockfd=socket(AF_INET,SOCK_STREAM,0);
 if(sockfd==-1) {
            cout<<"\nSocket creation error";
            exit(-1); }
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(3387);
serveraddr.sin_addr.s_addr=htonl(INADDR_ANY);
```

Socket Programming

```
retval=bind(sockfd,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
if(retval==-1) {
            cout<<"Binding error"; close(sockfd);
            exit(0); }
retval=listen(sockfd,1);
if(retval==-1) { close(sockfd);
exit(0); }
actuallen=sizeof(clientaddr);
```

Socket Programming

```
newsockfd=accept(sockfd,(struct sockaddr*)&clientaddr,&actuallen);
if(newsockfd==-1) {
            close(sockfd);
            exit(0); }
 recedbytes=recv(newsockfd,buff,sizeof(buff),0);
 if(recedbytes==-1) {
            close(sockfd);
            close(newsockfd);
            exit(0); }
 puts(buff);
 cout<<"\n";
```

Socket Programming

```
gets(buff);
sentbytes=send(newsockfd,buff,sizeof(buff),0);
 if(sentbytes==-1) {
            close(sockfd);
            close(newsockfd);
            exit(0);
 }
close(newsockfd);
close(sockfd);
}
```