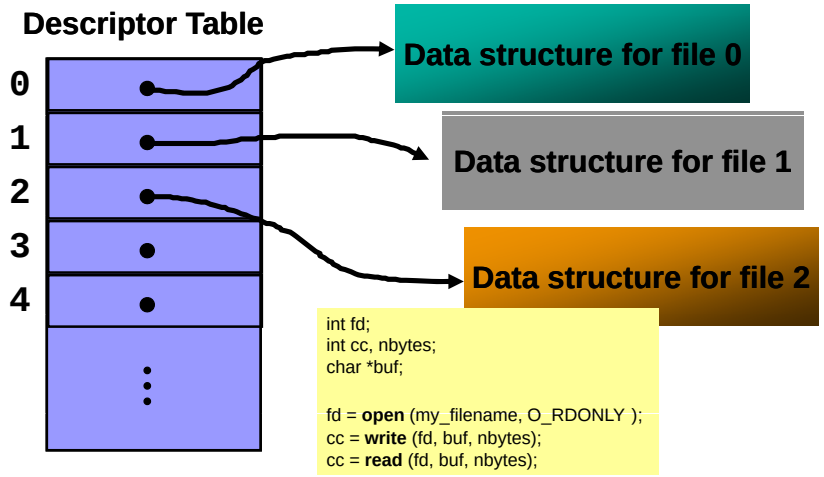
Network Programming

Dr. Thaier Hayajneh

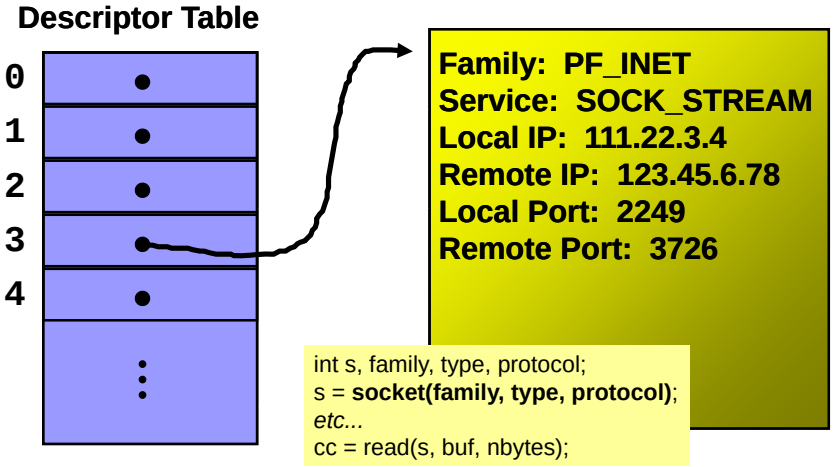
Computer Engineering Department

Sockets 3

Unix Descriptor Table

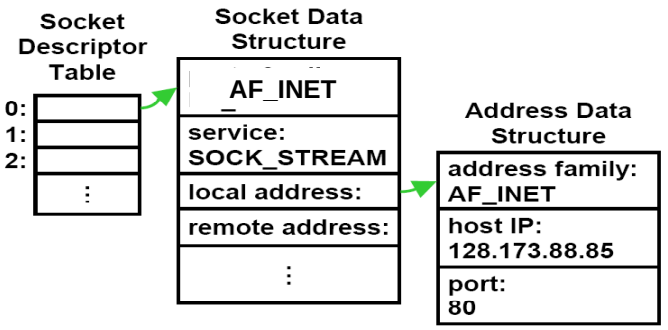


Socket Descriptor Data Structure



Socket Descriptors

- Operating system maintains a set of socket descriptors for each process
- Three data structures
  - ➢Socket descriptor table → Socket data structure →Address data structure



Client-Server Model

- *Server*
  - Create a socket with the socket() system call
  - Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
  - Listen for connections with the listen() system call
  - Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
  - Send and receive data

Client-Server Model

- *Client*
  - Create a socket with the socket() system call
  - Connect the socket to the address of the server using the connect() system call
  - Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

`socket()`

- The **`socket()`** system call returns a socket descriptor (small integer) or -1 on error.
- **`socket()`** allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.

Creating a Socket

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

- `family` specifies the protocol family
  - AF_INET: IPv4 protocols
  - AF_INET6: IPv6 protocols
  - AF_ROUTE: Routing sockets
- `type` specifies the type of service
  - **`SOCK_STREAM`**
  - **`SOCK_DGRAM`**
  - **`SOCK_RAW`**
- `protocol` specifies the specific protocol (usually 0, which means *the default*).
  - IPPROTO_TCP: TCP transport protocol
  - IPPROTO_UDP: UDP transport protocol

Specifying an Endpoint Address

- Remember that the sockets API is *generic*
- There must be a *generic* way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.

`bind()`

- calling **`bind()`** assigns the address specified by the **`sockaddr`** structure to the socket descriptor.
- It binds a socket to a local socket address by adding the local socket address to an already created socket

```
bind( mysock,
      (struct sockaddr*) &myaddr,
      sizeof(myaddr) );
```

`connect()`

- **`connect()`** is used by a process (usually a client) to establish an active connection to a remote process (normally server)
- **`Client have to call the socket function first`**

int connect (int sockfd , const struct sockaddress *serveraddr , socklen_t serveraddrlen) ;

Returns 0 if successful; -1 if error.

`listen()`

- **`listen()`** is called by the TCP server. It creates a passive socket from an unconnected socket.
- It informs the OS that the server is ready to accept connection through this socket
  - sockfd: is the socket descriptor
  - backlog: is the number of requests that can be queued for this connection

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

Returns 0 if successful; -1 if error.

accept()

- **accept()** is called by the TCP server to remove the first connection request from the corresponding queue.
- If there are no requests it is put to sleep
  - clientaddr is the pointer to the address of the client that has requested the connection
  - clinetaddrlen is a pointer to the client address length

```
int accept (int sockfd , struct sockaddress *clientaddr , socklen_t *clientaddrlen) ;
```

Returns a socket descriptor if successful; -1 if error.

sendto()

- **sendto()** is used by process using UDP to send a message to another process running on a remote machine.
  - sockfd: is the socket descriptor
  - buf: is a pointer to the buffer holding the message to be sent
  - buflen: defines the length of the buffer
  - Flags: specifies out-of-band data or lookahead message (normally set to zero)
  - toaddr: is a pointer to the socket address of the receiver

```
ssize_t sendto (int sockfd , const void *buf , size_t buflen , int flags ,
    const struct sockaddress *toaddr , socklen_t toaddrlen) ;
```

Returns number of bytes sent if successful; -1 if error.

recvfrom()

- **recvfrom()** extracts the next message that arrives at a socket. It also extracts the sender's socket address.
- It is mostly used by UDP process
  - sockfd: is the socket descriptor
  - buf: is a pointer to the buffer where the message will be stored
  - buflen: defines the length of the buffer
  - Flags: specifies out-of-band data or lookahead message (normally set to zero)
  - fromaddr: is a pointer to the socket address of the sender

```
ssize_t recvfrom (int sockfd , void *buf , size_t buflen , int flags,
    struct sockaddress *fromaddr , socklen_t *fromaddrlen) ;
```

Returns number of bytes received if successful; -1 if error.

read()

- **read()** is used by a process to receive data from another process running on a remote machine.
- This function assumes that there is already an open connection between two machines → TCP
  - sockfd: is the socket descriptor
  - buf: is a pointer to the buffer where data will be stored
  - buflen: defines the length of the buffer

```
ssize_t read (int sockfd , void *buf , size_t buflen ) ;
```

Returns number of bytes read if successful; 0 for end of file; -1 if error.

write()

- **write()** is used by a process to send data from another process running on a remote machine.
- This function assumes that there is already an open connection between two machines → TCP
  - sockfd: is the socket descriptor
  - buf: is a pointer to the buffer where data to be sent is stored
  - buflen: defines the length of the buffer

ssize_t **write** (int sockfd , const void *buf , size_t buflen) ;

Returns number of bytes written if successful; -1 if error.

close()

- **close()** is used by a process to close a socket and terminate a TCP connection
- The socket descriptor is not valid after calling this function

int **close** (int sockfd) ;

Returns 0 if successful; -1 if error.

TCP/IP Addresses

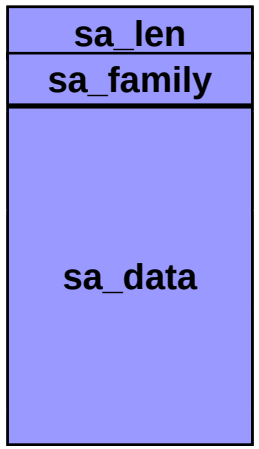
- We don't need to deal with **sockaddr** structures since we will only deal with a real protocol family.
- We can use **sockaddr_in** structures.

BUT: The C functions that make up the sockets API expect structures of type **sockaddr.**

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

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

sockaddr



sockaddr_in

Assigning an address to a socket

- The **`bind()`** system call is used to assign an address to an existing socket.

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

- **`bind`** returns 0 if successful or -1 on error.

`bind()` Example

```
int mysock,err;
struct sockaddr_in myaddr;

  mysock = socket(PF_INET,SOCK_STREAM,0);
  myaddr.sin_family = AF_INET;
  myaddr.sin_port = htons( portnum );
  myaddr.sin_addr = htonl( ipaddress);

  err=bind(mysock, (sockaddr *) &myaddr,
              sizeof(myaddr));
```

Uses for **`bind()`**

- There are a number of uses for **`bind()`**:
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the OS to assign *any available* port number.

Port schmo - who cares ?

- Clients typically don't care what port they are assigned.
- When you call bind you can tell it to assign you any available port:

Why htons? 0 is 1 byte

```
myaddr.port = htons(0);
```

  - ***1-1024: reserved port (assigned by privileged processes)***

What is my IP address ?

- How can you find out what your IP address is so you can tell **`bind()`** ?
- There is no realistic way for you to know the right IP address to give bind() - what if the computer has multiple network interfaces?
- specify the IP address as: **`INADDR_ANY`**, this tells the OS to take care of things.

1 byte, Why htonl?

```
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

IPv4 Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

Other socket system calls

- General Use
  - **`read()`**
  - **`write()`**
  - **`close()`**
- Connection-oriented (TCP)
  - **`connect()`**
  - **`listen()`**
  - **`accept()`**
- Connectionless (UDP)
  - **`send()`**
  - **`recv()`**

Value-Result Arguments (1)

•Length of socket passed as an argument
•Method by which length is passed depends on which direction the structure is being passed (from process to kernel, or vice versa)

•Value-only: *bind*, *connect*, *sendto* (from process to kernel)

```
struct sockaddr_in     serv;
connect (sockfd, (struct sockaddr *)  &serv,  sizeof (serv));
```

Here the Kernel is passed both the pointer and the size of what the pointer points, knows exactly how much data to copy from the process into the kernel

Value-Result Arguments (2)

•Value-Result: *accept*, *recvfrom*, *getsockname*, *getpeername* (from kernel to process, pass a pointer to an integer containing size)

- *Tells process how much information kernel actually stored*

```
struct sockaddr_in     clientaddr ;
socklen_t    len;
int      listenfd, connectfd;

len = sizeof (clientaddr);
connectfd = accept (listenfd,   (SA *) &clientaddr,   &len);
```