CSE 156/L

Socket Overview

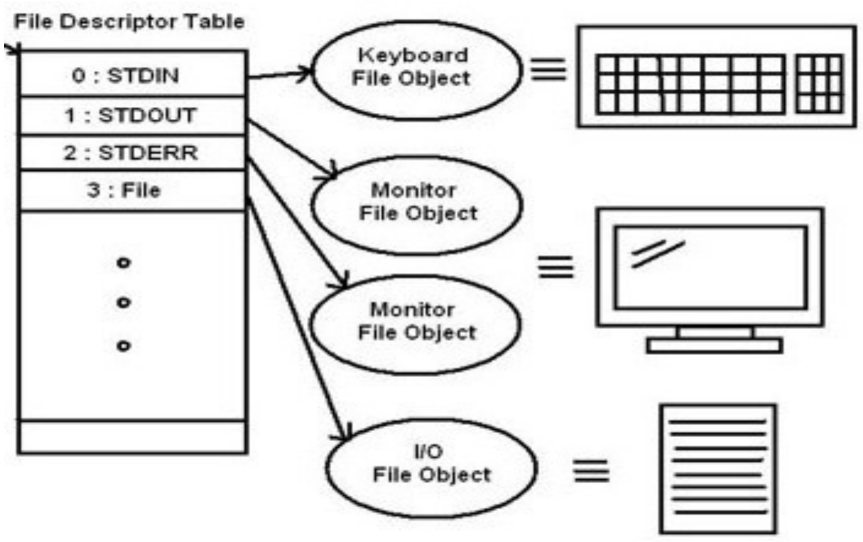
Motivation

- Developing network applications requires an Application Program Interface (API)
- TCP/IP protocol specifications do not include an API definition
- Many APIs defined by OS platforms
  - Sockets (BSD Unix), standardized by IEEE (POSIX)
  - TLI/XTI (AT&T System V)
  - Winsock (Microsoft)
  - MacTCP (Apple)

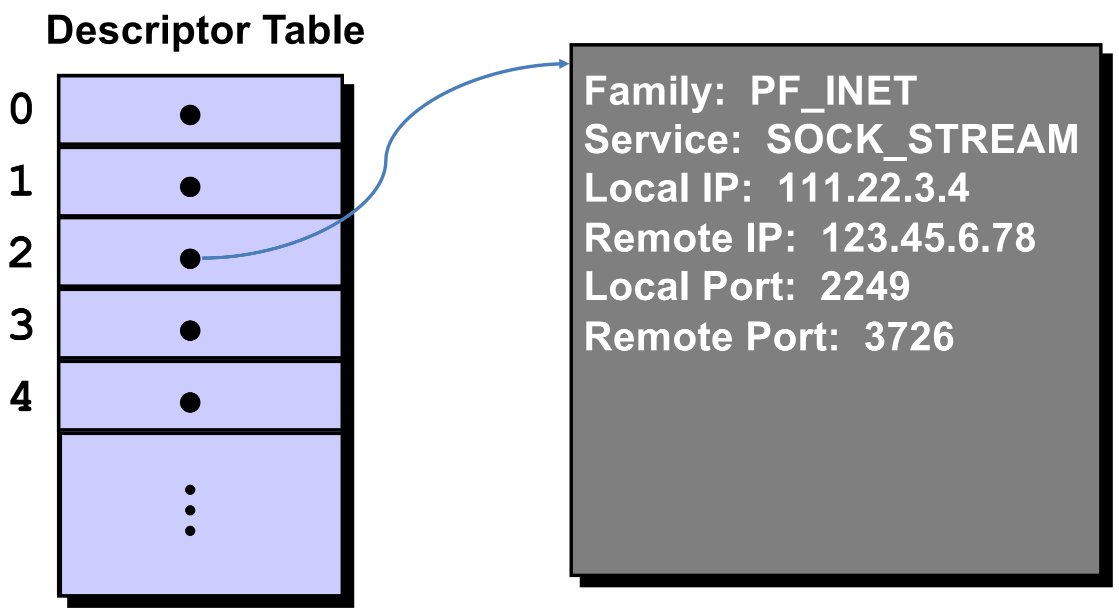
Typical Functions Needed

- Specify the endpoints of a network connection (local and remote)
- Open a connection (active open)
- Wait for an incoming connection (passive open or listen)
- Terminate a connection
- Abort a connection because of errors

Unix File Descriptor Table



Socket Descriptor Data Structure



POSIX Data Types

`int8_t`	signed 8-bit integer
`uint8_t`	unsigned 8-bit integer
`int16_t`	signed 16-bit integer
`uint16_t`	unsigned 16-bit integer
`int32_t`	signed 32-bit integer
`uint32_t`	unsigned 32-bit integer

Regular C data types: `u_char, u_short, u_int, u_long`

Creating a Socket

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

- *family*: specifies the protocol family
  - AF_INET for TCP/IPv4, AF_INET6 for TCP/IPv6
  - PF_INET: practically socket.h defines PF_* equal to AF_*
- *type*: type of service
  - E.g., SOCK_STREAM or SOCK_DGRAM
- *proto*: protocol to use
  - Usually is 0 = default for given family/type
- Returns integer socket descriptor (-1 on error)
- Allocates socket data structure but does **not** define endpoint addresses

Address Structure

POSIX data types for endpoint address:

`sa_family_t`	address family
`socklen_t`	size of structure
`in_addr_t`	IPv4 address
`in_port_t`	transport port number

TCP/IPv4 Socket Address Structure

POSIX data types for endpoint address:

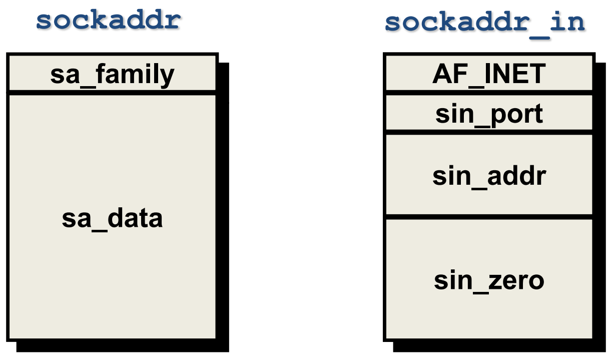
```
struct sockaddr_in {
    uint8_t           sin_len;
    sa_family_t       sin_family;    // uint8_t
    in_port_t         sin_port;      // uint16_t
    struct in_addr    sin_addr;
    char              sin_zero[8];
 };
```

```
// typedef uint32_t        in_addr_t;
struct in_addr {
       in_addr_t           s_addr;
};
```

TCP/IP Socket Address Structure

- Generic socket structure (sockaddr) used in socket API prototypes (16 bytes)
- Cast protocol structures (e.g., sockaddr_in for IPv4) to this generic type

```
struct sockaddr {
    unsigned short   sa_family;     // address family, AF_xxx
    char             sa_data[14];   // 14 bytes of protocol address
 };
```



Network Byte Order

- Network byte order for Internet protocols is big-endian
- IP address and TCP port number stored in `sockaddr` structure must be in network byte order
- Host byte order may be big- or little-endian
- Use library functions to convert between network and host byte order

Byte Order Conversion

'**h**' : host byte order

'**n**' : network byte order

'**s**' : short (16bit)

'**l**' : long (32bit)

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

Specifying Host Address

- The **bind()** system call is used to assign an address to an existing socket
  - I.e., IP address and port number of the local endpoint
- Typically used by servers to assign listening address and port.
- The servers have *well-known* port number assigned to the application
- Can also be used by clients to bind to a specific port
- `addrlen` is the sizeof `addr` specific structure

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

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

Specifying Host Address

- **bind()** returns 0 if successful or -1 on error
- Calling bind() with a port number of 0 results in the OS assigning an available port number
- How to determine IP address?
  - No general way to determine the IP address to bind to, when the host has multiple network interfaces
  - Common practice is to specify IP address as INADDR_ANY to let OS assign the IP address

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

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

bind() Example

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

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

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

TCP Client

Initiating a TCP Connection

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

- Clients initiate connection using **`connect()`** system call
  - Also sets up an endpoint address (IP address, port number) for the client socket
  - Clients don't need to call bind first
- Results in the OS performing TCP connection setup to remote host via 3-way handshake
  - *`sockfd`* is socket returned by **`socket()`** call
  - *`server`* contains the IP address and TCP port number of the server
  - Returns 0 normally, -1 on error

Closing a Connection

- Either side can initiate closing the connection with the **`close()`** system call
- Must be called by both sides to terminate cleanly
- If the other end has closed the connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF

Receiving Data from a Socket

```
int read(int fd, char *buf, int max);
```

`max` is the maximum number of bytes the read is willing to accept (size of buffer)

- By default, **`read()`** will block until data is available
  - Non-blocking option available
- Returns number of bytes read into the buffer
  - Between 1 and `max` bytes when connection is open
  - 0 indicates EOF (i.e., other side closed connection)

Sending Data to a Socket

```
int write(int fd, char *buf, int num);
```

`num` is the number of bytes to be written

- Writes can be made non-blocking
  - Might not be able to write all `num` bytes
  - Returns actual number of bytes written (between 0 and num)

TCP Server

Configuring a Passive-Mode Socket

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

- Server needs to initialize socket to receive incoming TCP connections
  - Also called “listening” socket
- Performed by **`listen()`**
  - *`sockfd`* is the TCP socket (already bound to an address)
  - *`backlog`* is the number of incoming connections the kernel should be able to queue for the application
  - Returns 0 normally, -1 on error

Accepting a Connection

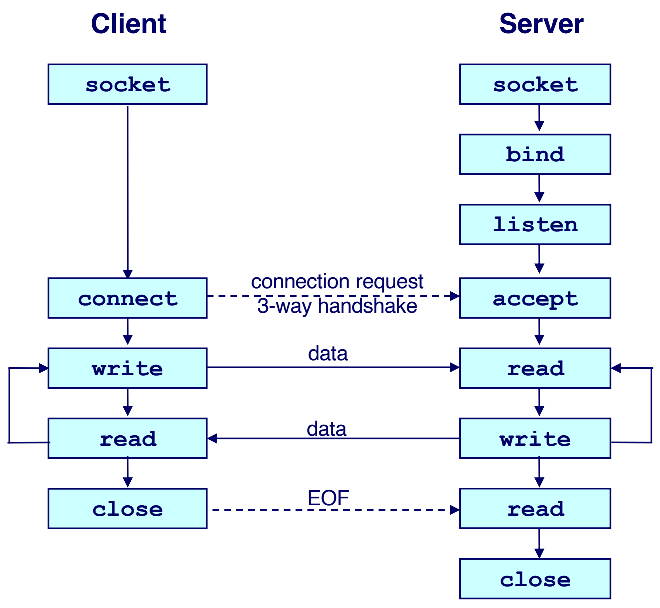
```
int accept(int sockfd,
           struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

- **`listen()`** must be followed by **`accept()`** to obtain parameters of accepted connection
  - *sockfd* is the passive mode TCP socket returned by the socket() call
  - *cliaddr* is a pointer to allocated space for the socket address structure
  - *addrlen* is a *value-result* argument
    - must be set to the size of `cliaddr` when calling
    - on return, will be set to be the number of used bytes in `cliaddr`

Accepting a Connection (continued)

- **`accept()`** returns new a socket descriptor or -1 on error
  - Different from the original listening socket descriptor
- Must use the new descriptor to read and write data

Example Sequence of Socket Calls



TCP Client/Server

UDP Client/Server

Sending Data to a Socket

```
ssize_t sendto(int sockfd,
    const void *buff, size_t nbytes, int flags,
    const struct sockaddr *to, socklen_t addrlen);
```

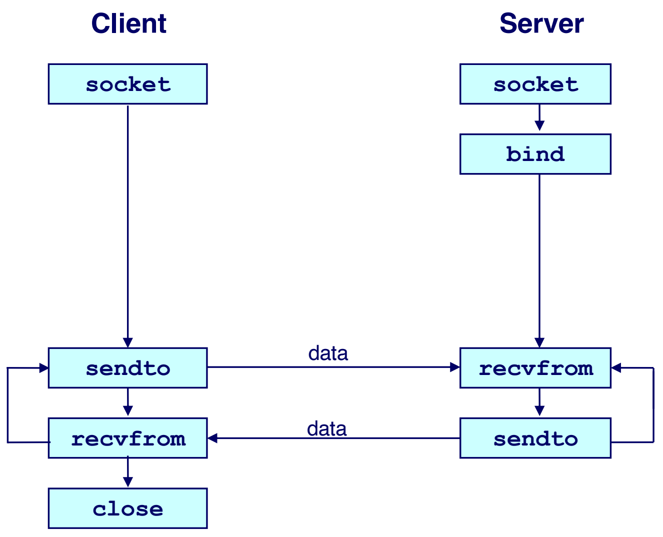
- The first three arguments, *sockfd*, *buff*, and *nbytes*, are identical to the first three arguments for write()
- *to* is a socket address structure containing the protocol address of where the data is to be sent
- Writes are non-blocking by default
  - Might not be able to write all `nbytes` bytes
  - Returns actual number of bytes written (between 0 and *nbytes*)

Receiving Data from a Socket

```
ssize_t recvfrom(int sockfd,
    void *buff, size_t nbytes, int flags,
    struct sockaddr *from, socklen_t *addrlen);
```

- The first three arguments, *sockfd*, *buff*, and *nbytes*, are identical to the first three arguments for read()
- The socket address structure of *from* is filled with the address of who sent the datagram, and the size this address structure is also returned in *addrlen*
- By default, **`recvfrom()`** will block until data is available
  - Non-blocking option available
- Returns number of bytes read into the buffer
  - Between 1 and `nbytes` bytes when connection is open
  - 0 indicates EOF (i.e., other side closed connection) with TCP

Example Sequence of Socket Calls



UDP Client/Server