

# NETWORK SOCKETS

**OSI, TCP/UDP, IP**

**Network Programming: socket, bind, listen, accept/connect, close**

**Addressing: sockaddr\_in, sockaddr structs & functions**

**Send/Recv: write/read, send/recv, sendto/recvfrom, sendmsg/recvmmsg**

**Client-Server: Sequential, Concurrent, Asynchronous/Multiplexed I/O,  
Nonblocking, Signal-based Asynchronous I/O, ... Socket**

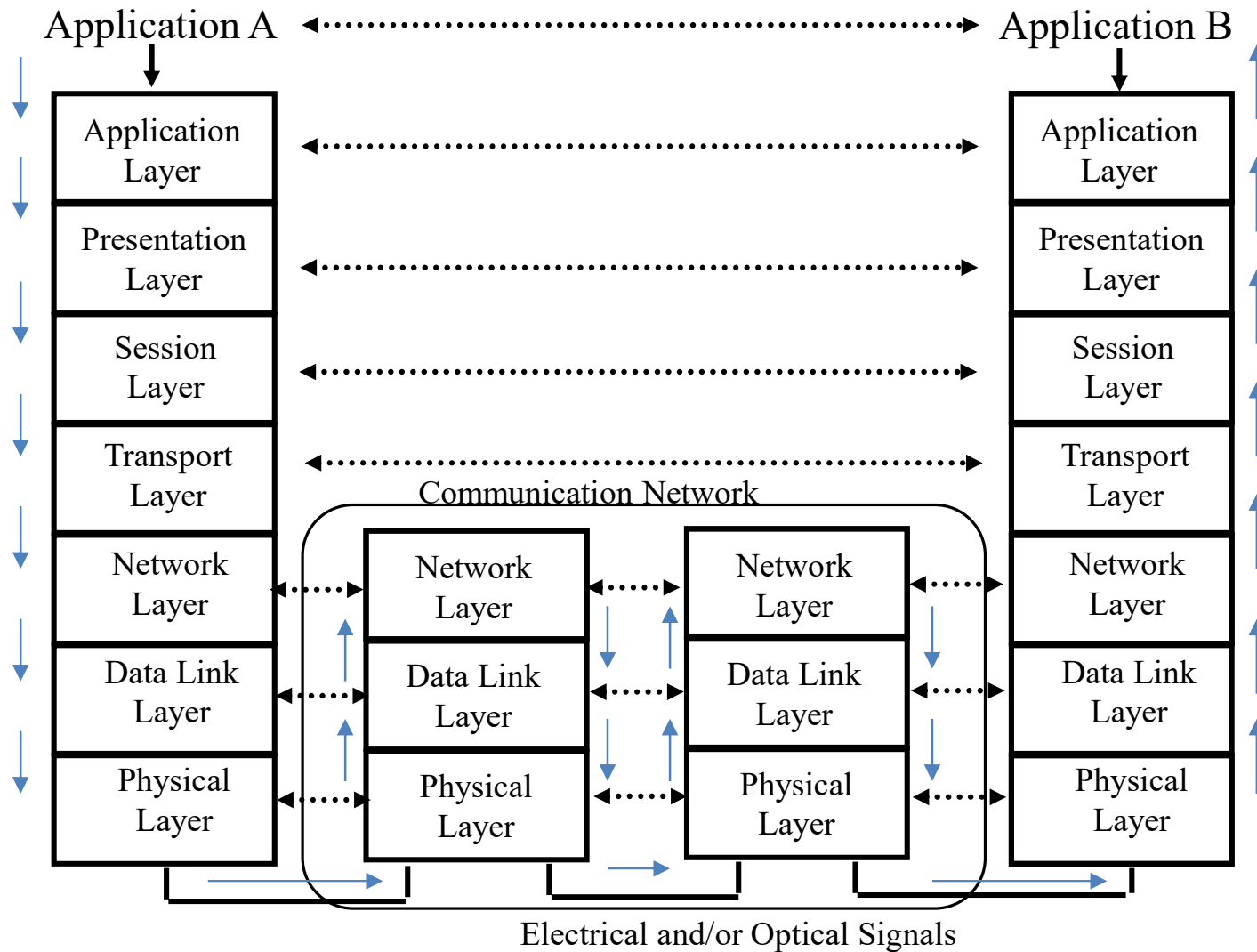
**Options, Flags**

# OSI

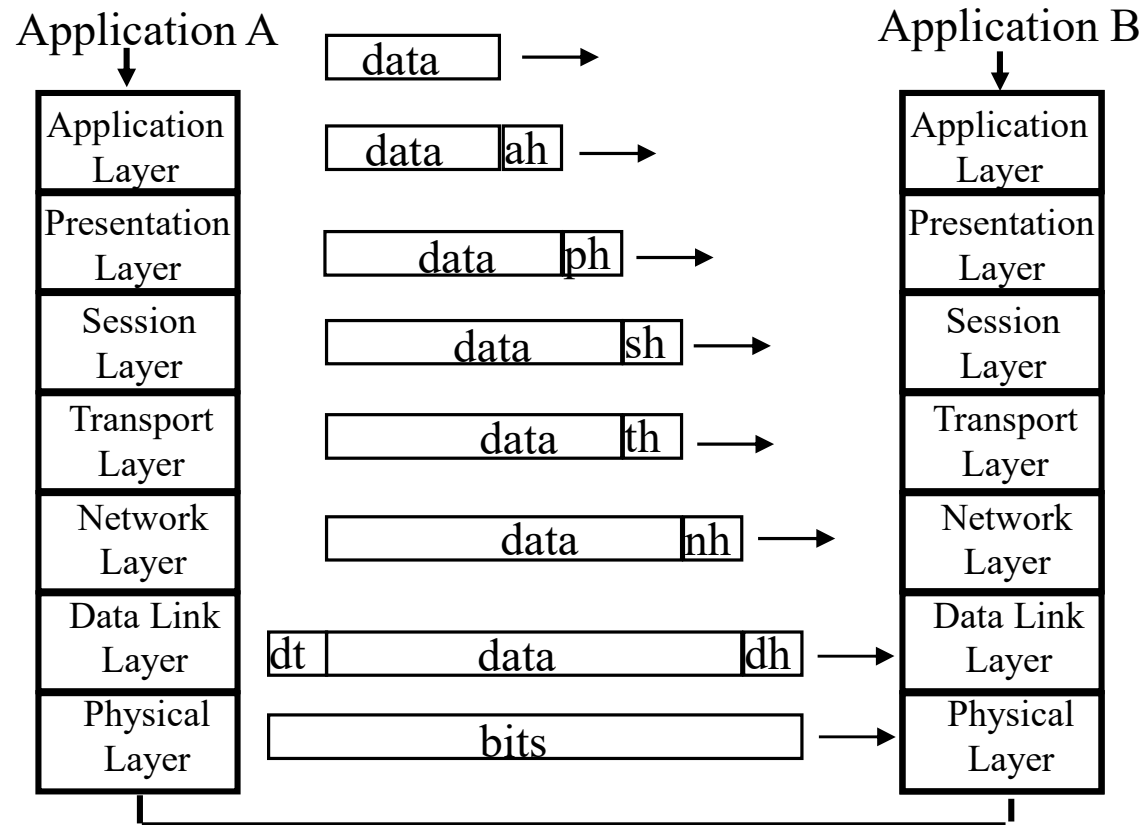
## OSI model

| Layer | Name               | Example protocols                    |
|-------|--------------------|--------------------------------------|
| 7     | Application Layer  | HTTP, FTP, DNS, SNMP, Telnet         |
| 6     | Presentation Layer | SSL, TLS                             |
| 5     | Session Layer      | NetBIOS, PPTP                        |
| 4     | Transport Layer    | TCP, UDP                             |
| 3     | Network Layer      | IP, ARP, ICMP, IPSec                 |
| 2     | Data Link Layer    | PPP, ATM, Ethernet                   |
| 1     | Physical Layer     | Ethernet, USB, Bluetooth, IEEE802.11 |

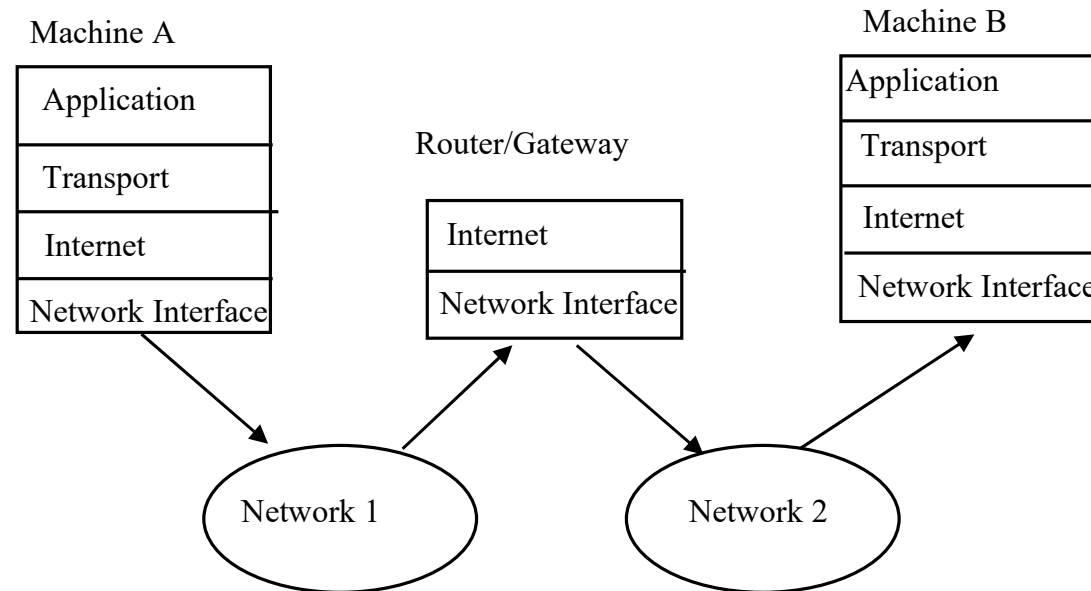
## The OSI reference model (proposed by ISO)



# HEADER ENCAPSULATION



# TCP/IP ARCHITECTURE - INTERNET LAYER



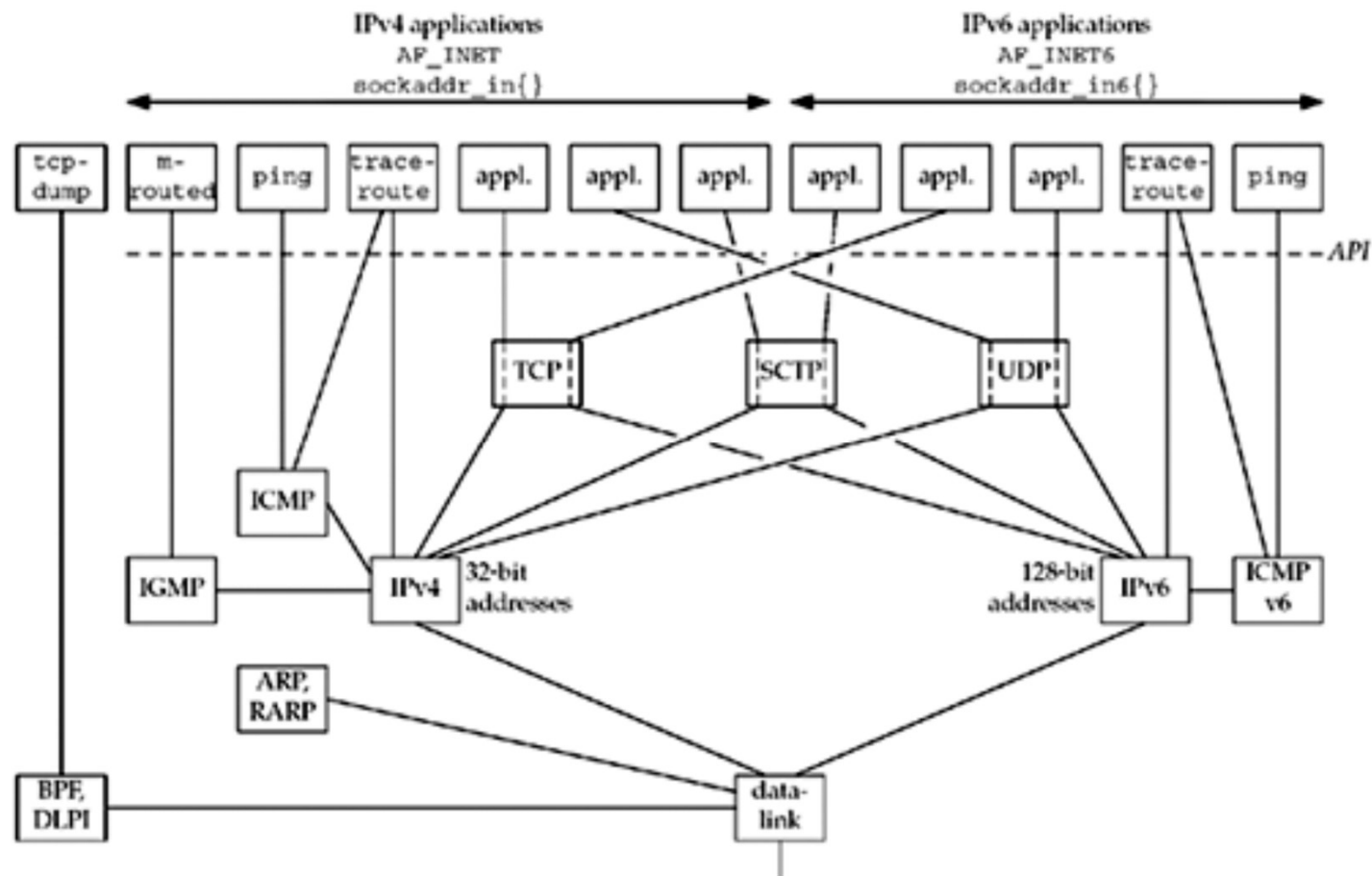
Internet Layer maps to OSI stack: routing & congestion control

- Transfer data across networks via gateways/routers (unique IP address& packets)
- Best-effort connectionless IP packet transfer

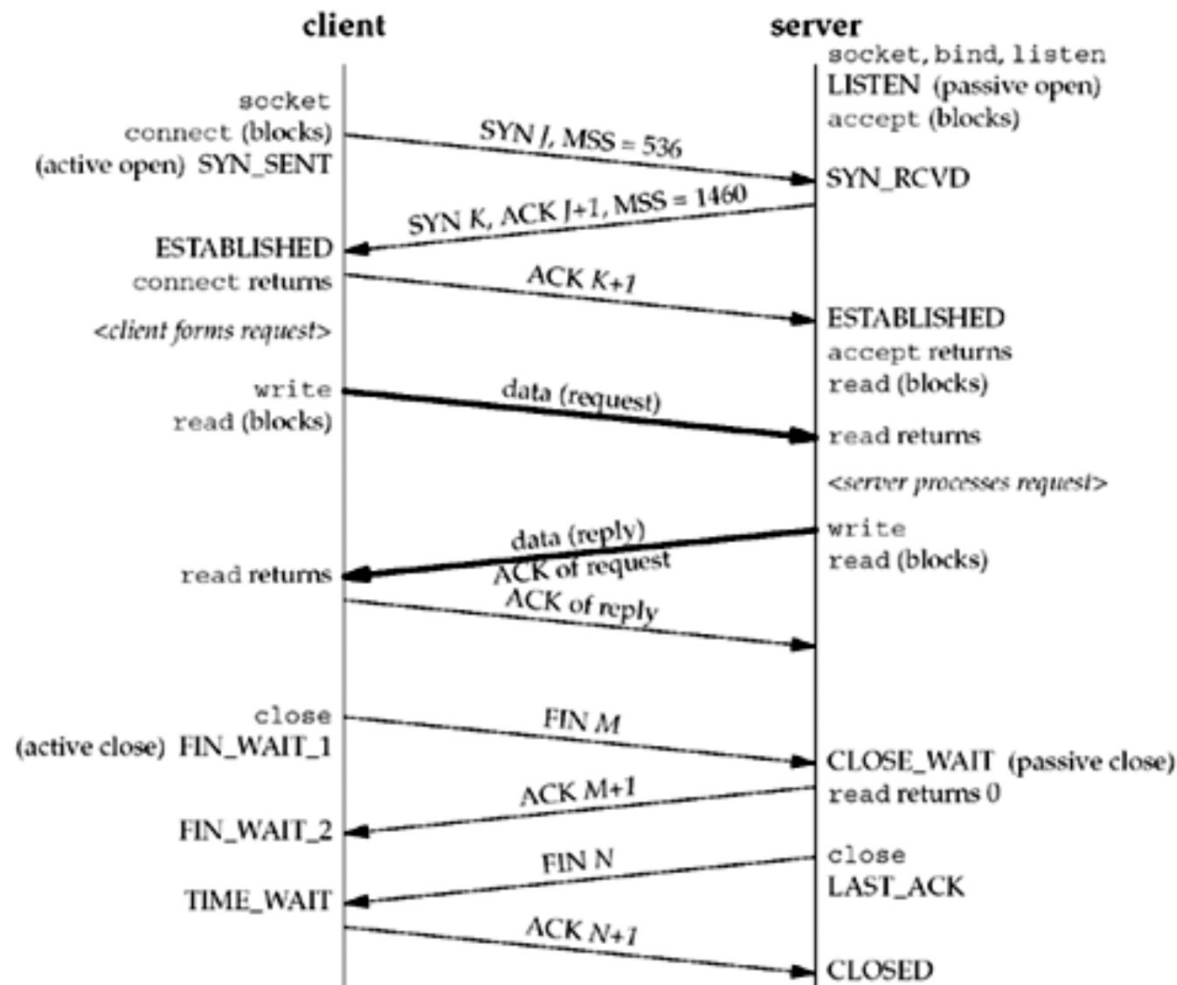
Network Layer maps to OSI network layer and data link layer

- Concerned with network-specific aspects of the transfer of packets
- Different network interfaces: X.25, ATM, frame relay, Ethernet, etc

# THE BIG PICTURE



# PACKET EXCHANGE FOR TCP CONNECTION



# USEFUL NETWORK COMMANDS

- `ifconfig`, `ip`
- `host(name)`, `/etc/hosts`
- `dig/drill` for `dns`
- `ping`, `hping`
- `netstat -tulpn` or `-peanut`, `ss`
- `nmap`
- `netlink`
- `netlookup`
- `traceroute`, `mtr -rw`
- `netcat (nc)`
- `tcpdump`, `wireshark`
- `iptables`, `ufw`
- `lsof ...`



# TCP/UDP PORT NUMBERS

[https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

- 20 [File Transfer Protocol](#) (FTP) Data Transfer
- 21 [File Transfer Protocol](#) (FTP) Command Control
- 22 [Secure Shell](#) (SSH) Secure Login
- 23 [Telnet](#) remote login service, unencrypted text messages
- 25 [Simple Mail Transfer Protocol](#) (SMTP) E-mail routing
- 53 [Domain Name System](#) (DNS) service
- 67, 68 [Dynamic Host Configuration Protocol](#) (DHCP)
- 80 [Hypertext Transfer Protocol](#) (HTTP) used in WWW
- 110 [Post Office Protocol](#) (POP3)
- 119 [Network News Transfer Protocol](#) (NNTP)
- 123 [Network Time Protocol](#) (NTP)
- 143 [Internet Message Access Protocol](#) (IMAP) of digital mail
- 161 [Simple Network Management Protocol](#) (SNMP)
- 194 [Internet Relay Chat](#) (IRC)
- 443 [HTTP Secure](#) (HTTPS) HTTP over TLS/SSL



# NETWORK SOCKETS: BASIC FUNCTIONS

```
/usr/include/sys/socket.h
```

```
int socket(int domain, int type, int protocol);
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *address, socklen_t *address_len);

int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
ssize_t send(int socket, const void *message, size_t length, int flags);
ssize_t recv(int socket, void *buffer, size_t length, int flags);
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
ssize_t sendto(int socket, const void *message, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t dest_len);
ssize_t recvfrom(int socket, void *buffer, size_t length, int flags,
                 struct sockaddr *address, socklen_t *address_len);

int close (it);
int shutdown(int socket, int how)
```



# SOCKET

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

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

Returns file descriptor on success, -1 on error

(in fact, socket descriptor, a communication endpoint with same read/write interface as pipes)



# COMMUNICATION DOMAIN

Specifies the nature of communication, including the address format

- AF\_INET: IPv4 Internet domain
- AF\_INET6: IPv6 Internet domain
- AF\_UNIX: UNIX domain (AF\_LOCAL)
- AF\_UNSPEC: unspecified (i.e., any)



# TYPE

## Communication characteristics

SOCK\_DGRAM: fixed-length, connectionless (SMTP-like), unreliable messages

SOCK\_STREAM: sequenced, reliable, bidirectional, connection-oriented (phone-like), byte streams, no message boundary

SOCK\_SEQPACKET: sequenced, reliable, connection-oriented (phone-like), fixed-length, message-based

SOCK\_RAW: user-defined protocol headers (superuser access), datagram interface to IP (optional in POSIX.1)



# PROTOCOL

Usually zero: default protocol for the given domain & socket type

`IPPROTO_TCP` socket in `AF_INET` domain (TCP)

`IPPROTO_UDP` socket in `AF_INET` domain (UDP)

When multiple protocols are supported for the same domain and socket type, we use the protocol argument to specify the specific protocol.



# INTERNET SOCKET ADDRESS (AF\_INET)

Define socket endpoint in IPv4 domain (AF\_INET)

```
struct in_addr {  
    in_addr_t    s_addr;  
}  
typedef uint32_t in_addr_t;  
// 0x0A0C6E39 -> 10.12.110.57
```

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

```
struct sockaddr {  
    sa_family_t sa_family;  
    char sa_data[14];  
}
```

1. Specify **sin\_port** & **sin\_addr** in network order
2. Cast **sockaddr\_in** to **sockaddr**



# INTERNET SOCKET ADDRESS (AF\_INET6)

Define socket endpoint in IPv6 domain (AF\_INET6)

```
struct in6_addr {  
    uint8_t s6_addr[16]; /* IPv6 address */  
};  
  
struct sockaddr_in6 {  
    sa_family_t sin6_family; /* address family */  
    in_port_t sin6_port; /* port number */  
    uint32_t sin6_flowinfo; /* traffic class, flow info */  
    struct in6_addr sin6_addr; /* IPv6 address */  
    uint32_t sin6_scope_id; /* interfaces for scope */  
};
```

Similarly, cast to generic sockaddr

```
struct sockaddr {  
    sa_family_t sa_family;  
    char sa_data[14];  
};
```



# HOST BYTE ORDER

How processor arch stores byte strings in larger data types?

- ***Little Endian systems*** (e.g., FreeBSD, Linux, MacOSX, Windows) store the *Least Significant Byte (LSB)* first
  - Example: int a = **0x0A0C6E39** (integer 168586809) is stored in memory as **396E0C0A**
- ***Big Endian systems*** (e.g., Solaris on SPARC, AIX, Mac@PowerPC) store the *Most Significant Byte (MSB)* first
  - Example: int a = **0x0A0C6E39** (integer 168586809) is stored in memory as **0A0C6E39**



# BYTE-ORDER CONVERSION FUNCTIONS

Convert byte to network order (**big endian**), otherwise two nodes cannot communicate (header issue)

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t host32bitval);
uint16_t htons(uint16_t host16bitval);
uint32_t ntohl(uint32_t net32bitval);
uint16_t ntohs(uint16_t net16bitval);
```

**h: host**  
**n: network**  
**l: long**  
**s: short**

## Byte manipulations

```
#include <string.h>
void *memset(void *dst, int c, size_t len);
void *memcpy(void *dst, void *src, size_t nbytes);
void *memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```



# SOCKET, NAME & ADDRESS FUNCTIONS

- Socket info (options, local/foreign protocol address)
  - `getsockopt` and `setsockopt`
  - `getsockname` and `getpeername` (return address after accept)
- Address conversion functions (dots to numbers and vice versa)
  - `inet_aton/inet_addr/inet_ntoa` (`_pton/_ntop` for IPv6)
- DNS name and address conversions
  - `getaddrinfo`, `gai_strerror`, `freeaddrinfo`
  - `gethostbyname`, `gethostbyaddr`
  - `getservbyname`, `getservbyport`



# REENTRANCY

Many functions are not reentrant (can cause errors in code with signals or threads)

- e.g., `malloc`, `inet_ntoa`, *gethostbyname* and *getaddrbyname*
- newer systems have thread-safe implementations



# BIND

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr &addr,
        socklen_t len);
```

addr specifies an integer socket address that contains IP & port

| Address         | port | result                                      |
|-----------------|------|---------------------------------------------|
| INADDR_ANY      | 0    | system selects addr & port                  |
| INADDR_ANY      | ≠0   | system selects addr, user selects port      |
| INADDR_LOOPBACK | 0    | user selects localhost, system selects port |
| INADDR_LOOPBACK | ≠0   | user selects both localhost and port        |

Returns 0 on success, -1 on error and [\*errno\*](#) is set appropriately

Client does not have to bind; system assigns a dynamic port number



# EXAMPLE: BIND (SERVER)

```
int port = 30000;
struct sockaddr_in server;
server.sin_family = PF_INET; // Internet domain
server.sin_addr.s_addr = htonl(INADDR_ANY); // IP address
server.sin_port = htons(port); // port=0-> auto (in client)
/* Bind socket to address */
if (bind(sockfd, (struct sockaddr *) &server,
        sizeof (server)) < 0) {
    perror("bind");
    exit(1);
}
```



# LISTEN

Listen means the server willing to accept connect requests

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

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

backlog: number of pending connections that the kernel should queue up for the socket (outstanding connections). The socket will refuse connection requests if the queue is full.

Returns 0 on success, -1 on error and [errno](#) is set appropriately



# ACCEPT

Accept: Retrieve a connect request (block if none)

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

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

cliaddr is the client's address in (length in addrlen)

Returns connection file descriptor (cfd) on success, -1 on error

(errno:= EAGAIN or EWOULDBLOCK)



# CONNECT

Connect file descriptor `sockfd` to the address specified by `addr`

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

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

Returns 0, if the connection or binding succeeds (`cfid`), -1 on error

(`errno`:= EACCES, EADDRINUSE,  
EADDRNOTAVAIL, EAFNOSUPPORT, EAGAIN, EALREADY, EBADF,  
ECONNREFUSED, EFAULT, EINPROGRESS, EINTR, EISCONN,  
ENETUNREACH, ENOTSOCK, EPROTOTYPE, ETIMEDOUT)



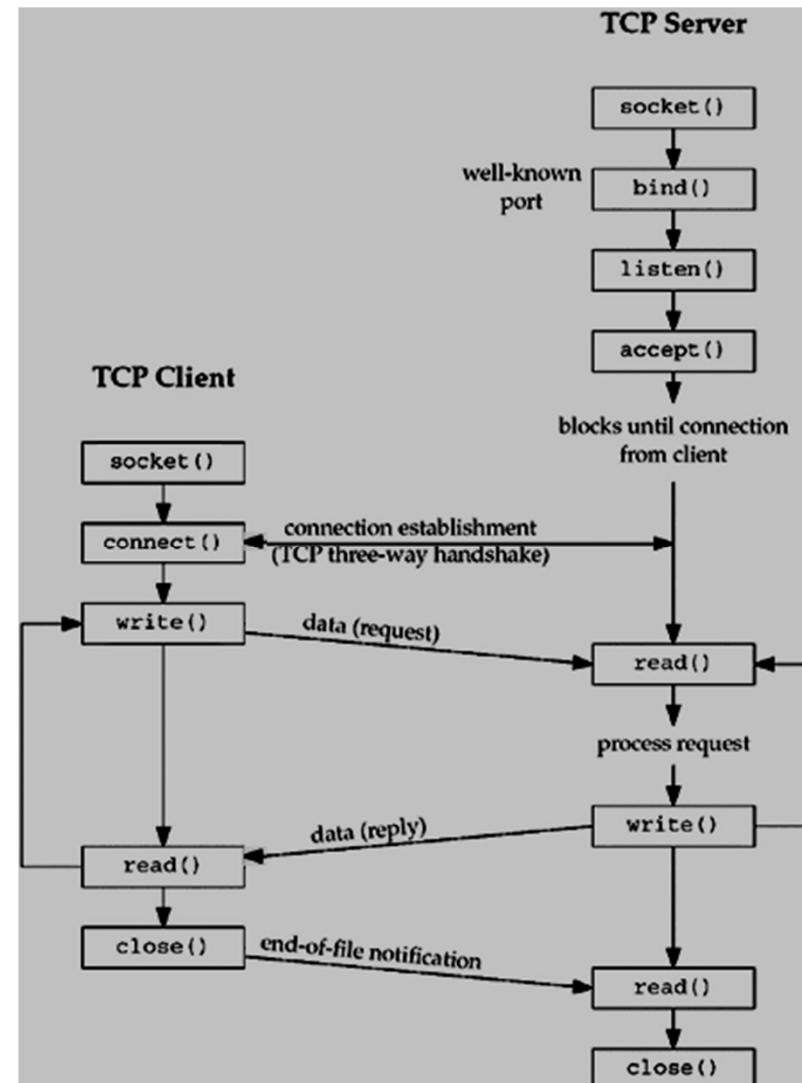
# DATA TRANSFER

## 3 functions to send data

- `send()`
- `sendto()`
- `sendmsg()`

## 3 functions to receive data

- `recv()`
- `recvfrom()`
- `recvmsg()`



## SEND - SIMILAR TO WRITE (IF FLAGS = 0)

```
#include <sys/socket.h>
ssize_t send(int cfd, const void *buf, size_t nbytes,
             int flags);
```

Returns: #bytes sent if OK, -1 on error

After send returns, data has been delivered to network drivers without error  
(but is not yet received!)



# SEND FLAGS

| Flag          | Description                                                        | POSIX.1 | FreeBSD<br>5.2.1 | Linux<br>2.4.22 | Mac OS X<br>10.3 | Solaris<br>9 |
|---------------|--------------------------------------------------------------------|---------|------------------|-----------------|------------------|--------------|
| MSG_DONTROUTE | Don't route packet outside of local network.                       |         | •                | •               | •                | •            |
| MSG_DONTWAIT  | Enable nonblocking operation (equivalent to using O_NONBLOCK).     |         | •                | •               | •                |              |
| MSG_EOR       | This is the end of record if supported by protocol.                | •       | •                | •               | •                |              |
| MSG_OOB       | Send out-of-band data if supported by protocol (see Section 16.7). | •       | •                | •               | •                | •            |



# RECV - SIMILAR TO READ (IF FLAGS = 0)

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

```
ssize_t recv(int cfd, void *buf, size_t nbytes,  
             int flags);
```

Returns: #bytes in msg, 0 if no msg & peer shutdown, or -1 on error

The flags argument is formed by ORing several values

| Flag        | Description                                                                       | POSIX.1 | FreeBSD<br>5.2.1 | Linux<br>2.4.22 | Mac OS X<br>10.3 | Solaris<br>9 |
|-------------|-----------------------------------------------------------------------------------|---------|------------------|-----------------|------------------|--------------|
| MSG_OOB     | Retrieve out-of-band data if supported by protocol (see Section 16.7).            | •       | •                | •               | •                | •            |
| MSG_PEEK    | Return packet contents without consuming packet.                                  | •       | •                | •               | •                | •            |
| MSG_TRUNC   | Request that the real length of the packet be returned, even if it was truncated. |         |                  | •               |                  |              |
| MSG_WAITALL | Wait until all data is available (SOCK_STREAM only).                              | •       | •                | •               | •                | •            |

## SENDTO – CONNECTIONLESS SOCKET (NO CONNECT)

```
#include <sys/socket.h>
ssize_t sendto(int cfd, const void *buf, size_t nbytes,
               int flags, const struct sockaddr *destaddr,
                   socklen_t destlen);
```

Returns: #bytes sent if OK, -1 on error

For connectionless socket, without first calling connect ()



# RECVFROM – CONNECTIONLESS SOCKET (NO CONNECT)

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

```
ssize_t recvfrom(int cfd, void *restrict buf, size_t len,  
                 int flags, struct sockaddr *restrict addr,  
                 socklen_t *restrict addrlen);
```

Returns: #bytes in msg, 0 if no msg & peer shutdown, or -1 on error

Used by connectionless socket



# SENDMSG - MULTIPLE BUFFERS

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

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Returns: #bytes sent if OK, -1 on error

```
struct msghdr {  
    void *msg_name; // opt addr  
    socklen_t msg_namelen; // #bytes in addr  
    struct iovec *msg_iov; // array of I/O bufs  
    int msg_iovlen; // #elements in array  
    void *msg_control; // ancillary data  
    socklen_t msg_controllen; // #bytes ancill  
    int msg_flags; // flags for recd msg (ignored)  
    ...  
};
```



# RECVMSG - MULTIPLE BUFFERS

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

```
ssize_t recvmsg(int cfd, struct msghdr *msg, int flags);
```

Returns: #bytes in msg, 0 if no msg & peer shutdown, -1 on error.

The msg\_flags field in the msghdr is set

| Flag                       | Description                                                               | POSIX.1 | FreeBSD<br>5.2.1 | Linux<br>2.4.22 | Mac OS X<br>10.3 | Solaris<br>9 |
|----------------------------|---------------------------------------------------------------------------|---------|------------------|-----------------|------------------|--------------|
| MSG_CTRUNC<br>MSG_DONTWAIT | Control data was truncated.<br>recvmsg was called in<br>nonblocking mode. | •       | •                | •<br>•          | •                | •<br>•       |
| MSG_EOR                    | End of record was received.                                               | •       | •                | •               | •                | •            |
| MSG_OOB                    | Out-of-band data was received.                                            | •       | •                | •               | •                | •            |
| MSG_TRUNC                  | Normal data was truncated.                                                | •       | •                | •               | •                | •            |

# CLOSE DESCRIPTOR

```
close(sockfd) ;
```

Returns: 0 if OK, -1 on error



# SOCKET SHUTDOWN

Disable I/O on socket

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

```
int shutdown(int sockfd, int how);
```

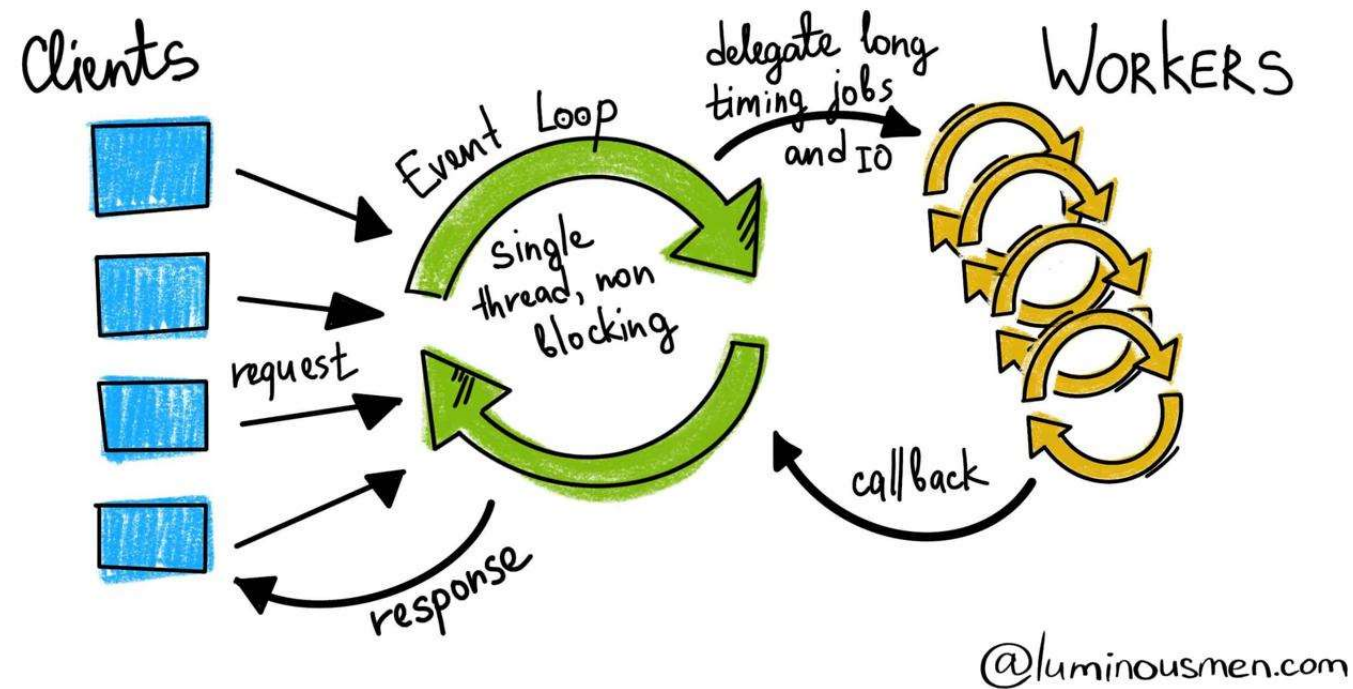
how:

- SHUT\_RD: disable read
- SHUT\_WR: disable write
- SHUT\_RDWR: disable read/write

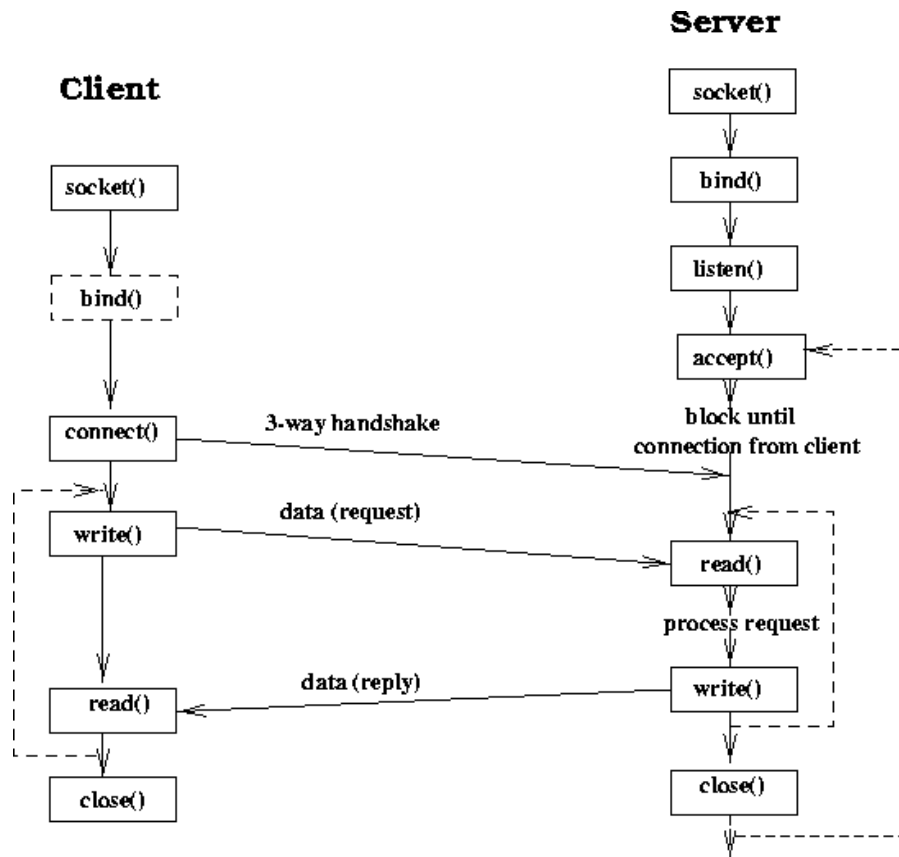
Returns: 0 if OK, -1 on error



# I/O MULTIPLEXING - CONCURRENT SERVERS ...



# BASIC SEQUENTIAL SERVER



## Standard Sequential I/O loop:

```
while ((n = read(cfd, buf, BUFSIZE)) > 0)
{
    ...
    if (write(fd2, buf, n) != n) {
        fprintf(stderr, "write error\n");
        exit(1);
    }
    ...
}
```

Read from multiple descriptors?

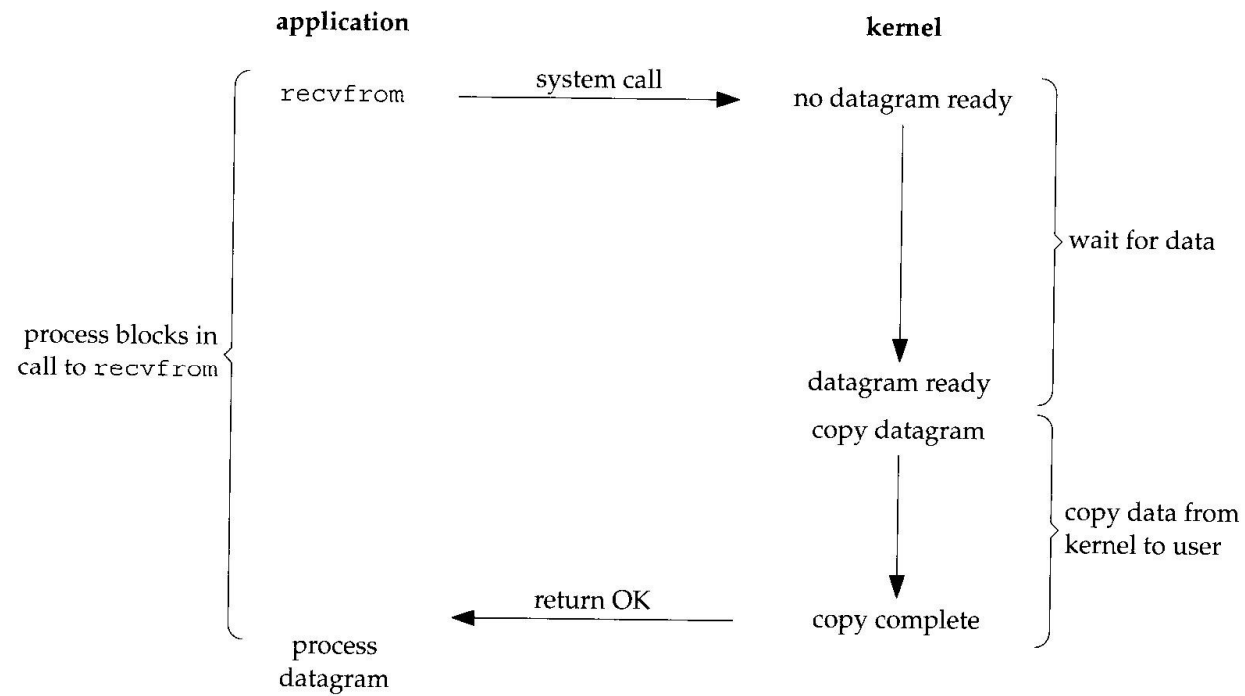


# SEQUENTIAL CODE

```
Int s1; /* socket descriptor 1 */
int s2; /* socket descriptor 2 */
/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */
while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */
    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```



# BLOCKING IO



# PERFORMANCE ISSUES

The code must process input from several peripheral devices

Input obtained using `read`

- Functions `read`, (`open`, `write`, `accept`, `connect`, `recv`, ...) may block causing the process to be put to sleep
- While asleep the process cannot read data that is ready from another device
  - TCP: how to sleep until some data arrives?
  - UDP: how to sleep until a UDP datagram arrives?
- Once input arrives (or fd is available), the OS wakes up the process and `read` (or `write ...`) return



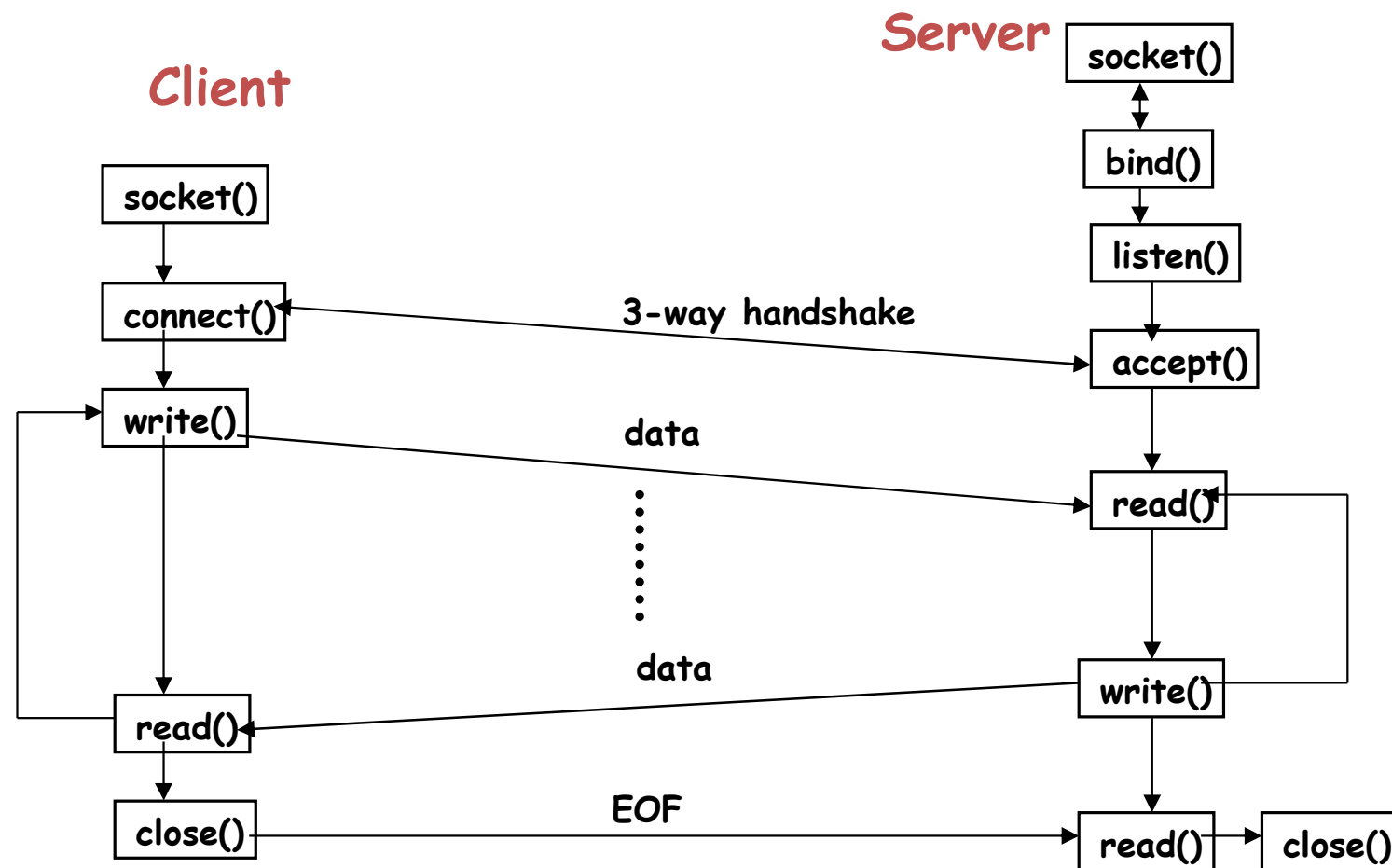
# I/O MULTIPLEXING - SOLUTIONS

Handling I/O on multiple file descriptors

- **blocking** : open one fd, block & wait (possibly forever), then test next fd (possibly fork)
- **forking/threading**: use one process/thread for each fd and communicate using signals or other IPC. Forked/Pre-forked servers vs Threaded/Pre-threaded servers.
- **asynchronous/multiplexed I/O**: get notified by the kernel when a fd is ready for I/O. Use select (or poll, epoll, kqueue in BSD, or libevent.org)
- **nonblocking I/O** (no busy-polling): open one fd, immediately get results, open next fd, get results, sleep for some time
- **signal-driven Asynchronous I/O**: POSIX or System V (signals SIGIO, SIGURG, ...)

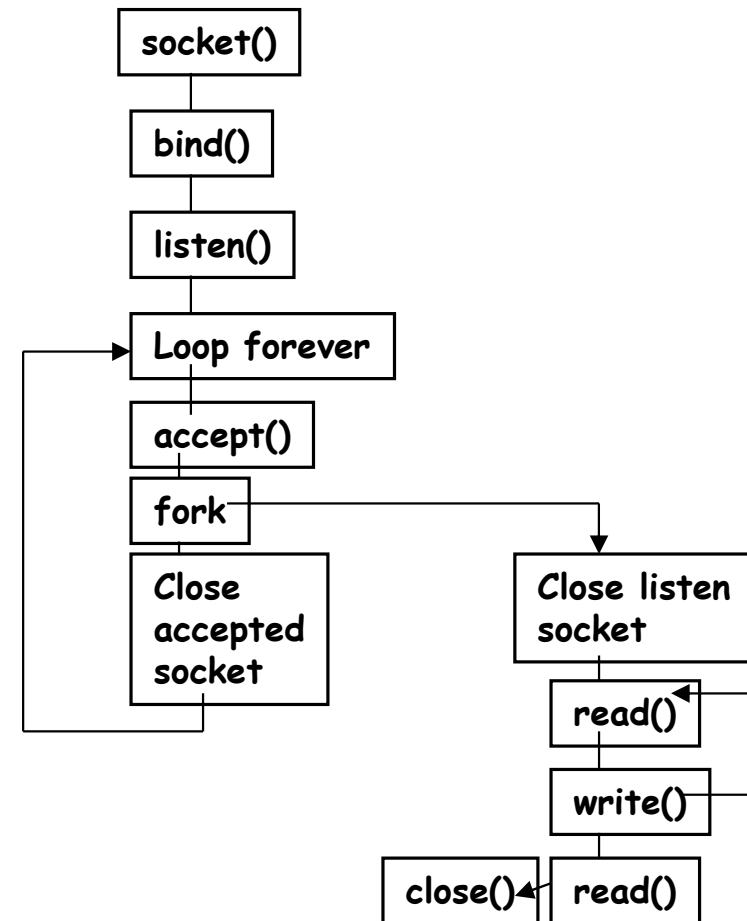
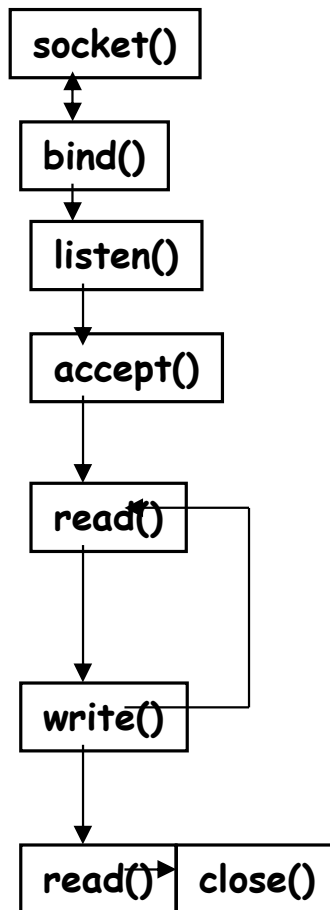


# SEQUENTIAL CLIENT/SERVER: TCP FLOW

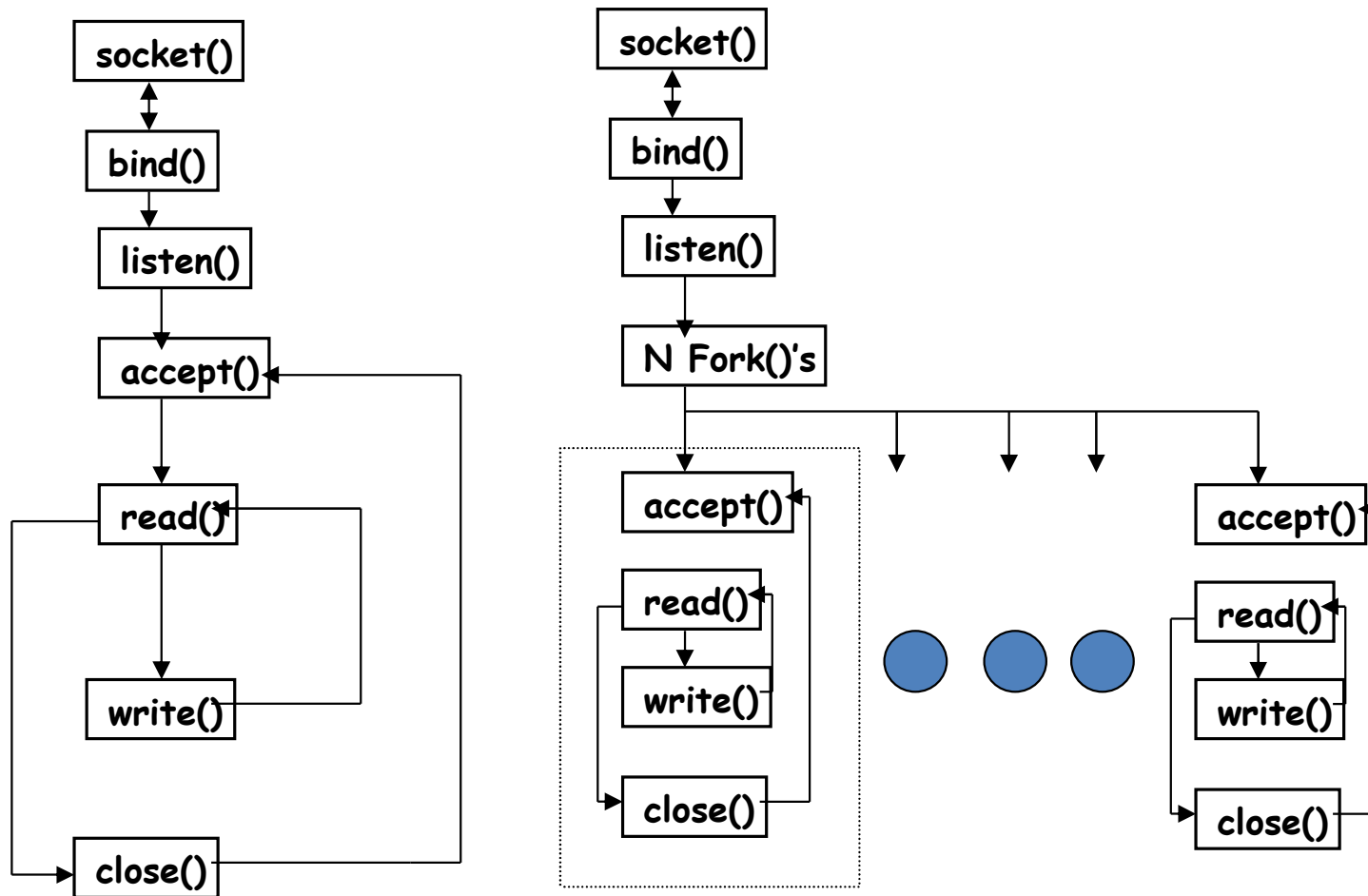


# SEQUENTIAL VS CONCURRENT SERVER

Handle one connection at a time



# SEQUENTIAL VS. PREFORKED SERVER



# FORKING CONCURRENT SERVER

```
listenfd = Socket( ... )  
initialize server address  
bind( listenfd, ... )  
listen(listenfd,...);  
for ( ;; ) {  
    // wait client connection  
    Connfd = Accept(listenfd,...);
```

```
    If ((pid = Fork() ) == 0) {  
        // Child Server  
        close(listenfd);  
        service_client(connfd);  
        close(connfd);  
        exit(0);  
    } else  
        // Parent  
        close(connfd);  
}
```



# THREADED SERVER

Limitations with process-based concurrent servers

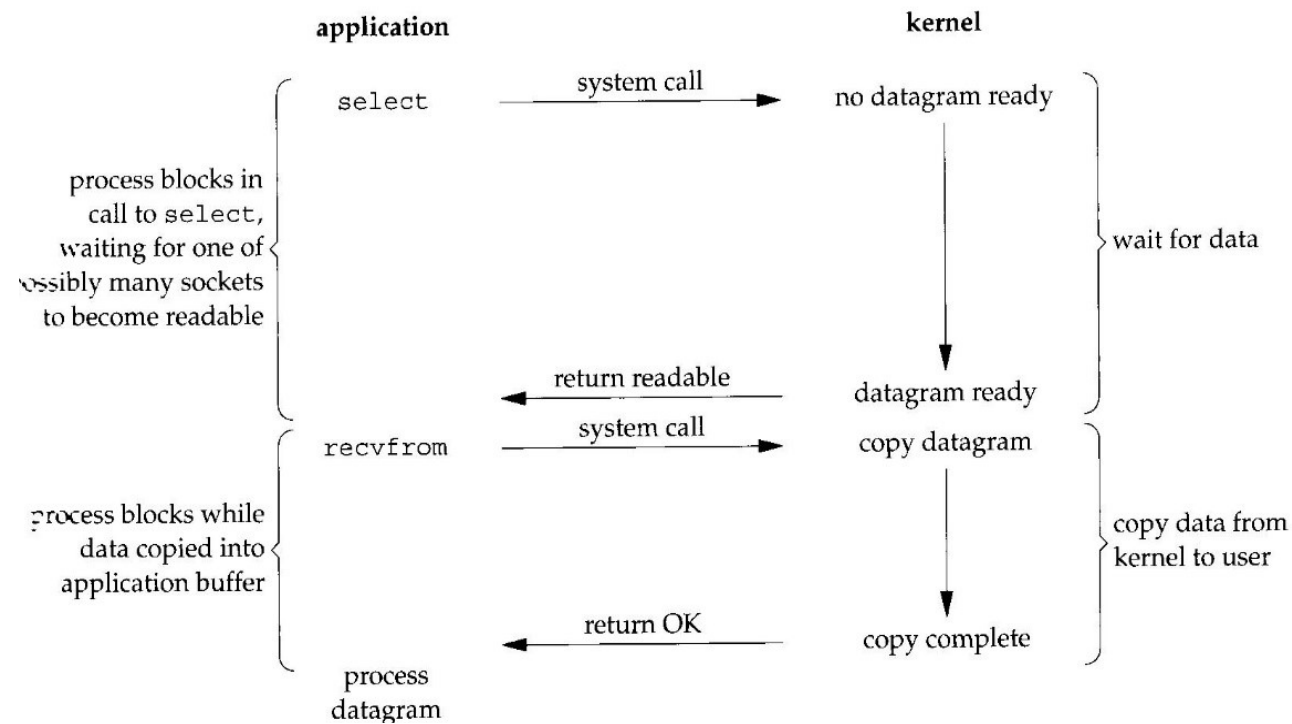
- Fork is expensive
  - Pre-fork? How many processes?
- Inter-process communications are expensive

Threads

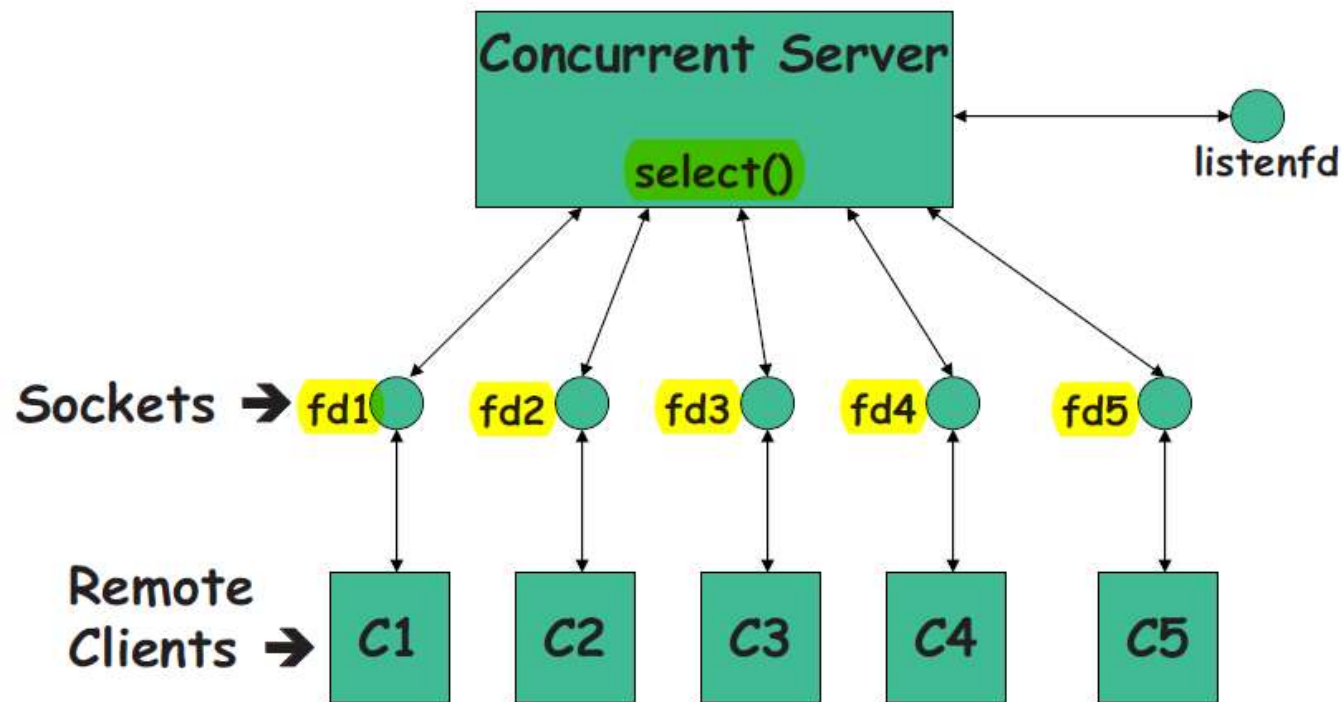
- Lightweight process
- Crashing one thread may kill the server (robustness?)
  - The client should know it right away!!
  - *Use select (allows monitoring many files and sockets)*



# NON-FORKING ASYNCHRONOUS/MULTIPLEXED I/O

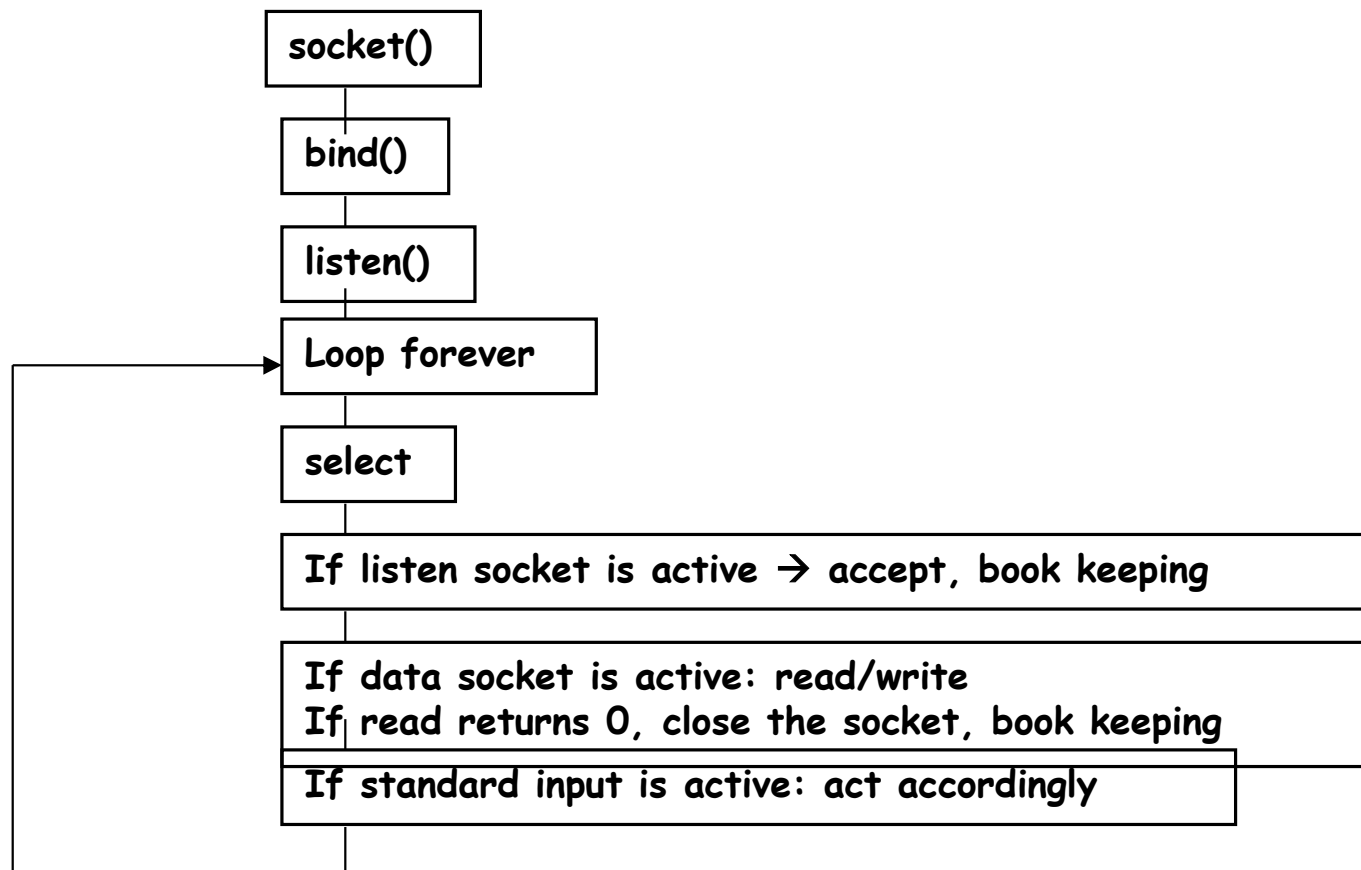


# ASYNCHRONOUS/MULTIPLEXED I/O - SELECT



- With `select`, instead of having a process for each request, there is *usually only one process* that "multiplexes" requests

# MULTIPLEXED SERVER



# ASYNCHRONOUS/MULTIPLEXED I/O – SELECT

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

int select(int maxfds, fd set *rfds, fd set *wfds, fd set *excfds,
           struct timeval *tptr);
```

maxfds: total no. descriptors, i.e., (0, 1, ... maxfds-1) will be tested

rfds/wfds descriptor sets ready for read/write

excfds indicates exception conditions (e.g, OOB data, certain terminal events)

tvptr a max wait period: (tptr == NULL wait forever, i.e., **block** until ready,  
tptr->tv\_sec=tptr->tv\_usec=0 **nonblocking**)

Returns: count of ready descriptors, 0 on timeout, -1 otherwise

file descriptors ready for I/O service within the specified time interval



# ASYNCHRONOUS I/O – FD\_\* FUNCTIONS

`poll` provides a conceptually similar interface

`pselect` provides fine grain time control and signal mask

`select` may block until being able to write to a file descriptor, or until data arrives in another file descriptor

File descriptor sets manipulated with `fd_*` functions/macros

```
FD_ZERO(fd_set *set) // clears a file descriptor set
```

```
FD_SET(int fd, fd_set *set) // add fd to set
```

```
FD_CLR(int fd, fd_set *set) // remove fd from set
```

```
FD_ISSET(int fd, fd_set *set) // check if fd in set
```

# EXAMPLE: ASYNCHRONOUS I/O (SELECT)

```
int s1, s2; // socket fd
fd_set readfds;
// create & bind s1, s2
while(1) {
    // init readfs set
    FD_ZERO(&readfds);
    // add s1, s2 to readfds
    FD_SET(s1, &readfds);
    FD_SET(s2, &readfds);
    if(select(2, &readfds,
              0, 0, 0) < 0) {
        perror("select");
        exit(1);
    }
}
```

```
if(FD_ISSET(s1,&readfds)){
    // process buf
    recvfrom(s1, buf, sizeof(buf), ...);
}
if(FD_ISSET(s2,&readfds)){
    // process buf
    recvfrom(s2, buf, sizeof(buf), ...);
}
```



# CONCURRENT VS ASYNCHRONOUS/MULTIPLEXED IO

Limitations with concurrent server

- Remove fork/thread overhead
  - Pre-fork/thread N processes to handle clients and use them forever
    - Not easy to guess the “right” number of child processes
- Is context switching a big problem?

With `select`, a single process to handle all requests. This is

- Implicit multiplexing results in simpler, more transparent programming. No need to express handling of multiple sockets/connections
- no IPC primitives (shared memory ...) for different tasks to communicate

Combine worlds: e.g., **pre-forked multiplexed servers?**



# NONBLOCKING I/O

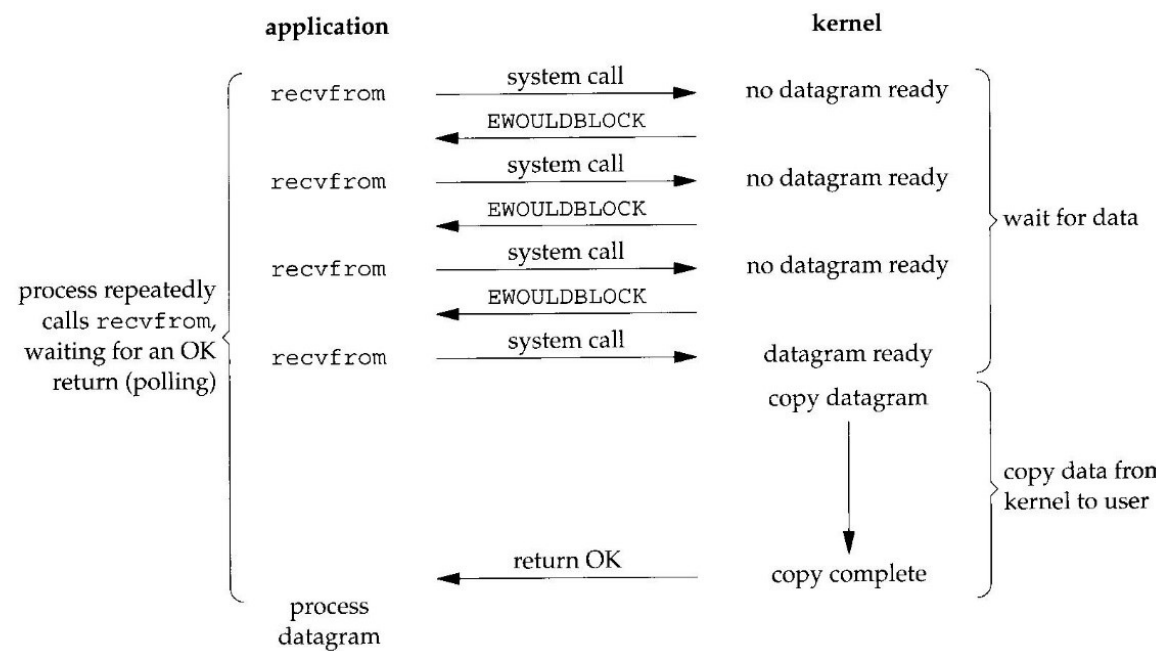


Figure 6.2 Nonblocking I/O model.

# NONBLOCKING I/O - SETUP USING FCNTL

Nonblocking I/O allows issuing an operation without blocking

- If operation cannot be completed, it returns immediately (EWOULDBLOCK or EAGAIN)
- Then, the process can do something else, or keep on polling (waste resources), or block until some operation can be performed (select)

To specify nonblocking mode use *file control* or *I/O control*

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

- `open(path, O_RDWR|O_NONBLOCK);` or
- `fd = socket(PF_INET, SOCK_STREAM, 0);`  
    `flags = fcntl(fd, F_GETFL, 0);`     `int fcntl(int fd, int cmd, long arg)`  
    `fcntl(fd, F_SETFL, flags|O_NONBLOCK);`

Starve other system applications by constantly calling read?

# NONBLOCKING I/O – SETUP USING IOCTL

## I/O control function

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ... )
```

- Request: FIONBIO

```
int on = 1
ioctl(sockfd, FIONBIO, (char *)&on);
```

