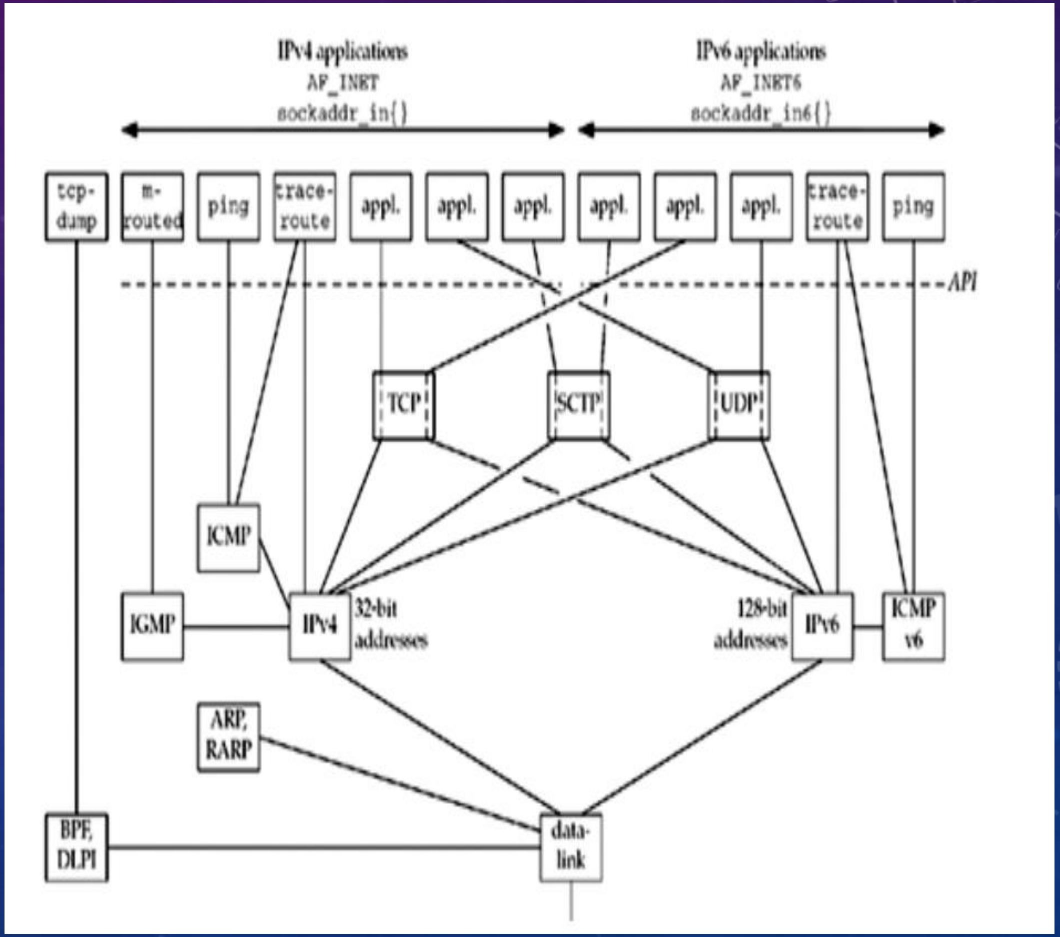
SOCKET

SOCKET OVERVIEW

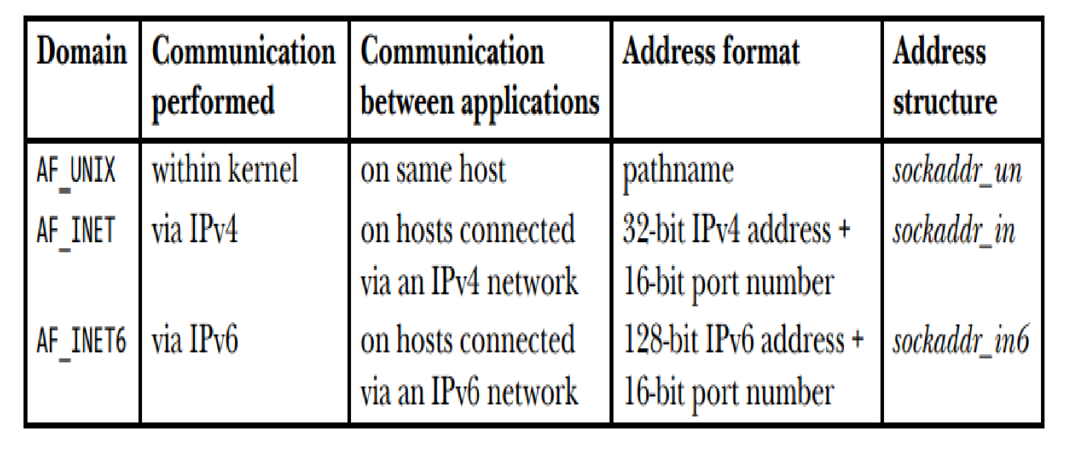
- Typical client-server scenario
  - Socket is window to the word
  - Each application open its own socket
    - Server bind its socket to a well-known address
    - Client connect its socket to that address

```
fd = socket(domain, type, protocol);
```



COMMUNICATION DOMAIN

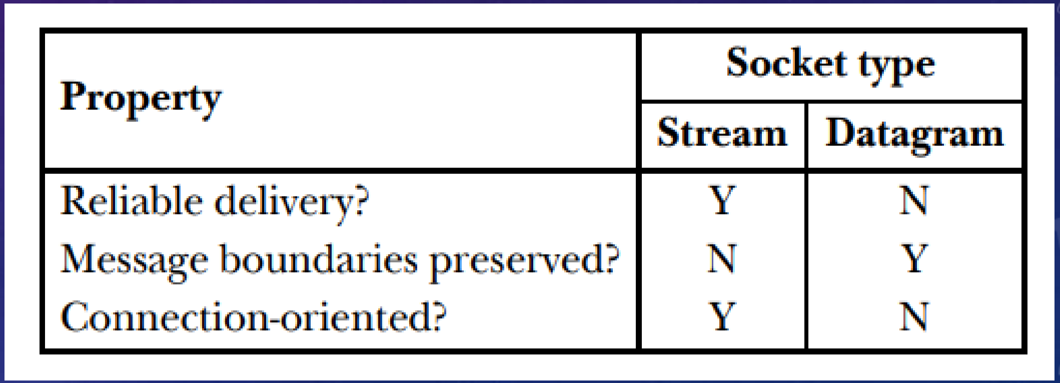
- Communication domain
  - Method of identifying a socket
  - The range of communication (same host or network)
- Type of Domain
  - UNIX : allow communication between application on the same host
  - IPv4: allow communication between applications running on hosts connected via IPv4 network
  - IPv6 allow communication between application running on hosts connected via IPv6



Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	*sockaddr_un*
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

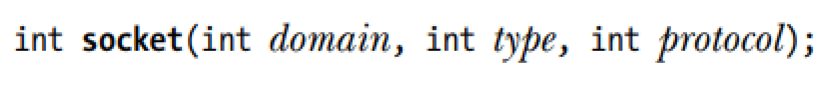
SOCKET TYPES

- Two types of socket
  - Stream
    - Reliable – Data at receiver exactly as data at transmitter
    - Bidirectional – Data transmit two directions
    - Byte-stream – No concept of boundary
    - TCP is typical
  - Datagram
    - Exchange in form of message called datagram
    - Message boundaries but data transmission is not reliable , message may be out-of-order, duplicate, not arrived
    - Don’t need to establish a connection
    - UDP is typical



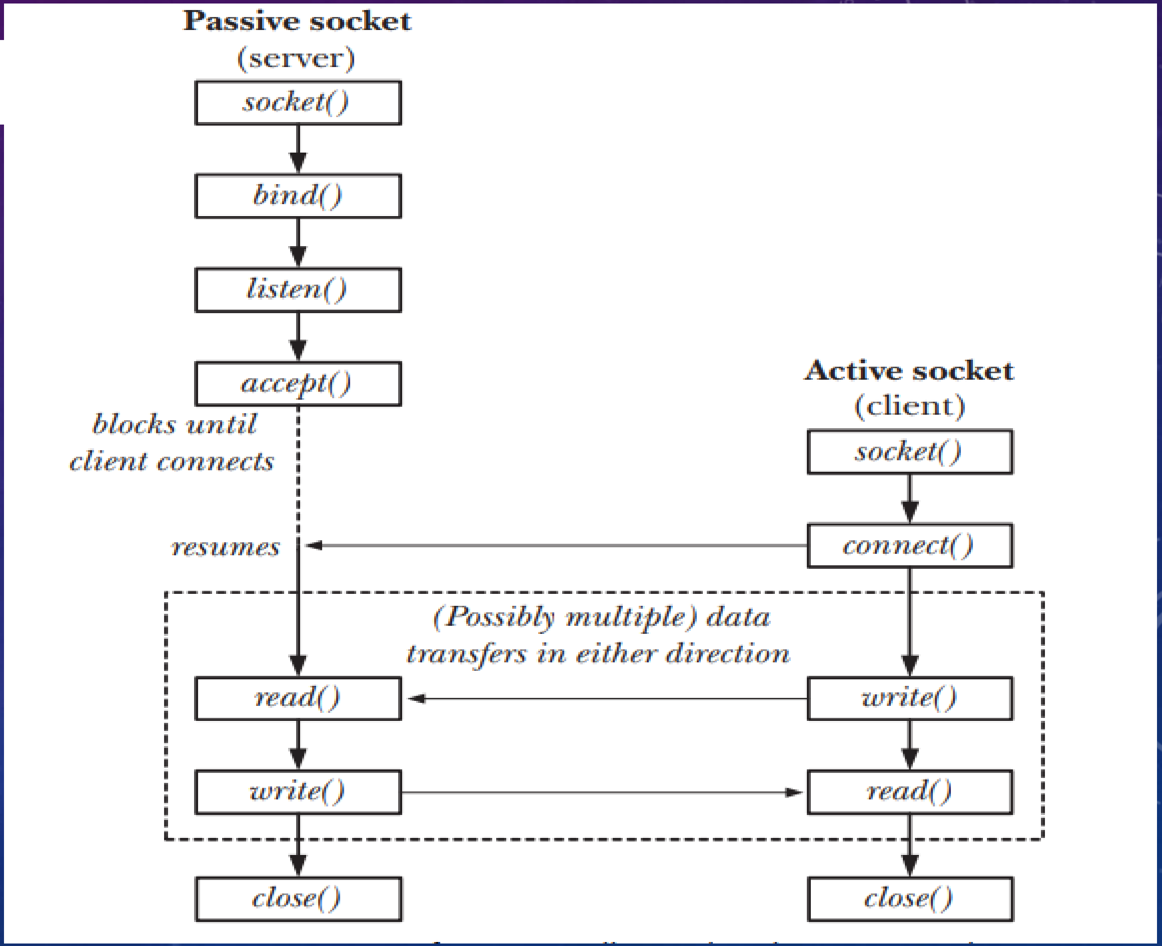
Property	Socket type	
	Stream	Datagram
Reliable delivery?	Y	N
Message boundaries preserved?	N	Y
Connection-oriented?	Y	N

SOCKET SYSTEM CALL- STREAM SOCKET FLOW

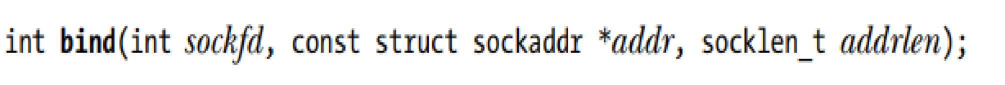


```
int socket(int domain, int type, int protocol);
```

- Socket
  - Domain AF_UNIX && AF_INET
  - Type SOCK_STREAM && SOCK_DGRAM && SOCK_RAW
  - Protocol IPPROTO_UDP, IPPROTO_TCP, 0
  - Return file descriptor
- Socket operate in connected pair
  - Peer socket – refer to socket at other end of a connection
  - Peer address – denotes the address of peer socket
  - Peer application – denotes application use peer socket

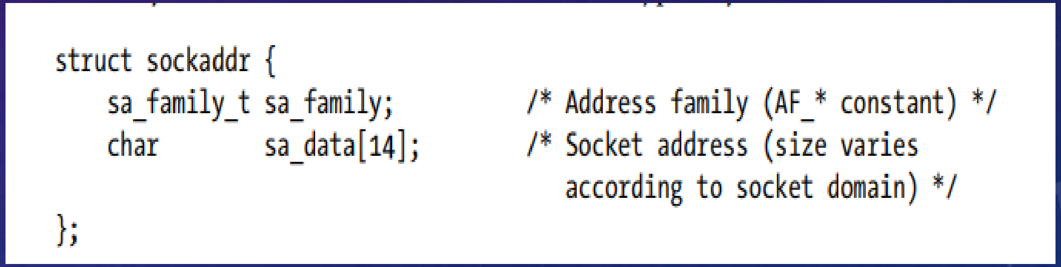


BIND A SOCKET TO AN ADDRESS



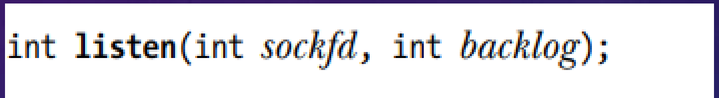
```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Bind a socket to an well-known address
  - Error Address already in use
  - Use netstat –a to find which application open it



```
struct sockaddr {
    sa_family_t sa_family;      /* Address family (AF_* constant) */
    char        sa_data[14];    /* Socket address (size varies
                                   according to socket domain) */
};
```

LISTENING FOR INCOMING CONNECTION

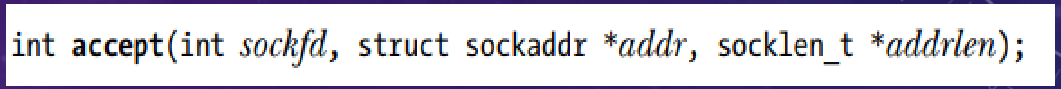


```
int listen(int sockfd, int backlog);
```

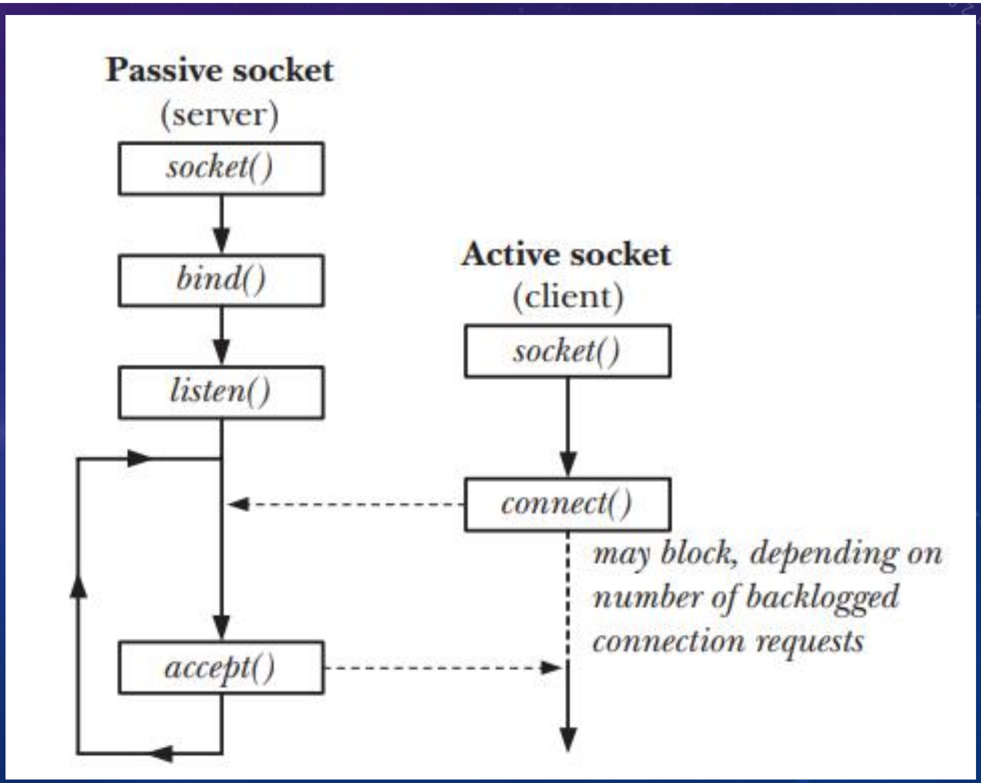
- Listen
  - Each connection request coming, kernel record information into pending connection queue → it consume kernel memory
  - /proc/sys/net/core/somaxconn
  - Backlog limit number of pending connection
  - netstat –l to find all listening port

ACCEPT A CONNECTION

- accept function
  - Accept an incoming connection on listening stream socket refer to by file descriptor
  - If no pending connection exist, accept block until connection request arrive
  - Accept create new socket and keep listening socket to accept new request
  - Addr and addrlen give the address of peer and length of that address
    - Pass NULL to irgnore it

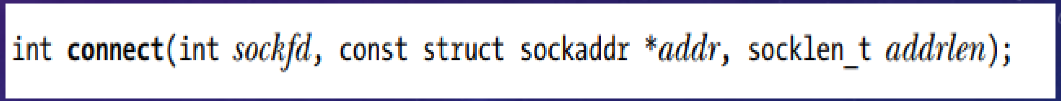


```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```



CONNECTING TO PEER SOCKET

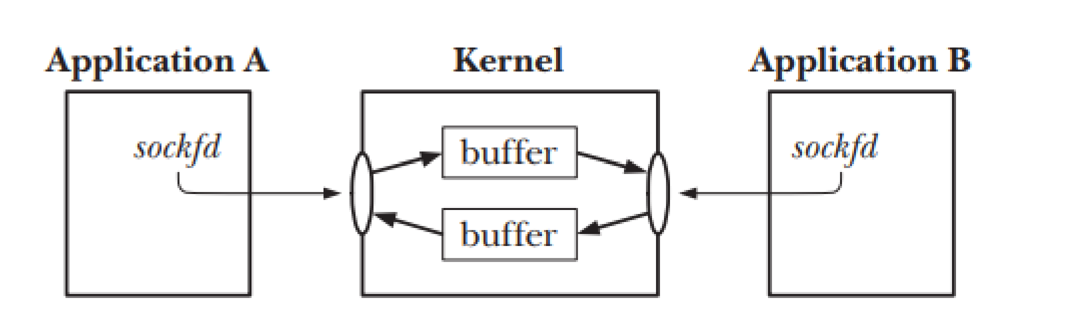
- Connect function
  - Connect to a socket refer to by file descriptor whose address specified by addr and addrlen
  - Establish a connection



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

IO STREAM OPERATION

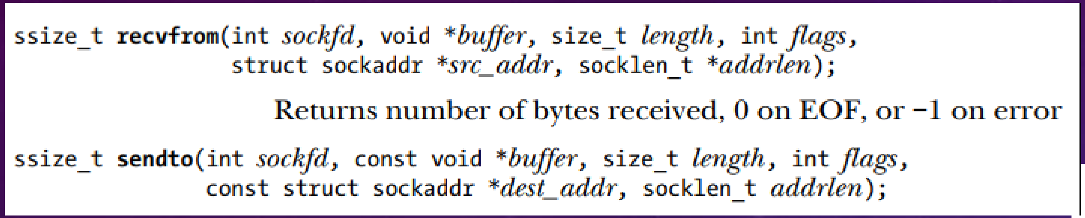
- Use read/write to request data and send data



TERMINATE A CONNECTION

- Usual way to terminate a connection use close
  - If multiple file descriptor refer to the same socket, the connection is terminated when all of file descriptor are closed
  - If peer crash, with datagram socket, we don’t know that event, with stream socket, it have some mechanism to notify that event
  -

DATAGRAM FLOW

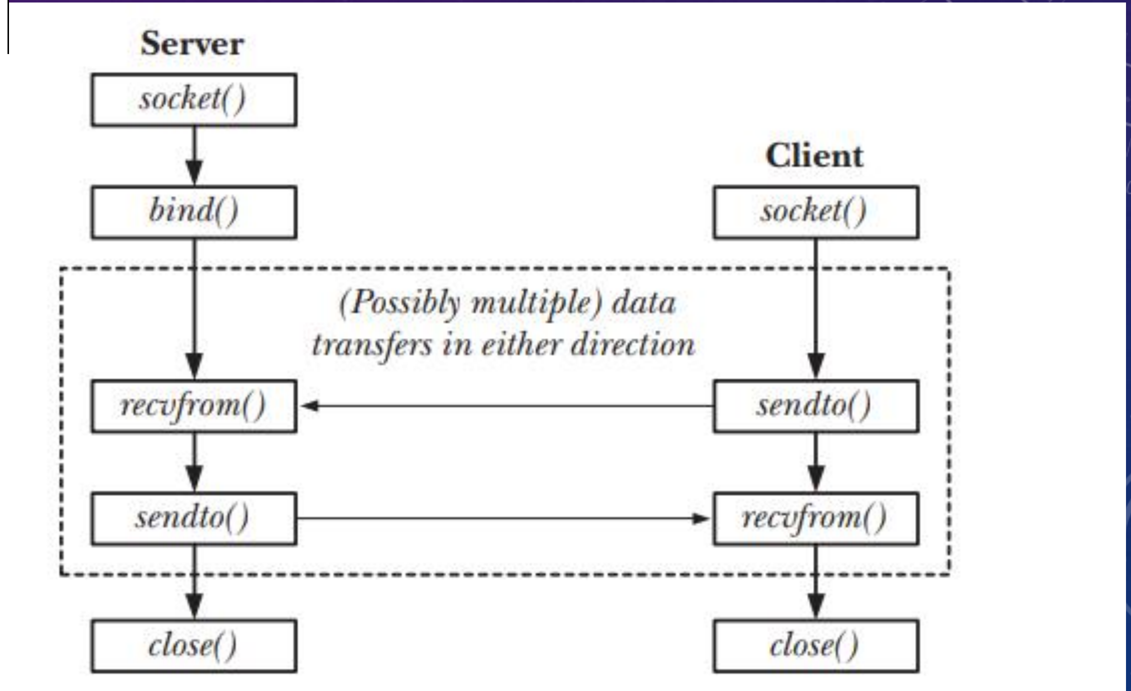


```
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Returns number of bytes received, 0 on EOF, or −1 on error

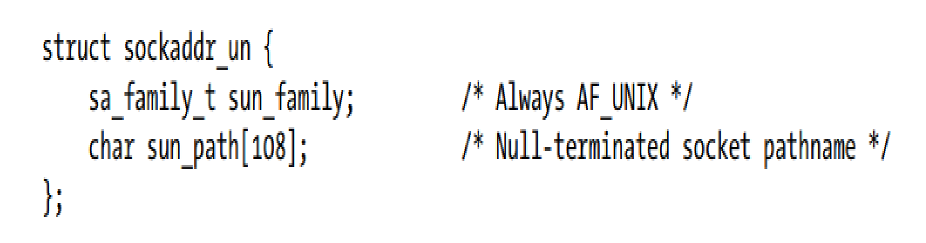
```
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

- sockfd – socket file descriptor
- buffer – start address of message to send
- Length – length of message to send
- Flag – control receive/send as ancillary data, it is usually set to 0
- addr – address of sender in recvfrom and address of receiver in sendto

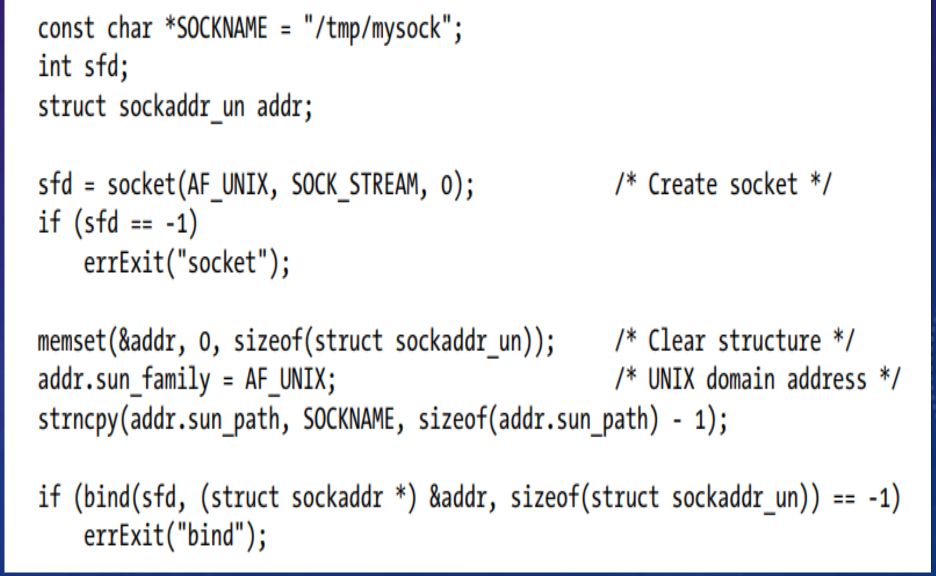


UNIX DOMAIN SOCKET

- Allow communication between processes on the same host system
- Unix Domain socket addresses
  - Take a form of pathname
  - Use snprintf or strncpy to avoid buffer overflow of pathname
  - Ls –l, we can see type s in first column
- Bind a Unix domain socket
  - cast an sockaddr_un to sockaddr
  - We can’t bind an existing pathname, error EADDRINUSE
  - Absolute pathname is appreciated
  - A socket can bind only one pathname, a pathname can be bound to only one socket
  - Can’t use open to open a socket
  - When the socket is no longer require, use unlink to remove it



```
struct sockaddr_un {
    sa_family_t sun_family;        /* Always AF_UNIX */
    char sun_path[108];            /* Null-terminated socket pathname */
};
```



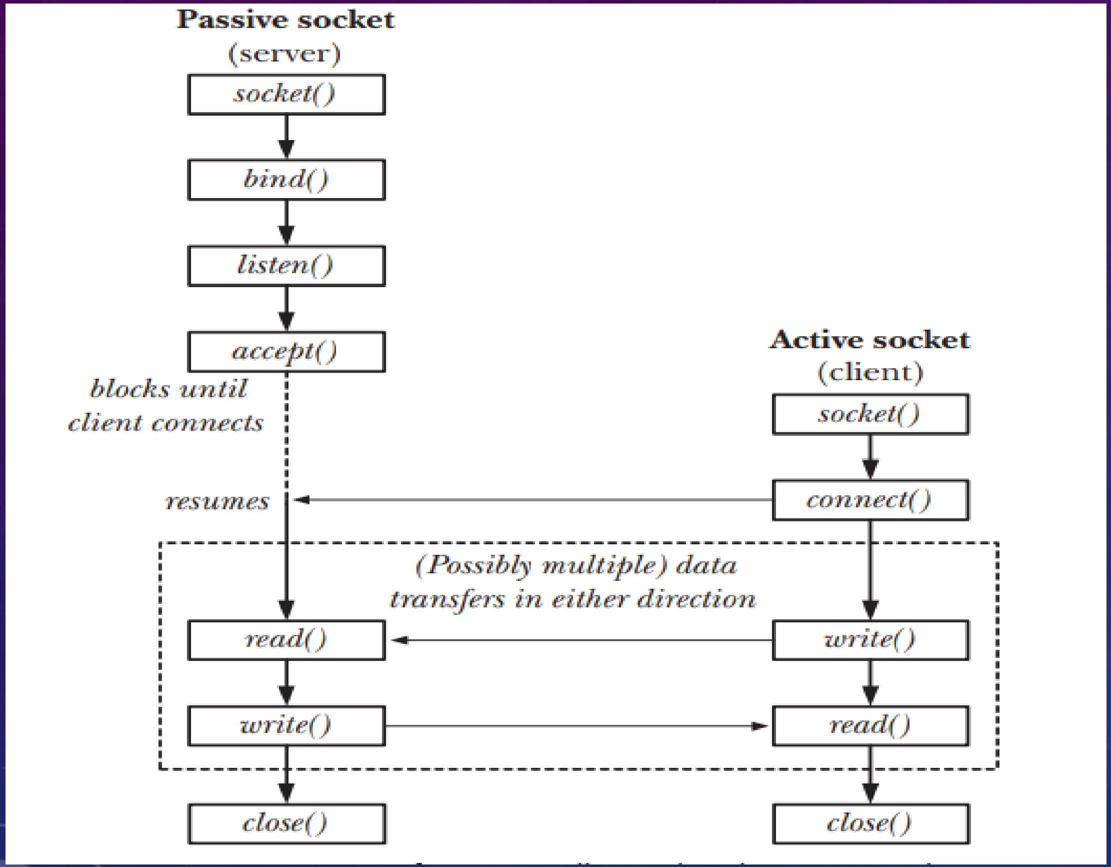
```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;

sfd = socket(AF_UNIX, SOCK_STREAM, 0);            /* Create socket */
if (sfd == -1)
    errExit("socket");

memset(&addr, 0, sizeof(struct sockaddr_un));    /* Clear structure */
addr.sun_family = AF_UNIX;                        /* UNIX domain address */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);

if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

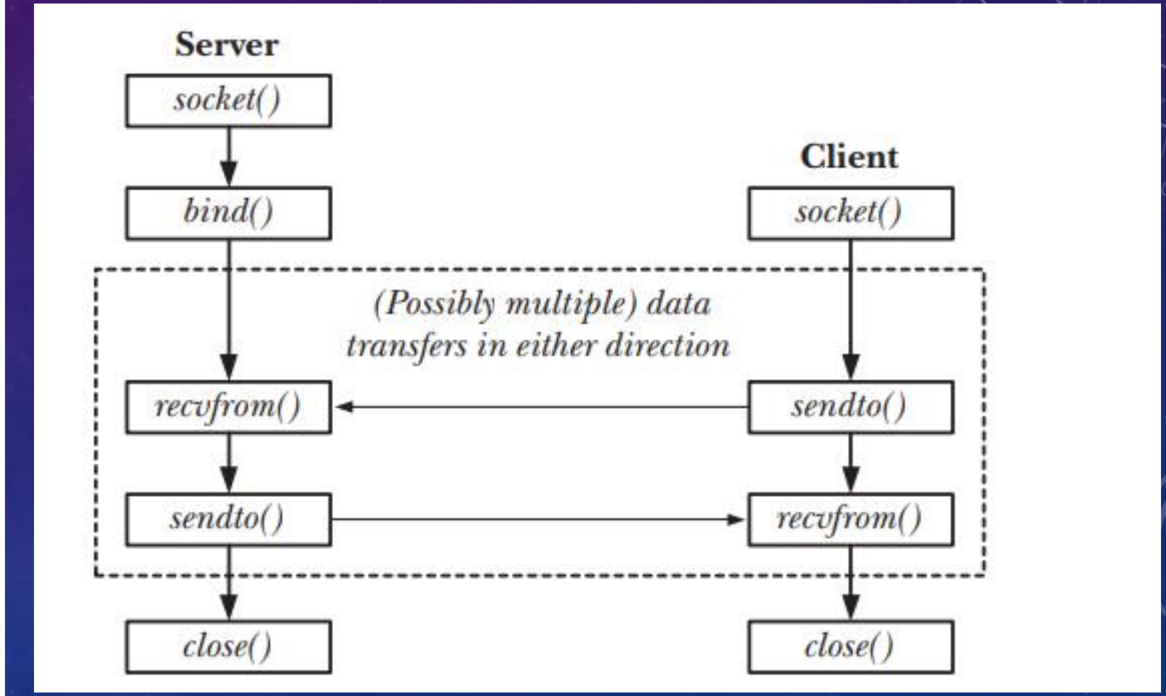
STREAM SOCKET IN UNIX DOMAIN

- iterative server
  - Create socket
  - Remove/unlink unix pathname
  - Construct an address structure for server socket, bind socket to address and mark socket as listening
  - Loop to handle incoming client request
    - Accept connection, obtain new socket
    - Read data from connected socket and write to standard output
    - Close the socket
  - Server terminate manually

- client
  - create socket
  - Construct an address structure for server socket and connect to socket at address
  - Loop to copies data from standard input to the socket connection until end of file

DATAGRAM SOCKET IN UNIX DOMAIN





MULTIPLEXING IO WITH SELECT

- Server/Client model
  - Server terminate and send goodbye message to client
  - Client busy to waiting command/message from stdio and cannot receive message from server
  - Client have to read from both stdin and socket
  - Client have multiple socket?

- Multiplexing is essential when
  - Client need to handling multiple descriptors(sockets + stdin)
  - TCP handle both a listening socket and connected socket
  - Server handle both TCP(Stream) and UDP(datagram)
  - Server handle multiple service and multiple protocol
  - Not limited to network programming. This technique use every where

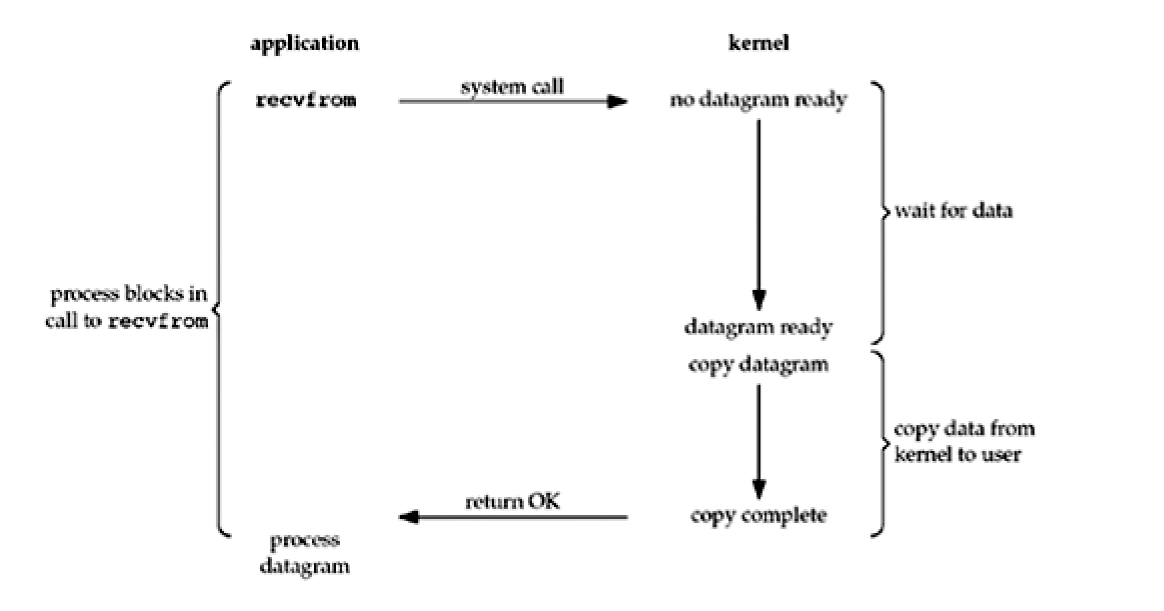
I/O MODEL – BLOCKING I/O

- 5 I/O model available
  - Blocking I/O
  - Non-blocking I/O
  - I/O multiplexing
  - Signal driven
  - Asynchronous I/O

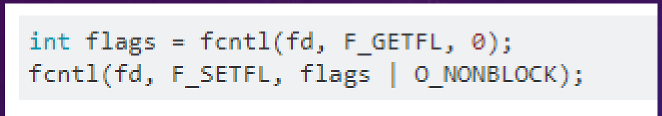
- Two phase of input operation
  - Waiting for the data to be ready
  - Copying the data from the kernel to the process

BLOCKING I/O

- By default, all sockets are blocking
  - Process waiting a message coming, but message has not arrived, process sleep to wait
    - Calling recvfrom cause context-switching and process fire CPU
  - Message is coming, process is wake up to process message
    - Recvfrom copy message from kernel buffer to user buffer
  - CPU usage optimization but can't handle multiple events

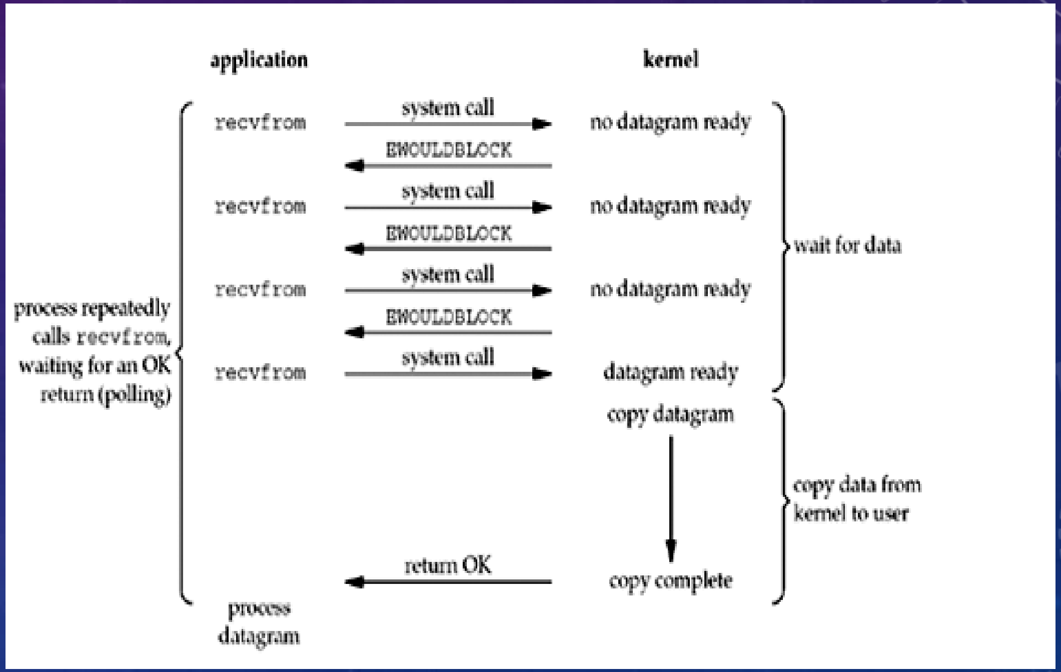


NON-BLOCKING I/O



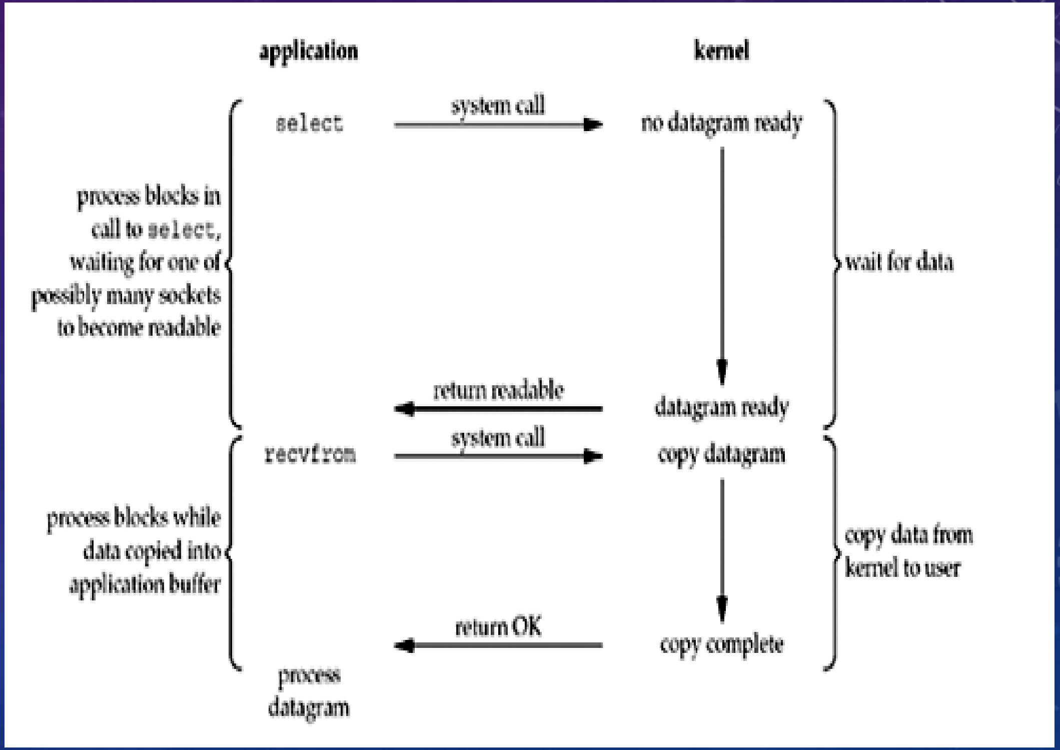
```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- Socket is set to Non-blocking I/O
  - O_NONBLOCK flags
  - Call recvfrom return immediately with EWOULDBLOCK error code
  - Can handle multiple file descriptor but waste CPU time



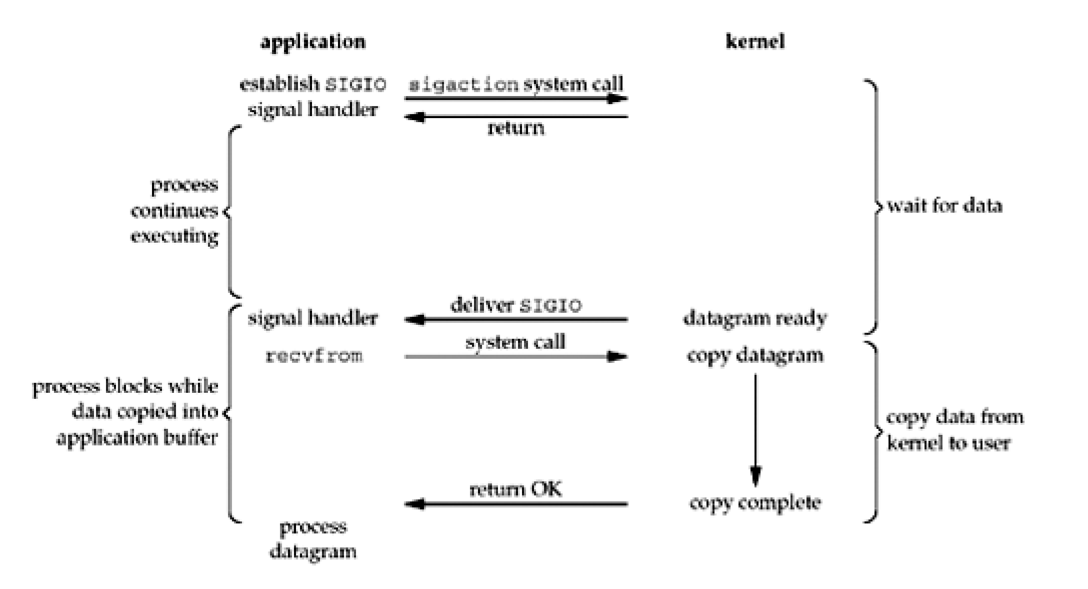
MULTIPLEXING I/O

- I/O multiplexing
  - Call select or polling to wait data from multiple descriptor available
  - Sleep on select instead recvfrom
  - Can handle multiple file descriptor and CPU usage optimization


application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

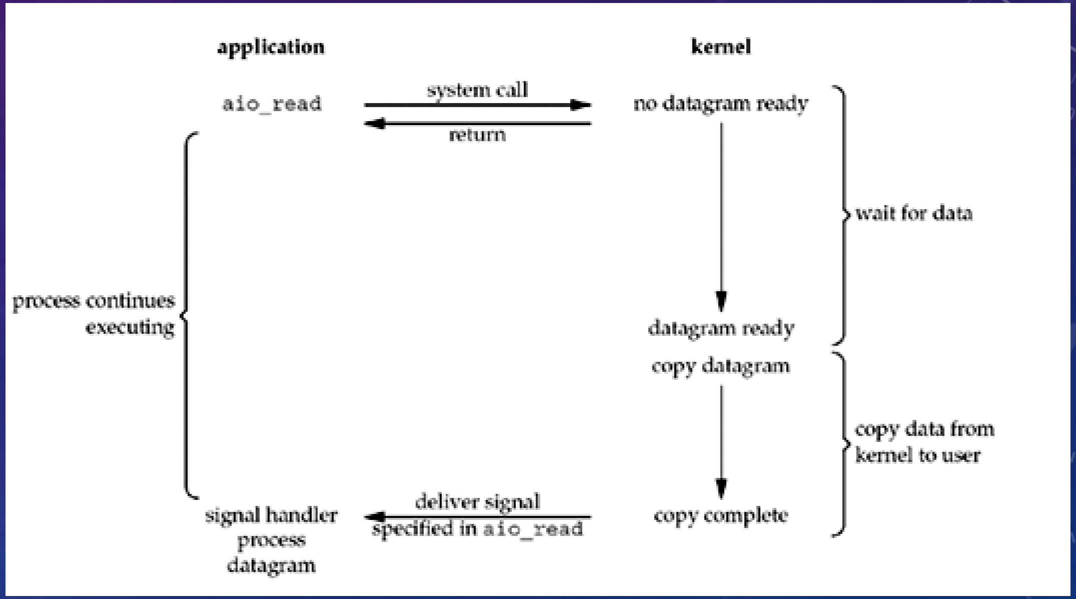
SIGNAL-DRIVEN I/O

- Signal-driven I/O
  - Enable socket for signal-driven I/O
  - Install a signal handler in process
  - When datagram ready to read, a SIGIO signal is generated and deliver to process
  - Can handle multiple file descriptor and CPU usage optimization but it it not popular



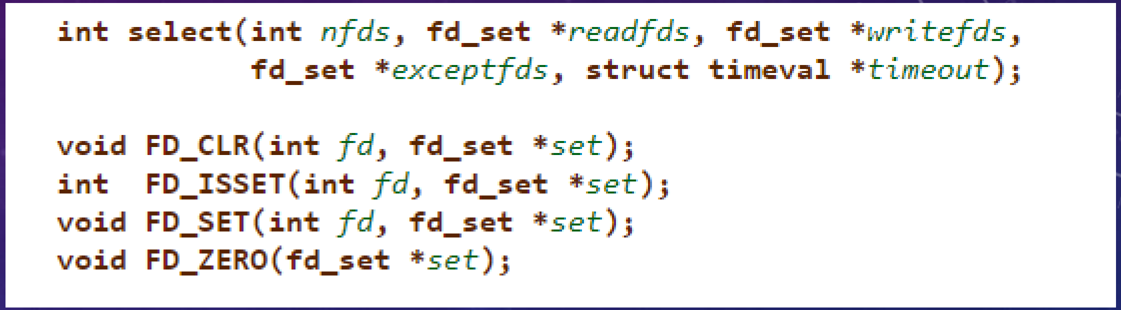
ASYNCHRONOUS I/O MODEL

- AIO
  - Call aio_read to notify to kernel start operation and notify us when the entire operation is complete
  - The main difference between this model and the signal-driven I/O model is kernel tell process when I/O operation initiated but in AIO, kernel tell us when an I/O operation is completed

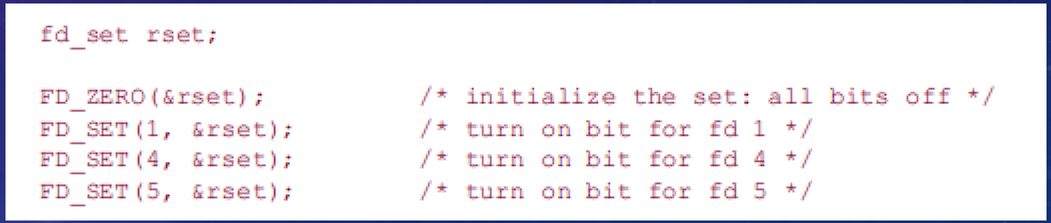

application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

SELECT

- Arguments
  - Max of file descriptor tested, its value is file descriptor has maximum value + 1
  - Readset, writeset, exeptionset collect all file descriptor tested (data ready)
  - Timeout - tell the kernel how long we wait the events
- Select blocking until
  - Any events in readset, writeset or exceptionset occur
  - Timer expire
  - File descriptor set/clear/zeroout/test by bitmap operation

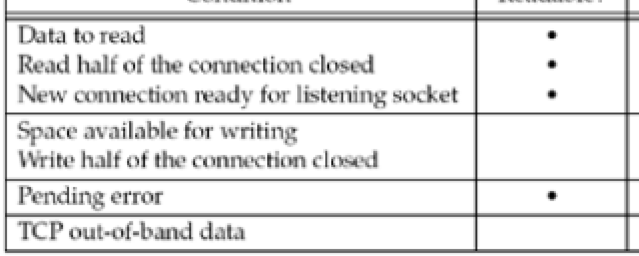


```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```



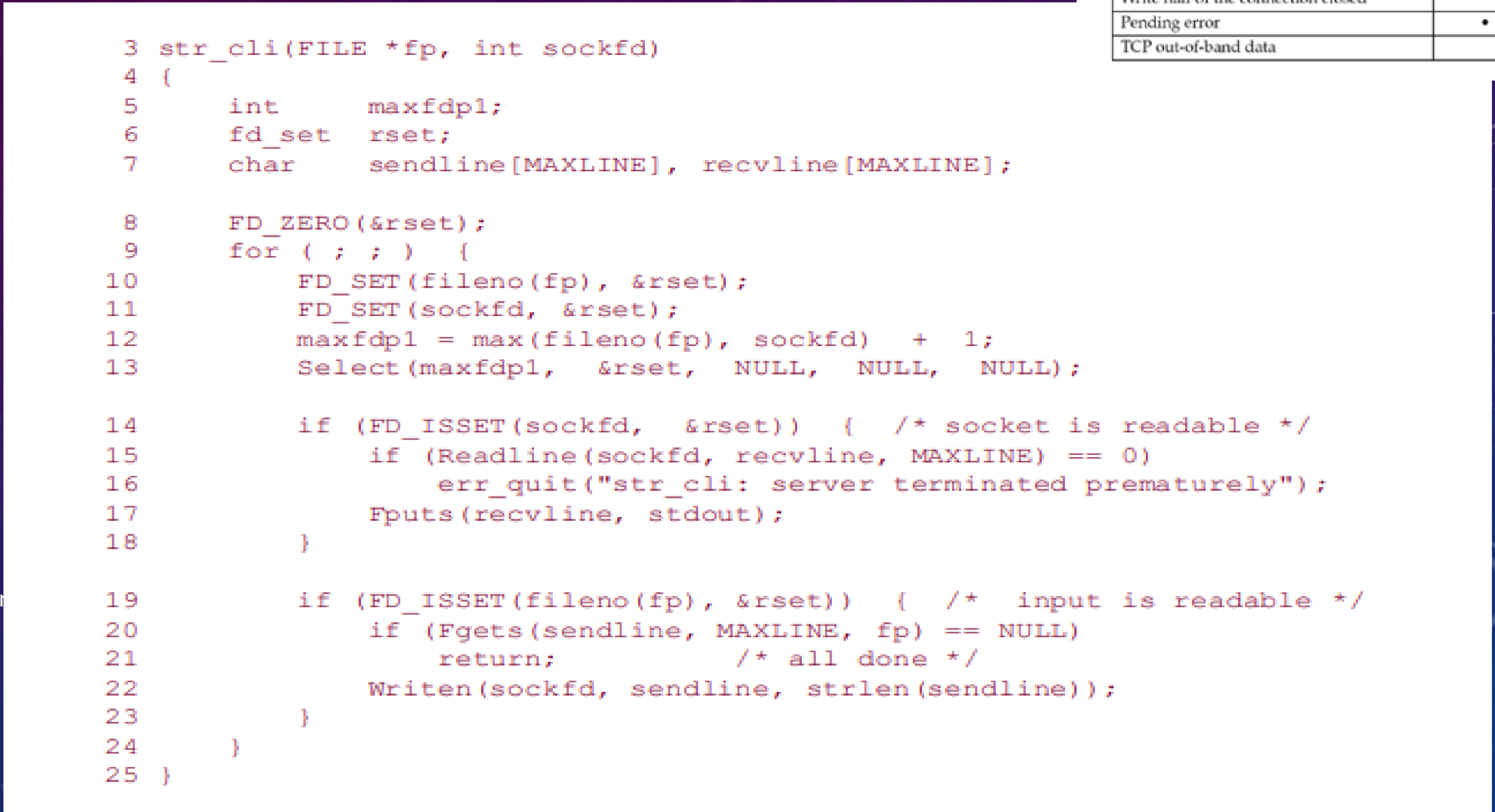
```
fd_set rset;

FD_ZERO(&rset);          /* initialize the set: all bits off */
FD_SET(1, &rset);        /* turn on bit for fd 1 */
FD_SET(4, &rset);        /* turn on bit for fd 4 */
FD_SET(5, &rset);        /* turn on bit for fd 5 */
```

SEL



Data to read	•
Read half of the connection closed	•
New connection ready for listening socket	•
Space available for writing	
Write half of the connection closed	
Pending error	•
TCP out-of-band data	

- With
    -
    -
    -
- Initia
    -
    -
- Wher



```
str_cli(FILE *fp, int sockfd)
{
    int     maxfdp1;
    fd_set  rset;
    char    sendline[MAXLINE], recvline[MAXLINE];

    FD_ZERO(&rset);
    for ( ; ; ) {
        FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd) + 1;
        Select(maxfdp1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0)
                err_quit("str_cli: server terminated prematurely");
            Fputs(recvline, stdout);
        }

        if (FD_ISSET(fileno(fp), &rset)) {  /*  input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL)
                return;         /* all done */
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```