**COMPUTER NETWORKS**
**UNIT-V**

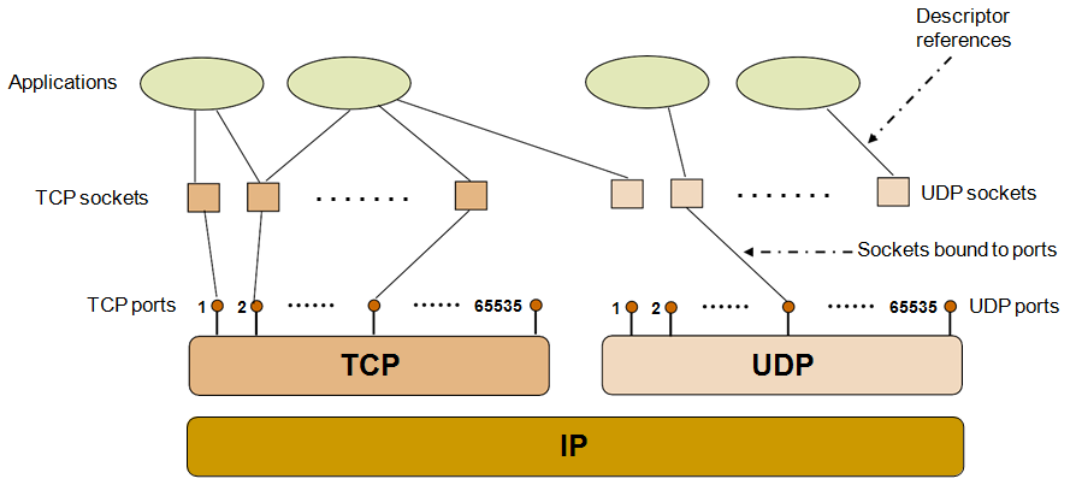
**Socket Programming:**

**Socket Address:**

- A *socket* is one endpoint of a two-way communication link between two programs running on the network.
- A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
- An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.
- TCP/IP creates the *socket address* as an identifier that is unique throughout all Internet networks. TCP/IP concatenates the Internet address of the local host interface with the port number to devise the Internet socket address.

Two types of (TCP/IP) sockets :

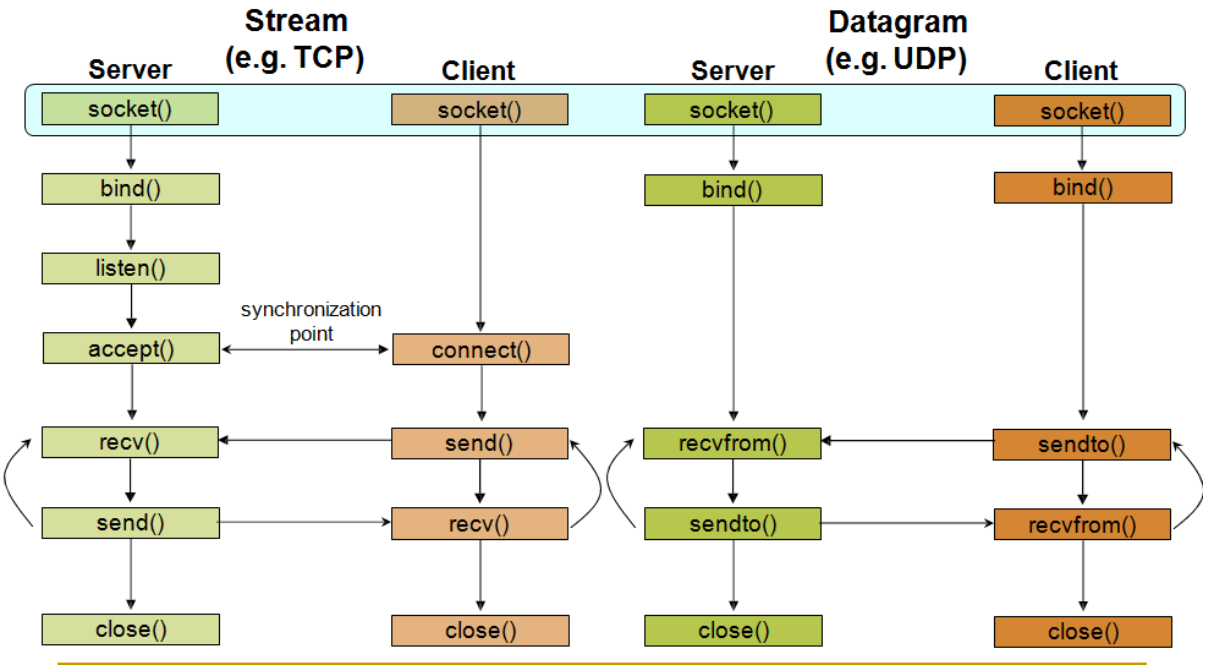
- **Stream sockets (e.g. uses TCP)** : provide reliable byte-stream service
- **Datagram sockets (e.g. uses UDP)** :provide best-effort datagram service , messages up to 65.500 bytes




**Sockets – Procedures:**

| Primitive | Meaning |
|---|---|
| Socket | Create a new communication endpoint |
| Bind | Attach a local address to a socket |
| Listen | Announce willingness to accept connections |
| Accept | Block caller until a connection request arrives |
| Connect | Actively attempt to establish a connection |
| Send | Send some data over the connection |
| Receive | Receive some data over the connection |
| Close | Release the connection |

**Client - Server Communication – Unix:**




**Elementary Socket System Calls:**

**socket**

To do network I/O, the first thing a process must do is to call the **socket** system call, specifying the type of communication protocol desired.

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

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

The *family* is one of

```
        AF_UNIX             -- Unix internal protocols
        AF_INET             -- Internet protocols
        AF_NS               -- Xerox NS Protocols
        AF_IMPLINK     -- IMP link layer
```

The AF_ prefix stands for "address family."

The socket *type* is one of the following:

```
        SOCK_STREAM   stream socket
        SOCK_DGRAM    datagram socket
        SOCK_RAW      raw socket
        SOCK_SEQPACKET       sequenced packet socket
        SOCK_RDM      reliably delivered message socket (not
implemented yet)
```

The *protocol* argument to the socket system call is typically set to 0 for most user applications. The valid combinations are shown as follows.

| *family* | *type* | *protocol* | *Actual protocol* |
|---|---|---|---|
| AF_INET | SOCK_DGRAM | IPPROTO_UDP | UDP |
| AF_INET | SOCK_STREAM | IPPROTO_TCP | TCP |
| AF_INET | SOCK_RAW | IPPROTO_ICMP | ICMP |
| AF_INET | SOCK_RAW | IPPROTO_RAW | (raw) |

**bind**

The **bind** system call assigns a name to an unnamed socket.

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

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

- The first argument is the socket descriptor returned from **socket** system call.
- The second argument is a pointer to a protocol-specific address.
- Third argument is the size of this address.
- There are three uses of **bind**.

1. Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.
2. A client can register a specific address for itself.
3. 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 corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we sent the letter to.

**Connect**

A client process **connect**s a socket descriptor following the **socket** system call to establish a connection with a server.

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

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

- The *sockfd* is a socket descriptor that was returned by the **socket** system call.
- The second and third arguments are a pointer to a socket address, and its size, as described earlier.
- For most connection-oriented protocols (TCP, for example), the **connect** system call results in the actual establishment of a connection between the local system and the foreign system.

- The **connect** system call does not return until the connection is established, or an error is returned to the process.
- The client does not have to **bind** a local address before calling **connect**. The connection typically causes these four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*, *foreign-addr*, and *foreign-process*.
- In all the connection-oriented clients, we will let **connect** assign the local address.

**listen**

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

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

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

- It is usually executed after both the **socket** and **bind** system calls, and immediately before the **accept** system call.
- The *backlog* argument specifies how many connection requests can be queued by the system while it waits for the server to execute the **accept** system call.
- This argument is usually specified as 5, the maximum value currently allowed.

**accept**

After a connection-oriented server executes the **listen** system call described above, an actual connection from some client process is waited for by having the server execute the **accept** system call.

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

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

**accept** takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there are no connection requests pending, this call blocks the caller until one arrives.

The *peer* and *addrlen* arguments are used to return the address of the connected peer process (the client). *addrlen* is called a value-result argument: the caller sets its value before the system call, and the system call stores a result in the variable. For this system call the caller sets *addrlen* to the size of the **sockaddr** structure whose address is passed as the *peer* argument.

**send, sendto, recv and recvfrom**

These system calls are similar to the standard **read** and **write** system calls, but additional arguments are required.

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

int send(int sockfd, char *buff, int nbytes, int flags);
int sendto(int sockfd, char *buff, int nbytes, int flags, struct
sockaddr *to, int addrlen);
int recv(int sockfd, char *buff, int nbytes, int flags);
int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct
sockaddr *from, int *addrlen);
```

The first three arguments, *sockfd*, *buff*, and *nbytes*, to the four system calls are similar to the first three arguments for **read** and **write**. The *flags* argument can be safely set to zero ignoring the details for it.

The *to* argument for **sendto** specifies the protocol-specific address of where the data is to be sent. Since this address is protocol-specific, its length must be specified by *addrlen*. The **recvfrom** system call fills in the protocol-specific address of who sent the data into *from*. The length of this address is also returned to the caller in *addrlen*. Note that the final argument to **sendto**is an integer value, while the final argument to **recvfrom** is a pointer to an integer value.

**close**

The normal Unix **close** system call is also used to close a socket.

```
int close(int fd);
```

If the socket being closed is associated with a protocol that promises reliable delivery (e.g., TCP or SPP), the system must assure that any data within the kernel that still has to be transmitted or acknowledged, is sent. Normally, the system returns from the **close** immediately, but the kernel still tries to send any data already queued.

**Advanced Socket System Calls:**

1) *readv* and *writev* system calls:

- These two functions are similar to read and write , but readv and writev let us read into or write from one or more buffers with a single function call. These operations are called scatter read (since the input data is scattered into multiple application buffers) and gather write (since multiple buffers are gathered for a single output operation).

```
#include <sys/uio.h>
int readv(int fd ,  struct iovec  iov[] , int iovcount) ;
int writev(int fd ,  struct iovec  iov[] , int iovcount) ;
```

**T**hese two system calls use the following structure that is defined in `<syst/uio.h>`:

```
struct iovec{
 caddr_t iov_base;  /*strating address of buffer*/
 int     iov_len;  /*size of buffer in size*/
```

- The writev system call write the buffers specified by `iov[0], iov[1],` through `iov[iovcount-1].`
- The readv sytem call does the input equivalent. It always fills one buffer (as specified but the `iov_len` value) before proceeding to the next buffer in the `iov` array.
- Both system calls return the total number of bytes read and written.

2) **getpeername** - get the name of the peer socket

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

```
 int getpeername(int socket, struct sockaddr *address,
socklen_t *address_len);
```

- The *getpeername()* function retrieves the peer address of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

- If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

- If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

3) **getsockname** - get the socket name

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

- `int getsockname(int socket, struct sockaddr *address, socklen_t *address_len);`

- The *getsockname()* function retrieves the locally-bound name of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

- If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

- If the socket has not been bound to a local name, the value stored in the object pointed to by *address* is unspecified.

4) **getsockopt and setsockopt** allow socket options values to be queried and set, respectively.

```
int getsockopt (sockid, level, optName, optVal,  optLen);
```

- **sockid**: integer, socket descriptor
- **level**: integer, the layers of the protocol stack (socket, TCP, IP)
- **optName**: integer, option
- **optVal**: pointer to a buffer; upon return it contains the value of the specified option
- **optLen**: integer, in-out parameter it returns -1 if an error occured

```
int setsockopt (sockid, level, optName, optVal,  optLen);
```

- **optLen** is now only an input parameter

| *optName* | Type | Values | Description |
|---|---|---|---|
| **SOL_SOCKET Level** | | | |
| SO_BROADCAST | **int** | 0,1 | Broadcast allowed |
| SO_KEEPALIVE | **int** | 0,1 | Keepalive messages enabled (if implemented by the protocol) |
| SO_LINGER | **linger**{} | time | Time to delay `close()` return waiting for confirmation (see Section 6.4.2) |
| SO_RCVBUF | **int** | bytes | Bytes in the socket receive buffer (see code on page 44 and Section 6.1) |
| SO_RCVLOWAT | **int** | bytes | Minimum number of available bytes that will cause `recv()` to return |
| SO_REUSEADDR | **int** | 0,1 | Binding allowed (under certain conditions) to an address or port already in use (see Section 6.4 and 6.5) |
| SO_SNDLOWAT | **int** | bytes | Minimum bytes to send a packet |
| SO_SNDBUF | **int** | bytes | Bytes in the socket send buffer (see Section 6.1) |
| **IPPROTO_TCP Level** | | | |
| TCP_MAX | **int** | seconds | Seconds between keepalive messages. |
| TCP_NODELAY | **int** | 0,1 | Disallow delay for data merging (Nagle's algorithm) |
| **IPPROTO_IP Level** | | | |
| IP_TTL | **int** | 0–255 | Time-to-live for unicast IP packets |
| IP_MULTICAST_TTL | **unsigned char** | 0–255 | Time-to-live for multicast IP packets (see `MulticastSender.c` on page 81) |
| IP_MULTICAST_LOOP | **int** | 0,1 | Enables multicast socket to receive packets it sent |
| IP_ADD_MEMBERSHIP | **ip_mreq**{} | group address | Enables reception of packets addressed to the specified multicast group (see `MulticastReceiver.c` on page 83)—set only |
| IP_DROP_MEMBERSHIP | **ip_mreq**{} | group address | Disables reception of packets addressed to the specified multicast group—set only |

5) **`shutdown` system call:** this system call terminates the network connect and provide more control on the full-duplex connection.

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

i. if `howto` argument is 0: no more data can be received on the socket.

ii. if `howto` argument is 1: no more output to be allowed on the socket.

iii. if `howto` argument is 2: both send and receive to be disallowed

- shutdown allows either directions to be closed independent of the other direction.

## 6.) **select ():**

When a server (or client) has multiple connections, it can be difficult to guess which clients( or servers) have written data on a socket. One approach, called **polling**, is to use nonblocking recv( ) and loop through all the connections. This is inefficient. Another approach, using **fork( ),** is to fork a child process for each connections. This is also inefficient. A better option is to wait on all the connections simultaneously. This can be done using select( ) function.

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const strut timeval *timeout);
-- returns # of ready descriptors, 0 if timeout occurs, -1 on error.
```

*maxfdp1* – the maximum descriptor to test +1, the possible number of descriptors to test, ⩽256.
*readset* – used to check which connections have data read.
*writeset* – used to check which connections have space for more output.
*exceptset* – used to check which connections have exceptions, such as OOB data.
*timeout* – specifies how long to block waiting for ready connction
There are three options;
= 0 means the call is nonblocking. Used for polling connections.
> 0 means the call times out after this amount of time if there are no ready connection during this time.
NULL means the call blocks until a connection is ready for I/O.

The format of the timeval structure is:

```
struct timeval {
  long tv_sec;     /*seconds*/
  long tv_usec;    /*microseconds*/
};
```

select( ) is used to determine which socket are ready for reading, writing, or exception handling. Use NULL for any fd_set that doesn't need to be checked.

The fd_set detatype typically uses one bit per socket fd. The appropriate method for using fd_set is to zero out all the bits and then set each one that is to be tested. The select( ) call modifies the *readset*, *writeset*, and *exceptset* variables by clearing the bits that are not ready for I/O. The user then tests each bit to see which are set and processes the corresponding sockets.

Operations on fd_sets should be performed using the following macros:

```
void FD_ZERO(fd_set *fdset);           /* clear all bits in fdset**/
void FD_SET(int fd, fd_set *dset);     /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);    /* clear off the bits in fdset*/
int   FD_ISSET(int fd, fd_set *fdset);  /* test the bit for fd in fdset */
```

7.)

# 14.5 recvmsg and sendmsg Functions

These two functions are the most general of all the I/O functions. Indeed, we could replace all calls to read, readv, recv, and recvfrom with calls to recvmsg. Similarly all calls to the various output functions could be replaced with calls to sendmsg.

```
#include <sys/socket.h>
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
```

Both return: number of bytes read or written if OK, −1 on error

Both functions package most arguments into a msghdr structure.

```
struct msghdr {
  void         *msg_name;        /* protocol address */
  socklen_t     msg_namelen;     /* size of protocol address */
  struct iovec *msg_iov;         /* scatter/gather array */
  int           msg_iovlen;      /* # elements in msg_iov */
  void         *msg_control;     /* ancillary data (cmsghdr struct) */
  socklen_t     msg_controllen;  /* length of ancillary data */
  int           msg_flags;       /* flags returned by recvmsg() */
};
```

The msghdr structure that we show is the one specified in POSIX. Some systems still use an older msghdr structure that originated with 4.2BSD. This older structure does not have the msg_flags member, and the msg_control and msg_controllen members are named msg_accrights and msg_accrightslen. The newer form of the msghdr structure is often available using conditional compilation flags. The only form of ancillary data supported by the older structure is the passing of file descriptors (called access rights).

**RESERVED PORTS:**

There are two ways for a process to have an internet port assigned to a socket.

a) The process can request for a specific port. This is typically for servers that need to assign a well-known port to a socket.

b) The process can let the system automatically assign a port.

4.3BSD provides a library function that assigns a reserved TCP stream socket to its caller:

```
int rresvport(int *aport);
```

| | **Internet** | **XNS** |
|---|---|---|
| reserved ports | **1-1023** | **1-2999** |
| ports automatically assigned bt system | **1024-5000** | **3000-65535** |
| ports assigned by `rresvport` | **512-1023** | **-** |

# Asynchronous I/O:

### Introduction

When the TCP client is handling two inputs at the same time: standard input and a TCP socket, we encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later).

We want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called **I/O multiplexing** and is provided by the select and poll functions, as well as a newer POSIX variation of the former, called pselect.

I/O multiplexing is typically used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket)
- When a client to handle multiple sockets at the same time (this is possible, but rare)
- If a TCP server handles both a listening socket and its connected sockets
- If a server handles both TCP and UDP
- If a server handles multiple services and perhaps multiple protocols

I/O multiplexing is not limited to network programming. Many nontrivial applications find a need for these techniques.

### I/O Models

We first examine the basic differences in the five I/O models that are available to us under Unix:

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- signal driven I/O (SIGIO)
- asynchronous I/O (the POSIX aio_ functions)

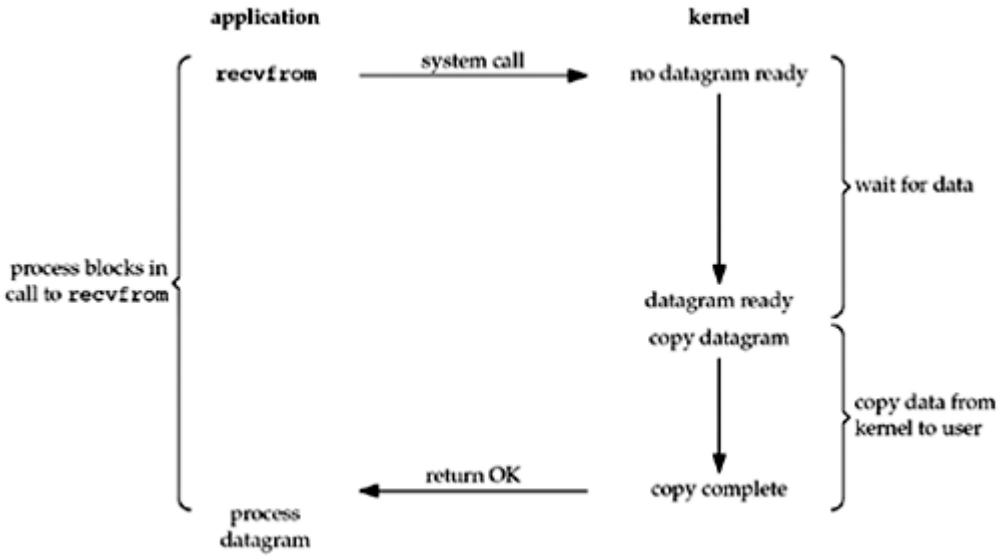
There are normally two distinct phases for an input operation:

1. Waiting for the data to be ready. This involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel.

2. Copying the data from the kernel to the process. This means copying the (ready) data from the kernel's buffer into our application buffer

**Blocking I/O Model**

The most prevalent model for I/O is the blocking I/O model (which we have used for all our examples in the previous sections). By default, all sockets are blocking. The scenario is shown in the figure below:



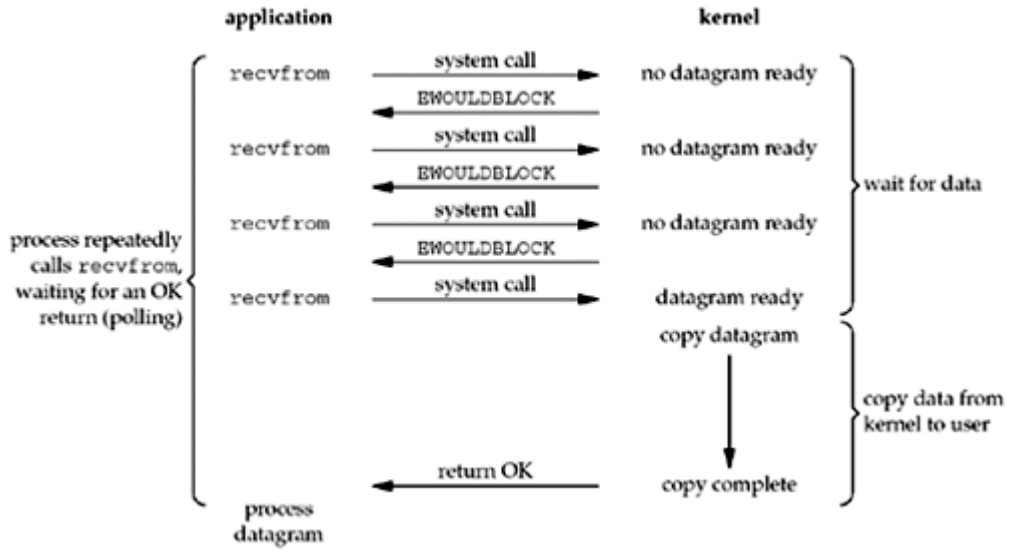

We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

We also refer to recvfrom as a system call to differentiate between our application and the kernel, regardless of how recvfrom is implemented (system call on BSD and function that invokes getmsg system call on System V). There is normally a switch from running in the application to running in the kernel, followed at some time later by a return to the application.

In the figure above, the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal, as we described in Section 5.9. We say that the process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram.

**Nonblocking I/O Model**

When a socket is set to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead". The figure is below:

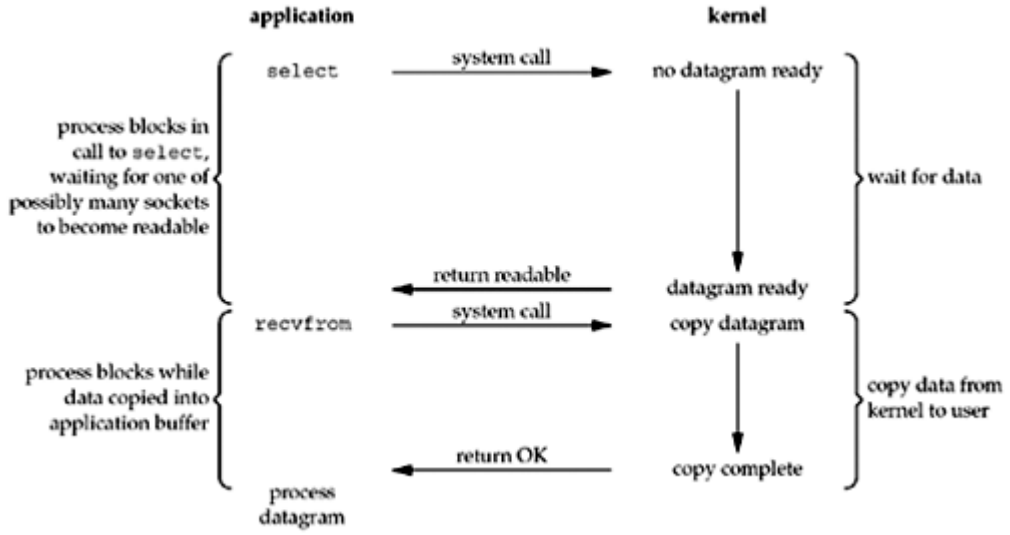



- For the first three recvfrom, there is no data to return and the kernel immediately returns an error of EWOULDBLOCK.
- For the fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called **polling**. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

**I/O Multiplexing Model**

With **I/O multiplexing**, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. The figure is a summary of the I/O multiplexing model:




We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

The select function allows the process to instruct the kernel to either:

- Wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs, or

- When a specified amount of time has passed.

This means that we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using select.

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        const struct timeval *timeout);

/* Returns: positive count of ready descriptors, 0 on timeout, –1 on error */
```

***The* timeout *argument*** *

The *timeout* argument tells the kernel how long to wait for one of the specified descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

```
struct timeval  {
  long   tv_sec;         /* seconds */
  long   tv_usec;        /* microseconds */
};
```

There are three possibilities for the *timeout*:

1. **Wait forever** (*timeout* is specified as a null pointer). Return only when one of the specified descriptors is ready for I/O.
2. **Wait up to a fixed amount of time** (*timeout* points to a timeval structure). Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure.
3. **Do not wait at all** (*timeout* points to a timeval structure and the timer value is 0, i.e. the number of seconds and microseconds specified by the structure are 0). Return immediately after checking the descriptors. This is called **polling**.

**Comparing to the blocking I/O model** *

Comparing Figure 6.3 to Figure 6.1:

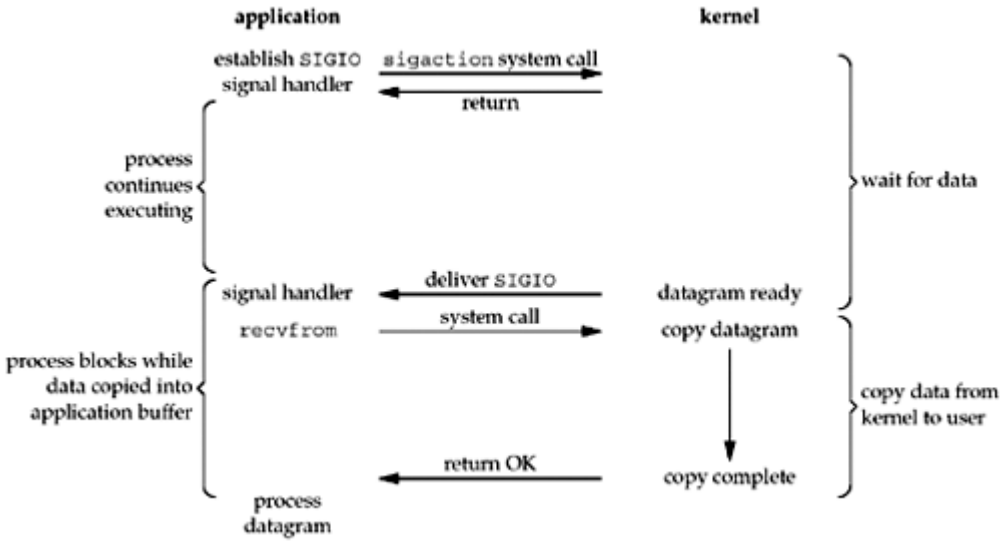
- Disadvantage: using select requires two system calls (select and recvfrom) instead of one
- Advantage: we can wait for more than one descriptor to be ready (see the select function later in this chapter)

**Multithreading with blocking I/O** *

Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.

**Signal-Driven I/O Model**

The **signal-driven I/O model** uses signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready. The figure is below:




- We first enable the socket for signal-driven I/O (Section 25.2) and install a signal handler using the sigaction system call. The return from this system call is immediate and our process continues; it is not blocked.
- When the datagram is ready to be read, the SIGIO signal is generated for our process. We can either:
    - read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be processed (Section 25.3)
    - notify the main loop and let it read the datagram.

The advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

## Socket-related Signals:

**1) SIGIO:** indicates that a socket is ready for asynchronous I/O as we have discussed. need to specify process ID or process group ID to receive the signal.

Need to enable asynchronous I/O.

**2) SIGURG:** indicates urgent data is coming due to 1)OOB data or 2) control status information.

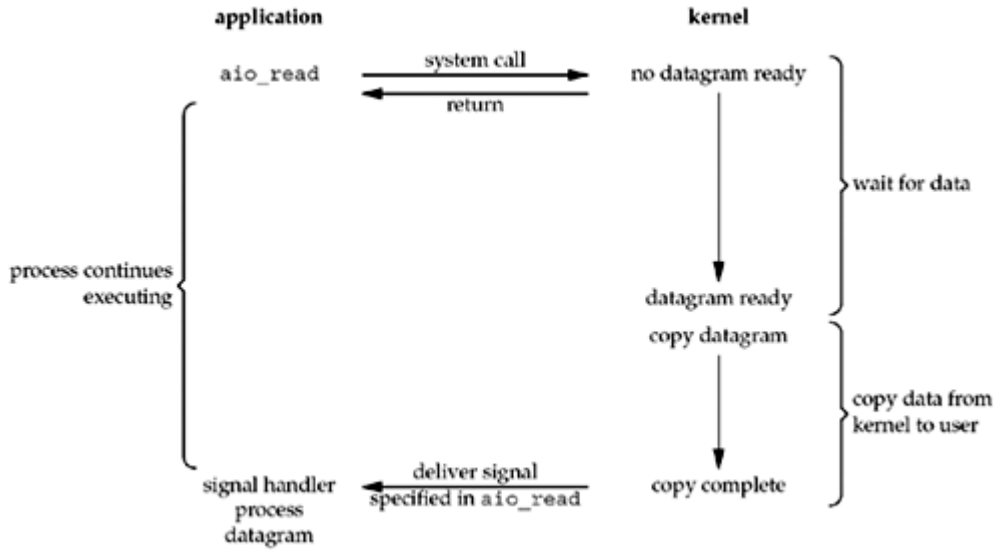
- need to specify process group ID to receive the signal,e.g., fcntl(sd,F_SETOWN, -getpgid( )). Use flag=MSG_URG to send and receive the OOB data.
- If O_OOBINLINE is set, we must use STOCATMARK ioctl to read OOB data. setsockopt(sd, SOL_SOCKET, SO_OOBINLINE, &seton, sizeof(seton)); /*let seton=1*/
- If ((n=ioctl(sd,STOCATMARK, &start)>0) read(sd, buf, n); /*OOB data is in buf with n bytes.*/

**3) SIGPIPE:** indicates socket, pipe, or FIFO can never be written to. Sent only to the associated process,

- **Asynchronous I/O Model**

**Asynchronous I/O** is defined by the POSIX specification, and various differences in the *real-time* functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled.

These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete. See the figure below for example:




Process can wait for the kernel to send signal SIGIO when a specified descriptor is ready for I/O. 3 things to do:

1) Establish a handler for SIGIO by calling signal(SIGIO, ???);

2) Set PID or PGID for the descriptor to receive SIGIO by calling fcntl(fd, F_SETOWN, getpid());

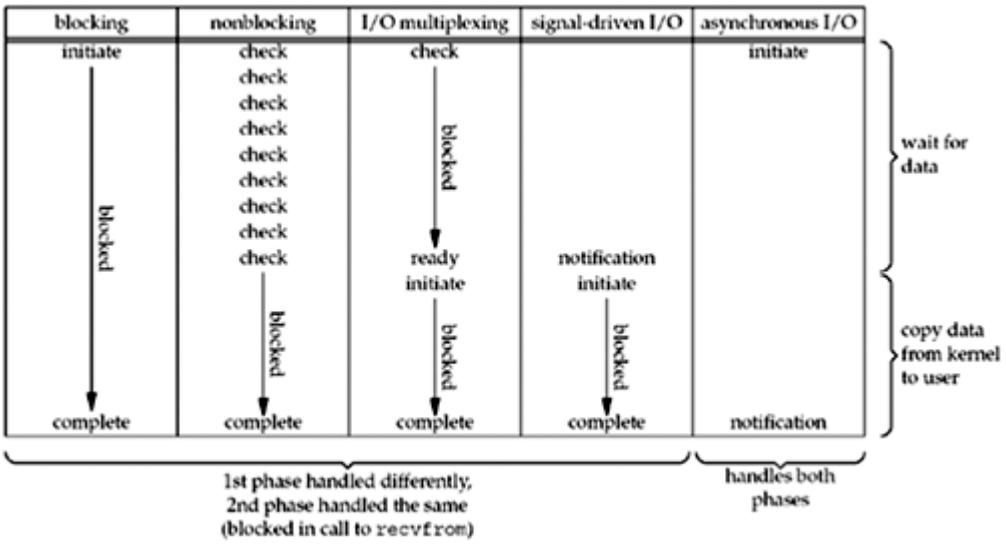
3) Enable asynchronous I/O by calling fcntl(fd, F_SETFL,FASYNC).

```
/*      Copy standard input to standard output.       */

#define BUFFSIZE  4096

main()

{    int        n;

     char       buff[BUFFSIZE];

}    while ( (n = read(0, buff, BUFFSIZE)) > 0) write(1, buff, n);
```

```
/*    Copy standard input to standard output, using asynchronous I/O.       */
#include<signal.h>
#include<fcntl.h>
#define BUFFSIZE  4096
int   sigflag;
main()
{     int         n;
      char        buff[BUFFSIZE];
      int         sigio_func();
      signal(SIGIO, sigio_func);                  /*    Step 1: set up signal handler*/
      fcntl(0, F_SETOWN, getpid();                /*    Step 2: set descriptor's process ID*/
      fcntl(0, F_SETFL, FASYNC) ;                 /*    Step 3: Enable Asynchronous I/O*/
      for ( ; ; ) {
            sigblock(sigmask(SIGIO));             /*    block signal SIGIO to avoid race condition */
            while (sigflag == 0)  sigpause(0);    /* release signals when waiting for a signal.
                                                     Note the difference between pause() and sigpause(0)*/
            /* We're here if (sigflag != 0).  Also, we know that the SIGIO signal is currently blocked.*/
            if ( (n = read(0, buff, BUFFSIZE)) > 0) write(1, buff, n) ;        /* not a loop structure */
           else if (n == 0)   exit(0);                /* EOF */
            sigflag = 0;                              /* turn off our flag */
            sigsetmask(0);                            /* and reenable signals */
      }
}

int sigio_func( )
{   sigflag = 1;/* just set flag and return */
      /* the 4.3BSD signal facilities leave this handler enabled for any further SIGIO signals. */
}
```

**Comparison of the I/O Models**

The figure below is a comparison of the five different I/O models.




The main difference between the first four models is the first phase, as the second phase in the first four models is the same: the process is blocked in a call to recvfrom while the data is copied from the kernel to the caller's buffer. Asynchronous I/O, however, handles both phases and is different from the first four.

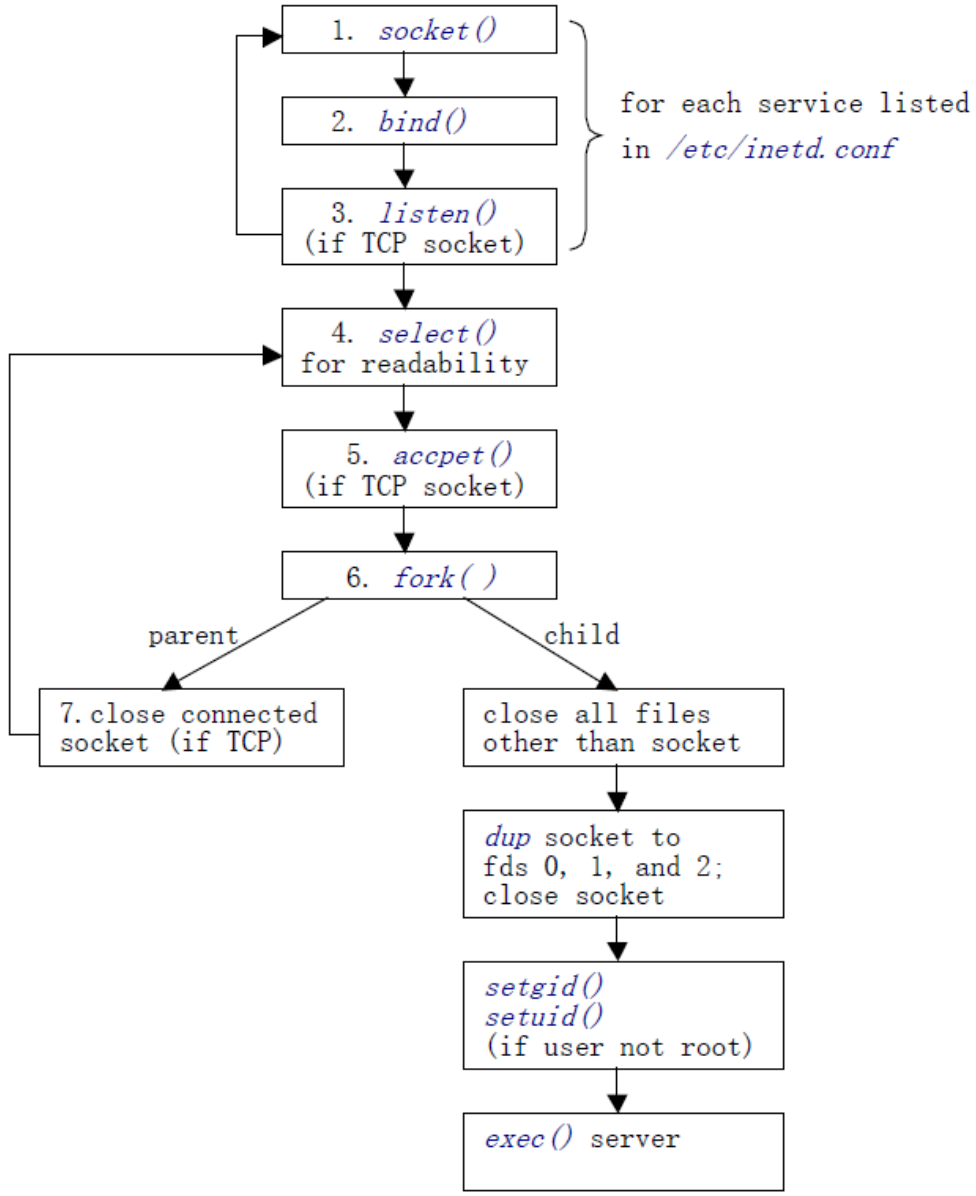
## Internet Super server:

As in most other Unix systems, networking services are implemented as daemons in Linux. Each networking daemon responds to requests on a particular port. The Telnet service, for example, operates on port 23. For networking services to function properly, some process must be alive and listening on each corresponding port. Instead of starting all the networking daemons so that each would listen to its own port, however, most systems make use of an internet "super-server." This super-server is a special daemon that listens to the ports of all the enabled networking services. When a request comes in from a particular port, the corresponding networking daemon is started, and the request is passed on to it for service.

There are two main benefits to this scheme. First, only the minimal set of needed daemons is active at all times, and therefore, no system resources are wasted. Second, there is a centralized mechanism for managing and monitoring network services.

Though many networking services can be managed by the internet super-server, some services such as an HTTP server or an SNMP agent are almost always set up to have direct control of their ports for reasons of scalability and reliability. In fact, the daemons providing such services will not require an internet super-server to operate properly. For each networking service discussed in the following sections, we will consider whether the service depends on the super-server.

There are two main internet super-servers available for Linux, inetd and xinetd. Though inetd used to be the standard super-server for most Linux distributions, it is gradually being replaced by xinetd, which contains more features. But because inetd contains fewer features than xinetd, it is also smaller and may be better for an embedded Linux system.




Steps performed by inetd

**Flow chart of inetd (version2: section 12.5 or version1:section 6.16)**

**1)** read /etc/inetd.conf to create one socket for each service in the file.

**2)** read /etc/services to bind well-known port numbers to each service.

**3)** Listen() only for TCP.

**4)** Select() can be used for connect requests that arrives at the socket for reading.

**5)** If it is TCP request, call accept().

**6)** Fork a child process to handle the request

6.1) close all files except socket

6.2) dup2(sd,0), dup2(sd,1), and dup2(sd, 2).

6.3) login program: a superuser can become any user. Must in the order of setgid() first and then setuid().

6.4) exec() to execute server_program accordingly.

**7)** Parent goes up to accept next request without wait.

## Out-of-Band Data:

Similarly, when a user at a terminal presses the attention key, it needs to be operated on as soon as possible. The Unix interrupt key (typically the Delete key or Control-C) is one example of this, as are the terminal flow control characters (typically Control-S and Control-Q). This type of information is termed *out-of-band data* or *expedited data*. With out-of-band data we want the byte-stream service layer on the sending side to send this data before any other data that it has buffered. Similarly we want the receiving end to pass this data to its user process ahead of any data that it might have buffered. Some method is also desired for the receiving byte-stream service layer to notify the user process asynchronously that out-of-band data has arrived. As you might guess, a Unix signal can be user for this notification. The term *out of band* is used because it appears to the user processes that a separate communication channel is being used for this data, in addition to the normal data channel (band). This is shown in Figure 4.18.

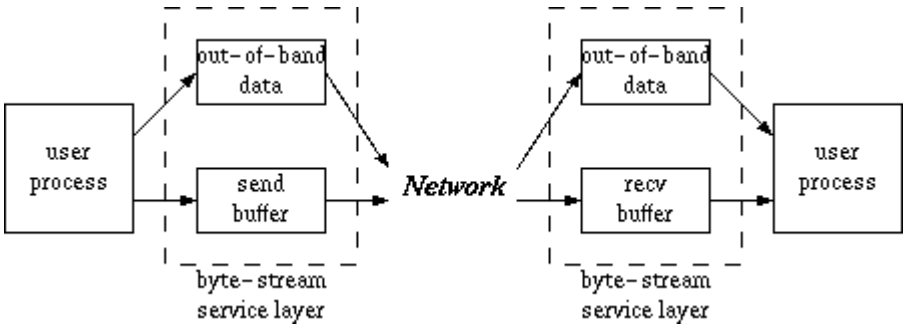



**Figure 4.18** In-band data and out-of-band data.

To send an out-of-band message, the MSG_OOB flag must be specified for the send, sendto, or sendmsg system calls. [...]

- **TCP Out-of-Band Data**

All this gets more complicated with TCP becuase it can send the notification that out-of-band data exists (termed "urgent data" by TCP) *before* it sends the out-of-band data. In this case, if the process executes one of the three receive system calls, an error of EWOULDBLOCK is returned if the out-of-band data has not arrived. As with the SPP example above [which I have not included on this page], an error of EINVAL is returned by these three system calls if the MSG_OOB flag is specified when there isn't any out-of-band data.

Still another option is provided with the TCP implementation, to allow for multiple bytes of out-of-band data. By default, only a single byte of out-of-band data is provided, and unlike SPP, this byte of data is not stored in the normal data stream. This data byte can only be read by specifying the MSG_OOB flag to the receive system calls. But if we set the SO_OOBINLINE option for the socket, using the setsockopt system call described in Section 6.11, the out-of-band data is left in the normal data stream and is read without specifying the MSG_OOB flag. If this socket option is enabled, we must use the SIOCATMARK ioctl to determine where in the data stream the out-of-band data occurs. In this case, if multiple occurrences of out-of-band data are received, all the data is left in-line (i.e, none of the data can be lost) but the mark returned by the SIOCATMARK ioctl corresponds to the final sequence of out-of-band data that was received. If the out-of-band data is not received in-line, it is possible to lose some intermediate out-of-band data when the out-of-band data arrives faster that it is processed by the user.

**Domain Name Servive(DNS):**

- **`struct hostent *gethostbyname(const char *name);`**
  - returns a structure of type `hostent` for the given host name
  - `name` is a hostname, or an IPv4 address in standard dot notation

  e.g. **`gethostbyname("www.csd.uoc.gr");`**

- **`struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);`**
  - returns a structure of type hostent for the given host address addr of length len and address type `type`

```
struct hostent {
   char  *h_name;        /* official name of host */
   char **h_aliases;     /* alias list (strings) */
   int    h_addrtype;    /* host address type (AF_INET) */
   int    h_length;      /* length of address */
   char **h_addr_list;   /* list of addresses (binary in network byte order) */
}

#define h_addr h_addr_list[0]  /* for backward compatibility */
```

- **`struct servent *getservbyname(const char *name, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using protocol `proto`.
  - if `proto` is NULL, any protocol will be matched.

  e.g. **`getservbyname("echo", "tcp");`**

- **`struct servent *getservbyport(int port, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using port `port`

```
struct servent {
   char  *s_name;        /* official service name */
   char **s_aliases;     /* list of alternate names (strings)*/
   int    s_port;        /* service port number */
   char  *s_proto;       /* protocol to use ("tcp" or "udp")*/
}
```