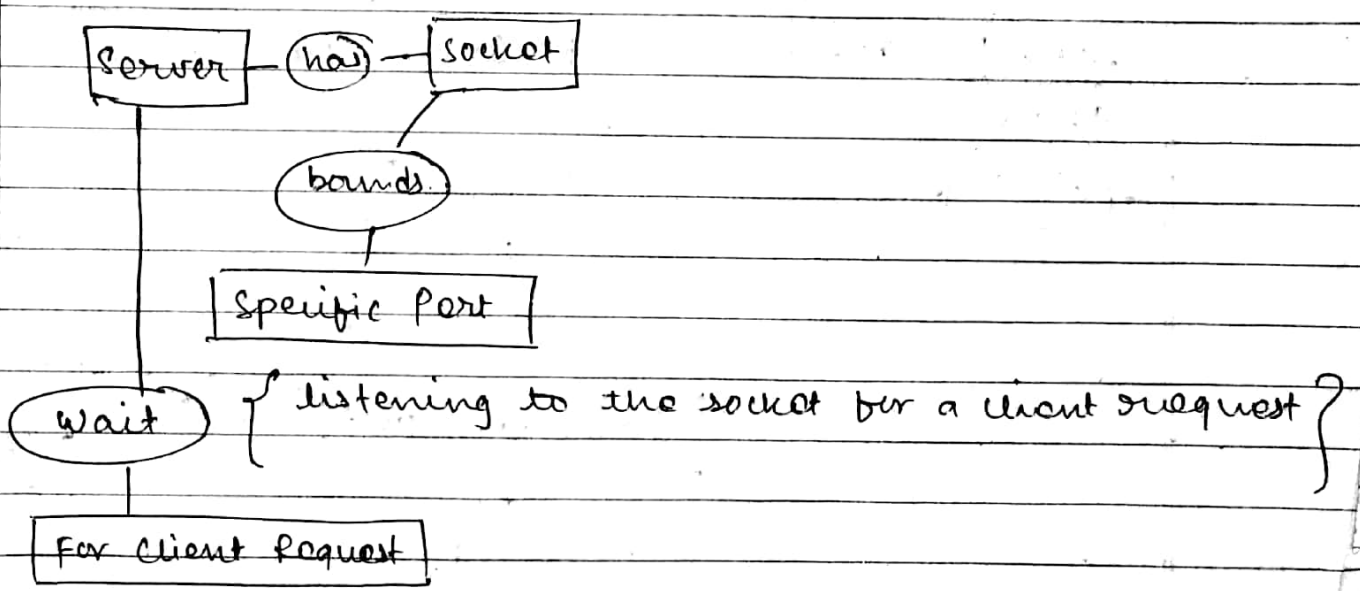
2 Unix Programming: 1

PUSPANJALI
Date / /
Page

Socket:

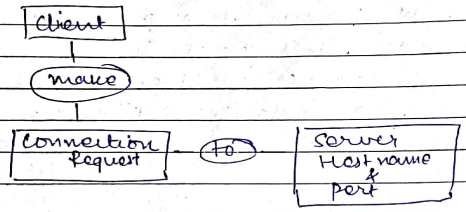
A socket is one endpoint of a two-way communication link between two program running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. ~~It~~ Endpoint is a combination of an IP address & port no.

At server:



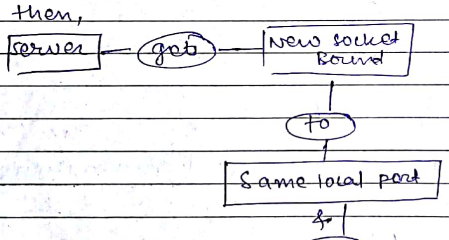
Teacher's Signature

At client



Is everything goes well?
- Yes! Server accept connection.

then,



So that it needs a new socket so that it can continue to listen to original socket for connection requests while tending to, the needs of connected client.

Teacher's Signature

3 Types of Port Number

1. Well-known ports: 0 – 1023
   - control & assign by IANA (Internet Assigned Number Authority).
2. Registered ports: 1024 – 49151
   - not controlled by IANA
   - But IANA registers & lists the uses of those ports as a convenience to the community
3. Dynamic or private port: 49152 – 65535
   - also called ~~ex~~ phemeral ports: i.e. short lived ports
   - assign automatically by TCP or UDP to the clients

Socket Address Structure:

sockaddr – [suffix for each protocol suite]

IPV4:

named as:

sockaddr_in

Posix definition:

```
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];
};
```

Teacher's Signature

```
struct in_addr {
    in_addr_t s_addr;
};
```

* Socket Function that pass socket address structure.

(1) From process to the kernel
- bind(), connect(), sendto() sendmsg()
- They copies socket address structure (SAS) from process & explicitly sets its sin_len member to the size of the structure that was passed as an argument to these 4 function

(2) From kernel to process:
- accept(), recvfrom(), recvmsg(), getpeername(), getsockname()
- set sin_len member before returning to the process

Generic Socket Address Structure

```
struct sockaddr {
    uint8_t     sa_len;
    sa_family_t sa_family;
    char        sa_data[14];
};
```

Eg:

```
struct sockaddr_in serv;
/* fill in serv */
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

IPV6:

```
struct in6_addr {
    uint8_t s6_addr[16];
};
#define SIN6_LEN

struct sockaddr_in6 {
    uint8_t     sin6_len;
    sa_family_t sin6_family;
    in_port_t   sin6_port;
    uint32_t    sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t    sin6_scope_id;
};
```

New Generic SAS:

```
struct sockaddr_storage {
    uint8_t     ss_len;
    sa_family_t ss_family;
    //
};
```

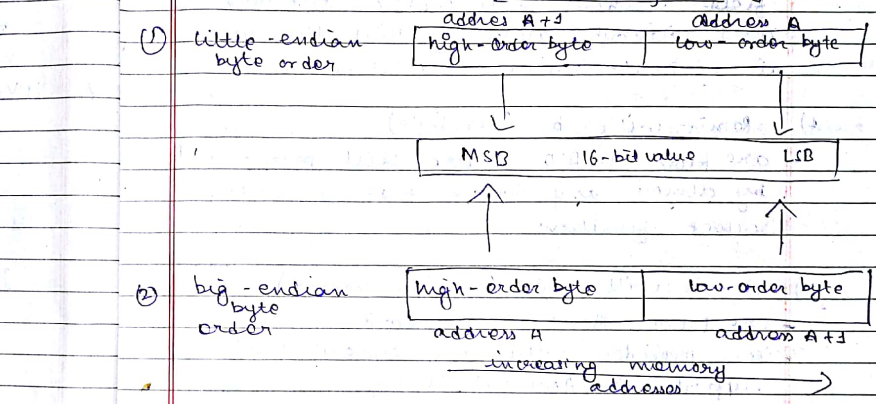
Teacher's Signature

Values Result arguments:

Teacher's Signature

Byte Ordering Functions

* 2 ways to store 2 bytes in memory



* 4 functions for byte ordering

```
#include <netinet/in.h>

(1) uint16_t htons (uint16_t host16bitvalue);
(2) uint32_t htonl (uint32_t host32bitvalue);
(3) uint16_t ntohs (uint16_t net16bitvalue);
(4) uint32_t ntohl (uint32_t net32bitvalue);
```


Teacher's Signature

Byte Manipulation functions

* 2 Groups that operate on multibyte fields without interpreting the data and without assuming that the data is a null-terminated C string

1) Begin with b (for byte)
- are from 4.2BSD & are still provided by almost any system that supports socket functions

2) begin with mem (~~me~~ for memory)
- are from ANSI C standard
- are provided with any system that supports an ANSI C library.

Functions

a) bzero():
- sets the specified no. of bytes to 0 in the destination
- used to initialize a socket address structure to 0.

```
void bzero(void *dest, size_t nbytes);
```

b) bcopy():
- moves the specified no. of bytes from the source to the destination

```
void bcopy(const void *src, void *dest, size_t nbytes);
```

All uses <string.h>

c) bcmp:
- compares 2 arbitrary byte strings.
  0 if 2 value are identical
  non-zero if not

ex.
```
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

d) memset():
- sets the specified no. of bytes to the value (C) in the destination

```
void *memset(void *dest, int c, size_t len);
```

e) memcpy()
- similar to bcopy but the order of 2 pointer arguments is swapped.

```
void *memcpy(void *dest, const void *src, size_t nbytes)
```

f) memcmp()
- compares 2 arbitrary byte strings and returns
  0 if they are identical
  >0 ptr1 > ptr2
  <0 ptr1 < ptr2

Address conversion functions

- They convert internet addresses between ASCII strings and network byte ordered binary values.

2 Groups:

1) Convert an IPV4 address from a doted-decimal string (eg: "192.168.1.1") to its 32 bit network byte ordered binary value.

They are: #include <arpa/inet.h>

a) inet_aton():

syntax: int inet_aton(const char *strptr, struct in_addr *addrptr);

- returns:
  - 1 if string is valid
  - 0 if error.

b) inet_addr():

- It is deprecated

syntax: in_addr_t inet_addr(const char *strptr);

Returns:

32-bit binary network byte ordered IPV4 address;

INADDR_NONE if error

c) inet_ntoa():

- Converts a 32-bit binary network byte ordered IPV4 address into its corresponding dotted-decimal string

Returns:

pointer to dotted-decimal string

2) For both IPV4 and IPV6 addresses.

- p = presentation
- n = numeric

<arpa/inet.h>

a) inet_pton():

either AF_INET - IPV4 or AF_INET6 - IPV6

int inet_pton(int family, const char *strptr, void *addrptr);

- convert the string pointed to by strptr, storing the binary result through the pointer addrptr.

Returns:

- 1 if OK
- -1 on error
- 0 if input not a valid presentation format

b) inet_ntop():

- does reverse conversion from numeric (addrptr) to presentation strptr.

Teacher's Signature

```
const char *inet_ntop(int family,
        const void *addrptr, char *strptr,
        size_t len);
                ↳ size of destination
```

Returns:
pointer to result if OK
null on error

sock_ntop
Pg: 86

readn, writen and readline functions
Pg: 88

Socket Function

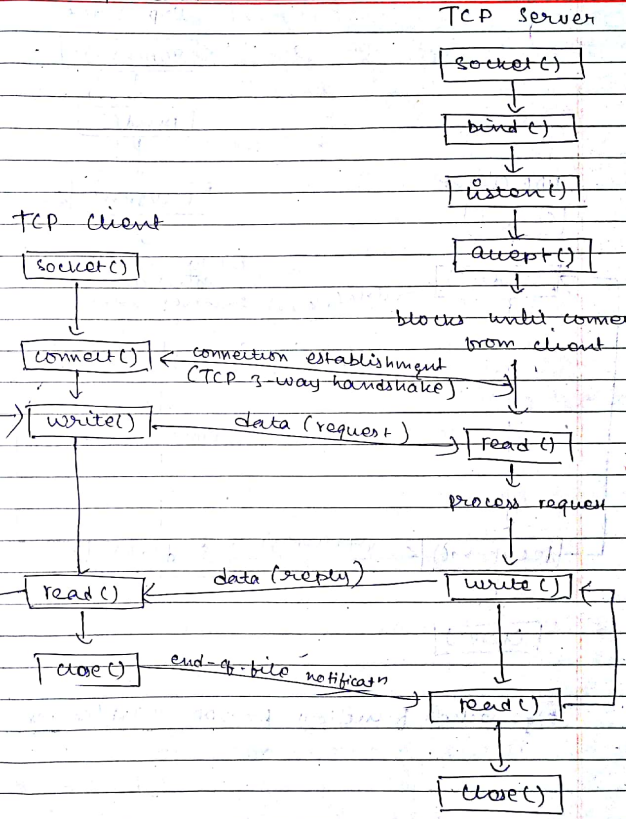


Fig: Socket functions for elementary TCP client/server

Teacher's Signature

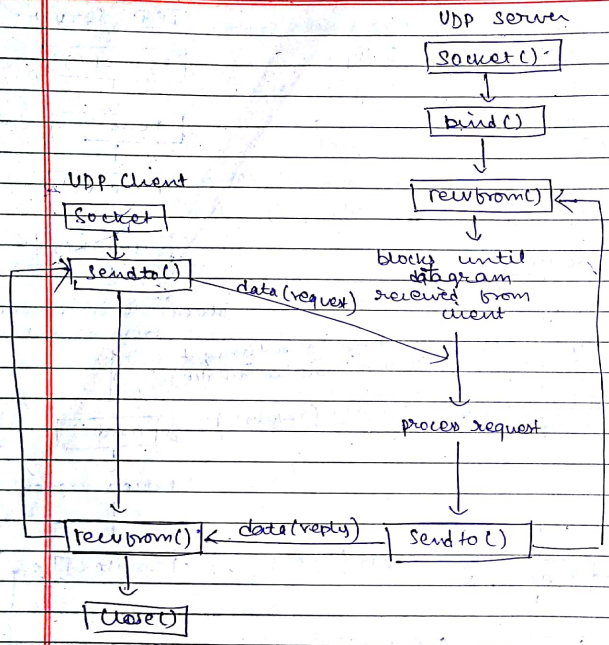


Fig: Socket functions for UDP client/server

1) socket(): <sys/socket.h>

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

Returns:

non-negative descriptor = OK

-1 = error

2) connect(): <sys/socket.h>

- used by TCP client to establish a connection with TCP server.

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

sockfd => socket descriptor returned by socket().

* several different error returns possible:

1) ETIMEDOUT:
- if client TCP receives no response to its SYN segment

2) ECONNREFUSED:
- hard error
- if the server's response to the client's SYN is a reset (RST)

3) EHOSTUNREACH or ENETUNREACH:

- Soft error
- If the client's SYN elicits an ICMP from some intermediate router

3) bind() : <sys/socket.h>

- It assigns a local protocol address to a socket.

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

Returns :
0 if OK
-1 on error

4) listen() :
- called only by a TCP server
- perform 2 actions :

(1) It converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket

(2) The second argument specifies the max. no. of connections the kernel should queue for this socket.

Teacher's Signature _____

int listen (int sockfd, int backlog);

Returns:
0 if OK
-1 on error

5) accept() ; <sys/socket.h>

- called by a TCP server to return the next completed connection from the front of the completed connection queue

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

Returns :
non-negative descriptor = OK
-1 = error

6) close() : <unistd.h>

- used to close a socket & terminate a TCP connection.

int close (int sockfd);

returns:
0 if OK
-1 on error

Teacher's Signature _____

7) getsockname and getpeername()

↓

return local protocol address | associated with a socket

foreign protocol address associated with a socket

include <sys/socket.h>

- int getsockname (int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
- int getpeername (int sockfd, struct sockaddr * peeraddr, socklen_t * addrlen);

Return:

0 = OK

-1 = error

UDP

8) recvfrom and sendto():

- similar to standard read and write function

include <sys/socket.h>

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

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

Returns:

no. of bytes read or written if OK

-1 on error

Fork and exec functions:

- fork() is used to create a new process.

include <unistd.h>

pid_t fork (void);

Returns:

0 in child

process ID of child in parent

-1 on error

- fork() is called once but it returns twice.

Return

① process ID of newly created child process to its calling parent process.

② It return 0 in child.

why?

Because a child has only one parent and can always obtain parent's process id by getppid.

* Typical 2 uses of fork():

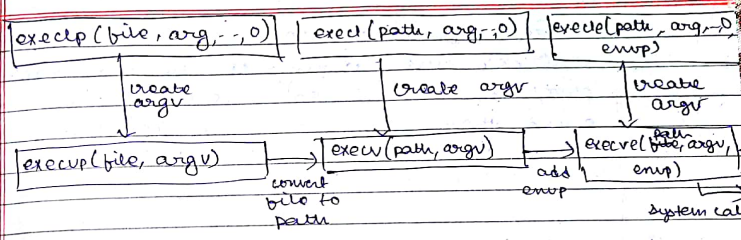
1) A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is common for many network servers.

2) A process wants to execute another program. Since the only way to create a new process is by calling fork()
   a) The process first calls fork() to make a copy of itself
   b) and then one of the copies (child process typically) calls exec() to replace itself with the new program.

* exec() :
- It replaces the current process image with the new program file, which normally starts at the "main" function.
- process ID does not change.
- process that calls exec() → calling process
- the newly executed program → new program.

+ Six type of exec();

envp → environment list



#include <unistd.h>

- int execl(const char *pathname, const char *arg0, ...);
- int execv(const char *pathname, char *const argv[]);
- int execle(const char *pathname, const char *arg0, ... char *const envp[]);
- int execve(const char *pathname, char *const arg[], char *const envp[]);
- int execlp(const char *filename, const char *arg0, ...);
- int execvp(const char *filename, char *const argv[]);

Return:
-1 on error
no return on success

Their differences are:

(1) whether the arguments to the new program are listed one by one or referenced through an array of pointers.

(2) whether the program file to be executed is specified by a "filename" or a "pathname"

(3) whether the environment of the calling process is passed to the new program or whether a new env is specified

Concurrent Servers:

* Reference count:
- every file or socket has it.
- It is maintained in the file table entry.

Teacher's Signature

- This is a count of the number of descriptors that are currently open that refer to this file or socket.

- Concurrent server helps to handle multiple clients at the same time.
- When a connection is established:
  a) accept returns
  b) server calls fork()
  c) child process services the client (on the connected socket) and the parent process waits for another connection (in the listening socket)
- To close:
  a) parent closes the connected socket since the child handles this new client
  b) The child closes the listening socket since the parent is listening anyway

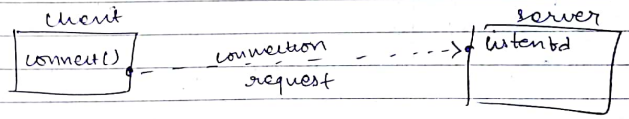


Fig: Status of client/server before call to accept returns.

Teacher's Signature

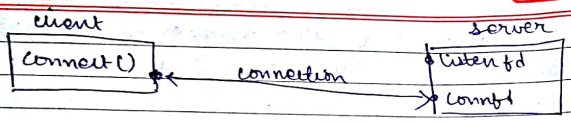


Fig: Status of client/server after return from accept.

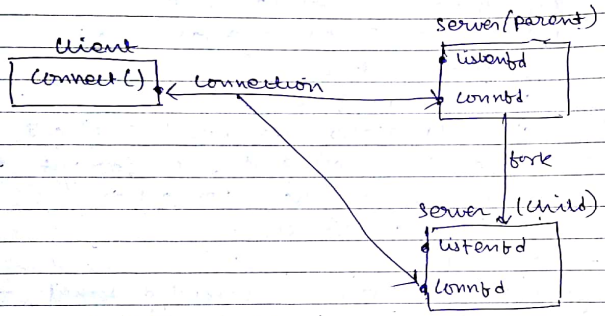


Fig: Status of client/server after fork returns.

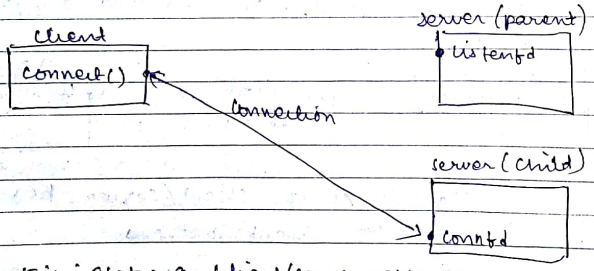


Fig: Status of client/server after parent & child close appropriate sockets

UNIX / INTERNET domain / socket

* Two types of sockets are

I/O models

There are 5 basic types of I/O models available in UNIX

(1) blocking I/O
(2) non-blocking I/O
(3) I/O multiplexing (select and poll)
(4) signal driven I/O (SIGIO)
(5) Asynchronous I/O (the POSIX "aio_" functions)

There are two distinct phases for an input operation:

1) Waiting for the data to be ready.
2) Copying the data from the kernel to the process. (kernel's buffer → application's buffer)

Blocking I/O model

Blocking I/O Model:

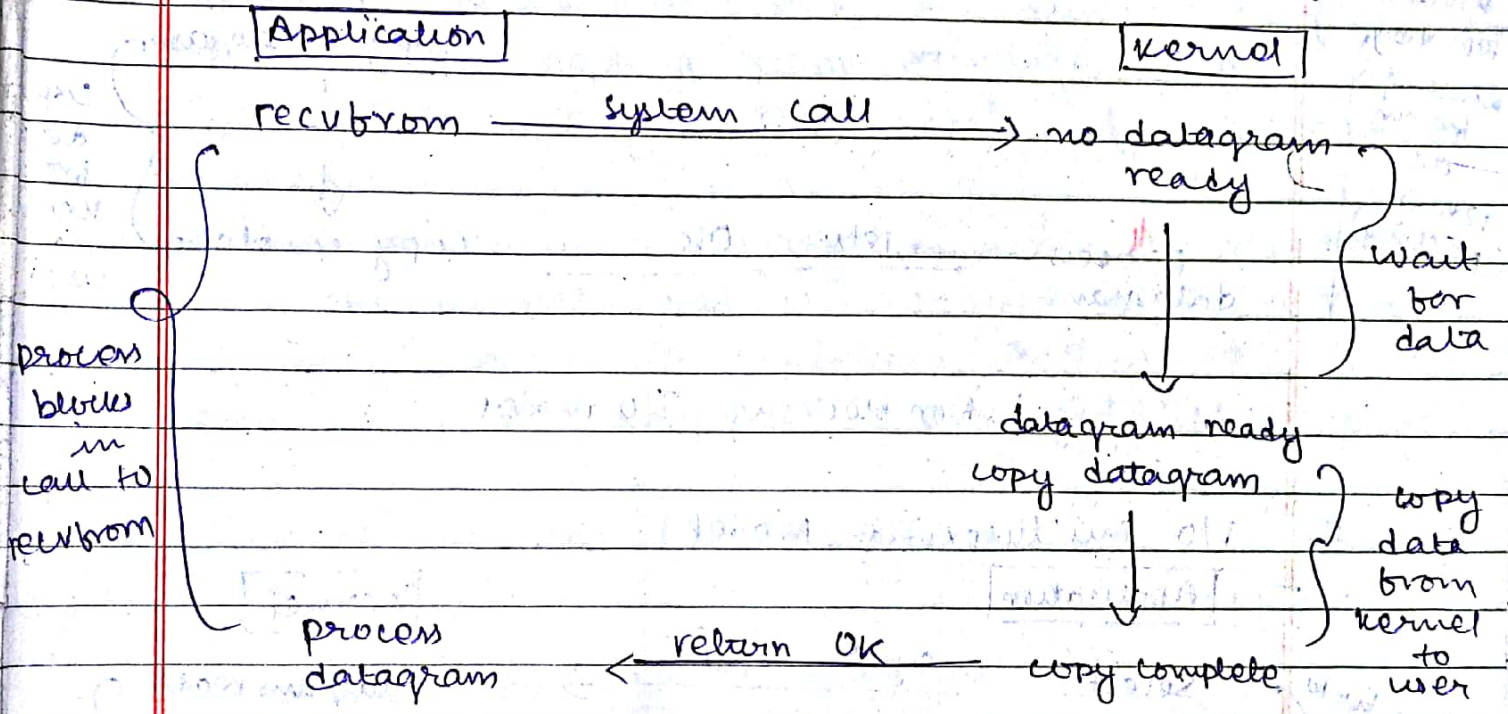
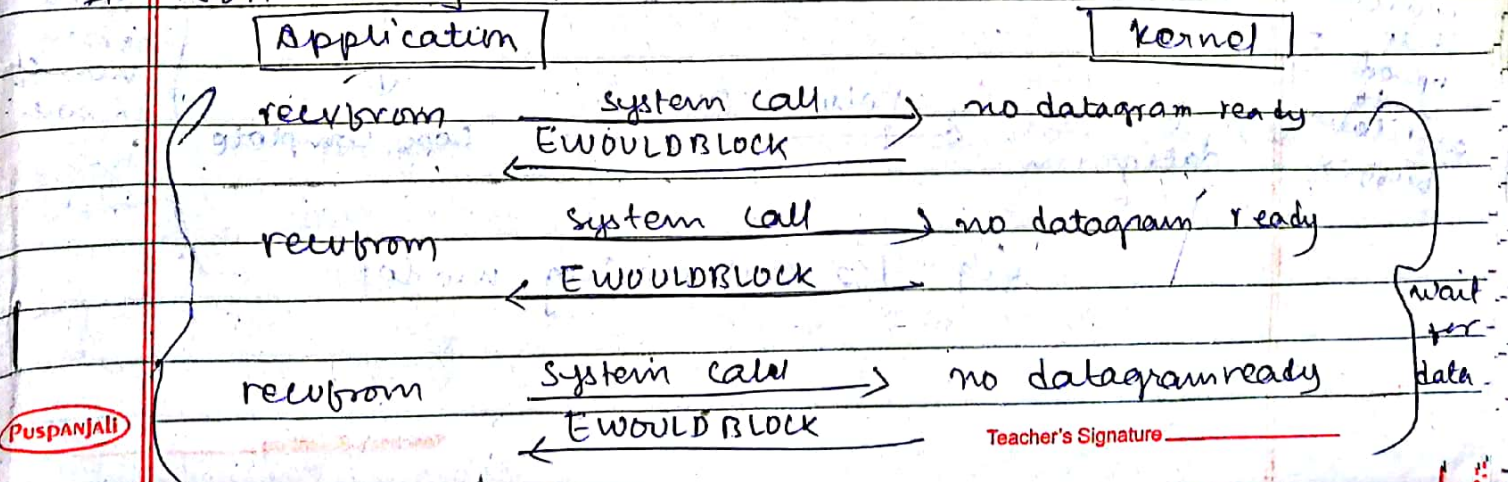


Fig: Blocking I/O model

Nonblocking I/O model



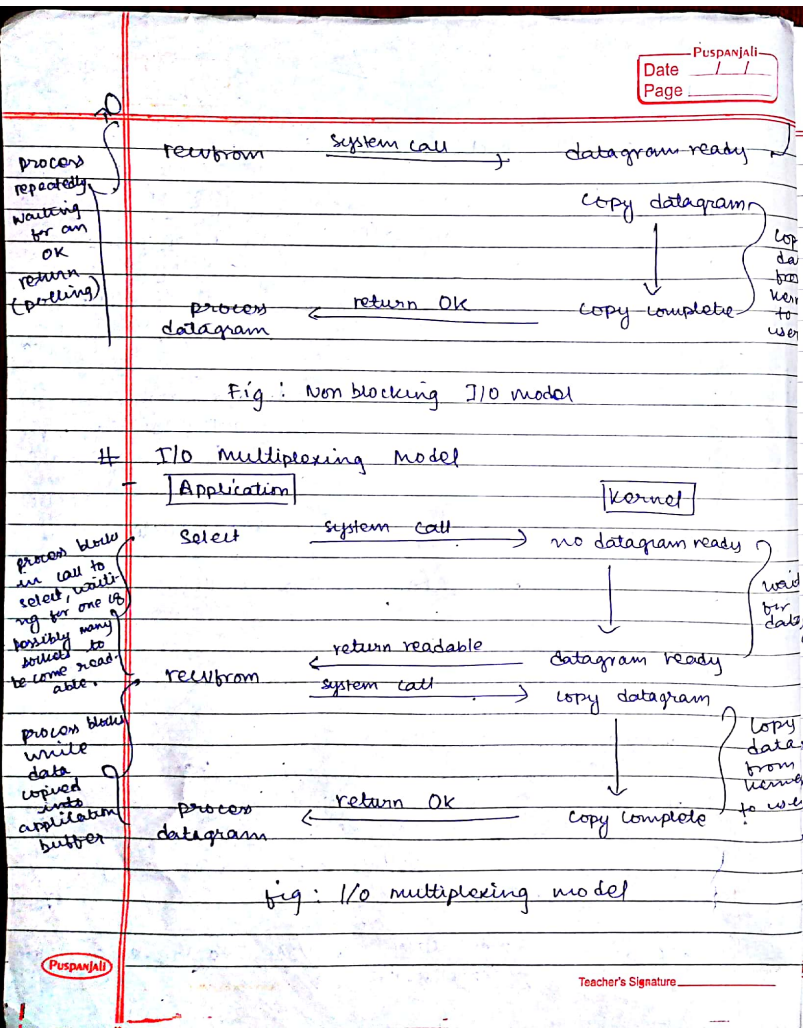


Fig: Non blocking I/O model

I/O multiplexing model

fig: I/O multiplexing model

Let it be assumed that the TCP server correctly sends a FIN to the client, but if the client process is blocked reading from standard input, it will never see the EOF until it read from the socket.

Hence it is necessary to make it capable to tell the kernel what is required to be notified if one or more I/O conditions are ready. This capability is known as "I/O multiplexing".

Signal-Driven I/O model:

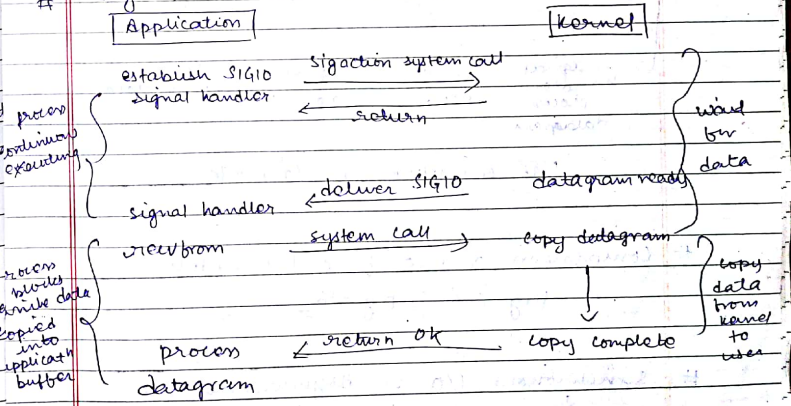


Fig: Signal-Driven I/O model

Asynchronous I/O model

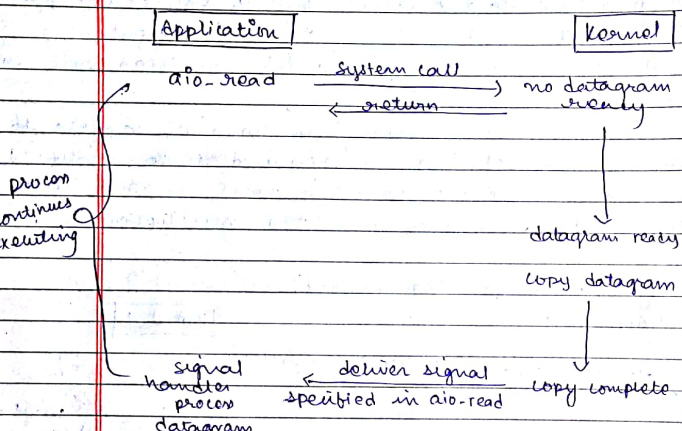


Fig: Asynchronous I/O model

Comparison of the five I/O models

Fig 6-6, pg-160

Synchronous I/O vs Asynchronous I/O

↓ 4 models (Clause requesting process to be blocked until I/O operation completes)

↓ Asynch. I/O model (doesn't)

Select():

allows the process to instruct the kernel to wait for any one of multiple events to occur & to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

```
#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:

+ve count of ready descriptors
0 on timeout
-1 on error

```
Struct timeval{
    long tv_sec;
    long tv_usec;
};
```

It's final argument tells the kernel how long to wait for one of the specified descriptors to become ready.

There are 3 possibilities:

1) Wait forever:
- return only when one of the specified descriptors is ready for I/O.
  For this, we specify
  timeout argument as null pointer

2) Wait up to a fixed ~~as~~ amount of time
- return when one of the specified descriptors is ready for I/O but do not wait beyond the time specified in timeval structure pointed to by the timeout argument

3) Do not wait at all:
- return immediately after checking the descriptors. called POLLing
  For this, we specify
  timeout argument must point to a timeval structure & timer value = 0.

- * Read Sir pdf * Good one -

Poll():
- sir note -
+
Book

Socket options:
- The system calls "getsockopt" and "setsockopt" can be used to set and inspect special options associated with sockets.
- "fcntl" and "ioctl" are system calls that make it possible to control files, sockets and devices in a variety of ways.

getsockopt and setsockopt functions:

```
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Both Return:
0 if OK
-1 if error

where,

- Sockfd refers → to an open socket descriptor.
- level specifies → the code in the system that interprets the option: the general socket code or some protocol-specific code (IPv4, IPv6, TCP, SCTP)
- optval → is a pointer to a variable:
  - from which the new value of the option is fetched by "setsockopt"
  - into which the current values of the option is stored by "getsockopt" and a value-result for "getsockopt".

* Generic Socket Option:

(1) SO_BROADCAST Socket Option:
- It enables or disables the ability of the process to send broadcast messages.
- Broadcasting is only supported for UDP (datagram sockets) and only on n/w that support the concept of a broadcast message like ethernet, ring token
- It is no supported on ~~p-to-p~~ point-to-point link or connection based transport protocol (TCP).

(2) SO_DEBUG :
- supported only by TCP
- kept in a circular buffer within the kernel
- when enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for socket.

3) SO_DONTROUTE :
- specifies that outgoing packets are to bypass the normal routing mechanism of the underlying protocol.

4) SO_ERROR :

5) SO_KEEPALIVE

8) SO_LINGER socket option
- It specifies how the "close" function operates for a connection-oriented protocol.

fcntl():
- fcntl stands for file control
- it performs various descriptor control operation.
- It provides the following features related to network programming

(i) Nonblocking I/O
- set O_NONBLOCK file status flag using F_SETFL command to set a socket as nonblocking

(ii) Signal-driven I/O:
- can set O_ASYNC file status flag using F_SETFL command which causes SIGIO signal to be generated

#include <fcntl.h>
- int fcntl(int fd, int cmd, ... /* int arg */);
Returns:
depends on cmd if OK
-1 on error

Daemon Process
- It is a process that runs in the background and is not associated with a controlling terminal.
- Ways to start a daemon:

1) During system startup, many daemon are started by the system initialization scripts. These scripts are often in /etc or /etc/rc and runs in superuser privileges

2) Many n/w servers are started by the inetd superserver. inetd itself is started from one of the scripts in step 1. inetd listens for n/w requests & when a request arrives, it invokes the actual server.

3) The execution of programs on a regular basis is performed by cron daemon

4) Daemons can be started from user terminals, either in the foreground or in the background.

5) The execution of a program at one time in the future is specified by the 'at' command.

- Since daemon does not have a controlling terminal, it needs some way to output messages when

something happens.
The Syslog function is the standard way to output those messages and it sends the messages to the syslogd daemon.

* Syslogd Daemon:
- Unix systems normally start a daemon named syslogd from one of the system initialization scripts, and it runs as long as the system is up.
- Implementation of syslogd perform following actions:

1) The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive.
These messages can be appended to a file (/dev/console), written to a specific user or forwarded to the syslogd daemon on another host.

2) A unix domain socket is created and bound to the pathname /var/run/log

3) A UDP socket is created & bound to port 514

4) The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.

- Syslogd daemon runs in a infinite loop that calls "select"
- If daemon receives the SIGHUP signal, it reread its configuration file.
- We could send log messages to syslogd from our daemons by creating a Unix domain datagram socket & sending our messages to the pathname that the daemon has bound but an easier interface is syslog()

Syslog():
- Since daemon doesnot have a controlling terminal, it cannot just fprintf to stderr.
- Common technique is to call syslog().

```
#include <syslog.h>
void syslog (int priority, const char *message, ...);
```

- priority argument is a combination of a level and a faility.

- message argument is like a format string to printf with the addition of a %.m specification.

- log messages have a level between 0-7.
- LOG_NOTICE is default if no level is specified by the sender.

- figure from the book -

openlog() & closelog()

#include <syslog.h>
void openlog (const char *ident, int options, int facility);

void closelog (void);

ioctl operation :
- A common use of ioctl by network programs is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether the interface supports broadcasting, whether the interface supports multicasting and so on.

Teacher's Signature

#include <unistd.h>
int ioctl (int fd, int request, -- /* void *arg */);

Returns:
0 if OK
-1 on error.

always a pointer

- The requests are divided into 6 categories
(1) Socket operation
(2) File operation
(3) Interface operation
(4) ARP cache operation
(5) Routing table "
(6) STREAMS system

Fig 17.1

summary of above 6 categories

File sharing & passing file descriptor :-
- Kernel uses 3 data structures to represent an open file and the relationship among them determine the effect one process has on another with regard to file sharing.

Teacher's Signature

1) Every process has an entry in the process table and within each process table entry is a table of open file descriptors (contain file descriptor flags & a pointer to a file table entry).

2) The kernel maintains a file table for all open files. Each file table contains
   a) The file status flags for the file such as read, write, append, sync & nonblocking
   b) The current file offset.
   c) A pointer to the v-node table entry for the file

3) Each open file (or device) has a v-node structure that contains information about the type of file and pointers to function that operate on the file.

Passing file descriptor (Pdf)

Unix Domain Socket

3 Winsock Programming

Origins of Windows Sockets:

Winsock specification:

- Winsock was proposed by Martin Hall of JSB softwares are at the Interop in October 1991.
- The windows socket specification defines a network programming interface for microsoft windows which is based on the "socket" paradigm popularized in the Berkeley software distribution from the university of California at Berkeley.
- It encompasses:
  Berkeley socket style routines + set of windows-specific extension
- It defines 2 interfaces:
  (1) API used by application developers
  (2) the SPI which provides a means for network software developers to add new protocol modules to the system.

- For example, UNIX applications were able to use the same errno variable to record both networking errors and errors detected within standard C library functions.
  Since this was not possible in windows, Winsock introduced a dedicated function, WSAGetLastError(), to retrieve error information

Teacher's Signature

Winsock 2.0 architecture:

- Winsock API is an API specification that defines how a windows network application should access underlying TCP/IP n/w services.

Two types of socket

1) Stream Socket (TCP socket)
2) Datagram Socket (UDP socket)

DLL:

- Dynamic link library is a collection of small programs, which can be called upon when needed by the executable program that is running.
- It lets the executable communicate with a specific device.

* Advantage of DLL

(1) They do not get loaded into RAM together with the main program, space is saved in RAM. When and if a DLL file is called, then it is loaded.

(2) Uses fewer resources:
When multiple programs use the same library of functions, a DLL can reduce the duplicat[ion] of code that is loaded on the disk & in RAM.

(3) Promotes modular architecture

(4) Eases deployment & installation.

- DLL is an executable file that cannot run on its own, it can only run from inside an executable file.

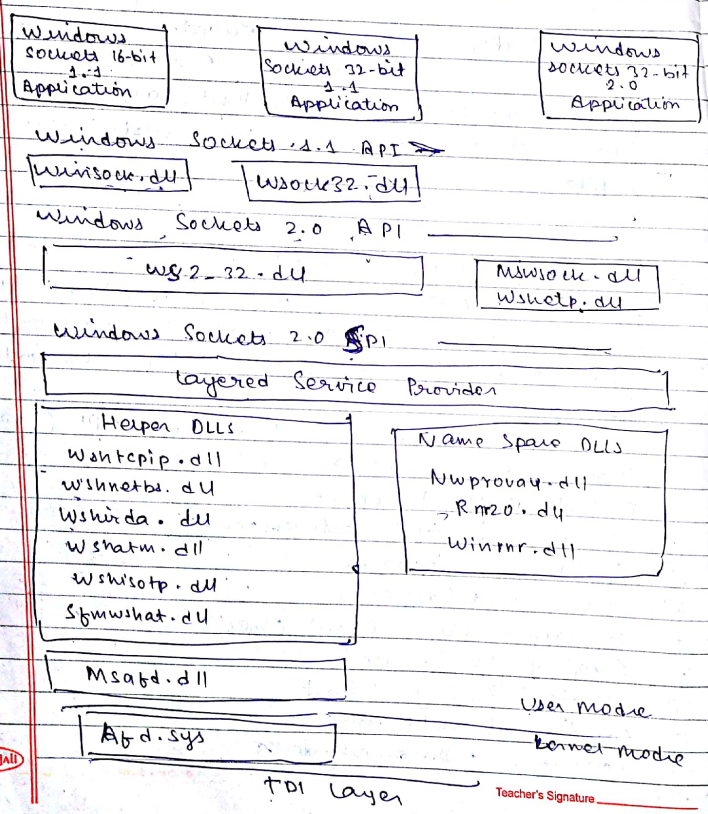
Winsock 2.0 architecture:

- Winsock API is an API specification that defines how a windows network application should access underlying TCP/IP network services.
- The current version of Winsock API (ver. 2.2.2) has several important features:

(1) Multiple Protocol Support
(2) Transport Protocol Independence
(3) Multiple Namespaces
(4) Scatter and Gather
(5) Overlapped I/O & Event Objects
(6) Quality of Service
(7) Multipoint & Multicast
(8) Conditional Acceptance
(9) Connect and Disconnect data

Teacher's Signature

10) Socket sharing
11) Vendor IDs and mechanism for vendor extensions
12) Layered Service Providers:

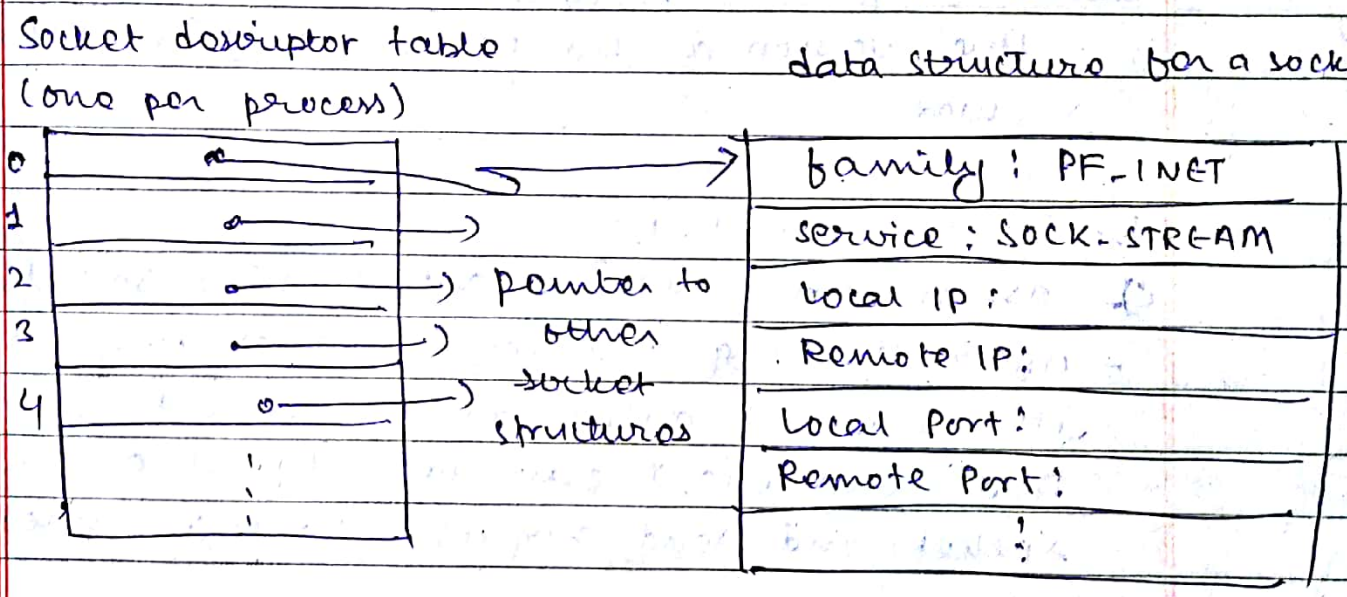


Teacher's Signature

Puspanjali
Date / /
Page

Socket Descriptor:

- Each socket is identified by an integer called socket descriptor which is an unsigned integer
- Windows keeps a table of socket descriptors for each process.
- Each socket descriptor is associated with a pointer which points to a data structure that holds the information about the communication session of that socket



Sketch of TCP client and TCP server.

* 3 major steps for communication :

1) Initialize Sockets
   It requires 3 steps
   a) Initialize Winsock DLL
      First application call.

Teacher's Signature

WSAStartup():
Then windows binds to the WinSock DLL and allocates the necessary resources.

include <winsock.h>
- int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData);

First argument: specifies the version of the requested WinSock.
Second argument: returns information about the version of the WinSock that is actually used

b) Create a socket:
c) Assign an endpoint address to a socket.
- After creating a socket, a server must assign its endpoint address to this socket.
Then the client can identify the server's socket and send requests to this server.

Steps:
- The server specifies its endpoint address. It assigns protocol family, IP address, port number to a structure of type sockaddr_in
- The server calls bind() to bind its endpoint address to the newly created socket

SOCKET socket (int af, int type, int protocol);

- int bind (SOCKET s, const struct sockaddr FAR *name, int namelen);

e) Communication betn sockets
Main steps
a) Client initiates a TCP connection
- After creating a socket, a client calls connect() to establish a TCP connection to a server.

- int connect (SOCKET s, const struct sockaddr FAR *name, int namelen);

b) Server listens to connection requests
- ~~Server~~ After creating a socket, the server calls listen() to place this socket in passive mode.

- int listen (Socket s, int backlog);

- Then the socket listens and accepts incoming connection requests from the clients.

c) Server accepts a connection request
- The server calls accept() to extract the next incoming connection request from the queue and then the server creates a new socket for this connection request and return the descriptor of new socket.

- SOCKET accept(SOCK s, struct sockaddr FAR* addr, int FAR* addrlen);

d) Send data
- The function send() copies the outgoing data into buffers in the OS kernel and allows the application to continue execution while the data is being sent across the network.
- If the buffers ~~have~~ become full, the call to send() may block temporarily.

- int send(SOCKET s, const char FAR* buf, int len, int flags);

e) Receive data
- Before calling recv(), the application must allocate a buffer for storing the data to be received.
- recv() extracts the data that has arrived at the specified socket and copies them to the application's buffer.

- int recv(SOCKET s, char FAR* buf, int len, int flags);

3) Close the sockets
- closesocket()
  terminate & deallocate socket.
- int closesocket(SOCKET s);

And WSACleanup() to deallocate all data structure and socket bindings
- int WSACleanup(void);

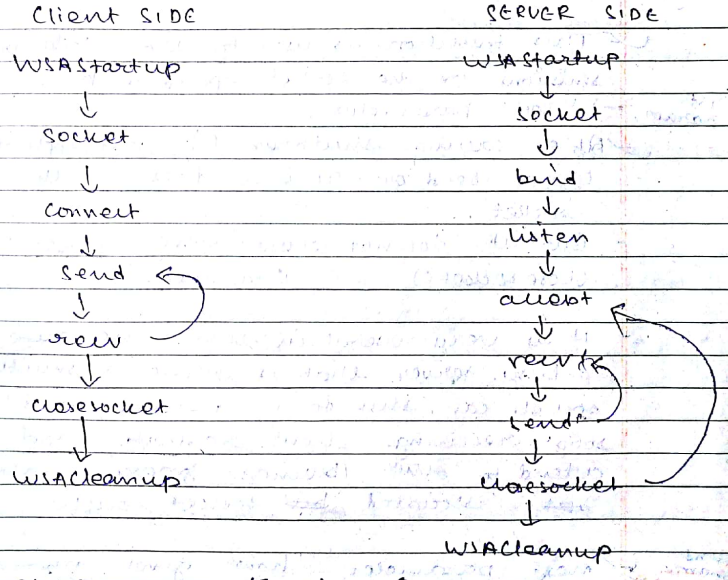


Fig: Sequence of Execution of WinSock Functions

Terminating a connection :-

- Once the client and server have completed their communication, they can terminate their connection by calling

1st: Shutdown() API

- to close down the connection ~~and the closesocket() API~~

int PASCAL FAR shutdown (SOCKET soc, int how);

This function is used to stop data transmissions on the socket specified by the

1st param. → "soc" parameter.

After calling shutdown(), an application cannot send or receive data on the socket.

- But it doesnot close the socket so closesocket() is used next.

- It is very useful in server because a busy server that is servicing multiple sockets can shut down a socket and stop receiving client requests and attend to other business freezing its ~~sock~~ associated ~~buf~~ socket buffer.

2nd param. next parameter "how" gives more control over how the transmissions are shut down.

int how = 0 ? what ?

- causes only the incoming traffic to cease ; all recv() function calls by the applicatn calling shutdown() will return an extended error (WSAESHUTDOWN)
- this may not affect the lower-level protocols such as TCP or UDP.
- TCP can continue to receive packets, but may not acknowledge them which leads to network inefficiency

&

- UDP may receive all incoming datagram packets and generate no ICMP (error) packets.

int how = 1 ? what ?

- causes all subsequent sends by shutdown() API caller to be disallowed.
- & all calls to the send() or sendto() APIs will return an extended error (WSAESHUTDOWN)

2nd: closesocket() API:

- to close down the connection the socket itself

int PASCAL FAR closesocket (SOCKET soc);

Teacher's Signature

This function call releases any system resources associated with that socket such as binding to protocols, and any queued data.
- It abborts the lower protocols which shutdown by sending an appropriate message to their counterpart on the remote end.
- It can block on a blocking stream socket until all the unsent data is successfully sent to the recipient.

Asynchronous Operations:

- While blocking sockets are easy to use and understand, they are not suitable for sophisticated NT client/server applications or Windows applications.

Reason

Why not blocking sockets in windows?

(1) Blocking call causes the application to stop processing user requests, leading to a "hung" program.
(2) Due to the single message queue system in the windows environment, a blocking application adversely affects entire system.
(3) In windows NT environment, a single thread blocking application cannot affect the rest of the system but the lack of response from the application may lead to an unhappy user.

Following function may block for an indefinite period of time when called on a blocking socket

(1) accept() - used by server
(2) connect() - used by client
(3) recv()/recvfrom() - used by both client & server
(4) send()/sendto() - " "
(5) close()

How to solve?

- Blocking can be avoided by using non-blocking calls on blocking socket
- By changing the default behavior of the socket from blocking to nonblocking.
- using non blocking socket

Select() & WSAAsyncSelect(), either of which can be used to avoid blocking on socket calls.

How?

* Using a Nonblocking Socket:

- A nonblocking socket is created by
(1) calling the socket() function to make a socket endpoint.

Teacher's Signature

(2) then calling ioctlsocket().

int PASCAL FAR ioctlsocket (SOCKET soc, long cmd, u_long FAR * argp);

1st param ⇒ ioctlsocket() gets a socket descriptor from the "soc" parameter and can be used to control blocking.

2nd param ⇒ - "cmd" parameter consist of :-

* FIONBIO :-
  - it changes the state of the socket from blocking to nonblocking.
  - when setting cmd = FIONBIO, argp points to an unsigned long number that is set to nonzero to change the socket mode to nonblocking.
  - A argp to zero changes non-blocking to blocking conversely.

* FIONREAD
  - It is used to determine the amount of data that can be read from the socket.
  in this case, argp points at an unsigned long in which ioctlsocket() function stores the result.
  For stream sockets,

Teacher's Signature

it returns the total amount of data that can be read in a single recv()
For datagram sockets,
It returns the size of the first datagram queued on the socket

* SIOCATMARK
  - It is used to determine whether all out-of-band data has been read.
  This applies only to stream sockets that have been configured for in-line reception of any out-of-band data (SO_OOBINLINE).

- Once the socket has been designated as nonblocking all socket functions return with an error (WSAEWOULDBLOCK) any time a call does not complete immediately.
So advantage is:
- application must do not hang on given socket call.
and disadvantage is:
- applications must poll the socket periodically to determine when a function call will complete successfully.

int PASCAL FAR ioctlsocket (SOCKET soc, long cmd, u_long FAR * argp);

Teacher's Signature

* Using non-blocking socket with connect():
- client application can block on a connect() API call for a long period of time.
- The time-out value depends on the time-out values and policies of the underlying protocol
- This long delay can be avoided by marking the socket as non-blocking prior to calling the connect() API.
* If connection is not available :
- connect() function will return the extended error (WSAEWOULDBLOCK) and try to connect later

* Connection is complete:
- the application can continue to use nonblocking socket for all subsequent I/O calls.

Note: it is also possible to change the socket to blocking mode after the connect() is successful.

* Using select() in conjunction with accept():
Use of select() ?
- It is the means by which a server can determine if a client request is pending in the request queue.

Teacher's Signature

Once server knows that a client request has arrived, it can call the accept() API and establish a logical connection without blocking on the accept() function

- select() is used primarily in the windows & ^NT environments to determine the status of one or more sockets and ask for information on read, write or error status.

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

typedef struct fd_set {
        u_short fd_count  /* how many in SET ? */
        SOCKET fd_array[FD_SETSIZE]; /* array of sockets */
} fd_set;
```

- Before using select(), a set of socket descriptors must be set by using fd_set structure through FD_SET() macro.
- Here caller can specify read, write or error status and time out value including zero,

Teacher's Signature

which allows control over how long select() can block when trying to determine the status before returning. through readfds, writefds, exceptfds and timeout parameter respectively.
And maximum no. of sockets in a socket set is specified by nfds.

- On successful completion, the select() API returns the no. of sockets for which it has information, including zero.

- There are 3 ways to use the time-out value to control the amount of time the socket() function blocks and the information that is returned by the select() function:

1) If the timeout parameter is set to Null, the select() function blocks indefinitely until one or more of the socket descriptors meet the condition.
This option is not so useful.

2) If a non zero time-out value is specified by the time-out parameter, then the select() function blocks until one of the sockets is ready or the specified time has expired.

3) If a zero time-out value is specified by the time-out parameter, then the select() function returns immediately, collecting status on all sockets specified by socket sets.

* not blocking on accept() on a stream socket:
- To determine if a client is trying to connect, the server puts the socket in a read socket set and calls the select() function.
- If the client request is pending, then the FD-ISSET() function will return TRUE on the socket, and the server can call the accept() function to establish the connection without blocking.

* Using select() with recv()/recvfrom()
- Recall that both the recv() and recvfrom() functions can block for an indefinite time on a blocking socket and that the select() function can be used to avoid blocking.
- And also that either the recv() or the recvfrom() functions block when the incoming buffer is empty and the receive has

Specified a non-null receive buffer.

How to avoid blocking?

This can be done if you know that one or more bytes are available in the socket. and this can be done by select().

- Using select() with a small or zero time-out value can determine if a socket is ready to be read.

* Using select() with send() / sendto()
- You can avoid blocking on send() or sendto() function if you can be sure no more data can be written into the socket.
- This is where select() function comes into play.
- Use the select() function with small or zero time-out value to determine if a socket to be written to is ready.

Teacher's Signature

Windows Sockets extension

Winsock extension have been designed to marry two orthogonal programming paradigms.

(A) A nonblocking I/O over a socket requires polling/looping mechanism.

(B) The message-driven windows and windows NT application programming paradigm

* Setup and cleanup Function
- necessary for all Winsock applications

(1) WSAStartup().
- Winsock application must call it before making any socket call.
- It registers application with windows socket implementation.

```
int PASCAL FAR WSAStartup (WORD wVersionRequired,
                           LPWSADATA lpWSAData);
```

- WSAStartup() causes the application to activate the winsock.DLL.
- It also allows the Winsock implementation to start keeping track of the application & causes a reference count to be set to one when called the first time.

Teacher's Signature

- An application can call WSAStartup() many times but it cannot request different version numbers after the first call.

(2) WSACleanup():

- application must call WSACleanup() before exiting.
- It frees any resources allocated by Winsock implementation for this application

- int PASCAL FAR WSACleanup(void);

Error handling functions

- Traditionally socket implementation & libraries follow Berkeley UNIX conventions, Returning:
  - -1 on error

  setting errno to reason for error.
- This has 2 problem

(1) In BSD, all return values are declared as C int type which can be either 16 or 32 bit and its byte ordering is also different both little & big endian.

This causes an acute problem in relaying the results of an Operation in DS.

2) while a single process-wide errno make sense in a single-thread environment, it leads to increased complexity and inefficiency in a multithread environment such as Windows NT.

These are solve by WinSock extension by:

(1) Using portable macro ERROR-INVALID-SOCKET
(2) Specifying that all applications call the WSAGetLastError() API to determine the exact error code.

2 function for error handling

(1) WSAGetLastError():

- int PASCAL FAR WSAGetLastError(void);
- gets error code maps to SetLastError() Win32 API

(2) WSASetLastError():

- void PASCAL FAR WSASetLastError (int iError);
- sets error code, maps to GetLastError() Win32 API.

Teacher's Signature

Functions for Handling Blocked Sockets

- Major challenges faced by socket based application writer is porting existing Berkeley ~~to~~ socket-based application to a windows environment.
- Porting those applications to Windows NT may be easy (beoz. it is 32 bit and support fork())
  But porting to a 16-bit windows env. is difficult (beoz. no multithreading support).
- In either case porting do not work with the message-based windows programming paradigm.

* To help those applications that use blocked sockets, WinSock provides 2 features:

(1) In a non-threaded Windows environment, while the socket call itself blocks, the WinSock implementation keeps processing the messages for the applicatn and sending it to the applicatn. Thus an application becomes re-entrant and does not become totally ~~res~~ unresponsive.

(2) In multithread windows env, there is no such default non-blocking mechanism in place and the thread making the socket call blocks while other threads continue.

Function list for handling blocks I/O are:

(1) WSAIsBlocking()
- check if a call to socket has blocked.
- returns true if there is pending call.
- BOOL PASCAL FAR WSAIsBlocking (void);

(2) WSACancelBlockingCall():
- Cancel a call to Windows socket that might be blocked.
- This API causes the blocked socket call to return as soon as possible with an error-interrupt message, WSAEINTR.
- When it is cancelled, the state of the socket depends on the particular call that was blocked.
- int PASCAL FAR WSACancelBlockingCall (void);

(3) WSASetBlockingHook():
- Associate an application-supplied function with WinSock that is called when a socket is blocked.
- This API help sophisticated applications

Teacher's Signature

use their own handlers to deal with a blocked socket & thus override the default blocking socket handling function of windows socket library.

- FARPROC PASCAL FAR WSASetBlockingHook(
  FARPROC lpBlockFunc);

④ WSAUnhookBlockingHook():-
- Dissociate the application-supplied function called when a socket is blocked.

- int PASCAL FAR WSAUnhookBlockingHook(void);

Asynchronous Database Function
- Windows Socket provides asynchronous database functions that send a message to an application upon completion of a function.
- Note: Winsock also supports the Berkeley socket (synchronous) database functions for backward compatibility.

Teacher's Signature

"Function are:

① WSAAsyncGetServByName():
- Asynchronous form of getservbyname()
- HANDLE PASCAL FAR
  WSAAsyncGetServByName(HWND hWnd,
  unsigned int wMsg,
  const char FAR * name,
  const char FAR * proto,
  char FAR * buf
  int buflen);

② WSAAsyncGetServByPort():
- HANDLE PASCAL FAR
  WSAAsyncGetServByPort(HWND hWnd,
  unsigned int wMsg,
  int port,
  const char FAR * proto,
  char FAR * buf,
  int buflen);

Teacher's Signature

(3) WSAAsyncGetProtoByName

```
HANDLE PASCAL FAR
WSAAsyncGetProtoByName( HWND hWnd,
        unsigned int wMsg,
        const char FAR * name,
        char FAR * buf,
        int buflen);
```

(4) WSAAsyncGetProtoByNumber :

```
HANDLE PASCAL FAR
WSAAsyncGetProtoByNumber ( HWND hWnd,
        unsigned int wMsg,
        int number,
        char FAR * buf,
        int buflen);
```

(5) WSAAsyncGetHostByAddr ( ) :

```
HANDLE PASCAL FAR
WSAAsyncGetHostByAddr ( HWND hWnd,
        unsigned int wMsg,
        const char FAR * addr,
        int len,
        int type,
        char FAR * buf,
        int buflen);
```

(6) WSAAsyncGetHostByName

```
HANDLE PASCAL FAR
WSAAsyncGetHostByName ( HWND hWnd,
        unsigned int wMsg,
        const char FAR * name,
        char FAR * buf,
        int buflen);
```

4. Asynchronous I/O Functions

(1) WSACancelAsyncRequest()
- Asynchronously cancel a pending request made

(2) WSAAsyncSelect()
- Asynchronous form of select()

(3) WSARecvEx()
- Asynchronous version of recv() available in windows NT only

Asynchronous Select :

- WSAAsyncSelect() API allows application to be socket-state driven as well as message-driven.
- Advantage of socket-state-driven?
  - Application reacts to changes in socket state rather than functioning as a single monolithic code full of if-then-else & write-do constructs.
  (simply in short - readability and maintainability of the code).
- Advantage of being both Socket and message driven?
  - It fits in nicely with the windows message driven paradigm and makes programs easy to understand & debug.
- An application can call the WSAAsyncSelect() to make itself available to receive certain messages when the state of the socket changes.
  Whenever that state changes, an event occurs and the application receives a specified message.

For example:

a socket-based server can request, by using the WSAAsyncSelect() API, that it receive a message whenever a new client connection request comes in. Upon receiving such a message, it can then call the accept() & start communicating with the client.

```
int PASCAL FAR WSAAsyncSelect (SOCKET soc,
                               HWND   hWnd,
                               unsigned int wMsg,
                               long   lEvent);
```

- An application can call WSAAsyncSelect() many times on same socket. However, the last call overrides all previous calls.

Out-of Band

Puspanjali
Date / /
Page

Steps involved in crashing & rebooting of server host

Socket utility function

socket primary "

network db administration func.

UDP echo server

Code for Sending & Receiving data in windows and Unix

Unix domain & Internet domain socket structure

Teacher's Signature

4

Date / /
Page

Sending and Receiving Data Over Connection :-

① On Stream Socket :

- Prototypes for the send() and recv() function :

```
int PASCAL FAR send ( SOCKET     soc,
                      const char FAR * buffer,
                      int        len,
                      int        flags);

int PASCAL FAR recv ( SOCKET     soc,
                      char FAR * buffer,
                      int        len,
                      int        flags);
```

Parameter :

① soc = valid socket descriptor

For server

soc = socket descriptor returned by accept()

For client,

soc = socket descriptor returned by connect(),

on error both accept() & connect() return ERROR-SOCKET.

② buffer = no. of bytes in data buffer to be transmitted.

Teacher's Signature

MSG_DONTROUTE - bypass message routing relevant to send() or sendto()
MSG_OOB = Send or receive out of band data
MSG_PEEK = Peek at incoming message relevant to recv() or recvfrom()

① len = length
② flags. = can modify data transmission semantics

- Send() function returns the number of bytes transmitted upon successful completion which can be in the range [1, length of data buffer length].
- recv() API returns the no. of bytes copied into the caller's address space when the data transmission completes.
- it return 0 when socket connection has been closed and WSAECONNRESET when a socket connection is abnormally disconnected.
- Stream socket is really a stream of bytes with no boundaries bet^n any two messages
- The incoming & outgoing socket buffer are distinct

The data structures given can be used by both client & server to send and receive variable length buffer

```
typedef struct _tagData {
    unsigned short usOpCode;   // allows C/Server to distinguish type of data
    unsigned long  ulBuffLen;  // Buffer length
    byte           bData[1];   // variable length data
};
```

For fixed length data buffer is much simpler.

Procedure for variable length buffer:
Sender (client or server) can put a command in usOpCode and specify the length of the data buffer to follow in ulBuffLen.
When the sender's data is less than receiver's buffer size, the receiver will block until the recv() API times out.
However when the incoming data does not fit into the receiver's buffer, recv() will return and fill up the receivers buffer.

(2) On SOCK_DGRAM Socket

- TO send and receive data with datagram socket, the client and server applications first create a datagram socket by calling the socket() API & binding their address to the socket by calling bind().

Teacher's Signature

- Then they can use sendto() & recvfrom() to send & receive data. These functions can be used with stream sockets as well.

- int PASCAL FAR sendto (SOCKET soc,
  const char FAR * buffer,
  int len,
  int flags
  const struct sockaddr FAR * to;
  int tolen);

- int PASCAL FAR recvfrom (SOCKET soc,
  char FAR * buffer,
  int len,
  int flags;
  struct sockaddr FAR * from,
  int FAR * fromlen);

* Using recvfrom() / sendto()

Client Calls	Server Calls
socket()	socket()
bind()	bind()
sendto()	recvfrom()
recvfrom()	sendto()
close()	close()

Functions used with Datagram Socket

(1) Using recvfrom()/sendto()

Server calls	Client calls	Description
socket()	socket()	Establish SOCK_DGRAM socket
bind()	bind()	Bind endpoint to local address
recvfrom()	sendto()	Client request sent to server
sendto()	recvfrom()	Server responds back to client
close()	close()	Close socket created by socket

(2) Using send()/recv()

Server calls	Client calls	Description
socket()	socket()	
bind()	bind()	
connect()	connect()	Establish remote end-point address to be used with send() & recv()
recv()	send()	
send()	recv()	
close()	close()	

Difference betn Datagram Socket and Stream Socket:

Issue	Stream Socket	Datagram Socket
Data delivery	Guaranteed and in order	Not guaranteed. Packets may be lost, duplicated, & arrive out of order.
Network packet utilization	Higher than datagram	Lower because receiver and socket sender's network address must be carried in each packet.
Protocol overhead	High because protocol handles errors, transmission	Lower because protocol does little error control
Application	Traditional client/ser application that let the protocol take care of error control, data packet ordering & other guaranteed data delivery issues	Application that either do not need error control or can achieve higher throughput by writing small application level error control protocol.

(250) #

Imp #

Difference betn Winsock & Unix(BSD) socket:

	BSD	Winsock
Header file	- It includes: # include <sys/types.h> # include <sys/socket.h> # include <netinet/in.h> # include <arpa/inet.h> # include <netdb.h>	- you don't need any of those. Instead just need to include winsock.h or winsock2.h (for winsock 2)
	- ~~WS~~ Errno	- WSAGetLastError() it do not return its error codes in errno variable. It must call WSAGetLastErrorMessage().
Closing Socket	- close()	closesocket()
	- Unix provides ioctl() call to allow you to set & get various bits of info on a file descriptor which include socket descriptors.	- winsock replicates some common Unix ioctls in the ioctlsocket() call, but much is missing
	- most BSD socket implementations do not pass delayed UDP errors to recvfrom()	- Winsock 2 does.
		- needs WSAStartup() & WSACleanup().

System call

It is sometimes referred to as a kernel call, is a request in a Unix-like operating system made via a software interrupt by an active process for a service performed by the kernel.

#

Unix domain socket implementation

* Unix domain stream protocol: echo server.id

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>

#define UNIXSTR_PATH "/home/shikhal/NP"

int main(int argc, char **argv)
{
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clilen;
    struct sockaddr_un cliaddr, servaddr;
    void sig_child(int);

    listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
    // delete if file name already exists
    unlink(UNIXSTR_PATH);
    bzero(&servaddr, sizeof(servaddr)); // fill 0 in structure
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, UNIXSTR_PATH);
```

```
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
Listen (listenfd, LISTENQ);

Signal(SIGCHLD, sig_chld);

for ( ; ; )
{
    clilen = sizeof(cliaddr);
    if ((connfd = accept(listenfd, (SA *) &cliaddr,
                         &clilen)) < 0)
        if (errno == EINTR)
            continue;
        else
            err_sys ("accept error");
    }

if ((childpid = Fork()) == 0)   //child process
{
    close (listenfd);
    str_echo (connfd);
    exit(0);
}
close (connfd);
}
}
```

Teacher's Signature

A unix domain stream protocol echo client

```
#

#define UNIXSTR_PATH "/home/nishal/NP"

int main (int argc, char **argv)
{
    int sockfd;
    struct sockaddr_un servaddr;
    sockfd = Socket (AF_LOCAL, SOCK_STREAM, 0);

    bzero( &servaddr, sizeof (servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy (servaddr.sun_path, UNIXSTR_PATH);

    Connect (sockfd, (SA *) &servaddr, sizeof (servaddr));

    str_cli (stdin, sockfd);
    exit(0);
}
```

Teacher's Signature

AF_LOCAL → to create a Unix domain datagram socket

Puspanjali Date / / Page

* Unix domain datagram protocol echo server

```
#
.
.
# define UNIXDG_PATH "/home/nishal/NP"
int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_un servaddr, cliaddr;

    sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

    unlink(UNIXDG_PATH);   // pehle (consolidate dekho pehli lecture)

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;

    strcpy(servaddr.sun_path, UNIXDG_PATH);

    bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

Teacher's Signature

Puspanjali Date / / Page

* Unix domain datagram protocol echo client

```
#
.
.
# define

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_un cliaddr, servaddr;
    sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sun_family = AF_LOCAL;
    strcpy(cliaddr.sun_path, tmpnam(NULL));

    bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, UNIXDG_PATH);

    dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
    exit(0)
}
```

Teacher's Signature

Winsock

TCP ~~client~~ echo client

```
#include <stdio.h>
#include <winsock2.h>

int main()
{
  WSADATA      wsadata;
  SOCKET       s;
  struct sockaddr_in ServerAddr;
  char *message = "Hello";
  char buf[5], *bptr;
  int i, buflen, count;

  // call WSAStartup & socket()
  WSAStartup (MAKEWORD(2,0), &wsadata);
  s = socket (PF_INET, SOCK_STREAM, 0);

  // call connect() to connect to the server /
  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(2000);
  ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  connect(s, (struct sockaddr *) &ServerAddr,
          sizeof(ServerAddr));

  send(s, message, strlen(message), 0);
  bptr = buf; buflen = 5;
```

Teacher's Signature

```
  recv(s, bptr, buflen, 0);
  // echo received mssg
  for (i=0; i<5; ++i)
  {
    printf("%c", buf[i]);
  }
  printf("\n%s\n", "Bye!");

  closesocket(s);
  WSACleanup();
  scanf("%s", buf);
}
```

* TCP echo server

```
#include <stdio.h>
#include <winsock2.h>
int main()
{
  WSADATA      wsadata;
  SOCKET       s, nsock;
  struct sockaddr_in ServerAddr, ClientAddr;
  char buf[5], *bptr;
  int i, buflen, count;
  WSAStartup(MAKEWORD(2,0), &wsadata);
  s = socket(PF_INET, SOCK_STREAM, 0);
  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(2000);
```

Teacher's Signature

```
ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

bind(s, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr));

listen(s, 1);
l = sizeof(ClientAddr);
nsock = accept(s, (struct sockaddr *)&ClientAddr, &l);

bptr = buf;
buflen = 5;

recv(nsock, bptr, buflen, 0);

send(nsock, buf, strlen(buf), 0);

closesocket(nsock);
closesocket(s);
WSACleanup();
return 0;
}
```

Teacher's Signature

Steps involved in crashing & rebooting of server host

1) We start the server & then the client. We type a line to verify that the connection is established
2) The Server host crashes & reboots.
3) We type a line of input to the client, which is sent as a TCP data segment to the server host.
4) When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
5) Our client is blocked in the call to readline when the RST is received, causing readline to return the error ECONNRESET.

Socket utility function:

- BOOL ReportStatusToSCMgr(DWORD dwCurrentState,
  DWORD dwWin32ExitCode,
  DWORD dwCheckPoint,
  DWORD dwWaitHint);
- This function is called by ServiceMainFunction() and ServiceCtrlHandler() to update the service's status to service control manager.

Teacher's Signature