CSCE 3600
Principles of Systems Programming

Interprocess Communication
Part 2

University of North Texas

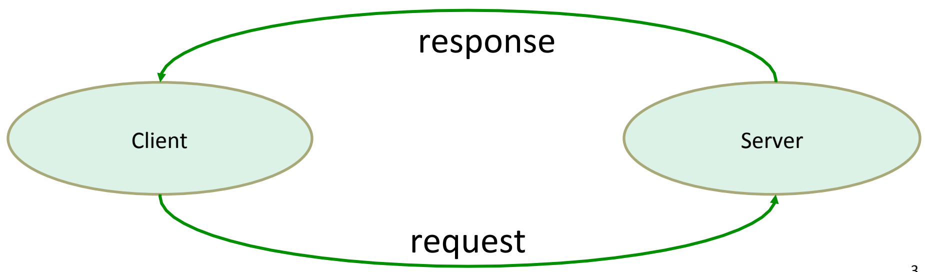
What are Sockets?

- Sockets are an extension of pipes, with the advantages that processes don't need to be related, or even on the same machine
  - Method for accomplishing interprocess communication
  - Bidirectional communication
- A socket is like the end point of a pipe – in fact, the Linux kernel implements pipes as a pair of sockets
- Two (or more) sockets must be connected before they can be used to transfer data

2

Socket Client-Server Model

- IPC using sockets described as client-server model
  - One process is usually called the server
  - A server process is usually responsible for satisfying requests made by other processes called clients
  - A server usually has a well-known address (e.g., IP address or pathname)



3

Socket Attributes

- Sockets are characterized by three attributes
  - Domain
  - Type
  - Protocol

- Socket Protocol
  - Determined by socket type and domain
  - Default protocol is 0

- For communication between processes, sockets can be implemented in the following domains
  - UNIX (e.g., AF_UNIX)
    - Processes are on the same machine
  - INET (e.g., AF_INET)
    - Each process is on a different machine (i.e., requires a network interface device)

4

Socket Attributes

- Three main types of sockets:
  - SOCK_STREAM (TCP sockets)
    - Provides a connection-oriented, sequenced, reliable, and bidirectional network communication service
    - E.g., telnet, ssh, http
  - SOCK_DGRAM (UDP sockets)
    - Provides a connectionless, unreliable, best-effort network communication service
    - E.g., streaming audio/video, IP telephony
  - SOCK_RAW
    - Allows direct access to other layer protocols such as IP, ICMP, or IGMP

5

CSE

Unix Domain Stream Sockets

6

UNIX Domain Sockets

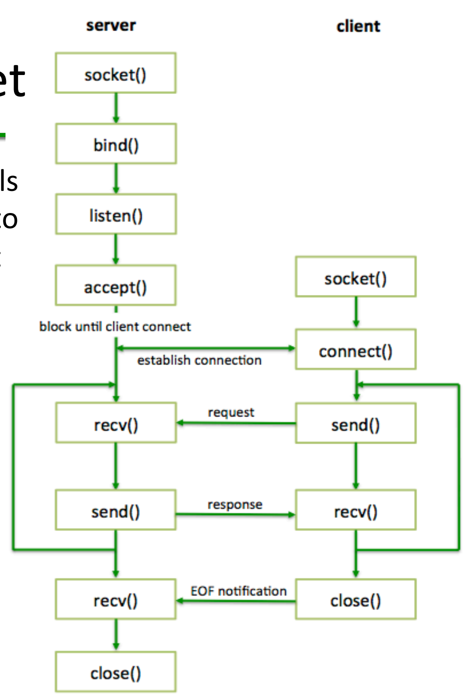
- Socket for communicating with another process on the same machine only
  - Provides an optimization since no network overhead
- Uses sockaddr_un structure instead of sockaddr_in

```
#include <sys/un.h>
struct sockaddr_un
{
    sa_family_t sun_family;    /* AF_UNIX */
    char sun_path[108];        /* pathname */
};
```

7

Stream Socket

- Sequence of function calls for a client and server to implement a stream socket



8

Steps in Server Process

1. Call `socket()` with proper arguments to create the socket
2. Call `bind()` to bind the socket to an address (in our case, it is just a pathname) in the UNIX domain
3. Call `listen()` to instruct the socket to listen for incoming connections from client programs
4. Call `accept()` to accept a connection from a client
5. Handle the connection and loop back to `accept()`
6. Close the connection

Steps in Client Process

1. Call `socket()` to get a UNIX domain socket to communicate through
2. Set up a `struct sockaddr_un` with the remote address (where the server is listening) and call `connect()` with that as an argument
3. Assuming no errors, you are connected to the remote side
   – Use `send()` and `recv()` to communicate

Creating the Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Creates an endpoint for communication (i.e., a socket) and returns a file descriptor
- Common domains (or address family)
  - AF_UNIX Unix domain sockets
  - AF_INET IPv4 Internet sockets
  - AF_INET6 IPv6 Internet sockets
- Type
  - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- Protocol
  - Set to 0 (chosen by OS) except for RAW sockets

11

Bind to a Name (Server Side)

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *serveraddr,
    socklen_t addrlen);
```

- Binds a name to the socket (i.e., reserves a port)
- sockfd is the socket file descriptor returned by socket
- serveraddr is the IP address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
- addrlen is the size (in bytes) of the structure
- Returns 0 on success or –1 if an error occurs
- When socket is bound, new special socket-type file (type "s") corresponding to sun_path is created
- This file is *not* automatically deleted, so need to unlink it
  - If bind finds the file already exists, it will fail

12

Set Up Queue (Server Side)

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- Listens for connections on a socket
  - After calling `listen()`, a socket is ready to accept connections
- Prepares a queue in the kernel where partially completed connections wait to be accepted
- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests or let them wait
  - backlog is the maximum number of partially completed connections that the kernel should queue
- What if too many clients arrive?
  - Some requests don't get through … makes no promises
  - And the client(s) can always try again!

13

Establish Connection (Server Side)

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr* cliaddr,
    socklen_t * addrlen);
```

- Accepts a connection on the socket and returns a new file descriptor that refers to the connection with the client that will be used for reads and writes on the connection
  - Blocks waiting for a connection (from the queue)
- sockfd is the listening socket
- cliaddr is the address of the client

14

Establish Connection (Client Side)

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr*
    servaddr, socklen_t * addrlen);
```

- Initiates a connection on the socket, where the kernel will select a source IP address and dynamic port
- Returns 0 on success or –1 on failure

Sending Data

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

- Alternative to write that sends a message on the socket
- sockfd is socket descriptor for open and connected socket
- buf is a buffer of information to send
- len is the size of the buffer in bytes
- flags
  - Bitwise OR of zero or more flags (e.g., MSG_EOR, MSG_OOB, etc.)
- Calls are blocking (i.e., returns only after data is sent)

Receiving Data

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t  recv(int  sockfd,  void  *buf,  size_t  len,  int
    flags);
```

- Alternative to read that receives a message on the socket
- sockfd is socket descriptor for open and connected socket
- buf is initially empty to store the data
- len is the size of the buffer in bytes
- flags
    - Bitwise OR of zero or more flags (e.g., MSG_EOR, MSG_OOB, etc.)
- Calls are blocking (returns only after data is received)

17

Close and Shutdown

```
#include <unistd.h>
int close(sockfd);
```

- Closes the file descriptor when done
- Will not deallocate the socket until we close the last descriptor that references it (we may have several)

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

- Forces a full or partial closure of a socket
- sockfd is a socket descriptor
- how can be SHUT_RD, SHUT_WR, or SHUT_RDWR

18

CSE

Socket Pair

Socket Pipe

- What if you wanted a `pipe()`, but you wanted to use a single pipe to send and receive data from both sides?
  - Since most pipes are unidirectional (with exceptions in SYSTEM V), you cannot do it!
- UNIX domain sockets can handle bi-directional data
  - But, you would have to set up all that code with `listen()` and `connect()` with everything just to pass data both ways
- However, you can use `socketpair()` that returns a pair of already connected sockets
  - No extra work needed and you can immediately use the socket descriptors for interprocess communication (only available in UNIX domain socket)

socketpair Function

```
#include <sys/socket.h>
int socketpair(int family, int type, int protocol, int
    sockfd[2]);
```

- Creates two unnamed sockets that are already connected
- They are full-duplex (i.e., data can go in each direction)
- Also called stream pipes if type is SOCK_STREAM
- Returns 0 on success, –1 on error
- Family must be AF_LOCAL or AF_UNIX
- Type is SOCK_DGRAM or SOCK_STREAM
- Protocol must be 0

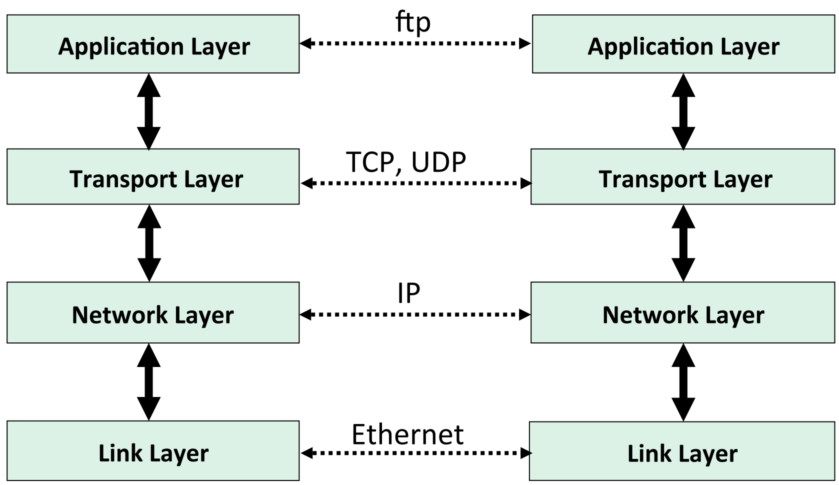
21

CSE

Internet Stream Sockets

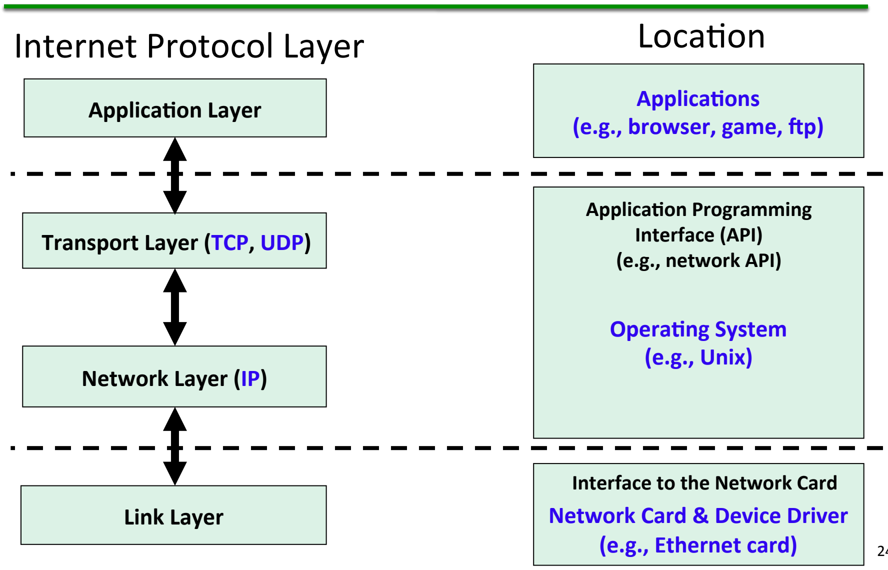
22

Layers of the IP Protocol Suite



23

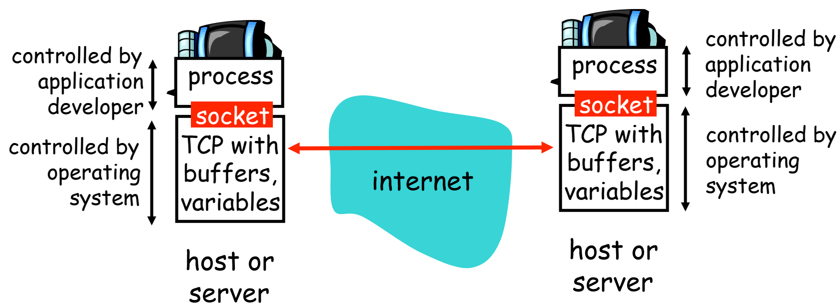
Protocol Suite Location



24

Internet Sockets

- Sockets provide a mechanism to communicate between computers across a network
- Internet Sockets
  - AF_INET – the addresses are IPv4 addresses
  - AF_INET6 – the addresses are IPv6 addresses



Byte Ordering of Integers

- Different CPU architectures have different byte ordering
- Little-Endian
  - Stores the least significant byte in the smallest address
- Big-Endian
  - Stores the most significant byte in the smallest address
- In order to connect to a remote computer and use a socket, we need to use its address
  - In fact, Linux is little-endian, but TCP/IP uses big-endian byte ordering

Byte Ordering

- Suppose we wanted to store the integer 305,441,741
  - Converts to $1234ABCD_{16}$

Little-Endian Byte Order

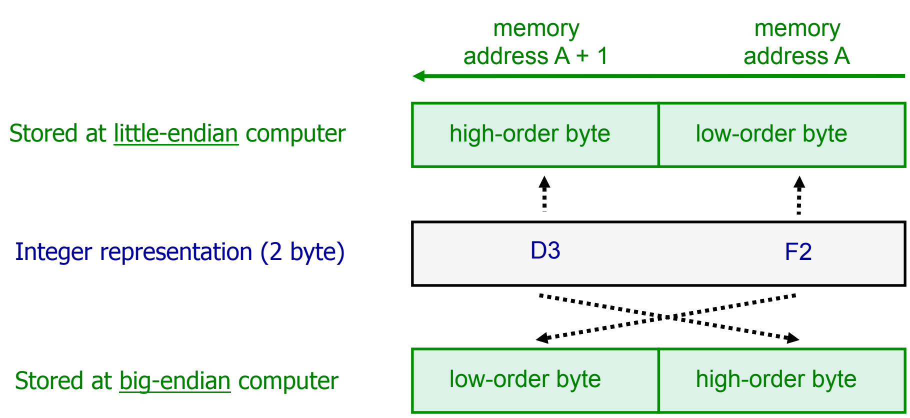
	Address	Value
Low Memory Address	0x8000	0xCD
	0x8001	0xAB
	0x8002	0x34
High Memory Address	0x8003	0x12

Big-Endian Byte Order

	Address	Value
Low Memory Address	0x8000	0x12
	0x8001	0x34
	0x8002	0xAB
High Memory Address	0x8003	0xCD

27

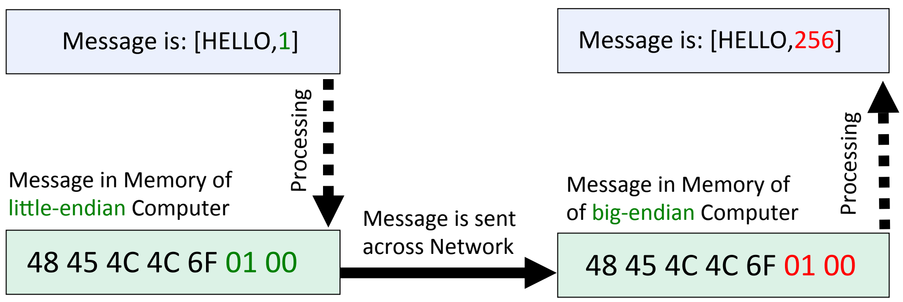
Byte Ordering of Integers



28

Byte Ordering Problem

- What would happen if two computers with different integer byte ordering communicate?
  - Nothing… if they do not exchange integers!
  - But… if they exchange integers, they would get the wrong order of bytes, and therefore, the wrong value!



29

Byte Ordering Solution

- There are two solutions if computers with different byte ordering system want to communicate
  - They must know the kind of architecture of the sending computer
    - Bad solution, it has not been implemented
  - Introduction of a network byte order. The functions are:

```
uint16_t htons(uint16_t host16bitvalue)
uint32_t htonl(uint32_t host32bitvalue)
uint16_t ntohs(uint16_t net16bitvalue)
uint32_t ntohl(uint32_t net32bitvalue)
```

- Use for all integers (short and long) that are sent across the network
  - Including port numbers and IP addresses

30

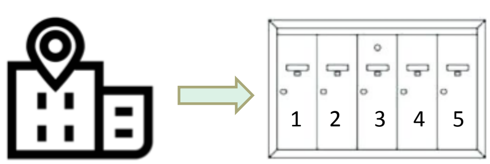
Naming and Addressing

- Host name
  - Identifies a single host
  - Variable length string (e.g., www.unt.edu, cse01)
  - Is mapped to one or more IP addresses
- IP Address (IPv4)
  - Written as dotted octets (e.g., 10.0.0.1)
  - 32 bits – not a number, but often needs to be converted to a 32-bit number to use
- Port number
  - Identifies a process on a host
  - 16 bit number

31

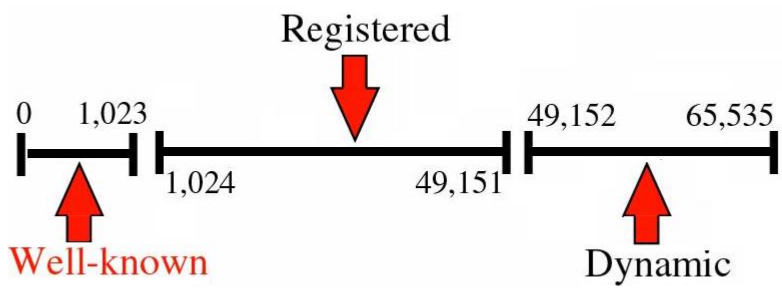
Addresses, Ports, and Sockets

- Like apartments and mailboxes
  - You are the application
  - Street address of your apartment building is the IP address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox
- How do you choose which port a socket connects to?



32

Port Numbers

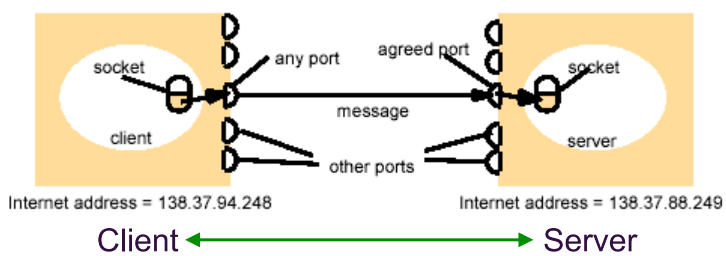


- You can find a list of all IANA registered ports at:
  - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

33

Addresses, Ports, and Sockets

- Choose a port number that is registered for general use, from 1024 to 49151
  - Do not use ports 0 to 1023. These ports are reserved (must be root) for use by the Internet Assigned Numbers Authority (IANA)
  - Avoid using ports 49152 through 65535. These are dynamic ports that operating systems use randomly
    - If you choose one of these ports, you risk a potential port conflict



34

IP Address Data Structure

- The sockaddr_in structure has four parts

```
struct sockaddr_in
{
  short int          sin_family;  // address family
  unsigned short int sin_port;    // port number
  struct in_addr     sin_addr;    // Internet address
  unsigned char      sin_zero[8]; // set to 0, padding
};

struct in_addr
{
  unsigned long      s_addr;      // 4 bytes
};
```

35

IP Address Data Structure

- Declare address structure

```
struct sockaddr_in sockAdd;
```

- Set family

```
sockAdd.sin_family = AF_INET;
```

- Set IP address (2 ways)

```
// specify address to listen to
inet_pton(AF_INET, "127.0.0.1", &sockAdd.sin_addr.s_addr)
// listen to any local address
sockAdd.sin_addr.s_addr = htonl(INADDR_ANY)
```

- Set port

```
sockAdd.sin_port = htons(9999);
```

36

Network Addressing Library Routines

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
in_addr_t inet_addr(const char* string)
```

- Converts an IP address in IPv4 numbers-and-dots notation into binary form in network byte order
- If string not contain legitimate Internet address, returns value INADDR_NONE

```
#include <unistd.h>
int gethostname(char* name, int nameLen)
```

- Gets the null-terminated hostname as a character array along with its length in bytes

Network Addressing Library Routines

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
```

- Converts an IP address in IPv4 numbers-and-dots notation into binary form in network byte order and stores it in an in_addr struct

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr in);
```

- Converts an IP address in the in_addr struct in network byte order into a string in IPv4 dotted-decimal notation

Other Library Routines

```
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *src, char
    *dst, socklen_t cnt);
```

- Converts a network address, either IPv4 or IPv6, from its binary to text form

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
```

- Converts a network address, either IPv4 or IPv6, from its text to binary form

CSE

Datagram Sockets

TCP vs. UDP

- TCP
  - Reliable byte stream service
  - Different ways to build clients and servers
    - Some problems when building clients/servers with TCP
      - How to get around blocking I/O
      - Data format conversion
    - Basic assumption: whatever sent will eventually be received!!
- UDP
  - Unreliable datagram service
    - Data may get lost – application may need to deal with more details in the communication

41

Why UDP?

- Applications that do not need 100% reliability communication
  - E.g., VoIP, video stream, DNS servers
- Applications care a lot about performance
  - High performance computing (TCP cares too much about other things than performance)
- Applications that need multicast or broadcast
  - TCP only supports point to point communication)

42

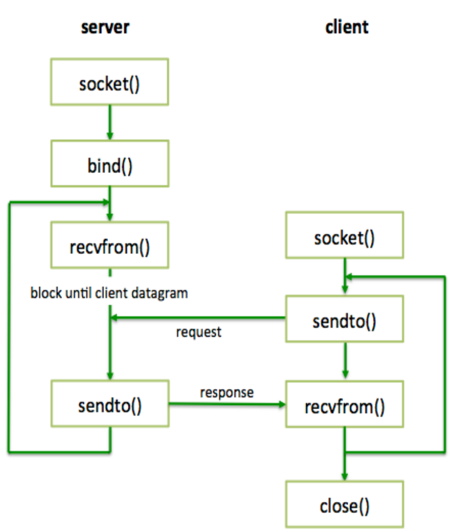
UDP Client/Server

- Typical UDP client
  - Client does *not* establish a connection with the server
  - Client sends a datagram to the server using sendto function
- Typical UDP server
  - Does *not* accept a connection from a client
  - Server calls recvfrom function that waits until data arrives from some client

43

Datagram Socket

- Sequence of function calls for a client and server to implement a datagram socket
  - No “handshake”
  - No simultaneous close



44

Receiving a Message

```
#include<sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes,
    int flags,struct sockaddr *from, socklen_t
    *addrlen);
```

- Receives a message on the socket
- If recvfrom is successful, the number of bytes read is returned, -1 on error

45

Sending a Message

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buff, size_t
    nbytes, int      flags, const struct sockaddr *to,
    socklen_t addrlen);
```

- Sends a message on the socket
- If sendto is successful, the number of bytes written, even if 0 bytes are written, is returned, -1 on error

46

CSE

I/O Multiplexing

47

Dealing with Blocking Calls

- Many functions block
  - accept — until a connection comes in
  - connect — until the connection is established
  - recv, recvfrom — until a packet (of data) is received
  - send, sendto — until data is pushed into socket's buffer
- For simple programs, this is fine
- What about complex connection routines?
  - Multiple connections
  - Simultaneous sends and receives
  - Simultaneously doing non-networking processing

48

Dealing with Blocking Calls

- Options
  - Create multi-process or multi-threaded code
  - Turn off blocking feature (e.g., use fcntl() file-descriptor control system call)
  - Use the select() function
- What does select() do?
  - Can be permanent blocking, time-limited blocking, or non-blocking
  - Input is a set of file descriptors
  - Output is information on the file-descriptors' status
  - Therefore, can identify sockets that are "ready for use" so that calls involving that socket will return immediately

49

I/O Blocking

```
socket();
bind() ;
listen();
while
   accept();
   recv();
   send();
```

- Simple server has blocking problem
  - Suppose 5 connections accepted
  - Suppose next *accept()* blocks
  - Other connections cannot send and receive
  - Cannot get keyboard input either

50

I/O Multiplexing

- select() waits on multiple file descriptors and timeout
- Returns when any file descriptor
  - Is ready to be read, or
  - Written, or
  - Indicates an error, or
  - Timeout exceeded
- Advantages
  - Simple
  - Application does not consume CPU cycles while waiting
- Disadvantages
  - Does not scale to large number of file descriptors

```
socket();
bind() ;
listen();
while
    accept();
    recv();
    send();
```

wait for `select()`

51

select Function Call

```
#include <sys/types.h>
#include <sys/socket.h>
int status = select(nfds, &readfds, &writefds,
    &exceptfds, &timeout);
```

- Provided a list of file descriptors to check the status if they are read-ready, write-ready, or have registered an exception
- Returns the number of ready objects, -1 if error
- nfds is 1 + largest file descriptor to check
- Uses bit-vector structure called fd_set to manage list of file descriptors
  - readfds — list of descriptors to check if read-ready
  - writefds — list of descriptors to check if write-ready
  - exceptfds — list of descriptors to check if an exception is registered
- timeout passed to indicate time after which select returns, even if no ready file descriptors

52

To Be Used With select

- select uses a structure struct fd_set
  - It is just a bit-vector
  - If bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select
  - FD_ZERO(&fdvar) clears the structure
  - FD_SET(i, &fdvar) to check file descriptor *i*
- After calling select
  - int FD_ISSET(i, &fdvar) boolean returns TRUE iff *i* is "ready"

53