CSCI 4061
Introduction to Operating Systems

Instructor: Abhishek Chandra

Outline

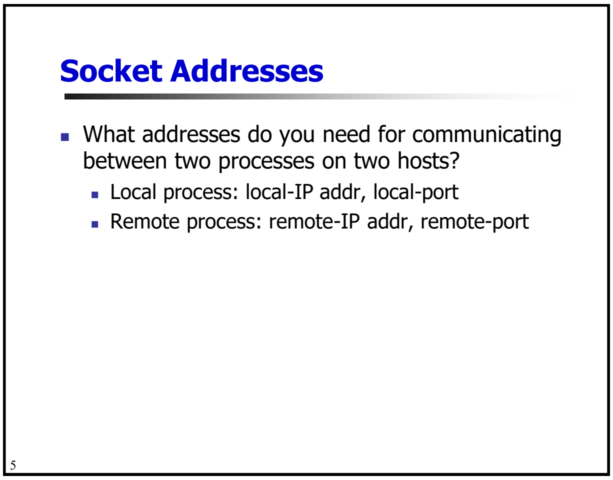
- Socket Overview
- Socket Types
- Socket Operations
- TCP Client-Server Sockets

What are Sockets?

- Networking API provided by the OS
  - Enable applications to “hook” onto the network
  - Support one end of a network connection
- Special files
  - Program is given socket file descriptors
  - Read, write data to them
  - Special operations needed

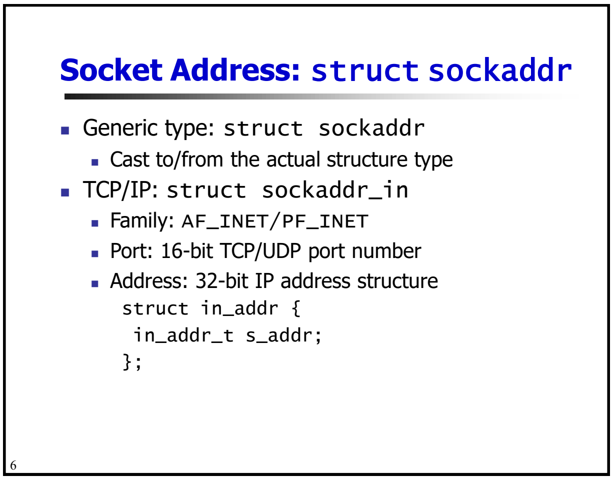
Socket Types

- Protocol families
  - E.g.: TCP/IP, UUCP, Appletalk
- TCP/IP Family:
  - Connection-oriented: TCP Sockets
  - Connectionless: UDP Sockets



Socket Addresses

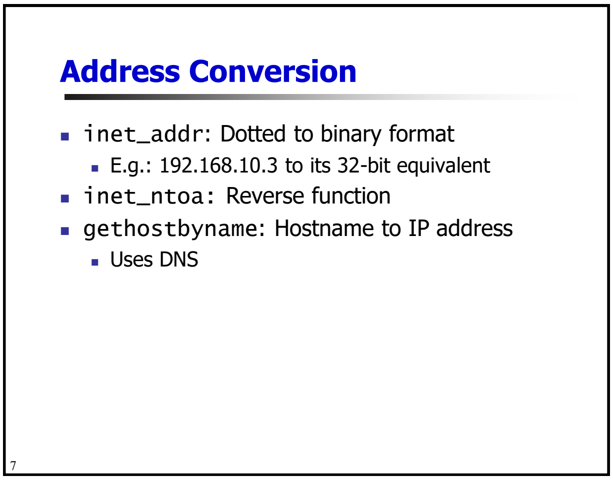
- What addresses do you need for communicating between two processes on two hosts?
  - Local process: local-IP addr, local-port
  - Remote process: remote-IP addr, remote-port



Socket Address: `struct sockaddr`

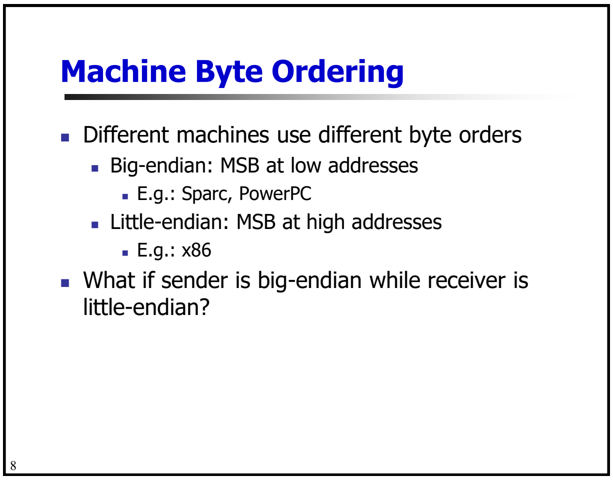
- Generic type: `struct sockaddr`
  - Cast to/from the actual structure type
- TCP/IP: `struct sockaddr_in`
  - Family: `AF_INET/PF_INET`
  - Port: 16-bit TCP/UDP port number
  - Address: 32-bit IP address structure

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



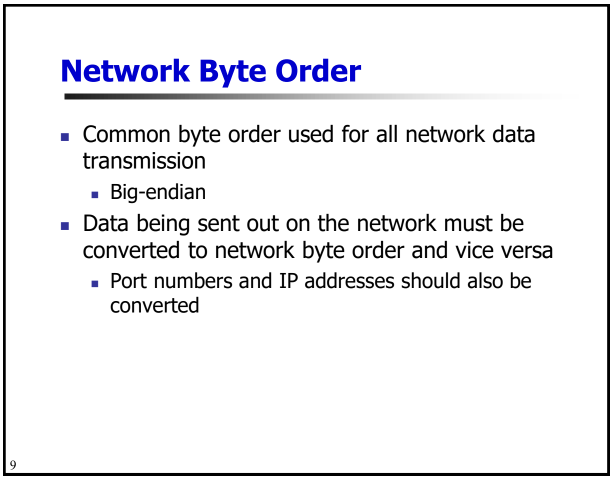
Address Conversion

- `inet_addr`: Dotted to binary format
  - E.g.: 192.168.10.3 to its 32-bit equivalent
- `inet_ntoa:` Reverse function
- `gethostbyname`: Hostname to IP address
  - Uses DNS



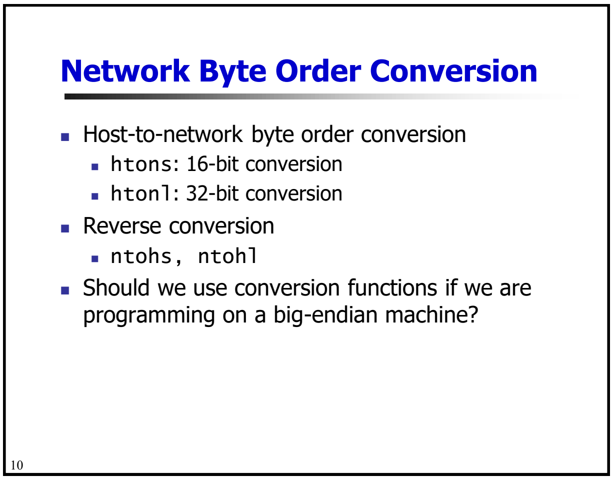
Machine Byte Ordering

- Different machines use different byte orders
  - Big-endian: MSB at low addresses
    - E.g.: Sparc, PowerPC
  - Little-endian: MSB at high addresses
    - E.g.: x86
- What if sender is big-endian while receiver is little-endian?



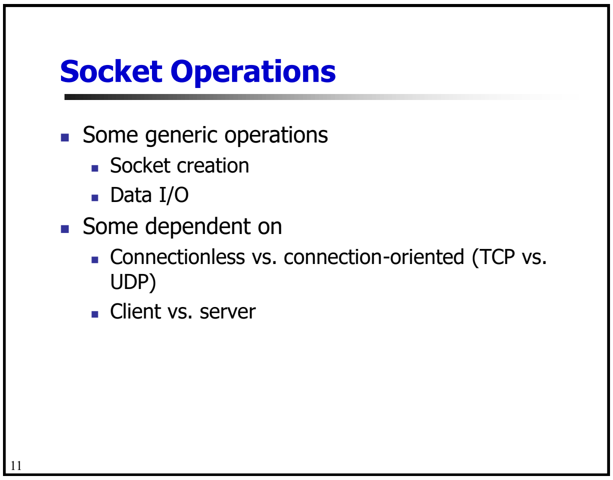
Network Byte Order

- Common byte order used for all network data transmission
  - Big-endian
- Data being sent out on the network must be converted to network byte order and vice versa
  - Port numbers and IP addresses should also be converted



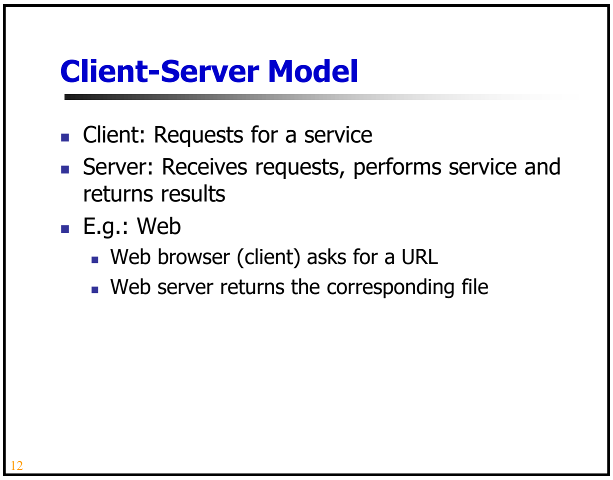
Network Byte Order Conversion

- Host-to-network byte order conversion
  - `htons`: 16-bit conversion
  - `htonl`: 32-bit conversion
- Reverse conversion
  - `ntohs, ntohl`
- Should we use conversion functions if we are programming on a big-endian machine?



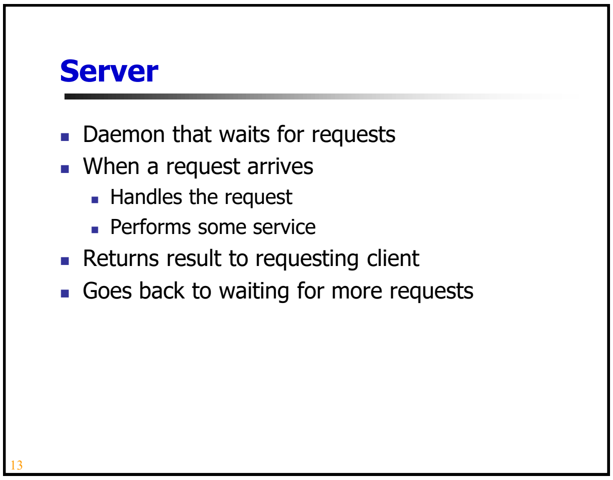
Socket Operations

- Some generic operations
  - Socket creation
  - Data I/O
- Some dependent on
  - Connectionless vs. connection-oriented (TCP vs. UDP)
  - Client vs. server



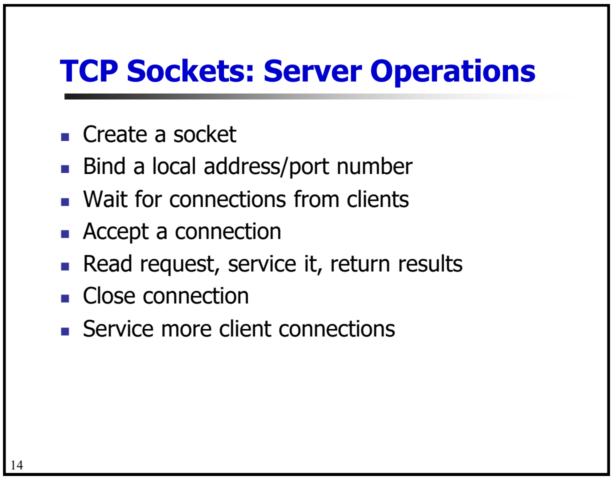
Client-Server Model

- Client: Requests for a service
- Server: Receives requests, performs service and returns results
- E.g.: Web
  - Web browser (client) asks for a URL
  - Web server returns the corresponding file



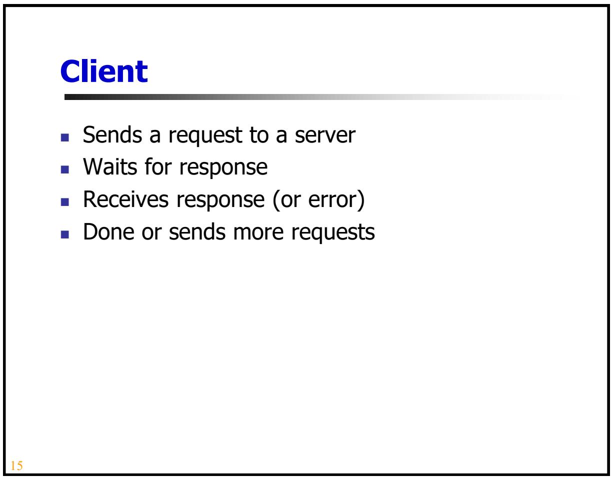
Server

- Daemon that waits for requests
- When a request arrives
  - Handles the request
  - Performs some service
- Returns result to requesting client
- Goes back to waiting for more requests



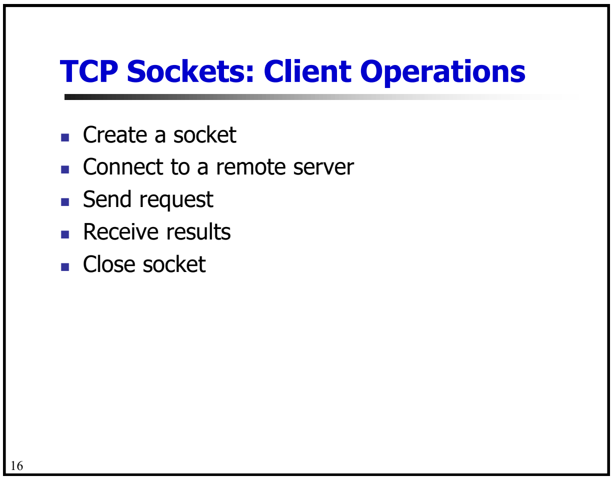
TCP Sockets: Server Operations

- Create a socket
- Bind a local address/port number
- Wait for connections from clients
- Accept a connection
- Read request, service it, return results
- Close connection
- Service more client connections



Client

- Sends a request to a server
- Waits for response
- Receives response (or error)
- Done or sends more requests



TCP Sockets: Client Operations

- Create a socket
- Connect to a remote server
- Send request
- Receive results
- Close socket

Creating a Socket: `socket`

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

- Returns a file descriptor
  - Identifier for the socket
- Parameters
  - `family`: Protocol family. E.g.: Internet or TCP/IP (`AF_INET`/`PF_INET`)
  - `type`: Protocol type
    - `SOCK_STREAM`: Stream (TCP)
    - `SOCK_DGRAM`: Datagram (UDP)
    - `SOCK_RAW`: Raw (IP)
  - `protocol`: Typically 0

Binding a Local Address/Port: `bind`

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

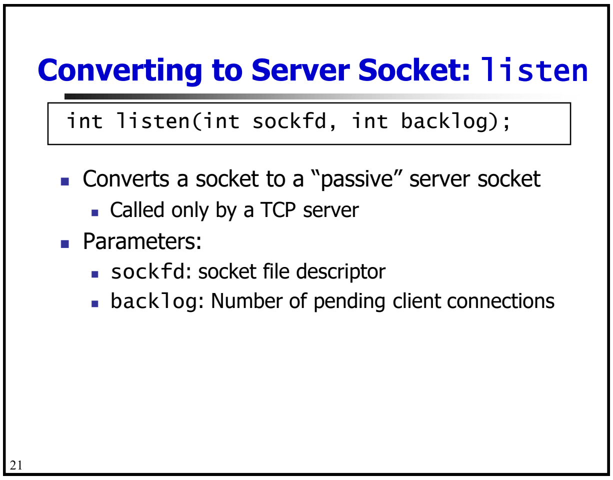
- Binds a local address/port based on values specified in the address structure
- Parameters:
  - `sockfd`: socket file descriptor
  - `myaddr`: address structure containing local address/port
  - `addrlen`: Length of address structure

Local Ports

- Well-known ports:
  - Process specifies a non-zero port number
  - Servers typically do this
  - E.g.: Web: 80, ftp: 21, ssh: 22
- Ephemeral ports:
  - Port number specified in address struct is 0
  - Kernel chooses an unused port number from a range
  - Clients typically do this

Local IP Addresses

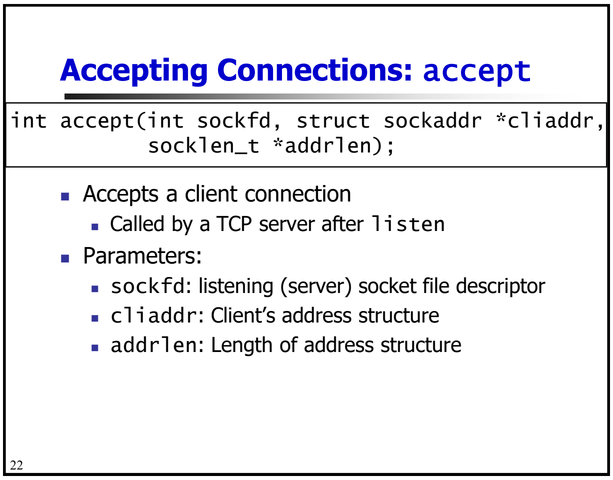
- Process may specify a local IP address
  - One of the valid network interface addresses
  - Communication would happen through the chosen address
- Process may specify a wildcard IP address
  - `INADDR_ANY`
  - Kernel will choose a default IP address



Converting to Server Socket: listen

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

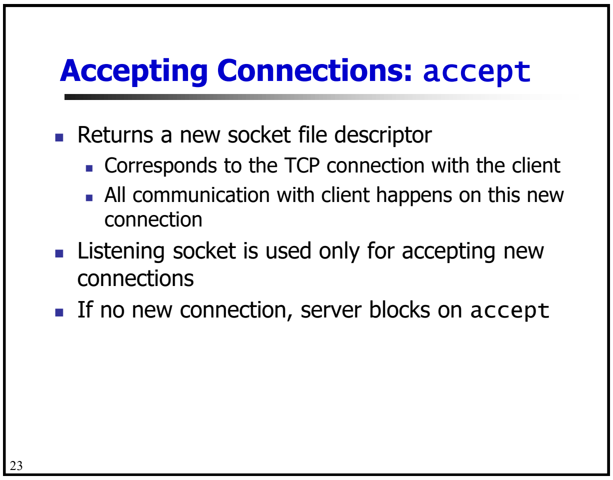
- Converts a socket to a “passive” server socket
  - Called only by a TCP server
- Parameters:
  - `sockfd`: socket file descriptor
  - `backlog`: Number of pending client connections



Accepting Connections: accept

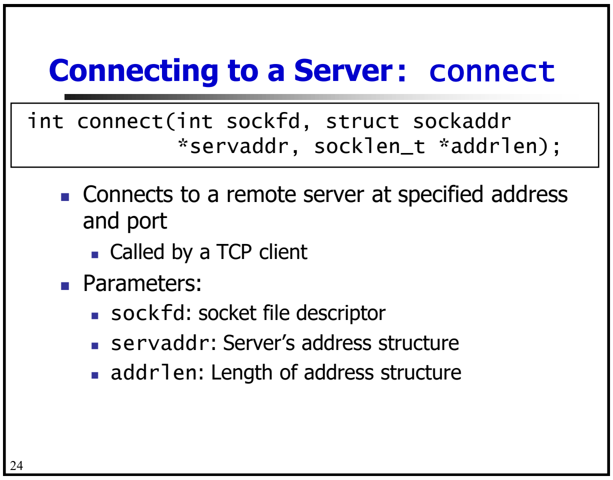
```
int accept(int sockfd, struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

- Accepts a client connection
  - Called by a TCP server after `listen`
- Parameters:
  - `sockfd`: listening (server) socket file descriptor
  - `cliaddr`: Client’s address structure
  - `addrlen`: Length of address structure



Accepting Connections: accept

- Returns a new socket file descriptor
  - Corresponds to the TCP connection with the client
  - All communication with client happens on this new connection
- Listening socket is used only for accepting new connections
- If no new connection, server blocks on `accept`



Connecting to a Server : connect

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

- Connects to a remote server at specified address and port
  - Called by a TCP client
- Parameters:
  - `sockfd`: socket file descriptor
  - `servaddr`: Server’s address structure
  - `addrlen`: Length of address structure

TCP Socket I/O

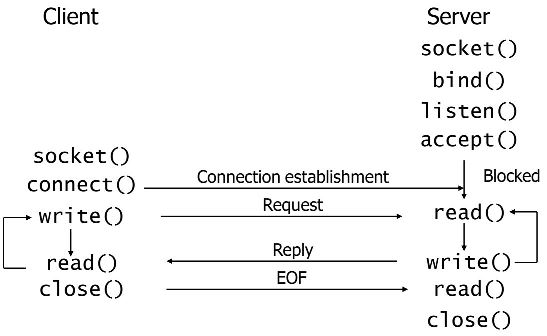
- `read, write`
  - Used like for regular file I/O
- Remember that the return value may be different than num_bytes specified
  - May need to call read/write multiple times
- Avoid using stream operations such as `fprintf, fread,` etc.
  - Buffering may cause problems

Closing a Connection: `close`

```
int close(int sockfd);
```

- Similar to file close
- In addition:
  - Closes TCP connection
  - Sends out any pending data before closing

TCP Client-Server Operations



Client

Server

`socket()`
`bind()`
`listen()`
`accept()`

`socket()`
`connect()`

Connection establishment

Blocked

`write()`

Request

`read()`

`read()`

Reply

`write()`

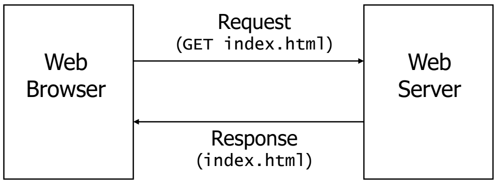
`close()`

EOF

`read()`

`close()`

TCP Client-Server Example: Web



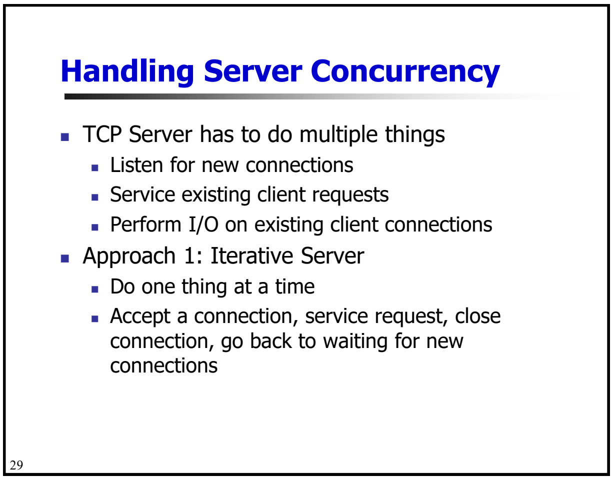
Web Browser

Request
(`GET index.html`)

Response
(`index.html`)

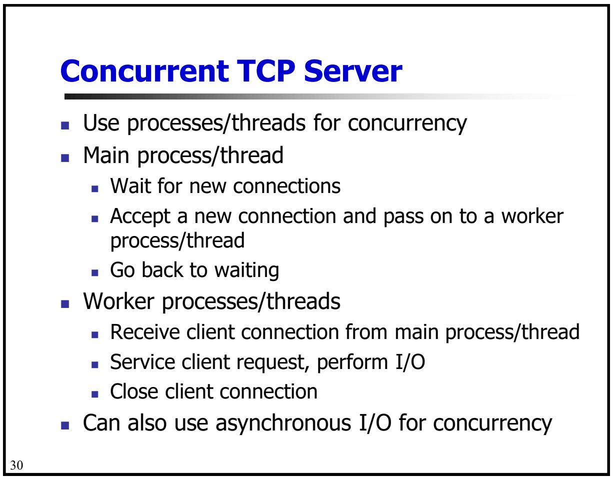
Web Server

- Which of the client-server operations are:
  - The same?
  - Specific to Web application?



Handling Server Concurrency

- TCP Server has to do multiple things
  - Listen for new connections
  - Service existing client requests
  - Perform I/O on existing client connections
- Approach 1: Iterative Server
  - Do one thing at a time
  - Accept a connection, service request, close connection, go back to waiting for new connections



Concurrent TCP Server

- Use processes/threads for concurrency
- Main process/thread
  - Wait for new connections
  - Accept a new connection and pass on to a worker process/thread
  - Go back to waiting
- Worker processes/threads
  - Receive client connection from main process/thread
  - Service client request, perform I/O
  - Close client connection
- Can also use asynchronous I/O for concurrency