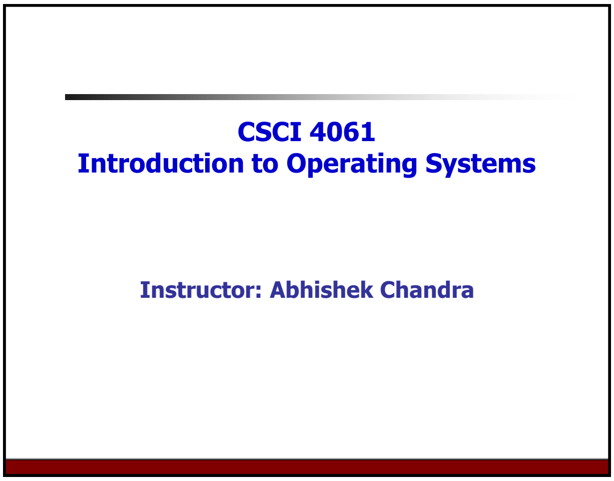
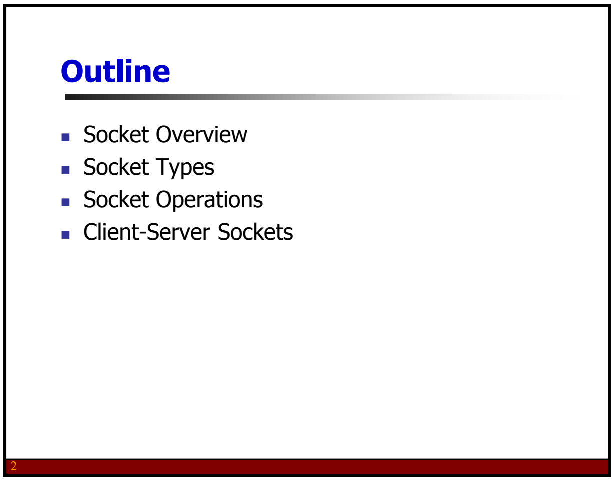


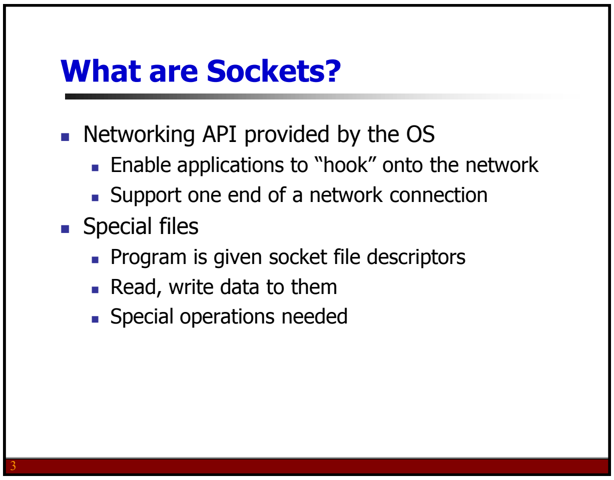
CSCI 4061
Introduction to Operating Systems

Instructor: Abhishek Chandra



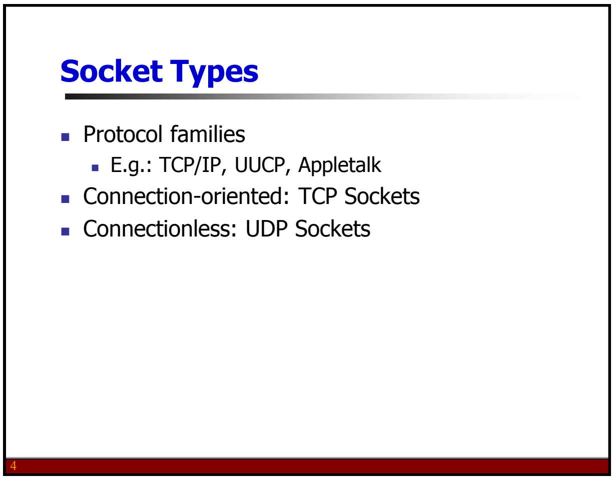
Outline

- Socket Overview
- Socket Types
- Socket Operations
- 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
- Connection-oriented: TCP Sockets
- Connectionless: UDP Sockets

Socket Addresses

- What addresses do you need for communicating between two processes on two hosts?
- Each socket is associated with a 5-tuple
  - `{protocol, local-addr, local-port, remote-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
  - Opposite of x86 machine architecture
- 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
  - Why?

Network Byte Order Conversion

- Host-to-network byte order conversion
  - `htons`: 16-bit conversion
  - `htonl`: 32-bit conversion
- Reverse conversion
  - `ntohs, ntohl`
- Why don't we have `htonc, ntohc`?
- Should we use conversion functions if we are programming on a big-endian machine?

Socket Operations

- Dependent on
  - Connectionless vs. connection-oriented
  - Client vs. server
- Some generic operations
  - Socket creation
  - Data I/O

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 client connection

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

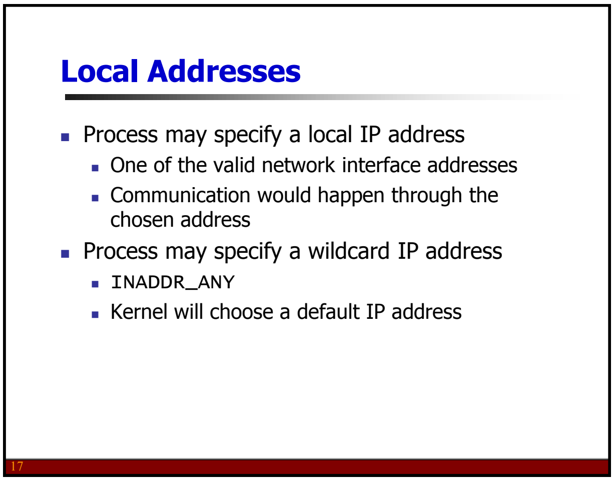
Binding a Local Address/Port: `bind`

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

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

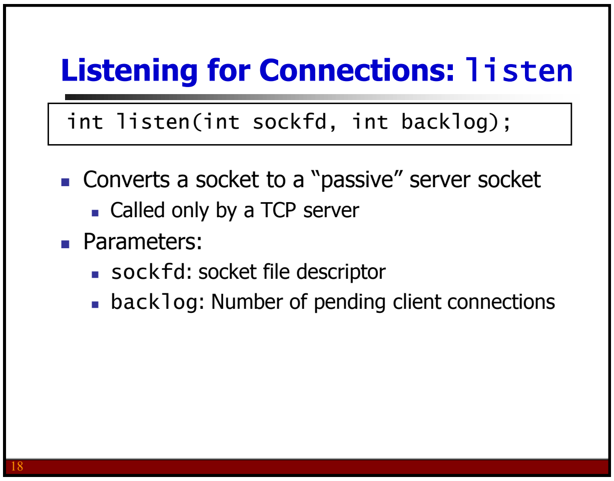
Local Ports

- 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 Addresses

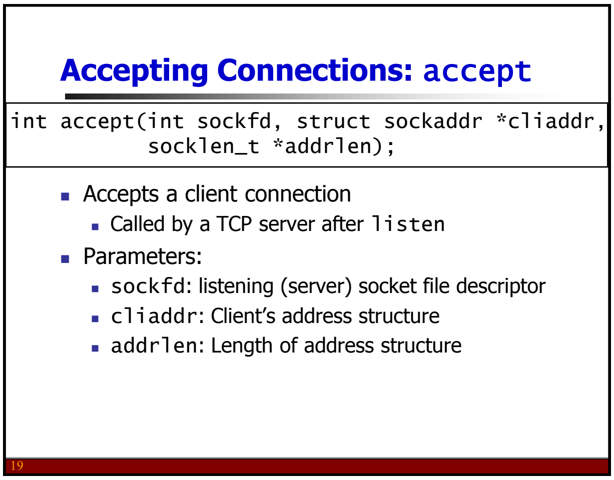
- Process may specify a local IP address
  - One of the valid network interface addresses
  - Communication would happen through the chosen address
- Process may specify a wildcard IP address
  - `INADDR_ANY`
  - Kernel will choose a default IP address



Listening for Connections: `listen`

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

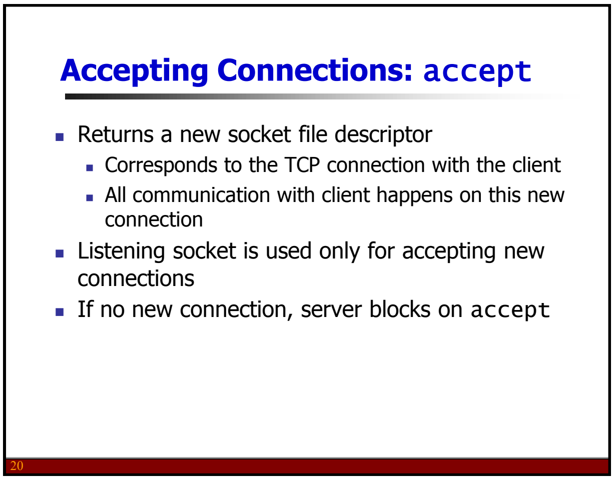
- Converts a socket to a "passive" server socket
  - Called only by a TCP server
- Parameters:
  - `sockfd`: socket file descriptor
  - `backlog`: Number of pending client connections



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);
```

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

TCP Socket I/O

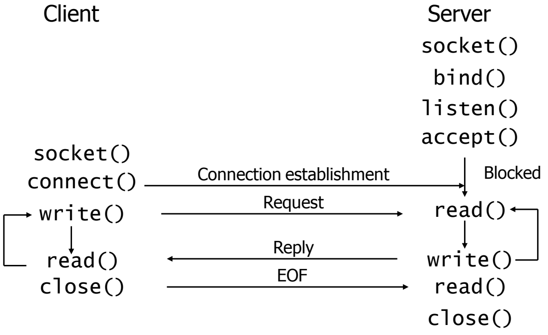
- `read, write`
- Remember that the return value may be different than num_bytes specified
- Should not use stream operations such as `fprintf, fread,` etc.
  - Buffering may cause problems

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()
write()
read()
close()
```

Connection establishment

Blocked

Request

```
read()
```

Reply

```
write()
```

EOF

```
read()
close()
```

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

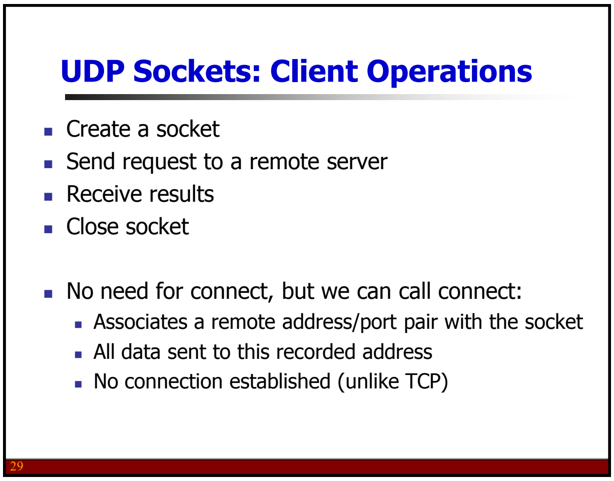
UDP Sockets

- UDP is connectionless
- No connection established between client-server
- Data is transmitted as datagrams instead of stream of bytes

UDP Sockets: Server Operations

- Create a socket
- Bind a local address/port number
- Wait for requests from clients
- Read request, service it, return results

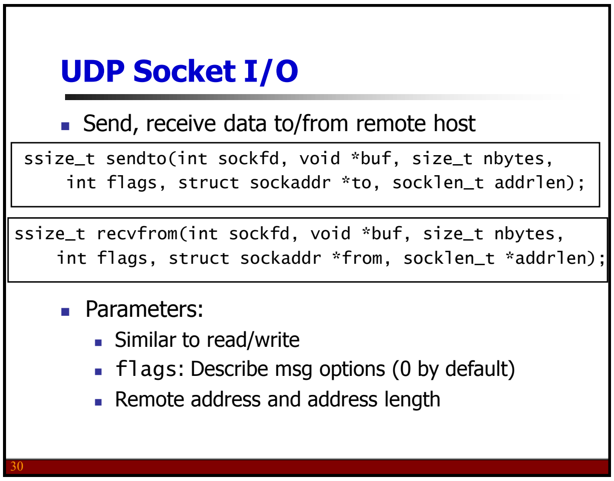
- No need for listen, accept



UDP Sockets: Client Operations

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

- No need for connect, but we can call connect:
  - Associates a remote address/port pair with the socket
  - All data sent to this recorded address
  - No connection established (unlike TCP)



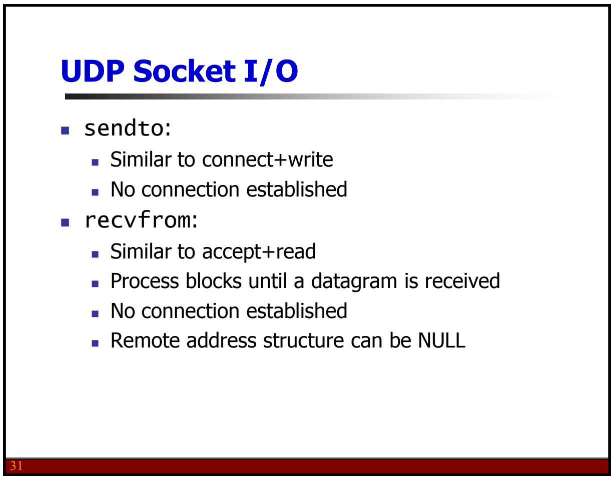
UDP Socket I/O

- Send, receive data to/from remote host

```
ssize_t sendto(int sockfd, void *buf, size_t nbytes,
    int flags, struct sockaddr *to, socklen_t addrlen);
```

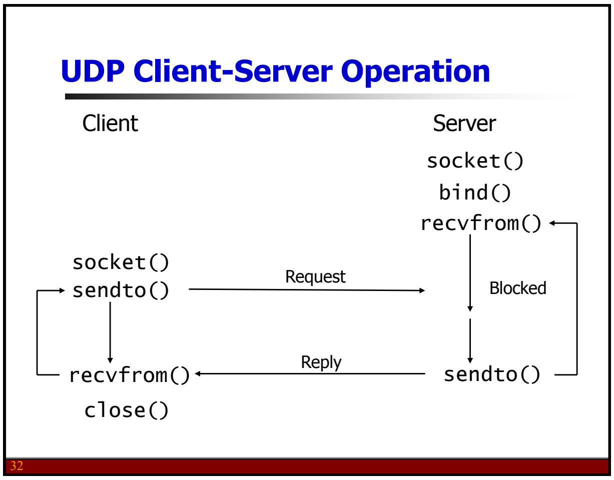
```
ssize_t recvfrom(int sockfd, void *buf, size_t nbytes,
    int flags, struct sockaddr *from, socklen_t *addrlen);
```

- Parameters:
  - Similar to read/write
  - `flags`: Describe msg options (0 by default)
  - Remote address and address length

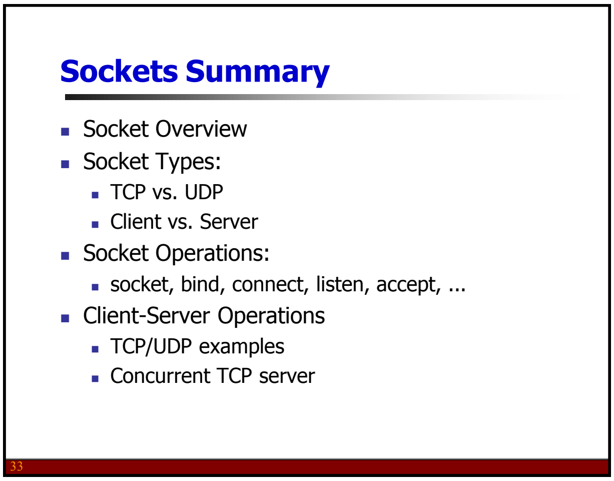


UDP Socket I/O

- `sendto`:
  - Similar to connect+write
  - No connection established
- `recvfrom`:
  - Similar to accept+read
  - Process blocks until a datagram is received
  - No connection established
  - Remote address structure can be NULL



UDP Client-Server Operation



Sockets Summary

- Socket Overview
- Socket Types:
  - TCP vs. UDP
  - Client vs. Server
- Socket Operations:
  - socket, bind, connect, listen, accept, ...
- Client-Server Operations
  - TCP/UDP examples
  - Concurrent TCP server