Week 12 Lecture 1

XMUT-NWEN 241 - 2024 T2

Systems Programming

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

Admin

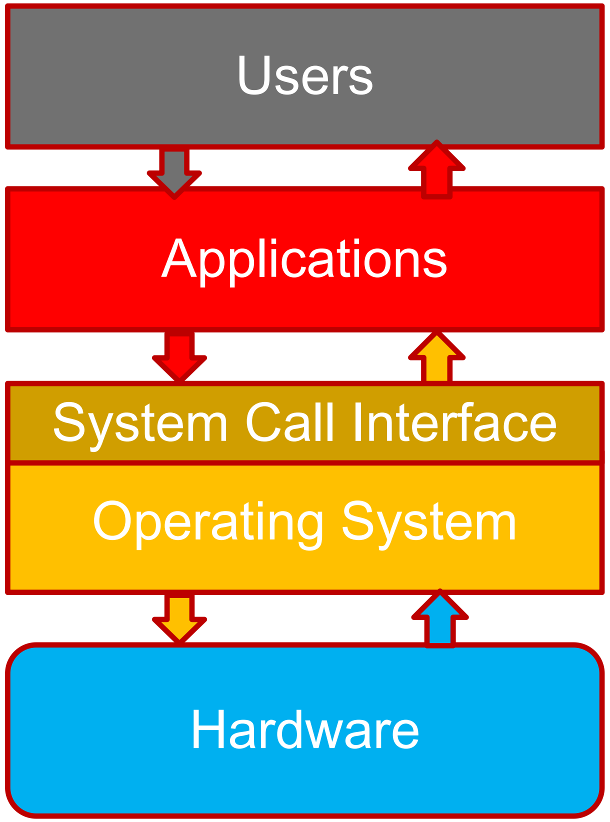
- Exercise #3
  - Due date: 1 December

Content

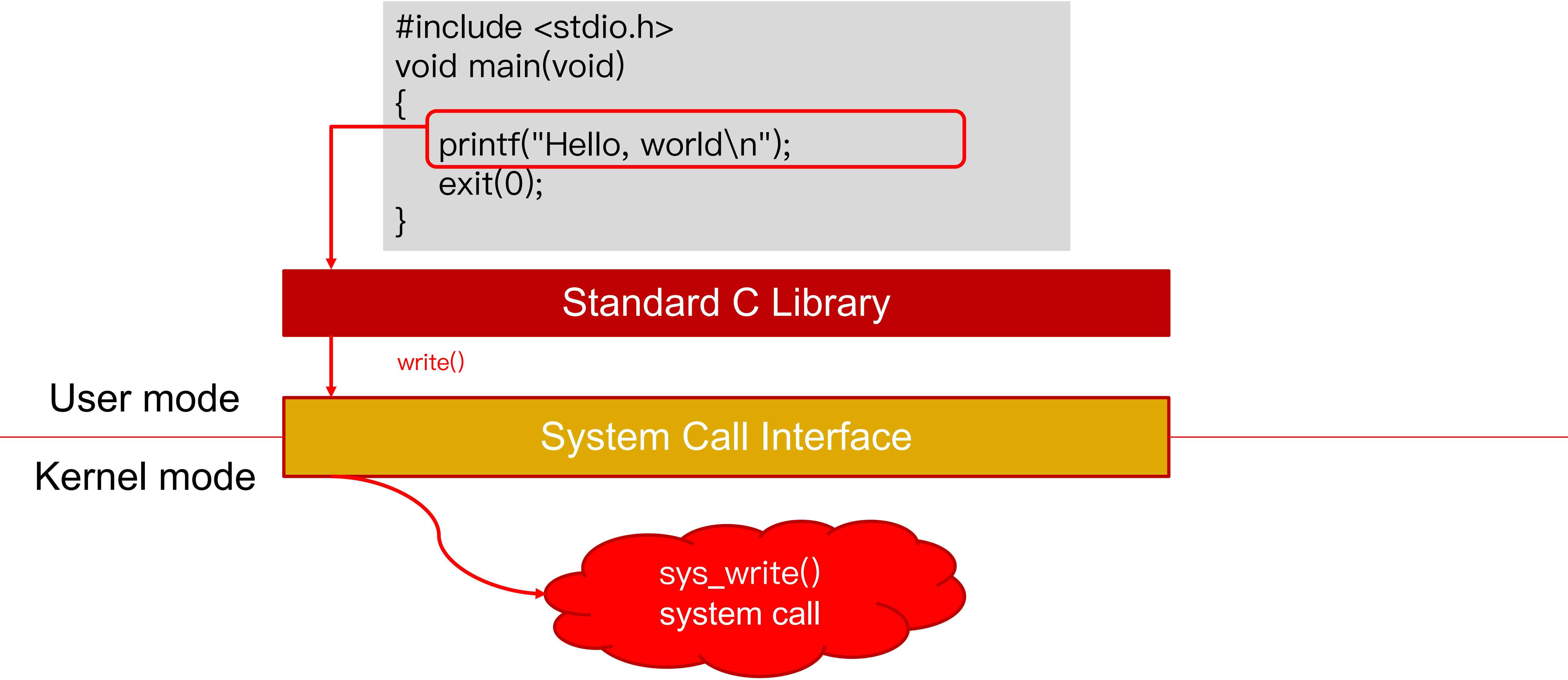
- Interprocess communication
  - TCP Socket Programming

Recall: System calls - What and Why?

- Operating Systems do not **allow application software to access system resources** directly due to security and reliability issues.
- A program can **request** the services of system resources from OS through **system calls**.
- **System calls** are **function invocations made from application into the OS** in order to request some service or resource from the operating system.
- Application developers often do not have direct access to system calls but can access them through a **system call API**, which in turn invokes the system call.



Recall: System call invocation –*Example*



```
#include <stdio.h>
void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```

Standard C Library

write()

User mode

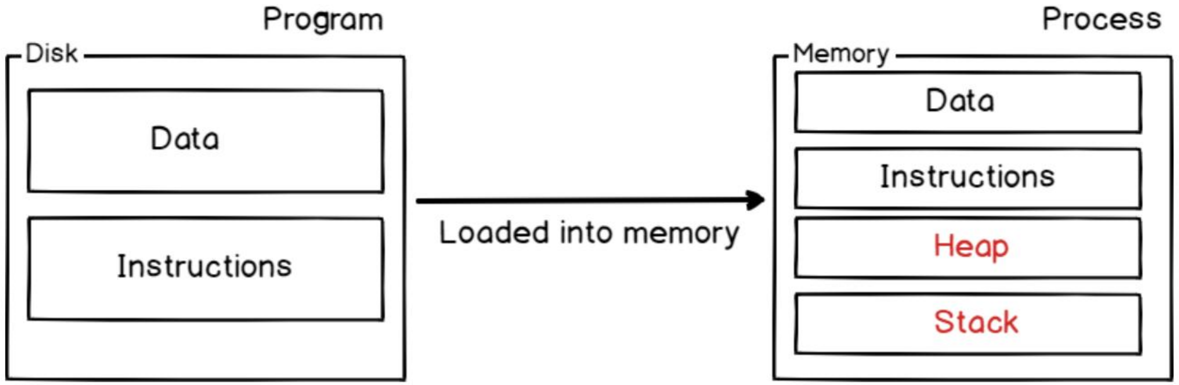
System Call Interface

Kernel mode

sys_write()
system call

Recap: What is a process ?

- Program and process are related terı

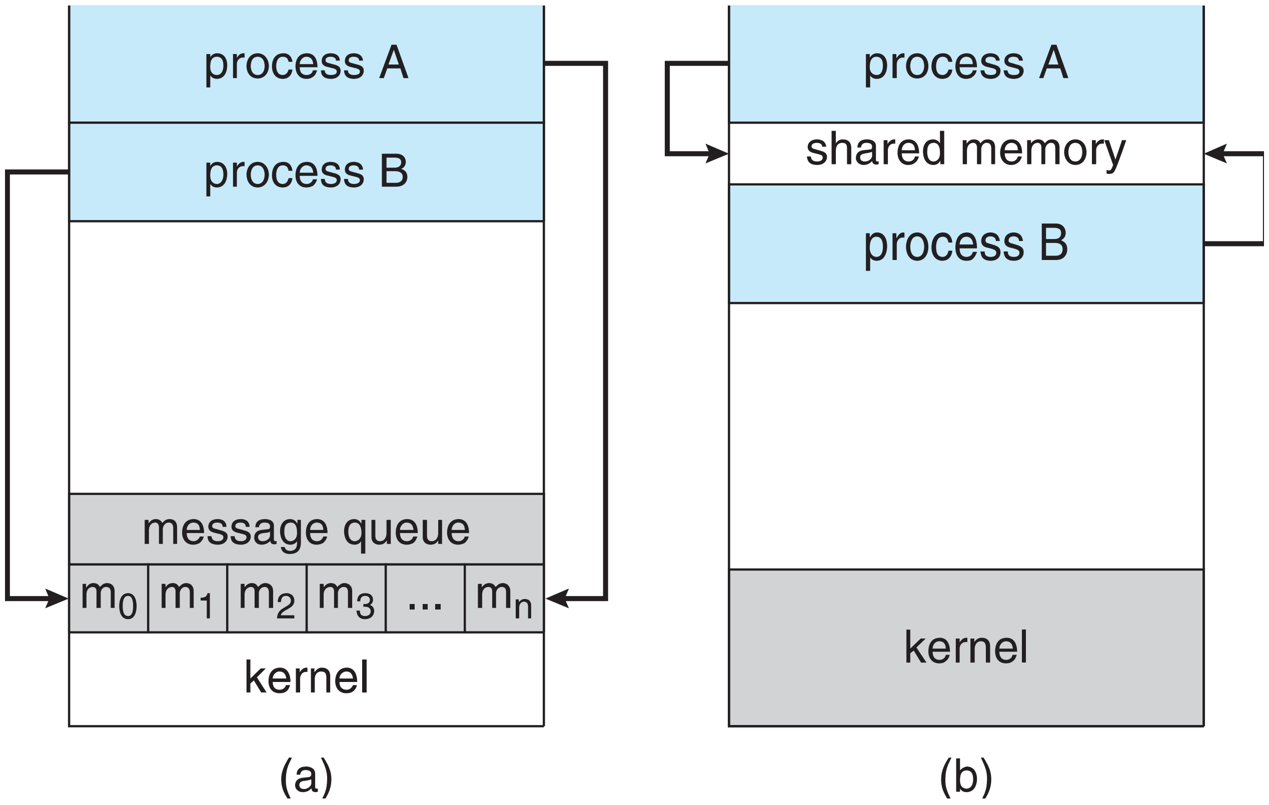


Program is a set of instructions to carry out a specified task	**Process** is a program in execution

Passive entity	Active entity
Program is a stored in disk and does not require any other resource.	**Process** requires system resources such as CPU, memory, I/O etc.
Life span - Longer	Life span – limited
Each time a program is run a new process is created.	

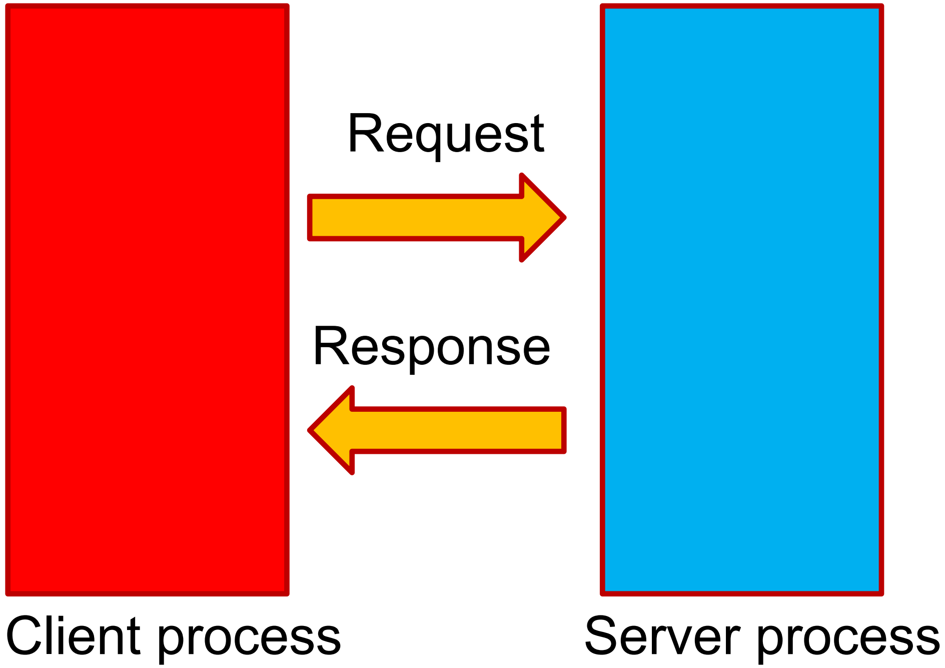
Recap: Interprocess communication

- Cooperating processes need **interprocess communication** (**IPC**)
- Two primary models of IPC
  - **Message passing**
  - **Shared memory**



Client-server model

- Most common IPC paradigm
- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client

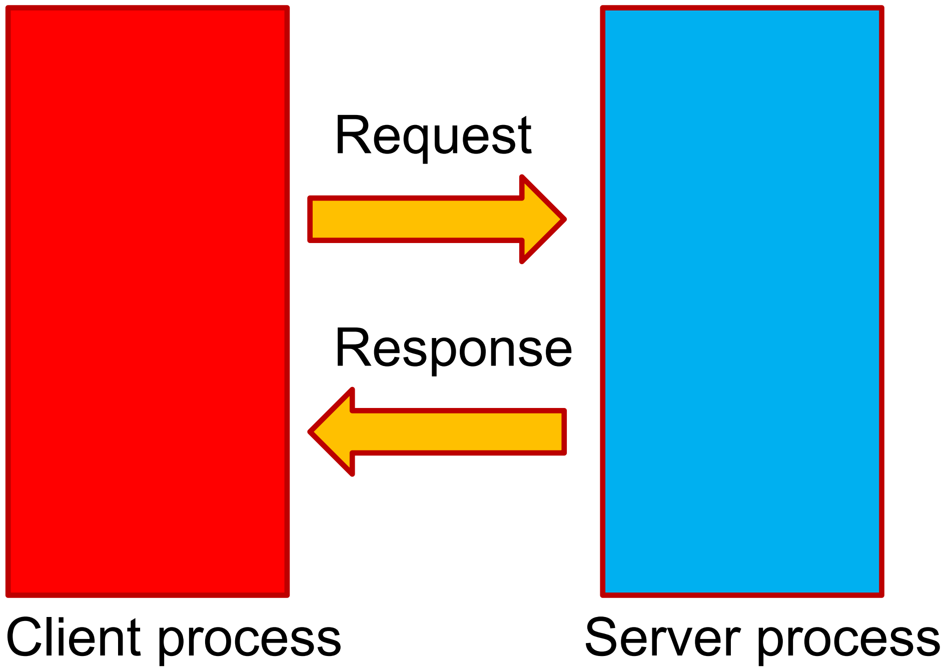


Client-server model

- **Client process** needs to know the existence and the address of the server

- However, the **Server** does not need to know the existence or address of the client prior to the connection

- **Once a connection** is established, both sides can send and receive information



Side Note: How to know which system calls are invoked?

Two commands:

a) **ltrace** – traces call to library functions

b) **strace** -traces system calls

- Details in Linux manual pages :

 - >Open terminal -> write **man <*command-name*>**

Example: **man** *ltrace*

- Usage : **ltrace ./<program executable file>**

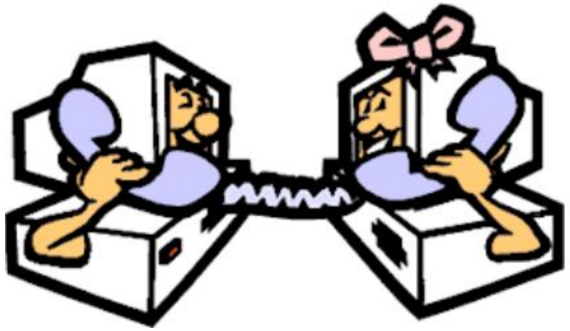
  ltrace –S ./<program executable file> (also display system calls)

Client-server communication

- Remote Procedure Calls
- Pipes
- Sockets

What is socket?

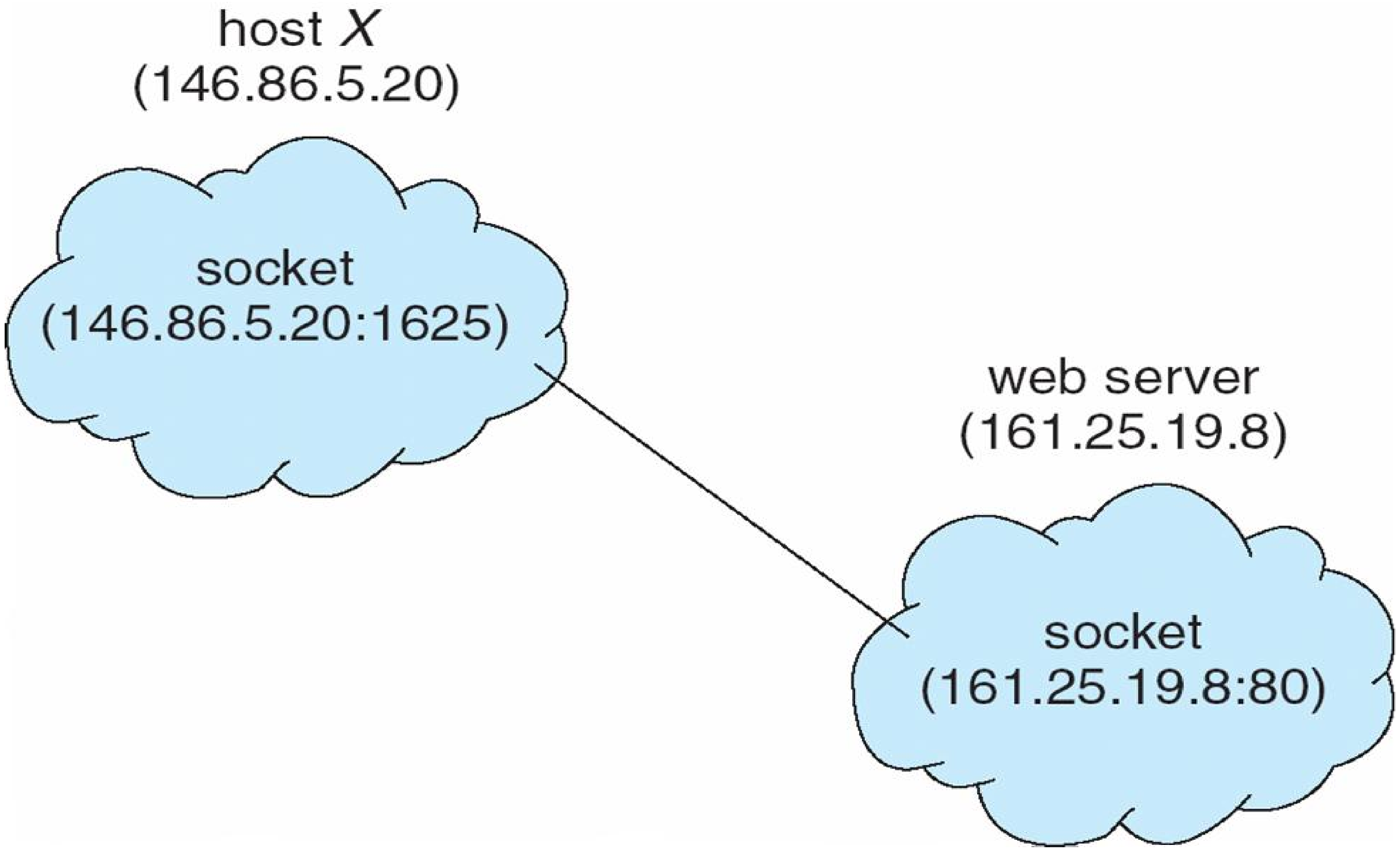
- What do we need to know to allow two processes on a network to communicate?
  - **Identity of the communicating machines**
    - **IP Address**
  - **Identity of the communicating processes on these machines**
    - **Port**



- Concatenation of **IP address** and **port** defines a **socket** - A **socket** is defined as an endpoint for communication

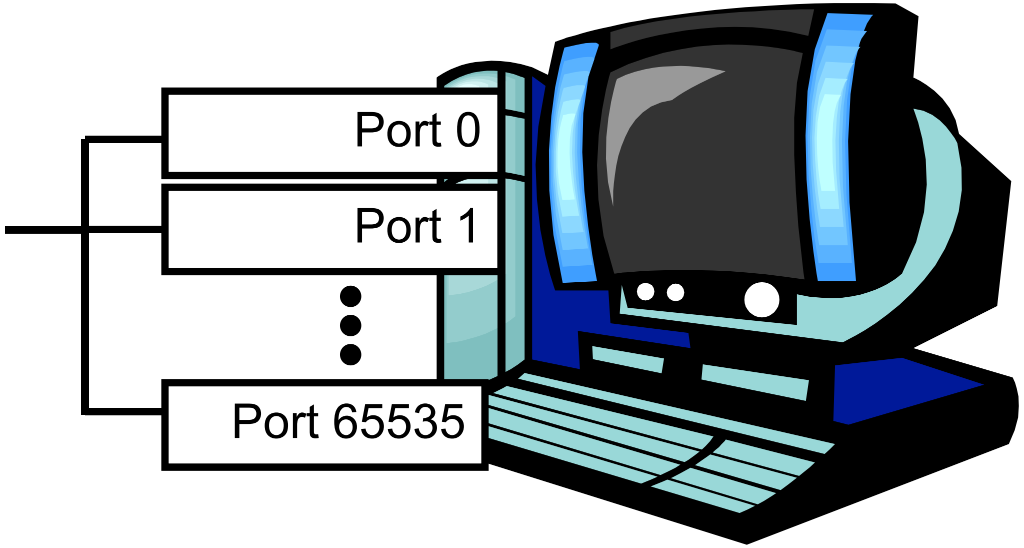
- Example: The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

Socket communication



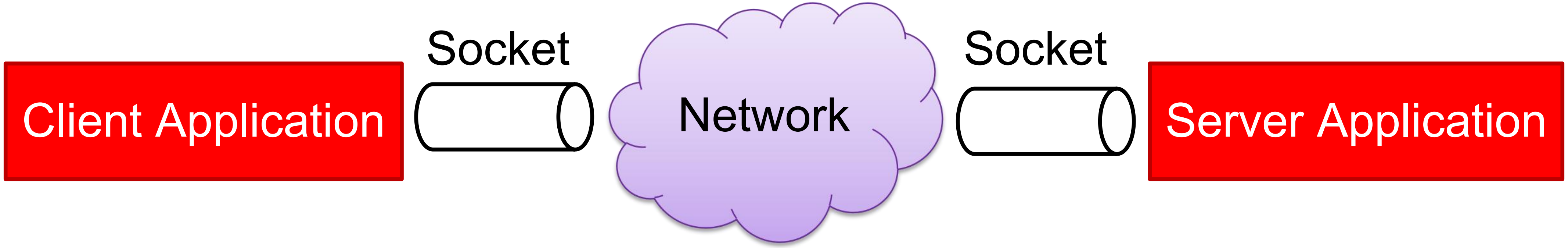
Port numbers

- Each host has **65,536** ports
- Use of ports 1-1023 requires privileges
- Some ports are reserved for specific apps
  - 20, 21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



Sockets as programming interface

- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
  - TCP VS UDP
    - reliable vs. best effort
    - connection-oriented vs. connectionless



Socket types

- SOCK_STREAM
  - a.k.a. **TCP**
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

- SOCK_DGRAM
  - a.k.a. **UDP**
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

We will focus on SOCK_STREAM or TCP socket type

System calls

- socket()
- bind()
- listen()
- accept()
- connect()
- send() / sendto()
- recv() / recvfrom()

Include

sys/types.h

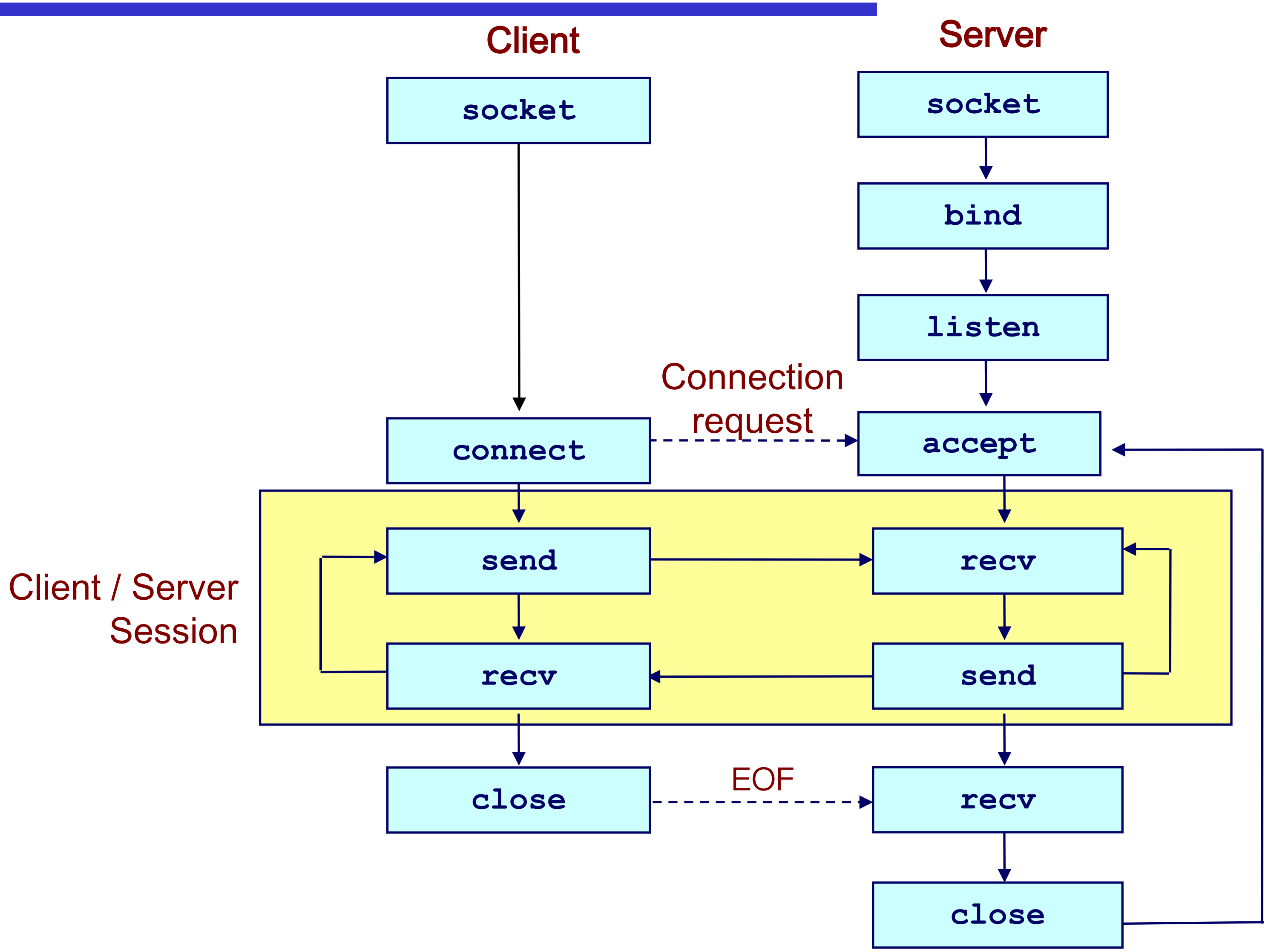
sys/socket.h

TCP Client overview

1) Create a socket with the socket() system call

2) Connect the socket to the address of the server using the connect() system call

3) Send and receive data



TCP Server overview

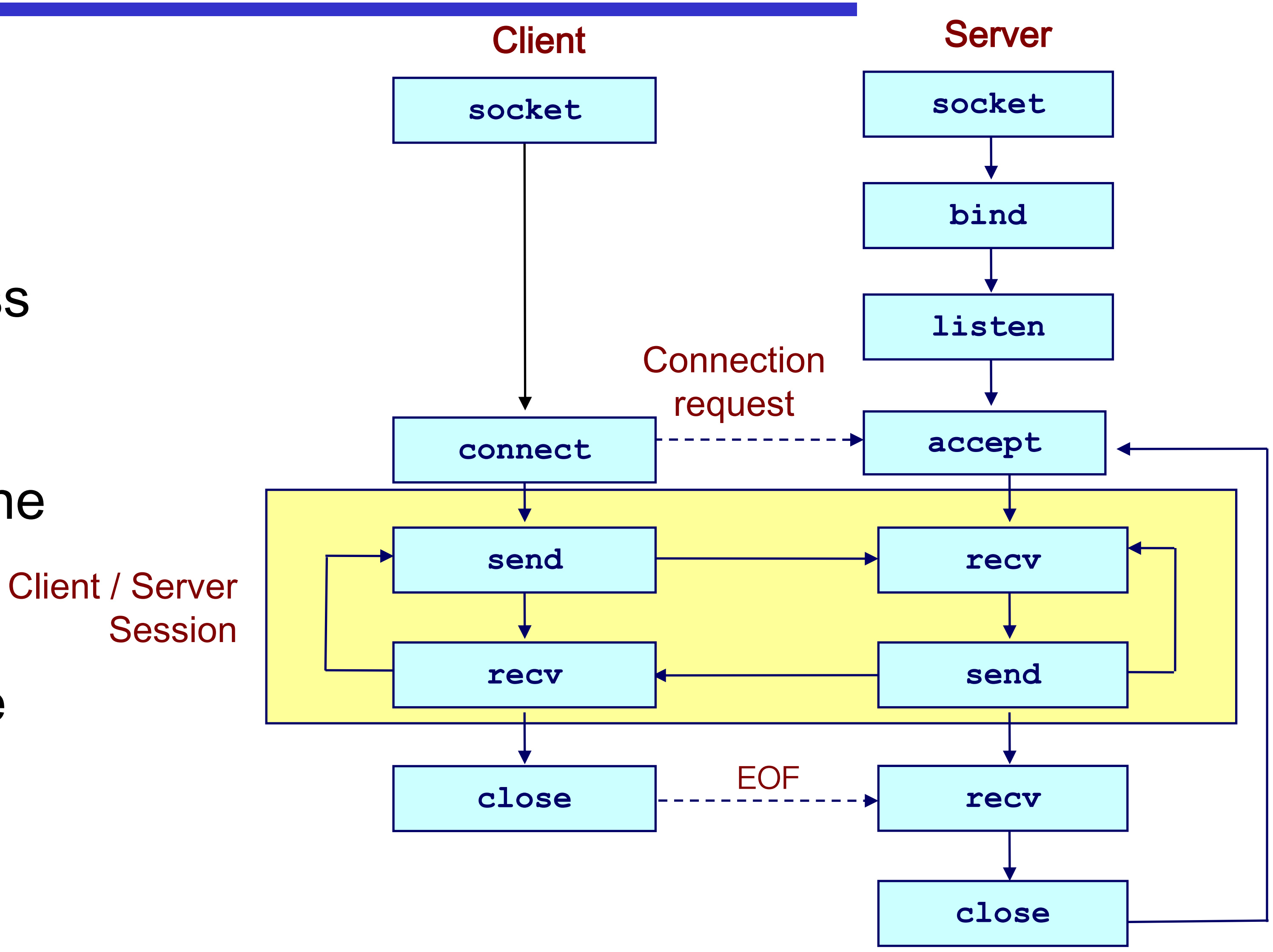
1) Create a socket with the socket() system call

2) Bind the socket to an address using the bind() system call

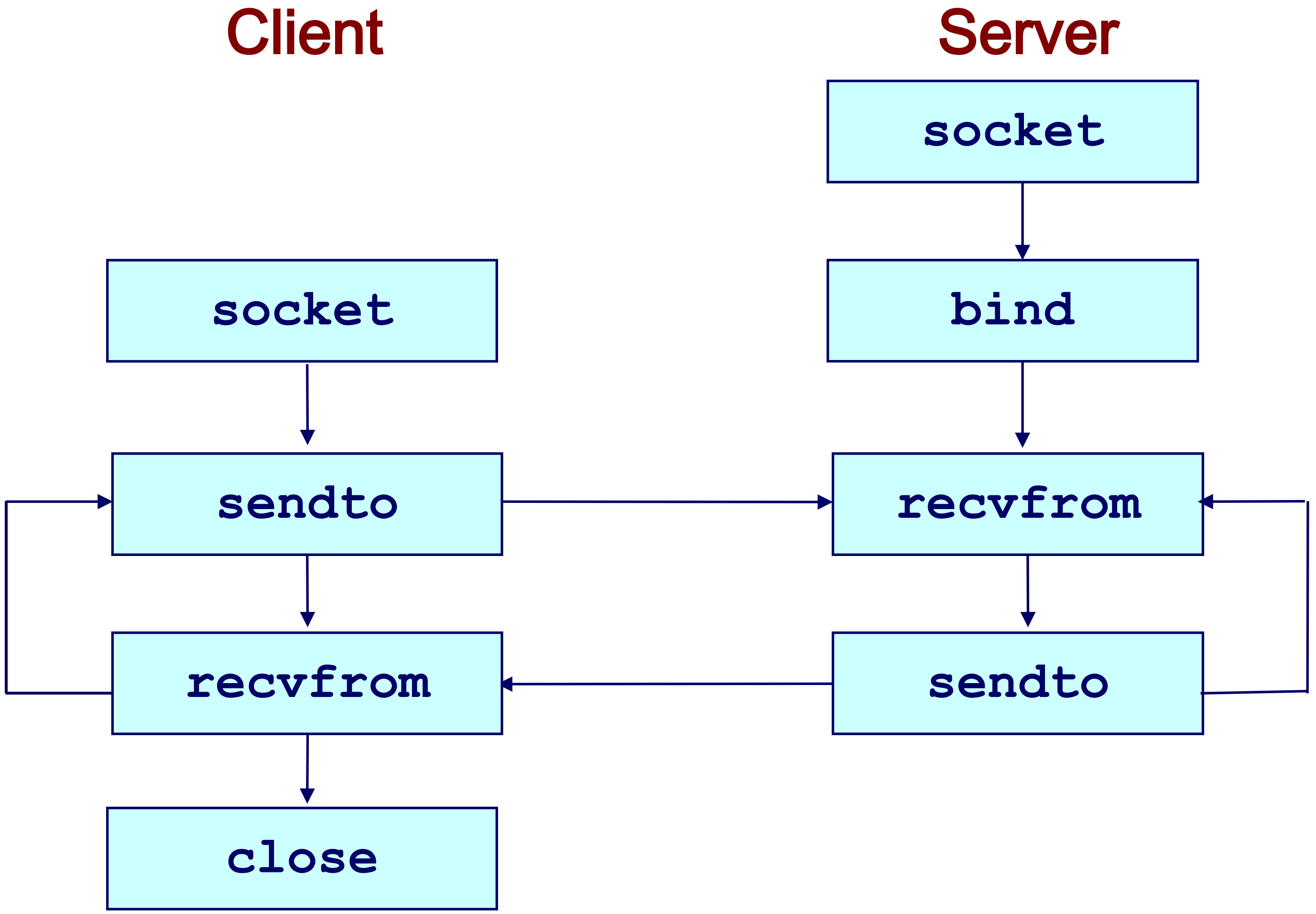
3) Listen for connections with the listen() system call

4) Accept a connection with the accept() system call

5) Send and receive data



Client-server communication overview - UDP



Server: step 1

- Create a socket with the socket() system call

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

  - *domain* – communication domain (protocol family) such as AF_INET (IPv4) or AF_INET6 (IPv6)
  - *type* – communication semantics such as SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
  - *protocol* specifies the protocol, usually 0.

  - Creates an endpoint of communication.
  - If successful, returns **socket file descriptor**, otherwise, returns -1

Server: step 1 example

- Create TCP socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd == –1) {
    printf("Error creating socket");
    exit(0);
}
```

- Create UDP socket

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd == –1) {
    printf("Error creating socket");
    exit(0);
}
```

Server: step 2

- Bind the socket to an address using the bind() system call

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

  - *sockfd* is the socket file descriptor (returned by socket())
  - *addr* is a pointer to the structure *struct sockaddr* (generic data type for address) which contains the host IP address and port number to bind to
  - *addrlen* is the length of what addr points to

  - Binding means associating and reserving a port number for use by the socket
  - If successful, returns 0, otherwise, returns -1

struct sockaddr

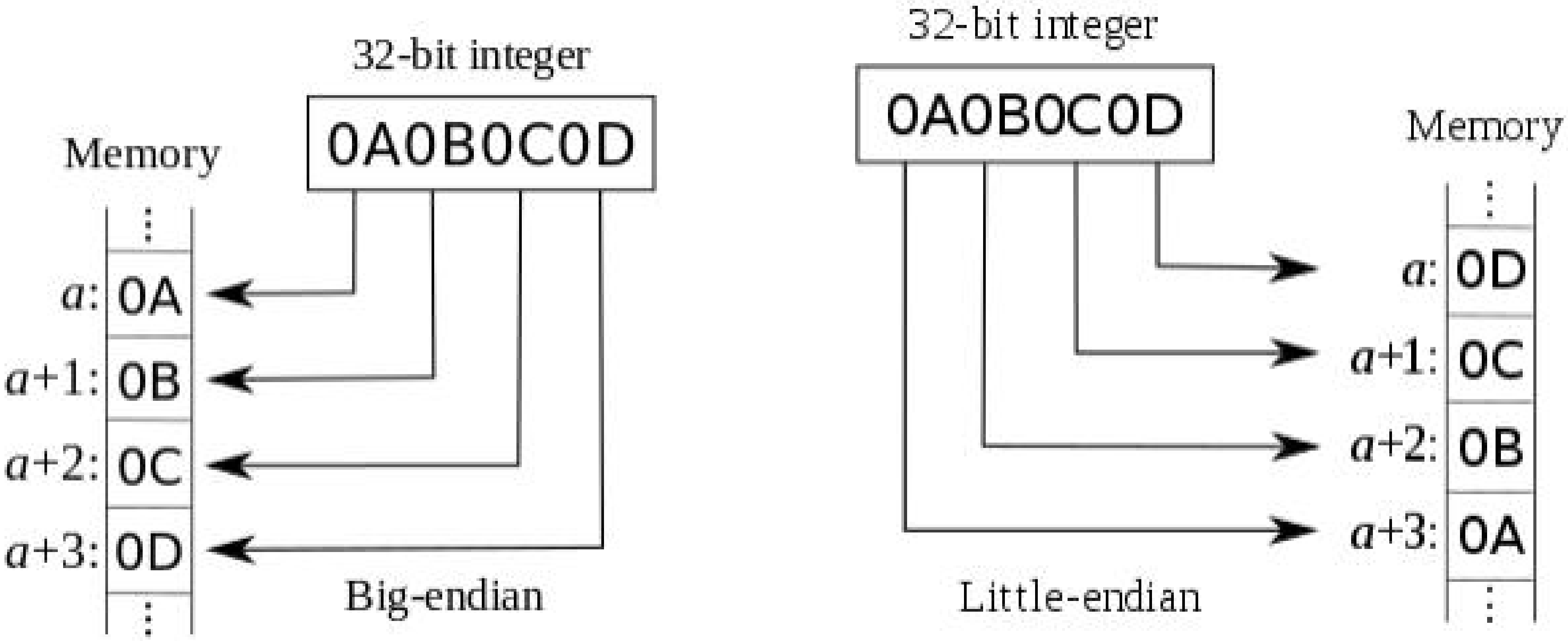
- struct sockaddr_in in IPv4 (included the <netinet/in.h> header)

```
struct sockaddr_in {
    short sin_family;          // AF_INET
    unsigned short sin_port;  // port number
    struct in_addr sin_addr;  // Internet address in
                              //network byte order
};

struct in_addr {
    unsigned long s_addr;     // IPv4 address in network
                             //byte order
};
```

Host and network byte order

- Little-endian and big-endian issue?



Host and network byte order

- Byte ordering also matters in network communication
  - Host and network may differ in byte ordering
  - Host byte order may be little-endian or big-endian
  - Network byte order is always big-endian

- Functions for converting between host and network byte order:

```
uint32_t htonl(uint32_t hostlong);  \\host to network long
uint16_t htons(uint16_t hostshort); \\host to network short
uint32_t ntohl(uint32_t netlong);   \\network to host long
uint16_t ntohs(uint16_t netshort);  \\network to host short
```

long is 32 bits.
short is 16 bits.

Server: step 2 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(1234);  // port 1234

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket");
    exit(0);
}
```

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
};

struct in_addr {
    unsigned long s_addr;
};
```

Server: step 3

- Listen for connections with the listen() system call

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

  - *sockfd* is the socket file descriptor (returned by socket())
  - *backlog* is the maximum number of pending connections to allow for this socket
    - SOMAXCONN is defined as the number of maximum pending connections allowed by the operating system

  - If successful, returns 0, otherwise, returns -1

Server: step 3 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

if(listen(fd, SOMAXCONN) < 0) {
    printf("Error listening for connections");
    exit(0);
}
```

Server: step 4

- Accept a connection with the accept() system call

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

  - *sockfd* is the socket file descriptor (returned by socket())
  - *addr* is a pointer to the structure struct sockaddr which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what addr points to

  - If successful, returns non-negative **socket file descriptor**, otherwise, returns -1

Server: step 4 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
struct sockaddr_in client_addr;
int addrlen = sizeof(client_addr);

int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       (socklen_t*)&addrlen);
if(client_fd < 0) {
    printf("Error accepting connection");
    exit(0);
}
```