SUNG KYUN KWAN UNIVERSITY

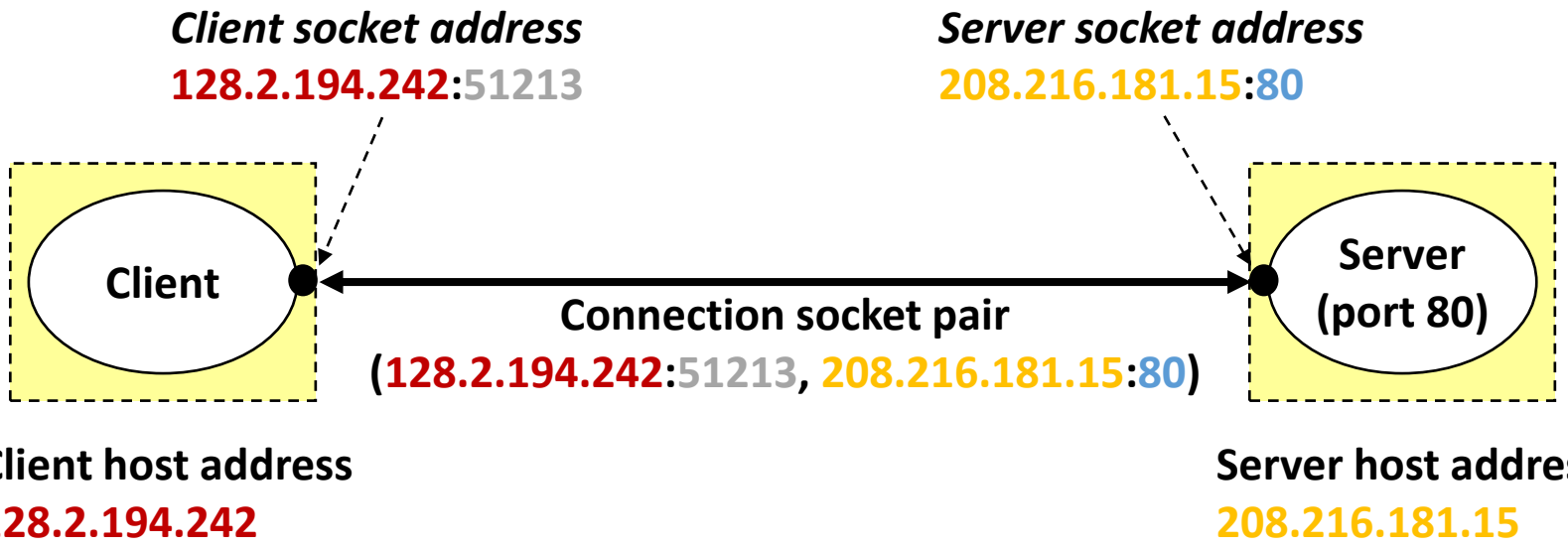
Socket

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Internet Connection (1)

- Clients and servers communicate by sending streams of bytes over connections:
  - Point-to-point, full-duplex, and reliable
- A **socket** is an endpoint of a connection
  - Socket address is an <IP address:port number> pair
- A port is a 16-bit integer that identifies a process
  - **Ephemeral port:** assigned automatically on client when client makes a connection request
  - **Well-known port:** associated with some service provided by a server (e.g. port 80 is associated with web servers)
- A connection is uniquely identified by the socket addresses of its endpoints (**socket pair**)
  - <client IP:client port no., server IP:server port no.>

Internet Connection (2)

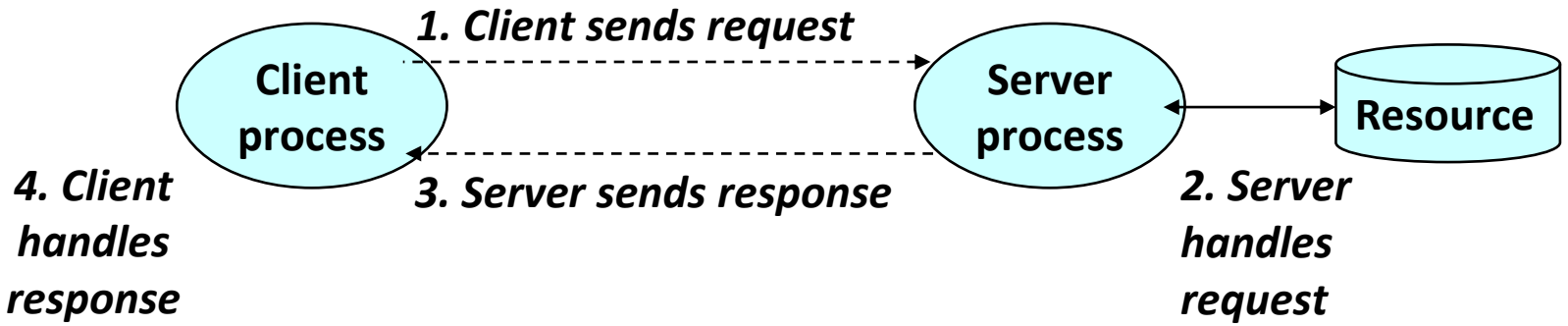


Note: 51213 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers

Client-Server Model

- Most network application is based on the client-server model
  - A server process and one or more client processes
    - Clients and servers are processes running on hosts (can be the same or different hosts)
  - Server manages some resources
  - Server provides service by manipulating resource for clients



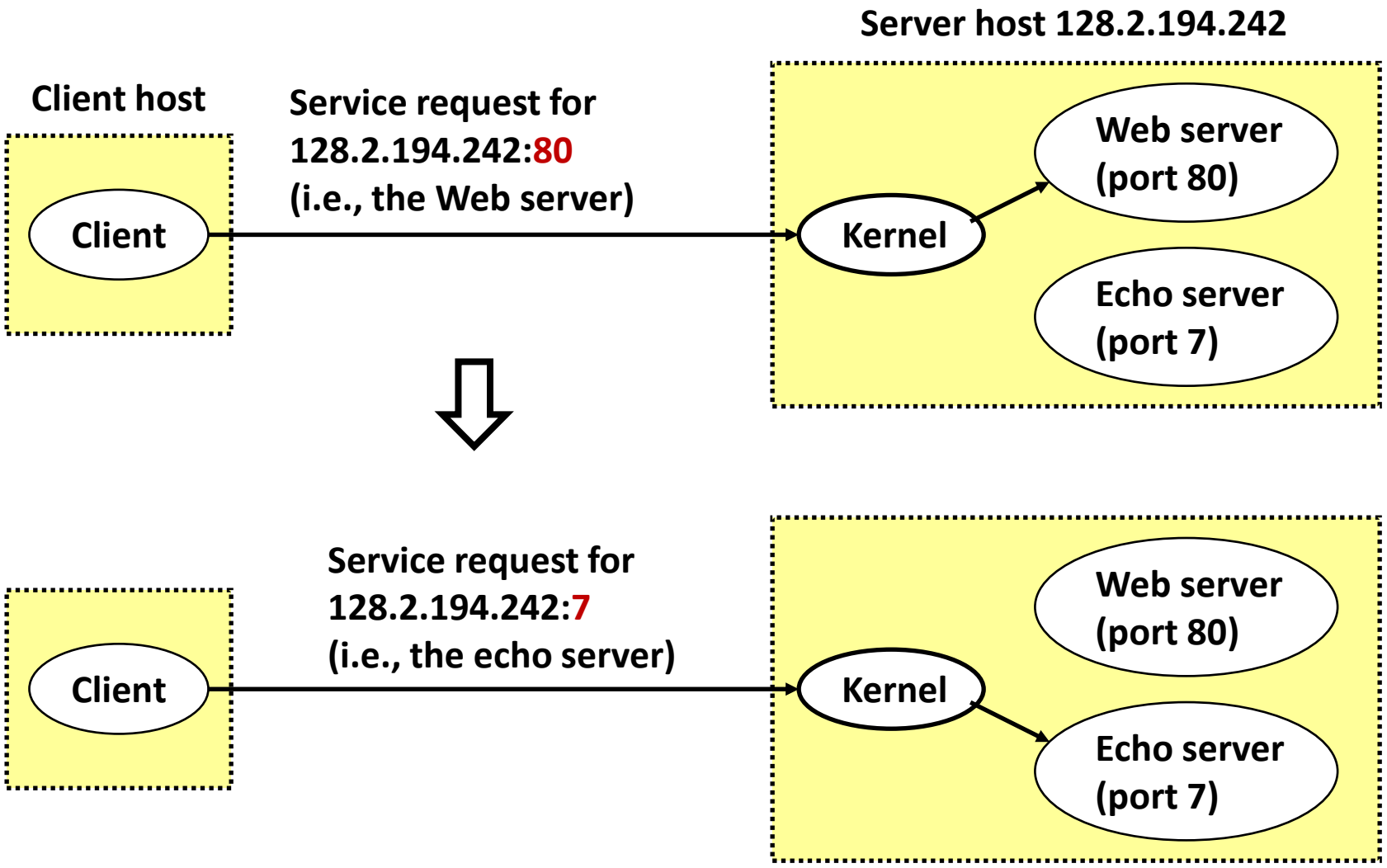
Clients

- Examples of client programs
  - Web browsers, FTP, telnet, ssh, etc.

- How does a client find the server?
  - **IP address** in server socket address identifies the host
  - The **(well-known) port** in server socket address identifies the service; thus, port implicitly identifies the server process that performs the service
  - Examples of well-known ports (cf. **/etc/services**)
    - Port 21: FTP
    - Port 23: telnet
    - Port 25: mail
    - Port 80: web

```
ftp     21/tcp
fsp     21/udp      fspd
ssh     22/tcp              # SSH Remote Login Protocol
telnet      23/tcp
smtp        25/tcp      mail
time        37/tcp      timserver
time        37/udp      timserver
whois       43/tcp      nicname
finger      79/tcp
http        80/tcp      www     # WorldWideWeb HTTP
ntp     123/udp             # Network Time Protocol
```

Using Ports


Server host 128.2.194.242
Client host
Service request for
128.2.194.242:80
(i.e., the Web server)
Client
Kernel
Web server
(port 80)
Echo server
(port 7)
Service request for
128.2.194.242:7
(i.e., the echo server)
Client
Kernel
Web server
(port 80)
Echo server
(port 7)

Servers

- Servers are long-running processes (**daemons**)
  - Created at boot-time (typically) by the *init* process (process 1)
  - Run continuously until the machine is turned off
- Each server waits for requests to arrive on a well-known port associated with a particular service
  - e.g., port 21: FTP server, port 80: HTTP server
- A machine that runs a server process is often referred to as “server”

Socket (1)

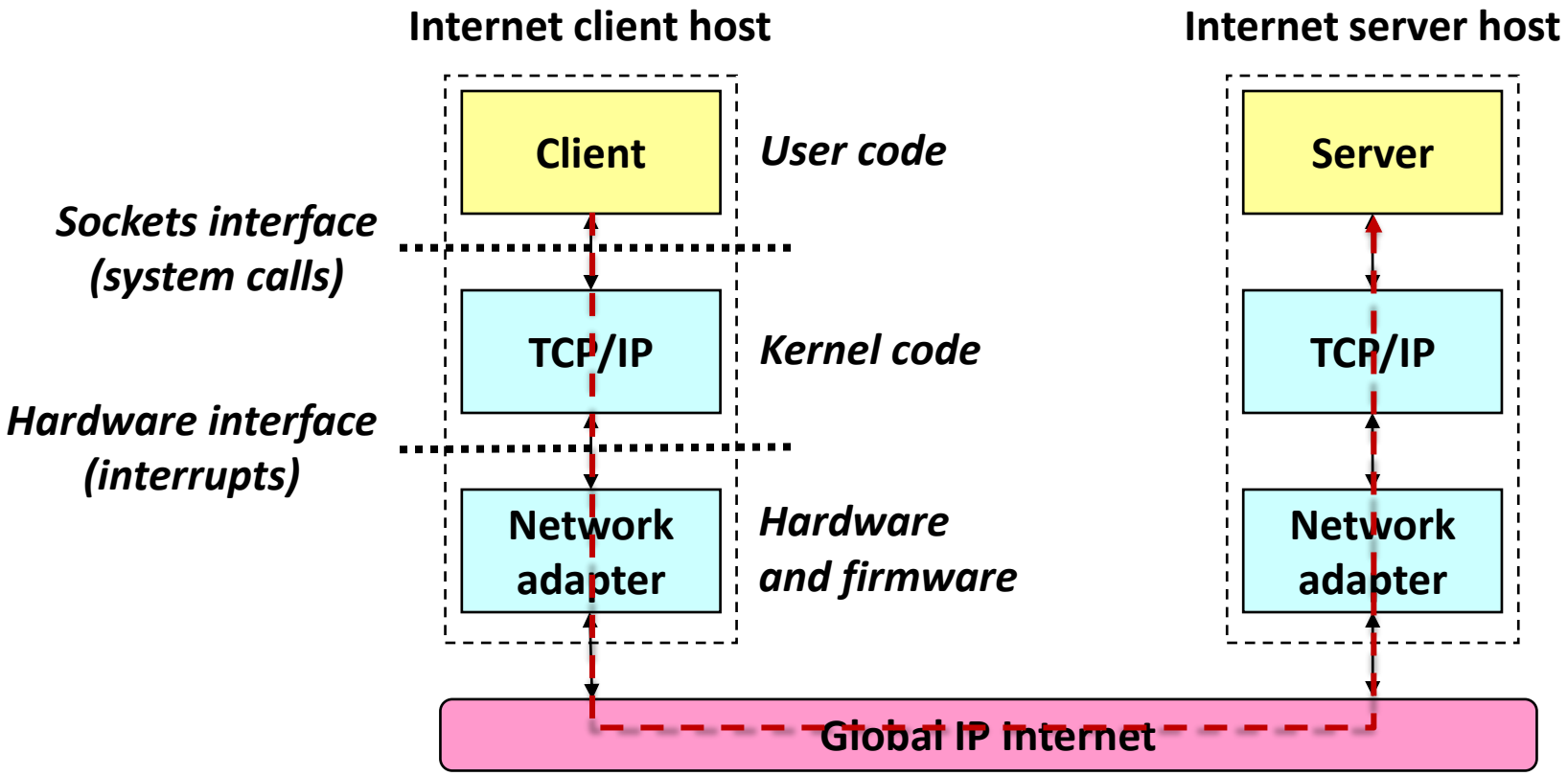
- Socket interface (circa 1981)
  - Was Introduced in BSD4.1 UNIX, 1981
  - Is based on **client-server paradigm**
  - Provides a user-level interface to the network
  - Can be explicitly created, used, released by applications
  - Consists of two types of transport service
    - Reliable, connection-oriented byte stream
    - Unreliable datagram
  - Is the underlying basis for all Internet applications

Socket (2)

- What is a **socket**?
  - Remote-local, application-created/owned, OS-controlled interface to network ("door")
    - To the kernel, socket is an endpoint of communication
    - To the application, socket is a file descriptor
      - Applications read/write from/to the network **using the file descriptor**
      - Note: All Unix I/O devices, including networks, are modeled **as files**
  - Clients and servers communicate with each other by reading from and writing to socket descriptors
    - The main distinction between regular file I/O and socket I/O is how applications "open" socket descriptors

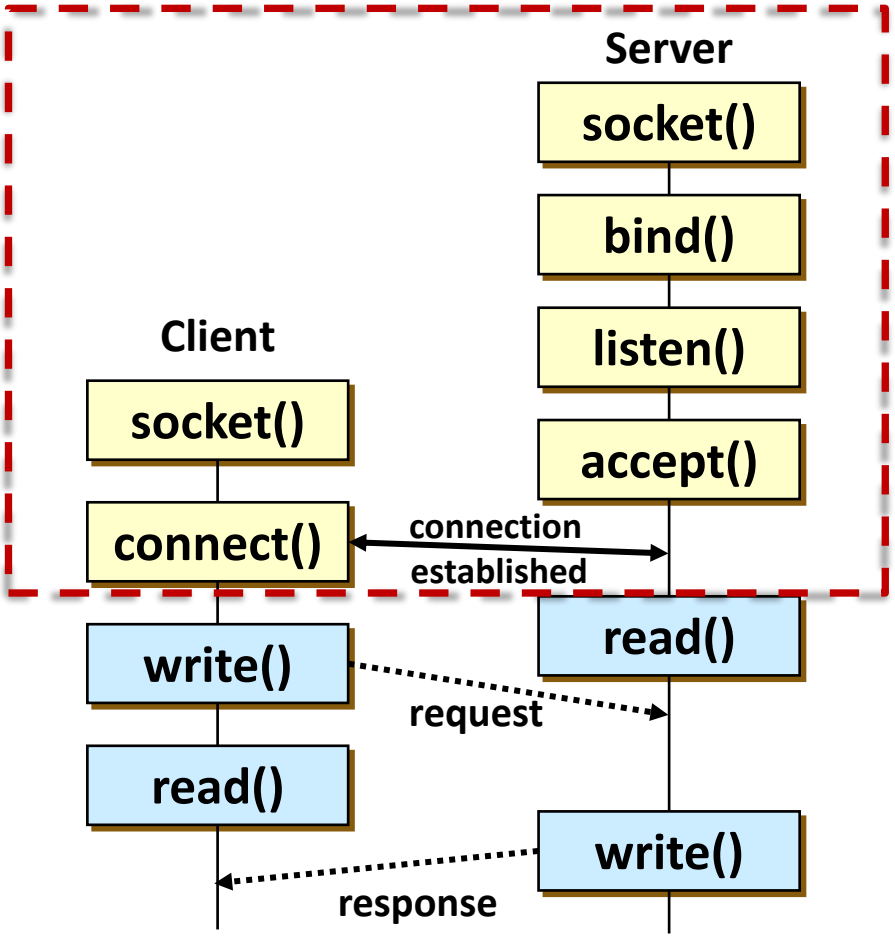
Socket (3)

- Hardware/Software organization of an Internet application

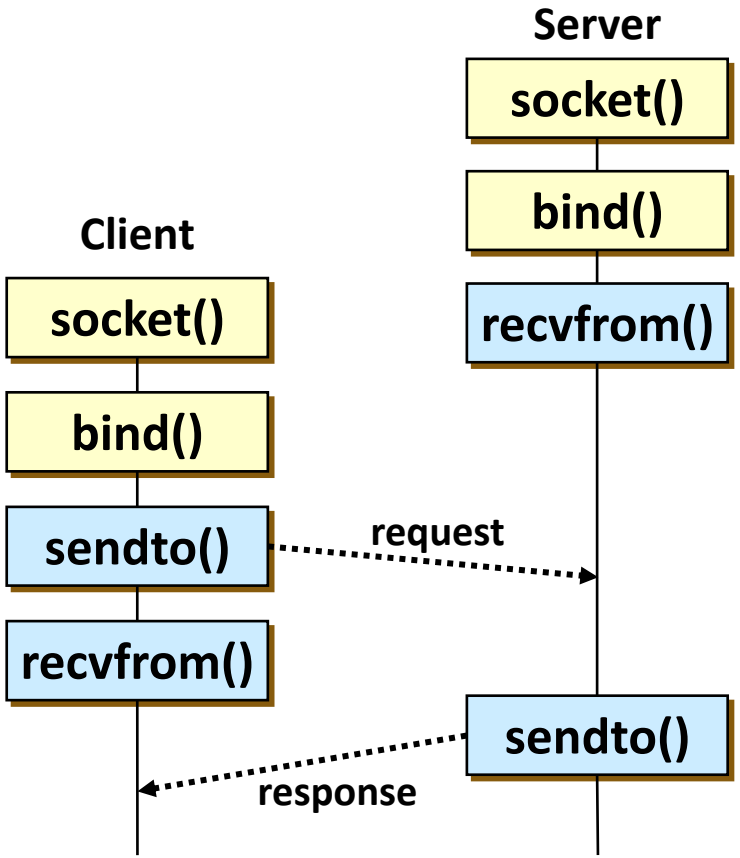


TCP vs. UDP

Connection-oriented service



Connectionless service



Socket Address Structure

- Generic socket address
  - For address arguments to **connect(), bind(),** and **accept()**

```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data    */
};
```

- Internet-specific socket address
  - Must cast **(sockaddr_in *)** to **(sockaddr *)** for **connect(), bind(),** and **accept()**

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order  */
  struct in_addr  sin_addr;    /* IP addr in network byte order   */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr)  */
};
```

socket()

- int socket (int ***domain***, int ***type***, int ***protocol***)
  - **socket()** creates a socket descriptor
  - ***domain*** specifies the communication domain
    - **AF_UNIX**: Local Unix domain protocols
    - **AF_INET**: IPv4 Internet protocols
  - ***type*** specifies the communication semantics
    - **SOCK_STREAM**: provides *sequenced, reliable, two-way, connection-based* **byte streams**
    - **SOCK_DGRAM**: supports **datagrams** (*connectionless, unreliable messages of a fixed maximum length*)
    - **SOCK_RAW**: provides raw network protocol access
  - ***protocol*** specifies a particular protocol to be used with the socket (`IPPROTO_IP (0)`, `IPPROTO_TCP`, `IPPROTO_UDP`)

connect()

- int connect (int ***sockfd***,
  const struct sockaddr ****servaddr***,
  socklen_t ***addrlen***)
  - Used by a **TCP client** to establish a connection with a TCP server
  - ***servaddr*** contains **<IP address, port number>** of the server
  - Client does not have to call **bind()** before calling **connect()**
    - Kernel will choose both ephemeral port and source IP address if necessary
  - Client process **suspends (blocks)** until the connection is created

Functions for Echo Client

```
MEMSET(3)                                Linux Programmer's Manual                                MEMSET(3)

NAME
       memset - fill memory with a constant byte

SYNOPSIS
       #include <string.h>

       void *memset(void *s, int c, size_t n);

DESCRIPTION
       The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.

RETURN VALUE
       The memset() function returns a pointer to the memory area s.
```

```
MEMCPY(3)                                Linux Programmer's Manual                                MEMCPY(3)

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memcpy()  function  copies n bytes from memory area src to memory area dest.  The memory areas must not overlap.  Use mem-
       move(3) if the memory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.
```

Echo Client (1)

```c
#include <arpa/inet.h> // htons, tcp, ipv4
#include <sys/types.h> // ssize_t
#include <sys/socket.h> // socket
#include <netdb.h> // hostent
#include <stdio.h> // printf
#include <unistd.h> // write, read
#include <stdlib.h> // atoi, exit
#include <string.h> // memset, memcpy
```

```c
// swe2024-41_23s/2023s/w10/echo_client.c

#define MAXLINE 80

int main (int argc, char *argv[]) {
   ssize_t num_bytes; // number of bytes read
   int num_bytes, socket_fd; // fd that handles socket
   struct hostent *host_entry; // contains info related to host
   struct sockaddr_in socket_address;
   char buffer[MAXLINE];

   if (argc < 3){
      printf("Received %d arguments. Please enter hostname and port
number!\n", argc - 1);
      exit(1);
   }

   char *host = argv[1]; // contains hostname
   uint32_t port = (uint32_t)strtol(argv[2], NULL, 10); // contains
port number
```

Echo Client (2)

```
   // creates a socket for TCP connections
   // AF_INET: IPv4, SOCK_STREAM: stream socket, IPPROTO_TCP: TCP
   if ((socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
      printf("socket() failed.\n");
      exit(2);
   }

   // Gets the host entry given the hostname
   if ((host_entry = gethostbyname(host)) == NULL) {
      printf("invalid hostname %s\n", host);
      exit(3);
   }

   memset((char *)&socket_address, 0, sizeof(socket_address)); //
initializes socket_address fields to zero
   socket_address.sin_family = AF_INET; // sets address family to IPv4
   memcpy((char *)&socket_address.sin_addr.s_addr, (char *)host_entry-
>h_addr, host_entry->h_length); // copies ip address
   socket_address.sin_port = htons(port); // sets port after converting
it to "network byte order"
```

Echo Client (3)

```
    // Connects the socket to the given IP address
    if (connect(socket_fd,(struct sockaddr
*)&socket_address,sizeof(socket_address)) < 0) {
        printf("connect() failed.\n");
        exit(4);
    }

    // Reads user input
    while ((num_bytes = read(STDIN_FILENO, buffer, MAXLINE)) >
0) {
        write(socket_fd, buffer, num_bytes); // Send input to
server
        num_bytes = read(socket_fd, buffer, MAXLINE); // Get
response from server
        write(STDOUT_FILENO, buffer, num_bytes); // Prints
response to server
    }
    close(socket_fd); // Destroys socket
}
```

Example Output

```
💻 nvim echo_client.c && gcc echo_client.c -Wall -Wextra -o echo_client && ./echo_client tcpbin.com 4242
Hi!
Hi!
😀
😀
안녕하세요~
안녕하세요~
```

bind()

- int bind (int ***sockfd***,
  const struct sockaddr ****addr***,
  socklen_t ***addrlen***)
  - **bind**() gives socket ***sockfd*** the local address ***addr***
  - ***addr*** is ***addrlen*** bytes long
  - Servers bind their well-known port when they start
  - If a TCP server binds a specific IP address to its socket, this **restricts** the socket to receive incoming client connections **destined only to that IP address**
  - Normally, a TCP client lets the kernel choose the ephemeral port and client IP address

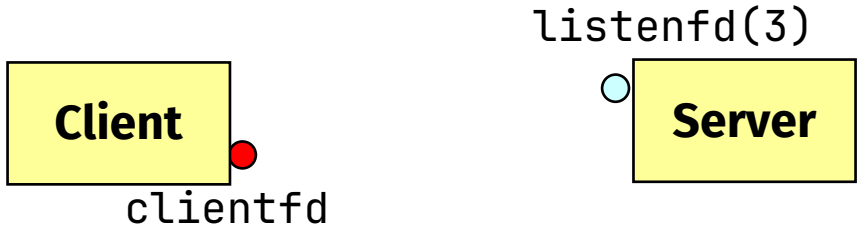
listen()

- int listen (int ***sockfd***, int ***backlog***)
  - **listen()** converts an unconnected socket into a **passive socket**, indicating that the **kernel should accept incoming connection requests**
    - When socket is created, it is assumed to be an active socket, that is, a client socket that will issue **connect()**
  - ***backlog*** specifies the **maximum number of connections** that kernel should queue for this socket
  - Historically, a backlog of 5 was used, as that was maximum value supported by 4.2BSD
    - Busy HTTP servers must specify much larger backlog, and newer kernels must support larger values

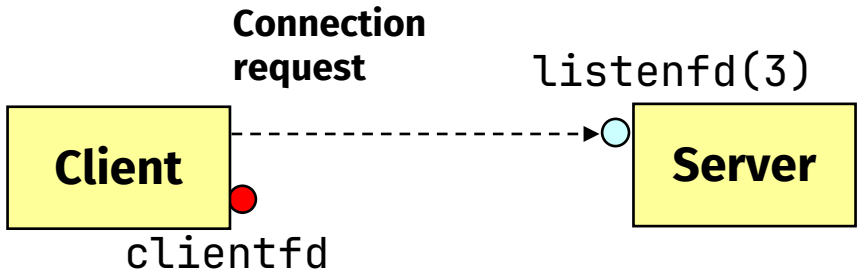
accept() (1)

- int accept (int ***sockfd***,
  struct sockaddr ****cliaddr***,
  socklen_t ****addrlen***)
  - **accept()** **blocks waiting** for a connection request
  - **accept()** returns a **connected descriptor** with the same properties as the listening descriptor
    - Kernel creates one connected socket for each client connection that is accepted
    - accept() returns when **the connection** between client and server is **created and ready for I/O transfers**
    - All I/O with the client will be done via the connected socket
  - The ***cliaddr*** and ***addrlen*** arguments are used to return the address of the **connected peer process** (the client)

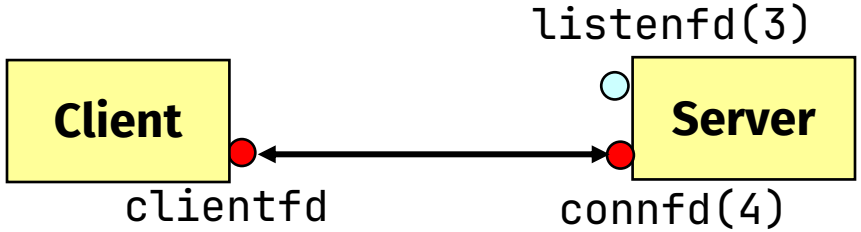
accept() (2)



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`



3. Server returns `connfd` from accept. Client returns from connect. Connection is now established between `clientfd` and `connfd`.

accept() (3)

- Listening descriptor
  - End point for client connection requests
  - Created once and **exists for lifetime of the server**
- Connected descriptor
  - End point of the **connection between client and server**
  - **A new descriptor** is created **each** time the server accepts a connection request from a **client**
  - It exist only as long as it takes to service client
- Why the distinction?
  - Allow for **concurrent servers** that can communicate over many client connections simultaneously

Echo Server (1)

```c
#include <arpa/inet.h> // htons/l,tcp,ipv4
#include <sys/types.h> // ssize_t
#include <sys/socket.h> // socket
#include <stdio.h> // printf
#include <unistd.h> // write, read
#include <stdlib.h> // atoi, exit
#include <string.h> // memset, memcpy
```

```c
// swe2024-41_23s/2023s/w10/echo_server.c

#define MAXLINE 80

int main (int argc, char *argv[]) {
   ssize_t num_bytes; // number of bytes read
   size_t connected_address_length;
   int listening_fd, connected_fd;
   struct sockaddr_in socket_address, connected_address;
   char buffer[MAXLINE];

   if (argc < 2){
      printf("Received %d arguments. Please enter port number!\n", argc - 1);
      exit(1);
   }

   uint32_t port = (uint32_t)strtol(argv[1], NULL, 10); // contains port
number

   // creates a socket for TCP connections
   // AF_INET: IPv4, SOCK_STREAM: stream socket, IPPROTO_TCP: TCP
   if ((listening_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
      printf("socket() failed.\n");
      exit(2);
   }
```

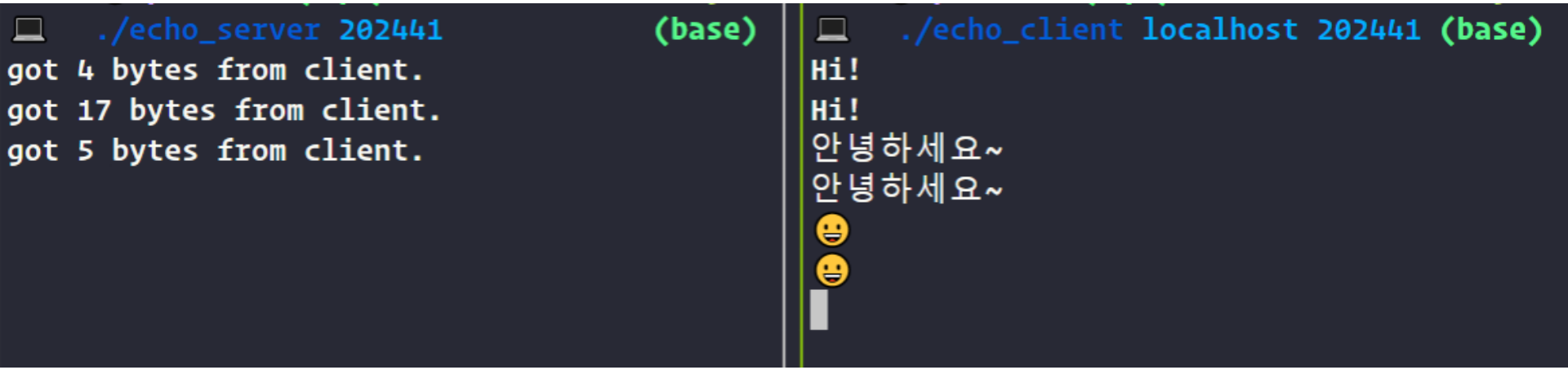
Echo Server (2)

```
   memset((char *)&socket_address, 0, sizeof(socket_address));  // initializes socket_address fields to zero
   socket_address.sin_family = AF_INET;  // sets address family to IPv4
   socket_address.sin_addr.s_addr = htonl(INADDR_ANY);  // copies ip address
   socket_address.sin_port = htons(port); // sets port after converting it to "network byte order"

   // Assigns socket address to the listening fd
   if (bind(listening_fd, (struct sockaddr *)&socket_address, sizeof(socket_address)) < 0) {
      printf("bind() failed.\n");
      exit(3);
   }

   // Listens to 5 connections at most
   if (listen(listening_fd, 5) < 0) {
      printf("listen() failed.\n");
      exit(4);
   }
```

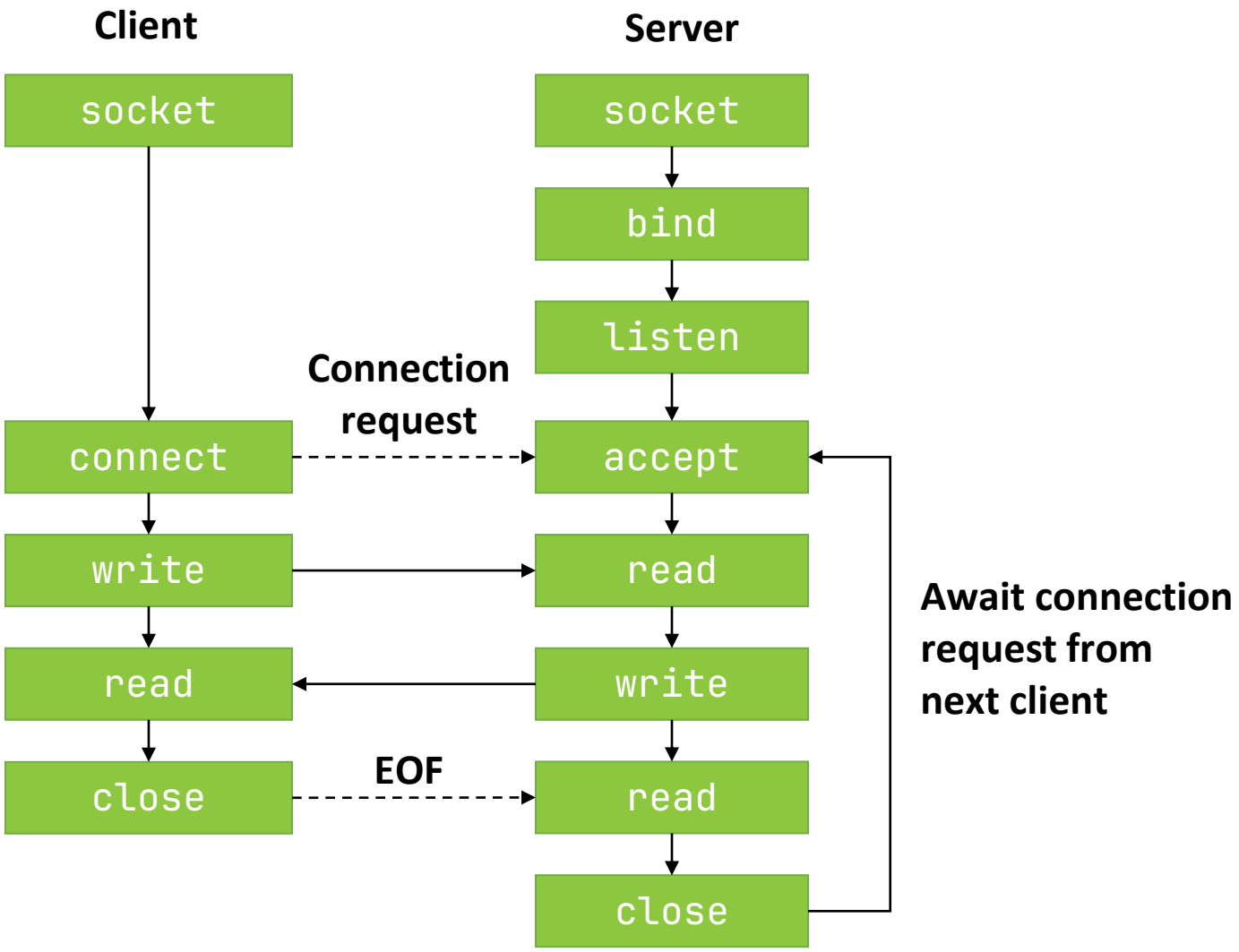
Echo Server (3)

```
    while (1) {
        connected_address_length = sizeof(connected_address);

        // Accepts connection
        if ((connected_fd = accept(
                listening_fd,
                (struct sockaddr *)&connected_address,
                (socklen_t *)&connected_address_length
            )) < 0) {
            printf ("accept() failed.\n");
            continue;
        }

        // Receives data from client
        while ((num_bytes = read(connected_fd, buffer, MAXLINE)) > 0) {
            printf ("got %ld bytes from client.\n", num_bytes);
            write(connected_fd, buffer, num_bytes);
        }

        // Terminates connection
        printf("connection terminated.\n");
        close(connected_fd);
    }
}
```

Example Output



Echo Server (4)



Lab Exercise

- Make *file transfer*
  - Server runs in background by attaching & when running
  - Client runs in foreground and is shown on the screen
  - Client should be under an infinite loop with conditional exit (“quit”)
  - If server receives a ‘quit’ string, the server shuts down
  - When name of the file is entered, client reads file and sends it to server
  - Server saves file name received from client by appending *_copy*
  - File is text file and does not have extension
    - e.g., a.txt(X), b.exe(X), c.c(X), Aladdin(O), a(O), b(O)
  - Size of file is smaller than 200,000 bytes
  - Length of file name is shorter than 50 bytes
  - When testing, use last 5 digits of your student ID for the port

Lab Exercise

- Skeleton code

```
cp ~swe2024-41_23s/2023s/p10  ./ -r
```

- When testing on the cluster, use last 5 digits of your student ID for the port (i.e., do not use 10080!)

- Example)

```
root@ubuntu:/home/seungwoo/swe2033/ex10# ls
a  b  client  server
root@ubuntu:/home/seungwoo/swe2033/ex10# ./server 10080 &
[1] 54822
root@ubuntu:/home/seungwoo/swe2033/ex10# ./client localhost 10080
a
got 31 bytes from client.
Send 31 bytes to server.
b
got 17 bytes from client.
Send 17 bytes to server.
quit
[1]+  완료                  ./server 10080
root@ubuntu:/home/seungwoo/swe2033/ex10# ls
a  a_copy  b  b_copy  client  server
```

Exercise Submission

- Submit your exercise code and Makefile
  - InUiYeJi cluster
  - Submit the folder into p10
  - ~swe2024-41_23s/bin/submit p10 p10
  - Due date: Friday, 05 May 2023, 23:59
  - We will compile by using command *make*
    - If compilation fails, your points for this exercise will be **zero**

```
./p10

   server.c
   client.c
   Makefile
```

Summary Report

- Summary report about man command result of
  - `socket()`
  - `bind()`
  - `listen()`
  - `accept()`
  - `connect()`

- Submission form
  - A4 size PDF format (No page limitation)
  - [SWE2024 Report-10] studentID_name
  - Ex) [SWE2024 Report-10] 2022XXXXXX_홍길동
  - Submit to iCampus
  - Due by Friday, 5 May 2023, 23:59