國立臺灣大學

Assignment 2

TCP Socket Programming

Prof. Ai-Chun Pang
TA / Yu-Yu Chen, Shao-Cheng Fan, Kuang-Hui Huang

Assignment 2 Announcement

Specification (1/22)

- In this assignment, you need to implement a simple Network Storage System with the following functions:
  - client can **list all files in the user folder of the server**.
  - client can **upload files** to the server.
  - client can **download files** from the server.
  - client can **watch a ".mpg" video (streaming)** on the server.
  - User "admin" can **ban users**.
  - User "admin" can **unban users**.
  - User "admin" can **list all banned users**.

Specification (2/22)

- For video streaming, you **don’t need to send audio**. You can just send frames in **RAW format**.
- After upload and download, you must **ensure the files are identical between source and destination**.
- In this assignment, all the transmissions should be implemented in **C/C++** and by **TCP socket**.

Specification – Commands (3/22)

- The server is required to support multiple connections. That is, **more than one client** can connect to the server **simultaneously**.
- After building up a connection, a **client** can use the commands below:

```
$ ls
$ put <filename>
$ get <filename>
$ play <videofile.mpg>
```

Specification – Admin Commands (4/22)

- After building up a connection, user “admin” can use **the basic commands and the commands below**:

```
$ ban <username1> <username2> … <usernameN>
$ unban <username1> <username2> … <usernameN>
$ blocklist
```

- Don’t assume the lengths of the commands are lower than a specific size in advance.

Specification – ls (5/22)

- For the command **ls**, a **client** should list all the files in the user folder of the server with the filename separated by a **newline** character.

```
$ ls
file1
file2
file3
…
video.mpg
```

Specification – put (6/22)

- For the command **put**, a **client** should upload the file specified in the arguments to the server.

```
$ put file1
putting file1…
$ put file99
file99 doesn’t exist.
```

- There won’t be multiple clients putting the same file at the same time.

Specification – get (7/22)

- For the command **get**, a **client** should download the file specified in the arguments from the server.

```
$ get file1
getting file1…
$ get file99
file99 doesn’t exist.
```

- There might be multiple clients getting the same file at the same time.
- A client will not get the file that is being put by another client.

Specification – play (8/22)

- For the command **play**, a **client** should play the **.mpg** video file specified in the arguments.

```
$ play video1.mpg
playing the video…
$ play video99.mpg
video99.mpg doesn’t exist.
```

- There might be multiple clients playing the same video at the same time.
- A client will not play the video that is being put.

Specification – play (9/22)

- After pressing **ESC**, a **client** should terminate the video and be able to keep sending another command.
- The video window should be closed automatically at the end of the video and after pressing ESC.

Specification – ban (10/22)

- User “admin” can add other users to the **blocklist** by the **ban** command.

```
$ ban Alice Bob Cat admin Bob
Ban Alice successfully!
Ban Bob successfully!
Ban Cat successfully!
You cannot ban yourself!
User Bob is already on the blocklist!
$ ban Cat
User Cat is already on the blocklist!
```

Specification – ban (cont.) (11/22)

- User “admin” can ban many users in one command.
- There may be no connection between the banned users and the server. That is, you don't need to check whether those banned users connect to the server.

Specification – unban (12/22)

- User “admin” can remove other users from the **blocklist** by the **unban** command.

```
$ unban Alice Peter admin Alice
Successfully removed Alice from the blocklist!
User Peter is not on the blocklist!
User admin is not on the blocklist!
User Alice is not on the blocklist!
```

- User “admin” can unban many users in one command.

Specification – blocklist (13/22)

- For the command **blocklist**, user “admin should list every banned user with the username separated by a **newline** character.

```
$ blocklist
Bob
Cat
```

- If the blocklist is empty, you don’t need to print anything.

Specification – Permission (14/22)

- Only user “admin” can run **ban**, **unban**, and **blocklist**. If other users run the commands, print out “Permission denied.” on the client side.
- The users on the blocklist can't run any commands, and they should print out “Permission denied.”
- If users are banned while running a command, they can keep running until the end of the command.

Specification – Permission (cont.) (15/22)

- User “Bob” was added to the blocklist when playing the video.
- The sample output of user “Bob”:

```
$ blocklist
Permission denied.
$ play video1.mpg
playing the video…                    // admin added user Bob to the blocklist.
$ play video1.mpg
Permission denied.
```

Specification – Compilation (16/22)

- You are required to write a Makefile for compilation.

```
$ make client          // To compile client code
$ make server          // To compile server code
```

- After compilation, there will be 2 binary files named “**client**” and “**server.**”

Specification – Usage (17/22)

- When we launch the server, we will use

```
$ ./server <port>                    // <port> will be determined
```

- After launching, the server should create a folder named **server_dir**.
- No matter what happens, **server** should NOT be terminated.

Specification – Usage (18/22)

- When we launch the client, we will use

```
$ ./client <username> <ip>:<port>
// <username>  consists of no more than 10 English letters and digits
// <ip> is the ip address of the server
// <port> is determined by the command above
```

- After launching, the client should create a folder named **client_dir**, and the server should create a folder named **<username>** under folder **server_dir**.
- A **client** might be terminated at any time.

Specification – Usage (19/22)

- If we launch the server and 2 clients with username **Alice** and **Bob**:

```
|-- server_dir
|    |-- Alice
|    |    |-- file00
|    |    `-- file01
|    `-- Bob
|         |-- file10
|         `-- file11
`-- client_dir
```

Specification – Error Handling (20/22)

- All of your outputs should be printed to **standard output (stdout)** and end with a **newline**.
- If the command doesn't exist or the command format is wrong, print out "Command not found." on the client side.
- If the file doesn't exist while putting, getting, or playing a file, print out "<filename> doesn't exist." on the client side.
- If the video file is not a ".mpg" file while playing a video file, print out "<videofile> is not an mpg file." on the client side.

Specification (21/22)

- The server should **output the file descriptor of the new connection** and **the username** when a client connects to the server.

```
Accept a new connection on socket [<file descriptor>]. Login as <username>.
```

- A **client** should be able to send another command after a command is finished.
- The multiple connections should be implemented with `pthread.h` or `select()`, while the video player should be implemented with **OpenCV**.
- The implementation must be in **C or C++**.
- The file size will be no more than 2GB.
- Every command should be done in 5 minutes.

Specification – Multiple Connections (22/22)

- There **can be more than one client** connecting to the server and sending commands to the server **simultaneously**.
- A single user can **log in using multiple clients simultaneously**.
- Commands from different **clients** should be executed **concurrently**. That is, a command from one client **cannot** be blocked by other commands from other clients.

Grading Policy (1/4)

- This assignment accounts for 15% of the total score.
- **Command Sending (10%)**
  - The client and the server handle commands correctly. (5%)
  - The client prints out responses correctly. (5%)
- **Basic File Transferring (20%)**
  - There would be 5 test cases. (4% * 5)
  - You will get **0 points** in a test case if the transfer of a test case **is terminated** or **halts** before finished, or the files are **not identical between source and destination** after the transfer.

Grading Policy (2/4)

- **User Management (12%)**
  - There will be 4 test cases. (3% * 4)
- **Video Streaming (18%)**
  - Correctly playing a **resolution-fixed** video. (960*540) (6%)
  - Correctly playing a **resolution-unknown** video. (6%)
    (The client has no idea about the resolution of the video.)
  - Playing video while others are transmitting files. (6%)

Grading Policy (3/4)

- **Multiple Connections (30%)**
  - There will be 5 test cases.
  - You need to implement one of the methods below and get the corresponding score for this part.
    - **Method 1**: Use **<pthread.h>** to achieve this function (basic) (20%)
    - **Method 2**: Use **select()** to achieve this function (advance) (30%)

Grading Policy (4/4)

- **Report (5% * 4)**
  - Draw a flowchart of the file transferring and explain how it works in detail.
  - Draw a flowchart of the video streaming and explain how it works in detail.
  - What is SIGPIPE? Is it possible that SIGPIPE is sent to your process? If so, how do you handle it?
  - Is blocking I/O equal to synchronized I/O? Please give some examples to explain it.

Submission

- Requirements
  - Your report can be a **pdf** or **clear image** file. Submit it to **Gradescope**.
  - Please put all the source code (i.e., without your report, the video file, and the execution file) into a folder named **<studentID>_hw2** and compress the folder as a **.zip** file. Submit your **.zip** file to **NTU COOL**.

```
B09902999_hw2.zip
 `-- B09902999_hw2              (<== folder)
     `-- your source code and Makefile
```

  - The penalty for the wrong format is **10 points**.
  - **No plagiarism is allowed. A plagiarist will be graded zero.**

Submission

- Deadline
  - Due Date : 23:59:00, November 8th, 2022
  - The penalty for late submission is **20 points per day**.

If You have any Problems…

- You can
  - Ask questions on NTU COOL Discussion Forum, or
  - Send a mail to TA with the tag **[HW2]** in the title
  - Ask questions in TA hours in R438 by appointment. Google Sheet Link.
- TA Email: ntu.cnta@gmail.com

Sample Codes

- We will provide sample codes for your reference.
  - *server.c* - Default port number is 8787
  - *client.c* - Default IP address is 127.0.0.1, port is 8787
  - *pthread.c*
  - *openCV.cpp* - It will play video.mpg
  - *video.mpg*
  - *Makefile*

Environment Setup

Environment

- We provide a VirtualBox VM for you to run our example code. If you use computers with ARM-based processors (e.g., M1 Macbook), you can use NTU CSIE workstations alternatively.
- **Please make sure you can compile and run your code well on either our VM or NTU CSIE workstation.**
- If you use NTU CSIE workstations, please enable X11 forwarding in order to play videos.
    - For mac users, you can use XQuartz.
    - For Windows users, you can use MobaXterm.
    - Linux supports this feature natively.

Environment(cont.)

- If you choose to set up the environment on your OS rather than using our VM, here is information about our environment
  - Ubuntu 20.04 x64
  - OpenCV 4.2.0
- You can install OpenCV 4.2.0 by the command below or follow the instruction here:

```
$ sudo apt install libopencv-dev=4.2.0+dfsg-5
```

VirtualBox Setup

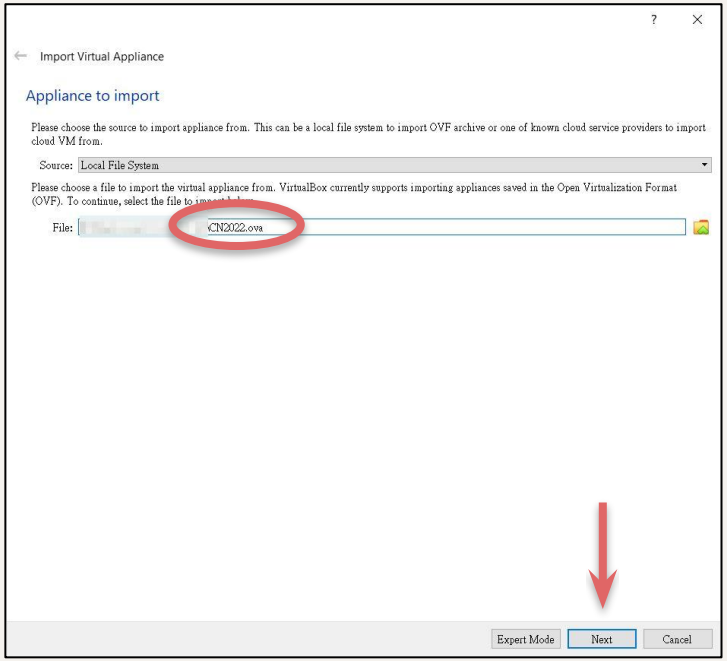
- Download the VM from
  - Our Google Drive
  - The password of our VM is **cn2022**.
- Install Virtualbox (natively installed on the computers of Lab R204).

VirtualBox Setup

- Click “**Import**” to import the “**CN2022.ova**”

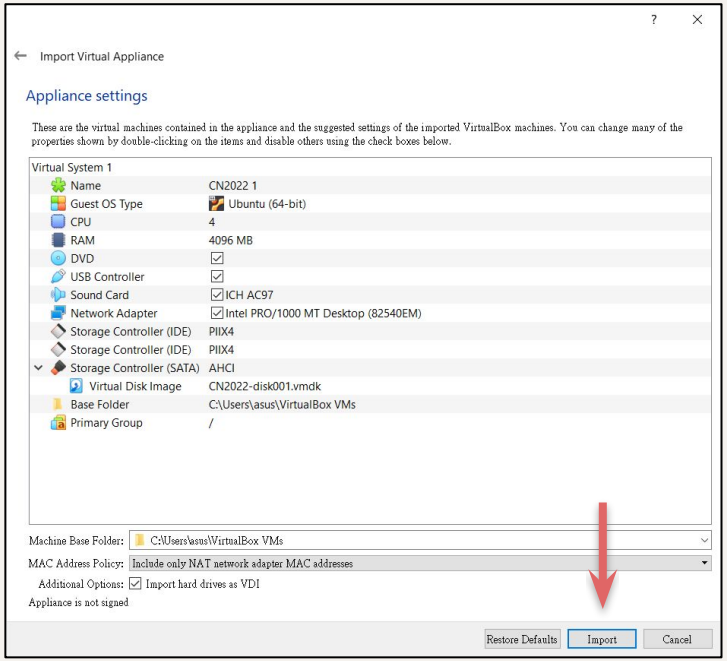
VirtualBox Setup

- Choose “**CN2022.ova**” file and click “**Next**”



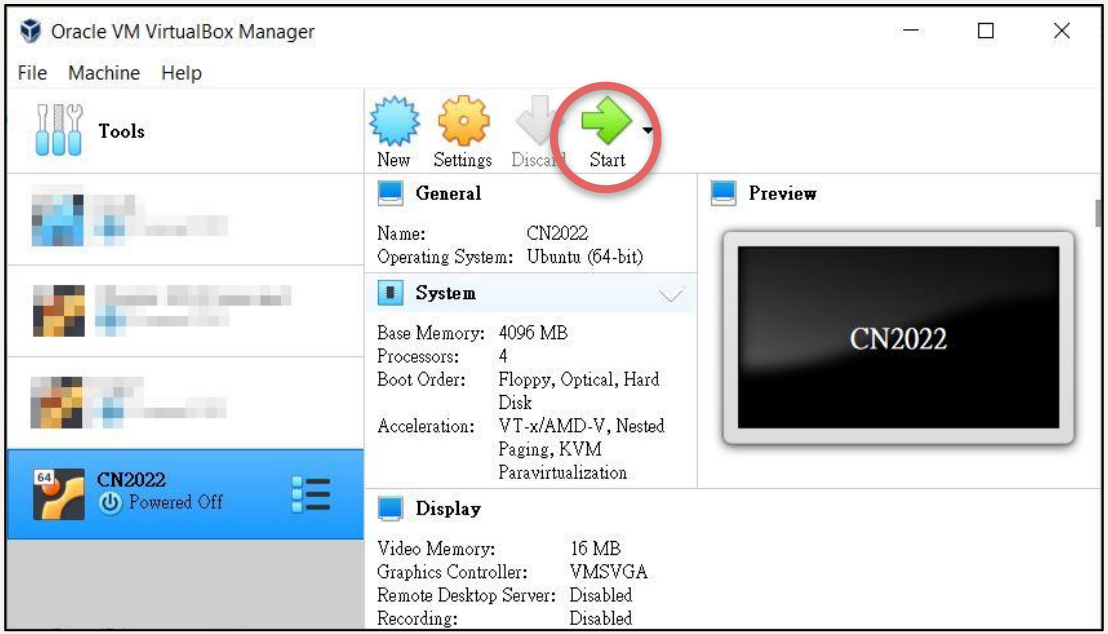
VirtualBox Setup

- Click “**Import**” to import the “**CN2022.ova**”



VirtualBox Setup

- Choose “**CN2022**” and then start the machine.



Auxiliary Libraries

OpenCV

- An open source library for computer vision.
- ***Mat*** is an image container to load an image so that you can easy to do image processing, recognition, etc.
- In this assignment, we use this library to get frames from videos on the server, and show frames on the client.
- We will provide a sample code and a .mpg file for you.
- To compile code with OpenCV,

```
$ g++ <file name> -o <output name> `pkg-config --cflags --libs opencv4`
```

Pthread

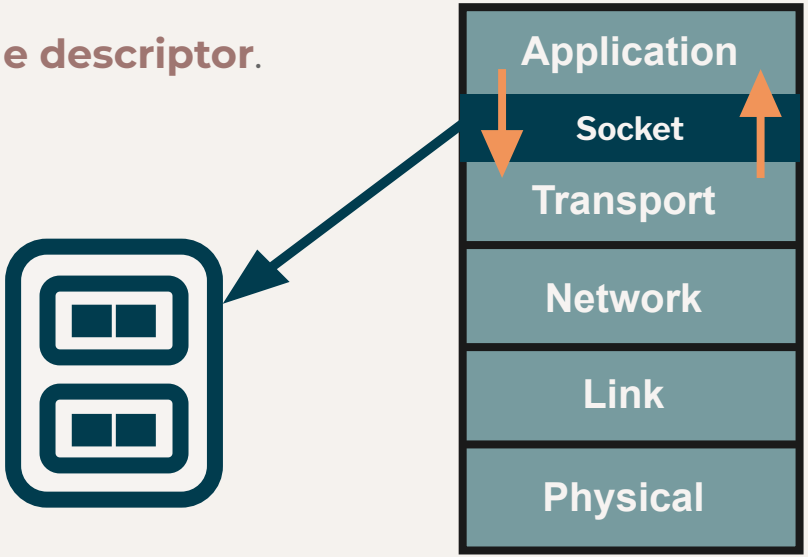
- Pthread, i.e., POSIX Thread, is used to implement multi-thread parallelization in a POSIX environment.
- You can use pthread to achieve multiple connections.
- You don’t need to deal with synchronization issues, i.e., in our test cases, it won’t put a file with the same file name.
- We will provide a sample code for you.
- To compile with Pthread,

```
$ g++ <file name> -o <output name> -pthread
```

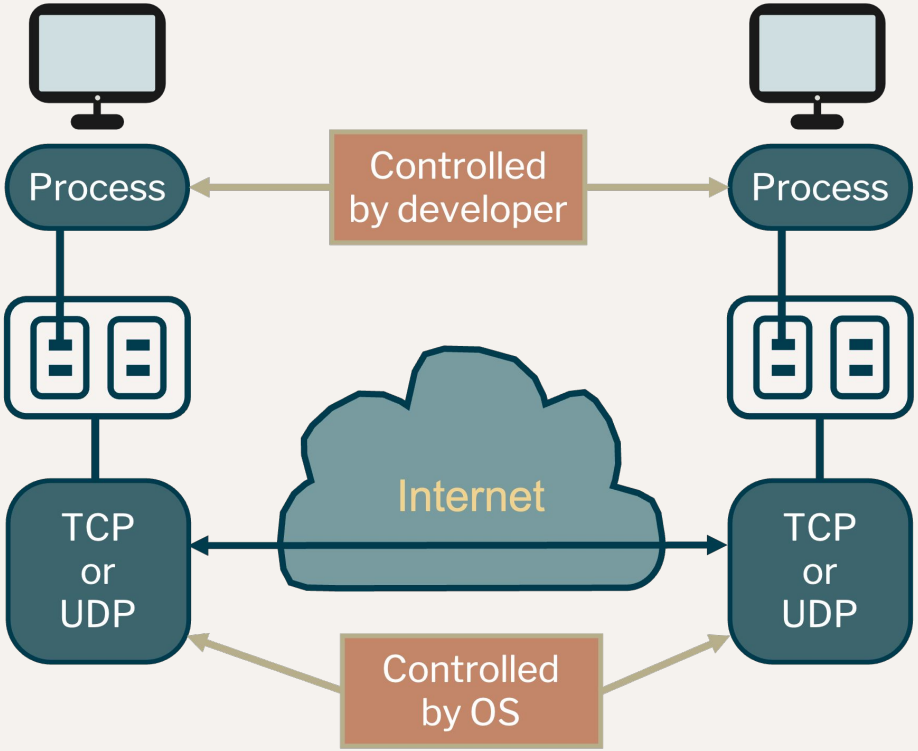
Socket Programming Tutorial

What is Socket?

- **Socket** is the **API for the TCP/IP protocol** stack.
- Provides communication **between the Application layer and Transport layer**.
- Make internet communication **like a file descriptor**.
  - read() and write()
- We will provide a sample code for you.



What is Socket?



File Descriptors

- When we open an existing file or create a new file, the kernel returns a file descriptor to the process.
- If we want to read or write a file, we identify the file with the file descriptor.

Interger vaule	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

```
FILE *fp = fopen("this.txt","w");
fprintf(fp, "Happy Coding.");
fclose(fp);
```

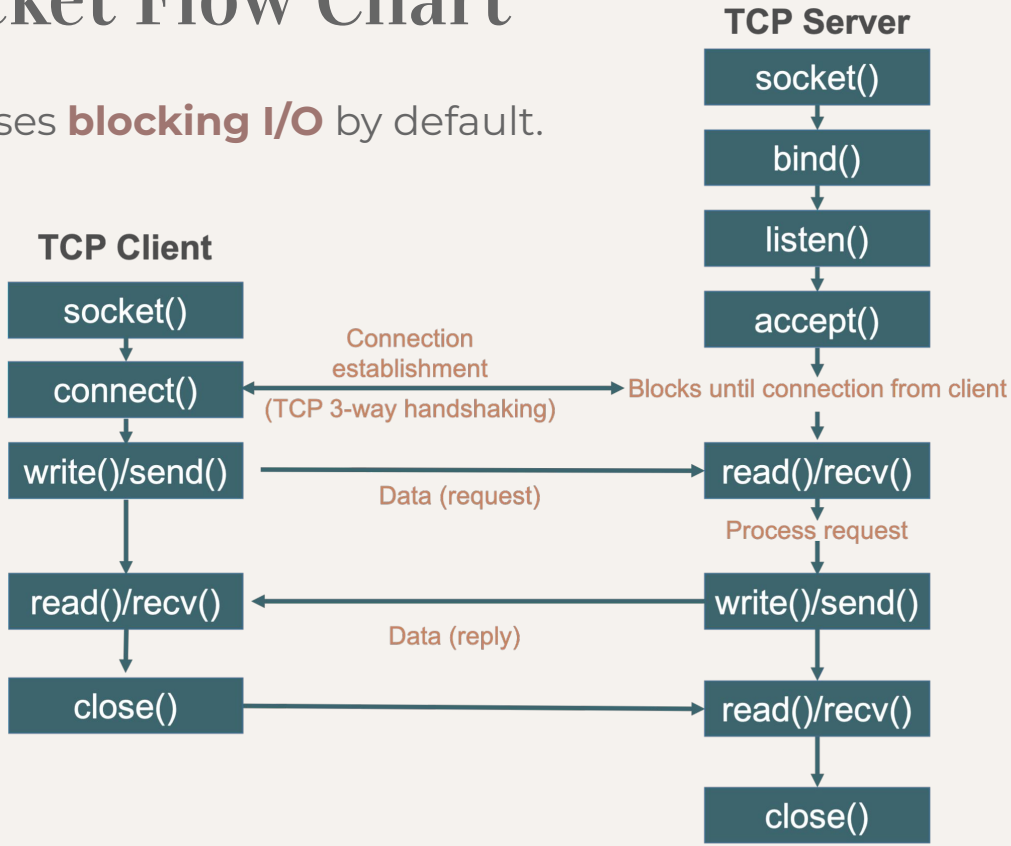
```
printf("This is Computer Networking.\n");
fprintf(stdout," This is Computer Networking.\n");
```

TCP Service

- **TCP (Transmission Control Protocol)**
  - **Connection-oriented**
  - **Reliable transport**
  - Flow control
  - Congestion control
- What is Socket-Address?
  - **IP address + Port number**
  - IP address: To find out the machine (Network Layer)
  - Port number: To find out the process (Transport Layer)

TCP Socket Flow Chart

- Socket uses **blocking I/O** by default.



socket()

- **Create** the endpoint for connection.

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

- domain
  - AF_UNIX/AF_LOCAL: communication between 2 processes on a host. So they can share a file system.
  - AF_INET, AF_INET6: communication between processes on different hosts through the Internet. AF_INET is for **IPv4**, whereas AF_INET6 is for **IPv6**.

socket()

- **Create** the endpoint for connection.

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

- type
  - SOCK_STREAM: sequential and connection-oriented (TCP)
  - SOCK_DGRAM: datagram (UDP)
- protocol: defined in /etc/protocols, usually set to 0
- return: socket file descriptor (an integer)

bind()

- **Bind** the address to the socket.

```
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr *addr, socklen_t len);
```

- sockfd: specifies the socket file descriptor to bind.
- addr
  - specifies the socket address to be associated with the sockfd
  - You can use “struct sockaddr_in*” defined in <netinet/in.h>, and then cast it into “struct sockaddr*”
- len: specifies the size of sockaddr (= **sizeof(struct sockaddr)**)

bind()

```
struct sockaddr_in {
        short sin_family;               // address family. EX:AF_INET
        unsigned short sin_port;        // port number for network
        struct in_addr sin_addr;        // IP address for network
        unsigned char sin_zero[8];      // pad to sizeof(struct sockaddr)
}
```

listen()

- Specify a socket to **listen** for connections.

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);      // returns 0 if it's success; -1 otherwise
```

- sockfd: specifies the socket file descriptor to listen.
- backlog: specifies the number of users allowed in queue.

accept()

- **Accept** the connection on a socket.

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

- sockfd: specifies the socket being listened to.
- addr: pointer to the sockaddr. It will be filled in with **the address of the peer socket**.
- **Blocking** until a user **connect()** call is received.
- After accepting the connection, it **creates a new file descriptor** for the client. The original socket is not affected.

connect()

- **Connect** to the socket from client to server.

```
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *addr, socklen_t len);
```

- The format is the same as **bind()**.

close()

- **Close the file descriptor**.

```
#include <unistd.h>
int close (int sockfd);        // returns 0 if it's success; -1 otherwise
```

read()/recv()

- **Read** data from the socket file descriptor.

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t count);
ssize_t recv (int fd, void *buf, size_t len, int flag);
```

- fd: specifies the socket file descriptor to read data from.
- buf: specifies the buffer to contain the received data.
- count/len: specifies the size to receive.
- flag: (**read()** has no this parameter.) It's about some details like blocking/nonblocking.

read()/recv()

- **Read** data from the socket file descriptor.

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t count);
ssize_t recv (int fd, void *buf, size_t len, int flag);
```

- Reading data from a file may be
  - Successful, return the number of bytes received
  - EOF. (end of file) (i.e., return = 0)
  - Failed, **errno** is set to indicate the error.
- It may be blocked. (**block I/O**).

write()/send()

- **Write** data to socket file descriptor.

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
ssize_t send (int fd, const void *buf, size_t len, int flags);
```

- fd: specifies the socket file descriptor to send data to.
- buf: specifies the buffer to contain the data to be transmitted.
- count/len: specifies the size to send.
- flag: (write() has no this parameter.) It's about some details.

write()/send()

- **Write** data to socket file descriptor.

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
ssize_t send (int fd, const void *buf, size_t len, int flags);
```

- Writing data to file may be
  - Successful, return the number of bytes written.
  - Failed, **errno** is set to indicate the error.
- It may be blocked. (**block I/O**)

Useful Functions

- Address and port numbers are stored as integers.
  - Different machines implement **different endian**.
  - They may communicate with each other on the network.
- Converting IP address and port number.
  - `htonl()`: for IP address (host -> network)
  - `ntohl()`: for IP address (network -> host)
  - `htons()`: for port number (host -> network)
  - `ntohs()`: for port number (network -> host)

Useful Functions

- An IP address is usually hard to remember.
  - We need to **translate the hostname to IP address**.
- Translate a hostname to IP address.

```
#include <netdb.h>
struct hostent *gethostbyname (const char *name);
```

Supplementary Materials

How to Emulate a Bad Network

- Some bugs may occur when the network is not good.
- We can emulate a bad network to test our program in advance.
- Linux
    - Get the network interfaces list in your machine

```
$ ifconfig
```

    - Then, to make your personal internet slow, you can enter

```
$ sudo tc qdisc add dev <interface> root netem delay 500ms
```

      In this way, the delay will be 500ms.

```
$ sudo tc qdisc del dev <interface> root netem
```

How to Trace Kernel

- Sometimes, the service may terminate without any error message.
- It happens usually because of some kernel issues.
- To trace the interactions between your code and kernel, we can use strace.
- To run your program with strace, you can enter

```
$ strace ./<name>
```

Behavior of send() and recv()

- In fact, a **send()** doesn't imply all the data in the buffer are sent **once**.
- In addition, a **send()** doesn't imply all the data in the buffer are sent in a packet, and even 2 **send()** don't imply they are in different packets.
- You are required to design a protocol so that each receive has the same size as the **send** in respect to it.

select()

- **select()** provides you to supervise multiple sockets, telling you which is able to read or write, etc.
- With **select()**, it is possible to achieve Asynchronous Blocking I/O.
- If you want to implement this assignment with **select()**, please refer to this website.

select()

- **Monitor** whether there is at least one fd available.

```
#include <unistd.h>
int select (int nfds, fd_set*, readfds, fd_set* writefds, fd_set* exceptfds, struct timeval* timeout);
```

- nfds: specifies the number of file descriptors to monitor.
- readfds: specifies the pointer to read file descriptor list.
- writefds: specifies the pointer to write file descriptor list.
- exceptfds: specifies the pointer to the error file descriptor list.
- timeout: deadline for **select()**.

select()

```
void FD_SET (int fd, fd_set *set);
void FD_CLR (int fd, fd_set *set);
int FD_ISSET (int fd, fd_set *set);  // return: 1 if it's available, else: 0
void FD_ZERO (fd_set *set);
```

- FD_SET(): Add the file descriptor into the set.
- FD_CLR(): Remove the file descriptor from the set.
- FD_ISSET(): Check if the file descriptor is available.
- FD_ZERO(): Clear the set.

Reference

- Beej's Guide to Network Programming (中文)
- Beej's Guide to Network Programming (English)
- Linux manual page

Happy coding! •ω•)ค

TA Email: ntu.cnta@gmail.com