

The background of the entire image is a solid red color. Overlaid on this background is a large, stylized arch made of numerous small, light-red dots. The arch is centered horizontally and spans most of the width of the image. In the center of this arch, the word "HUST" is written in a large, bold, white, sans-serif font.

# HUST

**ĐẠI HỌC BÁCH KHOA HÀ NỘI**  
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

# LẬP TRÌNH MẠNG (IT4060)

Giảng viên: Lê Bá Vui

Email: [vuilb@soict.hust.edu.vn](mailto:vuilb@soict.hust.edu.vn);  
[vui.leba@hust.edu.vn](mailto:vui.leba@hust.edu.vn)

Khoa KTMT – Trường CNTT & TT

- Cung cấp kiến thức cơ bản về lập trình ứng dụng trên mạng:
  - Xây dựng ứng dụng phía server.
  - Xây dựng ứng dụng phía client.
  - Tìm hiểu các kiến trúc client-server.
  - Tìm hiểu và thực hiện một số giao thức chuẩn.
- Cung cấp các kỹ năng cần thiết để thiết kế và xây dựng ứng dụng mạng:
  - Sử dụng thư viện, môi trường, tài liệu.
  - Thiết kế, xây dựng chương trình.

# Yêu cầu đối với sinh viên

- Yêu cầu về kiến thức nền tảng:
  - Mạng máy tính: địa chỉ IP, tên miền, giao thức, ...
  - Ngôn ngữ lập trình: C/C++
  - Các kỹ thuật lập trình: mảng, chuỗi ký tự, con trỏ, cấp phát bộ nhớ động, ...
  - Các kỹ năng lập trình, gỡ lỗi
- Yêu cầu khác:
  - Lên lớp đầy đủ
  - Hoàn thành bài tập về nhà
- **Điểm quá trình (50%) = Điểm thi giữa kỳ + Điểm danh + Bài tập trên lớp + Bài tập về nhà**
- **Điểm cuối kỳ (50%) = Điểm thi cuối kỳ**



- Slide bài giảng, code mẫu, tài liệu tham khảo ([Github](#))
- Unix Network Programming Volume 1, Third Edition: The Sockets Networking API. By: W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff.
- The Definitive Guide to Linux Network Programming. By: Keir Davis, John W. Tunner, and Nathan Yocom.
- The Linux Programming Interface. By: Michael Kerrisk.

Chương 1. Giới thiệu về Lập trình mạng

Chương 2. Lập trình socket cơ bản

Chương 3. Các kiến trúc client-server

Chương 4. Thiết kế giao thức mạng

Chương 5. Lập trình socket nâng cao

# Chương 1. Giới thiệu về Lập trình mạng

# Chương 1. Giới thiệu về Lập trình mạng

1.1. Khái niệm Lập trình mạng

1.2. Nhắc lại một số kiến thức Mạng máy tính

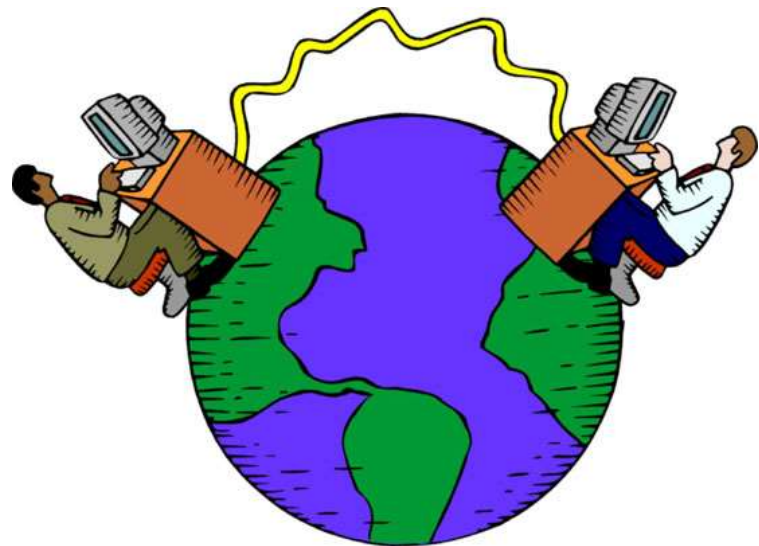
1.3. Lập trình trong môi trường Ubuntu/Linux





# 1.1. Khái niệm

Lập trình mạng bao gồm các kỹ thuật lập trình nhằm xây dựng ứng dụng, phần mềm với mục đích khai thác hiệu quả tài nguyên mạng máy tính.



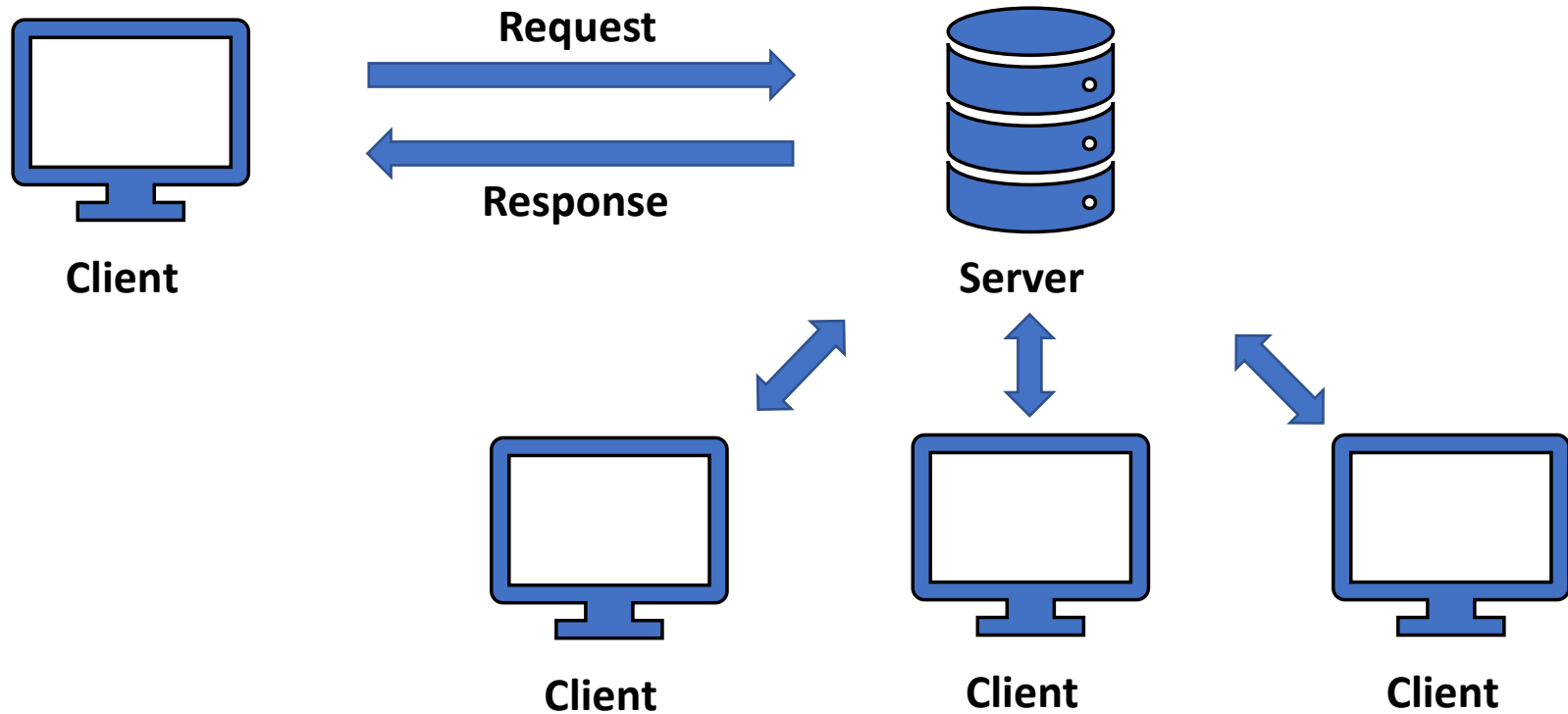
# Các vấn đề cần quan tâm

- Thông tin truyền nhận trên mạng
- Các giao thức truyền thông (Protocols)
  - Giao thức chuẩn (HTTP, FTP, SMTP, POP3, ...)
  - Giao thức tự định nghĩa
- Các kỹ thuật truyền nhận dữ liệu
- Các kỹ thuật nâng cao:
  - Nén dữ liệu
  - Mã hóa dữ liệu
  - Truyền nhận dữ liệu song song

# Các ngôn ngữ được sử dụng

- C/C++: Mạnh và phổ biến, được hầu hết các lập trình viên sử dụng để viết các ứng dụng mạng hiệu năng cao.
  - Java: Khá thông dụng, sử dụng nhiều trong các nền tảng di động (J2ME, Android).
  - C#: Mạnh và dễ sử dụng, tuy nhiên chạy trên nền .Net Framework và chỉ hỗ trợ họ hệ điều hành Windows.
  - Python, Perl, PHP... Ngôn ngữ thông dịch, sử dụng để viết các tiện ích nhỏ, nhanh chóng.
- ⇒ Giáo trình này sẽ chỉ đề cập đến hai ngôn ngữ C/C++

# Mô hình client – server



# Các kiểu ứng dụng hoạt động trên mạng

- Các ứng dụng máy chủ (servers)
  - HTTP, FTP, Mail server
  - Game server
  - Media server (DLNA), Streaming server (video, audio)
  - Proxy server
- Các ứng dụng máy khách (clients)
  - Game client
  - Mail client, FTP client, Web client (Browsers)
- Các ứng dụng mạng ngang hàng
  - uTorrent
- Các ứng dụng khác: Internet Download Manager, WireShark, Firewall, ...

# Ví dụ về các ứng dụng trên mạng

- Phần mềm web
  - Client (trình duyệt) gửi các yêu cầu đến web server
  - Web server thực hiện yêu cầu và trả lại kết quả cho trình duyệt
- Phần mềm chat
  - Server quản lý dữ liệu người dùng
  - Client gửi các yêu cầu đến server (đăng ký, đăng nhập, các đoạn chat)
  - Server thực hiện yêu cầu và trả lại kết quả cho client
    - Đăng ký
    - Đăng nhập
    - Chuyển tiếp dữ liệu giữa các client

# Ví dụ về các ứng dụng trên mạng

- Phần mềm nghe nhạc trên thiết bị di động
  - Server quản lý dữ liệu người dùng, lưu trữ các file âm thanh, xử lý các yêu cầu từ phần mềm di động, quản lý các kết nối.
  - Phần mềm di động gửi các yêu cầu và dữ liệu lên server, chờ kết quả trả về và xử lý.
- Phần mềm đồng bộ file giữa các thiết bị (Dropbox, Onedrive, ...)
  - Cài đặt phần mềm client trên PC
  - Đồng bộ thư mục và tập tin lên server
  - Theo dõi sự thay đổi của dữ liệu (từ phía server hoặc local) và cập nhật theo thời gian thực
- Phần mềm tăng tốc download IDM
  - Bắt và phân tích các gói tin được nhận bởi trình duyệt
  - Tách ra các liên kết quan tâm
  - Tải file bằng nhiều luồng song song

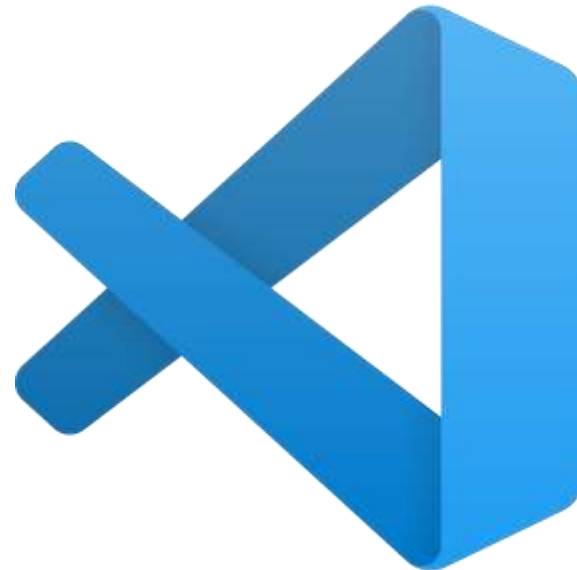
## Socket API

- Thư viện đa nền tảng, được hỗ trợ bởi các chương trình dịch trong các ngôn ngữ và hệ điều hành khác nhau.
- Thường sử dụng cùng với C/C++
- Cho hiệu năng cao nhất



# Các công cụ lập trình (trên nền tảng Linux/Unix)

- Trình biên dịch: **gcc**
- Công cụ soạn thảo mã nguồn: **Visual Studio Code**



- Wireshark: Công cụ phân tích gói tin
- Netcat: Công cụ tạo client/server thử nghiệm

- Sử dụng Netcat để gửi nhận dữ liệu đơn giản
- Netcat là một tiện ích mạng rất đa năng.
- Có thể sử dụng như TCP server:

```
nc -v -l -p <cổng đợi kết nối>
```

- Có thể sử dụng như TCP client:

```
nc -v <ip/tên miền> <cổng>
```

- Sử dụng như UDP receiver:

```
nc -v -l -u -p <cổng đợi kết nối>
```

- Sử dụng như UDP sender:

```
nc -v -u <ip/tên miền> <cổng>
```

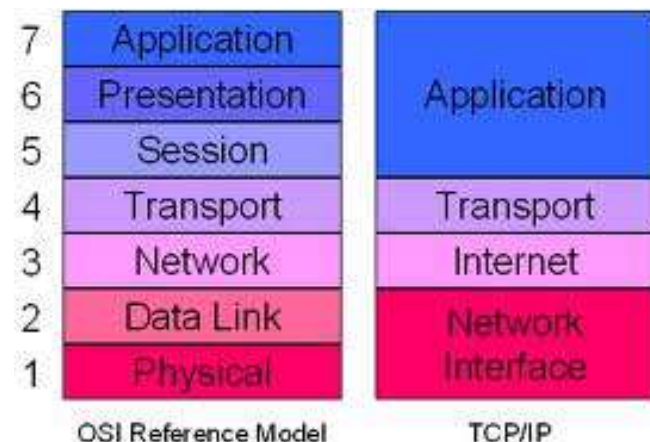
- <https://manpages.ubuntu.com/>
- Google/BING
- Stack Overflow

## 1.2. Nhắc lại một số kiến thức Mạng máy tính

- a. Giao thức TCP/IP
- b. Giao thức IPv4
- c. Giao thức IPv6

## a. Giao thức TCP/IP

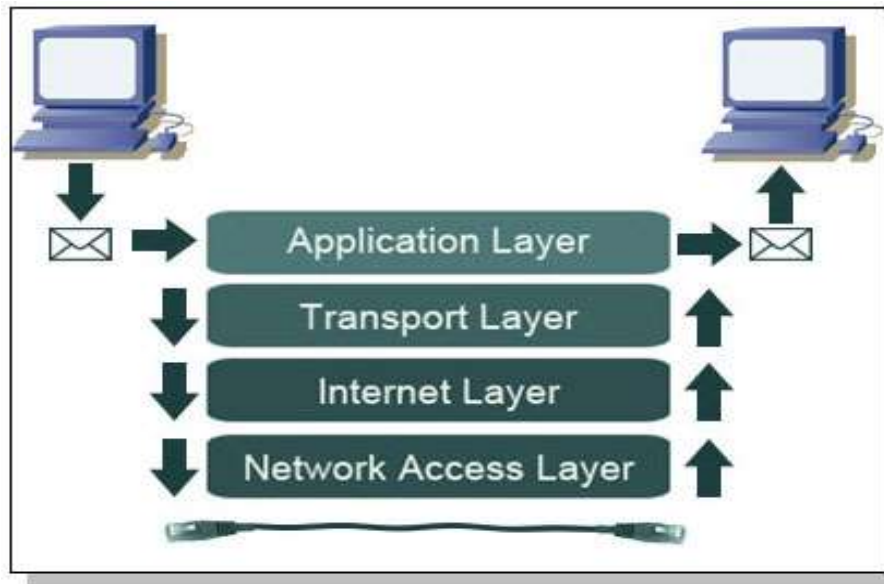
- TCP/IP: Transmission Control Protocol / Internet Protocol.
- Là bộ giao thức truyền thông được sử dụng trên Internet và hầu hết các mạng thương mại.
- Được chia thành các tầng gồm nhiều giao thức, thuận tiện cho việc quản lý và phát triển.
- Là thể hiện đơn giản hóa của mô hình lý thuyết OSI.



# Bộ giao thức Internet

Gồm bốn tầng

- Tầng ứng dụng – Application Layer.
- Tầng giao vận – Transport Layer.
- Tầng Internet – Internet Layer.
- Tầng truy nhập mạng – Network Access Layer.



## Tầng ứng dụng

- Đóng gói dữ liệu người dùng theo giao thức riêng và chuyển xuống tầng dưới.
- Các giao thức thông dụng: HTTP, FTP, SMTP, POP3, DNS, SSH, IMAP...
- Việc lập trình mạng sẽ xây dựng ứng dụng tuân theo một trong các giao thức ở tầng này hoặc giao thức do người phát triển tự định nghĩa



## Tầng giao vận

- Cung cấp dịch vụ truyền dữ liệu giữa ứng dụng - ứng dụng.
- Đơn vị dữ liệu là các đoạn (segment, datagram)
- Các giao thức ở tầng này: TCP, UDP.
- Việc lập trình mạng sẽ sử dụng dịch vụ do các giao thức ở tầng này cung cấp để truyền dữ liệu

## Tầng Internet

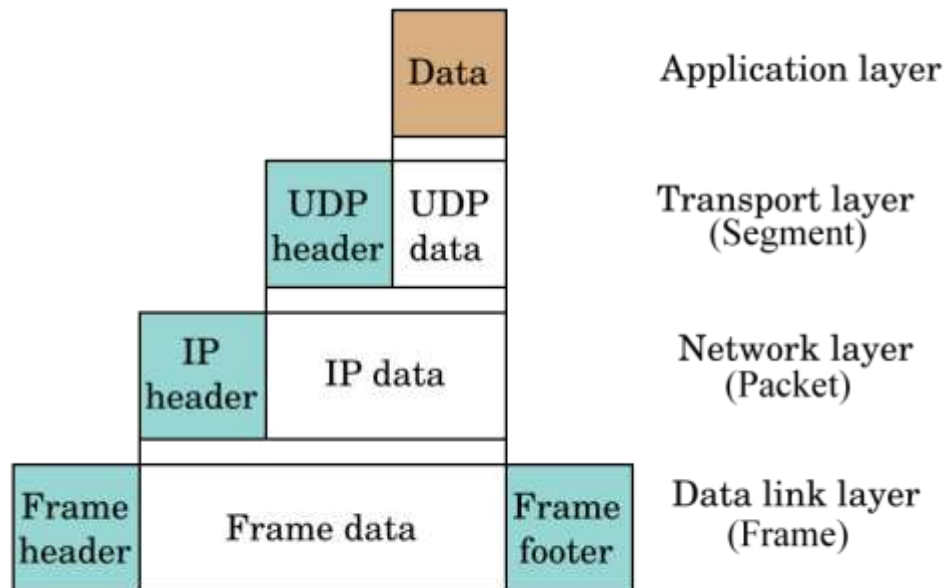
- Định tuyến và truyền các gói tin liên mạng.
- Cung cấp dịch vụ truyền dữ liệu giữa máy tính – máy tính trong cùng nhánh mạng hoặc giữa các nhánh mạng.
- Đơn vị dữ liệu là các gói tin (packet).
- Các giao thức ở tầng này: IPv4, IPv6
- Việc lập trình ứng dụng mạng sẽ rất ít khi can thiệp vào tầng này, trừ khi phát triển một giao thức liên mạng mới.

## Tầng truy nhập mạng

- Cung cấp dịch vụ truyền dữ liệu giữa các nút mạng trên cùng một nhánh mạng vật lý.
- Đơn vị dữ liệu là các khung (frame).
- Phụ thuộc rất nhiều vào phương tiện kết nối vật lý.
- Các giao thức ở tầng này đa dạng: MAC, LLC, ADSL, 802.11...
- Việc lập trình mạng ở tầng này là xây dựng các trình điều khiển phần cứng tương ứng, thường do nhà sản xuất thực hiện.

# Bộ giao thức Internet

- Dữ liệu gửi đi qua mỗi tầng sẽ được thêm phần thông tin điều khiển (header).
- Dữ liệu nhận được qua mỗi tầng sẽ được bóc tách thông tin điều khiển.



# Giao thức Internet (Internet Protocol)

- Giao thức mạng thông dụng nhất trên thế giới
- Chức năng
  - Định địa chỉ các máy chủ
  - Định tuyến các gói dữ liệu trên mạng
- Bao gồm 2 phiên bản: IPv4 và IPv6
- Thành công của Internet là nhờ IPv4
- Được hỗ trợ trên tất cả các hệ điều hành
- Là công cụ sử dụng để lập trình ứng dụng mạng

## b. Giao thức IPv4

- Được IETF công bố dưới dạng RFC 791 vào 9/1981.
- Phiên bản thứ 4 của họ giao thức IP và là phiên bản đầu tiên phát hành rộng rãi.
- Sử dụng trong hệ thống chuyển mạch gói.
- Truyền dữ liệu theo kiểu Best-Effort: không đảm bảo tính trật tự, trùng lặp, tin cậy của gói tin.
- Kiểm tra tính toàn vẹn của dữ liệu qua checksum

# IPv4 header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification															Flags			Fragment Offset													
8	64	Time To Live								Protocol							Header Checksum																
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Một số trường cần quan tâm:

**Version** (4 bit): có giá trị là 4 với IPv4

**IHL** – Internet Header Length (4 bit): chiều dài của header, tính bằng số từ nhớ 32 bit

**Total Length** (16 bit): kích thước của gói tin (theo bytes) bao gồm cả header và data

**Protocol**: giao thức được sử dụng ở tầng trên (nằm trong phần data)

**Source IP Address**: địa chỉ IP nguồn

**Destination IP Address**: địa chỉ IP đích

# IPv4 header – Ví dụ

Gói tin IPv4 có header như sau:

**45 00 00 40**

**7c da 40 00**

**80 06 fa d8**

**c0 a8 0f 0b**

**bc ac f6 a4**

Xác định các thông tin liên quan đến gói tin này: Header length, Total length, Protocol, Source IP, Destination IP address.



# Địa chỉ IPv4

- Sử dụng 32 bit để đánh địa chỉ các máy tính trong mạng.
- Bao gồm: phần mạng và phần host.
- Số địa chỉ tối đa:  $2^{32} \sim 4,294,967,296$ .
- Dành riêng một vài dải đặc biệt không sử dụng.
- Chia thành bốn nhóm 8 bit (octet).

Dạng biểu diễn	Giá trị
Nhị phân	11000000.10101000.00000000.00000001
Thập phân	192.168.0.1
Thập lục phân	0xC0A80001

# Các lớp địa chỉ IPv4

- Có năm lớp địa chỉ: A, B, C, D, E.
- Lớp A, B, C: trao đổi thông tin thông thường.
- Lớp D: multicast
- Lớp E: để dành

Lớp	MSB	Địa chỉ đầu	Địa chỉ cuối
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	240.0.0.0	255.255.255.255

# Mặt nạ mạng (Network Mask)

- Phân tách phần mạng và phần host trong địa chỉ IPv4.
- Sử dụng trong bộ định tuyến để tìm đường đi cho gói tin.
- Với mạng có dạng

Network	Host
192.168.0.	1
11000000.10101000.00000000.	00000001

# Mặt nạ mạng (Network Mask)

- Biểu diễn theo dạng /n
  - n là số bit dành cho phần mạng.
  - Ví dụ: 192.168.0.1/24
- Biểu diễn dưới dạng nhị phân
  - Dùng 32 bit đánh dấu, bit dành cho phần mạng là 1, cho phần host là 0.
  - Ví dụ: 11111111.11111111.11111111.00000000 hay 255.255.255.0
- Biểu diễn dưới dạng Hexa
  - Dùng số Hexa: 0xFFFFFFFF
  - Ít dùng

# Số lượng địa chỉ trong mỗi mạng

- Mỗi mạng sẽ có  $n$  bit dành cho phần mạng,  $32-n$  bit dành cho phần host.
- Phân phối địa chỉ trong mỗi mạng:
  - 01 địa chỉ mạng (các bit phần host bằng 0).
  - 01 địa chỉ quảng bá (các bit phần host bằng 1).
  - $2^{32-n} - 2$  địa chỉ gán cho các máy trạm (host).
- Với mạng 192.168.0.1/24
  - Địa chỉ mạng: 192.168.0.0
  - Địa chỉ quảng bá: 192.168.0.255
  - Địa chỉ host: 192.168.0.1 - 192.168.0.254

# Các dải địa chỉ đặc biệt

- Là những dải được dùng với mục đích riêng, không sử dụng được trên Internet.

Địa chỉ	Diễn giải
10.0.0.0/8	Mạng riêng
127.0.0.0/8	Địa chỉ loopback
172.16.0.0/12	Mạng riêng
192.168.0.0/16	Mạng riêng
224.0.0.0/4	Multicast
240.0.0.0/4	Dự trữ

# Dải địa chỉ cục bộ

- Chỉ sử dụng trong mạng nội bộ.

Tên	Dải địa chỉ	Số lượng	Mô tả mạng	Viết gọn
Khối 24-bit	10.0.0.0 – 10.255.255.255	16,777,216	Một dải trọn vẹn thuộc lớp A	10.0.0.0/8
Khối 20-bit	172.16.0.0 – 172.31.255.255	1,048,576	Tổ hợp từ mạng lớp B	172.16.0.0/12
Khối 16-bit	192.168.0.0 – 192.168.255.255	65,536	Tổ hợp từ mạng lớp C	192.168.0.0/16

## c. Giao thức IPv6

- IETF đề xuất năm 1998.
- Khắc phục vấn đề thiếu địa chỉ của IPv4.
- Đang dần phổ biến và chưa thể thay thế hoàn toàn IPv4.
- Sử dụng 128 bit để đánh địa chỉ các thiết bị, dưới dạng các cụm số hexa phân cách bởi dấu :

Ví dụ:

**FEDC : BA98 : 768A : 0C98 : FEBA : CB87 : 7678 : 1111**



# Quy tắc rút gọn địa chỉ IPv6

- Cho phép bỏ các số 0 nằm trước mỗi nhóm (octet).
- Thay bằng số 0 cho nhóm có toàn số 0.
- Thay bằng dấu “::” cho các nhóm liên tiếp nhau có toàn số 0.
- Chú ý: Dấu “::” chỉ sử dụng được 1 lần trong toàn bộ địa chỉ IPv6

Ví dụ: **1080:0000:0000:0070:0000:0989:CB45:345F**

Có thể viết tắt thành

**1080::70:0:989:CB45:345F** hoặc

**1080:0:0:70::989:CB45:345F**

# Giao thức TCP

- Giao thức lõi chạy ở tầng giao vận.
- Chạy bên dưới tầng ứng dụng và trên nền IP
- Cung cấp dịch vụ truyền dữ liệu theo dòng tin cậy giữa các ứng dụng.
- Được sử dụng bởi hầu hết các ứng dụng mạng.
- Chia dữ liệu thành các gói nhỏ, thêm thông tin kiểm soát và gửi đi trên đường truyền.
- Lập trình mạng sẽ sử dụng giao thức này để trao đổi thông tin.

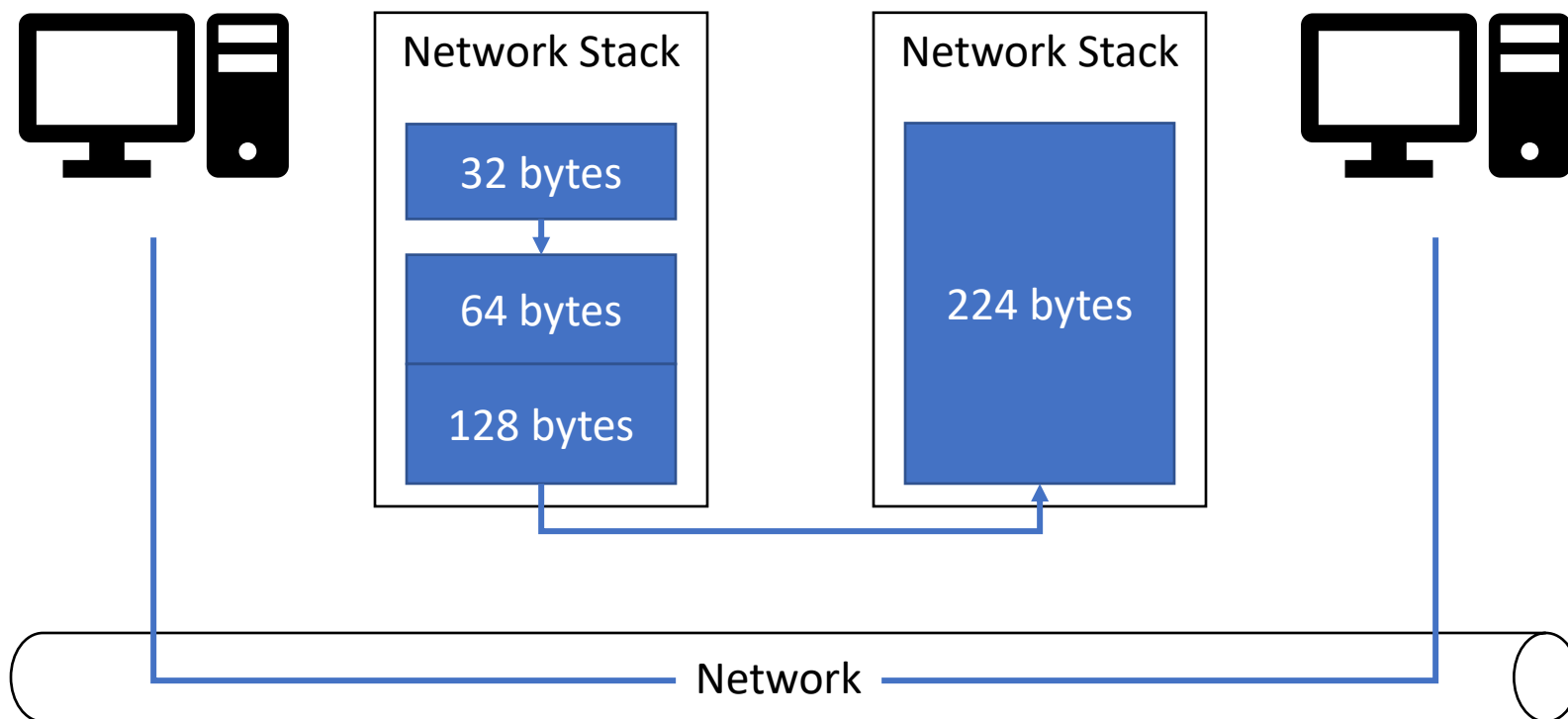
# Cổng (Port)

- Một số nguyên duy nhất trong khoảng 0 - 65535 tương ứng với một kết nối của ứng dụng.
- TCP sử dụng cổng để chuyển dữ liệu tới đúng ứng dụng hoặc dịch vụ.
- Một ứng dụng có thể mở nhiều kết nối => có thể sử dụng nhiều cổng.
- Một số cổng thông dụng: HTTP(80), FTP(21, 20), SMTP(25), POP3(110), HTTPS(443) ...

- Hướng kết nối (connection oriented)
  - Hai bên phải thiết lập kênh truyền trước khi truyền dữ liệu.
  - Được thực hiện bởi quá trình gọi là bắt tay ba bước (three ways handshake).
- Truyền dữ liệu theo dòng (stream oriented): tự động phân chia dòng dữ liệu thành các đoạn nhỏ để truyền đi, tự động ghép các đoạn nhỏ thành dòng dữ liệu và gửi trả ứng dụng.
- Đúng trật tự (ordering guarantee): dữ liệu gửi trước sẽ được nhận trước

# Đặc tính của TCP

- Tin cậy, chính xác: thông tin gửi đi sẽ được đảm bảo đến đích, không dư thừa, sai sót...
- Độ trễ lớn, khó đáp ứng được tính thời gian thực.



# Header của TCP

## Chứa thông tin về đoạn dữ liệu tương ứng

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if data offset > 5. Padded at the end with "0" bytes if necessary.)																															
...	...	...																															

Một số trường cần quan tâm:

**Source port:** cổng gửi dữ liệu

**Destination port:** cổng nhận dữ liệu

**Data offset:** độ dài TCP header tính bằng số từ 32-bit

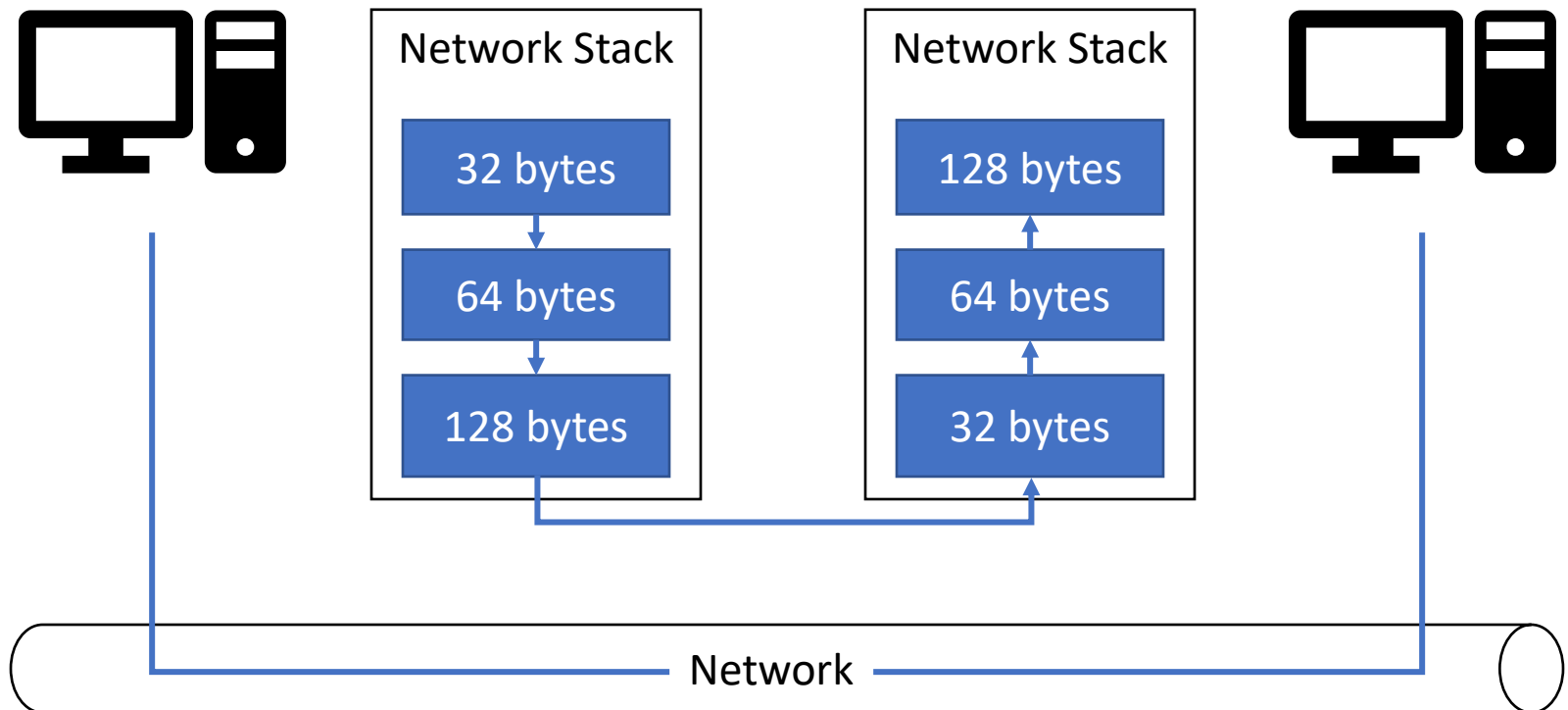
# Các dịch vụ trên nền TCP

Rất nhiều dịch vụ chạy trên nền TCP: FTP (21), HTTP (80), SMTP (25), SSH (22), POP3 (110), VNC (4899)...



# Giao thức UDP

- Cũng là giao thức lỗi trong TCP/IP.
- Cung cấp dịch vụ truyền dữ liệu giữa các ứng dụng.
- UDP chia nhỏ dữ liệu ra thành các datagram





# Đặc tính của UDP

- Không cần thiết lập kết nối trước khi truyền (connectionless).
- Nhanh, chiếm ít tài nguyên để xử lý.
- Sử dụng trong các ứng dụng khắt khe về mặt thời gian, chấp nhận sai sót: audio, video, game ...
- Hạn chế:
  - Không có cơ chế báo gửi (report).
  - Không đảm bảo trật tự các datagram (ordering).
  - Không phát hiện được mất mát hoặc trùng lặp thông tin (loss, duplication).

# Header của UDP

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Một số trường cần quan tâm:

**Source port:** cổng gửi dữ liệu

**Destination port:** cổng nhận dữ liệu

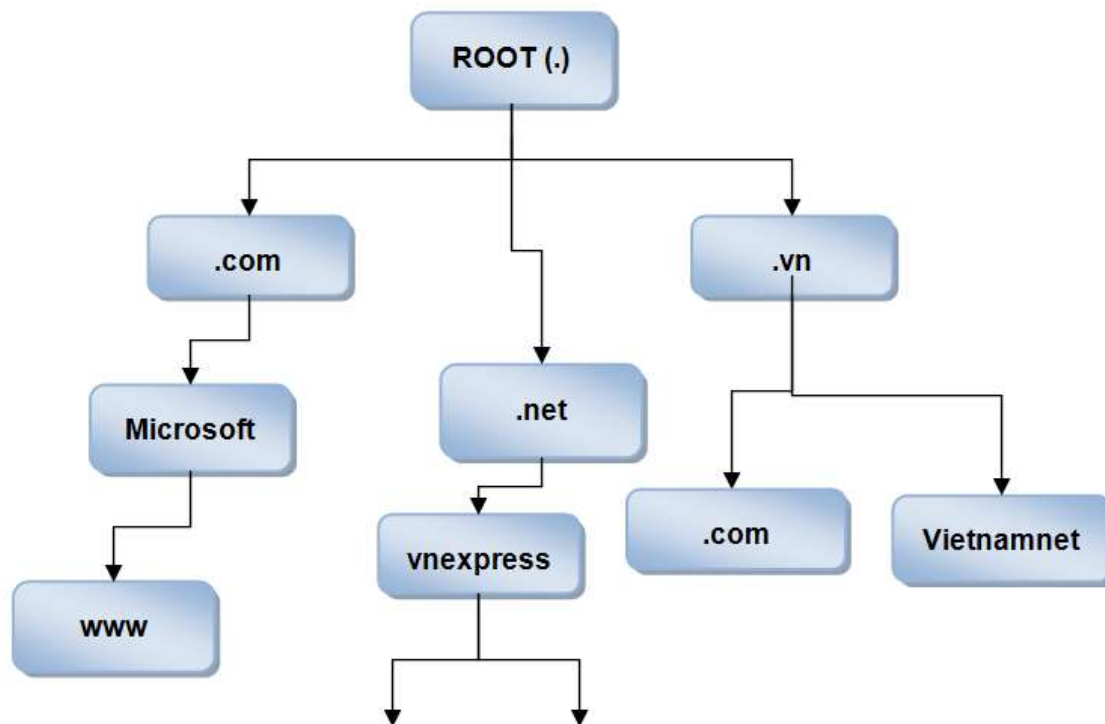
**Length:** độ dài của gói tin UDP (header luôn có kích thước cố định là 8 bytes)

# Các dịch vụ trên nền UDP

- Phân giải tên miền: DNS (53)
- Streaming: MMS, RTSP...
- Game

# Hệ thống phân giải tên miền DNS

- Địa chỉ IP khó nhớ với con người.
- DNS – Domain Name System: Hệ thống phân cấp làm nhiệm vụ ánh xạ tên miền sang địa chỉ IP và ngược lại.



# Hệ thống phân giải tên miền DNS

- Các tên miền được phân cấp và quản lý bởi INTERNIC
- Cấp cao nhất là ROOT, sau đó là cấp 1, cấp 2, ...
- Ví dụ: www.hust.edu.vn

Cấp	Cấp 4	Cấp 3	Cấp 2	Cấp 1
Tên miền	www.	hust.	edu.	vn

# Hệ thống phân giải tên miền DNS

- Tổ chức được cấp tên miền cấp 1 sẽ duy trì cơ sở dữ liệu các tên miền cấp 2 trực thuộc, tổ chức được cấp tên miền cấp 2 sẽ duy trì cơ sở dữ liệu các tên miền cấp 3 trực thuộc...
- Một máy tính muốn biết địa chỉ của một máy chủ có tên miền nào đó, nó sẽ hỏi máy chủ DNS mà nó nằm trong, nếu máy chủ DNS này không trả lời được nó sẽ chuyển tiếp câu hỏi đến máy chủ DNS cấp cao hơn, DNS cấp cao hơn nếu không trả lời được lại chuyển đến DNS cấp cao hơn nữa...

# Hệ thống phân giải tên miền DNS

- Việc truy vấn DNS sẽ do hệ điều hành thực hiện.
- Dịch vụ DNS chạy ở cổng 53 UDP.
- Công cụ thử nghiệm: nslookup

```
lebavui@Home-Desktop: ~  
lebavui@Home-Desktop:~$ nslookup google.com  
Server:          203.113.131.2  
Address:         203.113.131.2#53  
  
Non-authoritative answer:  
Name:   google.com  
Address: 142.251.220.110  
Name:   google.com  
Address: 2404:6800:4005:80a::200e  
  
lebavui@Home-Desktop:~$ nslookup hust.edu.vn  
Server:          203.113.131.2  
Address:         203.113.131.2#53  
  
Non-authoritative answer:  
Name:   hust.edu.vn  
Address: 202.191.57.199  
  
lebavui@Home-Desktop:~$
```

## 1.3. Lập trình trong môi trường Ubuntu/Linux

- Ngôn ngữ lập trình: **C/C++**
- Trình biên dịch: **gcc**
- Trình gỡ lỗi: **gdb**
- Công cụ soạn thảo mã nguồn: **Visual Studio Code**
- Công cụ quản lý phiên bản: **GitHub/Git**
- Hệ điều hành hỗ trợ: **Windows, Ubuntu, Mac OS**



## 1.3.1. Trình biên dịch gcc, trình gỡ lỗi gdb

- GNU Compiler Collection (GCC) là tập hợp các trình biên dịch được thiết kế cho nhiều ngôn ngữ lập trình khác nhau (C, C++, Objective C, Fortran, Ada, Go, ...)
- Tương thích với nhiều nền tảng kiến trúc máy tính.
- Cú pháp cơ bản để dịch chương trình

**`gcc -o [executable_name] [source_file].c`**

- GNU Debugger (GDB) là chương trình gỡ lỗi có thể chạy trên nhiều hệ điều hành và sử dụng cho nhiều ngôn ngữ lập trình.

## 1.3.2. Cài đặt các môi trường lập trình

### Hệ điều hành Windows:

1. Cài đặt Windows Subsystem for Linux ([MS Store](#))
2. Cài đặt Ubuntu ([MS Store](#))
3. Cài đặt Windows Terminal ([MS Store](#))
4. Cài đặt trình biên dịch gcc trong môi trường Ubuntu
5. Cài đặt phần mềm Visual Studio Code ([MS Store](#))

#### Cài đặt gcc trong môi trường Ubuntu

+ Cập nhật gói phần mềm

**sudo apt update**

+ Cài đặt gói phần mềm trình biên dịch

**sudo apt install build-essential**

+ Cài đặt công cụ tra cứu

**sudo apt install manpages-dev**

+ Kiểm tra kết quả cài đặt

**gcc --version**



## 1.3.2. Cài đặt các môi trường lập trình

### Hệ điều hành Ubuntu:

1. Cài đặt phần mềm Visual Studio Code (Ubuntu Software)
2. Cài đặt trình biên dịch gcc

#### Cài đặt gcc trong môi trường Ubuntu

+ Cập nhật gói phần mềm

**sudo apt update**

+ Cài đặt gói phần mềm trình biên dịch

**sudo apt install build-essential**

+ Cài đặt công cụ tra cứu

**sudo apt install manpages-dev**

+ Kiểm tra kết quả cài đặt

**gcc --version**

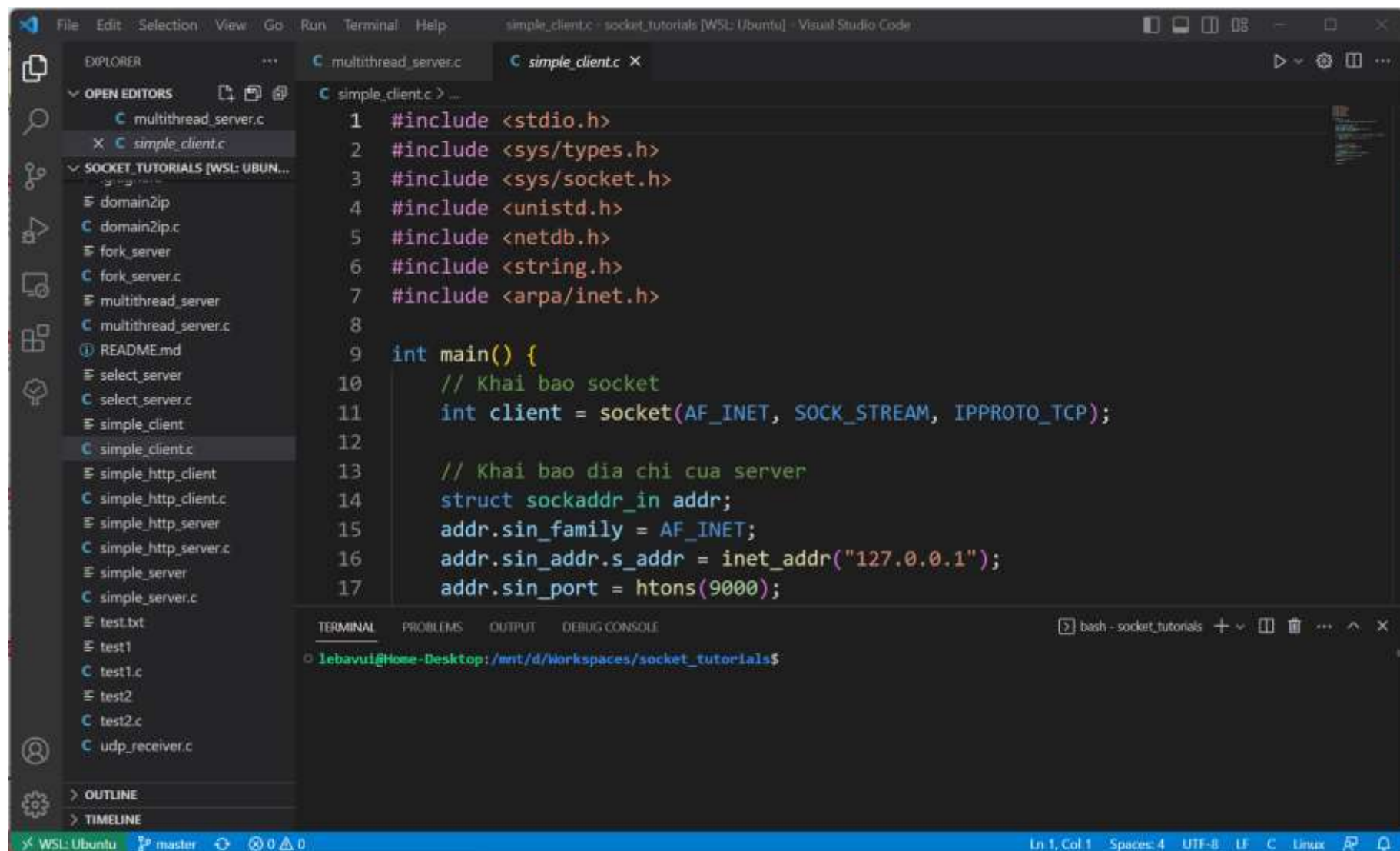
## 1.3.2. Cài đặt các môi trường lập trình

### Hệ điều hành MacOS:

1. Cài đặt phần mềm Visual Studio Code
2. Cài đặt trình biên dịch gcc

```
+ Cài đặt công cụ homebrew
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
+ Cài đặt gói phần mềm trình biên dịch
brew install gcc
+ Cài đặt công cụ tra cứu
brew install manpages-dev
+ Kiểm tra kết quả cài đặt
gcc --version
```

# 1.3.3. Sử dụng phần mềm Visual Studio Code



```
File Edit Selection View Go Run Terminal Help simple_client.c - socket_tutorials [WSL: Ubuntu] - Visual Studio Code

EXPLORER
OPEN EDITORS
  C multithread_server.c
  X C simple_client.c
SOCKET TUTORIALS [WSL: UBUN...
  domain2ip
  C domain2ip.c
  fork_server
  C fork_server.c
  multithread_server
  C multithread_server.c
  README.md
  select_server
  C select_server.c
  simple_client
  C simple_client.c
  simple_http_client
  C simple_http_client.c
  simple_http_server
  C simple_http_server.c
  simple_server
  C simple_server.c
  test.txt
  test1
  C test1.c
  test2
  C test2.c
  udp_receiver.c
OUTLINE
TIMELINE

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <unistd.h>
5 #include <netdb.h>
6 #include <string.h>
7 #include <arpa/inet.h>
8
9 int main() {
10     // Khai bao socket
11     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
12
13     // Khai bao dia chi cua server
14     struct sockaddr_in addr;
15     addr.sin_family = AF_INET;
16     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
17     addr.sin_port = htons(9000);

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
bash - socket_tutorials
lebauvi@Home-Desktop: /mnt/d/Workspaces/socket_tutorials$

Ln 1, Col 1 Spaces: 4 UTF-8 LF C Linux
```

## 1.3.3. Sử dụng phần mềm Visual Studio Code

Visual Studio Code:

- Cài đặt các Extension (WSL)
- Soạn thảo mã nguồn
- Dịch, chạy và gỡ lỗi
- Quản lý phiên bản mã nguồn

## 1.3.4. Nhắc lại một số kỹ thuật lập trình C

- a. Mảng và con trỏ
- b. Xử lý chuỗi ký tự
- c. Kiểu dữ liệu cấu trúc
- d. Khai báo và sử dụng hàm
- e. Đọc, ghi file
- f. Truyền tham số dòng lệnh

## a. Mảng và con trỏ

- **Khái niệm mảng:** Tập hợp các phần tử có cùng kiểu, được đặt liên tiếp trong bộ nhớ.
- Mỗi phần tử của mảng được xác định thông qua:
  - Tên mảng
  - Chỉ số của phần tử trong mảng



```
kiểu_dữ_liệu tên_mảng[kích_thước_mảng];
```

- **kiểu\_dữ\_liệu**: kiểu của các phần tử trong mảng (*số nguyên, số thực, kí tự, chuỗi, mảng,...*)
- **tên\_mảng**: tên của mảng
- **kích\_thước\_mảng**: số phần tử tối đa trong mảng

Ví dụ

```
int diem_thi[50];    // Khai báo mảng 50 phần tử  
    có kiểu dữ liệu int
```

```
float A[10]; // Mảng 10 phần tử kiểu số thực
```

# Cấp phát bộ nhớ cho mảng

- Các phần tử trong mảng được cấp phát các ô nhớ kế tiếp nhau trong bộ nhớ.
- Kích thước của mảng bằng kích thước một phần tử nhân với số phần tử được cấp phát.

Ví dụ:

```
int A[10]; // Mảng A gồm 10 phần tử nguyên  
=> Kích thước của mảng A:  $10 \times 4 = 40$  bytes
```

# Truy nhập phần tử của mảng

- Biến mảng lưu trữ địa chỉ ô nhớ đầu tiên trong vùng nhớ được cấp phát.
- Ngôn ngữ C đánh chỉ số các phần tử trong mảng bắt đầu từ 0.
- Các phần tử của mảng được truy nhập thông qua:
  - Tên mảng
  - Chỉ số của phần tử trong mảng

`tên_mảng[chỉ_số_phần_tử];`

- Nhập/xuất dữ liệu cho mảng
  - Mảng 1 chiều, ma trận (mảng 2 chiều)
- Bài toán đếm
  - Đếm số phần tử
  - Tính toán trên các phần tử
- Tìm kiếm phần tử
  - Lớn nhất/nhỏ nhất/bất kỳ
- Sắp xếp phần tử trong mảng
  - Theo thứ tự, theo nguyên tắc
- Chèn thêm phần tử, xóa phần tử

- Bộ nhớ gồm dãy các ô nhớ
  - Mỗi ô nhớ lưu trữ một byte dữ liệu
  - Mỗi ô nhớ có một địa chỉ riêng
- Các biến trong chương trình được lưu tại vùng nhớ nào đó trong bộ nhớ
- Khi khai báo biến, tùy thuộc vào kiểu, biến sẽ được cấp phát một số ô nhớ liên tục nhau
  - VD: Biến int được cấp 4 bytes, float được cấp 4 bytes,...
  - Địa chỉ của biến, là địa chỉ của byte đầu tiên trong số các byte được cấp
  - Khi gán giá trị cho biến, nội dung các byte cung cấp cho biến sẽ thay đổi

# Ví dụ về địa chỉ

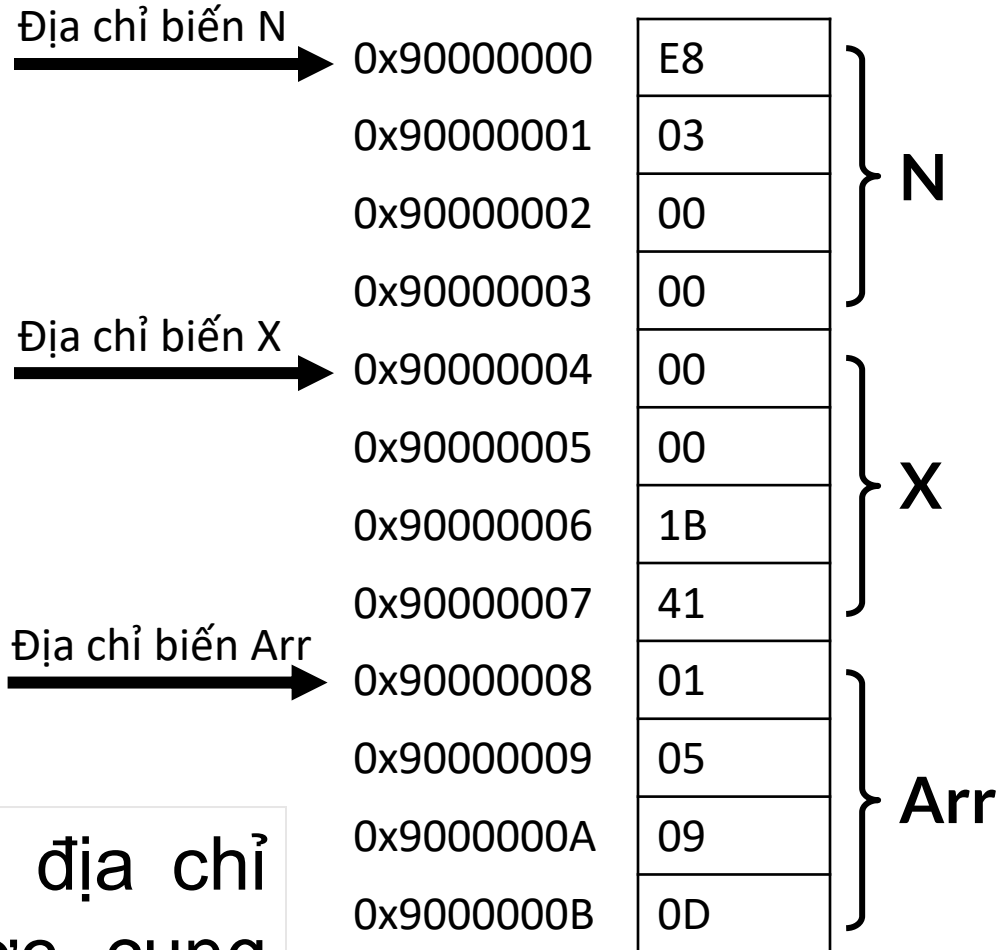
```
int N;  
float x;  
char Arr[4];
```

```
N=1000; // 0x000003E8
```

```
X=9.6875; // 411B0000
```

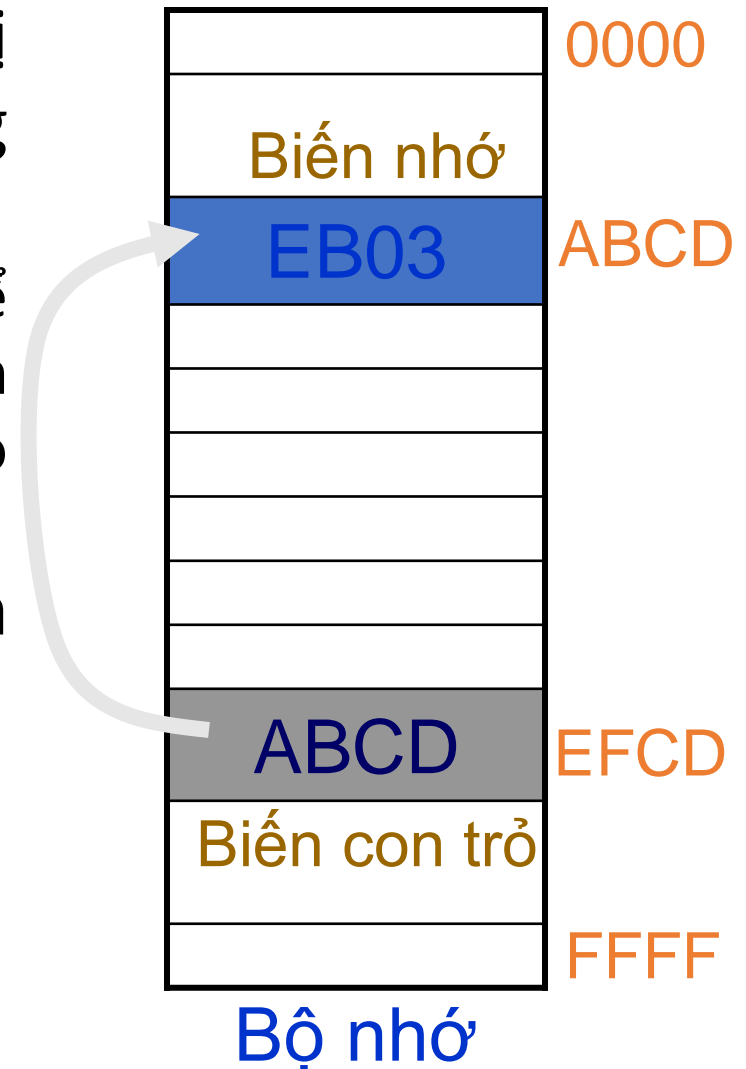
```
for(i=0; i<4; i++)  
    Arr[i]=4*i+1;
```

Địa chỉ của một biến là địa chỉ byte nhớ đầu tiên được cung cấp cho biến để lưu trữ dữ liệu.



# Con trỏ

- Con trỏ là một biến mà giá trị của nó là địa chỉ của một vùng nhớ
  - Vùng nhớ này có thể dùng để chứa các biến có kiểu cơ bản (nguyên, thực, kí tự, ...) hay có cấu trúc (mảng, bản ghi, ...)
- Con trỏ dùng “trỏ tới” một biến nhớ
  - Có thể trỏ tới một hàm
  - Có thể trỏ tới con trỏ khác



**kiểu \* tên;**

- **tên:** Tên của một biến con trỏ
- **kiểu:** Kiểu của biến mà con trỏ “tên” trỏ tới
  - Giá trị của con trỏ có thể thay đổi được
    - Trỏ tới các biến khác nhau, có cùng kiểu
  - Kiểu biến mà con trỏ trỏ tới không thay đổi được
    - Muốn thay đổi phải thực hiện “ép kiểu”
- Ví dụ:

```
int * pi; // Con trỏ, trỏ tới một biến kiểu nguyên  
char * pc; // Con trỏ, trỏ tới một biến kiểu kí tự
```



# Toán tử địa chỉ (&)

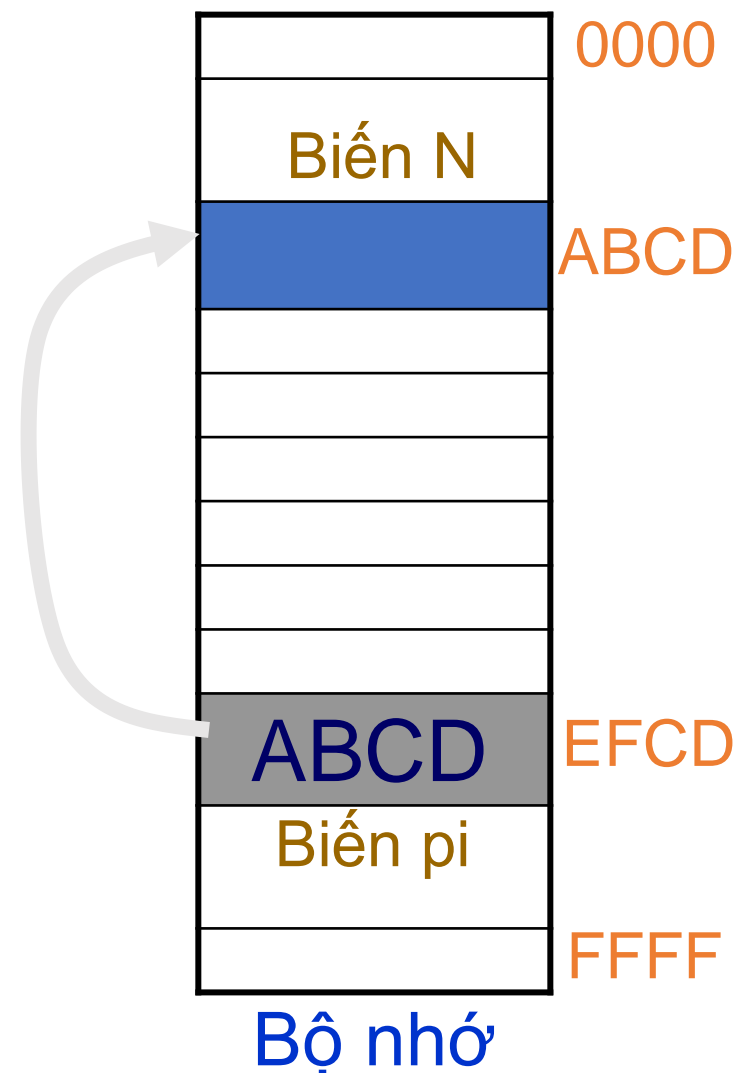
- Kí hiệu: &
- Là toán tử một ngôi, trả về địa chỉ của biến  
Địa chỉ biến có thể được gán cho một con trỏ, trỏ tới đối tượng cùng kiểu.

- Ví dụ

```
short int N; // &N → ABCD
```

```
short int *pi;
```

```
pi = &N; // pi ← ABCD
```



# Toán tử nội dung (\*)

- Kí hiệu: \*
- Là toán tử một ngôi, trả về giá trị (nội dung) của vùng nhớ mà con trỏ đang trỏ tới.

- Ví dụ

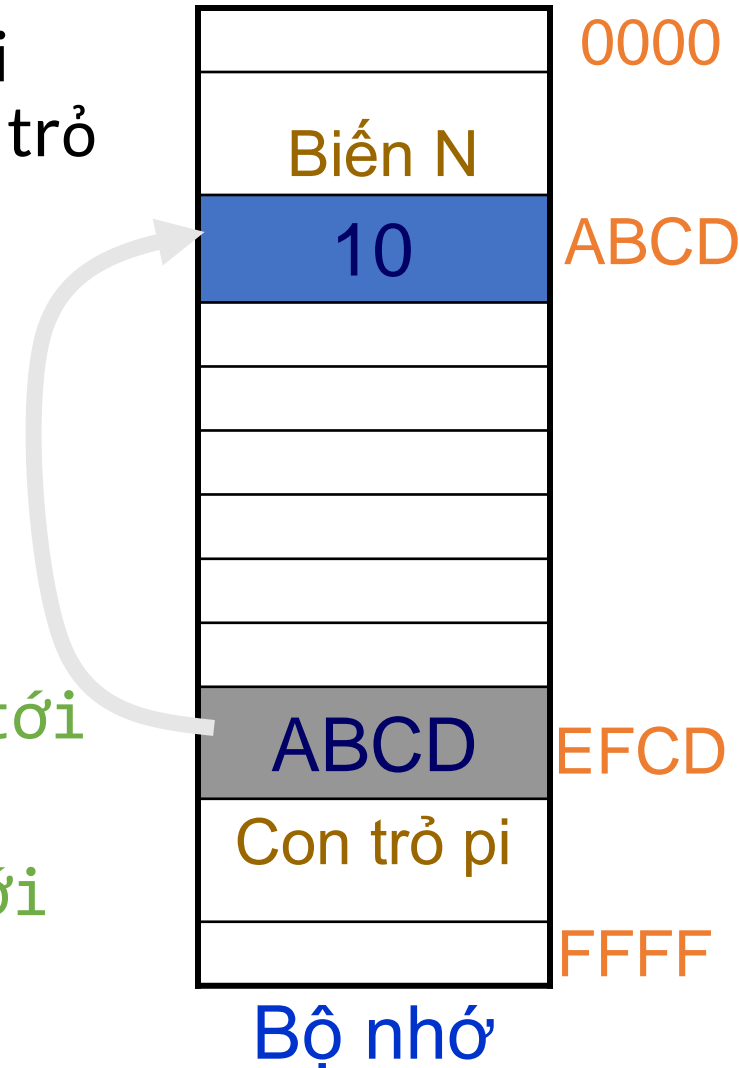
```
short int N;
```

```
short int * pi;
```

```
pi = &N;
```

```
N = 10; // Vùng nhớ mà pi trỏ tới  
mang giá trị 10; Vậy *pi=10
```

```
*pi = 20; // Vùng nhớ pi trỏ tới  
được gán giá trị 20; Vậy N=20
```



# Toán tử nội dung (\*)

- Kí hiệu: \*
- Là toán tử một ngôi, trả về giá trị (nội dung) của vùng nhớ mà con trỏ đang trỏ tới.

- Ví dụ

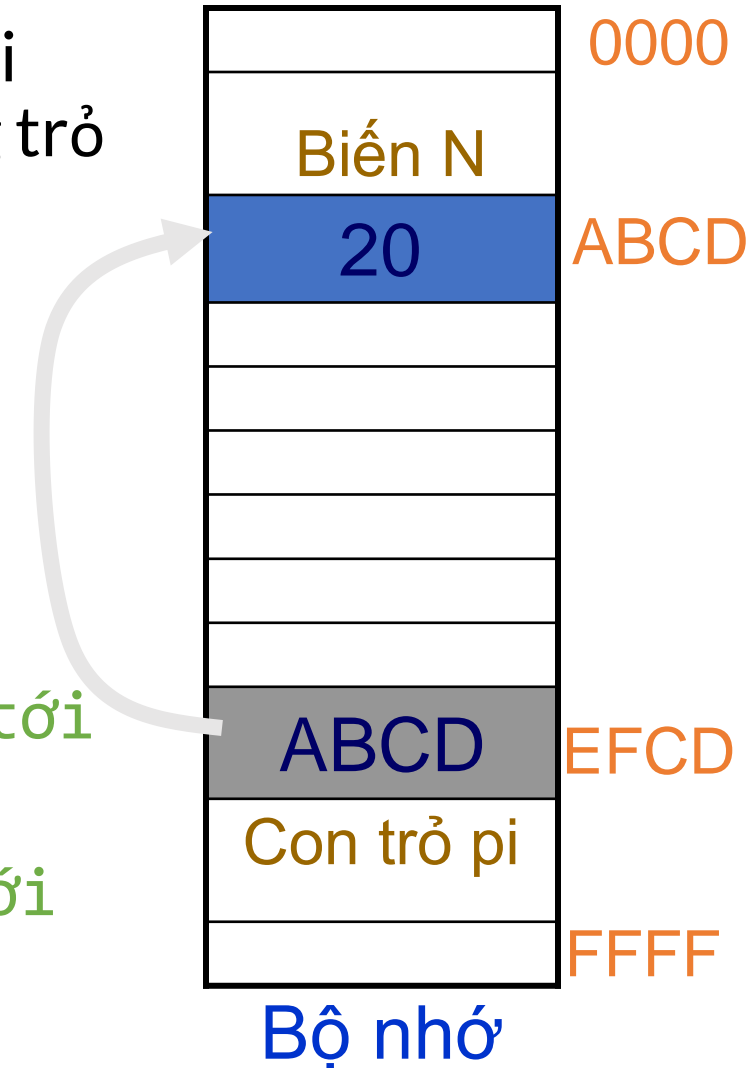
```
short int N;
```

```
short int * pi;
```

```
pi = &N;
```

```
N = 10; // Vùng nhớ mà pi trỏ tới  
mang giá trị 10; Vậy *pi=10
```

```
*pi = 20; // Vùng nhớ pi trỏ tới  
được gán giá trị 20; Vậy N=20
```



# Gán giá trị cho con trỏ

- Con trỏ được gán địa chỉ của một biến
  - Biến cùng kiểu với kiểu mà con trỏ trỏ tới  
Nếu không, cần phải ép kiểu
- Con trỏ được gán giá trị của con trỏ khác
  - Hai con trỏ sẽ trỏ tới cùng một biến (do cùng địa chỉ)
  - Hai con trỏ nên cùng kiểu trỏ đến  
Nếu không, phải ép kiểu
- Con trỏ được gán giá trị NULL  
Ví dụ: `int *p; p = 0;`
- Gán nội dung vùng nhớ 2 con trỏ trỏ tới.  
Ví dụ: `int *p1, *p2; *p1 = *p2;`

# Cấp phát bộ nhớ động

- Hàm **malloc()** cấp phát bộ nhớ.
  - Hàm **realloc()** cấp phát lại bộ nhớ đã cấp phát trước đó.
  - Hàm **free()** giải phóng bộ nhớ đã cấp phát.
- => Cần khai báo tệp tiêu đề `stdlib.h`

- Hàm **memset()** thiết lập nội dung cho vùng nhớ
  - Hàm **memcpy()** sao chép nội dung giữa 2 vùng nhớ
- => Cần khai báo tệp tiêu đề `string.h`

## b. Xử lý chuỗi ký tự

Một số hàm xử lý chuỗi ký tự thường dùng:

Hàm **strlen()** xác định độ dài chuỗi

Hàm **strcpy()/strncpy()** sao chép chuỗi

Hàm **strcmp()/strncmp()** so sánh chuỗi

Hàm **strcat()/strncat()** ghép chuỗi

Hàm **strchr()** tìm ký tự trong chuỗi

Hàm **strstr()** tìm chuỗi con trong chuỗi

Hàm **strtok()** tách chuỗi

Hàm **sscanf()** đọc dữ liệu có định dạng trong chuỗi

Hàm **sprintf()** ghi dữ liệu có định dạng vào chuỗi

\* Cần khai báo tệp tiêu đề **string.h**

# Ví dụ xử lý mảng, xâu ký tự

- VD1: Cho mảng chứa nội dung là header của gói tin IP hãy in ra giá trị version, ihl, total\_length, địa chỉ IP nguồn và đích.
- VD2: Lệnh gửi từ client là chuỗi ký tự có dạng “**CMD X Y**” trong đó CMD là các lệnh ADD, SUB, MUL, DIV, X và Y là 2 số thực. Viết đoạn chương trình kiểm tra một chuỗi ký tự có theo cú pháp trên không, xác định các giá trị của CMD, X và Y.
- VD3: Chuỗi ký tự sau là phản hồi của lệnh PASV trong giao thức FTP, hãy xác định giá trị địa chỉ IP và cổng.

**227 Entering Passive Mode (213,229,112,130,216,4)**

## c. Kiểu dữ liệu cấu trúc

- Khai báo cấu trúc dữ liệu:

```
struct tên_cấu_trúc {  
    kiểu_dữ_liệu tên_trường_1;  
    kiểu_dữ_liệu tên_trường_2;  
    ...  
};
```

- Đặt tên kiểu dữ liệu:

```
typedef kiểu_dữ_liệu tên_mới;
```

- Khai báo biến, con trỏ kiểu cấu trúc:

```
struct tên_cấu_trúc biến_cấu_trúc, *con_trỏ_cấu_trúc;
```

- Truy nhập trường từ biến, con trỏ kiểu cấu trúc:

```
biến_cấu_trúc.tên_trường
```

```
con_trỏ_cấu_trúc->tên_trường
```



## d. Khai báo và sử dụng hàm

- Khai báo hàm

**kiểu\_hàm tên\_hàm(danh\_sách\_tham\_số)**

**{**

**[các khai báo cục bộ]**

**[các câu lệnh]**

**}**

- Lệnh **return** được sử dụng để kết thúc hàm
- Sử dụng hàm:

**tên\_hàm(danh\_sách\_tham\_số\_thực)**

- Khai báo và truyền tham số cho hàm theo kiểu tham trị (giá trị của biến) và tham chiếu (địa chỉ của biến)



## e. Đọc, ghi file

- Khai báo con trỏ file: **FILE \*f;**
- Mở file: hàm **fopen()** mở file với các chế độ khác nhau
- Đọc dữ liệu từ file:
  - Hàm **fread()**: đọc theo từng mảng
  - Hàm **fscanf()**: đọc dữ liệu định dạng
  - Hàm **fgets()**: đọc một dòng
  - Hàm **fgetc()**: đọc một ký tự
- Ghi dữ liệu vào file:
  - Hàm **fwrite()**: ghi vào một mảng
  - Hàm **fprintf()**: ghi dữ liệu định dạng
- Xác định vị trí con trỏ file: hàm **ftell()**
- Di chuyển con trỏ file: hàm **fseek()**
- Kiểm tra kết thúc file: hàm **feof()**
- Đóng file: hàm **fclose()**

## f. Truyền tham số dòng lệnh

- Truyền tham số cho chương trình khi chạy ở chế độ dòng lệnh

```
int main(int argc, char *argv[])  
{  
    // argc là số tham số đã truyền  
    // argv là mảng các tham số kiểu chuỗi  
    // tham số đầu tiên là tên file thực thi  
}
```

# Chương 2. Lập trình socket cơ bản

# Chương 2. Lập trình socket cơ bản

2.1. Khái niệm socket

2.2. Cấu trúc địa chỉ của socket

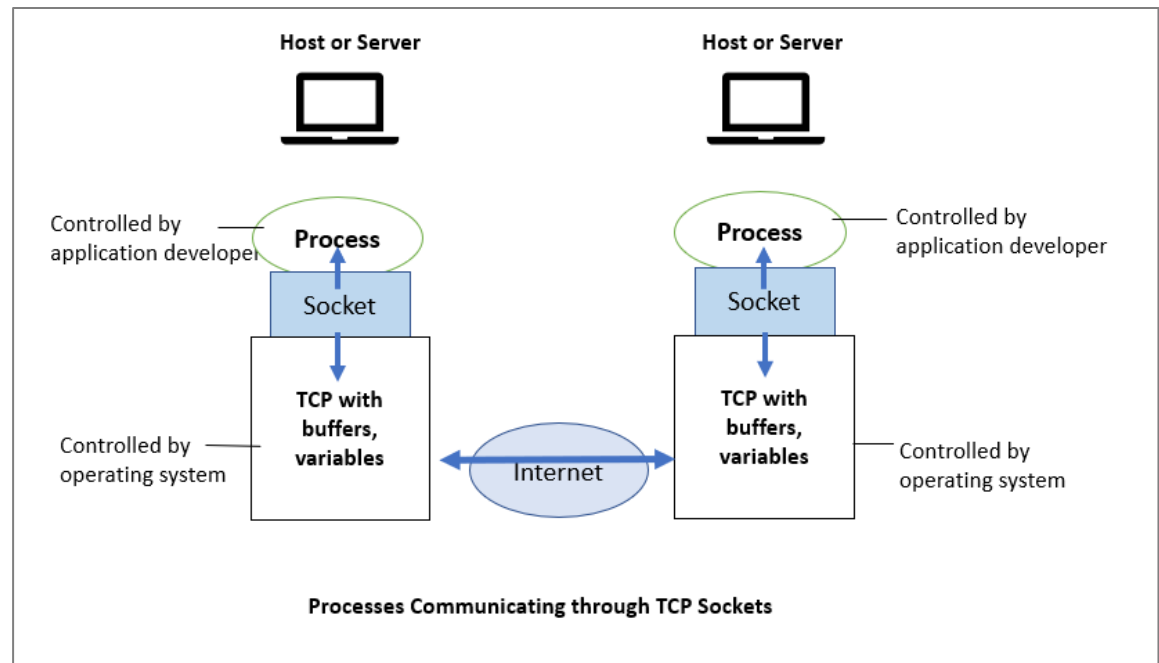
2.3. Ứng dụng TCP Server/Client

2.4. Ứng dụng UDP Sender/Receiver



## 2.1. Khái niệm socket

- Socket là điểm cuối (end-point) trong liên kết truyền thông hai chiều (two-way communication) biểu diễn kết nối giữa client – server.
- Các lớp socket được ràng buộc với một cổng port (thể hiện là một con số cụ thể) để các tầng TCP (TCP Layer) có thể định danh ứng dụng mà dữ liệu sẽ được gửi tới.
- Socket là giao diện lập trình mạng được hỗ trợ bởi nhiều ngôn ngữ, hệ điều hành khác nhau.
- Socket có thể được sử dụng để chờ các kết nối trong ứng dụng server hoặc để thiết lập kết nối trong ứng dụng client.



## 2.2. Cấu trúc địa chỉ socket

- Socket cần được gán địa chỉ để thực hiện chức năng truyền nhận dữ liệu trên mạng.
- Cấu trúc địa chỉ lưu trữ địa chỉ IP và cổng.
- Các cấu trúc địa chỉ:

`struct sockaddr` => Mô tả địa chỉ nói chung

`struct sockaddr_in` => Mô tả địa chỉ IPv4

`struct sockaddr_in6` => Mô tả địa chỉ IPv6

# Cấu trúc địa chỉ IPv4

- Cấu trúc `sockaddr_in` được sử dụng để lưu địa chỉ IPv4 của ứng dụng đích cần nối đến.
- Ứng dụng cần khởi tạo thông tin trong cấu trúc này

```
#include <sys/types.h>
#include <sys/socket.h>

struct in_addr {
    in_addr_t s_addr; /* địa chỉ IPv4 32 bit */
    /* network byte ordered - big-endian */
};

struct sockaddr_in {
    uint8_t sin_len; /* độ dài cấu trúc địa chỉ (16 bytes) */
    sa_family_t sin_family; /* họ địa chỉ IPv4 - AF_INET */
    in_port_t sin_port; /* giá trị cổng */
    /* network byte ordered */
    struct in_addr sin_addr; /* 32 bit địa chỉ */
    /* network byte ordered */
    char sin_zero[8]; /* không sử dụng */
};
```



# Ví dụ khai báo địa chỉ

- Khai báo địa chỉ trong ứng dụng server

```
struct sockaddr_in addr;  
addr.sin_family = AF_INET;  
addr.sin_addr.s_addr = INADDR_ANY;  
addr.sin_port = htons(9090);
```

- Khai báo địa chỉ trong ứng dụng client

```
struct sockaddr_in addr;  
addr.sin_family = AF_INET;  
addr.sin_addr.s_addr = inet_addr("127.0.0.1");  
addr.sin_port = htons(9090);
```

# Cấu trúc địa chỉ IPv6

- Cấu trúc **sockaddr\_in6** được sử dụng để lưu địa chỉ IPv6 của ứng dụng đích cần nối đến.

```
struct sockaddr_in6
{
    SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port; /* Transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* IPv6 scope-id */
};
```

# Các hàm chuyển đổi địa chỉ

- Cần khai báo tệp <arpa/inet.h>
- Chuyển đổi địa chỉ IP dạng chuỗi sang số nguyên 32 bit (IPv4)  
`in_addr_t inet_addr (`  
    `const char *cp // chuỗi ký tự chứa địa chỉ IPv4`  
`) => Hàm trả về địa chỉ dạng số nguyên, -1 nếu gặp lỗi`
- Chuyển đổi địa chỉ IP dạng chuỗi sang cấu trúc `in_addr`  
`int inet_aton (`  
    `const char *cp, // Chuỗi ký tự chứa địa chỉ IP`  
    `struct in_addr *inp // Cấu trúc địa chỉ IP`  
`) => Hàm trả về 1 nếu thành công, 0 nếu gặp lỗi`
- Chuyển đổi địa chỉ từ dạng `in_addr` sang dạng chuỗi (IPv4)  
`char *inet_ntoa (`  
    `struct in_addr in // Cấu trúc địa chỉ IPv4`  
`) => Hàm trả về chuỗi ký tự chứa địa chỉ`

# Các hàm chuyển đổi địa chỉ

- Chuyển đổi từ dạng số sang dạng xâu (cho IPv4 và IPv6)

```
const char *inet_ntop (  
    int af,                // AF_INET hoặc AF_INET6  
    const void *cp,        // con trỏ in_addr hoặc in6_addr  
    char *buf,             // xâu ký tự chứa địa chỉ  
    socklen_t len          // INET_ADDRSTRLEN hoặc INET6_ADDRSTRLEN  
)
```

) => hàm trả về xâu ký tự chứa địa chỉ, trả về NULL nếu gặp lỗi

- Chuyển đổi từ dạng xâu sang dạng số (cho IPv4 và IPv6)

```
int inet_pton (  
    int af,                // AF_INET hoặc AF_INET6  
    const char *cp,        // xâu địa chỉ  
    void *buf              // con trỏ in_addr hoặc in6_addr  
)
```

) => Hàm trả về 1 nếu thành công, 0 nếu xâu ký tự không hợp lệ, -1 nếu gặp lỗi khác

# Các hàm chuyển đổi big-endian ↔ little-endian

- Chuyển đổi little-endian => big-endian (network order)  
// Chuyển đổi 4 byte từ little-endian=>big-endian  
`uint32_t htonl (uint32_t hostlong)`  
// Chuyển đổi 2 byte từ little-endian=>big-endian  
`uint16_t htons (uint16_t hostshort)`
- Chuyển đổi big-endian => little-endian (host order)  
// Chuyển 4 byte từ big-endian=>little-endian  
`uint32_t ntohl (uint32_t netlong)`  
// Chuyển 2 byte từ big-endian=>little-endian  
`uint16_t ntohs (uint16_t netshort)`

# Phân giải tên miền

- Địa chỉ của máy đích được cho dưới dạng tên miền
- Ứng dụng cần thực hiện phân giải tên miền để có địa chỉ IP thích hợp
- Hàm **getaddrinfo()** sử dụng để phân giải tên miền ra các địa chỉ IP
- Cần thêm tệp tiêu đề **netdb.h**

```
int getaddrinfo(  
    const char* nodename, // Tên miền hoặc địa chỉ cần phân giải  
    const char* servname, // Dịch vụ hoặc cổng  
    const struct addrinfo* hints, // Cấu trúc gợi ý  
    struct addrinfo** res // Kết quả  
);
```

- Giá trị trả về
  - Thành công: 0
  - Thất bại: mã lỗi, sử dụng hàm **gai\_strerror()** để in ra thông báo lỗi
- Giải phóng: hàm **freeaddrinfo()**

# Phân giải tên miền

- Cấu trúc **addrinfo**: danh sách liên kết đơn chứa thông tin về tên miền tương ứng

```
struct addrinfo {  
    int ai_flags;           // Thường là AI_CANONNAME  
    int ai_family;         // Thường là AF_INET  
    int ai_socktype;       // Loại socket  
    int ai_protocol;       // Giao thức giao vận  
    socklen_t ai_addrlen;  // Chiều dài của ai_addr  
    char* ai_canonname;     // Tên miền  
    struct sockaddr* ai_addr; // Địa chỉ socket đã phân giải  
    struct addrinfo* ai_next; // Con trỏ tới cấu trúc sau  
};
```

# Phân giải tên miền

- Sử dụng cấu trúc gợi ý để lọc kết quả

```
struct addrinfo hints;  
// IPv4: AF_INET  
// IPv6: AF_INET6  
// Không xác định: AF_UNSPEC  
hints.ai_family = AF_UNSPEC;  
// TCP: SOCK_STREAM  
// UDP: SOCK_DGRAM  
// Không xác định: 0  
hints.ai_socktype = SOCK_STREAM;  
// TCP: IPPROTO_TCP  
// UDP: IPPROTO_UDP  
// Không xác định: 0  
hints.ai_protocol = IPPROTO_TCP;
```





# Phân giải tên miền

- Ví dụ: phân giải địa chỉ cho tên miền nhập vào từ tham số dòng lệnh

```
struct addrinfo *res, *p;  
int ret = getaddrinfo(argv[1], "http", NULL, &res);  
if (ret != 0)  
{  
    printf("Failed to get IP\n");  
    return 1;  
}  
  
// Tiếp trang sau
```

# Phân giải tên miền



```
p = res;
while (p != NULL) {
    if (p->ai_family == AF_INET) {
        printf("IPv4\n");
        struct sockaddr_in addr;
        memcpy(&addr, p->ai_addr, p->ai_addrlen);
        printf("IP: %s\n", inet_ntoa(addr.sin_addr));
    } else if (p->ai_family == AF_INET6) {
        printf("IPv6\n");
        char buf[64];
        struct sockaddr_in6 addr6;
        memcpy(&addr6, p->ai_addr, p->ai_addrlen);
        printf("IP: %s\n", inet_ntop(p->ai_family, &addr6.sin6_addr,
buf, sizeof(addr6)));
    }
    p = p->ai_next;
}
freeaddrinfo(res);
```

# Phân giải tên miền => Kết quả chạy

```
lebvui@Home-Desktop: /mr  X + v
lebvui@Home-Desktop: /mnt/d/Workspaces/socket_tutorials$ ./domain2ip gmail.com
IPv6
IP: 2404:6800:4005:804::2005
IPv4
IP: 172.217.24.69
lebvui@Home-Desktop: /mnt/d/Workspaces/socket_tutorials$ nslookup gmail.com
Server:      203.113.131.2
Address:     203.113.131.2#53

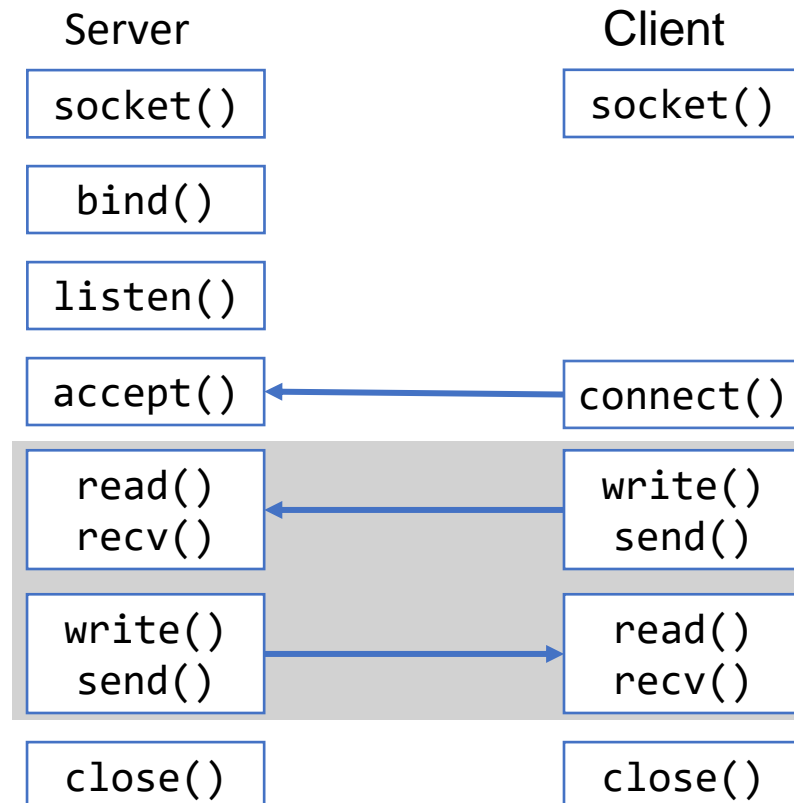
Non-authoritative answer:
Name:   gmail.com
Address: 142.250.199.69
Name:   gmail.com
Address: 2404:6800:4005:804::2005

lebvui@Home-Desktop: /mnt/d/Workspaces/socket_tutorials$ ./domain2ip vnexpress.net
IPv6
IP: 2402:dd40:f000:2::8
IPv4
IP: 111.65.250.2
lebvui@Home-Desktop: /mnt/d/Workspaces/socket_tutorials$ nslookup vnexpress.net
Server:      203.113.131.2
Address:     203.113.131.2#53

Non-authoritative answer:
Name:   vnexpress.net
Address: 111.65.250.2
Name:   vnexpress.net
Address: 2402:dd40:f000:2::8
```

## 2.3. Ứng dụng TCP server/client

- Các hàm được sử dụng để tạo ứng dụng server/client hoạt động theo giao thức TCP.
- Cần khai báo thư viện **<sys/socket.h>**



# Hàm socket()

- Ứng dụng phải tạo SOCKET trước khi có thể gửi nhận dữ liệu.
- Cú pháp

```
#include <sys/socket.h>
```

```
int socket (  
    int domain, // Giao thức AF_INET hoặc AF_INET6  
    int type, // Kiểu socket SOCK_STREAM hoặc SOCK_DGRAM  
    int protocol // Giao thức IPPROTO_TCP hoặc IPPROTO_UDP  
)
```

=> Hàm trả về giá trị mô tả của socket (kiểu int) nếu thành công, trả về -1 nếu gặp lỗi (biến **errno** chứa mã lỗi, cần khai báo **errno.h** để truy nhập **errno**)

# Hàm socket() – Ví dụ

```
// Tạo socket TCP
```

```
int s1 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  
if (s1 == -1) {  
    printf("Không tạo được socket\n");  
    return 1;  
}
```

```
// Tạo socket UDP
```

```
int s2 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);  
if (s2 == -1) {  
    printf("Không tạo được socket\n");  
    return 1;  
}
```

# Cách xác định lỗi

- Đa số các hàm trả về -1 nếu gặp lỗi.
- Biến **errno** trả về mã lỗi xảy ra gần nhất (cần khai báo thư viện `errno.h`)
- Hàm **strerror(int errnum)** trả về chuỗi ký tự mô tả mã lỗi (thư viện `string.h`)
- Hàm **perror(const char \*s)** in ra chuỗi ký tự mô tả mã lỗi gần nhất, **s** là chuỗi ký tự tiền tố, có thể bằng `NULL` (thư viện `stdio.h`).

# Ví dụ

```
int listener = socket(AF_INET, SOCK_STREAM, -1);
if (listener != -1)
    printf("Socket created: %d\n", listener);
else {
    printf("Failed to create socket: %d - %s\n", errno, strerror(errno));
    perror("socket() failed");
    exit(1);
}
```



A terminal window titled 'lebavui@Vui-Laptop: /mnt/c/l' with standard window controls. The terminal shows the user running './simple\_server' in the directory '/mnt/c/Users/lebavui/source/NetworkProgrammin'. The output indicates a failure to create a socket with error code 93, 'Protocol not supported', and a corresponding perror message. The prompt returns to the user.

```
lebavui@Vui-Laptop: /mnt/c/Users/lebavui/source/NetworkProgrammin
g/socket_tutorials$ ./simple_server
Failed to create socket: 93 - Protocol not supported
socket() failed: Protocol not supported
lebavui@Vui-Laptop: /mnt/c/Users/lebavui/source/NetworkProgrammin
g/socket_tutorials$
```



# Hàm bind()

- Gắn socket với cấu trúc địa chỉ trong ứng dụng server.
- Cú pháp

```
#include <sys/socket.h>
```

```
int bind (  
    int sockfd,                // mô tả của socket  
    const struct sockaddr *addr, // con trỏ cấu trúc địa chỉ  
    socklen_t addrlen          // độ dài cấu trúc địa chỉ  
)
```

=> Hàm trả về 0 nếu thành công, trả về -1 nếu gặp lỗi

- Khai báo địa chỉ của server bằng cấu trúc địa chỉ **sockaddr\_in** (IPv4) hoặc **sockaddr\_in6** (IPv6) => Cần ép kiểu sang **sockaddr**.

# Hàm bind() – Ví dụ

```
// Khai báo cấu trúc địa chỉ của server
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(9000);

// Gắn địa chỉ với socket
bind(listener, (struct sockaddr *)&addr, sizeof(addr));
```

# Hàm listen()

- Chuyển socket sang trạng thái chờ kết nối.
- Cú pháp

```
#include <sys/socket.h>
```

```
int listen (  
    int fd, // mô tả của socket  
    int n   // chiều dài hàng đợi chờ kết nối  
)
```

=> Hàm trả về 0 nếu thành công, trả về -1 nếu gặp lỗi

- Ví dụ:

```
// listener là socket đã được khởi tạo  
listen(listener, 5);
```

# Hàm accept()

- Chấp nhận kết nối đang nằm trong hàng đợi.
- Cú pháp

```
#include <sys/socket.h>
```

```
int accept (
```

```
    int sockfd, // socket chờ kết nối đã được khởi tạo
```

```
    struct sockaddr *addr, // con trỏ địa chỉ client
```

```
    socklen_t *addrlen // con trỏ độ dài địa chỉ client
```

) => Hàm trả về mô tả của socket nếu thành công (giá trị kiểu int), trả về -1 nếu gặp lỗi

- Nếu con trỏ **addr** và **addrlen** là NULL, hàm accept() sẽ không lấy địa chỉ của client.
- Socket trả về được sử dụng để truyền nhận dữ liệu giữa server và client.
- Hàm accept() cần được gọi nhiều lần để chấp nhận nhiều kết nối.

# Hàm accept() – Ví dụ

```
// s là socket đã được khởi tạo để chờ các kết nối

int s1 = accept(s, NULL, NULL);
// s1 là socket đại diện cho kết nối giữa server và client1
// trong trường hợp này không cần quan tâm đến địa chỉ của client1

struct sockaddr_in clientAddr;
int clientAddrLen = sizeof(clientAddr);
int s2 = accept(s, (struct sockaddr *)&clientAddr, &clientAddrLen);
// s2 là socket đại diện cho kết nối giữa server và client2
// clientAddr chứa dữ liệu địa chỉ của client2 (địa chỉ IP và cổng)
```

# Hàm send()

- Truyền dữ liệu trên socket.
- Cú pháp

```
#include <sys/socket.h>
```

```
ssize_t send (  
    int sockfd,          // socket ở trạng thái đã kết nối  
    const void *buf,     // buffer chứa dữ liệu cần gửi  
    size_t len,          // số byte cần gửi  
    int flags             // cờ quy định cách truyền, mặc định là 0  
)
```

=> Hàm trả về số byte đã gửi nếu thành công, trả về -1 nếu gặp lỗi

# Hàm send() – Ví dụ

```
// client là socket đã được chấp nhận bởi server

// gửi đi 1 chuỗi ký tự
char* str = "Hello Network Programming";
int ret = send(client, str, strlen(str), 0);
if (ret != -1)
    printf(" %d bytes are sent", ret);

// gửi đi 1 mảng dữ liệu
char buf[256];
for (int i = 0; i < 10; i++)
    buf[i] = i;
ret = send(client, buf, 10, 0);

// gửi đi biến dữ liệu bất kỳ
double d = 1.234;
ret = send(client, &d, sizeof(d), 0);
```

# Hàm write()

- Truyền dữ liệu trên socket.
- Cú pháp

```
#include <unistd.h>
```

```
ssize_t write (  
    int fd, // socket ở trạng thái đã kết nối  
    const void *buf, // buffer chứa dữ liệu cần gửi  
    size_t n // số byte cần gửi  
)
```

=> Hàm trả về số byte đã gửi nếu thành công, trả về -1 nếu gặp lỗi



# Hàm write() – Ví dụ

```
// client là socket đã được chấp nhận bởi server

// gửi đi 1 chuỗi ký tự
char* str = "Hello Network Programming";
int ret = write(client, str, strlen(str));
if (ret != -1)
    printf(" %d bytes are sent", res);

// gửi đi 1 mảng dữ liệu
char buf[256];
for (int i = 0; i < 10; i++)
    buf[i] = i;
ret = write(client, buf, 10);

// gửi đi biến dữ liệu bất kỳ
double d = 1.234;
ret = write(client, &d, sizeof(d));
```

# Hàm recv()

- Nhận dữ liệu từ socket.
- Cú pháp

```
#include <sys/socket.h>
```

```
ssize_t recv (  
    int sockfd, // socket ở trạng thái đã kết nối  
    void *buf,  // buffer chứa dữ liệu sẽ nhận được  
    size_t n,   // số byte muốn nhận (độ dài của buffer)  
    int flags   // cờ quy định cách nhận, mặc định là 0  
)
```

=> Hàm trả về số byte đã nhận nếu thành công, trả về 0 nếu kết nối bị đóng, trả về -1 nếu gặp lỗi

# Hàm recv() – Ví dụ

```
// client là socket đã được chấp nhận bởi server
char buf[256];
// nhận 1 buffer dữ liệu
int ret = recv(client, buf, sizeof(buf), 0);

// nhận biến dữ liệu bất kỳ
double d;
ret = recv(client, &d, sizeof(d), 0);

// nhận dữ liệu đến khi ngắt kết nối
while (true) {
    ret = recv(client, buf, sizeof(buf), 0);
    // kiểm tra điều kiện kết nối
    if (ret <= 0)
        break;
    // xử lý dữ liệu nhận được
}
```

# Hàm read()

- Nhận dữ liệu từ socket.
- Cú pháp

```
#include <unistd.h>
```

```
ssize_t read (  
    int fd, // socket ở trạng thái đã kết nối  
    void *buf, // buffer chứa dữ liệu sẽ nhận được  
    size_t nbytes // số byte muốn nhận (độ dài của buffer)  
)
```

=> Hàm trả về số byte đã nhận nếu thành công, trả về 0 nếu kết nối bị đóng, trả về -1 nếu gặp lỗi

# Hàm read() – Ví dụ

```
// client là socket đã được chấp nhận bởi server
char buf[256];
// nhận 1 buffer dữ liệu
int ret = read(client, buf, sizeof(buf));

// nhận biến dữ liệu bất kỳ
double d;
ret = read(client, &d, sizeof(d));

// nhận dữ liệu đến khi ngắt kết nối
while (true) {
    ret = read(client, buf, sizeof(buf));
    // kiểm tra điều kiện kết nối
    if (ret <= 0)
        break;
    // xử lý dữ liệu nhận được
}
```

# Hàm close()

- Đóng kết nối.
- Cú pháp

```
#include <unistd.h>
```

```
int close (  
    int sockfd // socket cần đóng kết nối  
)
```

=> Hàm trả về 0 nếu thành công, trả về -1 nếu gặp lỗi

# Hàm shutdown()

- Đóng kết nối.
- Cú pháp

```
#include <sys/socket.h>
```

```
int shutdown (  
    int sockfd, // socket cần đóng kết nối  
    int how, // cách thức đóng kết nối  
)
```

=> Hàm trả về 0 nếu thành công, trả về -1 nếu gặp lỗi

- Các giá trị của tham số **how**
  - SHUT\_RD: không nhận thêm dữ liệu
  - SHUT\_WR: không truyền thêm dữ liệu
  - SHUT\_RDWR: không truyền và nhận thêm dữ liệu
- Lệnh `shutdown(fd, SHUT_RDWR)` tương đương lệnh `close(fd)`

# Truyền dữ liệu sử dụng TCP - Ứng dụng server

- Tạo socket qua hàm **socket()**
- Gắn socket vào một giao diện mạng thông qua hàm **bind()**
- Chuyển socket sang trạng thái đợi kết nối qua hàm **listen()**
- Chấp nhận kết nối từ client thông qua hàm **accept()**
- Gửi dữ liệu tới client thông qua hàm **send()/write()**
- Nhận dữ liệu từ client thông qua hàm **recv()/read()**
- Đóng socket khi việc truyền nhận kết thúc bằng hàm **close()**



# Ví dụ ứng dụng server (1/3)



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>

int main()
{
    // Tao socket
    int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listener != -1)
        printf("Socket created: %d\n", listener);
    else {
        printf("Failed to create socket.\n");
        exit(1);
    }

    // Khai bao cau truc dia chi server
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(9000);
```

# Ví dụ ứng dụng server (2/3)

```
// Gán địa chỉ với socket
if (bind(listener, (struct sockaddr *)&addr, sizeof(addr)))
{
    printf("bind() failed.\n");
    exit(1);
}

if (listen(listener, 5))
{
    printf("listen() failed.\n");
    exit(1);
}

printf("Waiting for a new client ...\n");

// Chấp nhận kết nối
int client = accept(listener, NULL, NULL);
if (client == -1)
{
    printf("accept() failed.\n");
    exit(1);
}
printf("New client connected: %d\n", client);
```

# Ví dụ ứng dụng server (3/3)

```
// Nhan du lieu tu client
char buf[256];
int ret = recv(client, buf, sizeof(buf), 0);
if (ret <= 0)
{
    printf("recv() failed.\n");
    exit(1);
}

// Them ky tu ket thuc xau va in ra man hinh
if (ret < sizeof(buf))
buf[ret] = 0;
puts(buf);

// Gui du lieu sang client
send(client, buf, strlen(buf), 0);

// Dong ket noi
close(client);
close(listener);

return 0;
}
```

- Tạo socket qua hàm **socket()**
- Điền thông tin về server vào cấu trúc **sockaddr\_in**
- Kết nối tới server qua hàm **connect()**
- Gửi dữ liệu tới server thông qua hàm **send()**
- Nhận dữ liệu từ server thông qua hàm **recv()**
- Đóng socket khi việc truyền nhận kết thúc bằng hàm **close()**

# Hàm connect()

- Thiết lập kết nối đến server thông qua socket.
- Cú pháp

```
int connect (  
    int sockfd, // socket đã được tạo  
    const struct sockaddr *addr, // con trỏ địa chỉ server  
    socklen_t addrlen // độ dài cấu trúc địa chỉ  
)
```

=> Hàm trả về 0 nếu thành công, trả về -1 nếu gặp lỗi

- Địa chỉ server có thể xác định thông qua:
  - Khai báo cấu trúc địa chỉ IP và cổng
  - Phân giải tên miền

# Hàm connect() – Ví dụ 1

```
// Khai bao socket
int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

// Khai bao dia chi cua server
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_port = htons(9000);

// Ket noi den server
int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
if (res == -1) {
    printf("Khong ket noi duoc den server!");
    return 1;
}
```

# Hàm connect() – Ví dụ 2

```
// Khai bao socket
int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
// Phân giải tên miền thành IP
struct addrinfo *res;
int ret = getaddrinfo("httpbin.org", "http", NULL, &res);
if (ret == -1 || res == NULL) {
    printf("Failed to get IP address\n");
    return 1;
}
// Ket noi den server
ret = connect(client, res->ai_addr, res->ai_addrlen);
if (ret == -1) {
    printf("Khong ket noi duoc den server!");
    return 1;
}
```

# Ứng dụng client – Ví dụ



```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <arpa/inet.h>

int main() {
    // Khai bao socket
    int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Khai bao dia chi cua server
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    addr.sin_port = htons(9000);

    // Ket noi den server
    int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
```



# Ứng dụng client – Ví dụ

```
if (res == -1) {
    printf("Khong ket noi duoc den server!");
    return 1;
}

// Gui tin nhan den server
char *msg = "Hello server";
send(client, msg, strlen(msg), 0);

// Nhan tin nhan tu server
char buf[2048];
int len = recv(client, buf, sizeof(buf), 0);
buf[len] = 0;
printf("Data received: %s\n", buf);

// Ket thuc, dong socket
close(client);

return 0;
}
```

# Các ví dụ minh họa

- Tạo client gửi thông điệp đến netcat server.
- Tạo server nhận thông điệp từ netcat client.
- Tạo server nhận và hiển thị yêu cầu từ trình duyệt.
- Tạo client gửi lệnh GET đến <http://httpbin.org/get> và hiển thị kết quả trả về.
- Tạo client và server truyền nhận dữ liệu là chuỗi ký tự.
- Tạo client và server truyền nhận dữ liệu là số.
- Tạo client và server truyền nhận dữ liệu là file.

## 2.4. Ứng dụng UDP Sender / Receiver

- Giao thức UDP là giao thức không kết nối (connectionless)
- Ứng dụng không cần phải thiết lập kết nối trước khi gửi tin.
- Ứng dụng có thể nhận được tin từ bất kỳ máy tính nào trong mạng.
- Trình tự gửi thông tin ở bên gửi như sau:



# Ứng dụng UDP Sender – Hàm sendto()

- Gửi dữ liệu qua socket theo giao thức UDP
- Cú pháp

```
ssize_t sendto (  
    int sockfd, // socket đã được khởi tạo  
    const void *buf, // buffer chứa dữ liệu cần gửi  
    size_t len, // số byte cần gửi  
    int flags, // cờ quy định cách gửi, mặc định là 0  
    const struct sockaddr *addr, // con trỏ địa chỉ bên nhận  
    socklen_t addr_len // độ dài cấu trúc địa chỉ  
)
```

=> Hàm trả về số byte đã gửi nếu thành công, trả về -1 nếu gặp lỗi

# Ứng dụng UDP Sender – Ví dụ



```
// Tạo socket theo giao thức UDP
int sender = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

char *msg = "Hello. I am sending a message.\n";

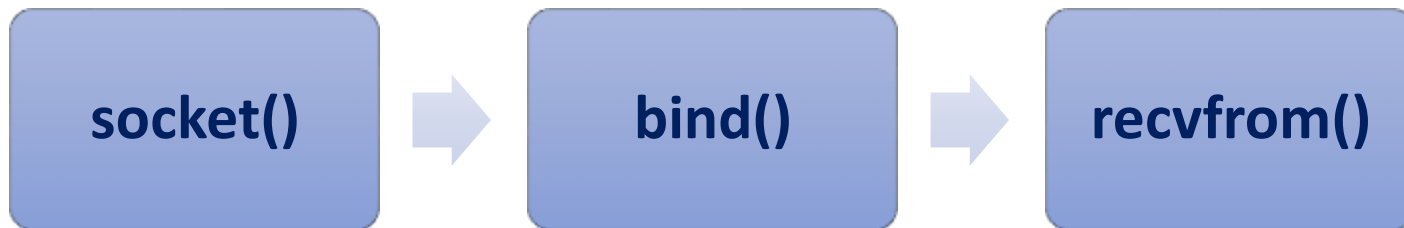
// Khai báo địa chỉ bên nhận
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_port = htons(9090);

// Gửi tin nhắn
char buf[256];
while (1)
{
    printf("Enter message: ");
    fgets(buf, sizeof(buf), stdin);
    sendto(sender, buf, strlen(buf), 0, (struct sockaddr *)&addr,
sizeof(addr));
}
```



# Ứng dụng UDP Receiver

- Trình tự nhận thông tin ở bên nhận như sau:



# Ứng dụng UDP Receiver – Hàm recvfrom()

- Nhận dữ liệu qua socket theo giao thức UDP
- Cú pháp

```
ssize_t recvfrom (  
    int sockfd, // socket đã khởi tạo  
    void *buf,  // buf chứa dữ liệu nhận được  
    size_t len, // số byte muốn nhận (kích thước buffer)  
    int flags,  // cờ quy định cách nhận, mặc định là 0  
    struct sockaddr *src_addr, // con trỏ địa chỉ bên gửi  
    socklen_t *addr_len        // con trỏ độ dài địa chỉ  
)
```

=> Hàm trả về số byte đã nhận nếu thành công, trả về -1 nếu gặp lỗi

# Ứng dụng UDP Receiver – Ví dụ



```
// Tạo socket theo giao thức UDP
int receiver = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

// Khai báo địa chỉ bên nhận
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(9090);

bind(receiver, (struct sockaddr *)&addr, sizeof(addr));

// Nhận tin nhắn
char buf[16];
while (1) {
    int ret = recvfrom(receiver, buf, sizeof(buf), 0, NULL, NULL);
    if (ret == -1) {
        printf("recvfrom() failed\n");
        break;
    } else {
        buf[ret] = 0;
        printf("%d - %s\n", ret, buf);
    }
}
```



# Chú ý

- Lệnh nhận dữ liệu:

```
recv(client, buf, sizeof(buf), 0);
```

```
recvfrom(client, buf, sizeof(buf), 0, NULL, NULL);
```

=> Hai lệnh này tương đương nhau, có thể sử dụng thay thế

- Lệnh truyền dữ liệu:

```
send(client, buf, strlen(buf), 0);
```

```
sendto(client, buf, strlen(buf), 0, NULL, 0);
```

=> Hai lệnh này tương đương nhau, có thể sử dụng thay thế

# Kịch bản kiểm thử giao thức hướng dòng - TCP

- Bên nhận tạm dừng chương trình trước khi nhận dữ liệu.
- Bên truyền thực hiện truyền 2 lần liên tiếp.
- Bên nhận tiếp tục chương trình và sẽ nhận được cả 2 gói tin trong một lần nhận.

# Kịch bản kiểm thử giao thức hướng thông điệp - UDP

- Bên nhận tạm dừng chương trình trước khi nhận dữ liệu.
- Bên truyền thực hiện truyền 2 lần liên tiếp.
- Bên nhận tiếp tục chương trình và sẽ nhận được 2 gói tin trong hai lần nhận.

1. Viết chương trình **tcp\_client**, kết nối đến một máy chủ xác định bởi địa chỉ IP và cổng. Sau đó nhận dữ liệu từ bàn phím và gửi đến server. Tham số được truyền vào từ dòng lệnh có dạng

**tcp\_client <địa chỉ IP> <cổng>**

2. Viết chương trình **tcp\_server**, đợi kết nối ở cổng xác định bởi tham số dòng lệnh. Mỗi khi có client kết nối đến, thì gửi xâu chào được chỉ ra trong một tệp tin xác định, sau đó ghi toàn bộ nội dung client gửi đến vào một tệp tin khác được chỉ ra trong tham số dòng lệnh

**tcp\_server <cổng> <tệp tin chứa câu chào>  
<tệp tin lưu nội dung client gửi đến>**

3. Viết chương trình **sv\_client**, cho phép người dùng nhập dữ liệu là thông tin của sinh viên bao gồm MSSV, họ tên, ngày sinh, và điểm trung bình các môn học. Các thông tin trên được đóng gói và gửi sang **sv\_server**. Địa chỉ và cổng của server được nhập từ tham số dòng lệnh.

4. Viết chương trình **sv\_server**, nhận dữ liệu từ **sv\_client**, in ra màn hình và đồng thời ghi vào file **sv\_log.txt**. Dữ liệu được ghi trên một dòng với mỗi client, kèm theo địa chỉ IP và thời gian client đã gửi. Tham số cổng và tên file log được nhập từ tham số dòng lệnh.

Ví dụ dữ liệu trong file log:

```
127.0.0.1 2023-04-10 09:00:00 20201234 Nguyen Van A 2002-04-10 3.99
```

```
127.0.0.1 2023-04-10 09:00:10 20205678 Tran Van B 2002-08-18 3.50
```

# Chương 3. Các kiến trúc client-server

# Chương 3. Các kiến trúc client-server

## 3.1. Các chế độ hoạt động của socket

- Blocking
- Non-blocking

## 3.2. Iterative server

## 3.3. Multiplexing

- Hàm select()
- Hàm poll()

## 3.4. Multiprocess / Forking

- Hàm fork()
- Preforking

## 3.5. Multithreading

- Thư viện pthread, hàm pthread\_create()
- Prethreading



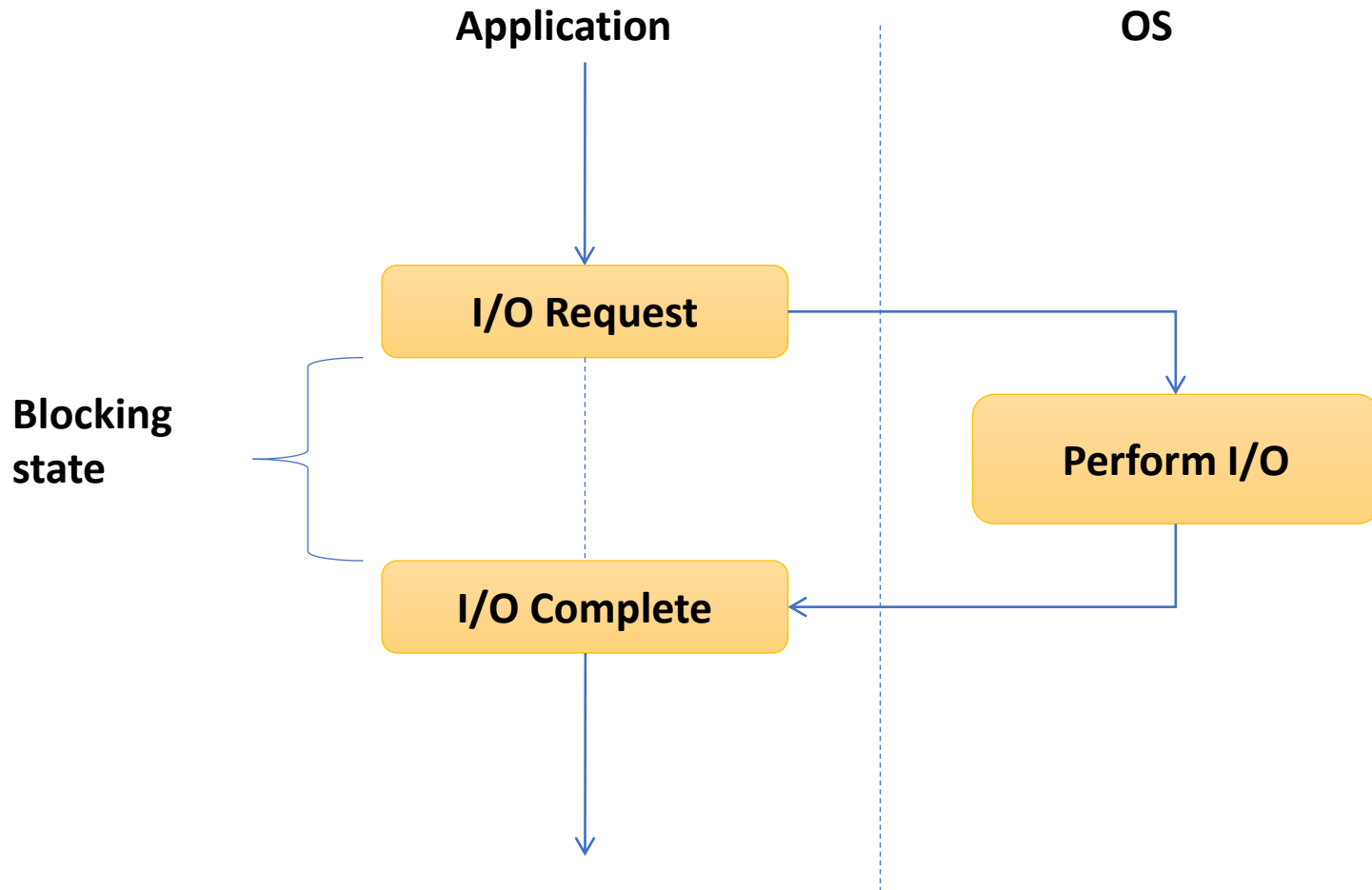
## 3.1. Các chế độ hoạt động của socket

### Chế độ đồng bộ (blocking)

- Là chế độ mà các hàm vào ra sẽ chặn thread đến khi thao tác vào ra hoàn tất (các hàm vào ra sẽ không trở về cho đến khi thao tác hoàn tất).
- Là chế độ mặc định của socket
- Các hàm ảnh hưởng:
  - accept()
  - connect()
  - send()
  - recv()
  - ...



# Chế độ đồng bộ (blocking)



# Chế độ đồng bộ (blocking)

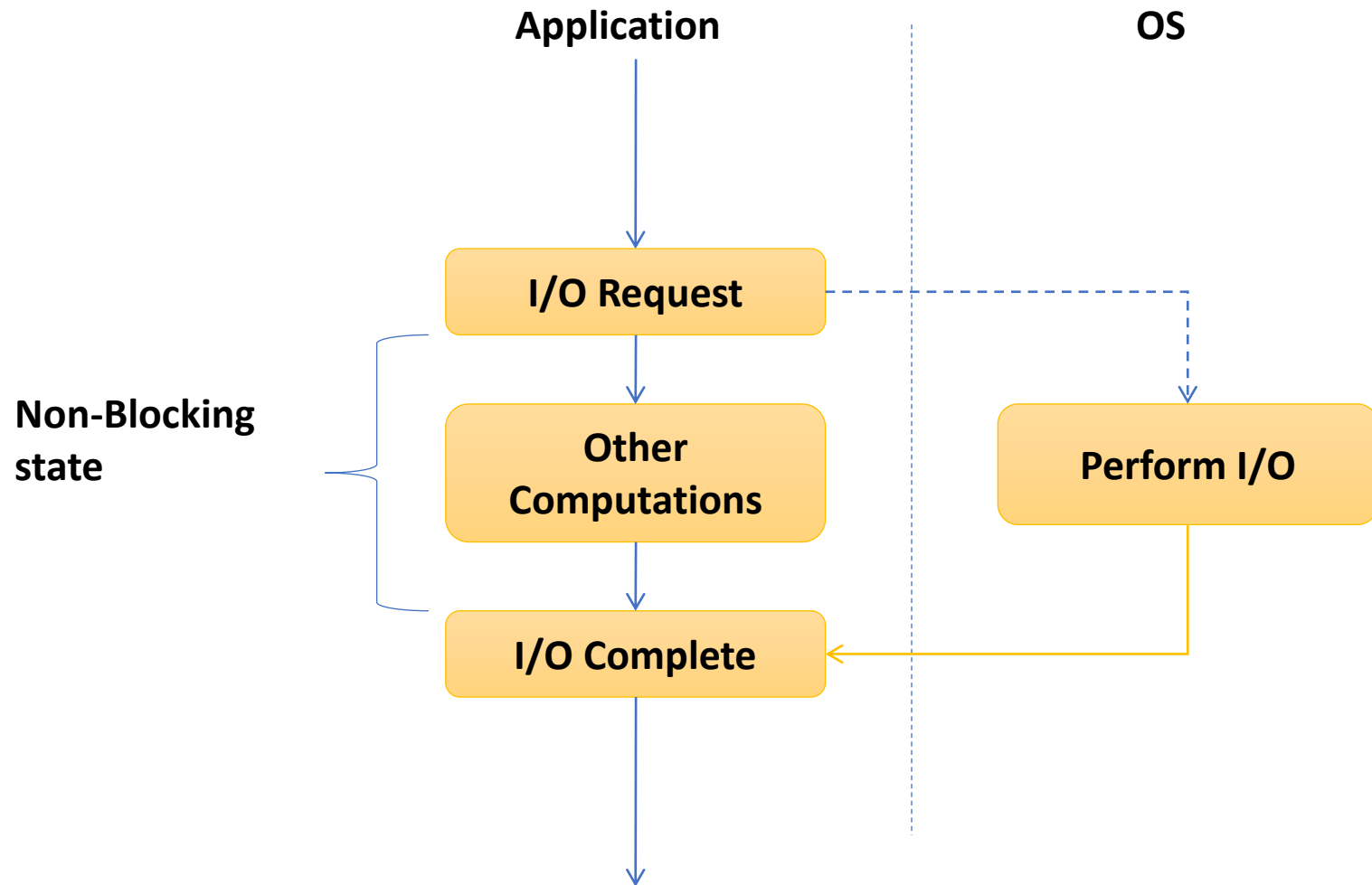
- Thích hợp với các ứng dụng xử lý tuần tự. Không nên gọi các hàm blocking khi ở thread xử lý giao diện (GUI thread).
- Ví dụ: thread bị chặn bởi hàm `recv()` thì không thể thực hiện các công việc khác.

```
...  
do {  
    // Thread sẽ bị chặn lại khi gọi hàm recv  
    // Trong lúc đợi dữ liệu thì không thể gửi dữ liệu  
    rc = recv (receiver, buf, sizeof(buf), 0);  
    // ...  
} while ();  
...
```

# Chế độ bất đồng bộ (non-blocking)

- Là chế độ mà các thao tác vào ra sẽ trở về nơi gọi ngay lập tức và tiếp tục thực thi thread. Kết quả của thao tác vào ra sẽ được thông báo cho chương trình dưới một cơ chế đồng bộ nào đó.
- Các hàm vào ra bất đồng bộ sẽ trả về mã lỗi **EWOULDBLOCK (EAGAIN - 11)** nếu thao tác đó không thể hoàn tất ngay và mất thời gian đáng kể (chấp nhận kết nối, nhận dữ liệu, gửi dữ liệu...). Đây là điều hoàn toàn bình thường.
- Có thể sử dụng trong thread xử lý giao diện của ứng dụng.
- Thích hợp với các ứng dụng hướng sự kiện.

# Chế độ bất đồng bộ (non-blocking)



# Chuyển socket sang chế độ bất đồng bộ

- Áp dụng cho socket chờ kết nối và socket truyền nhận dữ liệu.
- Hàm **ioctl()** được sử dụng để chuyển socket sang trạng thái bất đồng bộ.

```
#include <sys/ioctl.h>
```

```
unsigned long ul = 1;
```

```
ioctl(socket_fd, FIONBIO, &ul);
```

# Chuyển socket sang chế độ bất đồng bộ

- Hàm **fcntl()** cũng được sử dụng để thay đổi chế độ hoạt động của socket.

```
#include <fcntl.h>
```

```
fcntl(socket_fd, F_SETFL, O_NONBLOCK);
```

# VD: Socket chờ kết nối – Lệnh accept()



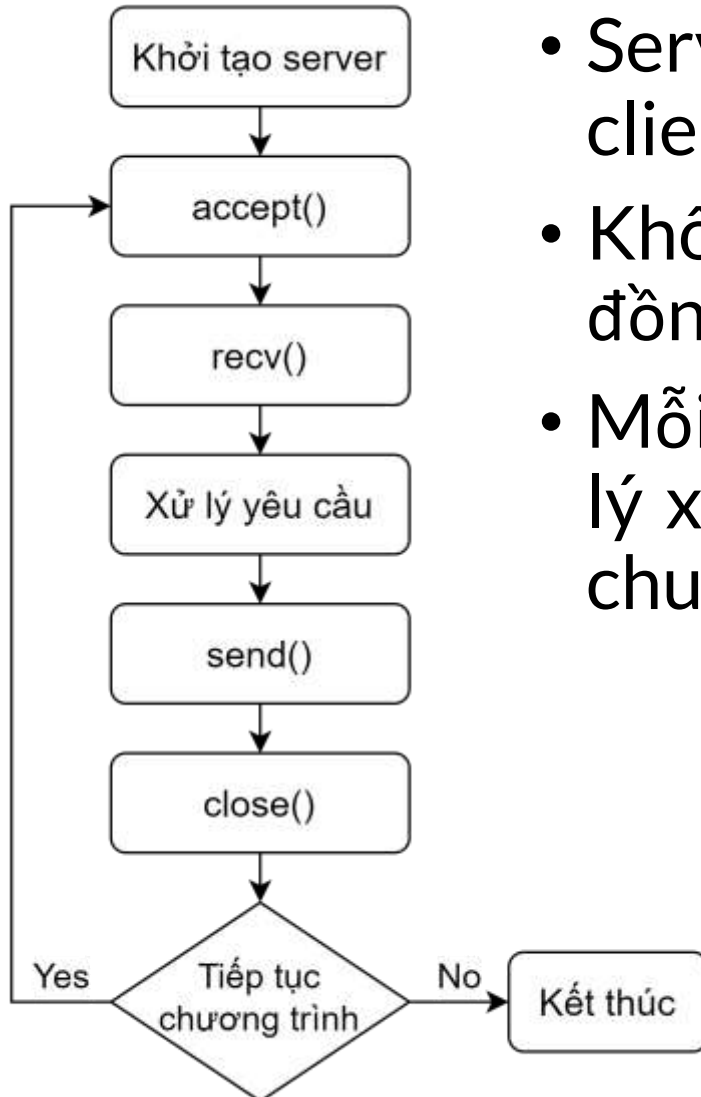
```
// Chấp nhận kết nối
int client = accept(listener, NULL, NULL);
if (client == -1) {
    if (errno != EWOULDBLOCK) {
        printf("accept() failed.\n");
        exit(1);
    } else {
        // Lỗi do thao tác vào ra chưa hoàn tất. Không cần xử lý gì thêm.
    }
} else {
    printf("New client connected: %d\n", client);
    clients[numClients++] = client;
    ul = 1;
    ioctl(client, FIONBIO, &ul);
}
```

# VD: Socket truyền nhận dữ liệu – Lệnh recv()

```
int ret = recv(clients[i], buf, sizeof(buf), 0);
if (ret == -1) {
    if (errno != EWOULDBLOCK) {
        printf("recv() failed.\n");
        continue;
    } else {
        // Lỗi do thao tác vào ra chưa hoàn tất. Không cần xử lý gì thêm.
    }
} else if (ret == 0) {
    printf("client disconnected.\n");
    close(clients[i]);
    continue;
} else {
    // Xử lý dữ liệu nhận được
}
```



## 3.2. Iterative server



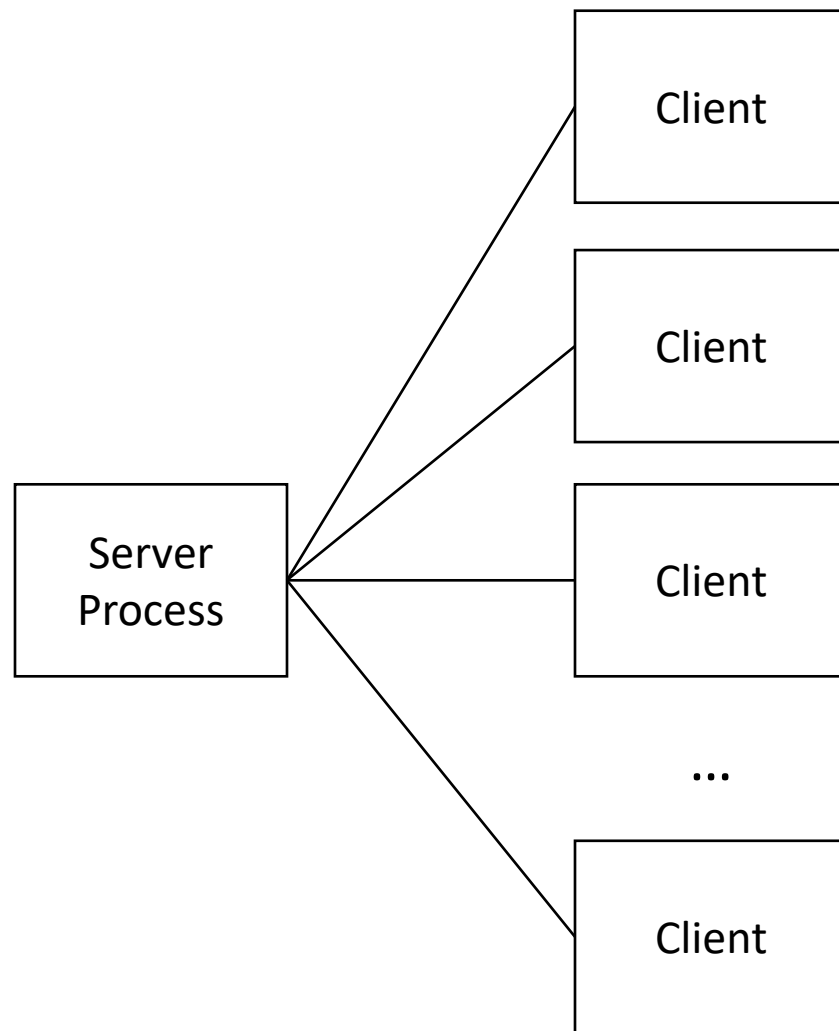
- Server xử lý tuần tự yêu cầu từ các client.
- Không phù hợp trong việc xử lý đồng thời nhiều kết nối.
- Mỗi client gửi 1 yêu cầu, server xử lý xong trả lại kết quả cho client và chuyển sang client khác.

# VD: HTTP server

```
// Khởi tạo server
while (1) {
    // Chờ kết nối mới
    int client = accept(listener, NULL, NULL);
    printf("New client connected: %d\n", client);
    // Nhận dữ liệu từ client và in ra màn hình
    char buf[256];
    int ret = recv(client, buf, sizeof(buf), 0);
    buf[ret] = 0;
    puts(buf);
    // Trả lại kết quả cho client
    char *msg = "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n<html><body><h1>Xin  
chao cac ban</h1></body></html>";
    send(client, msg, strlen(msg), 0);
    // Đóng kết nối
    close(client);
}
```

## 3.3. Multiplexing

- Là cơ chế quản lý nhiều kết nối client chỉ trong một tiến trình của server.
- Ứng dụng sử dụng mảng để quản lý các socket, hệ thống sẽ cho biết socket nào cần được phục vụ hoặc có yêu cầu kết nối mới.
- Chỉ có duy nhất 1 kết nối được phục vụ tại 1 thời điểm.
- Sử dụng hàm **select()** để thăm dò các kết nối.



# Hàm select()

- Hoạt động ở chế độ đồng bộ.
- Được sử dụng để chờ các sự kiện và trả về kết quả khi một hoặc nhiều sự kiện xảy ra hoặc đã qua một khoảng thời gian (timeout).
- Được sử dụng để thăm dò các sự kiện của socket (gửi dữ liệu, nhận dữ liệu, kết nối thành công, yêu cầu kết nối, ...)
- Hỗ trợ nhiều kết nối.
- Có thể xử lý tập trung tất cả các socket trong cùng một luồng.
- Xử lý các sự kiện tuần tự, tại một thời điểm chỉ xử lý một sự kiện của 1 socket.

# Cú pháp hàm select()

```
#include <sys/select.h>
```

```
int select (int nfds, fd_set *readfds,  
            fd_set *writefds, fd_set *exceptfds,  
            struct timeval *timeout)
```

- **nfds**: giá trị socket lớn nhất cộng 1 được gán vào 3 tập hợp (không vượt quá FD\_SETSIZE)
- **readfds**: tập các socket chờ sự kiện đọc
- **writefds**: tập các socket chờ sự kiện ghi
- **exceptfds**: tập các socket chờ sự kiện ngoại lệ hoặc lỗi
- **timeout**: thời gian chờ các sự kiện
  - NULL – chờ với thời gian vô hạn
  - 0 – không chờ sự kiện nào
  - > 0 – chờ với thời gian xác định

# Kết quả trả về của hàm `select()`

- Giá trị trả về:
  - Thành công: số lượng socket xảy ra sự kiện
  - Hết thời gian chờ: 0
  - Bị lỗi: -1
- Điều kiện thành công của hàm **`select()`**
  - Một trong các socket của tập **`readfds`** nhận dữ liệu hoặc kết nối bị đóng, bị hủy hoặc có yêu cầu kết nối.
  - Một trong các socket của tập **`writfds`** có thể gửi dữ liệu, hoặc hàm connect thành công trên socket non-blocking.
  - Một trong các socket của tập **`exceptfds`** nhận dữ liệu OOB, hoặc connect thất bại.
- Các tập **`readfds`**, **`writfds`**, **`exceptfds`** có thể NULL, nhưng không thể cả 3 cùng NULL.

# Cấu trúc `fd_set`

- `fd_set` là kiểu dữ liệu cấu trúc chứa mảng các bit mô tả các socket được gắn vào để thăm dò sự kiện.
- Các macro được sử dụng để thực hiện các thao tác với tập `fd_set`:

`void FD_CLR(int fd, fd_set *set);` => Xóa `fd` ra khỏi tập `set`

`int FD_ISSET(int fd, fd_set *set);` => Kiểm tra sự kiện của `fd` xảy ra với tập `set`

`void FD_SET(int fd, fd_set *set);` => Gắn `fd` vào tập `set`

`void FD_ZERO(fd_set *set);` => Xóa tất cả các socket khỏi tập `set`

- Hàm `select()` thay đổi giá trị của tập `fd_set` sau khi trả về kết quả để chỉ ra socket nào có sự kiện => Cần khởi tạo lại tập `fd_set` nếu gọi `select()` nhiều lần.

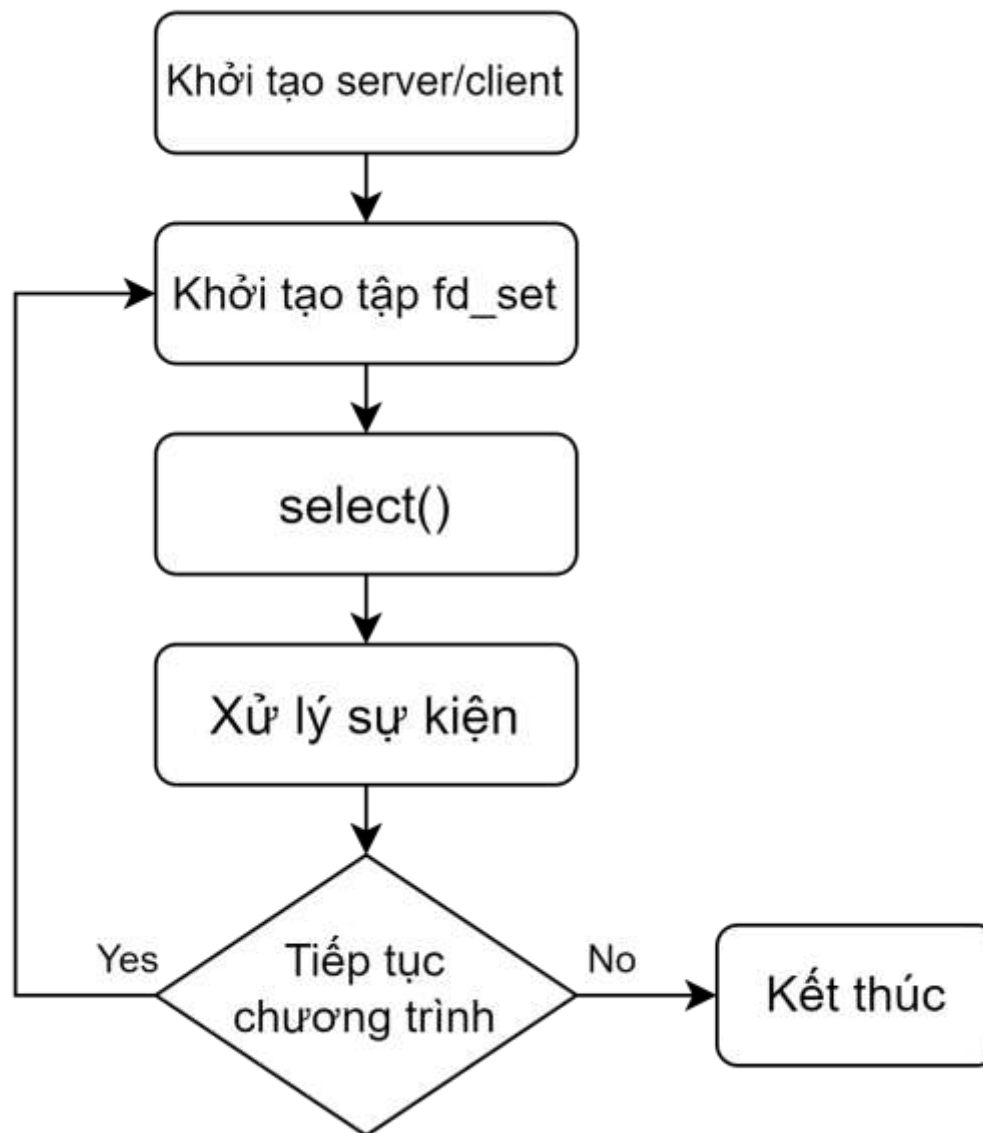
# Cấu trúc timeval

```
struct timeval
{
    time_t tv_sec;      /* Seconds.  */
    suseconds_t tv_usec; /* Microseconds.  */
};
```

- Xác định thời gian chờ của hàm **select()**
  - Bị thay đổi giá trị khi hàm select() trả về kết quả
- => Cần khởi tạo mỗi khi gọi hàm select().



# Cấu trúc chương trình sử dụng hàm select()



# Ví dụ 1 – Select client (1/2)

- Chương trình client vừa nhận dữ liệu từ socket vừa nhập dữ liệu từ bàn phím.

```
// Client đã được khởi tạo ...
```

```
fd_set fdread;
```

```
FD_ZERO(&fdread);
```

```
char buf[256];
```

```
while (1)
```

```
{
```

```
    FD_SET(STDIN_FILENO, &fdread);
```

```
    FD_SET(client, &fdread);
```

```
    int maxdp = client + 1; // STDIN_FILENO is 0
```

```
    select(client + 1, &fdread, NULL, NULL, NULL);
```



## Ví dụ 1 – Select client (2/2)

```
if (FD_ISSET(STDIN_FILENO, &fdread)) // Kiểm tra bàn phím
{
    fgets(buf, sizeof(buf), stdin);
    send(client, buf, strlen(buf), 0);
}

if (FD_ISSET(client, &fdread)) // Kiểm tra socket client
{
    ret = recv(client, buf, sizeof(buf), 0);
    buf[ret] = 0;
    printf("Received: %s\n", buf);
}

}
```

// Đóng client ...

# Ví dụ 2 – Select server (version 1) (1/2)



```
fd_set fdread;
int clients[64];
int numClients = 0;
struct timeval tv;
char buf[2048];
while (1) {
    // Khởi tạo và gán các socket vào tập fdread
    FD_ZERO(&fdread);
    FD_SET(listener, &fdread);
    int maxdp = listener + 1;
    for (int i = 0; i < numClients; i++) {
        FD_SET(clients[i], &fdread);
        if (clients[i] + 1 > maxdp) maxdp = clients[i] + 1;
    }
    // Thiết lập thời gian chờ
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    // Chờ đến khi sự kiện xảy ra
    int ret = select(maxdp, &fdread, NULL, NULL, &tv);
    if (ret < 0) {
        printf("select() failed.\n"); return 1;
    }
}
```



## Ví dụ 2 – Select server (version 1) (2/2)

```
if (ret == 0) {
    printf("Timed out.\n"); continue;
}
// Kiểm tra nếu là sự kiện có yêu cầu kết nối
if (FD_ISSET(listener, &fdread)) {
    int client = accept(listener, NULL, NULL);
    clients[numClients++] = client;
}
// Kiểm tra sự kiện nhận dữ liệu của các socket client
for (int i = 0; i < numClients; i++)
    if (FD_ISSET(clients[i], &fdread)) {
        ret = recv(clients[i], buf, sizeof(buf), 0);
        if (ret <= 0) {
            printf("Client %d disconnected\n", clients[i]);
            removeClient(clients, &numClients, i);
            i--;
            continue;
        }
        buf[ret] = 0;
        printf("Data from client %d: %s\n", clients[i], buf);
    }
}
```

# Ví dụ 3 – Select server (version 2) (1/2)



```
// Khai báo tập fdread chứa các socket và tập fdtest để thăm dò sự kiện
fd_set fdread, fdtest;
struct timeval tv;
char buf[2048];

FD_ZERO(&fdread);
FD_SET(listener, &fdread);

while (1) {
    fdtest = fdread; // Giữ nguyên các socket trong tập fdread
    tv.tv_sec = 5; // Khởi tạo lại giá trị cấu trúc thời gian
    tv.tv_usec = 0;

    // Chờ đến khi sự kiện xảy ra hoặc hết giờ
    int ret = select(FD_SETSIZE, &fdtest, NULL, NULL, &tv);
    if (ret < 0) {
        printf("select() failed.\n"); return 1;
    }
    if (ret == 0) {
        printf("Timed out.\n"); continue;
    }
}
```

# Ví dụ 3 – Select server (version 2) (2/2)

```
for (int i = 0; i < FD_SETSIZE; i++)
    if (FD_ISSET(i, &fdtest)) {
        if (i == listener) { // Socket listener có sự kiện yêu cầu kết nối
            int client = accept(listener, NULL, NULL);
            if (client < FD_SETSIZE) { // Chưa vượt quá số kết nối tối đa
                printf("New client connected %d\n", client);
                FD_SET(client, &fdread); // Thêm socket vào tập sự kiện
            } else { // Đã vượt quá số kết nối tối đa
                close(client);
            }
        } else { // Socket client có sự kiện nhận dữ liệu
            ret = recv(i, buf, sizeof(buf), 0);
            if (ret <= 0) {
                printf("Client %d disconnected\n", i);
                FD_CLR(i, &fdread); // Xóa socket ra khỏi tập sự kiện
            } else {
                buf[ret] = 0;
                printf("Received data from client %d: %s\n", i, buf);
            }
        }
    }
} // End while
```

# Hàm poll()

- Hàm **poll()** thực hiện chức năng tương tự hàm **select()**: đợi trên một tập mô tả cho đến khi các thao tác vào ra sẵn sàng.
- Cú pháp:

```
#include <poll.h>
```

```
int poll (
```

```
    struct pollfd *fds, // Tập hợp các mô tả cần đợi sự kiện
```

```
    nfds_t nfds, // Số lượng các mô tả, không vượt quá  
    RLIMIT_NOFILE
```

```
    int timeout // Thời gian chờ theo ms. Nếu bằng -1 thì hàm  
    chỉ trả về kết quả khi có sự kiện xảy ra.
```

```
) => Hàm trả về số lượng cấu trúc có sự kiện xảy ra nếu  
thành công, trả về -1 nếu bị lỗi. Trả về 0 nếu hết giờ.
```



# Cấu trúc pollfd

```
struct pollfd {  
    int fd; // Mô tả (socket) cần thăm dò  
    short int events; // Mặt nạ sự kiện cần kiểm tra  
    short int revents; // Mặt nạ sự kiện đã xảy ra  
}
```

- Chương trình cần thiết lập mặt nạ sự kiện trong trường **events** trước khi thăm dò và kiểm tra mặt nạ sự kiện trong trường **revents** sau khi thăm dò.
- Một số mặt nạ sự kiện hay dùng:
  - POLLIN/POLLRDNORM – Có kết nối / có dữ liệu để đọc
  - POLLOUT – Sẵn sàng ghi dữ liệu
  - POLLERR – Lỗi đọc / ghi dữ liệu

# Ví dụ 4 – Poll client



```
// Khởi tạo client

struct pollfd fds[2];
fds[0].fd = STDIN_FILENO; // Mô tả của thiết bị nhập dữ liệu
fds[0].events = POLLIN;
fds[1].fd = client;       // Mô tả của socket client
fds[1].events = POLLIN;

while (1) {
    int ret = poll(fds, 2, -1);
    if (fds[0].revents & POLLIN) { // Nếu có dữ liệu từ bàn phím
        fgets(buf, sizeof(buf), stdin);
        send(client, buf, strlen(buf), 0);
    }
    if (fds[1].revents & POLLIN) { // Nếu có dữ liệu từ socket
        ret = recv(client, buf, sizeof(buf), 0);
        if (ret <= 0) break;
        buf[ret] = 0;
        printf("Received: %s\n", buf);
    }
}
```

# Ví dụ 5 – Poll server (1/2)



```
// Khởi tạo server
struct pollfd fds[64];
int nfd = 1;

fds[0].fd = listener;
fds[0].events = POLLIN;

char buf[2048];

while (1) {
    int ret = poll(fds, nfd, -1);
    if (fds[0].revents & POLLIN) { // Sự kiện có kết nối mới
        int client = accept(listener, NULL, NULL);
        printf("New client connected %d\n", client);
        fds[nfd].fd = client;
        fds[nfd].events = POLLIN;
        nfd++;
    }
}
```



## Ví dụ 5 – Poll server (2/2)

```
for (int i = 1; i < nfd; i++)
    if (fds[i].revents & (POLLIN | POLLERR)) { // Sự kiện client
        ret = recv(fds[i].fd, buf, sizeof(buf), 0);
        if (ret <= 0) {
            // Xử lý khi kết nối client bị ngắt
        }

        // Xử lý dữ liệu từ client
        buf[ret] = 0;
        printf("Received data from client %d: %s\n", fds[i].fd, buf);
    }
} // end while
```

Sử dụng hàm `select()/poll()`, viết chương trình **chat\_server** thực hiện các chức năng sau:

Nhận kết nối từ các client, và vào hỏi tên client cho đến khi client gửi đúng cú pháp:

**“client\_id: client\_name”**

trong đó `client_name` là tên của client, xâu ký tự viết liền.

Sau đó nhận dữ liệu từ một client và gửi dữ liệu đó đến các client còn lại, ví dụ: client có id “abc” gửi “xin chào” thì các client khác sẽ nhận được: “abc: xin chào” hoặc có thể thêm thời gian vào trước ví dụ: “2023/05/06 11:00:00PM abc: xin chào”.

# Bài tập – Telnet Server

Sử dụng hàm `select()/poll()`, viết chương trình **telnet\_server** thực hiện các chức năng sau:

Khi đã kết nối với 1 client nào đó, yêu cầu client gửi user và pass, so sánh với file cơ sở dữ liệu là một file text, mỗi dòng chứa một cặp user + pass ví dụ:

admin admin

guest nopass

...

Nếu so sánh sai, không tìm thấy tài khoản thì báo lỗi đăng nhập.

Nếu đúng thì đợi lệnh từ client, thực hiện lệnh và trả kết quả cho client.

Dùng hàm `system("dir > out.txt")` để thực hiện lệnh.

dir là ví dụ lệnh dir mà client gửi.

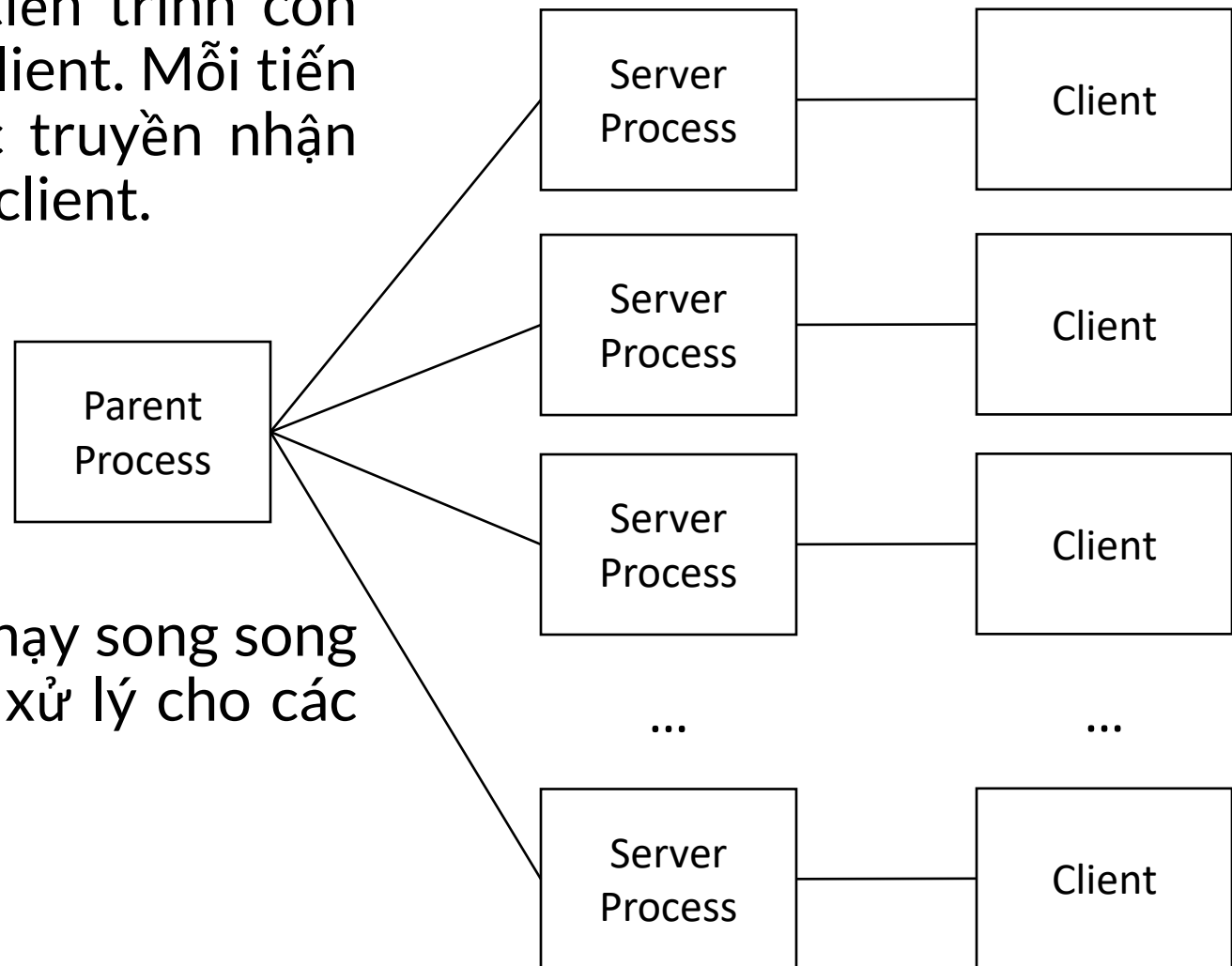
> out.txt để định hướng lại dữ liệu ra từ lệnh dir, khi đó kết quả lệnh dir sẽ được ghi vào file văn bản.

## 3.4. Multiprocess – Kỹ thuật đa tiến trình

- Khái niệm tiến trình (**process**): Tiến trình là một thực thể (instance) của một chương trình đang chạy trong hệ thống.
- So sánh tiến trình và chương trình (**process and program**): Chương trình là một file chứa các chỉ thị mô tả cách thực hiện tiến trình. Một chương trình có thể tạo nhiều tiến trình hoặc nhiều tiến trình có thể chạy cùng một chương trình.
- So sánh tiến trình và luồng (**process and thread**): Luồng là một kiểu tiến trình gọn nhẹ, chiếm ít tài nguyên hơn, và cho phép chia sẻ tài nguyên giữa các luồng. Một tiến trình có thể tạo thêm nhiều luồng.

## 3.4. Multiprocess – Kỹ thuật đa tiến trình

- Tạo thêm các tiến trình con để quản lý các client. Mỗi tiến trình xử lý việc truyền nhận dữ liệu cho một client.



- Các tiến trình chạy song song => Tăng tốc độ xử lý cho các client.



# Các hàm được sử dụng khi làm việc với tiến trình

- **fork()** – tạo tiến trình mới
- **getpid()** – trả về ID của tiến trình hiện tại
- **exit()** – kết thúc tiến trình đang thực hiện
- **wait()** – đợi tiến trình con kết thúc
- **kill()** – kết thúc tiến trình con từ tiến trình cha

# Hàm `fork()` – Tạo tiến trình mới

- Được sử dụng trong các chương trình đa tiến trình cần xử lý các công việc đồng thời.
- Khi hàm **`fork()`** được gọi, tiến trình cha tạo một bản sao của chính nó, bao gồm các lệnh, biến, con trỏ, ... ngoại trừ id của tiến trình.
- Tiến trình con chạy ngay sau lời gọi hàm **`fork()`** cho đến hết chương trình hoặc đến khi gặp lệnh **`exit`**.
- Cần kiểm soát trạng thái kết thúc của tiến trình con, tránh gây lãng phí tài nguyên hệ thống.
- Mặc định các tiến trình không chia sẻ bộ nhớ với nhau. Có thể sử dụng các kỹ thuật để chia sẻ bộ nhớ giữa các tiến trình như **`pipe`**, **`nmap`**, **`shmget`**, **`socket`**...

# Cú pháp hàm `fork()`

```
#include <unistd.h>
pid_t fork (void);
```

Hàm **`fork()`** trả về 2 giá trị trong 2 tiến trình:

- Trả về ID của tiến trình con mới được tạo trong tiến trình cha (là tiến trình gọi hàm **`fork()`**)
- Trả về 0 trong tiến trình con

⇒ Sử dụng giá trị ID trả về để phân biệt tiến trình cha và tiến trình con mới được tạo ra.

Trong trường hợp bị lỗi, hàm trả về -1, xác định lỗi thông qua **`errno`**.

# Ví dụ 1

```
#include <stdio.h>
```

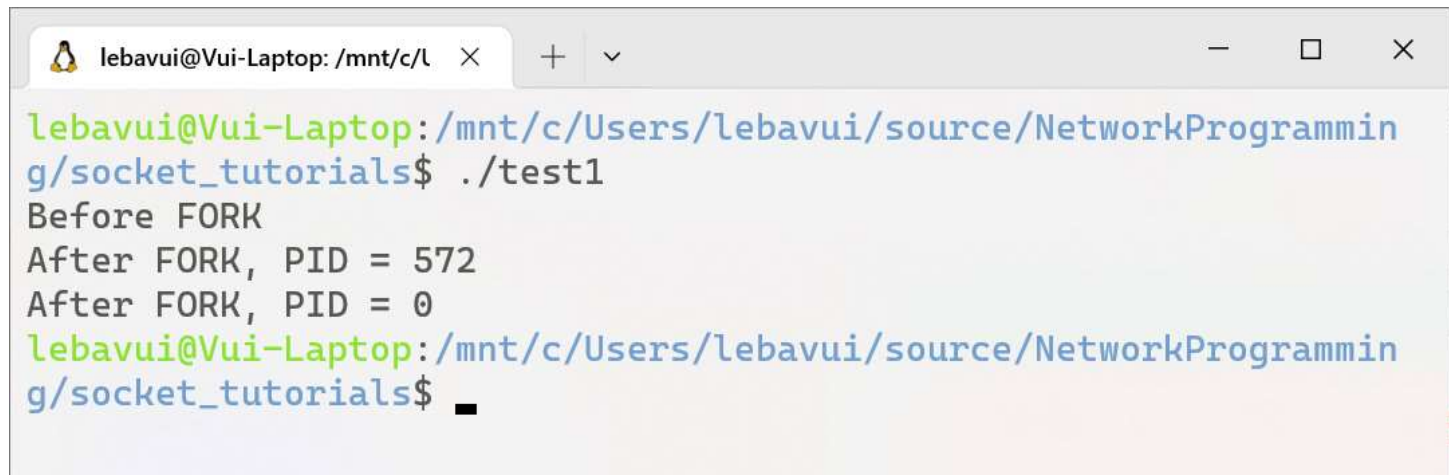
```
#include <unistd.h>
```

```
int main() {  
    printf("Before FORK\n");  
    int cid = fork();  
    printf("After FORK, PID = %d\n", cid);  
    return 0;  
}
```

Parent process

Parent process  
cid > 0

Child process  
cid = 0



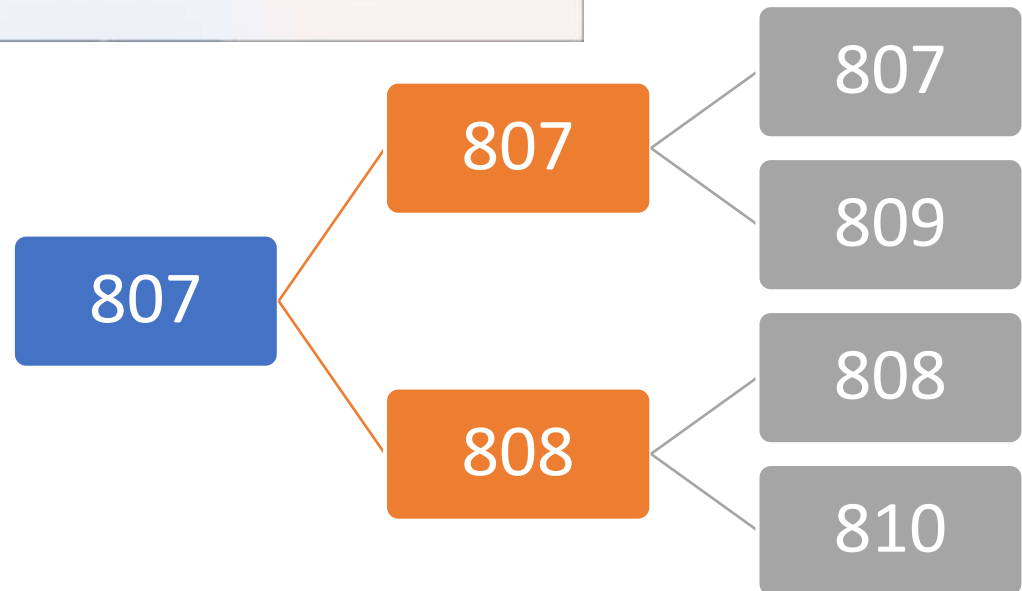
```
lebevui@Vui-Laptop: /mnt/c/l x + v  
lebevui@Vui-Laptop: /mnt/c/Users/lebevui/source/NetworkProgrammin  
g/socket_tutorials$ ./test1  
Before FORK  
After FORK, PID = 572  
After FORK, PID = 0  
lebevui@Vui-Laptop: /mnt/c/Users/lebevui/source/NetworkProgrammin  
g/socket_tutorials$ _
```

## Ví dụ 2

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int pid = getpid();
    printf("Before FORK, PID = %d\n", pid);
    fork();
    fork();
    pid = getpid();
    printf("After FORK, PID = %d\n", pid);
    return 0;
}
```

# Ví dụ 2 – Kết quả chạy

```
lelavui@Vui-Laptop: /mnt/c/l × + v
lelavui@Vui-Laptop: /mnt/c/Users/lelavui/source/NetworkProgrammin
g/socket_tutorials$ ./test1
Before FORK, PID = 807
PID = 807
PID = 809
PID = 808
PID = 810
lelavui@Vui-Laptop: /mnt/c/Users/lelavui/source/NetworkProgrammin
g/socket_tutorials$ _
```



# Hàm `exit()` – Kết thúc tiến trình

```
#include <unistd.h>  
void exit (int status);
```

Hàm **`exit()`** kết thúc tiến trình hiện tại (là tiến trình gọi hàm) kèm theo trạng thái kết thúc.

Trạng thái kết thúc có thể chọn 1 trong 2 hằng số sau:

`EXIT_SUCCESS (0)`

`EXIT_FAILURE (1)`

# Kiểm soát hoạt động của các tiến trình con

- Sử dụng hàm **wait()** để đợi tiến trình con kết thúc.
- Sử dụng hàm **kill()** để chủ động kết thúc tiến trình con.
- Sử dụng cơ chế báo hiệu (signal) để xử lý sự kiện kết thúc của các tiến trình con theo cơ chế bất đồng bộ.
- 2 trường hợp cần xử lý:
  - Tiến trình cha kết thúc trước tiến trình con => Cần chủ động kết thúc các tiến trình con có khả năng bị lặp vô hạn => Sử dụng lệnh **kill()**
  - Tiến trình con kết thúc khi tiến trình cha đang hoạt động => Sử dụng cơ chế báo hiệu



# Hàm wait()

```
#include <sys/wait.h>
pid_t wait (int *status);
```

Hàm đợi cho đến khi một trong các tiến trình con kết thúc.

Hàm trả về ID của tiến trình con vừa kết thúc, trả về -1 nếu bị lỗi.

Con trỏ status chứa trạng thái kết thúc của tiến trình con (EXIT\_SUCCESS hoặc EXIT\_FAILURE).

Nếu không có tiến trình con nào, hàm trả về -1 với mã lỗi là ECHILD.

# Ví dụ 3 - hàm wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    if (fork() == 0) {        // tiến trình con
        printf("Child process started\n");
        sleep(5);
        printf("Child process done\n");
        exit(EXIT_SUCCESS);
    }
    // tiến trình cha
    printf("Waiting for the child process\n");
    int status;
    int pid = wait(&status); // dừng và đợi cho đến khi tiến trình con kết thúc
    printf("Child process %d terminated with status %d\n", pid, status);
    return 0;
}
```

# Hàm kill()/killpg()

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
int killpg (int pgrp, int sig);
```

Hàm **kill()** thực hiện việc gửi tín hiệu **sig** đến tiến trình **pid**. Có thể được sử dụng để kết thúc tiến trình với **sig** là **SIGKILL**.

Hàm **killpg()** thực hiện việc gửi tín hiệu **sig** đến một nhóm các tiến trình. Có thể được sử dụng để kết thúc tiến trình cha và các tiến trình con với **pgrp** là 0 và **sig** là **SIGKILL**.

# Ví dụ 4 - hàm kill()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int main()
{
    int cid = fork();
    if (cid == 0) {    // child process
        printf("Child process started\n");
        while (1) {
            sleep(1);
            printf("Child process running\n");
        }
        exit(EXIT_SUCCESS);
    }
    printf("Parent process\n");
    sleep(5);
    kill(cid, SIGKILL); // Nếu không có hàm này, tiến trình con vẫn tiếp tục chạy
    printf("Parent done\n");
    return 0;
}
```

# Ví dụ 5 - hàm killpg()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int main() {
    for (int i = 0; i < 5; i++) {
        if (fork() == 0) {    // child process
            int pid = getpid();
            printf("Child process %d started\n", pid);
            while (1) {
                sleep(1); printf("Child process %d running\n", pid);
            }
        }
    }
    printf("Parent process\n");
    sleep(5);
    printf("Parent done\n");
    killpg(0, SIGKILL);
    return 0;
}
```

# Cơ chế báo hiệu (signalling)

- Tín hiệu được thông báo tới một tiến trình khi một sự kiện xảy ra.
- Tín hiệu xảy ra bất đồng bộ.
- Tín hiệu được gửi:
  - Từ tiến trình này sang tiến trình khác
  - Từ hệ thống tới tiến trình
- Tín hiệu **SIGCHLD** được gửi bởi hệ thống tới tiến trình cha khi một tiến trình con kết thúc.
- Nếu tín hiệu này không được xử lý bởi tiến trình cha, tiến trình con sẽ ở trạng thái zombie => gây tốn bộ nhớ

# Tiến trình zombie

```
lelavui@Home-Desktop: /mnt x lelavui@Home-Desktop: ~ x lelavui@Home-Desktop: ~ x + v
993 tty1 /home/lelavui/.vscode-serve Sl
1118 tty3 ps -e -o pid,tty,cmd,stat R
lelavui@Home-Desktop:~$ ps -e -o pid,tty,cmd,stat
PID TT CMD STAT
1 ? /init Ssl
6 ? plan9 --control-socket 6 -- Sl
12 tty1 /init Ss
13 tty1 sh -c "$VSCODE_WSL_EXT_LOCA S
14 tty1 sh /mnt/c/Users/ADMIN/.vsc S
19 tty1 sh /home/lelavui/.vscode-se S
23 tty1 /home/lelavui/.vscode-serve Sl
34 tty1 /home/lelavui/.vscode-serve Sl
439 tty1 /home/lelavui/.vscode-serve Sl
465 tty1 /home/lelavui/.vscode-serve Sl
493 pts/1 /bin/bash --init-file /home Ss
516 tty1 /home/lelavui/.vscode-serve Sl
558 tty1 /home/lelavui/.vscode-serve Sl
718 tty2 /init Ss
719 tty2 -bash S
745 tty3 /init Ss
746 tty3 -bash S
762 tty4 /init Ss
763 tty4 -bash S
947 tty1 /home/lelavui/.vscode-serve Sl
993 tty1 /home/lelavui/.vscode-serve Sl
1119 tty2 ./fork_server S
1121 tty2 [fork_server] <defunct> Z
1123 tty2 [fork_server] <defunct> Z
1124 tty3 ps -e -o pid,tty,cmd,stat R
lelavui@Home-Desktop:~$
```

Sử dụng lệnh  
ps -e -o pid,tty,cmd,stat  
để kiểm tra trạng thái các tiến  
trình đang hoạt động

← Tiến trình zombie

# Xử lý sự kiện kết thúc tiến trình con

- Sử dụng hàm **signal()** để đăng ký việc xử lý khi sự kiện **SIGCHLD** xảy ra.

```
signal(SIGCHLD, signalHandler);
```

=> Khi có tín hiệu **SIGCHLD** được gửi đến, hàm xử lý sự kiện **signalHandler()** được gọi.

=> Cần gọi hàm **signal()** trước hàm **fork()**.

- Hàm xử lý sự kiện

```
void signalHandler(int signo) {  
    int stat;  
    pid_t pid = wait(&stat);  
    printf("Child %d terminated.\n", pid);  
}
```



# Ví dụ 6 – cơ chế báo hiệu

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void signalHandler(int signo) {
    int pid = wait(NULL);
    printf("Child %d terminated\n", pid);
}

int main() {
    signal(SIGCHLD, signalHandler);
    if (fork() == 0) {    // child process
        printf("Child process started\n");
        sleep(5);
        printf("Child process done\n");
        exit(EXIT_SUCCESS);
    }
    // main process
    getchar();
    return 0;
}
```

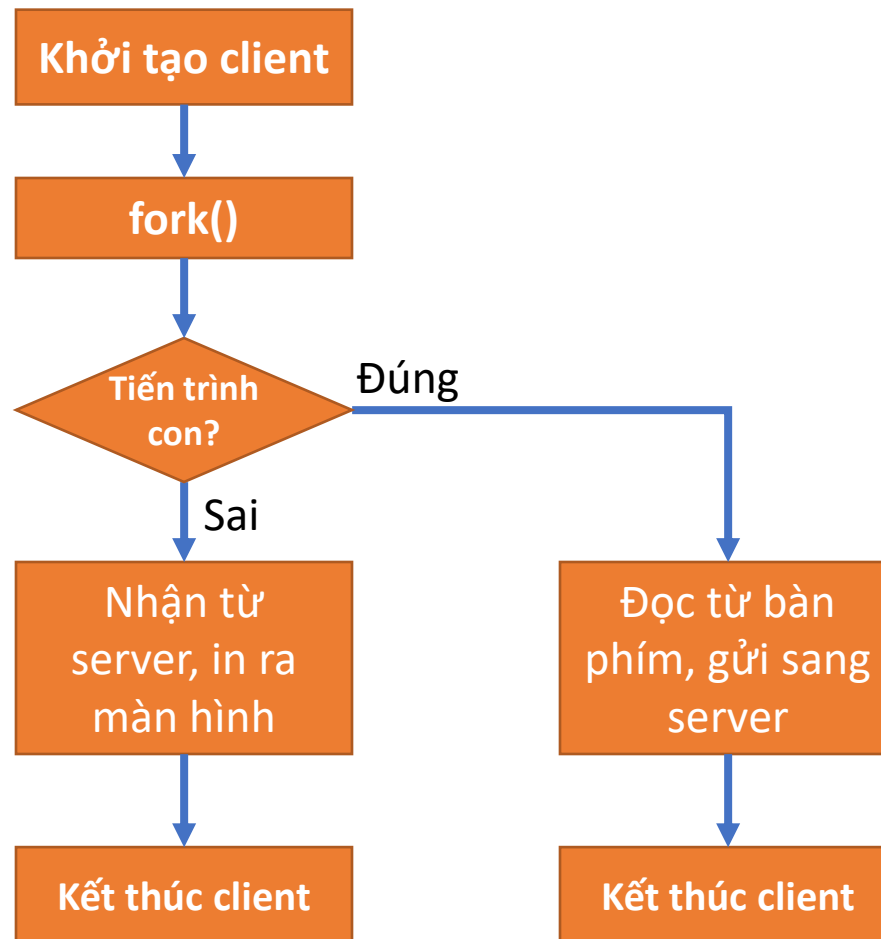
# Sử dụng kỹ thuật đa tiến trình để xử lý nhiều kết nối

- Với mỗi kết nối được chấp nhận, một tiến trình con được tạo ra để xử lý cho kết nối đó.
- Do các socket được sao chép ở các tiến trình con  
=> Cần đóng socket chờ kết nối ở tiến trình con, và socket client ở tiến trình cha, tránh việc tham chiếu sai socket hoặc gây tiêu tốn tài nguyên hệ thống.

# Sử dụng kỹ thuật đa tiến trình để xử lý nhiều kết nối

```
while (1) {  
    int client = accept(listener, NULL, NULL);  
    if (fork() == 0) {  
        // Trong tiến trình con  
        close(listener);  
  
        // Xử lý yêu cầu từ client ...  
  
        close(client);  
        exit(0);  
    }  
  
    // Trong tiến trình cha  
    close(client);  
    ...  
}
```

# Ứng dụng client



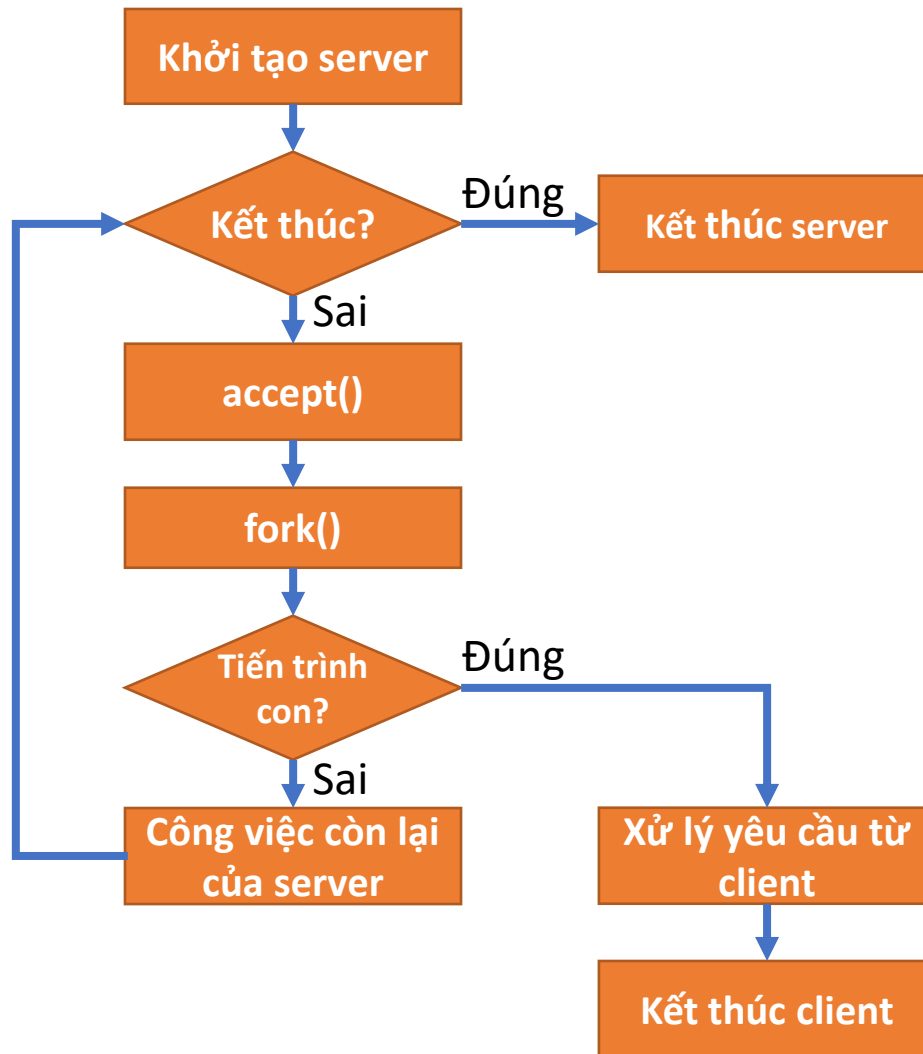
# Ví dụ 7 - Ứng dụng client



```
if (fork() == 0) { // Tiến trình con
    while (1) {
        fgets(buf, sizeof(buf), stdin);
        send(client, buf, strlen(buf), 0);
        if (strncmp(buf, "exit", 4) == 0) break;
    }
} else { // Tiến trình cha
    while (1) {
        int ret = recv(client, buf, sizeof(buf), 0);
        if (ret <= 0) break;
        buf[ret] = 0;
        printf("Received: %s\n", buf);
    }
}

close(client); // Đóng socket
killpg(0, SIGKILL); // Dừng các tiến trình
```

# Ứng dụng server



# Ví dụ 8 - Ứng dụng server (1/2)

```
void signalHandler(int signo) { // Xử lý sự kiện tiến trình con kết thúc
    int stat;
    printf("signo = %d\n", signo);
    int pid = wait(&stat);
    printf("child %d terminated.\n", pid);
    return;
}

int main() {
    // Server đã được khởi tạo
    signal(SIGCHLD, signalHandler);
    while (1) {
        printf("Waiting for new client\n");
        int client = accept(listener, NULL, NULL);
        printf("New client accepted: %d\n", client);
    }
}
```



## Ví dụ 8 - Ứng dụng server (2/2)

```
if (fork() == 0) {  
    // in child process  
    close(listener);  
    char buf[256];  
    while (1) {  
        int ret = recv(client, buf, sizeof(buf), 0);  
        if (ret <= 0) break;  
        buf[ret] = 0;  
        printf("Received from client %d: %s\n", client, buf);  
        send(client, buf, strlen(buf), 0);  
    }  
    close(client);  
    exit(0);  
}  
close(client);  
}
```



# Preforking

- Việc tạo một tiến trình cho mỗi client kết nối đến server gây tốn kém về thời gian và tài nguyên của hệ thống.
- Để hạn chế việc tạo quá nhiều tiến trình, kỹ thuật preforking có thể được áp dụng, đặc biệt với những server xử lý yêu cầu của các client trong thời gian ngắn.
- Ứng dụng tạo sẵn một số giới hạn các process, mỗi process sẽ thực hiện lặp lại các công việc: chờ kết nối, xử lý yêu cầu và trả kết quả cho client.
- Mỗi process sẽ xử lý luân phiên cho một client.

# Ví dụ 9 – Preforking server



```
int num_processes = 8;
char buf[256];
for (int i = 0; i < num_processes; i++)
    if (fork() == 0)
        while(1) {
            // Chờ kết nối
            int client = accept(listener, NULL, NULL);
            printf("New client accepted in process %d: %d\n", client, getpid());

            // Chờ dữ liệu từ client
            int ret = recv(client, buf, sizeof(buf), 0);
            if (ret <= 0)
                continue;

            // Xử lý dữ liệu, trả lại kết quả cho client
            buf[ret] = 0;
            printf("Received from client %d: %s\n", client, buf);
            send(client, buf, strlen(buf), 0);

            // Đóng kết nối
            close(client);
        }

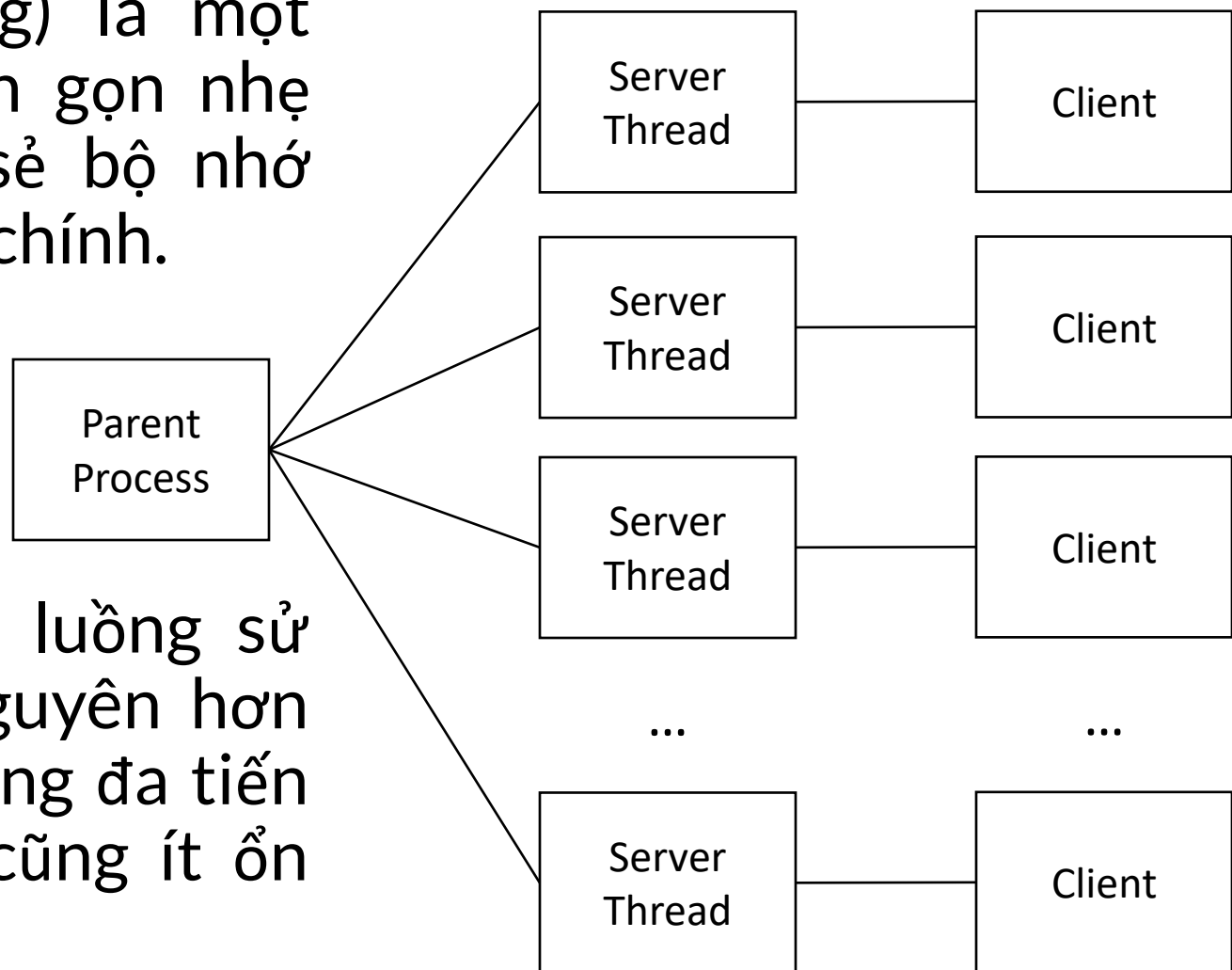
wait(NULL); // getchar(); killpg(0, SIGKILL);
```

- Lập trình lại ứng dụng **telnet\_server** ([slide 174](#)) sử dụng kỹ thuật multiprocessing.
- Lập trình lại ứng dụng **http\_server** ([slide 146](#)) sử dụng kỹ thuật preforking.

- Lập trình ứng dụng **time\_server** thực hiện chức năng sau:
  - Chấp nhận nhiều kết nối từ các client sử dụng kỹ thuật multiprocessing.
  - Client gửi lệnh GET\_TIME [**format**] để nhận thời gian từ server.
  - **format** là định dạng thời gian server cần trả về client. Các format cần hỗ trợ gồm:
    - dd/mm/yyyy – vd: 30/01/2023
    - dd/mm/yy – vd: 30/01/23
    - mm/dd/yyyy – vd: 01/30/2023
    - mm/dd/yy – vd: 01/30/23
  - Cần kiểm tra tính đúng đắn của lệnh client gửi lên server.

## 3.5. Multithreading

- Thread (luồng) là một loại tiến trình gọn nhẹ có thể chia sẻ bộ nhớ với tiến trình chính.



- Ứng dụng đa luồng sử dụng ít tài nguyên hơn so với ứng dụng đa tiến trình nhưng cũng ít ổn định hơn.

# Các hàm làm việc với luồng

- **pthread\_create()**: tạo luồng mới
- **pthread\_join()**: đợi luồng kết thúc
- **pthread\_self()**: trả về ID của luồng, được gọi bên trong luồng
- **pthread\_detach()**: đánh dấu luồng ở trạng thái “tách rời”, khi luồng kết thúc ở trạng thái này các tài nguyên của luồng sẽ được tự giải phóng.
- **pthread\_exit()**: dừng luồng

# Hàm pthread\_create()

```
#include <pthread.h>
```

```
int pthread_create (  
    pthread_t *newthread, // Trả về ID của luồng  
    const pthread_attr_t *attr, // Tham số tạo luồng, mặc định  
    NULL  
    void *(*start_routine) (void *), // Hàm thực thi  
    void *arg // Tham số truyền vào hàm thực thi  
)
```

=> Hàm trả về 0 nếu thành công, trả về mã lỗi nếu thất bại.

# Ví dụ 1 – Tạo luồng mới

```
#include <stdio.h>
#include <pthread.h>
void *thread_proc(void *);
int main() {
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, thread_proc, NULL)) {
        printf("Failed to create new thread\n");
        return 1;
    }
    /* MAIN THREAD JOB */
    return 0;
}

void *thread_proc(void *arg) {
    /* CHILD THREAD JOB */
}
```



# Hàm pthread\_join()

```
int pthread_join (  
    pthread_t thread_id, // ID của luồng cần đợi  
    void **thread_return // Trạng thái kết thúc của luồng  
)
```

Hàm **pthread\_join()** đợi cho đến khi luồng kết thúc, nếu luồng đã kết thúc rồi thì hàm trả về kết quả ngay lập tức. Hàm trả về 0 nếu thành công, trả về mã lỗi nếu thất bại.

# Ví dụ 2 – Đợi luồng kết thúc

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *thread_proc(void *);
int main() {
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, thread_proc, NULL)) {
        printf("Failed to create new thread\n");
        return 1;
    }
    pthread_join(thread_id, NULL);
    return 0;
}
void *thread_proc(void *arg) {
    for (int i = 0; i < 20; i++) {
        printf("Child thread %d\n", i);
        sleep(1);
    }
}
```

# Ví dụ 3 - Chương trình tính xấp xỉ số PI

```
#include <stdio.h>
static long num_steps = 100000;
int main() {
    int i;
    double x, sum = 0.0;

    double step = 1.0 / (double)num_steps;
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }

    double pi = step * sum;
    printf("PI = %.10f\n", pi);
}
```

Đoạn chương trình được sử dụng để tính xấp xỉ số PI.

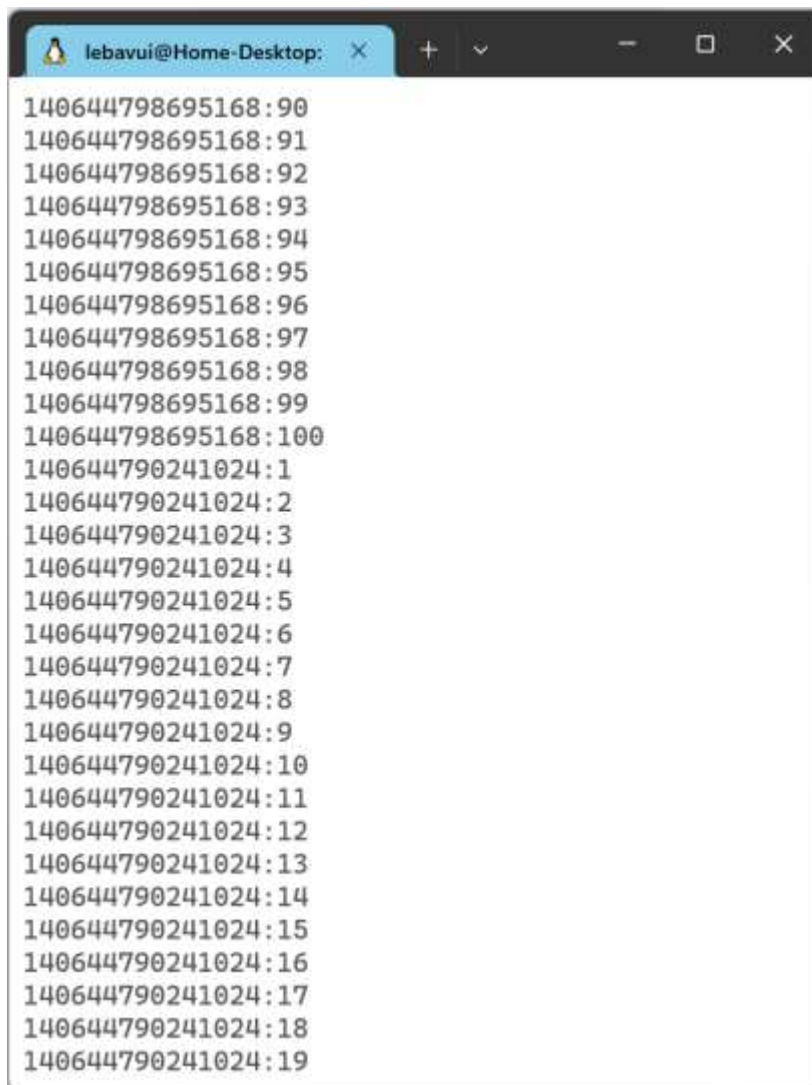
Hãy viết lại chương trình sử dụng kỹ thuật đa luồng.

# Đồng bộ giữa các luồng

- Việc truy nhập đồng thời vào cùng một biến từ nhiều luồng sẽ dẫn đến các xung đột:
  - Xung đột đọc / ghi dữ liệu
  - Xung đột ghi / ghi dữ liệu
- Lỗi phổ biến trong lập trình đa luồng.
- Lỗi thường không rõ ràng, khó gỡ lỗi.

# Ví dụ 4a – Xung đột giữa các luồng

```
#include <stdio.h>
#include <pthread.h>
#define NLOOP 100
int counter;
void *doit(void *);
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, doit, NULL);
    pthread_create(&t2, NULL, doit, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
void *doit(void *arg) {
    for (int i = 0; i < NLOOP; i++) {
        int val = counter;
        printf("%ld: %d\n", pthread_self(), val + 1);
        counter = val + 1;
    }
}
```



```
140644798695168:90
140644798695168:91
140644798695168:92
140644798695168:93
140644798695168:94
140644798695168:95
140644798695168:96
140644798695168:97
140644798695168:98
140644798695168:99
140644798695168:100
140644790241024:1
140644790241024:2
140644790241024:3
140644790241024:4
140644790241024:5
140644790241024:6
140644790241024:7
140644790241024:8
140644790241024:9
140644790241024:10
140644790241024:11
140644790241024:12
140644790241024:13
140644790241024:14
140644790241024:15
140644790241024:16
140644790241024:17
140644790241024:18
140644790241024:19
```

# Các phương pháp tránh xung đột

- Hạn chế sử dụng biến toàn cục, nên sử dụng biến cục bộ khai báo trong hàm thực thi của luồng.
- Quản lý việc truy nhập các tài nguyên dùng chung => Sử dụng kỹ thuật Mutex (Mutual Exclusion)

```
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
```

=> Khởi tạo đối biến mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```

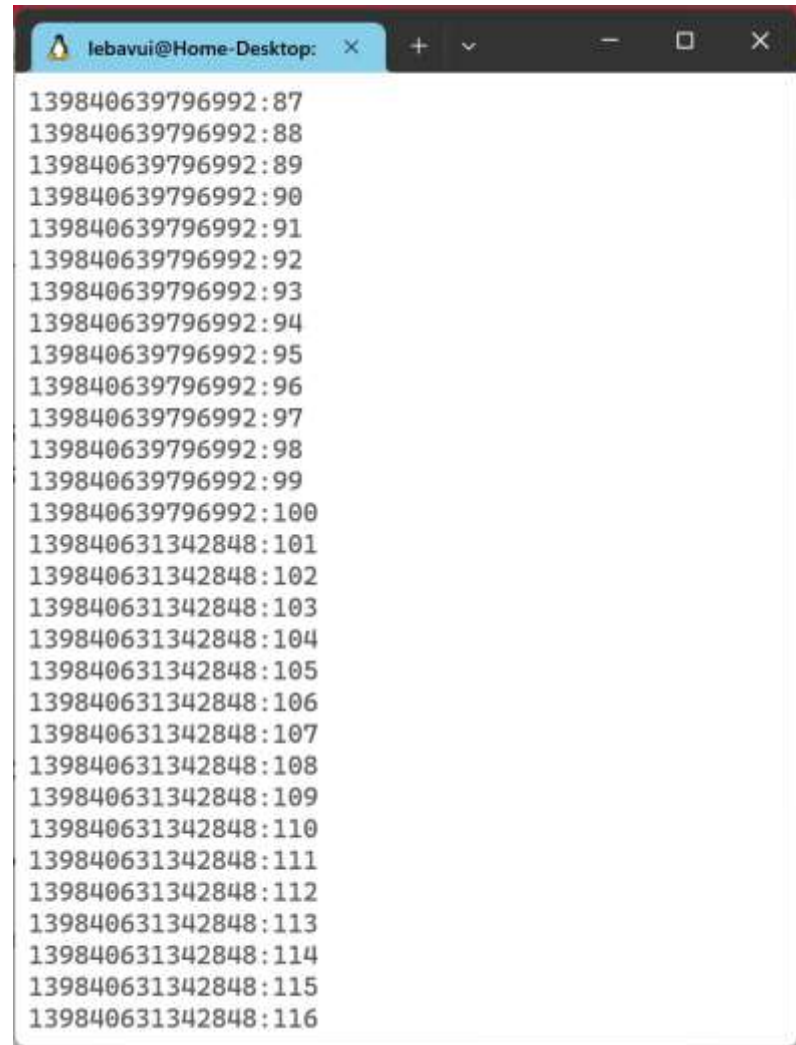
=> Khóa biến mutex. Nếu biến này đang được khóa bởi luồng khác, thì hàm sẽ đợi cho đến khi biến được mở khóa bởi luồng đó

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

=> Mở khóa biến mutex

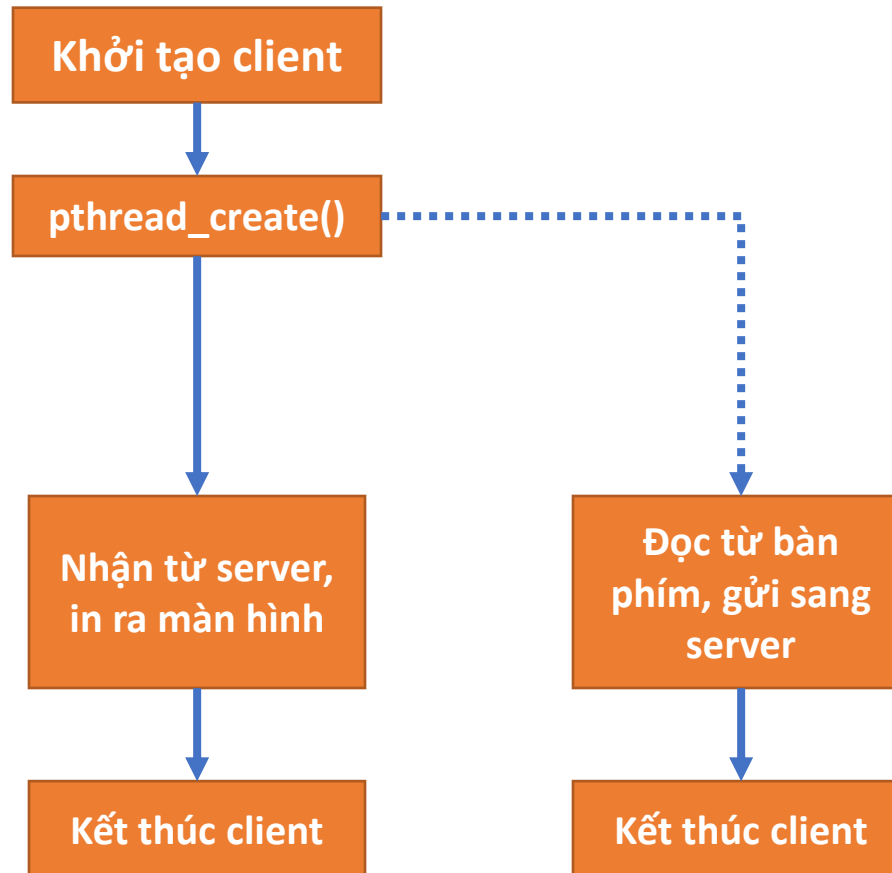
# Ví dụ 4b – Xử lý xung đột giữa các luồng

```
#include <stdio.h>
#include <pthread.h>
#define NLOOP 100
int counter;
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
void *doit(void *);
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, doit, NULL);
    pthread_create(&t2, NULL, doit, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
void *doit(void *arg) {
    for (int i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
        int val = counter;
        printf("%ld:%d\n", pthread_self(), val + 1);
        counter = val + 1;
        pthread_mutex_unlock(&counter_mutex);
    }
}
```



```
139840639796992:87
139840639796992:88
139840639796992:89
139840639796992:90
139840639796992:91
139840639796992:92
139840639796992:93
139840639796992:94
139840639796992:95
139840639796992:96
139840639796992:97
139840639796992:98
139840639796992:99
139840639796992:100
139840631342848:101
139840631342848:102
139840631342848:103
139840631342848:104
139840631342848:105
139840631342848:106
139840631342848:107
139840631342848:108
139840631342848:109
139840631342848:110
139840631342848:111
139840631342848:112
139840631342848:113
139840631342848:114
139840631342848:115
139840631342848:116
```

# Ứng dụng Multithread Client





# Ví dụ 5 - Multithread Client (1/2)

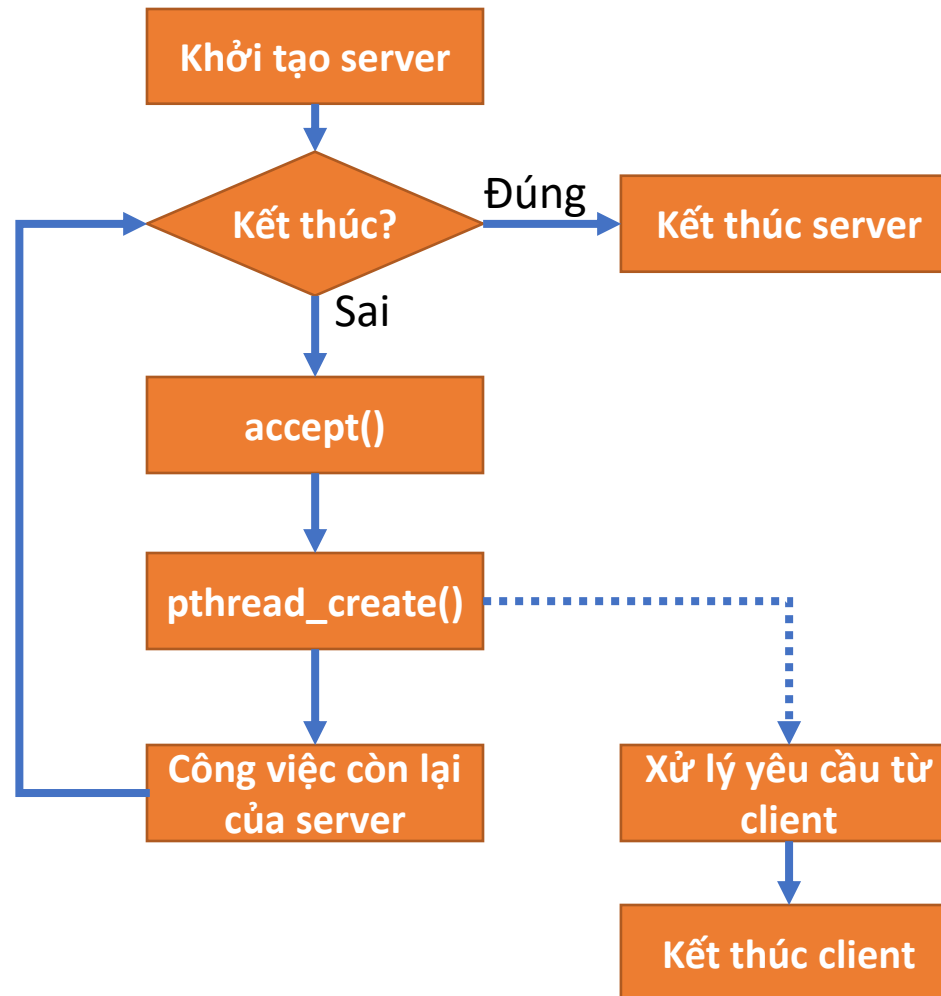


```
// Khai báo các tệp thư viện
void* thread_proc(void *arg);
int main() {
    // Khởi tạo client
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thread_proc, (void *)&client))
    char buf[256];
    while (1) {
        fgets(buf, sizeof(buf), stdin);
        send(client, buf, strlen(buf), 0);
        if (strncmp(buf, "exit", 4) == 0) break;
    }
    close(client);
    return 0;
}
```

# Ví dụ 5 - Multithread Client (2/2)

```
void* thread_proc(void *arg) {
    int client = *(int *)arg;
    char buf[256];
    while (1) {
        int len = recv(client, buf, sizeof(buf), 0);
        if (len <= 0)
            break;
        buf[len] = 0;
        printf("%s\n", buf);
    }
    close(client);
    return 0;
}
```

# Ứng dụng Multithread Server



# Ví dụ 6 - Multithread Server (1/2)



```
// Khai báo thư viện
void* thread_proc(void *arg);
int main() {
    // Khởi tạo server
    pthread_t thread_id;
    while (1) {
        int client = accept(listener, NULL, NULL);
        if (client == -1)
            continue;
        printf("New client connected: %d\n", client);
        pthread_create(&thread_id, NULL, thread_proc, (void *)&client);
        pthread_detach(thread_id);
    }
    return 0;
}
```

# Ví dụ 6 - Multithread Server (2/2)

```
void* thread_proc(void *arg) {
    int client = *(int *)arg;
    char buf[2048];
    while (1) {
        int len = recv(client, buf, sizeof(buf), 0);
        if (len <= 0)
            break;

        // Xử lý dữ liệu nhận được từ client
    }
    close(client);
}
```

# Prethreading

- Tương tự kỹ thuật Preforking nhưng không ổn định bằng do đặc tính chia sẻ bộ nhớ giữa các tiến trình.
- Ứng dụng tạo sẵn một số giới hạn các thread, mỗi thread sẽ thực hiện lặp lại các công việc: chờ kết nối, xử lý yêu cầu và trả kết quả cho client.
- Mỗi thread sẽ xử lý luân phiên cho một client.

# VD7 – Prethreading server (1/2)



```
// Khởi tạo server

int num_threads = 8;
pthread_t thread_id;

for (int i = 0; i < num_threads; i++) {
    int ret = pthread_create(&thread_id, NULL, thread_proc, &listener);
    if (ret != 0)
        printf("Could not create new thread.\n");
    sched_yield();
}

pthread_join(thread_id, NULL);
return 0;
}
```

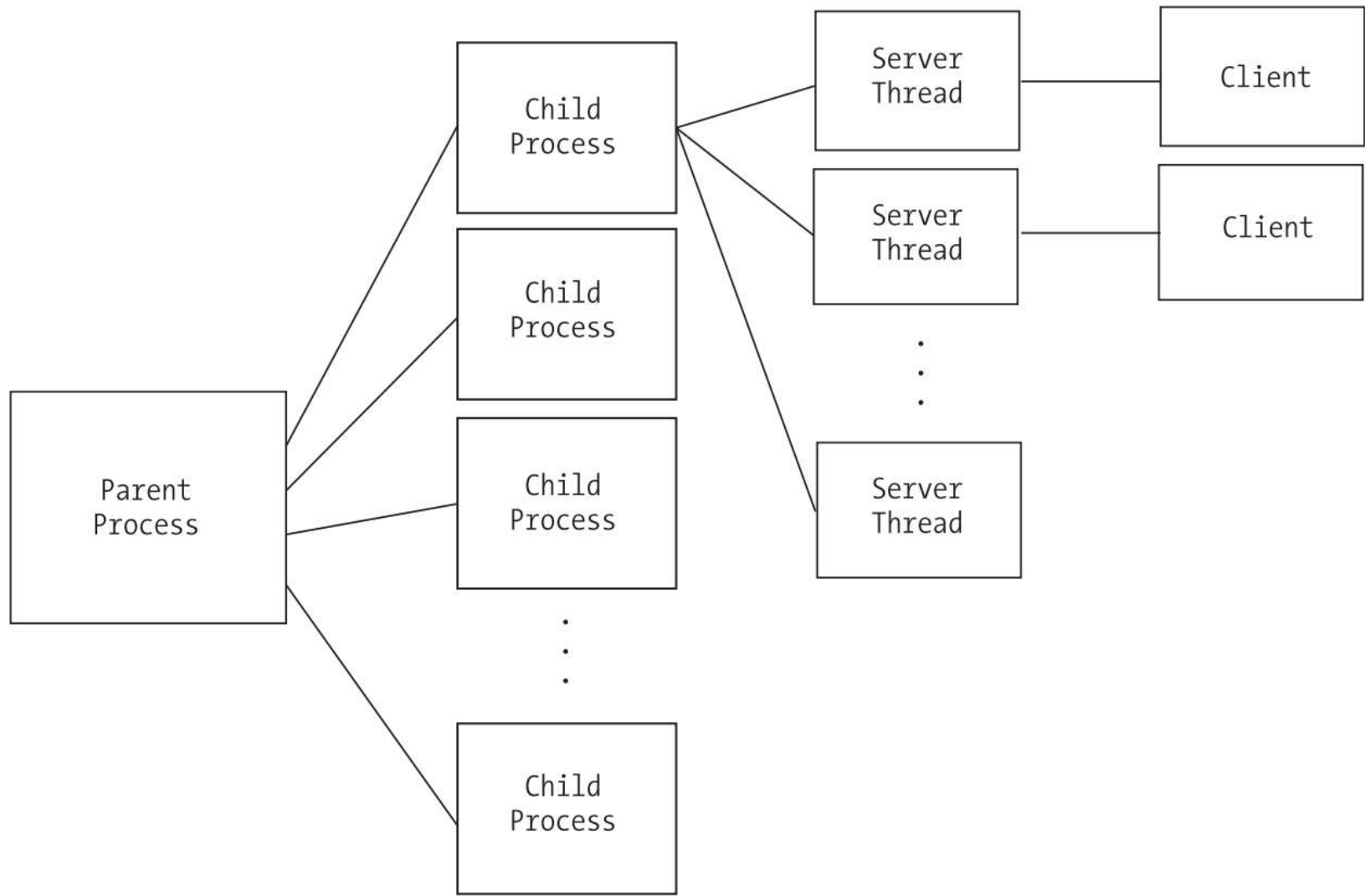
# VD7 – Prethreading server (2/2)

```
void* thread_proc(void *arg) {
    int listener = *(int *)arg;
    char buf[256];
    while (1) {
        // Chờ kết nối
        int client = accept(listener, NULL, NULL);
        printf("New client %d accepted in thread %ld with pid %d\n", client,
pthread_self(), getpid());
        // Chờ dữ liệu từ client
        int ret = recv(client, buf, sizeof(buf), 0);
        if (ret <= 0)
            continue;
        // Xử lý dữ liệu, trả lại kết quả cho client
        buf[ret] = 0;
        printf("Received from client %d: %s\n", client, buf);
        send(client, buf, strlen(buf), 0);
        // Đóng kết nối
        close(client);
    }

    return NULL;
}
```



# Kết hợp preforking và prethreading



- Lập trình ứng dụng **chat\_server** ([slide 173](#)) sử dụng kỹ thuật multithread.
- Lập trình ứng dụng **telnet\_server** ([slide 174](#)) sử dụng kỹ thuật multithread.
- Lập trình ứng dụng **time\_server** ([slide 204](#)) sử dụng kỹ thuật multithread.
- Lập trình ứng dụng **http\_server** ([slide 146](#)) sử dụng kỹ thuật prethreading.

# So sánh giữa các kiến trúc

Table 5-1. Method Pros and Cons

METHOD	CODE COMPLEXITY	SHARED MEMORY	NUMBER OF CONNECTIONS	FREQUENCY OF NEW CONNECTIONS	LENGTH OF CONNECTIONS	STABILITY	CONTEXT SWITCHING	RESOURCE USE	SMP AWARE
Multiplexing	Can be very complex and difficult to follow	Yes	Small	Can handle new connections quickly.	Good for long or short connections.	One client can crash the server.	N/A	Low	No
Forking	Simple	Only through shmget()	Large	Time is required for a new process to start.	Better for longer connections. Reduces start penalty.	One client cannot crash the server.	Not as fast as threads	High	Yes
Threading	Simple	Yes	Large, but not as large as forking	Time is required for a new thread to start.	Better for longer connections. Reduces thread start penalty.	One client can crash the server.	Fast	Medium	Yes
Preforking	Can be complex if using a dynamic pool and shared memory	Only through shmget()	Depends on the size of the process pool	Can handle new connections quickly if the process pool is large enough.	Good for long or short connections.	One client cannot crash the server.	Not as fast as threads	High	Yes
Prethreading	Only complex if using a dynamic pool	Yes	Depends on the size of the thread pool	Can handle new connections quickly if the thread pool is large enough.	Good for long or short connections.	One client can crash the server.	Fast	Medium	Yes
Preforking plus prethreading	Complex	Yes, but only within each process	Depends on pool sizes	Can handle new connections quickly if pools are large enough.	Good for long or short connections.	One client will crash only its parent process, not the whole server.	Fast	Similar to threading, but depends on how much preforking is done	Yes

# Chương 4. Thiết kế giao thức mạng

# Chương 4. Thiết kế giao thức mạng

## 4.1. Khái niệm về giao thức

## 4.2. Tìm hiểu một số giao thức phổ biến

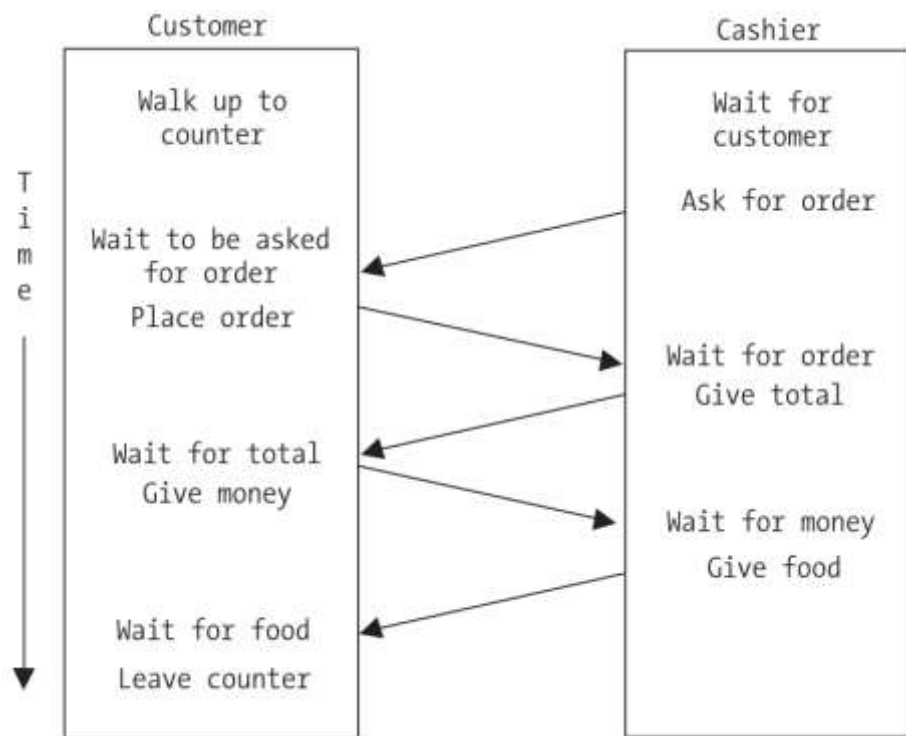
- HTTP
- FTP
- POP3

## 4.3. Thiết kế giao thức tự định nghĩa

- Ứng dụng chat

## 4.1. Khái niệm về giao thức

- Giao thức (Protocol) là tập hợp các quy ước mà client và server cần phải cùng thực hiện để có thể giao tiếp với nhau.
- Các giao thức thường chạy ở tầng ứng dụng.
- Gồm giao thức chuẩn (HTTP, FTP, POP3, ...) hoặc giao thức do người lập trình định nghĩa.
- Ví dụ giao thức giữa người phục vụ nhà hàng và khách hàng.



## 4.2. Tìm hiểu một số giao thức phổ biến

- a. Giao thức HTTP
- b. Giao thức FTP
- c. Giao thức POP3

## a. Giao thức HTTP

- Tìm hiểu về giao thức HTTP
- Lập trình ứng dụng máy chủ HTTP file
- Lập trình ứng dụng website quản lý thông tin

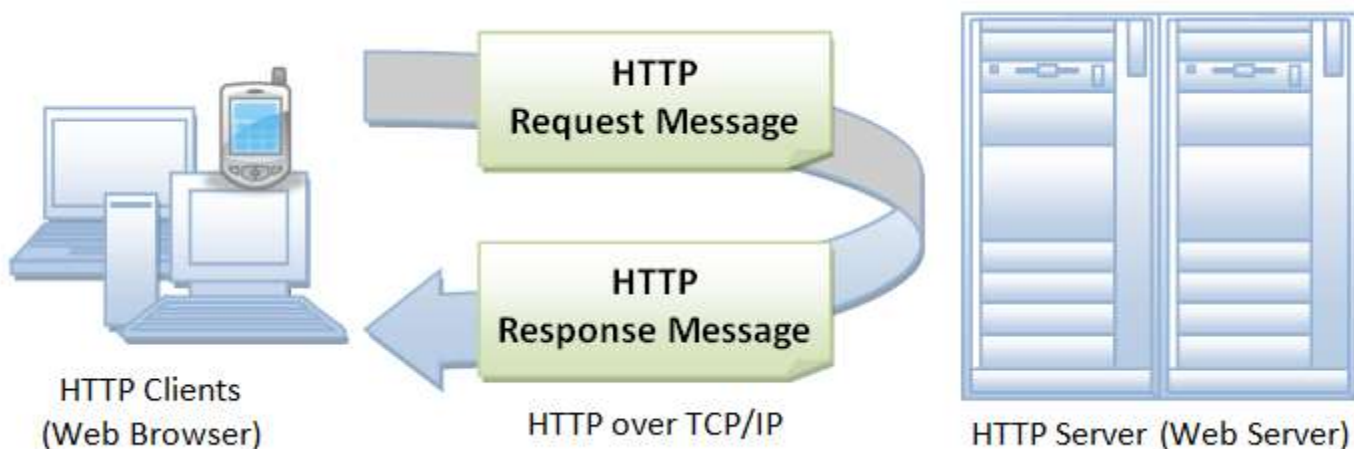


## Định nghĩa:

- HTTP (HyperText Transfer Protocol - Giao thức truyền tải siêu văn bản) là một trong các giao thức chuẩn về mạng Internet, được dùng để liên hệ thông tin giữa Máy cung cấp dịch vụ (Web server) và Máy sử dụng dịch vụ (Web client), là giao thức Client/Server dùng cho World Wide Web – WWW.
- HTTP là một giao thức ứng dụng của bộ giao thức TCP/IP (các giao thức nền tảng cho Internet).

# Tìm hiểu về giao thức HTTP

## Sơ đồ hoạt động:



- HTTP hoạt động dựa trên mô hình Client – Server. Trong mô hình này, các máy tính của người dùng sẽ đóng vai trò làm máy khách (Client). Sau một thao tác nào đó của người dùng, các máy khách sẽ gửi yêu cầu đến máy chủ (Server) và chờ đợi câu trả lời từ những máy chủ này.
- HTTP là một stateless protocol. Hay nói cách khác, request hiện tại không biết những gì đã hoàn thành trong request trước đó.

# Tìm hiểu về giao thức HTTP

HTTP Requests: Là phương thức để chỉ ra hành động mong muốn được thực hiện trên tài nguyên đã xác định.

Cấu trúc của một HTTP Request:

**Request-line = Phương thức + URI-Request + Phiên bản HTTP**

Giao thức HTTP định nghĩa một tập các phương thức GET, POST, HEAD, PUT ... Client có thể sử dụng một trong các phương thức đó để gửi request lên server.

Có thể có hoặc không các trường header: Các trường header cho phép client truyền thông tin bổ sung về yêu cầu, và về chính client, đến server. Một số trường: Accept-Charset, Accept-Encoding, Accept-Language, Authorization, Expect, From, Host, ...

Một dòng trống để đánh dấu sự kết thúc của các trường Header.

Tùy chọn một thông điệp

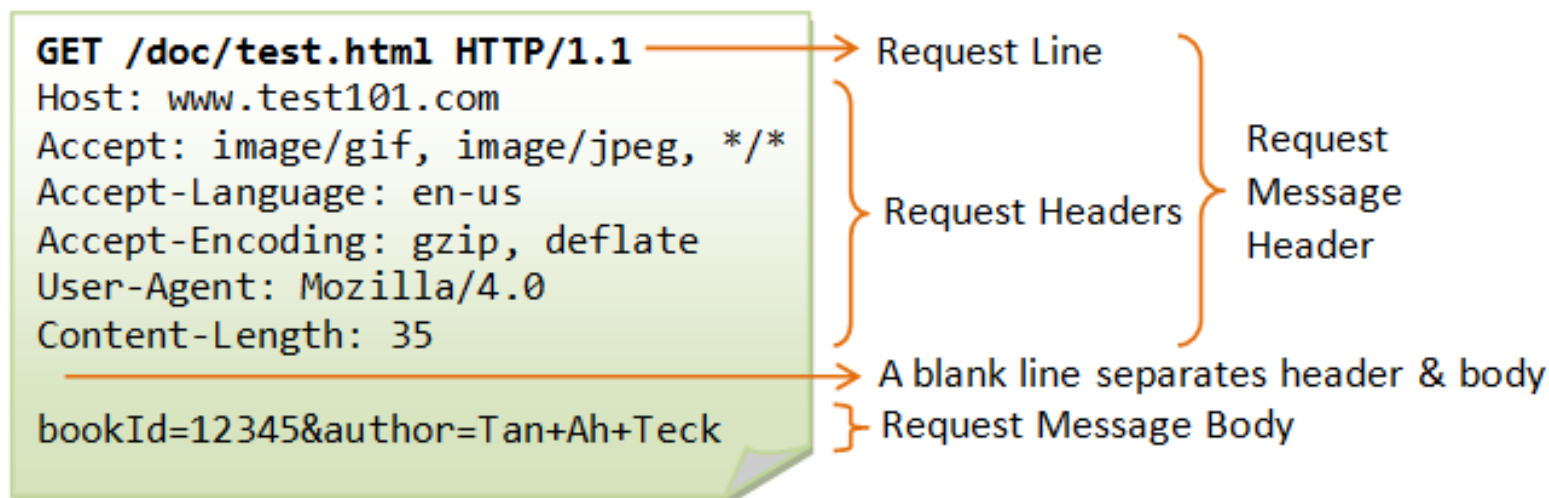
# Tìm hiểu về giao thức HTTP

- HTTP Requests:
- Các phương thức thường dùng

Method	Hoạt động	Chú thích
GET	được sử dụng để lấy lại thông tin từ Server một tài nguyên xác định.	Các yêu cầu sử dụng GET chỉ nên nhận dữ liệu và không nên có ảnh hưởng gì tới dữ liệu
POST	yêu cầu máy chủ chấp nhận thực thể được đính kèm trong request được xác định bởi URI, ví dụ, thông tin khách hàng, file tải lên, ...	
PUT	Nếu URI đề cập đến một tài nguyên đã có, nó sẽ bị sửa đổi; nếu URI không trở đến một tài nguyên hiện có, thì máy chủ có thể tạo ra tài nguyên với URI đó.	
DELETE	Xóa bỏ tất cả các đại diện của tài nguyên được chỉ định bởi URI.	
PATCH	Áp dụng cho việc sửa đổi một phần của tài nguyên được xác định.	
...	...	...

# Tìm hiểu về giao thức HTTP

- HTTP Requests:
- Ví dụ



## HTTP Responses:

- Cấu trúc của một HTTP response:

**Status-line = Phiên bản HTTP + Mã trạng thái + Trạng thái**

Có thể có hoặc không có các trường header

Một dòng trống để đánh dấu sự kết thúc của các trường header

Tùy chọn một thông điệp

## HTTP Responses:

- Mã trạng thái: Thông báo về kết quả khi nhận được yêu cầu và xử lý bên server cho client.
- Các kiểu mã trạng thái:

1xx: Thông tin (100 -> 101)

VD: 100 (Continue), ....

2xx: Thành công (200 -> 206)

VD: 200 (OK) , 201 (CREATED), ...

3xx: Sự điều hướng lại (300 -> 307)

VD: 305 (USE PROXY), ...

4xx: Lỗi phía Client (400 -> 417)

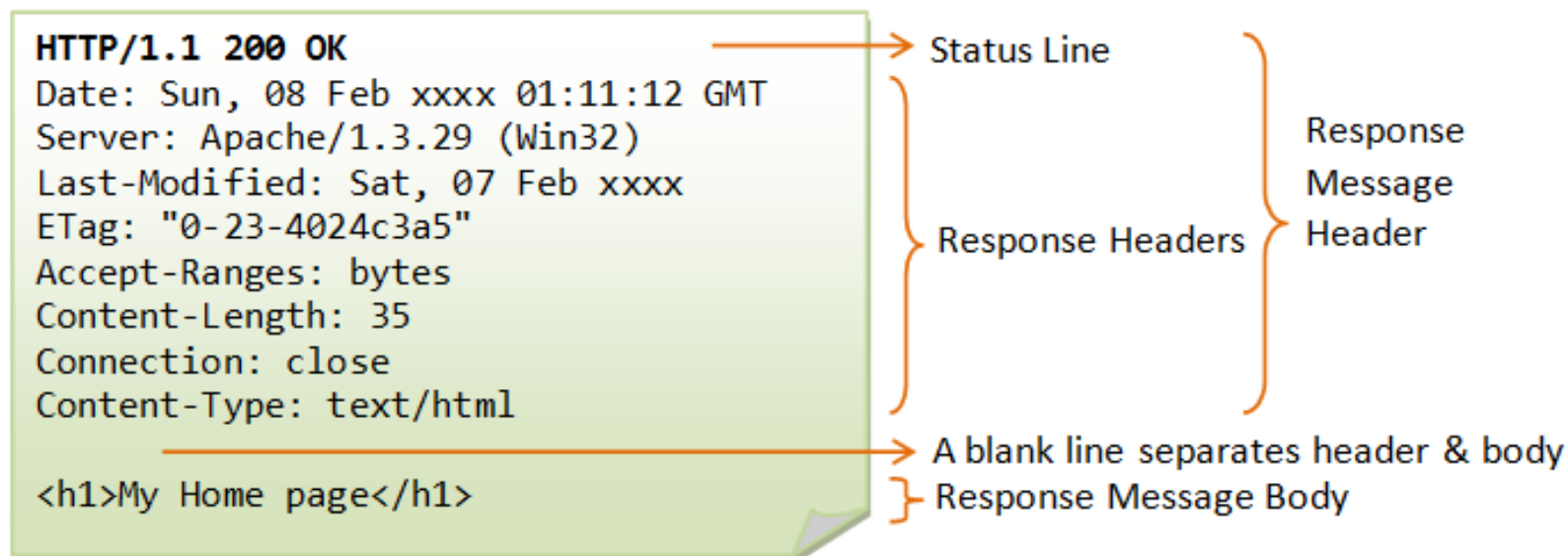
VD: 403 (FORBIDDEN), 404 (NOT FOUND), ...

5xx: Lỗi phía Server (500 -> 505)

VD: 500 (INTERNAL SERVER ERROR)

## HTTP Responses:

- Ví dụ





## 1. Lập trình ứng dụng HTTP client

- a. Gửi lệnh GET đến trang <http://httpbin.org/get>
- b. Gửi lệnh POST đến trang <http://httpbin.org/post>
- c. Download file từ Internet

[http://lebavui.github.io/network\\_programming/videos/ecard.mp4](http://lebavui.github.io/network_programming/videos/ecard.mp4)

## 2. Lập trình ứng dụng HTTP server

- a. Hiển thị trang web, hiển thị ảnh, phát audio
- b. Tạo trang web thực hiện các phép tính đơn giản
  - + Nhận tham số từ lệnh GET
  - + Nhận tham số từ lệnh POST

Server trả lại kết quả là trang web hiển thị phép tính và kết quả.

Lập trình ứng dụng máy chủ HTTP file với các chức năng:

- Hiển thị cấu trúc cây thư mục trên máy chủ.
- Khi trình duyệt yêu cầu thư mục, hiển thị nội dung của thư mục (thư mục con và files).
- Khi trình duyệt yêu cầu file, trả về nội dung của file, kèm theo kiểu file (ContentType) và kích thước file (ContentLength).

Lập trình ứng dụng máy chủ web với các chức năng:

- Thực hiện chức năng đăng nhập.
- Thực hiện chức năng đăng ký người dùng mới.
- Hiển thị danh sách người dùng với các chức năng cập nhật thông tin, xóa người dùng.

## b. Giao thức FTP

- Tìm hiểu về giao thức FTP
- Lập trình ứng dụng máy khách FTP

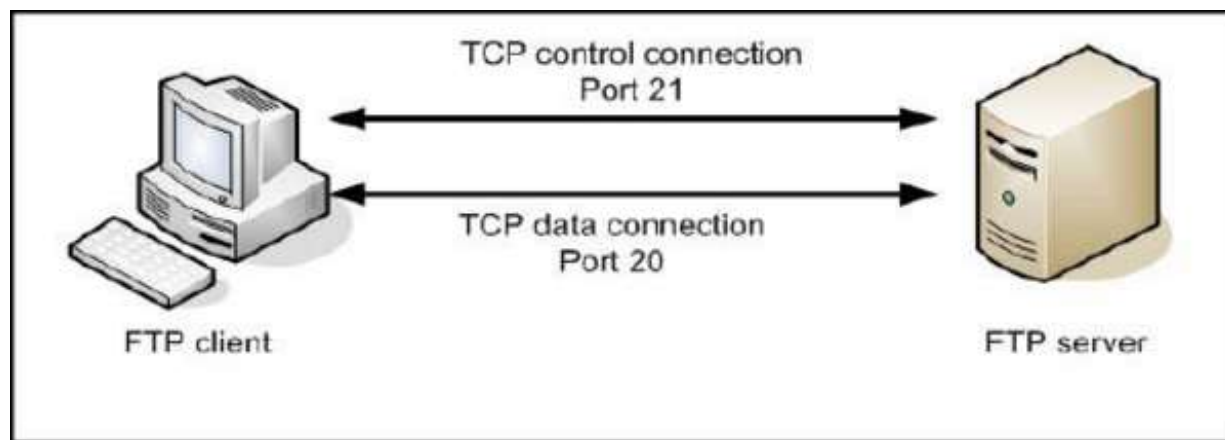
# Tìm hiểu giao thức FTP

- Được mô tả trong tài liệu [RFC959](#)
- FTP (File Transfer Protocol) là giao thức trao đổi file phổ biến.
- Hoạt động theo mô hình client-server trên nền giao thức TCP.
- Giao diện giữa client và server được cung cấp dưới dạng một tập các lệnh tương tác người dùng.

# Tìm hiểu giao thức FTP

Mô hình hoạt động: Quá trình truyền nhận dữ liệu giữa client và server được tạo nên từ 2 tiến trình:

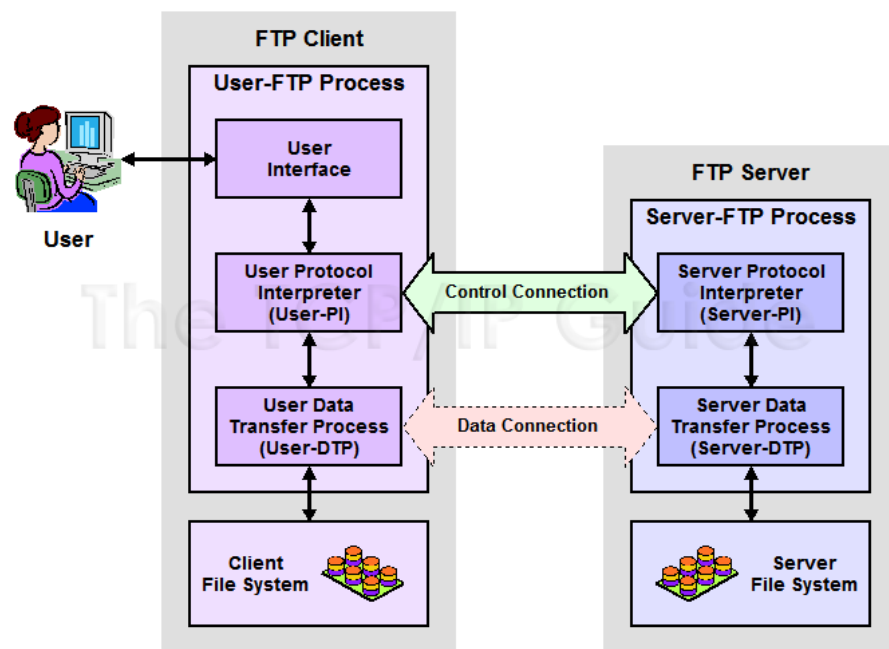
- Control connection:
  - Kết nối chính được tạo ra khi phiên làm việc được thiết lập
  - Được duy trì trong suốt phiên làm việc và chỉ cho các thông tin điều khiển đi qua ví dụ như lệnh và trả lời.
  - Không được sử dụng để gửi dữ liệu.
- Data connection:
  - Mỗi khi dữ liệu được gửi từ sever tới client hoặc ngược lại, một kết nối dữ liệu được thiết lập. Dữ liệu được truyền qua kết nối này. Khi hoàn tất việc truyền dữ liệu, kết nối được hủy bỏ.



# Tìm hiểu giao thức FTP

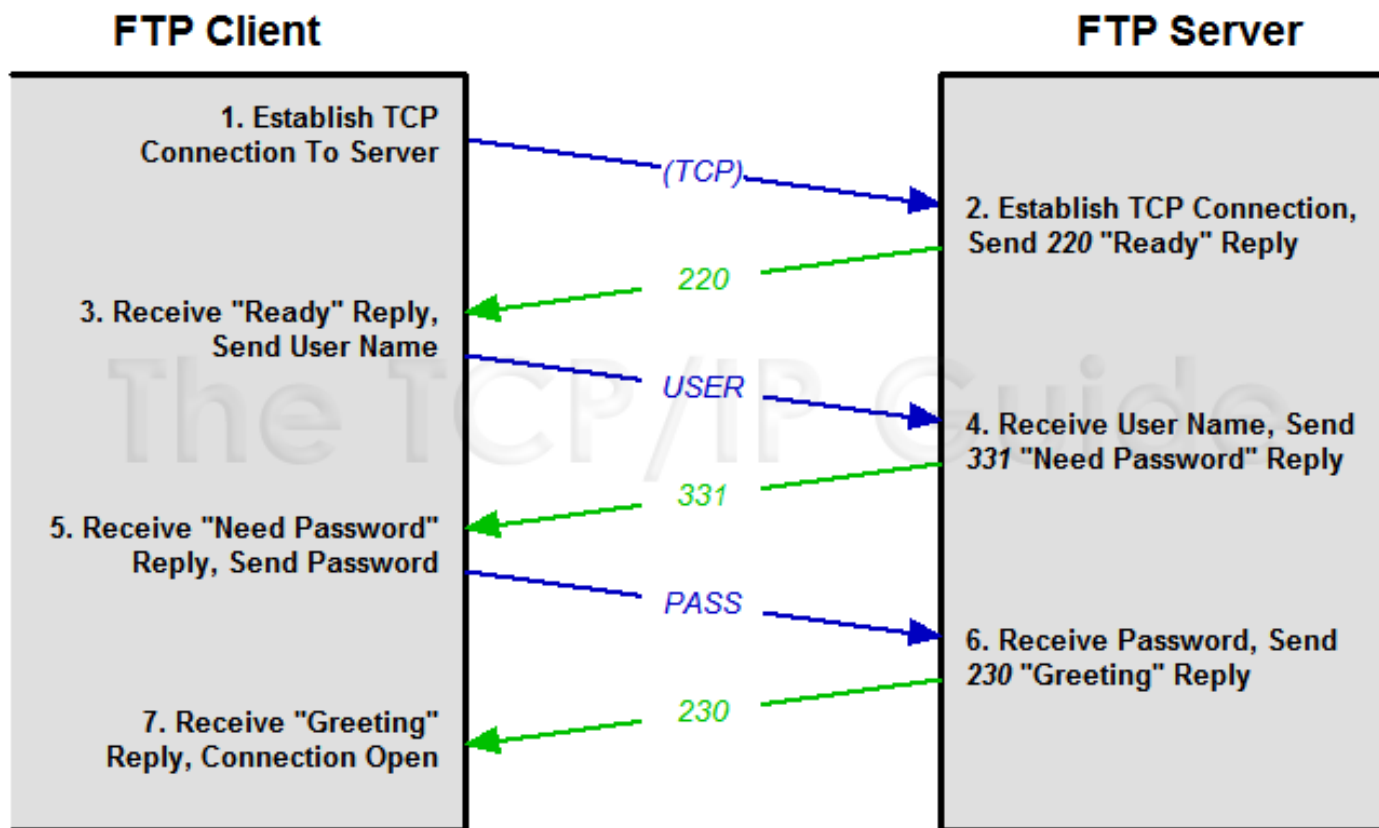
Mô hình hoạt động: mô hình FTP chia phần mềm trên mỗi thiết bị thành 2 thành phần giao thức logic chịu trách nhiệm cho mỗi kênh:

- **Protocol interpreter (PI):** chịu trách nhiệm quản lý kênh điều khiển, phát và nhận lệnh và trả lời.
- **Data transfer process (DTP):** chịu trách nhiệm gửi và nhận dữ liệu giữa client và server.



# Tìm hiểu giao thức FTP

- Trình tự truy cập và chứng thực FTP: client cung cấp username/password để đăng nhập.





## Quản lý kênh dữ liệu:

- Mỗi khi cần phải truyền dữ liệu giữa các server và client, một kênh dữ liệu cần phải được tạo ra.
- Kênh dữ liệu kết nối bộ phận User-DTP và Server-DTP, sử dụng để truyền file trực tiếp (gửi hoặc nhận một file) hoặc truyền dữ liệu ngầm, như là yêu cầu một danh sách file trong thư mục nào đó trên server.
- Hai phương thức được sử dụng để tạo ra kênh dữ liệu: phía client hay phía server là phía đưa ra yêu cầu khởi tạo kết nối.

# Tìm hiểu giao thức FTP

- Quản lý kênh dữ liệu: Chế độ chủ động

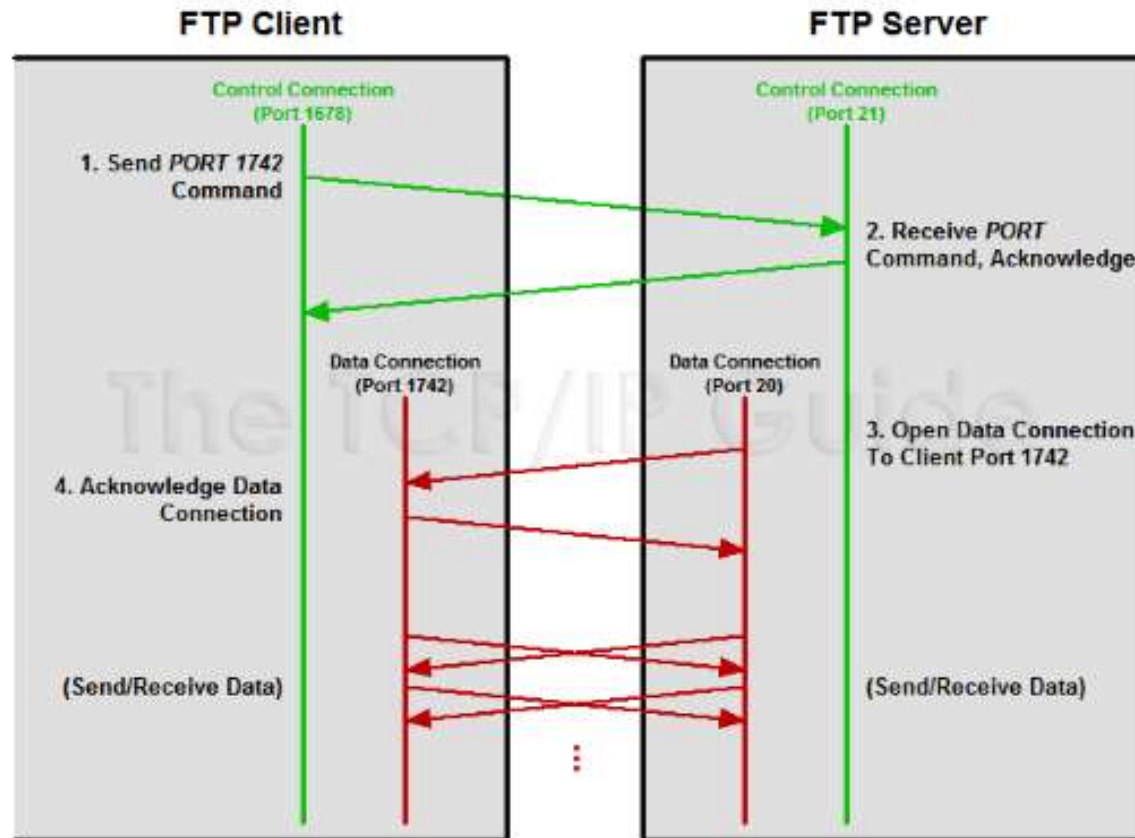


Figure 290: FTP Active Data Connection

# Tìm hiểu giao thức FTP

- Quản lý kênh dữ liệu: Chế độ bị động

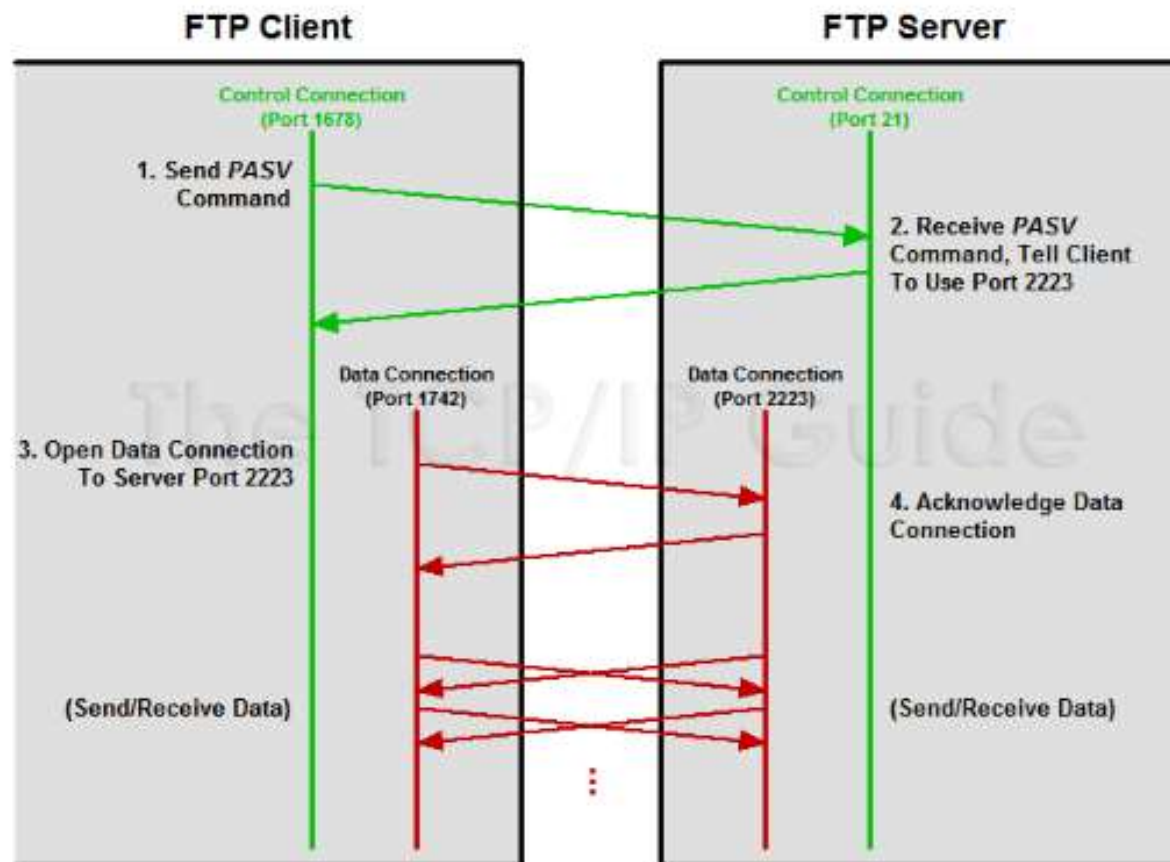


Figure 291: FTP Passive Data Connection

# Tìm hiểu giao thức FTP

- Một số lệnh FTP thường dùng:

Lệnh + tham số	Ý nghĩa
USER username	Gửi định danh người dùng đến server
PASS password	Gửi mật khẩu người dùng đến server
LIST	Hiển thị danh sách tập tin trong thư mục hiện tại
RETR filename	Tải file từ server về client
STOR filename	Tải file từ client đến server
RNFR remote-filename	Xác định file sẽ được đổi tên
RNTO remote-filename	Đổi tên file sang tên mới (sau lệnh RNFR)
DELE remote-filename	Xóa tập tin

# Tìm hiểu giao thức FTP

- Một số lệnh FTP thường dùng:

Lệnh + tham số	Ý nghĩa
MKD remote-directory	Tạo thư mục mới
RMD remote-directory	Xóa thư mục
PWD	In ra tên thư mục hiện tại
CWD remote-directory	Di chuyển đến thư mục khác
TYPE type-character	Thiết lập kiểu dữ liệu (A: ký tự, I: nhị phân)
PASV	Chuyển sang chế độ Passive
QUIT	Ngắt kết nối

Lập trình ứng dụng máy khách FTP với các chức năng cơ bản:

- Cung cấp giao diện cơ bản (người dùng không trực tiếp nhập các lệnh FTP)
- Đăng nhập
- Hiển thị nội dung thư mục
- Tạo, đổi tên, xóa thư mục
- Đổi tên, xóa file
- Client tải file lên server (upload)
- Client tải file từ server (download)

## c. Giao thức POP3

- Tìm hiểu về giao thức POP3
- Lập trình ứng dụng máy khách POP3

# Tìm hiểu giao thức POP3

- Post Office Protocol phiên bản 3.
- Được mô tả trong tài liệu [RFC1939](#)
- Giao thức ở tầng ứng dụng được sử dụng để lấy thư điện tử từ server mail, thông qua kết nối TCP/IP.
- Giao diện giữa client và server được cung cấp dưới dạng một tập các lệnh tương tác người dùng.



# Tìm hiểu giao thức POP3

- Post Office Protocol phiên bản 3.
- Được mô tả trong tài liệu [RFC1939](#)
- Giao thức ở tầng ứng dụng được sử dụng để lấy thư điện tử từ server mail, thông qua kết nối TCP/IP.
- Thường hoạt động ở cổng 110.
- Các trạng thái hoạt động trên server với mỗi kết nối:
  - AUTHORIZATION
  - TRANSACTION
  - UPDATE

# Tìm hiểu giao thức POP3

- Một số lệnh POP3 thường dùng:

Trạng thái	Lệnh + tham số	Ý nghĩa
AUTHORIZATION	USER username	Gửi định danh người dùng đến server
	PASS password	Gửi mật khẩu người dùng đến server
TRANSACTION	STAT	Hiển thị thông tin hộp thư (số thư + kích thước)
	LIST [msg]	Liệt kê các thư và kích thước
	RETR msg	Hiển thị nội dung thư
	DELE msg	Đánh dấu thư sẽ bị xóa
	NOOP	Giữ trạng thái kết nối
	RSET	Khôi phục trạng thái đánh dấu thư bị xóa
UPDATE	QUIT	Xóa các thư đã đánh dấu, ngắt kết nối

Lập trình ứng dụng máy khách POP3 với các chức năng cơ bản:

- Cung cấp giao diện cơ bản (người dùng không trực tiếp nhập các lệnh POP3)
- Đăng nhập
- Hiển thị danh sách email
- Hiển thị nội dung email

## 4.3. Thiết kế giao thức tự định nghĩa

- Việc thiết kế giao thức cần hình dung cách server và client giao tiếp với nhau: client sẽ gửi yêu cầu như thế nào và server sẽ trả lời ra sao.
- Những vấn đề cần quan tâm khi thiết kế giao thức tự định nghĩa:
  - Định dạng thông điệp ở mức text hoặc binary?
  - Server hoạt động bị động hay chủ động?
  - Server xử lý từng yêu cầu hoặc nhiều yêu cầu từ client?
  - Giao thức có làm việc theo phiên hay không?
  - Giao thức được lưu dưới dạng text hoặc binary?
  - Việc xác thực người dùng được thực hiện như thế nào?
  - Giao thức có được bảo mật hay không?

# Thiết kế giao thức ứng dụng Chat server

- Định dạng thông điệp ở mức text.
- Server hoạt động chủ động.
- Server xử lý lần lượt các yêu cầu client.
- Giao thức làm việc theo phiên.
- Giao thức được lưu dưới dạng text.
- Không xác thực người dùng.
- Giao thức không thực hiện bảo mật.

# Thiết kế giao thức ứng dụng Chat server

- Giao thức gồm các lệnh được gửi từ client lên server, phản hồi từ server về client và các lệnh server chủ động gửi cho các client.
- Tham khảo tài liệu “The definitive guide to linux network programming” mục **Out chat protocol**.

# Các lệnh gửi từ client

Lệnh	Mô tả
JOIN	Client tham gia phòng chat bằng nickname.
MSG	Gửi tin nhắn cho cả phòng chat.
PMSG	Gửi tin nhắn cá nhân cho một người dùng.
OP	Chuyển quyền chủ phòng chat cho người dùng khác. Thực hiện bởi chủ phòng chat.
KICK	Đưa người dùng ra khỏi phòng chat. Thực hiện bởi chủ phòng chat.
TOPIC	Thiết lập chủ đề cho phòng chat. Thực hiện bởi chủ phòng chat.
QUIT	Người dùng thoát khỏi phòng chat.

# Các lệnh và phản hồi từ server

- Lệnh **JOIN <NICKNAME>**
- **NICKNAME** chỉ chứa các ký tự chữ thường và chữ số, không trùng với những người dùng đã tham gia.
- Phản hồi từ server
  - 100 OK
  - 200 NICKNAME IN USE
  - 201 INVALID NICK NAME
  - 999 UNKNOWN ERROR



# Các lệnh và phản hồi từ server

- Lệnh **MSG <ROOM MESSAGE>**
- ROOM MESSAGE là chuỗi ký tự bất kỳ kết thúc bởi dấu xuống dòng.
- Phản hồi từ server
  - 100 OK
  - 999 UNKNOWN ERROR

# Các lệnh và phản hồi từ server

- Lệnh **PMSG <NICKNAME> <MESSAGE>**
- **NICKNAME** của người nhận
- **MESSAGE** thông điệp cần gửi
- Phản hồi từ server
  - 100 OK
  - 202 UNKNOWN NICKNAME
  - 999 UNKNOWN ERROR

# Các lệnh và phản hồi từ server

- Lệnh **OP <NICKNAME>**
- NICKNAME của người được chuyển quyền.
- Được thực hiện bởi chủ phòng chat hiện tại.
- Phản hồi từ server
  - 100 OK
  - 202 UNKNOWN NICKNAME
  - 203 DENIED
  - 999 UNKNOWN ERROR

# Các lệnh và phản hồi từ server

- Lệnh **KICK <NICKNAME>**
- **NICKNAME** của người bị đưa ra khỏi phòng chat.
- Được thực hiện bởi chủ phòng chat hiện tại.
- Phản hồi từ server
  - 100 OK
  - 202 UNKNOWN NICKNAME
  - 203 DENIED
  - 999 UNKNOWN ERROR

# Các lệnh và phản hồi từ server

- Lệnh **TOPIC <TOPIC NAME>**
- TOPIC NAME chủ đề phòng chat, có thể chứa dấu cách.
- Được thực hiện bởi chủ phòng chat hiện tại.
- Phản hồi từ server
  - 100 OK
  - 203 DENIED
  - 999 UNKNOWN ERROR

# Các lệnh và phản hồi từ server

- Lệnh **QUIT**
- Phản hồi từ server
  - 100 OK
  - 999 UNKNOWN ERROR

# Các thông điệp gửi từ server

Lệnh	Mô tả
JOIN <NICKNAME>	Có người dùng tham gia phòng chat.
MSG <NICKNAME> <ROOM MESSAGE>	Có người dùng nhắn tin cho cả phòng chat.
PMSG <NICKNAME> <MESSAGE>	Có người dùng gửi tin nhắn cá nhân.
OP <NICKNAME>	Được chuyển quyền chủ phòng từ người dùng khác.
KICK <KICKED NICKNAME> <OP NICKNAME>	Có người dùng bị đưa ra khỏi phòng chat bởi chủ phòng chat.
TOPIC <OP NICKNAME> <TOPIC>	Chủ đề phòng chat được thay đổi bởi chủ phòng chat.
QUIT <NICKNAME>	Có người dùng thoát khỏi phòng chat.

# Ví dụ phiên đăng nhập

**JOIN madchatter**

100 OK

TOPIC topdog Linux Network Programming

JOIN topdog

OP topdog

JOIN lovetochat

JOIN linuxlover

JOIN borntocode

**MSG hi all!**

100 OK

MSG topdog welcome madchatter.

OP madchatter

MSG linuxlover hi mad!

MSG lovetochat hi!

MSG borntocode sup chatter.

PMSG lovetochat linuxlover is getting a bit too obnoxious and topdog won't do anything.

**PMSG lovetochat no prob. i'll take care of it.**

100 OK

**KICK linuxlover**

100 OK

**PMSG lovetochat he's gone.**

100 OK

PMSG lovetochat thanks!

**QUIT**

100 OK





- Lập trình ứng dụng **chat\_server** và **chat\_client** theo giao thức trên.
- Thiết kế giao thức và lập trình ứng dụng game server:
  - Cờ caro
  - Cờ vua

# Chương 5. Lập trình socket nâng cao

# Chương 5. Lập trình socket nâng cao

5.1. Thư viện OpenSSL

5.2. Raw socket



## 5.1. Thư viện OpenSSL

- a. Giới thiệu
- b. Sử dụng công cụ Command Line
- c. Cách cài đặt vào dự án
- d. Một số ví dụ

## a. Giới thiệu

- OpenSSL là bộ công cụ mạnh và đầy đủ tính năng phục vụ cho giao thức TLS (Transport Layer Security) và SSL (Secure Sockets Layer).
- OpenSSL cung cấp thư viện bảo mật đa mục đích.
- Mã nguồn được tải miễn phí từ địa chỉ:

<https://www.openssl.org/source/>

- Cài đặt công cụ và thư viện OpenSSL trong môi trường Ubuntu:

B1. Cập nhật gói phần mềm

**sudo apt update**

B2. Cài đặt gói phần mềm libssl-dev

**sudo apt install libssl-dev**

B3. Kiểm tra kết quả cài đặt

**openssl version**

## b. Sử dụng công cụ Command Line

- Thực hiện kết nối qua giao thức HTTPS

**openssl s\_client vnexpress.net:443**

=> sau khi kết nối thành công, sử dụng phương thức GET để nhận dữ liệu.

## c. Cách cài đặt vào dự án

- Cài đặt thư viện **libssl-dev**
- Thêm tệp tiêu đề **openssl/ssl.h**
- Dịch chương trình với tham số **-lssl**

# Khởi tạo thư viện

```
SSL_library_init();           // Khởi tạo thư viện OpenSSL
const SSL_METHOD *meth = TLS_client_method(); // Khai báo phương
thức mã hóa TLS
SSL_CTX *ctx = SSL_CTX_new(meth); // Tạo context mới
SSL *ssl = SSL_new(ctx);        // Tạo con trỏ ssl
if (!ssl) {
    printf("Error creating SSL.\n");
    return -1;
}
SSL_set_fd(ssl, client);       // Gắn con trỏ ssl với socket
int err = SSL_connect(ssl);    // Tạo kết nối ssl
if (err <= 0) {
    printf("Error creating SSL connection.  err=%x\n", err);
    fflush(stdout);
    return -1;
}
```



# Truyền nhận dữ liệu thông qua SSL

- Sau bước khởi tạo, con trỏ ssl được sử dụng để truyền nhận dữ liệu mã hóa
- Nhận dữ liệu:

**int SSL\_read(SSL \*ssl, void \*buf, int num)**

- ssl: con trỏ ssl đã khởi tạo
- buf: buffer nhận dữ liệu
- num: số byte muốn nhận

=> Hàm trả về số byte nhận được trong trường hợp thành công

- Truyền dữ liệu:

**int SSL\_write(SSL \*ssl, const void \*buf, int num)**

- ssl: con trỏ ssl đã khởi tạo
- buf: buffer chứa dữ liệu muốn truyền
- num: số byte cần truyền

=> Hàm trả về số byte truyền được trong trường hợp thành công

# Giải phóng con trỏ ssl

SSL\_shutdown(*ssl*) => Đóng con trỏ ssl

SSL\_free(*ssl*) => Giải phóng con trỏ ssl

SSL\_CTX\_free(*ssl\_context*) => Giải phóng con trỏ ngữ cảnh

## d. Một số ví dụ

- Kết nối và nhận dữ liệu từ website HTTPS.
- Kết nối đến Gmail server thông qua giao thức POP3.

## 5.2. Raw socket

- Raw socket cho phép truy nhập vào các giao thức ở tầng giao vận (Transport Protocol).
- Raw socket có thể được sử dụng để tạo ra những tiện ích như ứng dụng ping, sniffer
- Cần có hiểu biết cơ bản về những giao thức như ICMP, TCP, ...

- Khởi tạo raw sockets:
  - Sử dụng hàm `socket()`
  - Tạo raw socket để bắt các gói tin IP:

```
int s1 = socket(AF_INET, SOCK_RAW, IPPROTO_IP);  
if (s1 == -1) {  
    printf("Failed to create socket\n");  
    return 1;  
}
```

- Tạo raw socket truyền gói tin ICMP:

```
int s2 = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

# Raw socket - Ứng dụng bắt gói tin

Các bước thực hiện:

- Tạo raw socket để bắt các gói tin TCP/IP
- Sử dụng hàm `recvfrom()` để nhận các gói tin.
- Mỗi lần gọi hàm `recvfrom()` với độ lớn buffer bằng chiều dài tối đa của gói tin TCP/IP (65535 bytes)
- Phân tích gói tin

=> Cần chạy chương trình với quyền super user (sudo) để có thể tạo raw socket.

=> Ứng dụng hoạt động trực tiếp trên hệ điều hành Ubuntu (không hoạt động được với WSL và MacOS).

# VD1 - Ứng dụng sniffer

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>

int main() {
    int sock_raw = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if (sock_raw < 0) {
        printf("Failed to create socket: %d\n", errno);
        return 1;
    }

    unsigned char *buffer = (unsigned char *)malloc(65535);
    int saddr_size, data_size;
    struct sockaddr saddr;
```

# VD1 - Ứng dụng sniffer

```
while (1) {
    data_size = recvfrom(sock_raw, buffer, 65535, 0, &saddr, &saddr_size);
    if (data_size < 0) {
        printf("recvfrom error, failed to get packets\n");
        return 1;
    }

    // In header gói tin
    printf("Data size: %d\n", data_size);
    for (int i = 0; i < 40; i++)
        printf("%x ", buffer[i]);
    printf("\n");

    // Phân tích nội dung gói tin ...
}

close(sock_raw);
return 0;
}
```



- Chỉnh sửa ví dụ 1 để có thể
    - Phân tích nội dung header, hiển thị địa chỉ IP nguồn và đích
    - Phân tích nội dung body, kiểm tra xem có phải là lệnh GET hoặc lệnh POST hay không?
- => Sử dụng trình duyệt để truy nhập trang sử dụng giao thức HTTP không bảo mật.

<http://httpbin.org/get>

<http://httpbin.org/post>

# Phụ lục

# Phụ lục 1 – Thư viện ncurses

- Là thư viện được sử dụng để tạo giao diện người dùng ở dạng văn bản (Text-based User Interface – TUI).

```
.config - Linux Kernel v2.6.32 Configuration

Linux Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >

General setup --->
[*] Enable loadable module support --->
-- Enable the block layer --->
  Processor type and features --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Executable file formats / Emulations --->
-- Networking support --->
  Device Drivers --->
  Firmware Drivers --->
  File systems --->

v(+)

<Select>  < Exit >  < Help >
```

- Được sử dụng trong phạm vi môn học để tạo giao diện client hoặc server đơn giản.
- Tham khảo tại <https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/intro.html>
- Cách cài đặt  
`sudo apt-get install libncurses5-dev libncursesw5-dev`
- Dịch chương trình với cờ `-lncurses`
- Các hàm cơ bản:
  - `initscr()` – chuyển sang chế độ curse
  - `endwin()` – kết thúc chế độ curse

# Các hàm với cửa sổ

`WINDOW * newwin(height, width, y, x)` => tạo cửa sổ mới với kích thước width x height tại vị trí (x, y)

`int wborder (WINDOW *, ch, ch, ch, ch, ch, ch, ch, ch)` => vẽ khung cho cửa sổ với các ký tự khung trái, phải, trên dưới và 4 góc

`wgetstr(WINDOW *, char *)` => đọc chuỗi ký tự từ bàn phím trong phạm vi cửa sổ

`int wprintw (WINDOW *, const char *,...)` => in dữ liệu có định dạng ra cửa sổ

`int wclear(WINDOW *)` => xóa nội dung cửa sổ

`int scrollok (WINDOW *, bool)` => cho phép cuộn nội dung cửa sổ

`int wrefresh(WINDOW *)` => cập nhật nội dung cửa sổ ra màn hình console

`delwin(WINDOW *)` => xóa cửa sổ

# Ví dụ - Tạo cửa sổ nhập, xuất dữ liệu (1/3)

```
#include <curses.h>
#include <string.h>

int main() {
    initscr();

    // Tạo cửa sổ để vẽ khung
    WINDOW *input_border = newwin(3, 50, 0, 0);
    wborder(input_border, '|', '|', '-', '-', '+', '+', '+', '+');
    wrefresh(input_border);

    WINDOW *log_border = newwin(16, 50, 3, 0);
    wborder(log_border, '|', '|', '-', '-', '+', '+', '+', '+');
    wrefresh(log_border);
}
```

# Ví dụ - Tạo cửa sổ nhập, xuất dữ liệu (2/3)

```
// Tạo cửa sổ nhập dữ liệu và cửa sổ ghi dữ liệu
WINDOW *input_win = newwin(1, 48, 1, 1);
wrefresh(input_win);
WINDOW *log_win = newwin(14, 48, 4, 1);
scrollok(log_win, TRUE);
wrefresh(log_win);

char str[256];
while (1) {
    // Xóa nội dung cửa sổ nhập liệu
    wclear(input_win);
    wrefresh(input_win);
    // Nhập chuỗi ký tự từ bàn phím
    wgetstr(input_win, str);

    if (strcmp(str, "exit") == 0) break;
```

# Ví dụ - Tạo cửa sổ nhập, xuất dữ liệu (3/3)

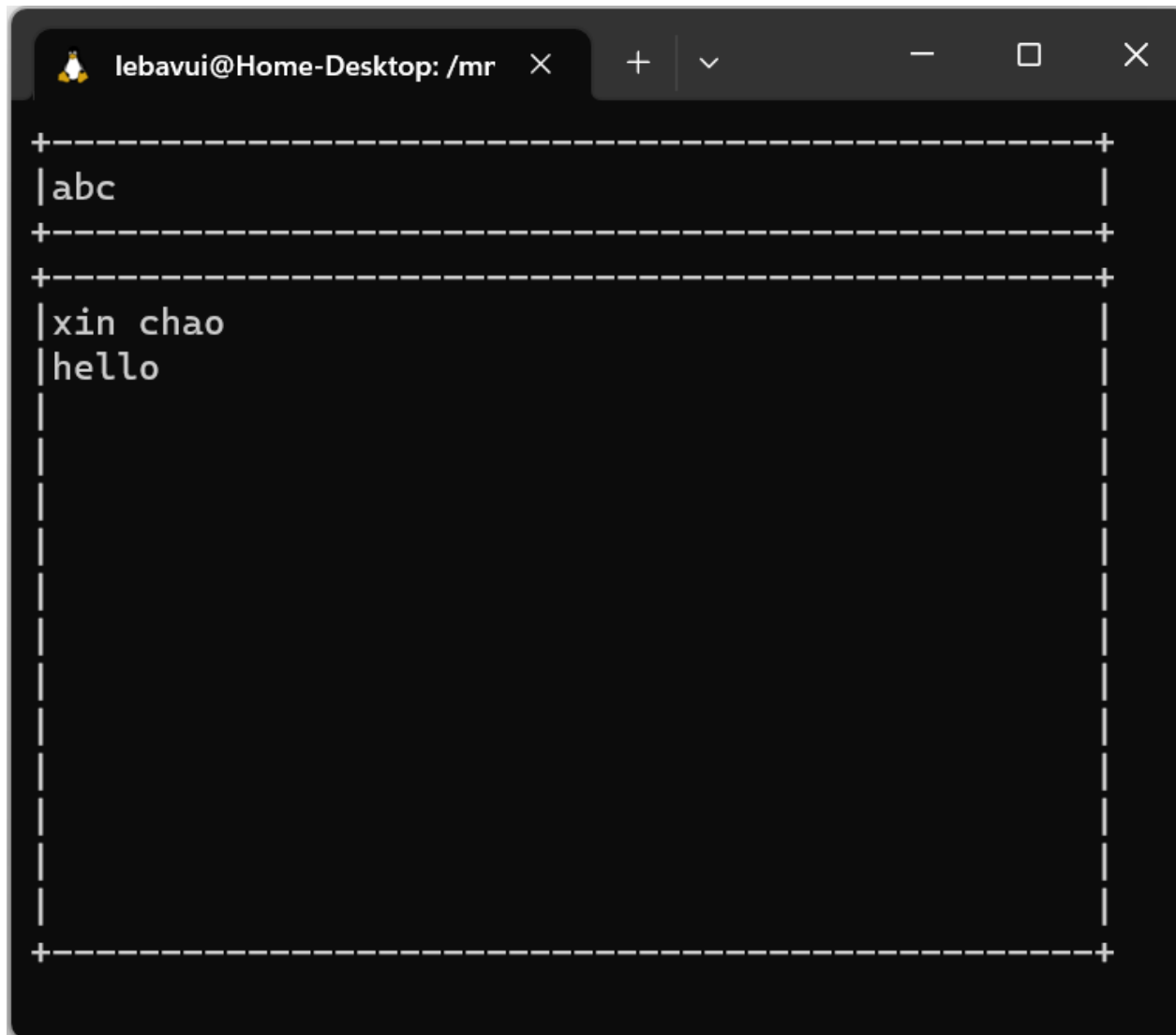
```
// Ghi chuỗi ký tự vào cửa sổ
wprintw(log_win, "%s\n", str);
wrefresh(log_win);
}

// Xóa các con trỏ cửa sổ
delwin(input_border);
delwin(log_border);
delwin(input_win);
delwin(log_win);

endwin();
return 0;
}
```



# Kết quả



```
lebavui@Home-Desktop: /mr × + ▾ − □ ×
+-----+
|abc|
+-----+
+-----+
|xin chao|
|hello|
+-----+
```

- Cập nhật ứng dụng server sử dụng giao diện TUI
  - Hiển thị dữ liệu nhận được từ client
  - Hiển thị danh sách client đã đăng nhập thành công

The background is a solid red color. A large, faint arch made of small red dots spans the top half of the image, framing the central text.

**THE END.**