

**ĐẠI HỌC BÁCH KHOA HÀ NỘI  
TRƯỜNG CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG**

**BÀI GIẢNG HỌC PHẦN  
LẬP TRÌNH MẠNG**

**Mã học phần: IT4060**

Tác giả: Nhóm chuyên môn giảng dạy

# Lời nói đầu

---

Môn học Lập trình mạng sử dụng thư viện Socket API trong môi trường Unix được thiết kế dành cho những người muốn tìm hiểu và phát triển các chương trình có khả năng giao tiếp với nhau thông qua cơ chế socket – một giao diện lập trình ứng dụng (API) đã trở thành chuẩn mực trong lĩnh vực lập trình mạng.

Khóa học hướng tới nhiều đối tượng: từ những sinh viên, lập trình viên mới bắt đầu làm quen với socket cho đến các kỹ sư phần mềm chuyên nghiệp muốn mở rộng hoặc duy trì các ứng dụng mạng hiện có. Không chỉ dừng lại ở việc viết mã, môn học còn giúp người học hiểu rõ cách các thành phần mạng hoạt động trong hệ điều hành, từ đó có thể vận dụng để xây dựng các ứng dụng mạng mới hoặc tối ưu hóa hệ thống sẵn có.

Ngoài việc làm quen với mô hình client-server và xây dựng các ứng dụng mạng cơ bản, môn học còn giới thiệu cách lập trình với những giao thức mạng chuẩn như HTTP, FTP – vốn là nền tảng cho các dịch vụ Internet phổ biến hiện nay. Đồng thời, người học cũng sẽ được hướng dẫn cách thiết kế và cài đặt các giao thức tự định nghĩa, phục vụ cho các ứng dụng chuyên biệt.

Các ví dụ và bài tập thực hành đều được triển khai bằng Unix – môi trường điển hình hỗ trợ mạnh mẽ Socket API. Tuy nhiên, hầu hết các khái niệm và mã nguồn đều mang tính hệ điều hành độc lập, có thể áp dụng trên nhiều nền tảng khác nhau. Trong suốt khóa học, người học sẽ xây dựng và thử nghiệm nhiều chương trình mạng thực tế, từ đó nắm vững nguyên lý hoạt động của các ứng dụng mạng phổ biến như trình duyệt web, email client hay máy chủ chia sẻ tập tin.

Bài giảng bao gồm 5 chương:

Chương 1. Giới thiệu về Lập trình mạng

Chương 2. Lập trình socket cơ bản

Chương 3. Các kiến trúc server - client

Chương 4. Thiết kế và lập trình giao thức mạng

Chương 5. Lập trình socket nâng cao

Với cấu trúc nội dung chặt chẽ và nhiều ví dụ thực hành, môn học sẽ giúp người

học xây dựng nền tảng vững chắc về lập trình mạng, từ các kỹ thuật socket cơ bản cho đến thiết kế giao thức và lập trình nâng cao. Sau khi hoàn thành, người học sẽ có khả năng phát triển các ứng dụng mạng hiệu quả, hiểu rõ cách các dịch vụ Internet vận hành, đồng thời tự tin ứng dụng kiến thức vào các dự án thực tế cũng như nghiên cứu chuyên sâu trong lĩnh vực hệ thống và mạng máy tính.

Hà Nội, ngày 15 tháng 09 năm 2025

**Nhóm tác giả**

# Mục lục

---

Lời nói đầu . . . . .	3
Mục lục . . . . .	5
<b>Chương 1. Giới thiệu về Lập trình mạng</b>	<b>7</b>
1.1. Các khái niệm về Lập trình mạng . . . . .	7
1.1.1. Khái niệm . . . . .	7
1.1.2. Các vấn đề cần quan tâm trong Lập trình mạng . . . . .	7
1.1.3. Các ngôn ngữ lập trình được sử dụng . . . . .	8
1.1.4. Mô hình client-server . . . . .	9
1.1.5. Các kiểu ứng dụng trên nền tảng mạng máy tính . . . . .	9
1.1.6. Các thư viện và công cụ lập trình . . . . .	10
1.2. Lập trình trong môi trường Ubuntu/Linux . . . . .	11
1.2.1. Trình biên dịch gcc, trình gõ lỗi gdb . . . . .	11
1.2.2. Cài đặt môi trường lập trình . . . . .	11
1.2.3. Hướng dẫn sử dụng phần mềm Visual Studio Code . . . . .	14
<b>Chương 2. Lập trình socket cơ bản</b>	<b>15</b>
2.1. Khái niệm socket . . . . .	15
2.2. Cấu trúc địa chỉ của socket . . . . .	16
2.3. Ứng dụng server/client theo giao thức TCP . . . . .	22
2.4. Ứng dụng sender/receiver theo giao thức UDP . . . . .	41
2.5. Bài tập . . . . .	47
<b>Chương 3. Các kiến trúc server-client</b>	<b>48</b>
3.1. Các chế độ hoạt động của socket . . . . .	48
3.2. Server hoạt động theo cơ chế lặp . . . . .	53
3.3. Kỹ thuật đa tiến trình - Multiprocessing . . . . .	55

---

3.4. Kỹ thuật đa luồng - Multithreading . . . . .	71
3.5. Kỹ thuật ghép kênh - Multiplexing . . . . .	84
3.6. Bài tập . . . . .	102
<b>Chương 4. Thiết kế giao thức mạng</b>	<b>104</b>
4.1. Khái niệm về giao thức . . . . .	104
4.2. Tìm hiểu một số giao thức phổ biến . . . . .	104
4.2.1. Giao thức HTTP . . . . .	104
4.2.2. Giao thức FTP . . . . .	108
4.2.3. Giao thức POP3 . . . . .	112
4.3. Thiết kế giao thức tự định nghĩa . . . . .	113
4.4. Bài tập . . . . .	117
<b>Chương 5. Lập trình socket nâng cao</b>	<b>118</b>
5.1. Thư viện OpenSSL . . . . .	118
5.1.1. Giới thiệu thư viện OpenSSL . . . . .	118
5.1.2. Sử dụng công cụ dòng lệnh . . . . .	119
5.1.3. Cách cài đặt vào dự án . . . . .	119
5.2. Raw socket . . . . .	123
5.3. Bài tập . . . . .	125
<b>Tài liệu tham khảo</b> . . . . .	<b>126</b>

# Chương 1

## Giới thiệu về Lập trình mạng

---

1.1. Các khái niệm về Lập trình mạng . . . . .	7
1.2. Lập trình trong môi trường Ubuntu/Linux . . . . .	11

---

### 1.1. Các khái niệm về Lập trình mạng

#### 1.1.1. Khái niệm

Lập trình mạng được định nghĩa là tập hợp các kỹ thuật lập trình nhằm mục đích xây dựng các ứng dụng và phần mềm có khả năng khai thác hiệu quả tài nguyên của mạng máy tính. Điều này bao gồm việc thiết kế, phát triển và triển khai các chương trình có thể giao tiếp, trao đổi dữ liệu và tương tác qua lại trên một mạng máy tính, cho dù đó là mạng cục bộ hay Internet. Mục tiêu của môn học Lập trình mạng là cung cấp kiến thức cơ bản để:

- Xây dựng ứng dụng phía server và 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, bao gồm việc sử dụng thư viện, môi trường, tài liệu, cũng như kỹ năng thiết kế và xây dựng chương trình.

#### 1.1.2. Các vấn đề cần quan tâm trong Lập trình mạng

Để xây dựng một ứng dụng mạng hiệu quả, lập trình viên cần chú ý đến nhiều khía cạnh quan trọng:

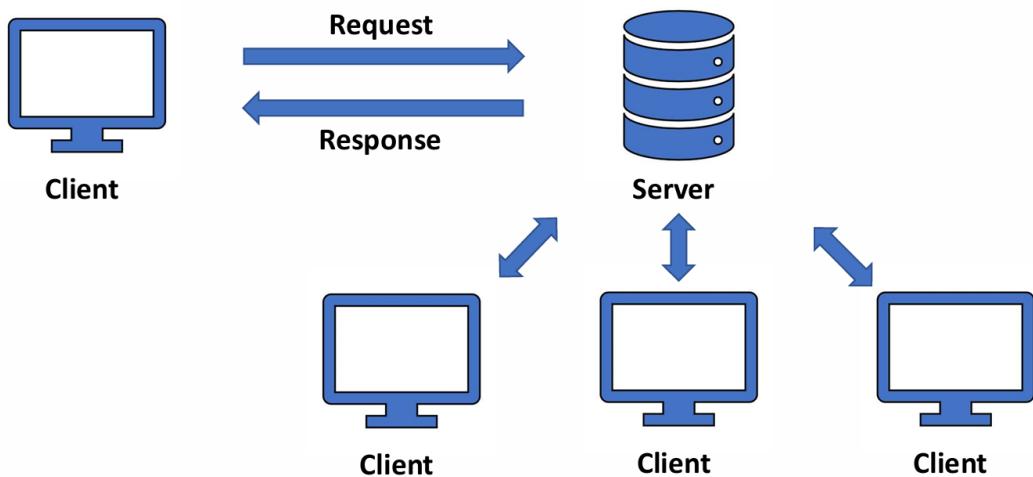
- Thông tin truyền nhận trên mạng: Đây là yếu tố cơ bản nhất, liên quan đến việc dữ liệu được gửi đi và nhận về giữa các thiết bị trên mạng.

- Các giao thức truyền thông (Protocols):
  - Giao thức chuẩn: Là các bộ quy tắc đã được định nghĩa và sử dụng rộng rãi như HTTP (HyperText Transfer Protocol) cho web, FTP (File Transfer Protocol) để truyền tải tập tin, SMTP (Simple Mail Transfer Protocol) để gửi email, hay POP3 (Post Office Protocol 3) để nhận email, DNS (Domain Name System) để phân giải tên miền, SSH (Secure Shell) cho truy cập bảo mật từ xa, IMAP (Internet Message Access Protocol) để quản lý email trên server. Việc lập trình mang thường xây dựng ứng dụng tuân theo một trong các giao thức này ở tầng ứng dụng.
  - Giao thức tự định nghĩa: Trong nhiều trường hợp, lập trình viên cần xây dựng các giao thức riêng biệt, tùy chỉnh để phù hợp với yêu cầu đặc thù của ứng dụng.
- Các kỹ thuật truyền nhận dữ liệu: Tập trung vào cách thức dữ liệu được gửi và nhận, bao gồm các cơ chế đảm bảo tính toàn vẹn, thứ tự và hiệu suất của việc truyền tải.
- Các kỹ thuật nâng cao: Để tối ưu hóa việc sử dụng tài nguyên mạng và đảm bảo an toàn thông tin, các kỹ thuật sau cần được xem xét:
  - Nén dữ liệu: Giảm kích thước dữ liệu trước khi truyền để tiết kiệm băng thông và tăng tốc độ.
  - Mã hóa dữ liệu: Bảo vệ thông tin khỏi sự truy cập trái phép bằng cách biến đổi nó thành một dạng không thể đọc được nếu không có khóa giải mã.
  - Truyền nhận dữ liệu song song: Tối ưu hóa hiệu suất bằng cách gửi và nhận nhiều phần dữ liệu cùng một lúc.

#### 1.1.3. Các ngôn ngữ lập trình được sử dụng

Các ngôn ngữ được sử dụng trong lập trình mạng rất đa dạng, mỗi ngôn ngữ có những ưu và nhược điểm riêng:

- C/C++: Được đánh giá là mạnh và phổ biến, được hầu hết các lập trình viên sử dụng để viết các ứng dụng mạng hiệu năng cao. Giáo trình này sẽ chỉ đề cập đến hai ngôn ngữ này.
- Java: Khá thông dụng, được sử dụng nhiều trong các nền tảng di động như J2ME và Android.



Hình 1.1 Mô hình client-server

- C#: Mạnh và dễ sử dụng, tuy nhiên chạy trên nền .Net Framework và chỉ hỗ trợ hệ điều hành Windows.
- Python, Perl, PHP...: Là các ngôn ngữ thông dịch, thường được sử dụng để viết các tiện ích nhỏ một cách nhanh chóng.

#### 1.1.4. Mô hình client-server

Trong mô hình client-server ở hình 1.1, Client là bên gửi yêu cầu (Request) và Server là bên phản hồi (Response) lại các yêu cầu đó. Mô hình này là nền tảng cho nhiều ứng dụng mạng, và khóa học "Lập trình mạng" nhằm cung cấp kiến thức cơ bản về lập trình ứng dụng trên môi trường mạng máy tính, bao gồm xây dựng ứng dụng phía server và xây dựng ứng dụng phía client, cũng như tìm hiểu các kiến trúc client-server.

#### 1.1.5. Các kiểu ứng dụng trên nền tảng mạng máy tính

Các kiểu ứng dụng hoạt động trên mạng rất phong phú, bao gồm:

- 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: Ví dụ như uTorrent.
- Các ứng dụng khác: Internet Download Manager, WireShark, Firewall, v.v..

Ví dụ về các ứng dụng kết nối mạng được sử dụng thường xuyên:

- Phần mềm web: Client là các ứng dụng trình duyệt như Chrome, Edge, Firefox, gửi yêu cầu đến web server, web server thực hiện yêu cầu và trả lại kết quả cho trình duyệt.
- Phần mềm nhắn tin (Facebook Messenger, Zalo, Telegram): Server quản lý dữ liệu người dùng. Client gửi các yêu cầu như đăng ký, đăng nhập, hoặc các đoạn chat đến server. Server thực hiện yêu cầu và trả lại kết quả cho client, bao gồm cả việc chuyển tiếp dữ liệu giữa các client.
- Phần mềm nghe nhạc trên thiết bị di động (Apple Music, Youtube Music, Spotify): Server quản lý dữ liệu người dùng, lưu trữ các file âm thanh, xử lý yêu cầu từ phần mềm di động và quản lý các kết nối. Phần mềm di động gửi yêu cầu và dữ liệu lên server, chờ kết quả và xử lý.
- Phần mềm đồng bộ file giữa các thiết bị (Google Drive, 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 phân tích gói tin: Bắt và phân tích các gói tin được nhận bởi trình duyệt, tách ra các liên kết quan tâm, và tải file bằng nhiều luồng song song.

#### **1.1.6. Các thư viện và công cụ lập trình**

Thư viện và công cụ lập trình được giới thiệu để hỗ trợ việc xây dựng ứng dụng mạng:

- Socket API: Là một thư viện đa nền tảng, được hỗ trợ bởi các chương trình dịch trong nhiều ngôn ngữ và hệ điều hành khác nhau, thường được sử dụng cùng với C/C++ và cho hiệu năng cao nhất.
- Các công cụ lập trình (trên nền tảng Linux/Unix):

- Trình biên dịch: gcc.
  - Trình gõ lỗi: gdb.
  - Công cụ soạn thảo mã nguồn: Visual Studio Code.
- Các công cụ hỗ trợ:
    - Wireshark: Công cụ phân tích gói tin.
    - Netcat: Công cụ tạo client/server với mục đích thử nghiệm, có thể dùng để gửi nhận dữ liệu theo các giao thức đơn giản.  
TCP server (nc -v -l -p <cổng đợi kết nối>),  
TCP client (nc -v <ip/tên miền> <cổng>),  
UDP receiver (nc -v -l -u -p <cổng đợi kết nối>),  
UDP sender (nc -v -u <ip/tên miền> <cổng>).
  - Công cụ tra cứu:
    - Trang web manpages.ubuntu.com.
    - Google/BING.
    - Stack Overflow.
    - Các công cụ AI: ChatGPT, Grok, Copilot, ...

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

### 1.2.1. Trình biên dịch gcc, trình gõ 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, ...) và tương thích với nhiều nền tảng kiến trúc máy tính. Cú pháp cơ bản để dịch chương trình:

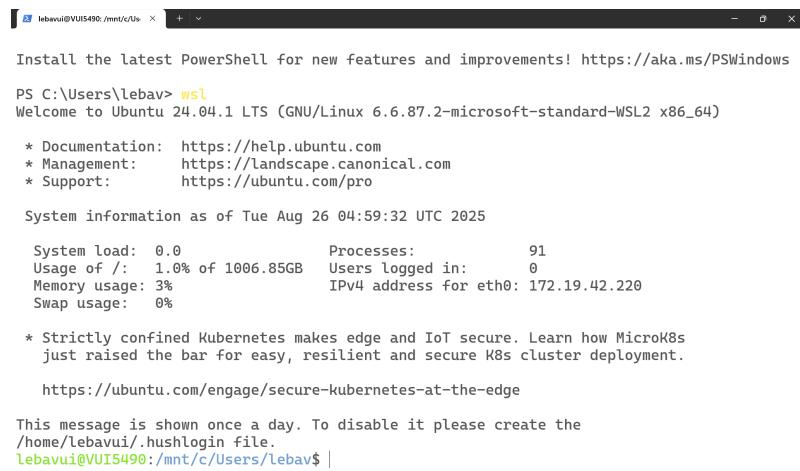
```
gcc -o [executable_name] [source_file].c
```

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

### 1.2.2. Cài đặt môi trường lập trình

#### Hệ điều hành Windows

1. Kích hoạt tính năng Windows Subsystem for Linux trong Windows Features.



```

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows
PS C:\Users\lebav> wsl
Welcome to Ubuntu 24.04.1 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

System information as of Tue Aug 26 04:59:32 UTC 2025

System load: 0.0          Processes:      91
Usage of /: 1.0% of 1006.85GB Users logged in: 0
Memory usage: 3%          IPv4 address for eth0: 172.19.42.220
Swap usage: 0%

* Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
just raised the bar for easy, resilient and secure K8s cluster deployment.

https://ubuntu.com/engage/secure-kubernetes-at-the-edge

This message is shown once a day. To disable it please create the
/home/lebavui/.hushlogin file.
lebavui@VUI5490:/mnt/c/Users/Lebav$ |

```

Hình 1.2 Truy nhập môi trường hệ điều hành Ubuntu trong cửa sổ Terminal

2. Cài đặt Windows Subsystem for Linux trong MS Store.
3. Cài đặt Ubuntu trong MS Store.
4. Cài đặt công cụ dòng lệnh Terminal trong MS Store.
5. Cài đặt công cụ soạn thảo mã nguồn Visual Studio Code trong MS Store.
6. Mở ứng dụng Terminal, nhập lệnh wsl để vào môi trường hệ điều hành Ubuntu.  
Sau đó tiếp tục cài đặt các công cụ trong môi trường Ubuntu.

### **Hệ điều hành Ubuntu**

1. Cập nhật gói phần mềm

```
sudo apt update
```

2. Cài đặt gói phần mềm biên dịch gcc

```
sudo apt install build-essential
```

3. Cài đặt gói phần mềm gõ lỗi gdb

```
sudo apt install gdb
```

4. Cài đặt công cụ tra cứu

```
sudo apt install manpages-dev
```

```

lebavui@VUI5490:/mnt/c/Users/Lebav$ gcc -v && gdb -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-linux-gnu/13/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:amdgcn-amdhsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 13.3.0-6ubuntu2-24.04' --with-bugurl=file:///usr/share/doc/gcc-13/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++,m2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-13 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/libexec --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale-gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-libstdcxx-backtrace --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system=zlib --enable-libphobos-checking=release --with-target-system=zlib=auto --enable-objc-gc=auto --enable-multilib --enable-cet --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-13-fG75Ri/gcc-13-13.3.0/debian/tmp-nvptx/usr/amdgcn-amdhsa=/build/gcc-13-fG75Ri/gcc-13-13.3.0/debian/tmp-gcn/usr --enable-offload-defaulted --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu --with-build-config=bootstrap-lto-lean --enable-link-serialization=2
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 13.3.0 (Ubuntu 13.3.0-6ubuntu2-24.04)
GNU gdb (Ubuntu 13.3.0-6ubuntu2-24.04) 13.3.0-6ubuntu2-24.04
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
lebavui@VUI5490:/mnt/c/Users/Lebav$
```

Hình 1.3 Kết quả khi kiểm tra phiên bản gcc và gdb

### 5. Kiểm tra kết quả cài đặt

```
gcc -v && gdb -v
```

Nếu máy tính sử dụng hệ điều hành Ubuntu thì có thể cài đặt thêm Visual Studio Code để soạn thảo và quản lý mã nguồn.

### Hệ điều hành MacOS

#### 1. Cài đặt công cụ homebrew

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

#### 2. Cài đặt gói phần mềm biên dịch gcc

```
brew install gcc
```

#### 3. Cài đặt gói phần mềm gõ lõi gdb

```
brew install gdb
```

#### 4. Cài đặt công cụ tra cứu

```
brew install manpages-dev
```

#### 5. Kiểm tra kết quả cài đặt

```
gcc -v && gdb -v
```

### **1.2.3. Hướng dẫn sử dụng phần mềm Visual Studio Code**

Cài đặt extension WSL. Vào môi trường Ubuntu trong Visual Studio Code bằng cách nhấn nút "Open a Remote Window" và chọn "Connect to WSL".

Trong môi trường WSL, cài đặt extension C++.

Để dịch chương trình, chọn menu Terminal, sau đó chọn Run Build Task ... Ở lần chạy đầu tiên VS Code sẽ cho phép lựa chọn công cụ dịch chương trình, hãy chọn công cụ gcc. File tasks.json sẽ được tạo để lưu lại lựa chọn này.

# Chương 2

## Lập trình socket cơ bản

---

2.1. Khái niệm socket . . . . .	15
2.2. Cấu trúc địa chỉ của socket . . . . .	16
2.3. Ứng dụng server/client theo giao thức TCP . . . . .	22
2.4. Ứng dụng sender/receiver theo giao thức UDP . . . . .	41
2.5. Bài tập . . . . .	47

---

### 2.1. Khái niệm socket

Socket là điểm cuối (end-point) trong liên kết truyền thông hai chiều (two-way communication):

Socket đại diện cho một đầu nối trong quá trình giao tiếp giữa hai ứng dụng trên mạng. Nó giống như một "cổng" mà qua đó dữ liệu được gửi và nhận. Trong mô hình client-server, socket biểu diễn kết nối giữa client (yêu cầu dịch vụ) và server (cung cấp dịch vụ).

Liên kết hai chiều: Dữ liệu có thể chảy từ client đến server và ngược lại (ví dụ: gửi yêu cầu HTTP và nhận phản hồi).

Socket được ràng buộc với một cổng (port):

Mỗi socket được gắn với một số port cụ thể (một số nguyên từ 0 đến 65535). Port giúp tầng giao vận (như TCP) định danh ứng dụng nào sẽ nhận dữ liệu.

Ví dụ: Port 80 cho HTTP, port 21 cho FTP. Port cho phép nhiều ứng dụng chạy trên cùng một máy tính mà không xung đột (ví dụ: web browser sử dụng port tạm thời để kết nối với server trên port 80).

Socket là giao diện lập trình mạng (API):

Socket là một giao diện lập trình được hỗ trợ rộng rãi trên nhiều ngôn ngữ (C/C++, Java, Python) và hệ điều hành (Linux, Windows, macOS). Nó cung cấp các hàm để tạo, kết nối, gửi/nhận dữ liệu, và đóng kết nối.

Trong C/C++, socket dựa trên thư viện `<sys/socket.h>`, cho phép lập trình

viên xây dựng ứng dụng mạng mà không cần quan tâm chi tiết phần cứng mạng.

Ứng dụng của socket:

Ở phía server: Socket được sử dụng để chờ (listen) các kết nối từ client. Server tạo socket, ràng buộc nó với địa chỉ IP và port, rồi chờ yêu cầu. Ở phía client: Socket được sử dụng để thiết lập kết nối đến server. Client tạo socket và kết nối đến địa chỉ IP/port của server.

Vai trò của Socket trong Mạng Máy Tính

Kết nối và Truyền Dữ Liệu: Socket trùu tượng hóa lớp mạng thấp hơn (như IP/TCP/UDP), cho phép lập trình viên tập trung vào logic ứng dụng. Nó xử lý việc gói dữ liệu thành gói tin và gửi qua mạng. Hỗ Trợ Các Giao Thức: Socket hỗ trợ TCP (kết nối đáng tin cậy, hướng dòng) và UDP (không kết nối, hướng thông điệp). Đa Nền Tảng: Socket là chuẩn POSIX, nên code socket trên Linux có thể chạy trên các hệ thống khác với ít thay đổi. Ví dụ Thực Tế:

Trong trình duyệt web: Socket client kết nối đến server HTTP trên port 80 để tải trang. Trong ứng dụng chat: Socket server chờ kết nối từ nhiều client, socket client gửi tin nhắn.

## 2.2. Cấu trúc địa chỉ của socket

Trong lập trình mạng, socket là điểm cuối cho truyền thông hai chiều, nhưng để socket hoạt động, nó cần được gắn với một địa chỉ mạng cụ thể. Địa chỉ này được mô tả bởi cấu trúc địa chỉ của socket, bao gồm cách lưu trữ địa chỉ IP và cổng, cũng như các cấu trúc dữ liệu liên quan trong lập trình C/C++ trên môi trường Linux. Bài giảng này sẽ trình bày chi tiết khái niệm, các cấu trúc địa chỉ (IPv4 và IPv6), các hàm chuyển đổi địa chỉ, và phân giải tên miền, kèm theo ví dụ minh họa để làm rõ cách sử dụng.

Trong thư viện lập trình socket (C/C++), ta có ba cấu trúc chính:

- struct sockaddr

Là cấu trúc chung, có thể mô tả nhiều loại địa chỉ (IPv4, IPv6 ...). Thường dùng để ép kiểu khi truyền tham số cho các hàm socket.

- struct sockaddr\_in

Dùng khi làm việc với địa chỉ IPv4 (32-bit).

- struct sockaddr\_in6

Dùng cho địa chỉ IPv6 (128-bit).

### 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.

Dưới đây là định nghĩa của các cấu trúc đã được khai báo trong thư viện:

```
#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 */
};
```

Với Linux chuẩn, sin\_len thường không tồn tại (nó là mở rộng của BSD).

Giải thích từng trường

#### **sin\_family**

Được sử dụng để xác định loại địa chỉ. Luôn gán giá trị AF\_INET khi làm việc với IPv4. Nếu là IPv6 thì dùng sockaddr\_in6 với AF\_INET6.

=> Giúp hệ điều hành biết nên xử lý theo giao thức nào.

#### **sin\_port**

Là cổng mạng (port number). 16 bit, giá trị từ 0 đến 65535. Cần chuyển về network byte order (big-endian) bằng hàm htons().

Ví dụ:

```
addr.sin_port = htons(8080); // cổng 8080
```

**sin\_addr** Chứa địa chỉ IPv4 (32 bit).

Kiểu dữ liệu: struct in\_addr

Thường được gán bằng:

inet\_addr("127.0.0.1"): chuyển chuỗi sang số.

INADDR\_ANY: 0.0.0.0 → nghĩa là “nhận kết nối từ bất kỳ IP nào”.

### **sin\_zero**

Chỉ là đệm (padding) để sockaddr\_in có kích thước bằng sockaddr. Không dùng, thường được set = 0 bằng memset.

Ví dụ khai báo địa chỉ trong ứng dụng server, server sẽ chờ các kết nối tại cổng 9090:

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

Ví dụ khai báo địa chỉ trong ứng dụng client, client sẽ kết nối đến server ở địa chỉ loopback 127.0.0.1 tại cổng 9090:

```
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 kết 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**

Để sử dụng các hàm này, cần khai báo tệp <arpa/inet.h>

Chuyển đổi địa chỉ IP từ dạng chuỗi ký tự sang số nguyên 32 bit (IPv4).

```
in_addr_t inet_addr (
    const char *cp // xâu ký tự chứa địa chỉ IPv4
) => Hồi trả về địa chỉ dạng số nguyên, -1 nếu gặp lỗi
```

Chuyển đổi địa chỉ IP dạng chuỗi ký tự sang cấu trúc in\_addr

```
int inet_aton (
    const char *cp, // Xâu ký tự chứa địa chỉ IP
    struct in_addr *inp // Cấu trúc địa chỉ IP
) => Hồi trả về 1 nếu thành công, 0 nếu gặp lỗi
```

Chuyển đổi địa chỉ từ dạng in\_addr sang dạng chuỗi ký tự (IPv4)

```
char *inet_ntoa (
    struct in_addr in // Cấu trúc địa chỉ IPv4
) => Hồi trả về chuỗi ký tự chứa địa chỉ
```

Chuyển đổi từ dạng số nguyên sang dạng chuỗi ký tự (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
) => hồi trả về xâu ký tự chứa địa chỉ, trả về NULL nếu gặp lỗi
```

Chuyển đổi từ dạng chuỗi ký tự sang dạng số nguyên (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
) => Hồi trả về 1 nếu thành công, 0 nếu xâu ký tự không hợp lệ, -1 nếu gặp lỗi khác
```

### Phân giải tên miền

Thông thường, địa chỉ của máy đích được cho dưới dạng tên miền, vì dễ nhớ hơn địa chỉ IP đối với con người. Do vậy ứng dụng cần thực hiện phân giải tên miền để có địa chỉ IP thích hợp làm tham số cho các hàm kết nối. Hàm getaddrinfo() sử dụng để phân giải tên miền ra các địa chỉ IP. Để sử dụng hàm này, cần thêm tệp tiêu đề netdb.h

Giao diện hàm

```

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 của hàm:

- Nếu thành công, hàm trả về 0.
- Nếu thất bại, hàm trả về mã lỗi, sử dụng hàm gal\_strerror() để in ra thông báo lỗi.

Để giải phóng cấu trúc chứa kết quả, sử dụng hàm freeaddrinfo().

Cấu trúc addrinfo là một cấu trúc danh sách liên kết đơn chứa thông tin về tên miền tương ứng. Định nghĩa cấu trúc như sau:

```

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 thứ 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
};

```

Trong đó, các trường thường được sử dụng như sau:

- Trường ai\_family cho biết địa chỉ IP thuộc họ IPv4 (AF\_INET) hay IPv6 (AF\_INET6).
- Trường ai\_addr và ai\_addrlen được sử dụng để xác định giá trị của địa chỉ IP. Cần phải sử dụng các hàm chuyển đổi để thu được địa chỉ IP dạng chuỗi ký tự.
- Trường ai\_next trỏ đến phần tử tiếp theo trong danh sách liên kết.

Ví dụ mã nguồn chương trình phân giải tên miền được nhập từ tham số dòng lệnh.

```
1 #include <stdio.h>
2 #include <netdb.h>
3 #include <string.h>
4 #include <arpa/inet.h>
5
6 // Ví dụ về phân giải tên miền thành địa chỉ IP
7 // Tham số nhập vào từ dòng lệnh là tên miền cần phân giải
8
9 int main(int argc, char* argv[]) {
10
11     // Kiểm tra tham số có được nhập vào không
12     if (argc != 2)
13     {
14         printf("Tham so khong hop le.\n");
15         return 1;
16     }
17
18     // Khai báo con trỏ kết quả
19     struct addrinfo *res, *p;
20
21     int ret = getaddrinfo(argv[1], "http", NULL, &res);
22     if (ret != 0 || res == NULL)
23     {
24         printf("Failed to get IP.\n");
25         return 1;
26     }
27
28     // Duyệt danh sách kết quả và in ra địa chỉ IP
29     p = res;
30     while (p != NULL) {
31         if (p->ai_family == AF_INET)           // IPv4
32         {
33             printf("IPv4\n");
34             struct sockaddr_in addr;
35             memcpy(&addr, p->ai_addr, p->ai_addrlen);
36             printf("IP: %s\n", inet_ntoa(addr.sin_addr));
37         }
38         else if (p->ai_family == AF_INET6)    // IPv6
39         {
40             printf("IPv6\n");
41             char buf[64];
```

```

42         struct sockaddr_in6 addr6;
43         memcpy(&addr6, p->ai_addr, p->ai_addrlen);
44         printf("IP: %s\n", inet_ntop(p->ai_family, &addr6.sin6_addr, buf,
45             sizeof(buf)));
46     }
47     p = p->ai_next;
48 }
49 // Giải phóng con trỏ kết quả
50 freeaddrinfo(res);
51
52 return 0;
53 }
```

Giải thích mã nguồn chương trình:

Kết quả chạy chương trình như sau:

The screenshot shows a terminal window with the following content:

```

lebavui@Home-Desktop:/mnt/d/Workspaces/socket_tutorials$ ./domain2ip gmail.com
IPv6
IP: 2404:6800:4005:804::2005
IPv4
IP: 172.217.24.69
lebavui@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

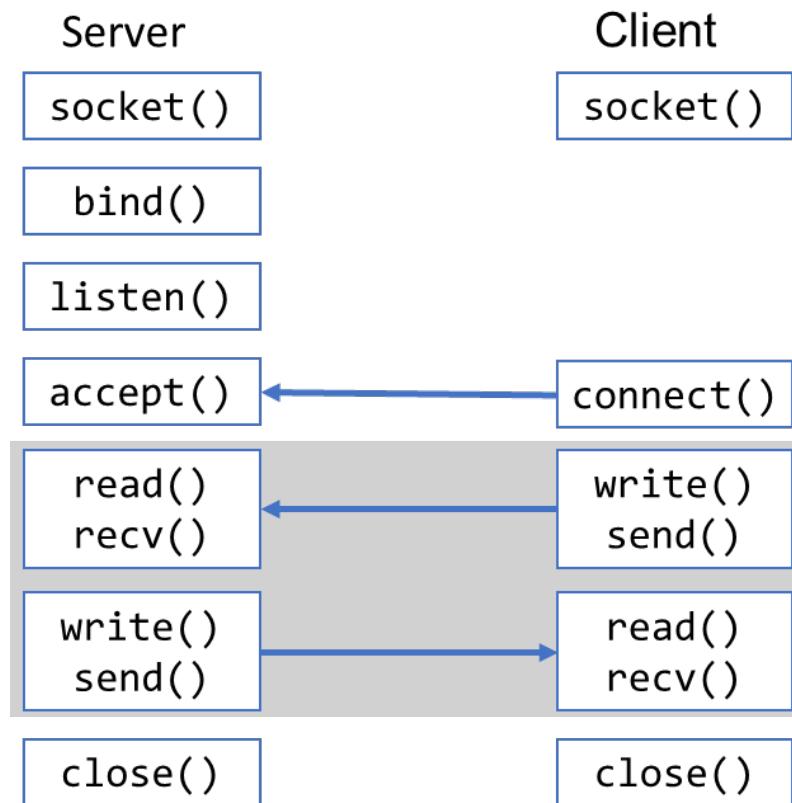
lebavui@Home-Desktop:/mnt/d/Workspaces/socket_tutorials$ ./domain2ip vnexpress.net
IPv6
IP: 2402:dd40:f000:2::8
IPv4
IP: 111.65.250.2
lebavui@Home-Desktop:/mnt/d/Workspaces/socket_tutorials$ nslookup vnexpress.net
Server:      203.113.131.2
Address:      203.113.131.2#53

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

```

### 2.3. Ứng dụng server/client theo giao thức TCP

Trình tự các hàm thường được sử dụng để lập trình ứng dụng server hoặc client hoạt động theo giao thức TCP được mô tả trong hình bên dưới. Ta cần khai báo thư viện `<sys/socket.h>` để có thể sử dụng được các hàm này.



Mô tả cụ thể các hàm như sau:

#### Hàm socket()

Hàm được sử dụng để tạo SOCKET để chờ kết nối hoặc gửi/nhận dữ liệu. Cú pháp của hàm như sau:

```

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
)

```

Kết quả trả về của hàm:

- Trong trường hợp thành công, hàm trả về mô tả của socket (là một số nguyên lớn hơn 0)
- Trong trường hợp gặp lỗi, không tạo được socket hợp lệ, hàm trả về giá trị -1.

Bên dưới là 2 ví dụ về việc tạo socket theo giao thức TCP và giao thức UDP.

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

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

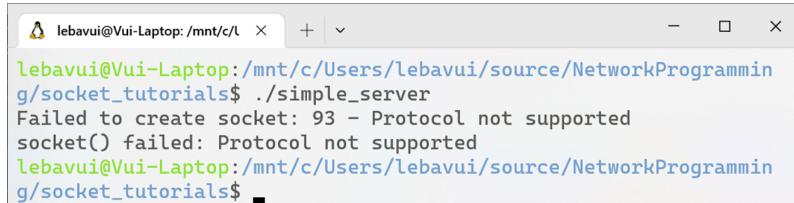
Hầu hết các hàm trong thư viện socket sẽ trả về giá trị -1 nếu gặp lỗi. Để xác định lỗi cụ thể, có thể thực hiện các cách sau:

- Để xác định mã lỗi (dưới dạng số nguyên) của hàm được gọi gần nhất, sử dụng biến **errno**, để sử dụng biến này cần khai báo thư viện **errno.h**.
- Để xác định mô tả lỗi của hàm được gọi gần nhất, sử dụng hàm **perror(const char \*s)**. Hàm này sẽ in ra mô tả lỗi gần nhất, với **s** là chuỗi ký tự tiền tố, được in trước mô tả lỗi. Nếu không sử dụng chuỗi tiền tố, có thể gán bằng NULL (cần khai báo thư viện **stdio.h**).
- Hàm **strerror(int errnum)** có thể được sử dụng để trả về mô tả của mã lỗi được truyền vào bởi tham số **errnum** (cần khai báo thư viện **string.h**).

Ví dụ

```
int listener = socket(AF_INET, SOCK_STREAM, -1); // Tham số -1 sẽ gây ra lỗi
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);
}
```

Kết quả sẽ hiển thị như sau:



```
lebavui@Vui-Laptop:/mnt/c/Users/lebavui/source/NetworkProgramming/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/NetworkProgramming/socket_tutorials$
```

### Hàm bind()

Hàm được sử dụng để gắn socket với cấu trúc địa chỉ trong ứng dụng server. Cú pháp của hàm như sau:

```
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
)
```

Kết quả trả về của hàm:

- Trường hợp thành công, trả về 0
- Trường hợp lỗi, trả về -1

Trước khi sử dụng hàm này, socket cần phải được khởi tạo thành công và cần phải có khai báo cấu trúc địa chỉ của server. Cấu trúc địa chỉ có thể khởi tạo theo giao thức IPv4 sockaddr\_in hoặc IPv6 sockaddr\_in6 và cần phải ép kiểu sang dạng cấu trúc sockaddr để truyền cho hàm.

Ví dụ về sử dụng hàm bind():

```
// 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));
```

Trong ví dụ trên server sẽ chờ kết nối tại cổng 9000.

### Hàm listen()

Hàm chuyển socket sang trạng thái chờ kết nối. Cú pháp của hàm như sau:

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

Giải thích các tham số của hàm:

- Tham số **sockfd**:

Đây là socket đã được tạo và liên kết với một địa chỉ ( thông qua bind()). Socket này phải là loại SOCK\_STREAM (ví dụ: TCP) hoặc các loại socket hỗ trợ kết nối khác. Hàm listen() không áp dụng cho socket SOCK\_DGRAM (UDP).

- Tham số **backlog**:

Xác định số lượng kết nối tối đa có thể được xếp hàng đợi trong hàng đợi kết nối (connection queue). Các kết nối trong hàng đợi là những kết nối đã hoàn thành giao thức bắt tay TCP (TCP three-way handshake) nhưng chưa được server gọi accept() để xử lý. Giá trị backlog thường được đặt trong khoảng từ 5 đến 128, tùy thuộc vào hệ thống. Nếu vượt quá giới hạn của hệ thống, hệ thống sẽ tự động giới hạn lại.

Giá trị trả về của hàm:

- Trường hợp thành công, trả về giá trị 0.
- Trường hợp lỗi, trả về giá trị -1.

### Hàm accept()

Hàm được sử dụng ở phía server để chấp nhận một kết nối đến từ client trên một socket đang ở trạng thái chờ sau khi đã gọi hàm listen(). Hàm này lấy một kết nối từ hàng đợi các kết nối đã hoàn thành và tạo ra một socket mới để giao tiếp với client đó.

Cú pháp của hàm:

```
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
)

```

Giải thích các tham số:

- **sockfd**: File descriptor của socket đang ở trạng thái nghe, được tạo bởi socket() và đã gọi listen().
- **addr**: Con trỏ đến một cấu trúc struct sockaddr để lưu thông tin địa chỉ của client (ví dụ: địa chỉ IP và cổng). Có thể là NULL nếu không cần thông tin này.
- **addrlen**: Con trỏ đến một biến kiểu socklen\_t, chỉ định kích thước của cấu trúc addr. Khi hàm accept() hoàn thành, nó cập nhật giá trị này để phản ánh kích thước thực tế của địa chỉ client. Có thể là NULL nếu addr là NULL.

Giá trị trả về của hàm:

- Trường hợp thành công, hàm trả về giá trị lớn hơn hoặc bằng 0, là mô tả của socket mới được tạo để giao tiếp với client.
- Trường hợp lỗi, hàm trả về giá trị -1.

#### Chú ý:

- Hàm accept() chấp nhận và trả về socket cho mỗi một kết nối với client. Do đó, muốn server xử lý được nhiều kết nối khác nhau thì cần gọi hàm accept() nhiều lần.
- Hàm accept() là một hàm blocking, nghĩa là nó sẽ dừng luồng thực thi cho đến khi có kết quả trả về, sau đó mới thực hiện lệnh tiếp theo.

Ví dụ về lời gọi hàm accept()

```

// Giả sử 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()

Hàm được sử dụng để gửi dữ liệu qua một socket đã được kết nối, thường là socket kiểu SOCK\_STREAM (như TCP) hoặc đôi khi là SOCK\_DGRAM (như UDP trong trường hợp đã kết nối). Hàm này được dùng ở cả client và server để truyền dữ liệu đến phía đối diện của kết nối.

Cú pháp của hàm send():

```

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
)

```

Giải thích các tham số:

- sockfd: File descriptor của socket đã được kết nối (thường được trả về từ connect() ở client hoặc accept() ở server).
- buf: Con trỏ đến bộ đệm chứa dữ liệu cần gửi.
- len: Độ dài (số byte) của dữ liệu trong bộ đệm buf cần gửi.
- flags: Các cờ điều khiển hành vi của hàm send(). Thường được đặt là 0 nếu không sử dụng cờ đặc biệt. Một số cờ phổ biến:
  - MSG\_DONTWAIT: Gửi ở chế độ không chặn (non-blocking).
  - MSG\_MORE: Báo hiệu rằng có thêm dữ liệu sẽ được gửi (dùng để tối ưu hóa TCP).
  - MSG\_NOSIGNAL: Ngăn gửi tín hiệu SIGPIPE nếu kết nối bị đóng (trên một số hệ thống).

Giá trị trả về:

- Trường hợp thành công, hàm trả về số byte thực sự được gửi đi. Có thể nhỏ hơn len nếu dữ liệu chỉ được gửi một phần (đặc biệt trong chế độ không chặn hoặc khi bộ đệm mạng đầy).
- Trường hợp lỗi, hàm trả về giá trị -1.

Ví dụ về sử dụng hàm send() để truyền dữ liệu:

```
// 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 * sizeof(char), 0);

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

Trong ví dụ thứ nhất, hàm send() được sử dụng để truyền một chuỗi ký tự. Để xác định số byte cần gửi, hàm strlen() được sử dụng để trả về độ dài của chuỗi, cũng chính là số ký tự trong chuỗi. Nếu muốn truyền thêm cả ký tự kết thúc xâu, thì có thể sửa lại tham số thứ 3 là strlen() + 1.

Ở ví dụ thứ 2, hàm send() được sử dụng để truyền đi một mảng dữ liệu. Số byte cần gửi được tính bằng số phần tử cần truyền nhân với kích thước của một phần tử.

Ở ví dụ thứ 3, hàm send() được sử dụng để truyền đi một biến có kiểu dữ liệu bất kỳ (bao gồm của kiểu dữ liệu cấu trúc). Toán tử địa chỉ được dùng để truy nhập vào vùng nhớ chứa dữ liệu của biến. Tham số thứ 3 truyền vào kích thước của biến (số byte cấp phát để lưu trữ biến trong bộ nhớ).

**Chú ý:** Trong trường hợp dữ liệu cần gửi rất lớn (ví dụ như nội dung của một file), thì cần tách dữ liệu ra thành các buffer với kích thước nhỏ hơn, và truyền

trong nhiều lần, mỗi một lần truyền một buffer.

### Hàm write()

Tương tự như hàm send(), tuy nhiên có cú pháp rút gọn hơn, để sử dụng hàm này cần phải khai báo tệp tiêu đề unistd.h.

Cú pháp hàm write():

```
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
)
```

Giá trị trả về:

- Trường hợp thành công, hàm trả về số byte thực sự được gửi đi. Có thể nhỏ hơn len nếu dữ liệu chỉ được gửi một phần.
- Trường hợp lỗi, hàm trả về giá trị -1.

Ví dụ về sử dụng hàm write() để truyền dữ liệu:

```
// 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 * sizeof(char));

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

### Hàm recv()

Hàm được sử dụng để nhận dữ liệu từ một socket đã kết nối, thường là socket kiểu SOCK\_STREAM (như TCP) hoặc SOCK\_DGRAM (như UDP trong trường hợp đã kết nối). Hàm này được dùng ở cả client và server để đọc dữ liệu được gửi từ phía đối diện của kết nối.

Cú pháp của hàm recv():

```
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
)
```

Giải thích các tham số:

- sockfd: File descriptor của socket đã kết nối (thường được trả về từ connect() ở client hoặc accept() ở server).
- buf: Con trỏ đến bộ đệm nơi dữ liệu nhận được sẽ được lưu trữ.
- len: Độ dài tối đa (số byte) mà bộ đệm buf có thể chứa.
- flags: Các cờ điều khiển hành vi của hàm recv(). Thường được đặt là 0 nếu không sử dụng cờ đặc biệt. Một số cờ phổ biến:
  - MSG\_WAITALL: Chờ cho đến khi nhận đủ len byte (chỉ áp dụng cho TCP).
  - MSG\_DONTWAIT: Thực hiện nhận ở chế độ không chặn (non-blocking).
  - MSG\_PEEK: Nhìn trước dữ liệu mà không xóa khỏi bộ đệm nhận.
  - MSG\_TRUNC: Yêu cầu trả về độ dài thực của gói tin, ngay cả khi nó bị cắt bớt (thường dùng với UDP).

Hàm recv() đọc dữ liệu từ socket sockfd và lưu vào bộ đệm buf. Nó thường được sử dụng trong các ứng dụng mạng để nhận dữ liệu từ phía đối diện (client hoặc server). Với TCP, dữ liệu được nhận dưới dạng luồng (stream), còn với UDP (socket đã kết nối), dữ liệu được nhận dưới dạng datagram.

Giá trị trả về

- > 0: Số byte thực sự nhận được và lưu vào buf.
- 0: Kết nối đã được đóng bởi phía đối diện (thường với TCP, nghĩa là phía kia gọi close() hoặc shutdown()).
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

Ví dụ về sử dụng hàm recv() để nhận dữ liệu:

```
// 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
}
```

**Chú ý:** Nếu phía đối diện gửi dữ liệu có kích thước nhỏ hơn kích thước của buffer, thì toàn bộ dữ liệu được nhận trong một lần gọi lệnh recv(), với giá trị trả về chính là kích thước của dữ liệu. Tuy nhiên nếu phía đối diện gửi dữ liệu lớn hơn kích thước của buffer, thì mỗi lần gọi lệnh recv() chỉ nhận được một phần của dữ liệu (bằng với kích thước buffer). Do đó trong trường hợp này, cần phải gọi lệnh recv() nhiều lần để có thể nhận được toàn bộ dữ liệu.

### Hàm read()

Tương tự như hàm recv(), tuy nhiên có cú pháp rút gọn hơn, để sử dụng hàm này cần phải khai báo tệp tiêu đề **unistd.h**.

Cú pháp hàm read():

```
#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)
)
```

Giá trị trả về

- > 0: Số byte thực sự nhận được và lưu vào buf.
- 0: Kết nối đã được đóng bởi phía đối diện (thường với TCP, nghĩa là phía kia gọi close() hoặc shutdown()).
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

Ví dụ về sử dụng hàm read() để nhận dữ liệu:

```
// 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()**

Hàm được sử dụng để đóng một file descriptor, bao gồm các socket, nhằm giải phóng tài nguyên liên quan và chấm dứt kết nối mạng (nếu có). Trong ngữ cảnh lập

trình mạng, close() thường được dùng để đóng socket sau khi hoàn tất giao tiếp, ở cả client và server.

Cú pháp của hàm close():

```
#include <unistd.h>

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

Giá trị trả về:

- 0: Nếu đóng file descriptor thành công.
- -1: Nếu có lỗi xảy ra, và biến **errno** được thiết lập để chỉ ra nguyên nhân lỗi.

### Hàm shutdown()

Hàm được sử dụng để đóng một hoặc cả hai chiều của kết nối trên một socket, chủ yếu áp dụng cho socket kiểu SOCK\_STREAM (như TCP). Không giống hàm close(), shutdown() cho phép kiểm soát linh hoạt hơn bằng cách đóng riêng lẻ chiều gửi hoặc nhận, mà không giải phóng hoàn toàn file descriptor của socket.

Cú pháp của hàm shutdown():

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

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

Giải thích các tham số của hàm:

- **sockfd**: File descriptor của socket cần thực hiện thao tác đóng. Đây là socket đã được kết nối (qua connect() ở client hoặc accept() ở server).
- **how**: Xác định cách đóng kết nối. Các giá trị hợp lệ:
  - **SHUT\_RD** (0): Đóng chiều nhận (read), ngăn socket nhận thêm dữ liệu.
  - **SHUT\_WR** (1): Đóng chiều gửi (write), ngăn socket gửi thêm dữ liệu.

- **SHUT\_RDWR** (2): Đóng cả hai chiều nhận và gửi.

Giá trị trả về:

- 0: Nếu thao tác đóng thành công.
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

**Chú ý:** Lệnh shutdown(fd, SHUT\_RDWR) tương đương lệnh close(fd).

Như vậy, quy trình thực hiện ứng dụng server theo giao thức TCP sẽ theo các bước như sau:

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

Sau đây là một ví dụ về ứng dụng server chờ kết nối ở cổng 9000, sau đó nhận dữ liệu từ client và gửi lại chính dữ liệu đó cho client.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <errno.h>
9
10 int main()
11 {
12     // Tao socket chở kết nối
13     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
14     if (listener == -1)
15     {

```

```
16         perror("socket() failed");
17         return 1;
18     }
19
20     // Khai báo cấu trúc địa chỉ của server
21     struct sockaddr_in addr;
22     addr.sin_family = AF_INET;
23     addr.sin_addr.s_addr = htonl(INADDR_ANY);
24     addr.sin_port = htons(9000);
25
26     // Gắn socket với cấu trúc địa chỉ
27     if (bind(listener, (struct sockaddr *)&addr, sizeof(addr)))
28     {
29         perror("bind() failed");
30         return 1;
31     }
32
33     // Chuyển socket sang trạng thái chờ kết nối
34     if (listen(listener, 5))
35     {
36         perror("listen() failed");
37         return 1;
38     }
39
40     printf("waiting for a new client ...\\n");
41
42     // Chờ và chấp nhận kết nối
43     int client = accept(listener, NULL, NULL);
44     if (client == -1)
45     {
46         perror("accept() failed");
47         return 1;
48     }
49     printf("new client connected: %d\\n", client);
50
51     // Nhận dữ liệu từ client
52     char buf[256];
53     int ret = recv(client, buf, sizeof(buf), 0);
54
55     // Kiểm tra kết nối có bị đóng hoặc hủy không
56     if (ret <= 0)
57     {
58         printf("recv() failed.\\n");
59         return 1;
60     }
```

```

61
62     // Thêm ký tự kết thúc xâu và in ra màn hình
63     if (ret < sizeof(buf))
64         buf[ret] = 0;
65     puts(buf);
66
67     // Gửi dữ liệu sang client
68     send(client, buf, strlen(buf), 0);
69
70     // Đóng kết nối
71     close(client);
72     close(listener);
73
74     return 0;
75 }
```

Dể thử nghiệm sự hoạt động của chương trình này, công cụ netcat được sử dụng để tạo TCP client và kết nối với server trên.

Chương trình được thực hiện sau khi dịch thành công, server ở trạng thái chờ kết nối từ client:

```

lebavui@VUI5490:~/network_programming$ ./simple_server
waiting for a new client ...
```

Tiếp theo, netcat được sử dụng để tạo client kết nối đến server:

```

lebavui@VUI5490:~$ nc -v 127.0.0.1 9000
Connection to 127.0.0.1 9000 port [tcp/*] succeeded!
Hello
Hello

lebavui@VUI5490:~$
```

Kết quả hiển thị trên màn hình của server:

```

lebavui@VUI5490:~/network x lebavui@VUI5490:~ x + x
lebavui@VUI5490:~/network_programming$ ./simple_server
waiting for a new client ...
new client connected: 4
Hello
lebavui@VUI5490:~/network_programming$
```

Quy trình thực hiện ứng dụng client theo giao thức TCP sẽ theo các bước như sau:

1. Tạo socket qua hàm **socket()**
2. Điền thông tin về server sử dụng cấu trúc **sockaddr\_in**
3. Kết nối tới server qua hàm **connect()**
4. Gửi dữ liệu tới server thông qua hàm **send()**
5. Nhận dữ liệu từ server thông qua hàm **recv()**
6. Đóng socket khi việc truyền nhận kết thúc bằng hàm **close()**

### Hàm connect()

Hàm được sử dụng chủ yếu ở phía client để thiết lập kết nối tới một server thông qua một socket. Hàm này áp dụng cho các socket kiểu SOCK\_STREAM (như TCP) hoặc SOCK\_DGRAM (như UDP trong một số trường hợp).

Cú pháp của hàm connect():

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

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
)
```

Giá trị trả về của hàm:

- 0: Nếu kết nối thành công (hoặc liên kết địa chỉ thành công với UDP).
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

**Chú ý:** Địa chỉ server mà client kết nối đến (tham số addr của hàm connect()) có thể được xác định thông qua:

- Khai báo trực tiếp địa chỉ IP và cổng
- Phân giải từ tên miền

Sau đây là một ví dụ về ứng dụng client chờ kết nối đến server ở địa chỉ IP 127.0.0.1, cổng 9000, sau đó gửi dữ liệu đến server rồi nhận dữ liệu từ server.

```

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 #include <stdlib.h>
9
10 int main() {
11     // Khai báo socket kết nối đến server
12     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
13
14     // Khai báo địa chỉ của server
15     struct sockaddr_in addr;
16     addr.sin_family = AF_INET;
17     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
18     addr.sin_port = htons(9000);
19
20     // Kết nối đến server
21     int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
22     if (res == -1) {
23         perror("connect() failed");
24         return 1;
25     }
26
27     // Gửi tin nhắn đến server
28     char *msg = "Hello server";
29     send(client, msg, strlen(msg), 0);
30
31     // Nhận tin nhắn từ server
32     char buf[2048];
33     int len = recv(client, buf, sizeof(buf), 0);
34 }
```

```

35     // Nếu kết nối bị đóng hoặc bị lỗi thì kết thúc chương trình
36     if (len <= 0)
37     {
38         printf("recv() failed.\n");
39         return 1;
40     }
41
42     // Xử lý dữ liệu nhận được
43     // Thêm ký tự kết thúc xâu và in ra màn hình
44     if (len < sizeof(buf))
45     {
46         buf[len] = 0;
47         printf("Data received: %s\n", buf);
48
49         // Kết thúc, đóng socket
50         close(client);
51
52     }

```

Để thử nghiệm sự hoạt động của chương trình này, công cụ netcat được sử dụng để tạo TCP server chờ client kết nối đến.

```

lebavui@VUI5490:~/network$ nc -v -l -p 9000
Listening on 0.0.0.0 9000

```

Sau đó, chương trình client chạy, kết nối và truyền dữ liệu đến server:

```

lebavui@VUI5490:~/network$ ./simple_client

```

Server chấp nhận kết nối và hiển thị dữ liệu nhận được từ client, server gửi dữ liệu cho client:

```
lebavui@VUI5490:~/network_programming$ nc -v -l -p 9000
Listening on 0.0.0.0 9000
Connection received on localhost 49720
Hello server Hello client
lebavui@VUI5490:~/network_programming$
```

Kết quả hiển thị trên màn hình client:

```
lebavui@VUI5490:~/network_programming$ ./simple_client
Data received: Hello client

lebavui@VUI5490:~/network_programming$
```

### Một số ví dụ minh họa

- Lập trình server nhận và hiển thị yêu cầu từ trình duyệt.
- Lập trình client gửi yêu cầu đến Web server và hiển thị kết quả trả về.
- Lập trình client và server truyền nhận dữ liệu là chuỗi ký tự.
- Lập trình client và server truyền nhận dữ liệu là dãy số.
- Lập trình client và server truyền nhận dữ liệu từ file.

## 2.4. Ứng dụng sender/receiver theo giao thức UDP

Giao thức UDP là giao thức không kết nối, các ứng dụng không cần phải thực hiện việc thiết lập kết nối trước khi truyền nhận dữ liệu như trong giao thức TCP. Các ứng dụng có thể truyền dữ liệu khi biết địa chỉ của bên nhận và nhận dữ liệu từ máy tính bất kỳ trong mạng.

### Ứng dụng UDP Sender - gửi dữ liệu

Trình tự các bước cần thực hiện ở bên gửi được mô tả bởi hình bên dưới:



1. Tạo socket theo giao thức UDP.
2. Xác định địa chỉ bên nhận.
3. Truyền dữ liệu bằng lệnh **sendto()**.

### Hàm sendto()

Hàm được sử dụng để gửi dữ liệu qua một socket, thường áp dụng cho socket kiểu SOCK\_DGRAM (như UDP) hoặc các socket không kết nối. Không giống như hàm send(), hàm sendto() cho phép chỉ định địa chỉ đích (IP và cổng) của nơi nhận dữ liệu trong mỗi lần gửi, giúp nó phù hợp cho các giao thức không kết nối như UDP.

Cú pháp của hàm sendto():

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

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
)
```

Giải thích các tham số của hàm:

- sockfd: File descriptor của socket, được tạo bởi hàm socket().
- buf: Con trỏ đến bộ đệm chứa dữ liệu cần gửi.
- len: Độ dài (số byte) của dữ liệu trong bộ đệm buf.
- flags: Các cờ điều khiển hành vi của hàm. Thường là 0, nhưng có thể bao gồm:
  - MSG\_DONTWAIT: Gửi ở chế độ không chặn (non-blocking).
  - MSG\_NOSIGNAL: Ngăn gửi tín hiệu SIGPIPE nếu kết nối bị đóng (chủ yếu với TCP).
- addr: Con trỏ đến cấu trúc struct sockaddr chứa thông tin địa chỉ của đích (IP và cổng).

- addr\_len: Kích thước của cấu trúc addr (tính bằng byte).

Giá trị trả về

- $>= 0$ : Số byte thực sự được gửi đi.
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

Ví dụ lập trình ứng dụng UDP Sender gửi dữ liệu, ứng dụng này sẽ nhận dữ liệu từ bàn phím và truyền sang bên nhận:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <string.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main()
10 {
11     // Tạo socket theo giao thức UDP
12     int sender = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
13
14     // Khai báo địa chỉ bên nhận
15     struct sockaddr_in addr;
16     addr.sin_family = AF_INET;
17     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
18     addr.sin_port = htons(9090);
19
20     // Gửi tin nhắn
21     char buf[256];
22     while (1)
23     {
24         printf("Enter message: ");
25         fgets(buf, sizeof(buf), stdin);
26         sendto(sender, buf, strlen(buf), 0, (struct sockaddr *)&addr, sizeof(addr));
27     }
28
29     return 0;
30 }
```

Để thử nghiệm hoạt động của chương trình này, ứng dụng netcat được sử dụng để tạo một UDP receiver để nhận dữ liệu.

### Ứng dụng UDP Receiver - nhận dữ liệu

Thứ tự các bước cần thực hiện ở bên nhận được mô tả theo hình sau:



1. Tạo socket theo giao thức UDP.
2. Khai báo cấu trúc địa chỉ nhận và gắn với socket bằng hàm **bind()**.
3. Nhận dữ liệu bằng hàm **recvfrom()**.

#### Hàm recvfrom()

Hàm được sử dụng để nhận dữ liệu từ một socket, chủ yếu áp dụng cho socket kiểu SOCK\_DGRAM (như UDP) hoặc các socket không kết nối. Không giống như hàm recv(), hàm recvfrom() cho phép nhận dữ liệu cùng với thông tin về địa chỉ nguồn (IP và cổng) của thiết bị gửi dữ liệu. Điều này làm cho nó đặc biệt phù hợp với các giao thức không kết nối như UDP.

Cú pháp của hàm recvfrom():

```

#include <sys/socket.h>

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
)
  
```

Giải thích các tham số:

- sockfd: File descriptor của socket, được tạo bởi hàm socket().
- buf: Con trỏ đến bộ đệm nơi dữ liệu nhận được sẽ được lưu trữ.
- len: Độ dài tối đa (số byte) mà bộ đệm buf có thể chứa.

- flags: Các cờ điều khiển hành vi của hàm. Thường là 0, nhưng có thể bao gồm:
  - MSG\_DONTWAIT: Nhận ở chế độ không chặn (non-blocking).
  - MSG\_PEEK: Nhìn trước dữ liệu mà không xóa khỏi bộ đệm nhận.
  - MSG\_TRUNC: Yêu cầu trả về độ dài thực của datagram, ngay cả khi nó bị cắt bớt (thường dùng với UDP).
- src\_addr: Con trỏ đến cấu trúc struct sockaddr để lưu thông tin địa chỉ của nguồn gửi dữ liệu (IP và cổng). Có thể là NULL nếu không cần thông tin này.
- addrlen: Con trỏ đến một biến kiểu socklen\_t, chỉ định kích thước của src\_addr. Khi hàm hoàn thành, nó cập nhật giá trị này để phản ánh kích thước thực tế của địa chỉ nguồn. Có thể là NULL nếu src\_addr là NULL.

Giá trị trả về:

- > 0: Số byte thực sự nhận được và lưu vào buf.
- -1: Nếu có lỗi xảy ra, và biến errno được thiết lập để chỉ ra nguyên nhân lỗi.

Ví dụ lập trình ứng dụng UDP Receiver nhận dữ liệu, ứng dụng này sẽ nhận dữ liệu từ bên gửi và in ra màn hình.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <string.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8 #include <poll.h>
9
10 int main()
11 {
12     // Tạo socket theo giao thức UDP
13     int receiver = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
14
15     // Khai báo địa chỉ bên nhận
16     struct sockaddr_in addr;
17     addr.sin_family = AF_INET;
18     addr.sin_addr.s_addr = htonl(INADDR_ANY);
19     addr.sin_port = htons(9090);

```

```

20
21     bind(receiver, (struct sockaddr *)&addr, sizeof(addr));
22
23     // Nhận tin nhắn
24     char buf[16];
25     while (1)
26     {
27         int ret = recvfrom(receiver, buf, sizeof(buf), 0, NULL, NULL);
28         if (ret == -1)
29         {
30             printf("recvfrom() failed\n");
31             break;
32         }
33         else
34         {
35             buf[ret] = 0;
36             printf("%d - %s\n", ret, buf);
37         }
38     }
39
40     return 0;
41 }
```

Để thử nghiệm hoạt động của chương trình này, ứng dụng netcat được sử dụng để tạo một UDP sender để truyền dữ liệu.

#### Chú ý:

- Các lệnh nhận dữ liệu bao gồm:

```

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 cho nhau trong các ứng dụng TCP, UDP.

- Các lệnh truyền dữ liệu bao gồm:

```

send(client, buf, sizeof(buf), 0);
sendto(client, buf, sizeof(buf), 0, NULL, 0);
```

Hai lệnh này tương đương nhau, có thể sử dụng thay thế cho nhau trong các ứng dụng TCP, UDP.

#### Kịch bản thử nghiệm giao thức hướng dòng TCP:

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

**Kịch bản thử nghiệm giao thức hướng thông điệp UDP:**

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

## 2.5. Bài tập

**Bài tập 2.1.** Lập trình ứng dụng client và server truyền nhận dữ liệu là chuỗi ký tự. Việc truyền nhận dừng lại khi server nhận được chuỗi “Ky thuat May tinh” từ client. Biết rằng chuỗi “Ky thuat May tinh” có thể không được nhận cùng một lệnh.

**Bài tập 2.2.** 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>

**Bài tập 2.3.** 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>

**Bài tập 2.4.** 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.

**Bài tập 2.5.** 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 server-client

---

3.1. Các chế độ hoạt động của socket . . . . .	48
3.2. Server hoạt động theo cơ chế lặp . . . . .	53
3.3. Kỹ thuật đa tiến trình - Multiprocessing . . . . .	55
3.4. Kỹ thuật đa luồng - Multithreading . . . . .	71
3.5. Kỹ thuật ghép kênh - Multiplexing . . . . .	84
3.6. Bài tập . . . . .	102

---

### 3.1. Các chế độ hoạt động của socket

#### Chế độ đồng bộ (Blocking)

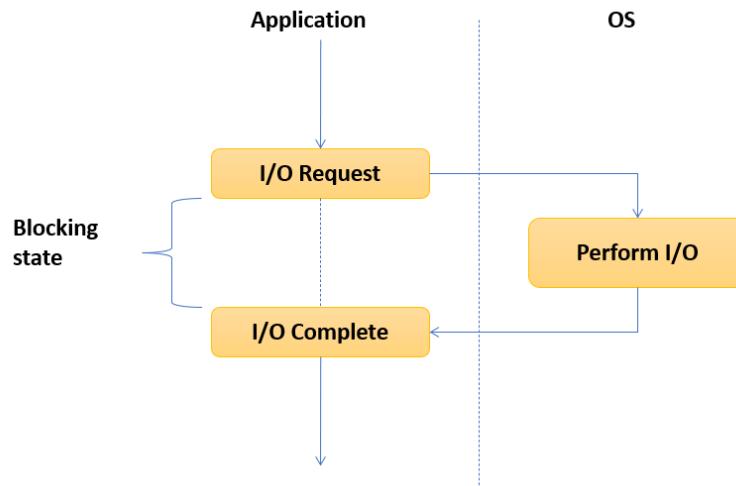
Là chế độ mà các hàm vào ra sẽ chặn luồng thực thi (luồng gọi hàm) cho đến khi thao tác vào ra hoàn tất (các hàm vào ra sẽ không trả về kết quả cho đến khi thao tác hoàn tất). Đây là chế độ mặc định của socket, được thiết lập khi socket được tạo.

Các hàm vào ra ảnh hưởng:

- accept()
- connect()
- send()
- recv()
- ...

Chế độ đồng bộ thích hợp với các ứng dụng xử lý tuần tự, các xử lý được thực hiện theo từng bước, kết quả bước này được sử dụng ở bước tiếp theo. Ta không nên gọi các hàm đồng bộ khi ở trong luồng xử lý giao diện (GUI thread), bởi vì việc chờ các hàm vào ra có thể gây ảnh hưởng đến việc xử lý giao diện, làm cho giao diện có thể bị treo.

Ví dụ: thread bị chặn bởi hàm recv() thì không thể thực hiện các công việc khác.



Hình 3.1 Chế độ đồng bộ

```

...
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 ();
...

```

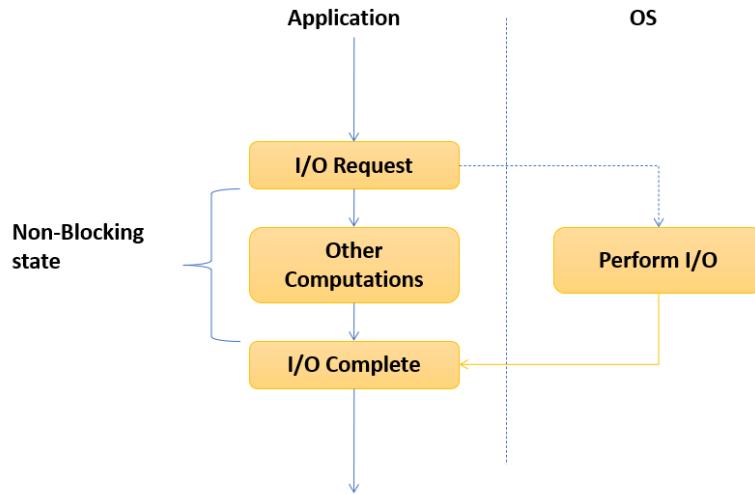
### Chế độ bất đồng bộ (Non-blocking)

Là chế độ mà các thao tác vào ra sẽ trả về nơi gọi ngay lập tức và tiếp tục thực thi luồng. 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 ở chế độ bất đồng bộ sẽ không dừng luồng gọi các hàm này.

Trong thư viện lập trình Linux, 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.

Các hàm bất đồng bộ có thể sử dụng trong luồng xử lý giao diện của ứng dụng, và thích hợp với các ứng dụng hướng sự kiện.

### Chuyển socket sang chế độ bất đồng bộ



Hình 3.2 Chế độ bất đồng bộ

Áp dụng cho socket chờ kết nối sử dụng hàm **accept()** và socket nhận dữ liệu sử dụng hàm **recv()**.

Hàm **ioctl()** được sử dụng để chuyển socket sang trạng thái bất đồng bộ. Để sử dụng hàm này cần phải khai báo thư viện **sys/ioctl.h**.

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

unsigned long mode = 1; // 1 = non-blocking, 0 = blocking
ioctl(socket_fd, FIONBIO, &mode);
```

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);
```

Chương trình dưới đây minh họa việc chuyển socket sang chế độ hoạt động bất đồng bộ, khi đó server có thể chấp nhận nhiều kết nối và xử lý các yêu cầu từ client một cách nhanh chóng.

---

1    `#include <stdio.h>`  
 2    `#include <stdlib.h>`

```
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <sys/ioctl.h>
9 #include <errno.h>
10
11 //brief Hàm xóa client khỏi mảng
12 //param clients Mảng client đang kết nối đến server
13 //param pNumClients Địa chỉ biến chứa số lượng client
14 //param index Thứ tự của phần tử cần xóa
15 void removeClient(int *clients, int *pNumClients, int index)
16 {
17     if (index < *pNumClients - 1)
18         clients[index] = clients[*pNumClients - 1];
19     *pNumClients = *pNumClients - 1;
20 }
21
22 int main()
23 {
24     // Tạo socket
25     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
26     if (listener == -1)
27     {
28         perror("socket() failed");
29         return 1;
30     }
31
32     // Chuyển socket sang trạng thái bắt đầu đồng bộ
33     unsigned long ul = 1;
34     ioctl(listener, FIONBIO, &ul);
35
36     // Khai báo cấu trúc địa chỉ server
37     struct sockaddr_in addr;
38     addr.sin_family = AF_INET;
39     addr.sin_addr.s_addr = htonl(INADDR_ANY);
40     addr.sin_port = htons(9000);
41
42     // Gắn địa chỉ với socket
43     if (bind(listener, (struct sockaddr *)&addr, sizeof(addr)))
44     {
45         perror("bind() failed");
46         return 1;
47     }
```

```

48
49     if (listen(listener, 5))
50     {
51         perror("listen() failed");
52         return 1;
53     }
54
55     int clients[64];
56     int numClients = 0;
57     char buf[256];
58
59     while (1)
60     {
61         // Chấp nhận kết nối
62         int client = accept(listener, NULL, NULL);
63         if (client == -1)
64         {
65             // Nếu lỗi không phải do đang chờ kết nối
66             if (errno != EWOULDBLOCK)
67             {
68                 perror("accept() failed");
69                 return 1;
70             }
71             else
72             {
73                 // Nếu lỗi do đang chờ kết nối thì bỏ qua, thực hiện công việc khác
74             }
75         }
76         else
77         {
78             // Nếu có kết nối mới thì thêm vào mảng và chuyển sang trạng thái bắt
79             // → đồng bộ
80             printf("New client connected: %d\n", client);
81             clients[numClients++] = client;
82             ul = 1;
83             ioctl(client, FIONBIO, &ul);
84         }
85
86         // Kiểm tra các client có truyền dữ liệu không
87         for (int i = 0; i < numClients; i++)
88         {
89             int ret = recv(clients[i], buf, sizeof(buf), 0);
90             if (ret == -1)
91             {
92                 if (errno != EWOULDBLOCK)

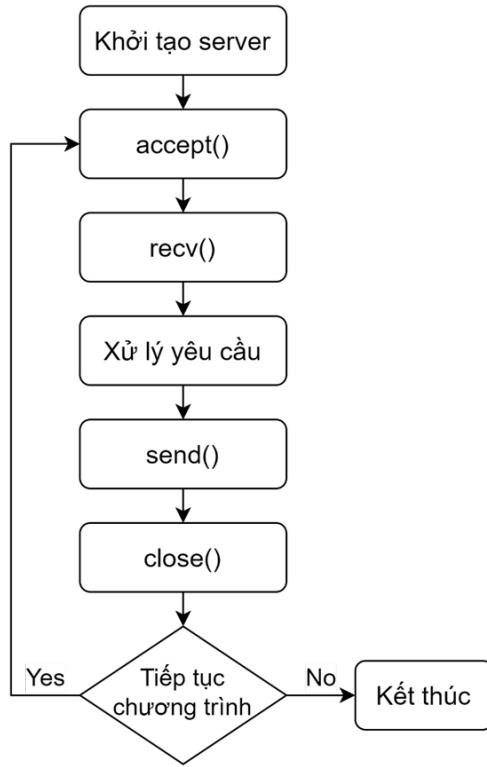
```

```

92     {
93         // Nếu lỗi không phải do đang chờ dữ liệu
94         // Xóa client khỏi mảng
95         // Chuyển sang kiểm tra kết nối khác
96         perror("recv() failed");
97         close(clients[i]);
98         removeClient(clients, &numClients, i--);
99         continue;
100    }
101    else
102    {
103        // Nếu lỗi do đang chờ dữ liệu thì bỏ qua
104    }
105    }
106    else if (ret == 0)
107    {
108        // Nếu kết nối bị đóng, thì xóa client khỏi mảng
109        printf("client disconnected.\n");
110        close(clients[i]);
111        removeClient(clients, &numClients, i--);
112        continue;
113    }
114    else
115    {
116        // Xử lý dữ liệu nhận được
117        // Thêm ký tự kết thúc xâu và in ra màn hình
118        if (ret < sizeof(buf))
119            buf[ret] = 0;
120        puts(buf);
121        // Trả lại kết quả cho client
122        send(clients[i], buf, strlen(buf), 0);
123    }
124 }
125 }
126
127 close(listener);
128
129 return 0;
130 }
```

### 3.2. Server hoạt động theo cơ chế lặp

Iterative server là một kiểu server đơn giản, trong đó server xử lý các yêu cầu client theo thứ tự, tuần tự từng cái một, thường chỉ chạy trên một tiến trình (process)



Hình 3.3 Iterative server

hoặc một luồng (thread).

Các bước xử lý của server bao gồm:

- Chờ một client gửi yêu cầu.
- Xử lý yêu cầu đó.
- Gửi trả kết quả cho client.
- Quay lại bước 1 để nhận client tiếp theo.

Đoạn chương trình dưới đây thực hiện ứng dụng webserver đơn giản, xử lý tuần tự yêu cầu từ trình duyệt.

---

```

1 // Khởi tạo server
2 while (1) {
3   // Chờ kết nối mới
4   int client = accept(listener, NULL, NULL);
  
```

```

5   printf("New client connected: %d\n", client);
6   // Nhận dữ liệu từ client và in ra màn hình
7   char buf[256];
8   int ret = recv(client, buf, sizeof(buf), 0);
9   buf[ret] = 0;
10  puts(buf);
11  // Trả lại kết quả cho client
12  char *msg = "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n<html><body><h1>Xin
13    → chao cac ban</h1></body></html>";
14  send(client, msg, strlen(msg), 0);
15  // Đóng kết nối
16  close(client);
}

```

### 3.3. Kỹ thuật đa tiến trình - Multiprocessing

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.

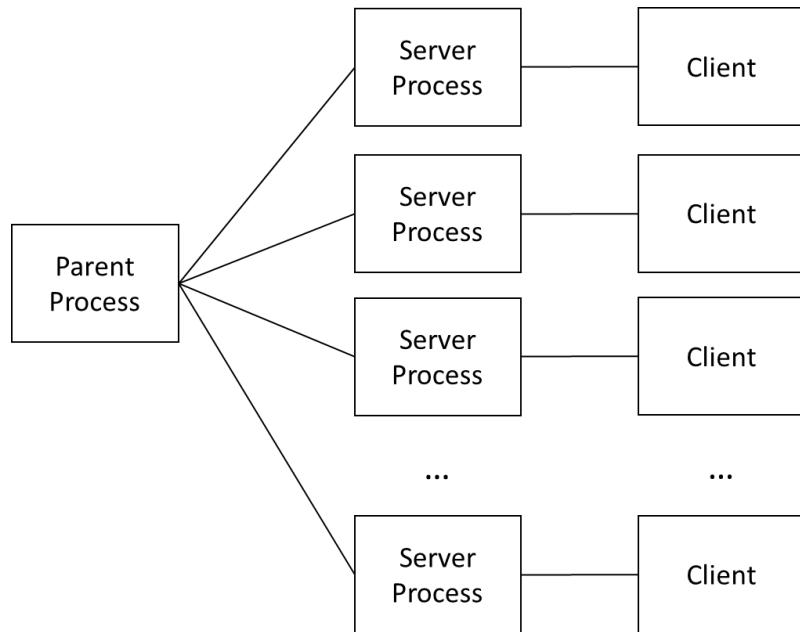
So sánh tiến trình và luồng (process and thread): Luồng là một kiểu tiến trình gọn nhẹ, chiếm ít tài nguyên hơn, và cho phép chia sẻ tài nguyên giữa các luồng. Một tiến trình có thể tạo thêm nhiều luồng.

Để giải quyết vấn đề đồng bộ giữa các hàm vào ra của các socket, một cách đơn giản là tạo ra thêm các tiến trình mới, mỗi tiến trình phụ trách xử lý truyền nhận dữ liệu cho một socket. Khi các tiến trình hoạt động song song sẽ tăng được 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()



Hình 3.4 Kỹ thuật đa tiến trình

Hàm 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 sẽ tạo một bản sao của chính nó, bao gồm các lệnh, biến, con trỏ, ... ngoại trừ id của tiến trình. Tiến trình con được chạy ngay sau lời gọi hàm `fork()` cho đến khi hết chương trình hoặc đến khi gặp lệnh `exit()`.

Trong các ứng dụng đa tiến trình, 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()` không nhận tham số truyền vào và 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. Nếu giá trị trả về của hàm fork() bằng 0, tức là đang ở tiến trình con, ngược lại nếu giá trị trả về khác 0, tức là đang ở tiến trình cha.

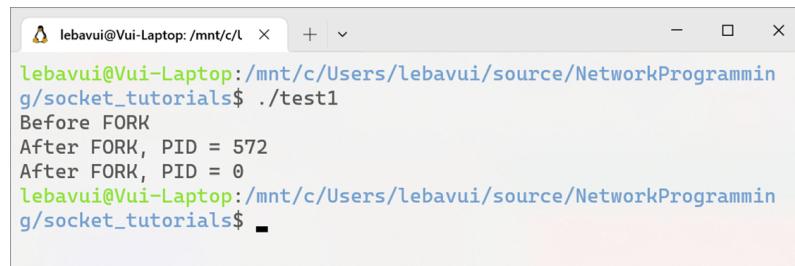
Trong trường hợp bị lỗi, hàm trả về -1, xác định lỗi thông qua errno.

Ví dụ 1: gọi hàm fork()

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

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

Kết quả khi chạy đoạn mã nguồn trên được mô tả trong hình bên dưới, tiến trình con có ID là 572.



```
lebavui@Vui-Laptop: /mnt/c/l ~ + v
lebavui@Vui-Laptop: /mnt/c/Users/lebavui/source/NetworkProgramming/socket_tutorials$ ./test1
Before FORK
After FORK, PID = 572
After FORK, PID = 0
lebavui@Vui-Laptop: /mnt/c/Users/lebavui/source/NetworkProgramming/socket_tutorials$ -
```

Ví dụ 2: Hãy xem đoạn mã nguồn sau

```
#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;
}
```

Trong đoạn mã trên, hàm fork() được gọi 2 lần liên tiếp, câu hỏi đặt ra là có bao nhiêu tiến trình con được tạo mới?

Kết quả khi chạy chương trình:

```
lebavui@Vui-Laptop:/mnt/c/Users/lebavui/source/NetworkProgramming/socket_tutorials$ ./test1
Before FORK, PID = 807
PID = 807
PID = 809
PID = 808
PID = 810
lebavui@Vui-Laptop:/mnt/c/Users/lebavui/source/NetworkProgramming/socket_tutorials$
```

Trước khi gọi hàm fork(), ID của tiến trình hiện tại được in ra là 807. Khi hàm fork() đầu tiên được gọi, một tiến trình con được tạo mới có ID là 808. Khi hàm fork() thứ 2 được gọi thì nó chạy trên cả 2 tiến trình 807 và 808. Do vậy có thêm 2 tiến trình con nữa được tạo với ID là 809 và 810.

### Hàm exit()

Hàm được sử dụng để kết thúc tiến trình hiện tại (tiến trình gọi hàm), có kèm theo trạng thái kết thúc.

Cú pháp hàm exit() như sau:

```
#include <unistd.h>

void exit (int status);
```

Tham số status là tham số báo trạng thái kết thúc, có thể truyền một trong 2 giá trị sau:

- EXIT\_SUCCESS (0)
- EXIT\_FAILURE (1)

### Kiểm soát hoạt động của tiến trình con

Để theo dõi hoặc can thiệp vào hoạt động của tiến trình con, tiến trình cha có thể thực hiện theo các phương án sau:

- 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.

Có 2 trường hợp cần thêm các bước 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()**

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

Cú pháp hàm wait() như sau:

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

pid_t wait (int *status);
```

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

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

Nếu không có tiến trình con nào, hàm trả về -1 với mã lỗi là ECHILD. Do vậy, trong ứng dụng có nhiều tiến trình con được tạo, để đợi tất cả các tiến trình con kết thúc thì cần phải gọi hàm **wait()** liên tục cho đến khi giá trị trả về của hàm **wait()** là -1 và giá trị mã lỗi errno là ECHILD.

Ví dụ về 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;
}

```

Kết quả chạy chương trình được minh họa trong hình bên dưới.

```

Ubuntu ~ cd /home/lebavui/network_programming
network_programming ~/test
Waiting for the child process
Child process started
Child process done
Child process 23010 terminated with status 0
network_programming ~

```

Trong ví dụ trên, mệnh đề if được sử dụng để tách biệt đoạn mã nguồn chạy ở tiến trình con và tiến trình chính. Tiến trình con chạy trong khoảng thời gian 5 giây rồi kết thúc. Tiến trình chính sau khi tạo tiến trình con thì chờ cho đến khi tiến trình con kết thúc rồi mới kết thúc chương trình.

### Hàm kill() / killpg()

Cú pháp

```

#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 tham số **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 tham số **pgrp** là 0 và **sig** là **SIGKILL**.

Ví dụ lập trình với 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 lập trình với 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 (signal) là một cơ chế trong hệ điều hành (Unix/Linux) để thông báo cho tiến trình biết rằng một sự kiện nào đó vừa xảy ra. Ví dụ: khi người dùng nhấn Ctrl+C, hệ thống sẽ gửi tín hiệu SIGINT đến tiến trình đang chạy ở foreground để yêu cầu dừng tiến trình đó.

Khác với lời gọi hàm hoặc ngắt đồng bộ, tín hiệu là sự kiện bất đồng bộ, nghĩa là tiến trình không biết trước khi nào tín hiệu sẽ đến. Ví dụ: tiến trình đang chạy bình thường thì đột ngột có tín hiệu SIGTERM được gửi đến từ một tiến trình khác để yêu cầu kết thúc.

Có hai nguồn chính thực hiện việc gửi tín hiệu:

- Từ tiến trình này sang tiến trình khác: Một tiến trình có thể gửi tín hiệu đến tiến trình khác bằng lệnh kill() trong lập trình hoặc bằng lệnh shell kill <PID>.

Ví dụ: kill -9 1234 gửi tín hiệu SIGKILL đến tiến trình có PID = 1234.

- Từ hệ thống tới tiến trình: Hệ điều hành sẽ tự động gửi một số tín hiệu trong những tình huống đặc biệt.

Ví dụ:

- SIGSEGV khi tiến trình truy cập vào vùng nhớ không hợp lệ.
- SIGFPE khi tiến trình chia cho 0.
- SIGCHLD khi một tiến trình con kết thúc.

Khi một tiến trình con (child process) kết thúc, hệ thống sẽ gửi tín hiệu SIGCHLD đến tiến trình cha (parent process). Tín hiệu này báo cho tiến trình cha rằng con của nó đã kết thúc và cần được “thu dọn” (reap) thông qua hàm wait() hoặc waitpid().

```

993 tty1  /home/Lebavui/.vscode-server Sl
1118 tty3  ps -e -o pid,tty,cmd,stat R
lebavui@Home-Desktop:~$ ps -e -o pid,tty,cmd,stat
PID TT      CMD          STAT
 1 ?     /init          Ss
 6 ?    plan9 --control-socket 6 -- S
12 tty1  /init          Ss
13 tty1  sh <- "$VS CODE WSL_EXT_Loca S
14 tty1  sh -c "/usr/bin/vscode-d S
19 tty1  sh /home/Lebavui/.vscode-se S
23 tty1  /home/Lebavui/.vscode-serv S
34 tty1  /home/Lebavui/.vscode-serv S
439 tty1  /home/Lebavui/.vscode-serv S
465 tty1  /home/Lebavui/.vscode-serv S
493 pts/1 /bin/bash --init-file /home Ss
516 tty1  /home/Lebavui/.vscode-serv S
558 tty1  /home/Lebavui/.vscode-serv S
718 tty2  /init          Ss
719 tty2  -bash          S
745 tty3  /init          Ss
746 tty3  -bash          S
762 tty4  /init          Ss
763 tty4  -bash          S
947 tty1  /home/Lebavui/.vscode-serv S
993 tty1  /home/Lebavui/.vscode-serv Sl
1119 tty2  ./fork_server   S
1121 tty2  [fork_server] <defunct> Z
1123 tty2  [fork_server] <defunct> Z
1124 tty3  ps -e -o pid,tty,cmd,stat R
lebavui@Home-Desktop:~$
```

Hình 3.5 Hình ảnh minh họa tiến trình zombie đang hoạt động

Nếu tiến trình cha không xử lý tín hiệu SIGCHLD, nghĩa là nó không gọi `wait()` để thu dọn thông tin của tiến trình con, thì tiến trình con sau khi kết thúc sẽ rơi vào trạng thái zombie. Zombie process vẫn chiếm một số tài nguyên trong bảng tiến trình (process table), mặc dù đã kết thúc. Nếu có nhiều tiến trình zombie, sẽ làm tốn bộ nhớ và tài nguyên hệ thống.

Để tránh tình trạng tiến trình con rơi vào trạng thái zombie, ta có thể sử dụng hàm `signal()` để đăng ký việc xử lý khi sự kiện SIGCHLD xảy ra.

Cú pháp hàm `signal()` như sau:

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

Tham số thứ nhất là tín hiệu cần xử lý, trong trường hợp này được thiết lập là **SIGCHLD**. Tham số thứ 2 là tên của hàm xử lý sự kiện. Khi có sự kiện xảy ra, hàm đã đăng ký sẽ được gọi. Chú ý cần gọi hàm `signal()` trước hàm `fork()`.

Ví dụ về hàm xử lý sự kiện:

```
#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;
}

```

Trong ví dụ trên, nội dung của hàm xử lý sự kiện đơn giản chỉ là chờ tiến trình con kết thúc và in ra dòng thông báo.

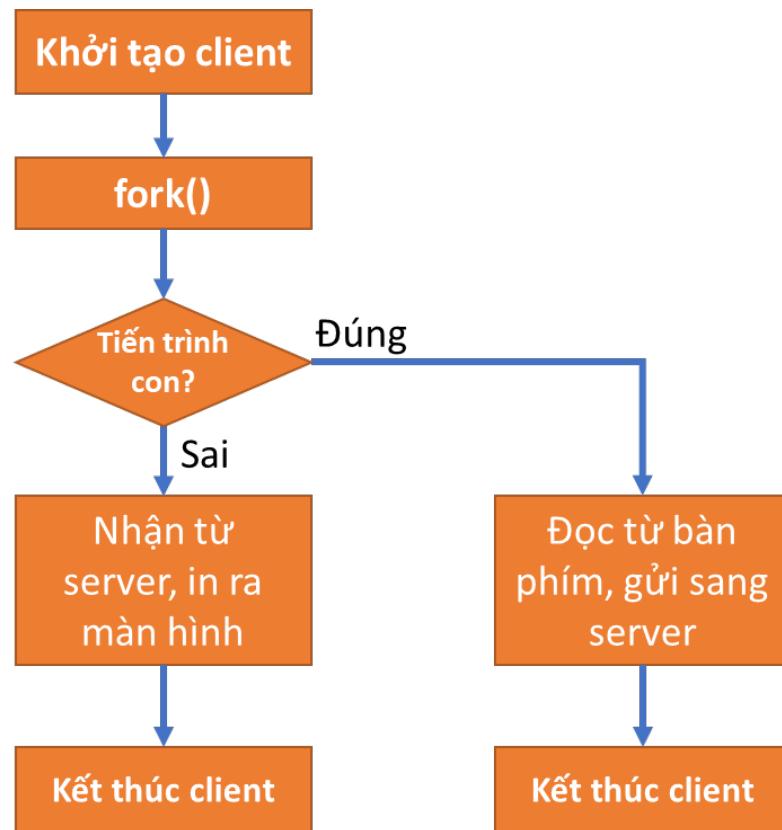
### **Áp dụng kỹ thuật đa tiến trình để xử lý nhiều kết nối trong ứng dụng server**

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, ta 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.

Ví dụ bên dưới tạo một chương trình client thực hiện 2 công việc đồng thời, vừa nhận dữ liệu từ server rồi in ra màn hình, vừa đọc dữ liệu từ bàn phím rồi gửi sang server. Các hàm vào ra này hoạt động đồng bộ, nếu thực hiện trong cùng một tiến trình thì sẽ ảnh hưởng lẫn nhau. Do vậy, trong ví dụ này, ta sẽ sử dụng 2 tiến trình để thực hiện 2 việc song song.

Ứng dụng hoạt động theo sơ đồ sau:



Mã nguồn của chương trình client:

```

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 #include <sys/signalf.h>
9
10 int main() {
11     // Khai báo socket
12     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
13
14     // Khai báo địa chỉ của server
15     struct sockaddr_in addr;
16     addr.sin_family = AF_INET;
17     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
18     addr.sin_port = htons(9000);

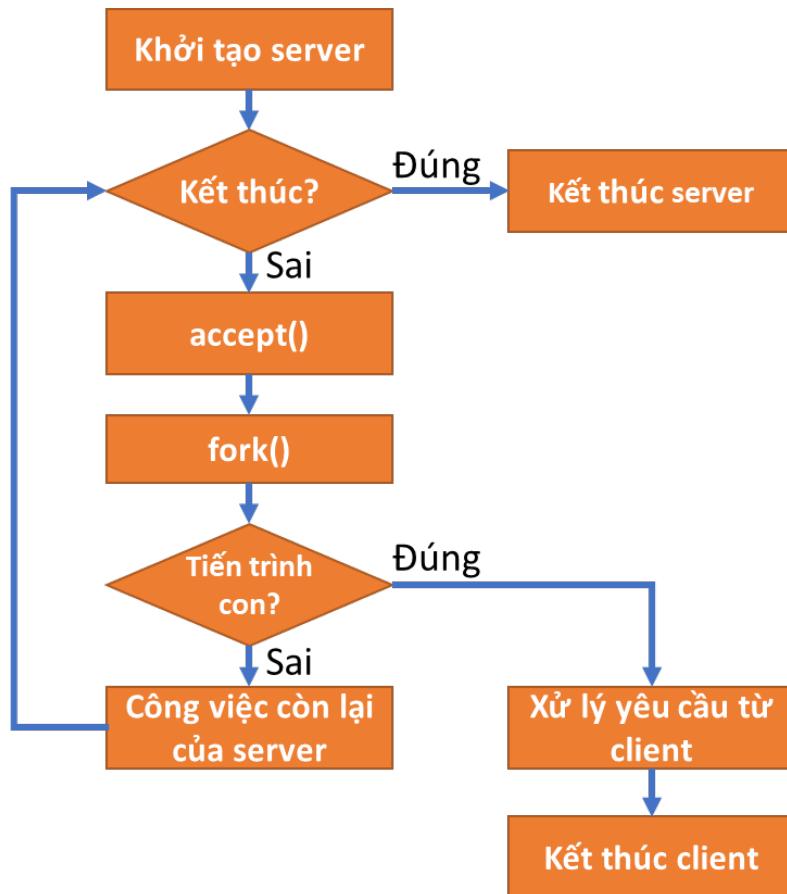
```

```

19
20     // Kết nối đến server
21     int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
22     if (res == -1) {
23         printf("Khong ket noi duoc den server!\n");
24         return 1;
25     }
26
27     char buf[256];
28
29     // Tạo tiến trình mới
30     int cid = fork();
31     if (cid == 0)
32     {
33         // Tiến trình con, nhận dữ liệu từ bàn phím
34         while (1)
35         {
36             fgets(buf, sizeof(buf), stdin);
37             send(client, buf, strlen(buf), 0);
38             if (strcmp(buf, "exit", 4) == 0)
39                 break;
40         }
41     }
42     else
43     {
44         // Tiến trình cha, nhận dữ liệu từ socket
45         while (1)
46         {
47             int ret = recv(client, buf, sizeof(buf), 0);
48             if (ret <= 0)
49                 break;
50             buf[ret] = 0;
51             printf("Received: %s\n", buf);
52         }
53     }
54
55     // Kết thúc, đóng socket
56     close(client);
57
58     // Dừng các tiến trình
59     killpg(0, SIGKILL);
60
61     return 0;
62 }
```

Ví dụ tiếp theo là một ứng dụng server có thể xử lý nhiều kết nối. Trong ví dụ này, tiến trình chính được sử dụng để quản lý việc chấp nhận các kết nối (xử lý kết quả của lệnh accept()). Mỗi khi một kết nối mới được chấp nhận thì một tiến trình con được tạo ra để xử lý yêu cầu của kết nối đó.

Server hoạt động theo sơ đồ sau:



Mã nguồn của chương trình server:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <sys/signal.h>
9 #include <sys/wait.h>
  
```

```
10 // Hàm xử lý tín hiệu SIGCHLD
11 void signalHandler(int signo)
12 {
13     pid_t pid;
14     int stat;
15     printf("signo = %d\n", signo);
16     pid = wait(&stat);
17     printf("child %d terminated.\n", pid);
18     return;
19 }
20
21
22 int main()
23 {
24     // Hiển thị số tiến trình tối đa có thể tạo
25     // struct rlimit lim;
26     // getrlimit(RLIMIT_NPROC, &lim);
27     // printf("Soft limit: %ld\n", lim.rlim_cur);
28     // printf("Hard limit: %ld\n", lim.rlim_max);
29
30     // Tạo socket chờ kết nối
31     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
32
33     // Khai báo cấu trúc địa chỉ của server
34     struct sockaddr_in addr;
35     addr.sin_family = AF_INET;
36     addr.sin_addr.s_addr = htonl(INADDR_ANY);
37     addr.sin_port = htons(9000);
38
39     // Gắn địa chỉ với socket và chuyển sang chờ kết nối
40     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
41     listen(listener, 5);
42
43     pid_t pid;
44
45     // Đăng ký xử lý tín hiệu SIGCHLD
46     signal(SIGCHLD, signalHandler);
47
48     while (1)
49     {
50         printf("Waiting for new client\n");
51         int client = accept(listener, NULL, NULL);
52         printf("New client accepted: %d\n", client);
53
54         // Tạo tiến trình cho kết nối mới
```

```

55     if ((pid = fork()) == 0)
56     {
57         // Trong tiến trình con
58
59         // Đóng socket listener vì không dùng đến
60         close(listener);
61
62         // Nhận và xử lý dữ liệu
63         char buf[256];
64         while (1)
65         {
66             int ret = recv(client, buf, sizeof(buf), 0);
67             if (ret <= 0)
68                 break;
69             buf[ret] = 0;
70             printf("Received from client %d: %s\n", client, buf);
71             send(client, buf, strlen(buf), 0);
72         }
73
74         // Đóng kết nối
75         close(client);
76
77         // Kết thúc tiến trình con
78         exit(0);
79     }
80
81     // Trong tiến trình cha, đóng socket client do không dùng đến
82     close(client);
83 }
84
85 return 0;
86 }
```

### Kỹ thuật 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 tiến trình, mỗi tiến trình sẽ thực hiện lặp lại các công việc: chờ kết nối, xử lý yêu cầu và trả kết quả cho client. Mỗi tiến trình sẽ xử lý luân phiên cho một client.

Mã nguồn của chương trình preforking server:

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <sys/wait.h>
9
10 int main()
11 {
12     // Tạo socket
13     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
14
15     // Khai báo cấu trúc địa chỉ server
16     struct sockaddr_in addr;
17     addr.sin_family = AF_INET;
18     addr.sin_addr.s_addr = htonl(INADDR_ANY);
19     addr.sin_port = htons(9000);
20
21     // Gắn địa chỉ với socket
22     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
23     listen(listener, 5);
24
25     int num_processes = 8;
26     char buf[256];
27
28     // Tạo trước các tiến trình, mỗi tiến trình lặp lại công việc chấp nhận kết nối
29     // và xử lý yêu cầu của client
30     for (int i = 0; i < num_processes; i++)
31     {
32         if (fork() == 0)
33             while(1)
34         {
35             // Chờ kết nối
36             int client = accept(listener, NULL, NULL);
37             printf("New client accepted in process %d: %d\n", client, getpid());
38
39             // Chờ dữ liệu từ client
40             int ret = recv(client, buf, sizeof(buf), 0);
41             if (ret <= 0)
42                 continue;
43
44             // Xử lý dữ liệu, trả lại kết quả cho client
45             buf[ret] = 0;

```

```

44         printf("Received from client %d: %s\n", client, buf);
45         send(client, buf, strlen(buf), 0);
46
47         // Đóng kết nối
48         close(client);
49     }
50
51     // Đợi vô thời hạn, đảm bảo chương trình tiếp tục hoạt động
52     wait(NULL);
53
54     return 0;
55 }
```

### 3.4. Kỹ thuật đa luồng - 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

Khi làm việc với luồng, các hàm sau thường được sử dụng:

- **pthread\_create()**: tạo luồng mới
- **pthread\_join()**: đợi luồng kết thúc
- **pthread\_self()**: trả về ID của luồng gọi hàm
- **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 gọi hàm
- **pthread\_cancel()**: dừng luồng con

**Chú ý:** Các hàm này thuộc thư viện Pthread APIs, cần thêm tham số “-pthread” khi dịch chương trình (được thiết lập trong file tasks.json trong VSCode)

#### Hàm tạo luồng pthread\_create()

Cú pháp của hàm:

```
#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
)
```

Giải thích tham số:

- `pthread_t *thread`
  - Con trỏ đến biến kiểu `pthread_t`.
  - Sau khi gọi thành công, biến này chứa ID của luồng mới.
  - Dùng để tham chiếu khi muốn quản lý luồng (ví dụ: `pthread_join`, `pthread_cancel`).
- `const pthread_attr_t *attr`
  - Cấu trúc thuộc tính của luồng.
  - Cho phép thiết lập các đặc tính như: kích thước stack, độ ưu tiên, detached/joinable...
  - Nếu truyền `NULL` → dùng giá trị mặc định (luồng joinable, stack mặc định).
- `void *(*start_routine)(void *)`
  - Con trỏ tới hàm mà luồng sẽ chạy.
  - Hàm này phải có dạng:

```
void* function\_name(void *arg);
```
  - Khi luồng được tạo, nó sẽ bắt đầu chạy từ đây.
  - Có thể trả về `NULL` hoặc con trỏ dữ liệu (có thể lấy bằng `pthread_join`).
- `void *arg`
  - Tham số truyền vào hàm `start_routine`.
  - Kiểu `void*` để có thể truyền bất kỳ kiểu dữ liệu nào (struct, int\*, char\*, ...).

- Nếu không cần tham số thì truyền NULL.

Giá trị trả về của hàm:

- Trả về 0 nếu luồng được tạo thành công.
- Trả về mã lỗi (khác 0) nếu thất bại.

Ví dụ về tạo luồng:

```
#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 đợi luồng kết thúc pthread\_join()

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.

Cú pháp của hàm:

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

Giải thích các tham số:

- pthread\_t thread
  - ID của luồng mà ta muốn chờ.
  - ID này thường lấy từ kết quả của pthread\_create.
- void \*\*retval
  - Địa chỉ của một con trỏ để lưu giá trị trả về từ hàm mà luồng đã chạy (start\_routine).
  - Nếu không quan tâm kết quả, thì truyền giá trị NULL.

Ví dụ: nếu hàm luồng trả về return (void\*) 42; thì khi join, \*retval sẽ nhận giá trị (void\*) 42.

Ví dụ mã nguồn minh họa sử dụng hàm pthread\_join() để đợi luồng con 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);
    }
}
```

Trong ví dụ trên, luồng chính khởi tạo một luồng con. Luồng con hoạt động trong 20 giây rồi kết thúc. Luồng chính sau khi tạo luồng con thì đợi luồng con kết thúc rồi mới kết thúc chương trình.

### 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

Đây là lỗi phổ biến trong lập trình đa luồng. Những lỗi này thường không rõ ràng, khó gỡ lỗi.

Hãy xem xét ví dụ sau:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NLOOP 10000
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;
    }
}
```

}

Trong ví dụ trên, ta có 2 luồng cùng truy nhập và thay đổi giá trị của biến toàn cục **counter**. Ở mỗi luồng, một vòng lặp thực hiện thao tác truy nhập giá trị hiện tại của biến counter, hiển thị và tăng thêm 1 vào giá trị của counter. Xung đột xảy ra ở đây là 2 luồng cùng đọc và ghi vào giá trị của biến toàn cục counter. Giá trị kỳ vọng của counter sau khi chạy xong chương trình là 20000 (10000 lần tăng cho mỗi luồng), tuy nhiên thực tế kết quả chạy lại không đạt được giá trị này, và có thể khác nhau với mỗi lần chạy thử.

Kết quả chạy chương trình như sau:

```

128229096548032: 9997
128229096548032: 9998
128229096548032: 9999
128229096548032: 10000
128229096548032: 10001
128229096548032: 10002
128229096548032: 10003
128229096548032: 10004
128229096548032: 10005
128229096548032: 10006
128229096548032: 10007
128229096548032: 10008
128229096548032: 10009
128229096548032: 10010
128229096548032: 10011
128229096548032: 10012
128229096548032: 10013
128229096548032: 10014
128229096548032: 10015
128229096548032: 10016
128229096548032: 10017
network_programming →

```

Một cách đơn giản để tránh xung đột là hạn chế việc dùng chung các biến toàn cục, thay bằng sử dụng các biến cục bộ trong hàm thực thi của luồng.

Nếu vẫn cần phải sử dụng các tài nguyên dùng chung, ta có thể sử dụng kỹ thuật Mutex (Mutual Exclusion).

**Mutex (Mutual Exclusion)** là một cơ chế đồng bộ hóa trong lập trình đa luồng, dùng để ngăn chặn nhiều luồng cùng lúc truy cập vào vùng tài nguyên chung (critical section), ví dụ: biến toàn cục, file, socket, hoặc vùng nhớ chia sẻ.

Ý tưởng cơ bản của cơ chế này như sau:

- Khi một luồng muốn sử dụng tài nguyên chung → nó phải khóa mutex (lock).
- Trong thời gian mutex đang bị khóa → các luồng khác không thể khóa mutex đó, chúng phải chờ cho đến khi mutex được mở khóa (unlock).

- Khi luồng đã dùng xong tài nguyên → nó mở khóa mutex (unlock) để luồng khác có thể tiếp tục.

Như vậy, mutex đảm bảo chỉ có một luồng duy nhất được truy cập tài nguyên tại một thời điểm.

Các hàm API cơ bản:

```
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
=> Khởi tạo 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
int pthread_mutex_unlock (pthread_mutex_t *mutex)
=> Mở khóa biến mutex
```

Ví dụ trên được cập nhật lại như sau:

```
#include <stdio.h>
#include <pthread.h>
#define NLOOP 10000
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);
    }
}

```

Kết quả chạy thử:

```

Ubuntu
139626522080960: 19980
139626522080960: 19981
139626522080960: 19982
139626522080960: 19983
139626522080960: 19984
139626522080960: 19985
139626522080960: 19986
139626522080960: 19987
139626522080960: 19988
139626522080960: 19989
139626522080960: 19990
139626522080960: 19991
139626522080960: 19992
139626522080960: 19993
139626522080960: 19994
139626522080960: 19995
139626522080960: 19996
139626522080960: 19997
139626522080960: 19998
139626522080960: 19999
139626522080960: 20000
network_programming →

```

Trong trường hợp này, ta thấy chương trình trả về kết quả đúng như mong đợi.

### Ứng dụng client sử dụng kỹ thuật đa luồng

Cũng giống như ví dụ client sử dụng kỹ thuật đa tiến trình, để xử lý đồng bộ giữa các hàm vào ra. Một luồng mới được tạo chạy đồng thời với luồng chính. Một luồng thực hiện việc nhận dữ liệu từ server rồi in ra màn hình, luồng còn lại đọc dữ liệu từ bàn phím rồi gửi sang server.

Mã nguồn ứng dụng client:

---

```

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 <pthread.h>
8 #include <arpa/inet.h>
9
10 void* thread_proc(void *arg);
11
12 int main()
13 {
14     // Tạo socket kết nối đến server
15     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
16
17     // Khai báo cấu trúc địa chỉ server
18     struct sockaddr_in addr;
19     addr.sin_family = AF_INET;
20     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
21     addr.sin_port = htons(9000);
22
23     // Kết nối đến server
24     if (connect(client, (struct sockaddr *)&addr, sizeof(addr)))
25     {
26         printf("Khong the ket noi den server.\n");
27         return 1;
28     }
29
30     // Tạo luồng mới
31     pthread_t thread_id;
32     if (pthread_create(&thread_id, NULL, thread_proc, (void *)&client))
33     {
34         printf("Khong the tao luong!\\n");
35         return 1;
36     }
37
38     // Chuyển luồng sang chế độ tự giải phóng
39     pthread_detach(thread_id);
40
41     char buf[256];
42
43     // Trong luồng chính, chờ dữ liệu từ bàn phím
44     while (1)
45     {
46         fgets(buf, sizeof(buf), stdin);
47         send(client, buf, strlen(buf), 0);
48         if (strcmp(buf, "exit", 4) == 0)
49             break;
```

```

50     }
51
52     close(client);
53     return 0;
54 }
55
56 void* thread_proc(void *arg)
57 {
58     // Trong luồng con, chờ dữ liệu từ socket
59
60     printf("child thread created.\n");
61     int client = *(int *)arg;
62     char buf[256];
63     while (1)
64     {
65         int len = recv(client, buf, sizeof(buf), 0);
66         if (len <= 0)
67             break;
68         buf[len] = 0;
69         printf("%s\n", buf);
70     }
71     close(client);
72     printf("child thread finished.\n");
73     return 0;
74 }
```

---

Ứng dụng server cũng hoạt động tương tự như khi sử dụng kỹ thuật đa tiến trình. Tuy nhiên trong trường hợp này, mỗi khi chấp nhận một kết nối mới thì ta sẽ tạo một luồng để xử lý việc truyền nhận dữ liệu với kết nối đó.

Mã nguồn ứng dụng server:

```

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 <pthread.h>
8 #include <sys/resource.h>
9
10 void* thread_proc(void *arg);
11
```

```
12 int main()
13 {
14     // Hiển thị số luồng tối đa có thể tạo
15     // struct rlimit lim;
16     // getrlimit(RLIMIT_NPROC, &lim);
17     // printf("Soft limit: %ld\n", lim.rlim_cur);
18     // printf("Hard limit: %ld\n", lim.rlim_max);
19
20     // Tạo socket chờ kết nối
21     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
22
23     // Khai báo cấu trúc địa chỉ server chờ ở cổng 9000
24     struct sockaddr_in addr;
25     addr.sin_family = AF_INET;
26     addr.sin_addr.s_addr = htonl(INADDR_ANY);
27     addr.sin_port = htons(9000);
28
29     // Gắn địa chỉ với socket và chuyển sang trạng thái chờ kết nối
30     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
31     listen(listener, 5);
32
33     pthread_t thread_id;
34
35     while (1)
36     {
37         printf("Waiting for new client ...\\n");
38         // Chấp nhận kết nối mới
39         int client = accept(listener, NULL, NULL);
40         if (client == -1)
41             continue;
42         printf("New client connected: %d\\n", client);
43
44         // Tạo luồng để xử lý yêu cầu từ client
45         int ret = pthread_create(&thread_id, NULL, thread_proc, (void *)&client);
46         if (ret != 0)
47         {
48             printf("Could not create thread!\\n");
49         }
50
51         // Yêu cầu luồng tự giải phóng khi kết thúc
52         pthread_detach(thread_id);
53         // Ưu tiên luồng mới tạo bắt đầu chạy
54         sched_yield();
55     }
56 }
```

```

57     return 0;
58 }
59
60 void* thread_proc(void *arg)
61 {
62     // Luồng xử lý yêu cầu từ client
63
64     printf("child thread created.\n");
65     int client = *(int *)arg;
66     char buf[2048];
67     while (1)
68     {
69         int len = recv(client, buf, sizeof(buf), 0);
70         if (len <= 0)
71             break;
72         buf[len] = 0;
73         printf("%s", buf);
74     }
75     close(client);
76     printf("child thread finished.\n");
77 }
```

---

### Kỹ thuật 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.

Mã nguồn ứng dụng server sử dụng kỹ thuật prethreading:

```

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 <pthread.h>
8
9 void* thread_proc(void *arg);
10
11 int main()
12 {
13     // Tạo socket chờ kết nối
```

```
14     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
15
16     // Khai báo cấu trúc địa chỉ server
17     struct sockaddr_in addr;
18     addr.sin_family = AF_INET;
19     addr.sin_addr.s_addr = htonl(INADDR_ANY);
20     addr.sin_port = htons(9000);
21
22     // Gắn địa chỉ với socket
23     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
24     listen(listener, 5);
25
26     int num_threads = 8;
27     pthread_t thread_id;
28
29     // Tạo trước các luồng
30     for (int i = 0; i < num_threads; i++)
31     {
32         int ret = pthread_create(&thread_id, NULL, thread_proc, &listener);
33         if (ret != 0)
34             printf("Could not create new thread.\n");
35         sched_yield();
36     }
37
38     // Đợi vô thời hạn, đảm bảo chương trình tiếp tục hoạt động
39     pthread_join(thread_id, NULL);
40     return 0;
41 }
42
43 void* thread_proc(void *arg)
44 {
45     int listener = *(int *)arg;
46     char buf[256];
47
48     while (1)
49     {
50         // Chờ kết nối
51         int client = accept(listener, NULL, NULL);
52         printf("New client %d accepted in thread %ld with pid %d\n",
53                client,
54                pthread_self(), getpid());
55
56         // Chờ dữ liệu từ client
57         int ret = recv(client, buf, sizeof(buf), 0);
58         if (ret <= 0)
59             continue;
```

```

58
59     // Xử lý dữ liệu, trả lại kết quả cho client
60     buf[ret] = 0;
61     printf("Received from client %d: %s\n", client, buf);
62     send(client, buf, strlen(buf), 0);
63
64     // Đóng kết nối
65     close(client);
66 }
67
68 return NULL;
69 }
```

### 3.5. Kỹ thuật ghép kênh - Multiplexing

Multiplexing trong ngữ cảnh socket là cơ chế cho phép một tiến trình có thể theo dõi và quản lý nhiều socket cùng lúc mà không cần phải tạo một tiến trình/luồng riêng cho từng kết nối.

Thay vì chặn (block) ở một socket duy nhất, tiến trình có thể chờ sự kiện trên nhiều socket (ví dụ: có dữ liệu đến, sẵn sàng ghi, kết nối mới). Điều này rất quan trọng khi xây dựng server đồng thời (concurrent server) mà vẫn chỉ dùng một tiến trình hoặc một vài luồng. Tuy nhiên do chỉ hoạt động trên một luồng, nên chỉ có duy nhất một client được phục vụ tại một thời điểm.

Các ứng dụng server thường sử dụng mảng để quản lý các socket client, và dùng vòng lặp để kiểm tra trạng thái của các client này.

Các cơ chế multiplexing trong socket API:

- select()
  - API chuẩn, có mặt trên hầu hết hệ thống UNIX.
  - Cho phép theo dõi tập hợp socket để xem socket nào: Có dữ liệu đến (readable), Gửi được dữ liệu (writable), Có lỗi.
  - Hạn chế: số socket tối đa thường bị giới hạn (FD\_SETSIZE = 1024).
- poll()
  - Cải tiến hơn select().
  - Không bị giới hạn số lượng socket bởi FD\_SETSIZE.
  - Vẫn phải duyệt qua toàn bộ danh sách socket mỗi lần kiểm tra → chưa tối ưu cho số lượng socket cực lớn.

- epoll (Linux) / kqueue (BSD, macOS) / IOCP (Windows)
  - Thiết kế để xử lý hàng chục nghìn socket đồng thời.
  - Cơ chế event-driven: chỉ thông báo khi có sự kiện mới → hiệu quả hơn nhiều so với select() và poll().

### Hàm select()

Hàm select() hoạt động ở chế độ đồng bộ, hàm sẽ dừng luồng và đợi cho đến khi sự kiện của một trong các socket đã đăng ký xảy ra. Trong thời gian này, tiến trình không làm gì khác ngoài việc chờ kết quả trả về của hàm.

Hàm cho phép tiến trình chờ nhiều sự kiện vào ra cùng lúc và trả về kết quả khi một trong các sự kiện xảy ra, bao gồm:

- Có socket sẵn sàng đọc hoặc ghi dữ liệu.
- Có sự kiện ngoại lệ.
- Hết thời gian chờ.

Một lời gọi select() có thể theo dõi hàng trăm hoặc hàng nghìn socket cùng lúc, giúp server có thể xử lý nhiều client mà không cần tạo nhiều luồng hoặc tiến trình.

Tất cả các socket được quản lý trong một vòng lặp duy nhất, lập trình viên chỉ cần gọi **select()** và xử lý lần lượt các socket sẵn sàng, dễ kiểm soát hơn so với đa luồng.

Mặc dù select() theo dõi nhiều socket, nhưng khi trả về, chương trình sẽ xử lý từng socket một cách tuần tự, tại một thời điểm, chỉ xử lý được một sự kiện trên một socket, không song song thực sự. Điều này có thể trở thành hạn chế khi số lượng kết nối quá lớn hoặc khi một socket xử lý quá lâu.

### Cú pháp hàm select()

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

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

Giải thích các tham số của hàm:

- nfds

- Là giá trị lớn nhất của file descriptor cần theo dõi + 1.
- Ví dụ: nếu bạn muốn theo dõi các socket có fd = 3, 4, 5 → thì nfds = 6.
- Mục đích: giúp kernel biết phạm vi cần quét, tránh kiểm tra cả dải fd vô hạn.

- readfds

- Con trả đến một tập hợp fd\_set chứa các socket/file descriptor cần kiểm tra sẵn sàng để đọc. Bao gồm:
  - \* Socket có dữ liệu đến (recv() không block).
  - \* Socket lắng nghe (listening socket) có kết nối mới (accept() không block).
- Nếu không quan tâm, truyền giá trị NULL.

- writefds

- Con trả đến một tập hợp fd\_set chứa các socket/file descriptor cần kiểm tra sẵn sàng để ghi. Bao gồm:
  - \* Socket có thể gửi dữ liệu ngay mà không bị block trong send().
  - \* Socket non-blocking vừa hoàn thành kết nối TCP.
- Nếu không quan tâm, truyền giá trị NULL.

- exceptfds

- Con trả đến một tập hợp fd\_set chứa các socket cần kiểm tra sự kiện ngoại lệ. Bao gồm:
  - \* Out-of-band data (dữ liệu khẩn TCP – OOB).
  - \* Một số lỗi kết nối.
- Ít dùng trong lập trình thực tế, thường để NULL.

- timeout - Con trả đến struct timeval, chỉ định thời gian tối đa select() sẽ chờ.

### Cấu trúc timeval

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

Cấu trúc này được sử dụng để xác định thời gian chờ của hàm select(). Mỗi khi hàm select() trả về kết quả thì biến cấu trúc bị khởi tạo lại. Có 3 trường hợp sử dụng biến cấu trúc này:

- timeout = NULL => Hàm select() chờ vô hạn cho đến khi có sự kiện.
- timeout = 0, 0 => Hàm select() kiểm tra và trả về kết quả ngay.
- timeout = X, Y => Hàm select() chờ trong X giây và Y micro giây.

### Giá trị trả về của hàm

- > 0: số lượng socket sẵn sàng (có sự kiện).
- = 0: không có sự kiện nào trong khoảng thời gian chờ timeout.
- -1: xảy ra lỗi khi gọi hàm (có thể kiểm tra mã lỗi bằng **errno**)

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 **writefds** 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.

**Chú ý:** Các tập **readfds**, **writefds**, **exceptfds** có thể được gán giá trị NULL, nhưng không thể cả 3 tập đều NULL.

### Cấu trúc fd\_set

**fd\_set** là kiểu dữ liệu cấu trúc dùng để chứa các mô tả file (bao gồm socket, file, ...) được sử dụng để theo dõi sự kiện bằng hàm **select()**.

Cấu trúc được định nghĩa như sau:

```

typedef long int __fd_mask;
#define __NFDBITS (8 * (int) sizeof (__fd_mask))
#define __FD_SETSIZE 1024

typedef struct
{
    __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;

```

Cấu trúc này lưu trữ một tập hợp các bit, mỗi bit ứng với một mô tả file được quản lý. Giá trị của mô tả file cũng chính là thứ tự của bit tương ứng.

Để thực hiện được việc lưu trữ này, cấu trúc định nghĩa một mảng các số nguyên (kiểu long int), mỗi phần tử của mảng là một giá trị 64 bit quản lý trạng thái của 64 mô tả file. Số lượng phần tử của mảng được xác định bởi biểu thức **\_\_FD\_SETSIZE / \_\_NFDBITS** trong đó **\_\_FD\_SETSIZE** là số mô tả tối đa mà **fd\_set** quản lý được (mặc định là 1024) và **\_\_NFDBITS** là kích thước của một phần tử mảng (mặc định là 64).

Như vậy các mô tả có giá trị từ 0 đến 63 được quản lý bởi phần tử đầu tiên của mảng (chỉ số 0), các mô tả có giá trị từ 64 đến 127 được quản lý bởi phần tử thứ hai (chỉ số 1), và cứ tiếp tục như vậy.

**Các macro được sử dụng để thao tác với tập fd\_set** Thư viện cung cấp 4 macro như sau:

- Khởi tạo rỗng

```

FD_ZERO(fd_set *set);
=> Xóa toàn bộ các bit, không fd nào được chọn.

```

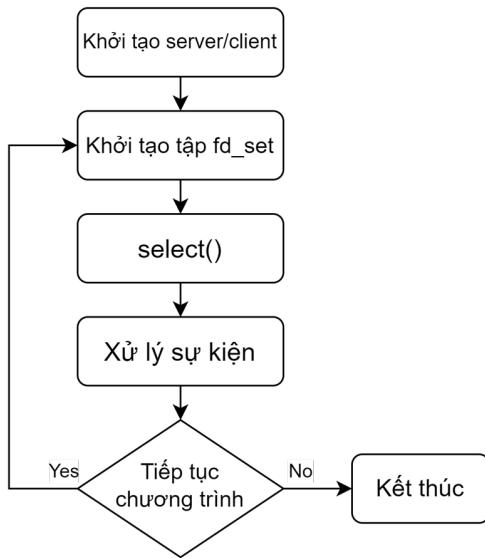
- Thêm fd vào tập

```

FD_SET(int fd, fd_set *set);
=> Đặt bit tương ứng với fd thành 1.

```

- Xóa fd khỏi tập



Hình 3.6 Sơ đồ khái niệm client sử dụng hàm select()

```

FD_CLR(int fd, fd_set *set);
=> Đặt bit tương ứng với fd thành 0.
  
```

- Kiểm tra fd có trong tập không

```

FD_ISSET(int fd, fd_set *set);
=> Trả về khác 0 nếu fd có trong tập, ngược lại trả về 0.
  
```

**Chú ý:** Hàm **select()** thay đổi giá trị của tập **fd\_set** sau khi trả về kết quả để chỉ ra socket nào có sự kiện (socket nào có sự kiện thì bit tương ứng là 1, các socket khác bit tương ứng là 0, mặc dù trước đó có thể đã được thiết lập là 1). Do vậy cần khởi tạo lại tập **fd\_set** nếu gọi **select()** nhiều lần.

Cấu trúc chương trình dùng hàm **select()** thường hoạt động theo sơ đồ 3.6.

### Chương trình client xử lý đồng bộ bằng hàm select()

Dưới đây là một ví dụ về chương trình client thực hiện hai việc đồng bộ gồm nhận dữ liệu từ server (dùng lệnh **recv()**) rồi in ra màn hình và đọc dữ liệu từ bàn phím (dùng lệnh **fgets()**) rồi truyền sang server.

Mã nguồn minh họa cho ứng dụng client như sau:

```
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 báo socket client
11     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
12
13     // Khai báo địa chỉ 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);
18
19     // Thực hiện kết nối đến server
20     int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
21     if (res == -1) {
22         printf("Không kết nối được đến server!\n");
23         return 1;
24     }
25
26     // Khai báo tập fdset
27     fd_set fdread;
28     char buf[256];
29
30     while (1)
31     {
32         // Khởi tạo lại tập fdread
33         FD_ZERO(&fdread);
34
35         // Gắn các descriptor vào tập fdread
36         FD_SET(STDIN_FILENO, &fdread);
37         FD_SET(client, &fdread);
38
39         // Chờ đến khi sự kiện xảy ra
40         int ret = select(client + 1, &fdread, NULL, NULL, NULL);
41         if (ret == -1)
42         {
43             perror("select() failed");
44             break;
```

```

45 }
46
47     // Kiểm tra sự kiện có dữ liệu từ bàn phím
48     if (FD_ISSET(STDIN_FILENO, &fdread))
49     {
50         fgets(buf, sizeof(buf), stdin);
51         send(client, buf, strlen(buf), 0);
52
53         // Nếu nhập "exit" thì kết thúc
54         if (strncmp(buf, "exit", 4) == 0)
55             break;
56     }
57
58     // Kiểm tra sự kiện có dữ liệu truyền đến qua socket
59     if (FD_ISSET(client, &fdread))
60     {
61         ret = recv(client, buf, sizeof(buf), 0);
62
63         // Nếu ngắt kết nối thì kết thúc
64         if (ret <= 0)
65             break;
66
67         buf[ret] = 0;
68         printf("Received: %s\n", buf);
69     }
70 }
71
72     // Kết thúc, đóng socket
73     close(client);
74
75     return 0;
76 }
```

Ví dụ tiếp theo là ứng dụng server có khả năng chấp nhận và xử lý nhiều kết nối. Trong phiên bản này, một mảng được sử dụng để quản lý các client đã kết nối. Sau mỗi lần gọi hàm `select()`, tất cả các phần tử của mảng được kiểm tra để biết socket nào có sự kiện.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <sys/select.h>
```

```
6 #include <unistd.h>
7 #include <netdb.h>
8 #include <string.h>
9 #include <arpa/inet.h>
10
11 #define MAX_CLIENTS FD_SETSIZE
12
13 // Xóa client ra khỏi mảng
14 void removeClient(int *clients, int *numClients, int clientIndex)
15 {
16     if (clientIndex < *numClients - 1)
17         clients[clientIndex] = clients[*numClients - 1];
18     *numClients = *numClients - 1;
19 }
20
21 int main()
22 {
23     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
24
25     struct sockaddr_in addr;
26     addr.sin_family = AF_INET;
27     addr.sin_addr.s_addr = INADDR_ANY;
28     addr.sin_port = htons(9000);
29
30     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
31     listen(listener, 5);
32
33     fd_set fdread;
34
35     // Mảng clients lưu các socket đã được chấp nhận
36     // Sử dụng trong việc thăm dò sự kiện
37     int clients[MAX_CLIENTS];
38     int numClients = 0;
39
40     // Cấu trúc thời gian đợi
41     struct timeval tv;
42     tv.tv_sec = 5;
43     tv.tv_usec = 0;
44
45     char buf[2048];
46
47     while (1)
48     {
49         // Khởi tạo lại tập fdread
50         FD_ZERO(&fdread);
```

```
51
52     // Gắn các socket listener và clients vào tập fdread
53     // maxdp lưu giá trị descriptor lớn nhất
54     FD_SET(listener, &fdread);
55     int maxdp = listener;
56     for (int i = 0; i < numClients; i++)
57     {
58         FD_SET(clients[i], &fdread);
59         if (clients[i] > maxdp)
60             maxdp = clients[i];
61     }
62
63     // Khởi tạo lại giá trị cấu trúc thời gian
64     tv.tv_sec = 5;
65     tv.tv_usec = 0;
66
67     // Chờ đến khi sự kiện xảy ra hoặc hết giờ
68     printf("Waiting for new event.\n");
69     int ret = select(maxdp + 1, &fdread, NULL, NULL, &tv);
70     if (ret < 0)
71     {
72         printf("select() failed.\n");
73         return 1;
74     }
75     if (ret == 0)
76     {
77         printf("Timed out.\n");
78         continue;
79     }
80
81     // Thăm dò sự kiện có yêu cầu kết nối
82     if (FD_ISSET(listener, &fdread))
83     {
84         int client = accept(listener, NULL, NULL);
85
86         if (numClients < MAX_CLIENTS)
87         {
88             printf("New client connected %d\n", client);
89             // Lưu vào mảng để thăm dò sự kiện
90             clients[numClients] = client;
91             numClients++;
92         }
93         else
94         {
95             // Đã vượt quá số kết nối tối đa
```

```

96         close(client);
97     }
98 }
99
100    // Thăm dò sự kiện có dữ liệu truyền đến các socket client
101   for (int i = 0; i < numClients; i++)
102     if (FD_ISSET(clients[i], &fdread))
103     {
104       ret = recv(clients[i], buf, sizeof(buf), 0);
105       if (ret <= 0)
106       {
107         printf("Client %d disconnected\n", clients[i]);
108         // Xóa client khỏi mảng
109         removeClient(clients, &numClients, i);
110         i--;
111         continue;
112       }
113       buf[ret] = 0;
114       printf("Received data from client %d: %s\n", clients[i], buf);
115     }
116   }
117
118   return 0;
119 }
```

Một phiên bản khác của ứng dụng server được trình bày ở dưới. Điểm khác biệt của phiên bản này là không dùng mảng để lưu trữ các client đã kết nối. Sau mỗi lần gọi hàm **select()** thì sử dụng vòng lặp để kiểm tra tất cả các giá trị mô tả có thể có (từ 0 đến 1023).

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <sys/select.h>
6 #include <unistd.h>
7 #include <netdb.h>
8 #include <string.h>
9 #include <arpa/inet.h>
10
11 int main()
12 {
13   int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
14
15     struct sockaddr_in addr;
16     addr.sin_family = AF_INET;
17     addr.sin_addr.s_addr = INADDR_ANY;
18     addr.sin_port = htons(9000);
19
20     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
21     listen(listener, 5);
22
23     // Khai báo tập fdread chứa các socket và tập fdtest để thăm dò sự kiện
24     fd_set fdread, fdtest;
25
26     // Cấu trúc thời gian đợi
27     struct timeval tv;
28     char buf[2048];
29
30     // Khởi tạo tập fdread
31     FD_ZERO(&fdread);
32
33     // Gắn socket listener vào tập fdread
34     FD_SET(listener, &fdread);
35
36     while (1)
37     {
38         // Tập fdtest để thăm dò sự kiện, giữ nguyên các socket trong tập fdread
39         fdtest = fdread;
40
41         // Khởi tạo lại giá trị cấu trúc thời gian
42         tv.tv_sec = 5;
43         tv.tv_usec = 0;
44
45         // Chờ đến khi sự kiện xảy ra hoặc hết giờ
46         printf("Waiting for new event.\n");
47         int ret = select(FD_SETSIZE, &fdtest, NULL, NULL, &tv);
48         if (ret < 0)
49         {
50             printf("select() failed.\n");
51             return 1;
52         }
53         if (ret == 0)
54         {
55             printf("Timed out.\n");
56             continue;
57         }
58 }
```

```

59     for (int i = 0; i < FD_SETSIZE; i++)
60         if (FD_ISSET(i, &fdtest))
61     {
62         if (i == listener)
63     {
64         // Socket listener có sự kiện yêu cầu kết nối
65         int client = accept(listener, NULL, NULL);
66
67         if (client < FD_SETSIZE)
68     {
69             printf("New client connected %d\n", client);
70
71             // Thêm vào tập fdread
72             FD_SET(client, &fdread);
73         }
74         else
75     {
76         // Đã vượt quá số kết nối tối đa
77         close(client);
78     }
79     }
80     else
81     {
82         // Socket client có sự kiện nhận dữ liệu
83         ret = recv(i, buf, sizeof(buf), 0);
84         if (ret <= 0)
85     {
86             printf("Client %d disconnected\n", i);
87             FD_CLR(i, &fdread);
88         }
89         else
90     {
91             buf[ret] = 0;
92             printf("Received data from client %d: %s\n", i, buf);
93         }
94     }
95 }
96
97     return 0;
98 }

```

### Hàm poll()

Hàm **poll()** thực hiện chức năng tương tự hàm **select()**: đợi trên một tập hợp

các mô tả cho đến khi các thao tác vào ra sẵn sàng.

Cú pháp của hàm poll() như sau:

```
#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
) //
```

Giải thích các tham số:

- \*fds - Mảng (con trỏ) các phần tử cấu trúc pollfd, mỗi phần tử mô tả một file descriptor (socket, file, ...) cần theo dõi. Cấu trúc pollfd sẽ được mô tả ở dưới.
- nfds - Số lượng phần tử của mảng fds
- timeout - Thời gian chờ tối đa tính theo mili giây. Có thể truyền một trong các giá trị sau:
  - > 0: thời gian chờ tối đa theo mili giây
  - = 0: không chờ, hàm kiểm tra và trả về kết quả ngay
  - -1: chờ vô hạn cho đến khi có sự kiện

Giá trị trả về của hàm:

- > 0: số lượng file descriptor có sự kiện xảy ra.
- = 0: hết thời gian chờ timeout mà không có sự kiện xảy ra.
- -1: xảy ra lỗi khi gọi hàm (có thể kiểm tra mã lỗi bằng **errno**).

### 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 về ứng dụng client vừa nhận dữ liệu từ server rồi in ra màn hình, vừa đọc dữ liệu từ bàn phím rồi truyền sang server:

```

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 #include <poll.h>
9
10 int main() {
11     // Khai báo socket
12     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
13
14     // Khai báo địa chỉ của server
15     struct sockaddr_in addr;
16     addr.sin_family = AF_INET;
17     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
18     addr.sin_port = htons(9000);
19
20     // Kết nối đến server
21     int res = connect(client, (struct sockaddr *)&addr, sizeof(addr));
22     if (res == -1) {
23         printf("Khong ket noi duoc den server!\n");
24         return 1;
25     }
26
27     // Mảng cấu trúc thăm dò
28     struct pollfd fds[2];
29
30     // Thêm descriptor của sự kiện bàn phím

```

```

31     fds[0].fd = STDIN_FILENO;
32     fds[0].events = POLLIN;
33
34     // Thêm descriptor của sự kiện socket
35     fds[1].fd = client;
36     fds[1].events = POLLIN;
37
38     char buf[256];
39
40     while (1)
41     {
42         // Chờ đến khi sự kiện xảy ra, không sử dụng timeout
43         int ret = poll(fds, 2, -1);
44
45         // Nếu sự kiện là có dữ liệu từ bàn phím
46         if (fds[0].revents & POLLIN)
47         {
48             fgets(buf, sizeof(buf), stdin);
49             send(client, buf, strlen(buf), 0);
50
51             // Nếu nhập "exit" thì kết thúc
52             if (strncmp(buf, "exit", 4) == 0)
53                 break;
54         }
55
56         // Nếu sự kiện là có dữ liệu từ socket
57         if (fds[1].revents & POLLIN)
58         {
59             ret = recv(client, buf, sizeof(buf), 0);
60             if (ret <= 0)
61                 break;
62             buf[ret] = 0;
63             printf("Received: %s\n", buf);
64         }
65     }
66
67     close(client);
68
69     return 0;
70 }
```

Ví dụ về ứng dụng server chấp nhận và xử lý nhiều kết nối sử dụng lệnh **poll()**:

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

```
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <arpa/inet.h>
9 #include <poll.h>
10 #include <sys/resource.h>
11
12 #define MAX_CLIENTS 1024
13
14 // Hàm xóa client ra khỏi mảng thăm dò
15 void removeClient(struct pollfd *fds, nfds_t *nfds, int index)
16 {
17     if (index < *nfds - 1)
18         fds[index] = fds[*nfds - 1];
19     *nfds = *nfds - 1;
20 }
21
22 int main()
23 {
24     // struct rlimit lim;
25     // getrlimit(RLIMIT_NOFILE, &lim);
26     // printf("Soft limit: %ld\n", lim.rlim_cur);
27     // printf("Hard limit: %ld\n", lim.rlim_max);
28
29     int listener = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
30
31     struct sockaddr_in addr;
32     addr.sin_family = AF_INET;
33     addr.sin_addr.s_addr = INADDR_ANY;
34     addr.sin_port = htons(9000);
35
36     bind(listener, (struct sockaddr *)&addr, sizeof(addr));
37     listen(listener, 5);
38
39     // Mảng thăm dò sự kiện
40     struct pollfd fds[MAX_CLIENTS];
41     nfds_t nfds = 1;
42
43     // Thêm socket listener vào mảng
44     fds[0].fd = listener;
45     fds[0].events = POLLIN;
46
```

```
47     char buf[2048];
48
49     while (1)
50     {
51         // Chờ sự kiện xảy ra hoặc hết giờ
52         printf("Waiting for new event.\n");
53         int ret = poll(fds, nfds, 5000);
54         if (ret < 0)
55         {
56             printf("poll() failed.\n");
57             return 1;
58         }
59         if (ret == 0)
60         {
61             printf("Timed out.\n");
62             continue;
63         }
64
65         // Nếu sự kiện có yêu cầu kết nối
66         if (fds[0].revents & POLLIN)
67         {
68             int client = accept(listener, NULL, NULL);
69
70             if (nfds < MAX_CLIENTS)
71             {
72                 printf("New client connected %d\n", client);
73                 // Lưu vào mảng để thăm dò
74                 fds[nfds].fd = client;
75                 fds[nfds].events = POLLIN;
76                 nfds++;
77             }
78             else
79             {
80                 // Đã vượt quá số kết nối tối đa
81                 close(client);
82             }
83         }
84
85         // Kiểm tra sự kiện dữ liệu đến socket
86         for (int i = 1; i < nfds; i++)
87             if (fds[i].revents & POLLIN)
88             {
89                 ret = recv(fds[i].fd, buf, sizeof(buf), 0);
90                 if (ret <= 0)
91                 {
```

```

92         printf("Client %d disconnected\n", fds[i].fd);
93         // Remove from clients
94         removeClient(fds, &nfds, i);
95         i--;
96         continue;
97     }
98     buf[ret] = 0;
99     printf("Received data from client %d: %s\n", fds[i].fd, buf);
100    }
101 }
102
103 return 0;
104 }
```

---

### 3.6. Bài tập

**Bài tập 3.1.** Sử dụng hàm select(), 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 chao” thì các client khác sẽ nhận được: “abc: xin chao” hoặc có thể thêm thời gian vào trước ví dụ: “2023/05/06 11:00:00PM abc: xin chao”.

**Bài tập 3.2.** Sử dụng hàm select(), 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.

**Bài tập 3.3.** Lập trình ứng dụng **chat\_server** và **telnet\_server** sử dụng hàm `poll()`.

**Bài tập 3.4.** Lập trình ứng dụng **telnet\_server** sử dụng kỹ thuật đa tiến trình.

**Bài tập 3.5.** Lập trình ứng dụng **chat\_server** sử dụng kỹ thuật đa luồng.

# Chương 4

## Thiết kế giao thức mạng

---

4.1. Khái niệm về giao thức . . . . .	104
4.2. Tìm hiểu một số giao thức phổ biến . . . . .	104
4.3. Thiết kế giao thức tự định nghĩa . . . . .	113
4.4. Bài tập . . . . .	117

---

### 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

#### 4.2.1. Giao thức HTTP

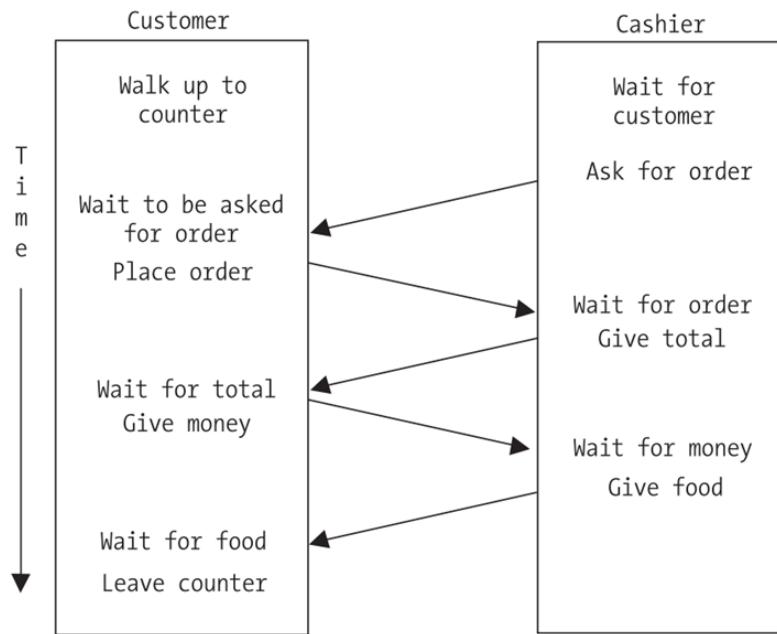
##### Định nghĩa giao thức HTTP

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.

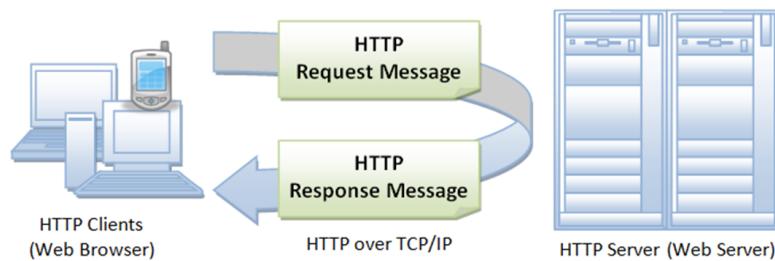
HTTP là một giao thức ứng dụng của bộ giao thức TCP/IP (các giao thức nền tảng cho Internet).

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.



Hình 4.1 Giao thức giữa khách hàng và thu ngân



Hình 4.2 Sơ đồ hoạt động giao thức HTTP

Giao thức HTTP gồm 2 đối tượng chính:

- HTTP Request: yêu cầu từ trình duyệt (client) gửi lên server.
- HTTP Response: phản hồi từ server về trình duyệt.

### HTTP Request

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:

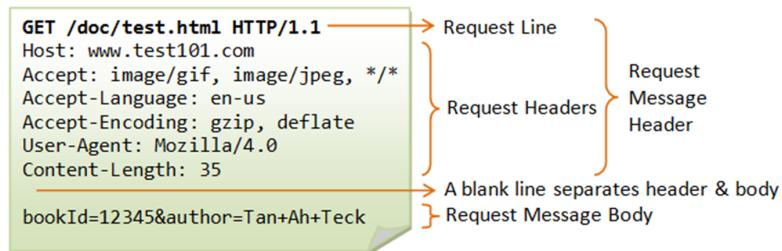
- Request-line = Phương thức + URI-Request + Phiên bản HTTP

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

Method	Hoạt động	Chú thích
<b>GET</b>	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
<b>POST</b>	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, ...	
<b>PUT</b>	Nếu URI đề cập đến một tài nguyên đã có, nó sẽ bị sửa đổi; nếu URI không trỏ đến một tài nguyên hiện có, thì máy chủ có thể tạo ra tài nguyên với URI đó.	
<b>DELETE</b>	Xóa bỏ tất cả các đại diện của tài nguyên được chỉ định bởi URI.	
<b>PATCH</b>	Áp dụng cho việc sửa đổi một phần của tài nguyên được xác định.	
...	...	...

Ví dụ về một HTTP request:



### HTTP Response

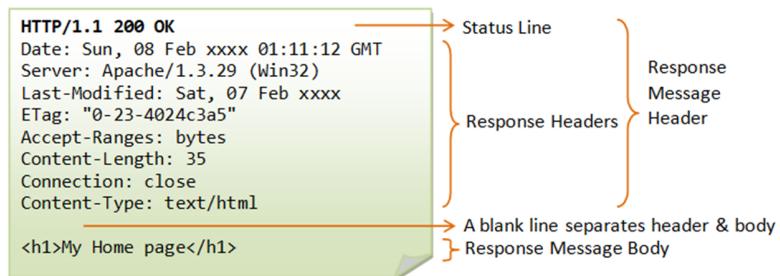
Cấu trúc của một HTTP response

- 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

Mã trạng thái được sử dụng để thông báo về kết quả khi nhận được yêu cầu và xử lý bên server cho client. Các kiểu mã trạng thái như sau:

- 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)

Ví dụ một HTTP response:



#### 4.2.2. Giao thức FTP

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. Giao thức được mô tả cụ thể trong tài liệu RFC959.

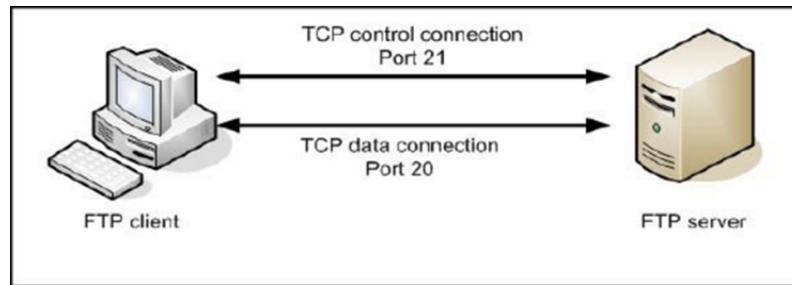
##### Mô hình hoạt động

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

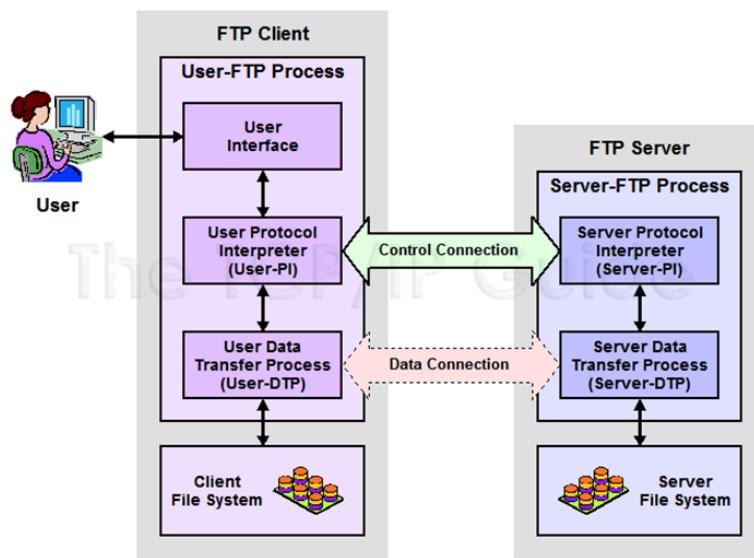
- Control connection:
  - Kết nối chính được tạo ra khi phiên làm việc được thiết lập
  - Được duy trì trong suốt phiên làm việc và chỉ cho các thông tin điều khiển đi qua ví dụ như lệnh và trả lời.
  - Không được sử dụng để gửi dữ liệu.
- Data connection:
  - Mỗi khi dữ liệu được gửi từ sever tới client 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.

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.

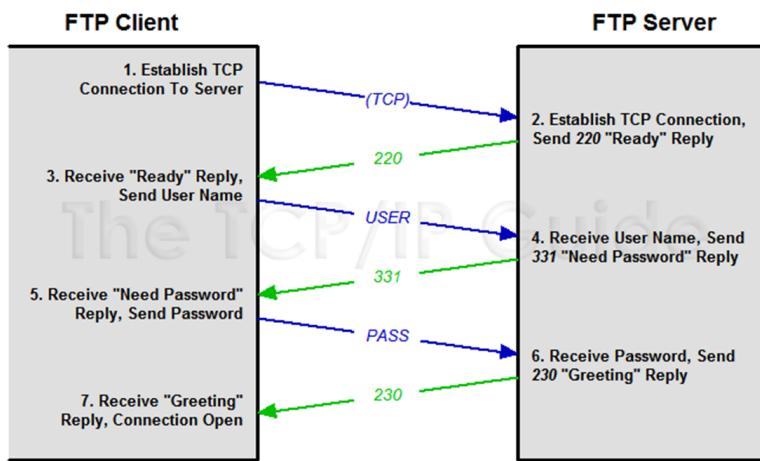


Hình 4.3 Mô hình hoạt động giao thức FTP



Hình 4.4 Mô hình hoạt động 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.

Chế độ chủ động: Client tạo kênh truyền dữ liệu

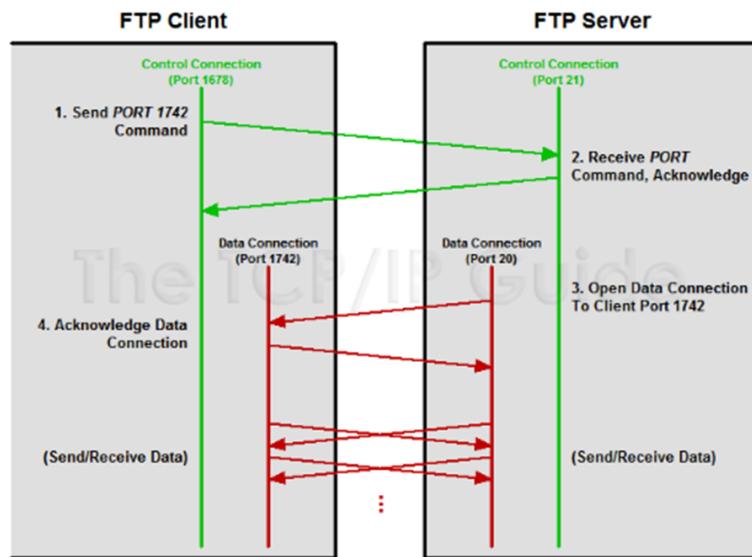


Figure 290: FTP Active Data Connection

Chế độ bị động: Server tạo kênh truyền dữ liệu

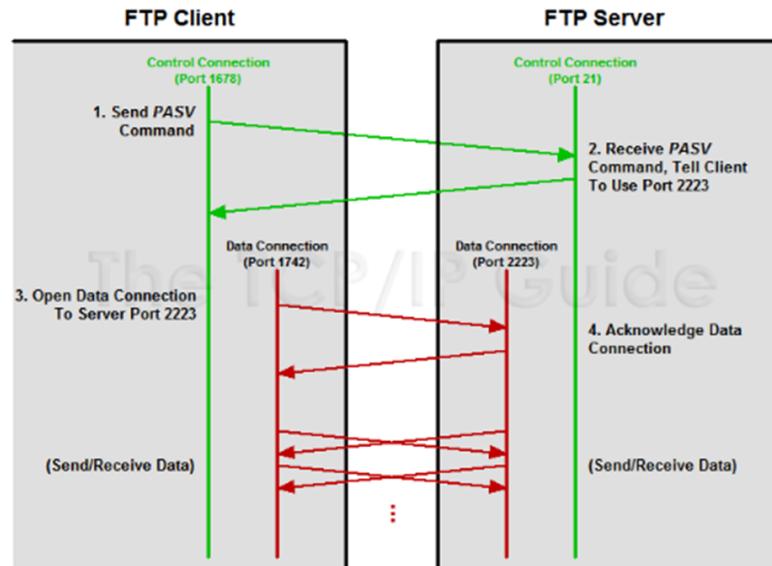


Figure 291: FTP Passive Data Connection

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

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/EPSV	Chuyển sang chế độ Passive
NOOP	Duy trì kết nối
QUIT	Ngắt kết nối

### Lập trình ứng dụng FTP client

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)

#### 4.2.3. Giao thức POP3

POP3 là viết tắt của Post Office Protocol phiên bản 3. Đây là 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. Giao thức được mô tả chi tiết trong tài liệu RFC1939.

Các trạng thái hoạt động trên server với mỗi kết nối:

- AUTHORIZATION
- TRANSACTION
- UPDATE

Các lệnh POP3 thường dùng được mô tả trong bảng sau:

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?
- 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.
- Không xác thực người dùng.
- Giao thức không thực hiện bảo mật.

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.

Các lệnh gửi từ client:

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

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

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

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

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

Lệnh **QUIT**

Phản hồi từ server

- 100 OK
- 999 UNKNOWN ERROR

Các thông điệp từ server chủ động gửi các client:

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 về một phiên làm việc:

```

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
```

#### 4.4. Bài tập

**Bài tập 4.1.** Lập trình ứng dụng **chat\_server** và **chat\_client** theo giao thức được mô tả trong bài giảng.

**Bài tập 4.2.** Thiết kế giao thức và lập trình ứng dụng game server:

- Cờ caro
- Cờ vua

# Chương 5

## Lập trình socket nâng cao

---

5.1. Thư viện OpenSSL . . . . .	118
5.2. Raw socket . . . . .	123
5.3. Bài tập . . . . .	125

---

### 5.1. Thư viện OpenSSL

#### 5.1.1. Giới thiệu thư viện OpenSSL

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 theo các bước như sau:

1. Cập nhật gói phần mềm

```
sudo apt update
```

2. Cài đặt gói phần mềm libssl-dev

```
sudo apt install libssl-dev
```

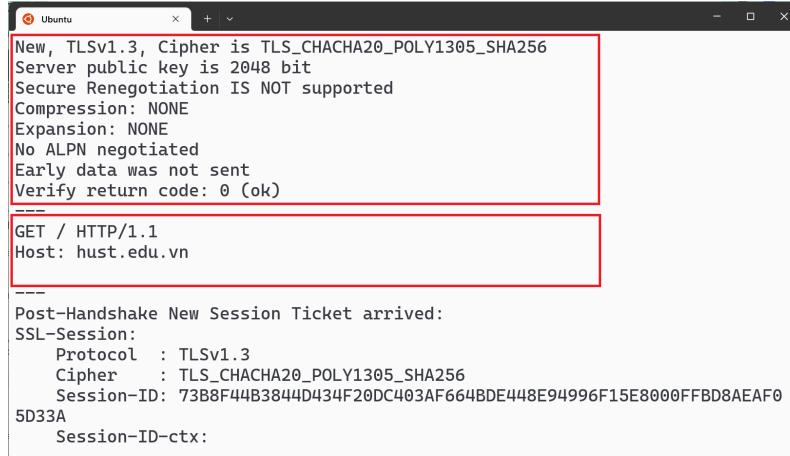
3. Kiểm tra kết quả cài đặt

```
openssl version
```

### 5.1.2. Sử dụng công cụ dòng lệnh

Công cụ dòng lệnh cho phép tạo đường truyền bảo mật theo giao thức HTTPS đến web server. Khi kết nối thành công, ta có thể sử dụng các lệnh của giao thức HTTP để gửi yêu cầu đến server và nhận về kết quả.

Ví dụ sau tạo kết nối đến trang web <https://hust.edu.vn/>



```

Ubuntu
New, TLSv1.3, Cipher is TLS_CHACHA20_POLY1305_SHA256
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)

GET / HTTP/1.1
Host: hust.edu.vn

-----
Post-Handshake New Session Ticket arrived:
SSL-Session:
Protocol : TLSv1.3
Cipher   : TLS_CHACHA20_POLY1305_SHA256
Session-ID: 73B8F44B3844D434F20DC403AF664BDE448E94996F15E8000FFBD8AEAF0
5D33A
Session-ID-ctx:
```

Kết quả nhận được là nội dung của website.



```

Ubuntu
Timeout  : 7200 (sec)
Verify return code: 0 (ok)
Extended master secret: no
Max Early Data: 0
-----
read R BLOCK
HTTP/1.1 200 OK
set-cookie: nv4s_c523Wo_ctr=MjAzXzIxMF8xNDhfMTg0LlZ0; expires=Thu, 10-Sep-2
026 08:32:25 GMT; Max-Age=31536000; path=/; HttpOnly; SameSite=Lax
set-cookie: nv4s_c523Wo_sess=p0j3l6btf5q9cm824m744m5oi8; path=/; secure; Ht
tpOnly; SameSite=Lax
set-cookie: nv4s_c523Wo_u_lang=U7au8HZYeRiUWEwxTsTsog%2C%2C; expires=Sat, 0
5-Sep-2026 08:32:25 GMT; Max-Age=31104000; path=/; HttpOnly; SameSite=Lax
set-cookie: nv4s_c523Wo_statistic_vi=ENoY9bQTuSGlpWglk_Q5A%2C%2C; expires=
Wed, 10-Sep-2025 09:02:25 GMT; Max-Age=1800; path=/; HttpOnly; SameSite=Lax
set-cookie: nv4s_c523Wo_nvvitthemever=VoyWrOpXFZc_u4Gf2vawYA%2C%2C; expires=
Sat, 05-Sep-2026 08:32:25 GMT; Max-Age=31104000; path=/; HttpOnly; SameSite
=Lax
x-content-type-options: nosniff
x-xss-protection: 1; mode=block
```

### 5.1.3. Cách cài đặt vào dự án

Sau khi cài đặt, để thêm thư viện openssl vào trong project, ta cần thực hiện các bước sau:

- 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
// Giả thiết socket đã được khởi tạo kết nối đến server
int err = SSL_connect(ssl); // Tạo kết nối ssl
if (err <= 0) {
    printf("Error creating SSL connection. err=%x\n", err);
    return -1;
}
```

Sau bước khởi tạo, con trỏ ssl được sử dụng để truyền nhận dữ liệu mã hóa. Ta sẽ không sử dụng các hàm truyền nhận trong thư viện socket API mà sử dụng các hàm sau:

- Nhận dữ liệu:

```
int SSL_read(SSL *ssl, void *buf, int num)
```

Các tham số:

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

Giá trị trả về của hàm:

- > 0 Số byte nhận được
- = 0 Kết nối đã bị đóng
- < 0 Có lỗi, cần gọi SSL\_get\_error(ssl, ret) để biết nguyên nhân.

- Truyền dữ liệu:

```
int SSL_write(SSL *ssl, const void *buf, int num)
```

Các tham số:

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

Giá trị trả về của hàm:

- > 0 Số byte thực sự đã gửi thành công
- = 0 Kết nối đã bị đóng
- < 0 Có lỗi, cần gọi SSL\_get\_error(ssl, ret) để biết nguyên nhân.

Để giải phóng con trỏ ssl ta sử dụng các hàm sau:

- 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

Ví dụ kết nối và nhận dữ liệu từ website https://hust.edu.vn:

```
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 #include <openssl/ssl.h>
9
10 int main() {
11     SSL_library_init();
12
13     const SSL_METHOD *my_ssl_method = TLS_client_method();
14 }
```

```
15     SSL_CTX *my_ssl_context = SSL_CTX_new(my_ssl_method);
16     if (my_ssl_context == NULL) {
17         printf("ERROR to create SSL context.\n");
18         return 1;
19     }
20
21     SSL *my_ssl = SSL_new(my_ssl_context);
22     if (my_ssl == NULL) {
23         printf("ERROR to create SSL.\n");
24         return 1;
25     }
26
27     int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
28
29     struct sockaddr_in addr;
30     addr.sin_family = AF_INET;
31     addr.sin_addr.s_addr = inet_addr("202.191.59.134");
32     addr.sin_port = htons(443);
33
34     connect(client, (struct sockaddr *)&addr, sizeof(addr));
35
36     SSL_set_fd(my_ssl, client);
37
38     if (SSL_connect(my_ssl) <= 0) {
39         printf("ERROR to SSL connect.\n");
40         return 1;
41     }
42
43     char buf[2048] = "GET / HTTP/1.1\r\nHost: hust.edu.vn\r\n\r\n";
44     SSL_write(my_ssl, buf, strlen(buf));
45
46     while (1) {
47         int len = SSL_read(my_ssl, buf, sizeof(buf));
48         if (len <= 0)
49             break;
50         if (len < sizeof(buf))
51             buf[len] = 0;
52
53         printf("%s", buf);
54     }
55
56     SSL_shutdown(my_ssl);
57     SSL_free(my_ssl);
58     SSL_CTX_free(my_ssl_context);
59     close(client);
```

60

}

## 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. Ta cần có hiểu biết cơ bản về những giao thức như ICMP, TCP, ... để có thể tạo hoặc xử lý gói tin ở tầng này.

Để tạo raw socket, ta sử dụng hàm socket() với tham số là SOCK\_RAW.

Ví dụ: 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 để truyền gói tin ICMP

```
int s2 = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
if (s2 == -1) {
    printf("Failed to create socket\n");
    return 1;
}
```

### Ứng dụng bắt gói tin

Các bước thực hiện như sau:

- 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 để trích xuất được thông tin mong muốn.

Chú ý:

- 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 WSLv1 hoặc MacOS).

Mã nguồn chương trình:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <unistd.h>
6 #include <netdb.h>
7 #include <string.h>
8 #include <errno.h>
9
10 int main() {
11     int sock_raw = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
12     if (sock_raw < 0) {
13         printf("Failed to create socket: %d\n", errno);
14         return 1;
15     }
16
17     unsigned char *buffer = (unsigned char *)malloc(65535);
18     int saddr_size, data_size;
19     struct sockaddr saddr;
20
21     while (1) {
22         data_size = recvfrom(sock_raw, buffer, 65535, 0, &saddr, &saddr_size);
23         if (data_size < 0) {
24             printf("recvfrom error, failed to get packets\n");
25             return 1;
26         }
27
28         // In header gói tin
29         printf("Data size: %d\n", data_size);
30         for (int i = 0; i < 40; i++)
31             printf("%x ", buffer[i]);
32         printf("\n");
33
34         // Phân tích nội dung gói tin ...
35     }
36
37     close(sock_raw);
38     return 0;

```

39

}

Kết quả khi chạy chương trình:

```
Ubuntu          X  Windows PowerShell  X  +  -  X
network_programming > sudo ./test4
[sudo] password for lebavui:
Data size: 113
45 0 0 71 a9 10 40 0 40 6 93 74 7f 0 0 1 7f 0 0 1 a1 2a 8a b3 1a 97 50 b5 a 9b d4 1 80 18 bc f9 fe 65 0 0
Data size: 52
45 0 0 34 96 d2 40 0 40 6 a5 ef 7f 0 0 1 7f 0 0 1 8a b3 a1 2a a 9b d4 1 1a 97 50 f2 80 10 bc f9 fe 28 0 0
Data size: 90
45 0 0 5a 96 d3 40 0 40 6 a5 c8 7f 0 0 1 7f 0 0 1 8a b3 a1 2a a 9b d4 1 1a 97 50 f2 80 18 bc f9 fe 4e 0 0
Data size: 52
45 0 0 34 a9 11 40 0 40 6 93 b0 7f 0 0 1 7f 0 0 1 a1 2a 8a b3 1a 97 50 f2 a 9b d4 27 80 10 bc f9 fe 28 0 0
Data size: 71
45 0 0 47 a9 12 40 0 40 6 93 9c 7f 0 0 1 7f 0 0 1 a1 2a 8a b3 1a 97 50 f2 a 9b d4 27 80 18 bc f9 fe 3b 0 0
Data size: 52
45 0 0 34 96 d4 40 0 40 6 a5 ed 7f 0 0 1 7f 0 0 1 8a b3 a1 2a a 9b d4 27 1a 97 51 5 80 10 bc f9 fe 28 0 0
Data size: 67
45 0 0 43 fb d3 40 0 40 6 40 df 7f 0 0 1 7f 0 0 1 8a b3 a1 26 3e ac 57 fc fc 7e ca fc 80 18 3 5e fe 37 0 0
Data size: 52
45 0 0 34 14 85 40 0 40 6 28 3d 7f 0 0 1 7f 0 0 1 a1 26 8a b3 fc 7e ca fc 3e ac 58 b 80 10 a9 96 fe 28 0 0
Data size: 71
```

### 5.3. Bài tập

**Bài tập 5.1.** Chính 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?

# Tài liệu tham khảo

---

- [1] W.Richard Stevens, *Unix Network Programming Vol.1, 3rd Ed.*, Prentice Hall
- [2] Keir Davis, John W. Turner, and Nathan Yocom, *The Definitive Guide to Linux Network Programming*, Apress.
- [3] Michael Donahoo, Kenneth Calvert, *TCP/IP Sockets in C: Practical Guide for Programmers*, Elsevier.