

1. Nội dung 1: Hàm select()	3
1.1. Ôn tập lý thuyết	3
Giới thiệu về hàm select()	3
Cú pháp của hàm select()	3
Quy trình sử dụng hàm select()	4
Ví dụ minh họa	4
Lưu ý quan trọng khi sử dụng select()	5
Một số lưu ý bổ sung:	5
1.2. Bài tập và gợi ý trả lời	6
2. Nội dung 2: Hàm poll() và cấu trúc pollfd	13
2.1. Ôn tập lý thuyết	13
Giới thiệu về hàm poll()	13
Cú pháp hàm poll()	13
Cấu trúc pollfd	13
Quy trình sử dụng poll()	14
Ví dụ minh họa sử dụng poll()	14
Lưu ý khi sử dụng poll()	15
So sánh poll() và select()	15
2.2. Bài tập mẫu	15
3. Nội dung 3: đa tiến trình với fork()	19
3.1. Lý thuyết	19
Giới thiệu về fork()	19
Quá trình tạo tiến trình con	19
Điều kiện xác định tiến trình cha và con	19
Quản lý tiến trình con	20
Mô hình server đa tiến trình trong lập trình mạng	20
Ví dụ server TCP đa tiến trình sử dụng fork()	20
Các vấn đề cần lưu ý	21
Lưu ý đặc biệt	21
3.2. Bài tập	22
4. Nội dung 4: Lập trình mạng đa luồng với pthread	26
4.1. Lý thuyết	26
Giới thiệu về lập trình đa luồng (multi-threaded programming)	26

<b>Thư viện pthread (POSIX Threads).....</b>	<b>26</b>
<b>Tạo và quản lý luồng với pthread .....</b>	<b>26</b>
<b>Truyền tham số vào luồng .....</b>	<b>27</b>
<b>Đồng bộ hóa luồng với mutex .....</b>	<b>27</b>
<b>Ví dụ server TCP đa luồng sử dụng pthread.....</b>	<b>27</b>
<b>Các vấn đề cần lưu ý .....</b>	<b>28</b>
<b>Lưu ý đặc biệt.....</b>	<b>28</b>
4.2. Bài tập .....	29
5. Nội dung 5: Socket cơ bản .....	33
5.1. Lý thuyết .....	33
Socket là gì? .....	33
Các bước lập trình socket TCP cơ bản .....	33
Cú pháp và mô tả các hàm socket cơ bản.....	33
Lập trình socket UDP .....	35
Ví dụ socket TCP client và server đơn giản .....	35
Các nội dung cần lưu ý: .....	36
5.2. Bài tập .....	36

# ÔN TẬP LÝ THUYẾT LẬP TRÌNH MẠNG

## 1. Nội dung 1: Hàm select()

### 1.1. Ôn tập lý thuyết

#### Giới thiệu về hàm select()

Hàm **select()** là một hàm trong thư viện **sys/select.h** của ngôn ngữ C, được sử dụng để theo dõi nhiều **socket** hoặc **file descriptor (FD)** đồng thời, kiểm tra xem có FD nào **sẵn sàng** để thực hiện các thao tác **đọc, ghi**, hoặc có **sự kiện ngoại lệ** xảy ra hay không.

Nó rất hữu ích trong các ứng dụng **mạng phi đồng bộ (asynchronous networking)** khi bạn cần quản lý **nhiều kết nối** mà không cần tạo nhiều luồng (thread) hoặc tiến trình (process).

---

#### Cú pháp của hàm select()

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

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

- **Tham số:**
  - **nfds**: Số lớn nhất của FD trong ba tập FD (readfds, writefds, exceptfds) cộng thêm **1**. Nó là số lượng FD cần theo dõi (không phải tổng số socket, mà là FD lớn nhất + 1).
  - **readfds**: Tập hợp các FD cần kiểm tra xem có **dữ liệu để đọc**.
  - **writefds**: Tập hợp các FD cần kiểm tra xem có thể **ghi dữ liệu** được không.
  - **exceptfds**: Tập hợp các FD cần kiểm tra **sự kiện ngoại lệ** (ví dụ, lỗi trên socket).
  - **timeout**: Thời gian chờ tối đa (giới hạn thời gian) cho hàm **select()**.
    - Nếu **timeout == NULL**: **select()** sẽ chờ **vô thời hạn** cho đến khi có FD sẵn sàng.
    - Nếu **timeout = 0** giây: **select()** sẽ **kiểm tra ngay lập tức** rồi trả về, không chờ.
    - Nếu **timeout > 0**: **select()** chờ cho đến khi FD sẵn sàng hoặc hết thời gian.
- **Giá trị trả về:**
  - **0**: Số lượng FD **sẵn sàng** cho thao tác.

- 0 : Hết thời gian chờ nhưng không có FD nào sẵn sàng.
  - -1 : Có lỗi xảy ra (thường gặp lỗi do signal ngắt hoặc lỗi khi truyền FD).
- 

## Quy trình sử dụng hàm select()

### Bước 1: Khởi tạo tập FD bằng FD\_ZERO và FD\_SET

```
fd_set readfds;
FD_ZERO(&readfds);           // Xóa tập hợp FD
FD_SET(sockfd, &readfds);    // Thêm sockfd vào tập hợp để kiểm tra
```

- **FD\_ZERO(fd\_set \*fdset):** Xóa tất cả các FD khỏi tập hợp.
- **FD\_SET(int fd, fd\_set \*fdset):** Thêm FD vào tập hợp.
- **FD\_CLR(int fd, fd\_set \*fdset):** Xóa FD khỏi tập hợp.
- **FD\_ISSET(int fd, fd\_set \*fdset):** Kiểm tra xem FD có nằm trong tập hợp không.

### Bước 2: Gọi select() và xử lý kết quả

```
int result = select(max_fd + 1, &readfds, NULL, NULL, &timeout);
if (result > 0) {
    if (FD_ISSET(sockfd, &readfds)) {
        // sockfd sẵn sàng để đọc
    }
} else if (result == 0) {
    // Hết thời gian chờ mà không có FD nào sẵn sàng
} else {
    // Có lỗi xảy ra
}
```

---

## Ví dụ minh họa

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <unistd.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr = {0};
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8080);
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    listen(sockfd, 5);
```

```

fd_set readfds;
struct timeval timeout;

while (1) {
    FD_ZERO(&readfds);
    FD_SET(sockfd, &readfds);
    timeout.tv_sec = 5; // Chờ 5 giây
    timeout.tv_usec = 0;

    int result = select(sockfd + 1, &readfds, NULL, NULL, &timeout);

    if (result > 0) {
        if (FD_ISSET(sockfd, &readfds)) {
            // Có kết nối mới
            printf("Có kết nối mới!\n");
        }
    } else if (result == 0) {
        printf("Hết thời gian chờ, không có kết nối mới.\n");
    } else {
        perror("select error");
        break;
    }
}
close(sockfd);
return 0;
}

```

---

### Lưu ý quan trọng khi sử dụng select()

- **nfds = FD lớn nhất + 1:** Thường tính bằng `max_fd + 1`.
  - Phải **khởi tạo lại** các tập FD (`FD_ZERO`, `FD_SET`) **mỗi lần trước khi gọi select()**, vì `select()` thay đổi nội dung của các tập này sau khi chạy.
  - **timeout** có thể dùng để tránh bị **block vô thời hạn**, giúp chương trình kiểm soát thời gian chờ.
  - `select()` **không kiểm tra FD đã đóng** nên cần đảm bảo các FD hợp lệ trước khi thêm vào tập FD.
- 

### Một số lưu ý bổ sung:

1. Giá trị trả về của `select()` là số lượng FD sẵn sàng hoặc 0 nếu hết thời gian chờ.
2. Tham số `nfds` phải là FD lớn nhất + 1.
3. Nếu `timeout` bằng `NULL`, `select()` sẽ chờ vô thời hạn.
4. Hết thời gian chờ mà không có FD nào sẵn sàng, `select()` trả về 0.

## 1.2. Bài tập và gợi ý trả lời

### Bài 0: Hiểu rõ cấu trúc `fd_set` và `fds_bits`

**Yêu cầu:** Viết đoạn mã minh họa cách các bit trong `fds_bits` được set và hiển thị giá trị của `fds_bits[0]` khi thêm các socket vào.

**Giải thích:**

- `fd_set` được cài đặt dưới dạng mảng các số nguyên (long int) với tên `fds_bits[]`, mỗi bit trong `fds_bits` đại diện cho một socket.
- Ví dụ: Nếu thêm socket 3 và 4 vào tập, các bit thứ 3 và 4 trong `fds_bits[0]` sẽ được set thành 1.

**Code mẫu:**

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

int main() {
    fd_set fdset;
    FD_ZERO(&fdset);

    FD_SET(3, &fdset); // Set bit thứ 3
    FD_SET(4, &fdset); // Set bit thứ 4

    printf("fds_bits[0] = %ld\n", fdset.fds_bits[0]);
    // Hiển thị giá trị nhị phân của fds_bits[0]
    for (int i = sizeof(fdset.fds_bits[0])*8 - 1; i >= 0; i--) {
        printf("%d", (fdset.fds_bits[0] >> i) & 1);
    }
    printf("\n");
    return 0;
}
```

**Kết quả mong đợi:**

- `fds_bits[0] = 24` (vì 2 bit thứ 3 và 4 được set:  $2^3 + 2^4 = 8 + 16 = 24$ ).
- Dòng hiển thị nhị phân kết thúc bằng 00011000 (bit 3 và 4 được set).

### Bài 1: Khởi tạo tập file descriptor và thêm socket vào

**Yêu cầu:** Viết đoạn mã khởi tạo tập `fd_set` và thêm socket vào tập này.

**Giải thích:** `fd_set` là kiểu dữ liệu đặc biệt dùng trong hàm `select()` để quản lý nhiều file descriptor (FD). Trước khi thêm FD, bạn cần khởi tạo tập FD bằng `FD_ZERO()`. Sau đó, sử dụng `FD_SET()` để thêm socket vào.

**Code mẫu:**

```
fd_set read_fds;
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
FD_ZERO(&read_fds); // Khởi tạo tập FD rỗng
FD_SET(sockfd, &read_fds); // Thêm sockfd vào tập
```

---

## Bài 2: Kiểm tra socket có trong tập file descriptor không

**Yêu cầu:** Viết đoạn mã kiểm tra xem một socket có trong tập `fd_set` hay không.

**Giải thích:** Dùng `FD_ISSET()` để kiểm tra xem FD có trong tập hay không. Hàm trả về 1 nếu có, 0 nếu không.

**Code mẫu:**

```
if (FD_ISSET(sockfd, &read_fds)) {
    printf("Socket có sẵn để đọc\n");
}
```

---

## Bài 3: Xóa một socket khỏi tập file descriptor

**Yêu cầu:** Viết đoạn mã xóa một socket khỏi tập `fd_set`.

**Giải thích:** Dùng `FD_CLR()` để xóa FD khỏi tập.

**Code mẫu:**

```
FD_CLR(sockfd, &read_fds);
```

---

## Bài 4: Gọi hàm `select()` chờ socket sẵn sàng đọc

**Yêu cầu:** Viết đoạn mã sử dụng `select()` để chờ socket có thể đọc.

**Giải thích:** Hàm `select()` nhận các tập FD cho việc đọc, ghi, và ngoại lệ. Truyền số lượng FD lớn nhất + 1 làm tham số đầu tiên.

**Code mẫu:**

```
fd_set read_fds;
```

```
FD_ZERO(&read_fds);
FD_SET(sockfd, &read_fds);
select(sockfd + 1, &read_fds, NULL, NULL, NULL);
```

---

## Bài 5: Cấu hình timeout cho select()

**Yêu cầu:** Viết đoạn mã thiết lập timeout 5 giây cho `select()`.

**Giải thích:** Dùng struct `timeval` để chỉ định thời gian timeout cho `select()`.

**Code mẫu:**

```
struct timeval timeout;
timeout.tv_sec = 5;
timeout.tv_usec = 0;
select(sockfd + 1, &read_fds, NULL, NULL, &timeout);
```

---

## Bài 6: Phân biệt timeout NULL và timeout > 0

**Yêu cầu:** So sánh hai cách gọi `select()` với `timeout = NULL` và `timeout` có giá trị cụ thể.

**Giải thích:**

- `timeout = NULL`: `select()` sẽ chờ vô thời hạn.
  - `timeout` có giá trị: `select()` chờ tối đa thời gian chỉ định.
- 

## Bài 7: Sử dụng select() với nhiều socket

**Yêu cầu:** Viết đoạn mã sử dụng `select()` để theo dõi 2 socket.

**Giải thích:**

- Thêm cả hai socket vào tập `fd_set`.
- Dùng `select()` để kiểm tra cả hai.

**Code mẫu:**

```
int sockfd1, sockfd2;
fd_set read_fds;
FD_ZERO(&read_fds);
FD_SET(sockfd1, &read_fds);
FD_SET(sockfd2, &read_fds);
int maxfd = (sockfd1 > sockfd2) ? sockfd1 : sockfd2;
select(maxfd + 1, &read_fds, NULL, NULL, NULL);
```



---

## Bài 8: Xử lý nhiều socket sau select()

**Yêu cầu:** Viết đoạn mã kiểm tra socket nào sẵn sàng sau khi `select()` trả về.

**Giải thích:** Dùng vòng lặp kết hợp với `FD_ISSET()` để kiểm tra từng socket.

**Code mẫu:**

```
for (int i = 0; i <= maxfd; i++) {
    if (FD_ISSET(i, &read_fds)) {
        // Xử lý socket i
    }
}
```

---

## Bài 9: Gọi select() và xử lý timeout hết hạn

**Yêu cầu:** Viết đoạn mã để in ra "timeout" nếu `select()` hết thời gian chờ.

**Giải thích:** `select()` trả về 0 nếu timeout hết hạn mà không có FD nào sẵn sàng.

**Code mẫu:**

```
int result = select(sockfd + 1, &read_fds, NULL, NULL, &timeout);
if (result == 0) {
    printf("Timeout xảy ra\n");
}
```

---

## Bài 10: Phân biệt giá trị trả về của select()

**Yêu cầu:** Viết đoạn mã xử lý 3 trường hợp trả về của `select()`.

**Giải thích:**

- `>0`: có ít nhất một FD sẵn sàng.
- `=0`: timeout.
- `<0`: lỗi.

**Code mẫu:**

```
int result = select(maxfd + 1, &read_fds, NULL, NULL, &timeout);
if (result > 0) {
    // Có FD sẵn sàng
} else if (result == 0) {
    printf("Timeout\n");
}
```

```
} else {  
    perror("select");  
}
```

---

## Bài 11: Làm mới tập fd\_set sau mỗi lần select()

**Yêu cầu:** Giải thích lý do tại sao cần gọi lại `FD_ZERO()` và `FD_SET()` trước mỗi lần gọi `select()`.

**Giải thích:** `select()` thay đổi tập fd\_set (chỉ giữ lại FD sẵn sàng). Nếu muốn tiếp tục theo dõi các FD ban đầu, cần khởi tạo lại.

---

## Bài 12: Theo dõi socket lắng nghe và client cùng lúc

**Yêu cầu:** Viết đoạn mã theo dõi socket lắng nghe và một socket client đồng thời.

**Code mẫu:**

```
FD_ZERO(&read_fds);  
FD_SET(listener, &read_fds);  
FD_SET(client, &read_fds);  
maxfd = (listener > client) ? listener : client;  
select(maxfd + 1, &read_fds, NULL, NULL, NULL);
```

---

## Bài 13: Sử dụng select() để kiểm tra stdin và socket

**Yêu cầu:** Viết đoạn mã kiểm tra đồng thời stdin và socket.

**Code mẫu:**

```
FD_ZERO(&read_fds);  
FD_SET(0, &read_fds); // stdin  
FD_SET(sockfd, &read_fds);  
maxfd = (sockfd > 0) ? sockfd : 0;  
select(maxfd + 1, &read_fds, NULL, NULL, NULL);
```

---

## Bài 14: Thêm timeout động

**Yêu cầu:** Viết đoạn mã thay đổi timeout giữa các lần gọi `select()`.

**Giải thích:**

- Tạo struct `timeval` mới cho mỗi lần gọi `select()`.
- 

## Bài 15: Đếm số socket sẵn sàng sau `select()`

**Yêu cầu:** In ra số socket sẵn sàng dựa trên giá trị trả về của `select()`.

**Code mẫu:**

```
int ready = select(maxfd + 1, &read_fds, NULL, NULL, &timeout);
printf("%d socket sẵn sàng\n", ready);
```

---

## Bài 16: Đóng socket không còn sẵn sàng

**Yêu cầu:** Sau khi `select()` trả về, đóng các socket không còn sẵn sàng.

**Code mẫu:**

```
for (int i = 0; i <= maxfd; i++) {
    if (!FD_ISSET(i, &read_fds)) {
        close(i);
    }
}
```

---

## Bài 17: Xử lý ngoại lệ bằng `select()`

**Yêu cầu:** Sử dụng `exceptfds` để kiểm tra ngoại lệ.

**Code mẫu:**

```
fd_set exceptfds;
FD_ZERO(&exceptfds);
FD_SET(sockfd, &exceptfds);
select(sockfd + 1, NULL, NULL, &exceptfds, &timeout);
```

---

## Bài 18: Tăng số lượng socket lên tối đa `FD_SETSIZE`

**Yêu cầu:** Thêm tối đa số socket có thể theo dõi với `select()`.

**Giải thích:**

- `FD_SETSIZE` (thường là 1024).

---

## Bài 19: Kiểm tra hiệu suất `select()` với nhiều socket

**Yêu cầu:** Đánh giá độ trễ khi theo dõi số lượng lớn socket.

---

## Bài 20: Viết hàm wrapper cho `select()`

**Yêu cầu:** Viết một hàm bọc `select()` đơn giản.

**Code mẫu:**

```
int my_select(int maxfd, fd_set *readfds, struct timeval *timeout) {  
    return select(maxfd + 1, readfds, NULL, NULL, timeout);  
}
```

## 2. Nội dung 2: Hàm poll() và cấu trúc pollfd

### 2.1. Ôn tập lý thuyết

#### Giới thiệu về hàm poll()

Hàm **poll()** là một giải pháp thay thế hiện đại cho **select()** dùng để **theo dõi nhiều file descriptor (FD)** (socket, stdin, file, ...) nhằm kiểm tra xem FD nào sẵn sàng cho các thao tác **đọc, ghi**, hoặc xảy ra **sự kiện ngoại lệ**.

Ưu điểm so với select():

- **Không bị giới hạn bởi FD\_SETSIZE (thường là 1024).**
  - **Dễ dàng xử lý số lượng lớn socket** mà không cần thao tác trực tiếp với bitmask.
- 

#### Cú pháp hàm poll()

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

#### Tham số:

- **fds[]**: Mảng các cấu trúc **pollfd**, mỗi phần tử đại diện cho một FD cần theo dõi.
- **nfds**: Số lượng phần tử trong mảng **fds[]**.
- **timeout**:
  - Đơn vị: **mili-giây (ms)**.
  - **-1**: Chờ vô thời hạn.
  - **0**: Không chờ, kiểm tra ngay lập tức.
  - **>0**: Chờ trong khoảng thời gian chỉ định.

#### Giá trị trả về:

- **0**: Số lượng FD **sẵn sàng** cho thao tác.
  - **0**: **Hết thời gian chờ (timeout)**.
  - **-1**: **Lỗi xảy ra**.
- 

#### Cấu trúc pollfd

```
struct pollfd {
    int fd;           // File descriptor (socket hoặc file)
    short events;     // Sự kiện muốn theo dõi (input)
    short revents;    // Sự kiện thực tế xảy ra (output)
};
```

### Các cờ sự kiện cho **events** và **revents**:

Cờ sự kiện	Ý nghĩa
<b>POLLIN</b>	Dữ liệu sẵn sàng để <b>đọc</b> .
<b>POLLOUT</b>	Có thể <b>ghi</b> dữ liệu mà không bị block.
<b>POLLERR</b>	Có lỗi xảy ra.
<b>POLLHUP</b>	Đầu kia của kết nối đã đóng (hung up).
<b>POLLNVAL</b>	FD không hợp lệ (chưa mở hoặc đã đóng).

---

### Quy trình sử dụng **poll()**

1. **Tạo mảng `pollfd[]`:**
    - o Gán các FD cần theo dõi vào trường `fd`.
    - o Đăng ký sự kiện muốn theo dõi vào trường `events`.
  2. **Gọi `poll()`:**
    - o Truyền mảng `pollfd`, số lượng phần tử và `timeout`.
  3. **Kiểm tra kết quả với trường `revents`:**
    - o Sau khi `poll()` trả về, trường `revents` cho biết sự kiện nào đã xảy ra.
- 

### Ví dụ minh họa sử dụng **poll()**

#### Ví dụ 1: Theo dõi một socket và **stdin**

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

int main() {
    struct pollfd fds[2];
    int timeout = 5000; // 5 giây

    fds[0].fd = 0;           // stdin
    fds[0].events = POLLIN;  // Theo dõi đọc

    fds[1].fd = sockfd;      // Socket
    fds[1].events = POLLIN;  // Theo dõi đọc

    int ret = poll(fds, 2, timeout);
```

```

if (ret == -1) {
    perror("poll error");
} else if (ret == 0) {
    printf("Timeout sau 5 giây.\n");
} else {
    if (fds[0].revents & POLLIN) {
        printf("Có dữ liệu từ bàn phím.\n");
    }
    if (fds[1].revents & POLLIN) {
        printf("Socket có dữ liệu để đọc.\n");
    }
}
return 0;
}

```

---

### Lưu ý khi sử dụng `poll()`

- Trường `revents` chỉ được **cập nhật bởi** `poll()` → luôn kiểm tra `revents` sau khi gọi `poll()`.
  - Nếu **FD không hợp lệ**, `revents` sẽ chứa **POLLNVAL**.
  - `poll()` không bị giới hạn số lượng FD như `select()`, nhưng hiệu suất giảm dần khi số lượng FD quá lớn.
- 

### So sánh `poll()` và `select()`

Tiêu chí	<code>select()</code>	<code>poll()</code>	
<b>Giới hạn FD</b>	Có giới hạn ( <code>FD_SETSIZE</code> , thường là 1024).	Không giới hạn số FD.	
<b>Kiểu dữ liệu</b>	Bitmask ( <code>fd_set</code> ).	Mảng <code>pollfd</code> .	
<b>Cấu trúc lưu trữ</b>	Phức tạp hơn (bitmask).	Đơn giản (mảng các struct).	
<b>Sự kiện xảy ra</b>	Kiểm tra bằng <code>FD_ISSET()</code> .	Kiểm tra bằng <code>revents</code> .	
<b>Thời gian chờ</b>	<code>struct timeval</code> (microseconds).	Đơn vị mili-giây (milliseconds).	

## 2.2. Bài tập mẫu

### Bài 1: Khởi tạo mảng `pollfd` để theo dõi `stdin`

**Yêu cầu:** Viết đoạn mã khởi tạo mảng `pollfd` để theo dõi `stdin` (file descriptor 0).

### Giải thích:

- `pollfd.fd = 0` để theo dõi `stdin`.
- `events = POLLIN` để kiểm tra khi có dữ liệu nhập từ bàn phím.

### Code mẫu:

```
struct pollfd fds[1];
fds[0].fd = 0;           // stdin
fds[0].events = POLLIN; // Theo dõi sự kiện đọc
```

---

## Bài 2: Gọi `poll()` và xử lý khi `stdin` có dữ liệu nhập vào

**Yêu cầu:** Viết đoạn mã chờ dữ liệu nhập từ bàn phím trong 5 giây.

### Code mẫu:

```
int ret = poll(fds, 1, 5000); // 5 giây timeout
if (ret == -1) {
    perror("poll error");
} else if (ret == 0) {
    printf("Timeout, không có dữ liệu\n");
} else {
    if (fds[0].revents & POLLIN) {
        printf("Có dữ liệu từ stdin\n");
    }
}
```

---

## Bài 3: Theo dõi nhiều socket với `poll()`

**Yêu cầu:** Viết đoạn mã theo dõi 2 socket `sockfd1` và `sockfd2` để kiểm tra khi có dữ liệu đến.

### Code mẫu:

```
struct pollfd fds[2];
fds[0].fd = sockfd1;
fds[0].events = POLLIN;
fds[1].fd = sockfd2;
fds[1].events = POLLIN;
poll(fds, 2, 10000); // 10 giây timeout
```

---

## Bài 4: Kiểm tra sự kiện trên từng phần tử mảng `pollfd`

**Yêu cầu:** Viết vòng lặp kiểm tra sự kiện đã xảy ra (`revents`) trên từng socket.

### Code mẫu:



```
for (int i = 0; i < 2; i++) {
    if (fds[i].revents & POLLIN) {
        printf("Socket %d có dữ liệu\n", fds[i].fd);
    }
}
```

---

## Bài 5: Sử dụng poll() với timeout = 0

**Yêu cầu:** Viết đoạn mã kiểm tra các socket ngay lập tức (không chờ).

**Giải thích:**

- Timeout = 0 giúp poll() kiểm tra và trả về ngay lập tức.

**Code mẫu:**

```
poll(fds, 2, 0); // Không chờ
```

---

## Bài 6: Xử lý lỗi khi file descriptor không hợp lệ (POLLNVAL)

**Yêu cầu:** Viết đoạn mã kiểm tra sự kiện POLLNVAL nếu socket chưa khởi tạo.

**Code mẫu:**

```
struct pollfd fds[1];
fds[0].fd = -1; // FD không hợp lệ
fds[0].events = POLLIN;
if (poll(fds, 1, 1000) > 0) {
    if (fds[0].revents & POLLNVAL) {
        printf("FD không hợp lệ\n");
    }
}
```

---

## Bài 7: Theo dõi socket ghi (POLLOUT)

**Yêu cầu:** Viết đoạn mã kiểm tra khi socket sẵn sàng ghi dữ liệu.

**Code mẫu:**

```
fds[0].events = POLLOUT;
int ret = poll(fds, 1, 5000);
if (fds[0].revents & POLLOUT) {
    printf("Socket sẵn sàng ghi\n");
}
```

---

## Bài 8: Kiểm tra nhiều sự kiện (POLLIN | POLLOUT)

**Yêu cầu:** Viết đoạn mã kiểm tra cả đọc và ghi trên cùng một socket.

**Code mẫu:**

```
fds[0].events = POLLIN | POLLOUT;
if (poll(fds, 1, 5000) > 0) {
    if (fds[0].revents & POLLIN) printf("Có dữ liệu để đọc\n");
    if (fds[0].revents & POLLOUT) printf("Sẵn sàng ghi\n");
}
```

---

## Bài 9: Phát hiện client đóng kết nối (POLLHUP)

**Yêu cầu:** Viết đoạn mã kiểm tra khi client đóng kết nối.

**Giải thích:**

- POLLHUP cho biết kết nối bị đóng.

**Code mẫu:**

```
if (fds[0].revents & POLLHUP) {
    printf("Client đóng kết nối\n");
}
```

---

## Bài 10: Thực hiện server đơn giản theo dõi nhiều client

**Yêu cầu:** Viết server TCP đơn giản sử dụng poll() để theo dõi nhiều client.

**Gợi ý:**

- Tạo socket lắng nghe.
- Khi có kết nối mới, thêm vào mảng pollfd.
- Kiểm tra dữ liệu từ các client.

**Code mẫu:**

```
// Gợi ý logic
struct pollfd fds[MAX_CLIENTS];
// Thêm socket lắng nghe và client vào fds[]
// Dùng poll(fds, num_fds, timeout);
// Kiểm tra từng revents
```

## 3. Nội dung 3: đa tiến trình với fork()

### 3.1. Lý thuyết

#### Giới thiệu về fork()

- `fork()` là một hàm trong thư viện **unistd.h** dùng để tạo **một tiến trình con** mới từ tiến trình cha.
- Sau khi gọi `fork()`, **hai tiến trình** (cha và con) cùng thực thi tiếp các câu lệnh sau `fork()`, nhưng **có không gian bộ nhớ tách biệt**.

#### Cú pháp:

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

```
pid_t fork(void);
```

- **Giá trị trả về:**
  - **0**: trong **tiến trình con**.
  - **PID của tiến trình con (lớn hơn 0)**: trong **tiến trình cha**.
  - **-1**: lỗi (không tạo được tiến trình con).

---

#### Quá trình tạo tiến trình con

- `fork()` sao chép toàn bộ tiến trình cha thành một tiến trình con (bao gồm cả không gian bộ nhớ, các biến, descriptors).
- **Hai tiến trình cha và con** thực thi song song, nhưng **không chia sẻ vùng dữ liệu (biến toàn cục, heap)**.
- **Socket descriptor** (nếu mở trước `fork`) được **sao chép** cho cả cha và con (giống nhau nhưng không dùng chung bộ đệm).

---

#### Điều kiện xác định tiến trình cha và con

Sau khi `fork()`:

```
pid_t pid = fork();
if (pid == 0) {
    // Tiến trình con
} else if (pid > 0) {
    // Tiến trình cha
} else {
    // fork thất bại
}
```

```
}
```

- Có thể viết điều kiện kiểm tra tiến trình cha:

```
if (fork() != 0) {  
    // Tiến trình cha  
}
```

---

## Quản lý tiến trình con

- Tiến trình **cha** nên gọi `wait()` hoặc `waitpid()` để thu thập trạng thái kết thúc của tiến trình con nhằm **tránh tiến trình zombie**.
- **Zombie process**: Xảy ra khi **tiến trình con kết thúc**, nhưng **tiến trình cha chưa thu thập trạng thái** của nó.

### Ví dụ:

```
#include <sys/wait.h>  
wait(NULL); // Chờ bất kỳ tiến trình con nào kết thúc
```

---

## Mô hình server đa tiến trình trong lập trình mạng

- **Mỗi khi có kết nối từ client**, server gọi `fork()` để tạo **một tiến trình con** xử lý client đó.
- **Tiến trình cha** tiếp tục lắng nghe kết nối mới.

### Quy trình:

1. Tạo socket và listen.
  2. Khi có client kết nối:
    - Gọi `fork()`.
    - **Tiến trình con**: xử lý client.
    - **Tiến trình cha**: đóng socket client và quay lại lắng nghe.
- 

## Ví dụ server TCP đa tiến trình sử dụng `fork()`

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <sys/socket.h>  
#include <sys/wait.h>  
  
int main() {  
    int listener = socket(AF_INET, SOCK_STREAM, 0);  
    struct sockaddr_in addr;
```

```

addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(8080);

bind(listener, (struct sockaddr*)&addr, sizeof(addr));
listen(listener, 5);

while (1) {
    int client = accept(listener, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(listener); // Tiến trình con đóng socket lắng nghe
        char buffer[100];
        read(client, buffer, sizeof(buffer));
        printf("Received: %s\n", buffer);
        close(client);
        exit(0);
    } else {
        close(client); // Tiến trình cha đóng socket client
        waitpid(-1, NULL, WNOHANG); // Thu dọn tiến trình zombie
    }
}
return 0;
}

```

---

### Các vấn đề cần lưu ý

- **Fork bomb:** Nếu không kiểm soát số lượng tiến trình con tạo ra, có thể gây **quá tải hệ thống**.
  - **Thu dọn tiến trình zombie:** Tiến trình cha cần **thu thập trạng thái tiến trình con** để tránh zombie.
  - **Socket descriptor:**
    - Sau `fork()`, cả cha và con có bản sao descriptor giống nhau.
    - Nên **đóng socket không sử dụng** trong mỗi tiến trình để tránh rò rỉ.
- 

### Lưu ý đặc biệt

- **Giá trị trả về của `fork()`** giúp xác định tiến trình cha hoặc con.
- **Số lượng tiến trình con** tạo ra sau nhiều lệnh `fork()`:
  - Ví dụ: Hai lần `fork()` → 4 tiến trình (1 cha + 3 con).
- **Zombie process:** Khi tiến trình con kết thúc mà cha chưa thu dọn.
- **Server đa tiến trình:** Mỗi kết nối client được xử lý bởi một tiến trình con tạo từ `fork()`.

## 3.2. Bài tập

### Bài 1: Viết chương trình kiểm tra fork()

**Yêu cầu:** Viết chương trình tạo một tiến trình con và in ra thông báo từ cả tiến trình cha và con.

**Code mẫu:**

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Tiến trình con\n");
    } else if (pid > 0) {
        printf("Tiến trình cha\n");
    } else {
        perror("fork thất bại");
    }
    return 0;
}
```

---

### Bài 2: Xác định số lượng tiến trình sau nhiều lần fork()

**Yêu cầu:** Viết chương trình gọi `fork()` hai lần và in PID trong mỗi tiến trình.

**Giải thích:**

- Hai lần `fork()` → tạo 4 tiến trình (1 cha + 3 con).

**Code mẫu:**

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

int main() {
    fork();
    fork();
    printf("PID: %d\n", getpid());
    return 0;
}
```

---

### Bài 3: Kiểm tra fork() với điều kiện

**Yêu cầu:** Viết đoạn mã kiểm tra tiến trình cha bằng điều kiện (`fork() != 0`).

**Giải thích:**

- Điều kiện `fork() != 0` đúng với tiến trình cha.

**Code mẫu:**

```
if (fork() != 0) {
    printf("Tiến trình cha\n");
} else {
    printf("Tiến trình con\n");
}
```

---

**Bài 4: Viết chương trình tạo zombie process**

**Yêu cầu:** Viết chương trình để tạo tiến trình zombie.

**Giải thích:**

- Tiến trình cha không gọi `wait()` khi con kết thúc.

**Code mẫu:**

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

int main() {
    if (fork() == 0) {
        printf("Tiến trình con kết thúc\n");
        _exit(0);
    } else {
        sleep(10); // Tiến trình cha tạm dừng, con trở thành zombie
    }
    return 0;
}
```

---

**Bài 5: Dùng `wait()` để thu dọn zombie**

**Yêu cầu:** Viết đoạn mã cha chờ con kết thúc để tránh zombie.

**Code mẫu:**

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

int main() {
    if (fork() == 0) {
        printf("Tiến trình con\n");
    }
    wait(0);
}
```

```
        _exit(0);
    } else {
        wait(NULL); // Thu dọn tiến trình con
        printf("Tiến trình cha kết thúc\n");
    }
    return 0;
}
```

---

## Bài 6: Viết server TCP đa tiến trình đơn giản

**Yêu cầu:** Viết server lắng nghe kết nối và tạo tiến trình con xử lý mỗi client.

**Gợi ý:**

- Dùng `fork()` tạo tiến trình con khi có kết nối mới.

**Code mẫu:**

```
// Lắng nghe socket và accept() client
// fork() để xử lý client trong tiến trình con
```

---

## Bài 7: Đóng socket không cần thiết sau fork()

**Yêu cầu:** Viết đoạn mã đóng socket lắng nghe ở tiến trình con sau `fork()`.

**Giải thích:**

- Tiến trình con chỉ cần xử lý socket client.

**Code mẫu:**

```
if (fork() == 0) {
    close(listener); // Tiến trình con đóng socket lắng nghe
    // Xử lý client
}
```

---

## Bài 8: Tạo nhiều tiến trình con liên tiếp

**Yêu cầu:** Viết chương trình tạo 5 tiến trình con liên tiếp để xử lý công việc song song.

**Code mẫu:**

```
for (int i = 0; i < 5; i++) {
    if (fork() == 0) {
        printf("Tiến trình con %d\n", i);
        _exit(0);
    }
}
```



```
    }  
}  
for (int i = 0; i < 5; i++) wait(NULL); // Thu dọn
```

---

## Bài 9: Kiểm tra mô hình fork() trước và sau accept()

**Yêu cầu:** Viết đoạn mã phân biệt mô hình fork trước và sau accept().

**Giải thích:**

- Fork trước accept: dùng để tạo nhiều tiến trình lắng nghe.
  - Fork sau accept: dùng để tạo tiến trình con xử lý client.
- 

## Bài 10: Hiển thị sơ đồ cây tiến trình sau fork()

**Yêu cầu:** Viết chương trình tạo 2 tiến trình con từ tiến trình cha và hiển thị PID từng tiến trình.

**Code mẫu:**

```
pid_t pid1 = fork();  
if (pid1 == 0) {  
    printf("Tiến trình con 1, PID = %d\n", getpid());  
} else {  
    pid_t pid2 = fork();  
    if (pid2 == 0) {  
        printf("Tiến trình con 2, PID = %d\n", getpid());  
    } else {  
        printf("Tiến trình cha, PID = %d\n", getpid());  
        wait(NULL); wait(NULL);  
    }  
}
```

## 4. Nội dung 4: Lập trình mạng đa luồng với pthread

### 4.1. Lý thuyết

#### Giới thiệu về lập trình đa luồng (multi-threaded programming)

- **Luồng (Thread):** Là đơn vị xử lý nhỏ nhất trong một tiến trình (process).
  - **Nhiều luồng trong cùng một tiến trình** có thể **chạy song song**, chia sẻ **cùng không gian bộ nhớ**, bao gồm:
    - **Biến toàn cục.**
    - **Heap.**
    - **Mô tả file/socket descriptors.**
  - **Khác với đa tiến trình (multi-process):**
    - **Các tiến trình** có **không gian bộ nhớ tách biệt**.
    - **Các luồng** chia sẻ bộ nhớ nhưng có **ngăn xếp (stack) riêng** cho từng luồng.
- 

#### Thư viện pthread (POSIX Threads)

- **pthread** là thư viện **chuẩn POSIX** hỗ trợ lập trình đa luồng trong C.
  - **Header:** `#include <pthread.h>`
- 

#### Tạo và quản lý luồng với pthread

##### Tạo luồng:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

- **thread:** Con trỏ tới biến kiểu `pthread_t` (ID của luồng).
- **attr:** Thuộc tính luồng (NULL để mặc định).
- **start\_routine:** Hàm luồng thực thi, có dạng `void *func(void *arg)`.
- **arg:** Tham số truyền vào cho hàm luồng.

##### Giá trị trả về:

- **0:** Thành công.
  - **Khác 0:** Lỗi.
-

## Kết thúc luồng:

- **Tự kết thúc:**
  - `void pthread_exit(void *retval);`
  - **Chờ luồng kết thúc (join):**
  - `int pthread_join(pthread_t thread, void **retval);`
  - **Tách luồng (detached):**
  - `pthread_detach(pthread_t thread);`
    - Khi **detached**, luồng **tự giải phóng tài nguyên** sau khi kết thúc (không cần `pthread_join`).
- 

## Truyền tham số vào luồng

- Dữ liệu truyền vào **hàm luồng** qua **con trỏ void**.
- Thường cần **ép kiểu** về kiểu dữ liệu cụ thể.

### Ví dụ:

```
void *thread_func(void *arg) {
    int *num = (int *)arg;
    printf("Luồng nhận số: %d\\n\\n", *num);
    return NULL;
}
```

---

## Đồng bộ hóa luồng với mutex

- Khi **nhiều luồng cùng truy cập tài nguyên chung** (biến toàn cục, file, socket), cần **đồng bộ hóa** để tránh **xung đột** (race condition).
- **Mutex (Mutual Exclusion)** giúp đảm bảo **chỉ một luồng được truy cập tài nguyên** tại một thời điểm.

### Sử dụng mutex:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);    // Khóa mutex
// Vùng chỉ một luồng truy cập
pthread_mutex_unlock(&mutex);  // Mở khóa mutex
```

---

## Ví dụ server TCP đa luồng sử dụng pthread

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

```

#include <arpa/inet.h>
#include <pthread.h>

void *client_handler(void *arg) {
    int client = *(int *)arg;
    free(arg); // Giải phóng vùng nhớ cấp phát
    char buffer[100];
    read(client, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);
    close(client);
    pthread_exit(NULL);
}

int main() {
    int listener = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);

    bind(listener, (struct sockaddr*)&addr, sizeof(addr));
    listen(listener, 5);

    while (1) {
        int *client = malloc(sizeof(int));
        *client = accept(listener, NULL, NULL);
        pthread_t tid;
        pthread_create(&tid, NULL, client_handler, client);
        pthread_detach(tid); // Tách luồng, không cần join
    }
    return 0;
}

```

---

## Các vấn đề cần lưu ý

- **Chia sẻ bộ nhớ:**
    - **Biến toàn cục** và **heap** được chia sẻ giữa các luồng.
    - **Biến cục bộ** (trong hàm) thuộc về **stack riêng** của mỗi luồng.
  - **Race condition:** Khi nhiều luồng cùng truy cập và thay đổi tài nguyên chia sẻ mà không đồng bộ hóa → gây lỗi logic.
  - **Khi nào cần đồng bộ hóa?**
    - Khi cùng ghi hoặc đọc/ghi vào **biến toàn cục**, **file**, hoặc **socket**.
    - **Không cần đồng bộ hóa** khi các luồng chỉ thao tác với **biến cục bộ**.
- 

## Lưu ý đặc biệt

- **Chia sẻ bộ nhớ:** Các luồng chia sẻ **biến toàn cục**, không chia sẻ **biến cục bộ**.
- **Đồng bộ hóa bằng mutex** để tránh xung đột khi ghi dữ liệu vào file.
- **Sự kiện cần đồng bộ hóa:**

- Ghi vào biến toàn cục hoặc file.
  - Không cần đồng bộ khi chỉ đọc biến toàn cục (trừ khi có luồng ghi song song).
- **Cách kiểm tra sự kiện trong pollfd:**
  - **events** để đăng ký sự kiện.
  - **revents** để kiểm tra sự kiện xảy ra.

## 4.2. Bài tập

### Bài 1: Viết chương trình tạo một luồng đơn giản

**Yêu cầu:** Viết chương trình tạo một luồng mới in ra thông báo "Hello from thread!".

**Code mẫu:**

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

void *thread_func(void *arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread_func, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

---

### Bài 2: Truyền tham số vào luồng

**Yêu cầu:** Viết chương trình truyền một số nguyên vào luồng và in ra giá trị đó.

**Code mẫu:**

```
void *print_number(void *arg) {
    int *num = (int *)arg;
    printf("Number: %d\n", *num);
    return NULL;
}

int main() {
    pthread_t tid;
    int n = 42;
    pthread_create(&tid, NULL, print_number, &n);
}
```

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

---

### Bài 3: Tách luồng (detached thread)

**Yêu cầu:** Viết chương trình tạo một luồng tách biệt (detached), không cần pthread\_join.

**Code mẫu:**

```
pthread_t tid;  
pthread_create(&tid, NULL, thread_func, NULL);  
pthread_detach(tid);  
sleep(1); // Đợi luồng chạy xong
```

---

### Bài 4: Tạo nhiều luồng song song

**Yêu cầu:** Viết chương trình tạo 5 luồng, mỗi luồng in ra chỉ số của nó.

**Code mẫu:**

```
void *print_index(void *arg) {  
    int idx = *(int *)arg;  
    printf("Thread %d\n", idx);  
    return NULL;  
}  
  
int main() {  
    pthread_t tids[5];  
    int indices[5];  
    for (int i = 0; i < 5; i++) {  
        indices[i] = i;  
        pthread_create(&tids[i], NULL, print_index, &indices[i]);  
    }  
    for (int i = 0; i < 5; i++) pthread_join(tids[i], NULL);  
    return 0;  
}
```

---

### Bài 5: Chia sẻ biến toàn cục giữa các luồng

**Yêu cầu:** Viết chương trình các luồng cùng ghi vào biến toàn cục.

**Giải thích:**

- Gây ra race condition nếu không đồng bộ.

**Code mẫu:**

```

int counter = 0;
void *increment(void *arg) {
    for (int i = 0; i < 1000; i++) counter++;
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, increment, NULL);
    pthread_create(&tid2, NULL, increment, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Counter: %d\n", counter);
    return 0;
}

```

---

## Bài 6: Sử dụng mutex để đồng bộ hóa

**Yêu cầu:** Sửa bài 5 bằng cách sử dụng mutex để tránh race condition.

**Code mẫu:**

```

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

```

---

## Bài 7: Viết server TCP đa luồng sử dụng pthread

**Yêu cầu:** Viết server lắng nghe kết nối và tạo luồng để xử lý mỗi client.

**Gợi ý:**

- Dùng `pthread_create()` với mỗi socket client.
- 

## Bài 8: Kiểm tra chia sẻ bộ nhớ giữa các luồng

**Yêu cầu:** Viết chương trình cho thấy các luồng chia sẻ biến toàn cục nhưng có biến cục bộ riêng.

**Gợi ý:**

- In ra địa chỉ biến cục bộ trong từng luồng để thấy sự khác biệt.
- 

## Bài 9: Ghi dữ liệu vào file từ nhiều luồng

**Yêu cầu:** Viết chương trình nhiều luồng ghi vào cùng một file, sử dụng mutex để đồng bộ.

**Gợi ý:**

- Sử dụng `fprintf()` để ghi dữ liệu.
- 

## Bài 10: Dùng `pthread_exit` để kết thúc luồng

**Yêu cầu:** Viết chương trình trong đó luồng tự kết thúc bằng `pthread_exit()`.

**Code mẫu:**

```
void *thread_func(void *arg) {  
    printf("Luồng kết thúc\n");  
    pthread_exit(NULL);  
}
```



## 5. Nội dung 5: Socket cơ bản

### 5.1. Lý thuyết

#### Socket là gì?

- **Socket** là điểm cuối (endpoint) của một kết nối mạng.
- Dùng để giao tiếp giữa:
  - Hai **chương trình** chạy trên **cùng một máy** (Local Inter-Process Communication).
  - Hai **chương trình** chạy trên **các máy khác nhau qua mạng**.
- **Giao thức** thường dùng:
  - **TCP (SOCK\_STREAM)**: Kết nối có kiểm soát, đảm bảo độ tin cậy.
  - **UDP (SOCK\_DGRAM)**: Kết nối không kiểm soát, nhanh, không đảm bảo độ tin cậy.

---

#### Các bước lập trình socket TCP cơ bản

Bước	Server	Client
1. <b>socket()</b>	Tạo socket	Tạo socket
2. <b>bind()</b>	Gán địa chỉ IP + port cho socket	Không cần
3. <b>listen()</b>	Đặt socket vào trạng thái lắng nghe	Không cần
4. <b>accept()</b>	Chấp nhận kết nối từ client	<b>connect()</b> đến server
5. Giao tiếp <b>send()</b> , <b>recv()</b>		<b>send()</b> , <b>recv()</b>
6. Kết thúc <b>close()</b>		<b>close()</b>

---

#### Cú pháp và mô tả các hàm socket cơ bản

##### socket()

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

- **domain:**
  - **AF\_INET**: IPv4.
  - **AF\_INET6**: IPv6.
- **type:**
  - **SOCK\_STREAM**: TCP.
  - **SOCK\_DGRAM**: UDP.
- **protocol**: Thường là **0** (dùng giao thức mặc định theo `type`).

### Ví dụ:

```
int s = socket(AF_INET, SOCK_STREAM, 0); // TCP socket
```

---

### bind()

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

- Gán địa chỉ IP + cổng (port) cho socket server.

### Ví dụ:

```
struct sockaddr_in addr;  
addr.sin_family = AF_INET;  
addr.sin_port = htons(8080); // Port 8080  
addr.sin_addr.s_addr = INADDR_ANY; // Địa chỉ IP bất kỳ  
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

- **Lưu ý về htons():**
    - Chuyển giá trị từ **host byte order** sang **network byte order (big-endian)**.
- 

### listen()

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

- Đặt socket vào trạng thái **lắng nghe**.
  - **backlog**: Số lượng kết nối tối đa được xếp hàng chờ.
- 

### accept()

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

- Chấp nhận kết nối từ client.
  - Tạo ra **socket mới** để giao tiếp với client.
- 

### connect() (client)

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

- Kết nối socket client đến địa chỉ server.
-

## send() và recv()

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- **send()**: Gửi dữ liệu.
  - **recv()**: Nhận dữ liệu.
- 

## close()

```
int close(int sockfd);
```

- Đóng socket.
- 

## Lập trình socket UDP

- Không có **listen()** hay **accept()**.
- Sử dụng **sendto()** và **recvfrom()** thay cho **send()** và **recv()**.

### sendto() và recvfrom():

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

---

## Ví dụ socket TCP client và server đơn giản

### Server TCP:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in addr = {AF_INET, htons(8080), INADDR_ANY};
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
listen(sockfd, 5);
int client = accept(sockfd, NULL, NULL);
send(client, "Hello", 5, 0);
close(client);
close(sockfd);
```

### Client TCP:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in addr = {AF_INET, htons(8080), inet_addr("127.0.0.1")};
connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));
recv(sockfd, buffer, sizeof(buffer), 0);
close(sockfd);
```

---

## Các nội dung cần lưu ý:

- **Quy trình server TCP:** socket -> bind -> listen -> accept.
- **UDP không cần listen() và accept().**
- **connect():** Dùng cho client.
- **sendto() / recvfrom():** Dùng cho UDP.
- **htons():** Chuyển đổi port sang network byte order.

## 5.2. Bài tập

### Bài 1: Tạo socket TCP

**Yêu cầu:** Viết đoạn mã tạo một socket TCP.

**Code mẫu:**

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    perror("socket error");
}
```

---

### Bài 2: Tạo socket UDP

**Yêu cầu:** Viết đoạn mã tạo một socket UDP.

**Code mẫu:**

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd == -1) {
    perror("socket error");
}
```

---

### Bài 3: Gán địa chỉ IP và port cho socket bằng bind()

**Yêu cầu:** Viết đoạn mã gán IP và port cho socket server.

**Code mẫu:**

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
```

```
addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

---

## Bài 4: Đặt socket vào trạng thái lắng nghe với listen()

**Yêu cầu:** Viết đoạn mã cho socket TCP vào trạng thái lắng nghe.

**Code mẫu:**

```
listen(sockfd, 5);
```

---

## Bài 5: Chấp nhận kết nối từ client với accept()

**Yêu cầu:** Viết đoạn mã chấp nhận kết nối từ client.

**Code mẫu:**

```
int clientfd = accept(sockfd, NULL, NULL);
if (clientfd == -1) {
    perror("accept error");
}
```

---

## Bài 6: Kết nối đến server bằng connect()

**Yêu cầu:** Viết đoạn mã để client kết nối đến server.

**Code mẫu:**

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

---

## Bài 7: Gửi dữ liệu với send()

**Yêu cầu:** Viết đoạn mã gửi chuỗi "Hello" qua socket TCP.

**Code mẫu:**

```
char *msg = "Hello";
send(sockfd, msg, strlen(msg), 0);
```

---

## Bài 8: Nhận dữ liệu với recv()

**Yêu cầu:** Viết đoạn mã nhận dữ liệu từ socket TCP.

**Code mẫu:**

```
char buffer[100];
recv(sockfd, buffer, sizeof(buffer), 0);
printf("Received: %s\n", buffer);
```

---

## Bài 9: Gửi dữ liệu UDP với sendto()

**Yêu cầu:** Viết đoạn mã gửi "Hello UDP" đến một địa chỉ IP và port bằng UDP.

**Code mẫu:**

```
char *msg = "Hello UDP";
struct sockaddr_in dest;
dest.sin_family = AF_INET;
dest.sin_port = htons(8080);
dest.sin_addr.s_addr = inet_addr("127.0.0.1");
sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr*)&dest, sizeof(dest));
```

---

## Bài 10: Nhận dữ liệu UDP với recvfrom()

**Yêu cầu:** Viết đoạn mã nhận dữ liệu UDP.

**Code mẫu:**

```
char buffer[100];
struct sockaddr_in src;
socklen_t srclen = sizeof(src);
recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr*)&src, &srclen);
printf("Received UDP: %s\n", buffer);
```

## 6. NỘI DUNG 6: GIAO THỨC HTTP

### 6.1. Lý thuyết

#### 1. HTTP là gì?

- **HTTP (HyperText Transfer Protocol)** là **giao thức tầng ứng dụng** sử dụng để **trao đổi tài nguyên** (web page, hình ảnh, video...) giữa **client (trình duyệt)** và **server (web server)**.
- **Client** gửi **HTTP request** đến **server**, server trả về **HTTP response**.
- **Giao thức không duy trì trạng thái (stateless)**: Mỗi yêu cầu/đáp ứng là **độc lập**, server không lưu trạng thái giữa các yêu cầu.

---

#### 2. Cấu trúc HTTP Request

```
<Phương thức> <Đường dẫn tài nguyên> <Phiên bản HTTP>  
<Header 1>: <Giá trị>  
<Header 2>: <Giá trị>
```

[Body] (nếu có)

##### Ví dụ HTTP GET Request:

```
GET /index.html HTTP/1.1  
Host: www.example.com
```

---

#### 3. Cấu trúc HTTP Response

```
<Phiên bản HTTP> <Mã trạng thái> <Thông điệp>  
<Header 1>: <Giá trị>  
<Header 2>: <Giá trị>
```

[Body] (nếu có)

##### Ví dụ HTTP Response:

```
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 125
```

```
<html>...</html>
```

---

#### 4. Các phương thức HTTP phổ biến

Phương thức	Mục đích
GET	Lấy tài nguyên từ server.

<b>POST</b>	Gửi dữ liệu lên server để xử lý.
<b>PUT</b>	Tải lên hoặc cập nhật tài nguyên.
<b>DELETE</b>	Xóa tài nguyên.
<b>HEAD</b>	Giống GET nhưng chỉ lấy header, không body.

## 5. Mã trạng thái HTTP phổ biến

Mã trạng thái	Ý nghĩa
<b>200 OK</b>	Thành công.
<b>404 Not Found</b>	Không tìm thấy tài nguyên.
<b>500 Internal Server Error</b>	Lỗi phía server.
<b>301 Moved Permanently</b>	Tài nguyên đã chuyển vĩnh viễn.

## 6. Gửi HTTP request bằng ngôn ngữ C

- Giao thức **HTTP hoạt động trên TCP** (port mặc định **80**).
- **Client** cần xây dựng một **TCP connection** tới server và gửi **HTTP request** dưới dạng **chuỗi ký tự**.

### Ví dụ gửi HTTP GET Request trong C:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(80);
addr.sin_addr.s_addr = inet_addr("93.184.216.34"); // IP www.example.com

connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));

// Gửi HTTP GET request
char *request = "GET /index.html HTTP/1.1\r\nHost:
www.example.com\r\n\r\n";
send(sockfd, request, strlen(request), 0);

// Nhận HTTP response
char buffer[4096];
recv(sockfd, buffer, sizeof(buffer), 0);
printf("%s", buffer);

close(sockfd);
```



## 7. Kiểm tra HTTP response với curl

- **curl** là công cụ dòng lệnh dùng để gửi HTTP request.
- **Ví dụ:**
- `curl -I http://localhost:8080/index.html`
- **Ý nghĩa:**
  - `-I`: Gửi **HTTP HEAD request** (chỉ nhận header).

---

## 8. Lưu ý quan trọng:

- **HTTP hoạt động trên TCP, port mặc định là 80.**
- **GET, POST, PUT, DELETE, HEAD** là các phương thức HTTP hợp lệ.
- **UPDATE** không phải phương thức HTTP chuẩn.
- **curl -I** gửi HTTP HEAD request, chỉ nhận **header**.
- Biết cách phân tích **mã trạng thái HTTP** trong response (200 OK, 404 Not Found...).

## 6.2. Bài tập

### Bài 1: Gửi HTTP GET request tới server

**Yêu cầu:** Viết chương trình gửi HTTP GET request tới server và in ra response.

**Code mẫu:**

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in addr = {AF_INET, htons(80), inet_addr("93.184.216.34")};
connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));
char *request = "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n";
send(sockfd, request, strlen(request), 0);
char buffer[4096];
recv(sockfd, buffer, sizeof(buffer), 0);
printf("%s", buffer);
close(sockfd);
```

---

### Bài 2: Gửi HTTP HEAD request bằng curl

**Yêu cầu:** Dùng lệnh `curl` để gửi HTTP HEAD request và phân tích kết quả.

**Lệnh mẫu:**

```
curl -I http://example.com/index.html
```

### Giải thích:

- -I: Gửi **HEAD request** (chỉ lấy header).
- 

### Bài 3: Gửi HTTP POST request trong C

**Yêu cầu:** Viết chương trình gửi HTTP POST request tới server.

#### Gợi ý:

- POST request cần thêm **Content-Length** và **body**.

#### Code mẫu:

```
char *request = "POST /submit HTTP/1.1\r\nHost: example.com\r\nContent-  
Length: 13\r\n\r\nname=Student";  
send(sockfd, request, strlen(request), 0);
```

---

### Bài 4: Phân tích HTTP response header

**Yêu cầu:** Viết đoạn mã tách và in phần **header** trong HTTP response.

#### Gợi ý:

- HTTP header kết thúc bằng `"\r\n\r\n"`.
- 

### Bài 5: Kiểm tra mã trạng thái HTTP trong response

**Yêu cầu:** Viết đoạn mã kiểm tra nếu mã trạng thái HTTP là **200 OK**.

#### Gợi ý:

- Phân tích dòng đầu tiên của HTTP response.
- 

### Bài 6: Viết HTTP server đơn giản trả về response

**Yêu cầu:** Viết server lắng nghe kết nối TCP và trả về HTTP response.

#### Gợi ý:

- Trả về HTTP/1.1 200 OK kèm Content-Type: text/plain.

#### Code mẫu:

```
char *response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 12\r\n\r\nHello world";  
send(clientfd, response, strlen(response), 0);
```

---

### Bài 7: Xử lý nhiều phương thức HTTP

**Yêu cầu:** Viết server đơn giản phân biệt và xử lý **GET** và **POST**.

#### Gợi ý:

- Kiểm tra dòng đầu tiên của request.
- 

### Bài 8: Gửi HTTP PUT request trong C

**Yêu cầu:** Viết chương trình gửi HTTP PUT request cập nhật tài nguyên.

#### Gợi ý:

- PUT tương tự POST nhưng mục đích là **ghi đè tài nguyên**.
- 

### Bài 9: Phân biệt các phương thức HTTP hợp lệ

**Yêu cầu:** Liệt kê các phương thức HTTP hợp lệ: **GET, POST, PUT, DELETE, HEAD**.

#### Gợi ý:

- **UPDATE** không phải phương thức HTTP hợp lệ.
- 

### Bài 10: Viết client HTTP nhận dữ liệu dạng HTML

**Yêu cầu:** Viết chương trình nhận response từ server và lưu vào file **output.html**.

#### Gợi ý:

- Dùng `fwrite()` để ghi dữ liệu vào file.

## 7. Nội dung 7: giao thức FTP

### 7.1. Lý thuyết

#### 1. Giới thiệu về FTP

- **FTP (File Transfer Protocol)** là giao thức tầng ứng dụng sử dụng để **truyền tệp tin** giữa client và server.
- FTP hoạt động qua **TCP** với **hai kết nối**:
  - **Control connection**: Truyền lệnh và phản hồi (**port 21**).
  - **Data connection**: Truyền dữ liệu (file, danh sách file).

#### 2. Chế độ kết nối của FTP

- **Active Mode (PORT)**: Client yêu cầu server kết nối ngược lại vào IP và port do client chỉ định.
- **Passive Mode (PASV)**: Server mở port và yêu cầu client kết nối vào port đó.

#### 3. Các lệnh FTP cơ bản

Lệnh	Mục đích
USER	Gửi username để xác thực
PASS	Gửi password để xác thực
PORT	Client yêu cầu server kết nối data (Active)
PASV	Server chờ client kết nối data (Passive)
LIST	Liệt kê file/directory
RETR	Tải file về client (download)
STOR	Tải file lên server (upload)
RNFR	Đổi tên file (From)
RNTO	Đổi tên file (To)
QUIT	Ngắt kết nối

#### 4. Mã phản hồi FTP phổ biến

Mã	Ý nghĩa
220	Server sẵn sàng
331	Username hợp lệ, yêu cầu password

Mã	Ý nghĩa
230	Xác thực thành công
200	Lệnh thành công
226	Truyền dữ liệu thành công
425	Không thiết lập được kết nối data

## 5. Quy trình FTP client kết nối và gửi lệnh

- Tạo control connection:**
  - TCP connect đến server FTP tại port 21.
  - Server trả lời với mã 220 (Server ready).
- Gửi lệnh xác thực:**
  - Gửi lệnh USER → nhận phản hồi 331.
  - Gửi lệnh PASS → nhận phản hồi 230 (xác thực thành công).
- Thiết lập data connection:**
  - Gửi PORT hoặc PASV để thiết lập kết nối data:
    - PORT: Client cung cấp IP và port, server kết nối ngược lại.
    - PASV: Server cung cấp IP và port, client kết nối vào.
- Gửi lệnh truyền dữ liệu:**
  - LIST, RETR, STOR yêu cầu data connection.
  - Ví dụ gửi LIST để liệt kê file:

```
char *cmd = "LIST\r\n";
send(control_sock, cmd, strlen(cmd), 0);
```
  - Dữ liệu được truyền qua data connection.
- Nhận phản hồi:**
  - Sau khi truyền dữ liệu, server gửi 226 Transfer complete.
- Ngắt kết nối:**
  - Gửi lệnh QUIT để đóng control connection.

## 6. Mã hóa địa chỉ và port trong lệnh PORT/PASV

- Địa chỉ và port được mã hóa dạng: **h1,h2,h3,h4,p1,p2**.
- Port =  $p1 * 256 + p2$ .
- Ví dụ: 103,28,36,99,198,96 → IP 103.28.36.99, port  $198 * 256 + 96 = 50784$ .

## 7. Ví dụ tạo FTP client đơn giản bằng C

```
int control_sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in server_addr = {AF_INET, htons(21),
inet_addr("127.0.0.1")};
connect(control_sock, (struct sockaddr*)&server_addr, sizeof(server_addr));

char *cmd = "USER anonymous\r\n";
send(control_sock, cmd, strlen(cmd), 0);
recv(control_sock, buffer, sizeof(buffer), 0);

cmd = "PASS anonymous\r\n";
send(control_sock, cmd, strlen(cmd), 0);
```

```
recv(control_sock, buffer, sizeof(buffer), 0);  
close(control_sock);
```

## 8. Lưu ý:

- **PORT/PASV** dùng cho **data connection**.
- **LIST/RETR/STOR** yêu cầu **data connection**.
- **RNTO** không yêu cầu data connection (chỉ control).
- Biết giải mã IP và port từ **h1,h2,h3,h4,p1,p2**.