6. 입출력 멀티플렉싱 (I/O Multiplexing)

6.1 `select` 함수 사용법

✅ `select()` 함수란?

`select()`는 여러 파일 디스크립터를 동시에 감시하여,
읽기/쓰기/예외 이벤트가 발생한 디스크립터를 알려주는 시스템 콜이다.

> 🔍 사용 목적: I/O 이벤트가 발생할 때까지 **효율적으로 대기** → 다중 연결 처리 가능

✅ 함수 시그니처

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

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

인자	설명
`nfds`	감시할 **가장 큰 fd + 1**
`readfds`	**읽기 감시** 대상 fd 집합
`writefds`	**쓰기 감시** 대상 fd 집합
`exceptfds`	**예외 감시** 대상 fd 집합
`timeout`	대기 시간 설정 (`NULL`이면 무한 대기)

✅ 주요 매크로

```
FD_ZERO(fd_set *set);          // 집합 초기화
FD_SET(int fd, fd_set *set);   // 집합에 fd 추가
FD_CLR(int fd, fd_set *set);   // 집합에서 fd 제거
FD_ISSET(int fd, fd_set *set); // 이벤트 발생 여부 확인
```

✅ 간단한 예제

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(server_sock, &readfds);  // 서버 소켓 등록
int maxfd = server_sock;

struct timeval timeout;
```

```
timeout.tv_sec = 5;
timeout.tv_usec = 0;

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

if (result == -1) {
    perror("select error");
} else if (result == 0) {
    printf("Timeout occurred\n");
} else {
    if (FD_ISSET(server_sock, &readfds)) {
        int client_sock = accept(server_sock, NULL, NULL);
        // 클라이언트 연결 처리
    }
}
```

✅ 특징 및 한계

장점	단점
모든 유닉스 계열 시스템에서 사용 가능	**fd_set의 크기 제한 (`FD_SETSIZE`)**
간단한 구조	매번 전체 fd 집합을 다시 설정해야 함
낮은 규모의 서버에 적합	고성능 서버에는 비효율적 (선형 스캔, 복사 오버헤드)

> 보통 100~500개 이상의 동시 연결에는 `poll()` 또는 `epoll()`을 추천함.

✅ select 기반 서버 구조 흐름

1. `FD_SET()`으로 읽을 fd들 등록
2. `select()` 호출
3. 반환되면 `FD_ISSET()`으로 이벤트 발생 확인
4. I/O 수행
5. 루프 반복

📌 실전 팁

- `timeout == NULL` → 무한 대기
- `tv_sec = 0, tv_usec = 0` → **non-blocking** select
- 반환값 > 0 → 발생한 이벤트 수

6.2 `poll` 함수 사용법

📘 개요

`poll()` 함수는 다수의 파일 디스크립터를 감시하여 I/O 이벤트가 발생했는지를 확인하는 시스템 콜이다. `select()`와 유사한 기능을 제공하지만, 더 유연하고 파일 디스크립터 수의 제한이 없다는 점에서 `select()`보다 실용적이다. 특히 동시 접속 수가 많은 네트워크 서버 프로그래밍에서 자주 사용된다.

📚 함수 정의

```
#include <poll.h>

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

- `fds` : 감시 대상 파일 디스크립터 배열
- `nfds` : 배열의 크기
- `timeout` : 대기 시간 (단위: 밀리초)
  - `0` : 즉시 반환 (non-blocking)
  - `-1` : 무한 대기

📐 구조체 정의

```
struct pollfd {
    int fd;         // 감시할 파일 디스크립터
    short events;   // 감시할 이벤트 (입력용)
    short revents;  // 발생한 이벤트 (출력용)
};
```

📌 주요 이벤트 플래그

플래그	설명
`POLLIN`	읽기 가능 상태
`POLLOUT`	쓰기 가능 상태
`POLLERR`	오류 발생
`POLLHUP`	연결 종료 또는 끊김
`POLLNVAL`	잘못된 파일 디스크립터

`events` 필드는 감시 대상 이벤트를, `revents` 필드는 실제로 발생한 이벤트를 나타낸다.

🧪 사용 예시

다음 코드는 표준 입력(STDIN)에 대해 5초 동안 입력 대기를 수행하고, 데이터가 입력되면 이를 읽어 출력한다.

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

int main() {
    struct pollfd fds[1];
    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    int ret = poll(fds, 1, 5000); // 5000ms = 5초 대기

    if (ret == -1) {
        perror("poll");
    } else if (ret == 0) {
        printf("Timeout\n");
    } else {
        if (fds[0].revents & POLLIN) {
            char buf[128];
            ssize_t len = read(STDIN_FILENO, buf, sizeof(buf));
            write(STDOUT_FILENO, buf, len);
        }
    }
    return 0;
}
```

🔬 `select()`와의 비교

항목	`select()`	`poll()`
FD 개수 제한	있음 (`FD_SETSIZE`, 보통 1024)	없음
감시 구조	`fd_set` 비트마스크 기반	`pollfd` 배열 기반
이벤트 정보 처리	`FD_ISSET()` 필요	`revents`로 즉시 확인 가능
구조의 간결함	매 호출마다 `fd_set` 초기화 필요	구조체 배열만 갱신

💡 구현 및 운용상의 주의점

- `poll()`은 내부적으로 선형 순회(linear scan)를 수행하므로, 파일 디스크립터 수가 수천 개 이상일 경우 성능 저하가 발생할 수 있다. 이 경우 `epoll`의 도입이 적절하다.
- 반환값이 양수일 경우, `revents` 필드를 반드시 확인하여 실제 이벤트 발생 여부를 판별해야 한다.
- `timeout`은 밀리초 단위로 설정되며, non-blocking 처리를 위해 `0`, 무한 대기를 위해 `-1`을 설정할 수 있다.

6.3 `epoll` 구조와 사용법

📘 개요

`epoll`(event poll)은 리눅스에서 다수의 파일 디스크립터를 효율적으로 감시하기 위해 도입된 이벤트 기반 I/O 감시 메커니즘이다. `select()`와 `poll()`은 모든 파일 디스크립터를 매번 순회해야 하므로 수천 개 이상의 소켓을 감시할 경우 성능이 급격히 저하된다. `epoll`은 **O(1)** 수준의 성능으로 이벤트를 감지하며, 이벤트 기반 고성능 서버에서 핵심적으로 사용된다.

📚 핵심 특징

- 감시 대상 등록 → 감시 실행 → 이벤트 수신의 세 단계 구조
- **수정 없는 감시 유지** (등록된 이벤트는 내부 커널 공간에 저장)
- **엣지 트리거(Edge-Triggered)**, **레벨 트리거(Level-Triggered)** 모드 지원
- 최대 수십만 개의 파일 디스크립터 감시 가능

🧱 구조적 흐름

```
int epfd = epoll_create1(0); // epoll 인스턴스 생성

struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = socket_fd;

epoll_ctl(epfd, EPOLL_CTL_ADD, socket_fd, &ev); // 소켓 등록

struct epoll_event events[MAX_EVENTS];
int nfds = epoll_wait(epfd, events, MAX_EVENTS, timeout); // 이벤트 대기

for (int i = 0; i < nfds; ++i) {
    // events[i].data.fd 를 기반으로 처리
}
```

📌 주요 API

🔹 `epoll_create1()`

```
int epoll_create1(int flags);
```

- `flags` : `0` 또는 `EPOLL_CLOEXEC`
- 반환값: epoll 인스턴스 디스크립터 (`epfd`)

🔹 `epoll_ctl()`

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

- `op` : 감시 작업 종류 (`EPOLL_CTL_ADD` , `EPOLL_CTL_MOD` , `EPOLL_CTL_DEL`)
- `fd` : 대상 파일 디스크립터
- `event` : 감시할 이벤트 정보

🔹 `epoll_wait()`

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

- `epfd` : epoll 인스턴스
- `events` : 이벤트 발생 정보를 저장할 배열
- `maxevents` : 처리할 최대 이벤트 수
- `timeout` : 대기 시간(ms), -1은 무한 대기

🎯 이벤트 플래그

플래그	설명
`EPOLLIN`	읽기 이벤트
`EPOLLOUT`	쓰기 이벤트
`EPOLLERR`	오류 발생
`EPOLLHUP`	연결 끊김
`EPOLLET`	엣지 트리거 모드(Edge Triggered)
`EPOLLONESHOT`	1회성 이벤트 감시

💡 Edge Triggered vs Level Triggered

구분	Level Triggered (기본)	Edge Triggered (`EPOLLET`)
동작 방식	데이터가 남아있을 때마다 알림	새로운 데이터가 도착했을 때만 알림
반복 호출 필요성	낮음	높음 (`read()` 반복 필요)
성능	안정적이나 반복 이벤트 처리 불리함	성능 우수, 세심한 처리 필요

🧪 예제: epoll을 이용한 단일 소켓 감시

```c
#include <sys/epoll.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_EVENTS 10

int main() {
    int epfd = epoll_create1(0);
    struct epoll_event ev, events[MAX_EVENTS];

    int sockfd = /* 소켓 생성 및 바인딩 */;
    ev.events = EPOLLIN;
    ev.data.fd = sockfd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &ev);

    while (1) {
        int nfds = epoll_wait(epfd, events, MAX_EVENTS, -1);
        for (int i = 0; i < nfds; ++i) {
            if (events[i].events & EPOLLIN) {
                // 데이터 수신 처리
            }
        }
    }

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

🚨 주의 사항

- Edge Triggered 모드 사용 시, 반드시 `read()` 나 `recv()` **를 반복 호출**하여 버퍼를 비워야 한다.
- 각 파일 디스크립터는 **비동기(non-blocking) 모드**로 설정되어 있어야 한다.
- 이벤트 제거 시 `EPOLL_CTL_DEL`을 명시적으로 호출해야 한다.

📊 성능 및 적용 분야

항목	설명
감시 가능한 소켓 수	이론상 무제한 (단, 커널 자원 제한 적용됨)
적합한 용도	대규모 멀티접속 서버, 웹 서버, 채팅 서버 등
스케일링 방식	내부 커널 데이터 구조 (Red-Black Tree 기반)

6.4 `epoll`의 LT(Level-Triggered) vs ET(Edge-Triggered)

📘 개요

`epoll`은 이벤트 트리거 방식으로 두 가지 모드를 지원한다.

- **Level-Triggered (LT)**: 기본 동작 모드
- **Edge-Triggered (ET)**: 성능 최적화를 위해 제공되는 고급 모드

이 두 방식은 I/O 이벤트가 감지되고 처리되는 타이밍과 방식에서 본질적인 차이를 보이며, 정확한 이해 없이 `ET` 모드를 사용할 경우 데이터 유실, 무한 대기 등의 심각한 문제가 발생할 수 있다.

🧱 개념적 정의

항목	Level-Triggered (LT)	Edge-Triggered (ET)
동작 방식	버퍼에 데이터가 **남아있는 동안 반복적으로** 이벤트 발생	데이터가 **새로 도착한 순간에 한 번만** 이벤트 발생
반복 호출 필요성	필요 없음 (한 번 읽고 나면 나중에 다시 호출 가능)	필요 있음 (`read()` 또는 `recv()` 등으로 **버퍼 완전히 비움**) 필수
누락 위험	낮음 (계속 알림 받음)	높음 (한 번만 알림 → 데이터 남아 있어도 **추가 알림 없음**)
디폴트 모드	예 (명시하지 않으면 기본값)	아니오 (`EPOLLET` 플래그 필요)
I/O 처리 방식	단순하고 안전함	복잡하나 고성능

🧪 예시 비교

[LT 모드 예시]

```
struct epoll_event ev;
ev.events = EPOLLIN;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

- 버퍼에 데이터가 남아있는 동안 `epoll_wait()`는 반복적으로 해당 `fd`에 대해 이벤트를 반환한다.

[ET 모드 예시]

```
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

- 데이터가 **도착한 순간 딱 한 번만** 이벤트 발생.
- 이후에는 **반드시 read/recv를 루프 돌려서 버퍼를 완전히 비워야** 한다.

🔄 실전 처리 방식

ET 모드에서는 일반적으로 다음과 같은 루프 구조가 필요하다:

```
while (1) {
    ssize_t n = recv(fd, buf, sizeof(buf), 0);
    if (n == -1) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) break;
        else {
            // 에러 처리
        }
    } else if (n == 0) {
        // 연결 종료
        break;
    } else {
        // 받은 데이터 처리
    }
}
```

※ `recv()`가 `-1`을 반환하고 `errno == EAGAIN`일 때까지 반복해야 안전하다.

💡 성능 관점

기준	LT	ET
시스템 콜 호출 수	상대적으로 많음 (`epoll_wait()` 반복 호출)	적음 (이벤트 발생 시에만 진입)
CPU 효율성	낮음 (불필요한 감시 반복)	높음 (불필요한 wake-up 방지)
프로그래밍 난이도	쉬움	높음 (논블로킹, 버퍼 처리 루프 필요)

🚨 주의 사항

- **ET 모드는 반드시 소켓을 `non-blocking` 모드로 설정**해야 한다. (`fcntl(fd, F_SETFL, O_NONBLOCK)`)
- 버퍼를 다 읽지 않으면 이후에는 **이벤트가 발생하지 않으며**, 해당 fd는 **영원히 블로킹될 수 있음**.
- `EPOLLONESHOT` 플래그와 함께 사용할 경우, 처리 완료 후 반드시 `epoll_ctl(..., EPOLL_CTL_MOD, ...)`로 이벤트를 다시 등록해야 한다.

📌 추천 사용 시나리오

상황	추천 트리거 방식
단순한 소규모 서버	LT
고성능 서버 (수천~수만 소켓 감시)	ET

상황	추천 트리거 방식
멀티스레드 서버에서 1회성 이벤트 필요	ET + ONESHOT

🧠 요약 정리

항목	LT (기본)	ET (고급)
이벤트 발생 빈도	조건 지속 시 반복 발생	변화가 있을 때만 단발 발생
설계 난이도	낮음	높음
성능 효율	보통	높음
데이터 유실 가능성	낮음	높음 (버퍼 미처리 시)
활용도	일반 서버, 데모	대규모 서버, 고성능 시스템

6.5 고성능 서버를 위한 `epoll` 기반 구조 설계

📘 목적

`epoll`은 수천~수만 개의 클라이언트 접속을 효율적으로 처리할 수 있도록 설계된 리눅스 커널 수준의 I/O 이벤트 감시 메커니즘이다. 고성능 네트워크 서버에서는 `epoll`의 구조적 특성과 트리거 방식(ET/LT), 병렬 처리 전략, 자원 재사용 기법을 모두 결합하여 높은 처리량과 안정성을 확보한다.

🧱 서버 아키텍처 구성 요소

1. 비차단 소켓 (Non-blocking Socket)

- 모든 소켓은 `fcntl(fd, F_SETFL, O_NONBLOCK)`을 통해 논블로킹으로 설정

2. epoll 인스턴스

- `epoll_create1(0)`으로 생성

3. 이벤트 루프

- `epoll_wait()`에 의해 감지된 이벤트에 따라 적절한 작업 분기

4. 이벤트 핸들러 분리

- `accept()`, `read()`, `write()` 등 이벤트 유형별 처리 함수 분기

5. 자원 재사용

- `thread pool`, `connection pool`, `object pool` 등을 활용한 효율적 메모리 관리

🖼️ 구조 다이어그램 (논리 흐름)

```
[Client] --TCP--> [Server Socket]
                        ↓
                  [epoll 인스턴스]
                        ↓
                ┌─────────────────┐
                | epoll_wait()  |
                └─────────┬───────┘
                        ↓
             +--------+--------+
             |  Accept 이벤트   |
             |  Read 이벤트     |
             |  Write 이벤트    |
             +--------+--------+
                        ↓
                  [Thread Pool]
                        ↓
                  [비즈니스 로직]
```

⚙️ 핵심 설계 요소

1. 논블로킹 소켓 설정

```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- ET 모드에서는 **반드시 필수**이다.

---

2. epoll 인스턴스 생성 및 등록

```
int epfd = epoll_create1(0);

struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;
ev.data.fd = listen_fd;

epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &ev);
```

---

3. 이벤트 루프

```
struct epoll_event events[MAX_EVENTS];
while (1) {
    int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
    for (int i = 0; i < n; i++) {
        if (events[i].data.fd == listen_fd) {
            // 신규 연결 수락 루틴
            accept_connection(epfd, listen_fd);
        } else {
            // 클라이언트 데이터 수신 루틴
            handle_client(epfd, events[i].data.fd);
        }
    }
}
```

---

4. 연결 수락 (`accept()`) 및 등록

```
void accept_connection(int epfd, int listen_fd) {
    while (1) {
        int client_fd = accept(listen_fd, NULL, NULL);
        if (client_fd == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK)
                break;
            else
                perror("accept");
        }

        fcntl(client_fd, F_SETFL, O_NONBLOCK);

        struct epoll_event ev;
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = client_fd;
        epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &ev);
    }
}
```

---

5. 클라이언트 핸들링 (`read`, `write`)

- **ET 모드에서는 반드시 루프 처리 필요**

```
void handle_client(int epfd, int client_fd) {
    char buf[4096];
    while (1) {
        ssize_t count = read(client_fd, buf, sizeof(buf));
        if (count == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK)
                break; // 더 이상 읽을 데이터 없음
            else {
```

```
                close(client_fd);
                break;
            }
        } else if (count == 0) {
            close(client_fd); // 연결 종료
            break;
        } else {
            // 수신 데이터 처리 (예: echo)
            write(client_fd, buf, count);
        }
    }
}
```

🔄 스레드 풀 병행 처리

- `epoll_wait()` 는 단일 스레드에서 처리하되, 수신된 `fd` 는 **비동기 큐**를 통해 `worker thread` 에게 위임
- C 기반 경량 스레드 풀 구현 또는 POSIX `pthread` + `queue` 활용

🧠 고급 전략 요약

항목	권장 설계
이벤트 감시 방식	`epoll` + `ET`
소켓 설정	`O_NONBLOCK` 필수
이벤트 처리 구조	이벤트 루프 + 핸들러 분기
처리 병렬화	Thread Pool 또는 Task Queue
FD 최대값 대비 설계	메모리 풀, FD 재사용 전략
성능 최적화 요소	CPU affinity, zero-copy (`sendfile`), `TCP_NODELAY`

🛠️ 실무 적용 예시

- Nginx, HAProxy 등은 `epoll` + `ET` + `worker process` 기반 구조를 사용
- DB connection pool, task queue, timer wheel 등과 결합하여 고성능 이벤트 기반 애플리케이션 구현 가능