8. 고급 I/O 및 논블로킹 처리

8.1 `non-blocking I/O` 설정 (`O_NONBLOCK`)

📘 개요

리눅스에서 기본적으로 대부분의 소켓 및 파일 디스크립터는 **블로킹(Blocking)** 모드로 동작한다.
즉, `read()`, `write()`, `recv()`, `send()`와 같은 시스템 호출은 데이터가 준비될 때까지 **스레드를 중단(sleep)**시킨다.

Non-blocking I/O는 시스템 호출이 즉시 반환되도록 설정하며,
`EAGAIN` **또는** `EWOULDBLOCK` **에러를 통해 상태를 점검**하는 방식으로 이벤트 기반 처리 구조를 가능하게 한다.

🛠️ 설정 방법: `fcntl()`을 통한 `O_NONBLOCK` 플래그 적용

```
#include <fcntl.h>

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

- `F_GETFL`: 기존 플래그를 읽어옴
- `O_NONBLOCK`: 논블로킹 모드 지정
- `F_SETFL`: 플래그 설정

📂 예시 코드: Non-blocking 소켓 설정

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

int sock = socket(AF_INET, SOCK_STREAM, 0);

// 논블로킹 설정
int flags = fcntl(sock, F_GETFL, 0);
fcntl(sock, F_SETFL, flags | O_NONBLOCK);
```

이제 `connect()`, `recv()`, `send()` 등의 함수 호출이 **즉시 반환**된다.

🔁 논블로킹 I/O의 동작 방식

- `read()` / `recv()`
  - 읽을 데이터가 없으면 `-1` 반환 및 `errno == EAGAIN` 또는 `EWOULDBLOCK`
- `write()` / `send()`
  - 버퍼가 가득 차면 `-1` 반환 및 동일한 `errno`

→ **이 에러를 처리하여 루프 내에서 재시도하거나, epoll/select 등으로 이벤트를 감지**하는 구조를 만들 수 있음

❗ 주의점

- 논블로킹 모드에서는 **모든 입출력 함수가 예외 처리를 전제로 설계**되어야 한다.
- `EAGAIN`은 오류가 아닌 "아직 준비되지 않음"의 의미이므로 **재시도 로직 또는 이벤트 대기**가 필요하다.
- 논블로킹 connect()는 즉시 반환되고, `select()` 또는 `epoll`로 연결 완료 여부를 확인해야 한다.

🧠 실무 활용 포인트

- `O_NONBLOCK`은 **epoll, select, poll 기반의 이벤트 기반 서버 구현의 전제 조건**
- 수천 개의 클라이언트를 동시에 처리하는 고성능 서버에 필수
- **표준 입력(stdin)**도 논블로킹으로 설정 가능

📊 블로킹 vs 논블로킹 비교

항목	블로킹 (기본)	논블로킹 (`O_NONBLOCK`)
동작 방식	데이터 준비 전까지 대기	즉시 반환 (`EAGAIN`)
CPU 효율	낮음 (불필요한 sleep)	높음 (event loop 설계 가능)
구현 난이도	쉬움	비교적 복잡 (에러 처리 필수)
활용 시나리오	단일/소수 클라이언트	수천 개 동시 연결, 비동기 처리

8.2 `read`, `write`의 반환값 처리

📘 개요

`read()`와 `write()`는 C 언어에서 파일, 소켓, 파이프 등 다양한 **입출력 스트림에 대한 저수준 I/O 함수**이다.
이 함수들은 항상 **반환값(return value)을 확인**해야 하며, 이 값에 따라 에러, EOF, 논블로킹 상태 등을 구분할 수 있다.

🔹 함수 원형

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd` : 파일 디스크립터
- `buf` : 읽거나 쓸 버퍼
- `count` : 읽거나 쓸 바이트 수
- **반환값**: 실제로 읽거나 쓴 바이트 수, 또는 음수(`-1`)는 에러

🔍 `read()` 반환값의 의미

반환값	의미
`> 0`	읽은 바이트 수
`== 0`	**EOF (End of File)**: 연결 종료, 파일 끝 등
`< 0`	오류. `errno` 값을 조사해야 함 (`EINTR`, `EAGAIN` 등 포함)

```
ssize_t n = read(fd, buf, sizeof(buf));
if (n == 0) {
    // 연결 종료됨 (예: 소켓 닫힘)
} else if (n < 0) {
    if (errno == EINTR) {
        // 시그널에 의해 인터럽트됨 → 다시 시도
    } else if (errno == EAGAIN || errno == EWOULDBLOCK) {
        // 논블로킹 소켓에서 데이터가 아직 없음
    } else {
        // 기타 오류
    }
}
```

🔍 `write()` 반환값의 의미

반환값	의미
`> 0`	실제로 전송된 바이트 수
`< 0`	오류. `errno` 확인 필요 (`EINTR`, `EPIPE`, `EAGAIN` 등)

```c
ssize_t n = write(fd, buf, len);
if (n < 0) {
    if (errno == EPIPE) {
        // 상대방이 이미 연결을 종료함
    } else if (errno == EAGAIN) {
        // 논블로킹 모드에서 버퍼가 꽉 찼음
    } else if (errno == EINTR) {
        // 인터럽트 → 다시 시도 가능
    }
}
```

💡 부분 읽기/쓰기 (Partial Read/Write)

특히 **네트워크나 파일 디스크립터는 요청한 바이트 수만큼 항상 처리되지 않는다**.
→ 루프를 사용해 반복적으로 처리하는 것이 안전한 방식이다.

예: 안전한 전체 쓰기 함수

```c
ssize_t safe_write(int fd, const void *buf, size_t len) {
    size_t total = 0;
    const char *p = buf;

    while (total < len) {
        ssize_t n = write(fd, p + total, len - total);
        if (n < 0) {
            if (errno == EINTR) continue;
            return -1;
        }
        total += n;
    }
    return total;
}
```

📌 실무에서의 처리 전략

상황	처리 방법
반복 호출로 전체 데이터 처리	`while (total < expected)` 루프 사용
`EAGAIN` 발생 시	`epoll/select` 또는 재시도 로직 적용
`EINTR` 발생 시	`continue`로 재시도
`read()` 결과가 0	상대방 종료 또는 EOF 처리
`write()` 후 `EPIPE`	소켓이 끊어진 상태 → 종료 처리 필요

📋 종합 요약

- `read()` / `write()`는 **부분적 성공이 가능**하므로 반드시 반환값 확인
- 에러 발생 시 `errno` **값에 따라 로직 분기**
- `EAGAIN` / `EWOULDBLOCK`은 비동기 I/O에서의 자연스러운 상태
- 전체 전송 보장을 위해 **loop 기반 처리 함수 구현 필수**

8.3 `recv`, `send`의 `MSG_DONTWAIT`, `MSG_PEEK` 옵션

📘 함수 원형

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

- `flags` 인자를 통해 동작을 세밀하게 제어할 수 있음
- 소켓 통신에서 **입출력 흐름을 유연하게 다루는 수단**으로 사용됨

✅ `MSG_DONTWAIT`: 비동기 호출 플래그

의미

- 시스템 호출을 **비차단(non-blocking)** 방식으로 동작하게 함
- 소켓이 블로킹 모드일지라도, 해당 호출은 **즉시 반환**

동작 방식

- 소켓이 준비되지 않은 경우 `recv()` 또는 `send()`는 `-1`을 반환하며, `errno`는 `EAGAIN` 또는 `EWOULDBLOCK`으로 설정됨

예시 코드

```
char buffer[1024];
ssize_t n = recv(sock, buffer, sizeof(buffer), MSG_DONTWAIT);
if (n < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
    // 데이터가 아직 없음 → 이벤트 대기 또는 재시도
}
```

용도

- 이벤트 기반 서버에서 `recv()`를 논블로킹으로 처리할 때
- **epoll/select**과 함께 사용하여 busy loop 방지

✅ `MSG_PEEK`: 데이터 미소비(Peek) 기능

의미

- 소켓 수신 버퍼에 있는 데이터를 **읽지 않고 복사만** 함
- 즉, 다음 `recv()` 시 동일한 데이터를 다시 받을 수 있음

동작 방식

- 커널 소켓 수신 큐에서 데이터를 **소비하지 않고 미리보기**

예시 코드

```
char peek_buf[4];
ssize_t n = recv(sock, peek_buf, sizeof(peek_buf), MSG_PEEK);
if (n > 0) {
    // peek_buf는 여전히 커널 내부에 존재
    // 다음 recv()로 같은 데이터를 재수신 가능
}
```

용도

- **프로토콜 헤더 검사** 시 유용 (예: 고정된 4바이트 헤더 확인)
- 수신 대기 데이터의 종류를 먼저 판별 후 처리 흐름 결정

✅ 플래그 조합 사용 예

```
recv(sock, buffer, sizeof(buffer), MSG_PEEK | MSG_DONTWAIT);
```

- 수신 버퍼를 **즉시 검사하면서 읽지는 않음**
- 데이터가 없으면 `EAGAIN` 반환
- 데이터가 있으면 **읽지 않고 미리보기**

📋 종합 요약 비교

플래그	의미	주 용도
`MSG_DONTWAIT`	비동기 처리, 블로킹 방지	이벤트 루프, 논블로킹 입출력
`MSG_PEEK`	버퍼 미소비 읽기	프로토콜 헤더 분석, 데이터 검사

📌 주의 사항

- `MSG_PEEK`는 수신 큐의 데이터를 소비하지 않기 때문에, **지속적으로 호출 시 중복 처리 주의** 필요
- `MSG_DONTWAIT`는 소켓 상태와 커널 버퍼 상황에 따라 즉시 반환되므로, **적절한 예외 처리 로직** 필수

8.4 `ioctl`을 이용한 소켓 제어

📘 개요

`ioctl()` 함수는 장치나 파일 디스크립터에 대한 **입출력 제어**를 수행하는 범용 인터페이스로, 소켓에도 활용된다.
일반적인 함수 호출로 제어할 수 없는 **저수준 속성 및 상태 질의/설정**에 사용된다.

✅ 함수 원형

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

int ioctl(int fd, unsigned long request, ...);
```

- `fd`: 제어 대상 파일 또는 소켓의 디스크립터
- `request`: 제어할 동작 종류를 나타내는 매크로
- 추가 인자: 요청에 따라 포인터나 정수 등 가변 인자

✅ 주요 소켓 관련 `request` 매크로

매크로 명	설명
`FIONREAD`	수신 버퍼에 남은 데이터 바이트 수 조회
`FIONBIO`	소켓을 논블로킹 모드로 설정 (`O_NONBLOCK` 과 유사)
`SIOCGIFCONF`	네트워크 인터페이스 목록 가져오기
`SIOCGIFFLAGS`	인터페이스 플래그 가져오기 (UP, RUNNING 등)
`SIOCSIFFLAGS`	인터페이스 플래그 설정

🔍 예제 1: 수신 버퍼에 남은 데이터 확인 (`FIONREAD`)

```
int nbytes = 0;
if (ioctl(sock, FIONREAD, &nbytes) == 0) {
    printf("읽을 수 있는 바이트 수: %d\n", nbytes);
}
```

- 소켓 수신 큐에 **얼마나 많은 데이터가 쌓여 있는지** 확인 가능
- 데이터 수신 시점 결정 또는 `recv()` 버퍼 크기 결정에 사용됨

🔍 예제 2: 논블로킹 모드 설정 (`FIONBIO`)

```
int enable = 1; // 1로 설정하면 논블로킹 모드
ioctl(sock, FIONBIO, &enable);
```

- `fcntl()`과 동일하게 소켓을 비동기(논블로킹)으로 설정할 수 있음
- 내부적으로는 `O_NONBLOCK` 플래그를 설정하는 것과 유사

🔍 예제 3: 인터페이스 목록 가져오기 (`SIOCGIFCONF`)

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

struct ifconf ifc;
struct ifreq ifr[10];  // 인터페이스 최대 10개 가정
int sock = socket(AF_INET, SOCK_DGRAM, 0);

ifc.ifc_len = sizeof(ifr);
ifc.ifc_req = ifr;

ioctl(sock, SIOCGIFCONF, &ifc);

for (int i = 0; i < ifc.ifc_len / sizeof(struct ifreq); ++i) {
    printf("인터페이스 이름: %s\n", ifr[i].ifr_name);
}
```

- 시스템의 **활성화된 네트워크 인터페이스 목록**을 가져오는 데 유용
- 커맨드라인 `ifconfig`의 내부 동작과 유사

📌 `ioctl` vs `fcntl`

구분	ioctl	fcntl
목적	범용적인 I/O 제어	파일 디스크립터 속성 설정 중심
인터페이스	다양한 장치 및 네트워크 제어	파일/소켓의 동작 제어
활용 예	FIONREAD, SIOCGIFCONF 등	O_NONBLOCK, FD_CLOEXEC 등

⚠️ 주의사항

- `ioctl()`은 매우 강력한 만큼, **매크로 종류에 따라 인자의 형식과 크기**를 정확히 맞춰야 함
- 잘못된 호출은 `EINVAL`, `ENOTTY`, `EPERM` 등의 오류로 이어질 수 있음
- 일부 플랫폼에서는 `ioctl()`보다 `getsockopt()` 또는 `fcntl()`을 권장하기도 함

🧩 실제 사용 시기

- **정밀한 네트워크 상태 점검**이 필요할 때 (예: 대기 중 데이터 확인)
- **논블로킹 제어**를 시스템 호출 단위로 처리할 때
- **인터페이스/주소/플래그 정보 조회**가 필요할 때

8.5 파일 디스크립터 기반 이벤트 루프 구성

📘 개요

네트워크 프로그래밍에서 여러 클라이언트와의 입출력을 **동시에 처리**하기 위해 이벤트 루프(event loop)가 필요하다. 이벤트 루프는 **파일 디스크립터(file descriptor, FD)**의 상태를 감시하여, **준비된 작업만 비동기적으로 처리**하는 구조를 말한다.

이 방식은 고성능 서버나 비동기 IO 처리의 핵심이다.

✅ 기본 개념

구성 요소	설명
파일 디스크립터	커널이 열어놓은 소켓, 파일, 파이프 등 리소스를 참조하는 정수 값
이벤트 감시	`select()`, `poll()`, `epoll()` 등의 시스템 콜을 이용해 FD 상태를 감시
이벤트 처리 루프	감시 중인 FD에 이벤트가 발생하면 해당 작업(수신, 송신, 연결 등) 수행

🔁 일반적인 이벤트 루프 흐름

1. 소켓 생성 및 바인딩
2. 관심 이벤트 등록 (읽기/쓰기 등)
3. `select`, `poll`, `epoll`로 이벤트 대기
4. 이벤트가 발생한 FD를 순회하며 처리
5. 처리 후 다시 이벤트 대기

🧪 간단한 `select()` 기반 이벤트 루프 예제

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

void event_loop(int listen_fd) {
    fd_set readfds;
    int maxfd = listen_fd;

    while (1) {
        FD_ZERO(&readfds);
```

```
        FD_SET(listen_fd, &readfds);

        int ret = select(maxfd + 1, &readfds, NULL, NULL, NULL);
        if (ret < 0) {
            perror("select error");
            break;
        }

        if (FD_ISSET(listen_fd, &readfds)) {
            int client_fd = accept(listen_fd, NULL, NULL);
            printf("클라이언트 접속: %d\n", client_fd);
            // 클라이언트 FD를 감시 목록에 추가하는 방식 필요
        }
    }
}
```

⚙️ 이벤트 루프 설계 요소

항목	설명
감시 목록 갱신	새로운 클라이언트가 접속하면 FD 집합에 추가해야 함
이벤트 종류 분기	읽기/쓰기/에러 이벤트에 따라 다르게 처리
타임아웃 설정	비정상 연결을 감지하거나 주기적인 작업을 위해 필요
클라이언트 FD 관리	배열, `fd_set`, 연결 리스트 등으로 FD 관리 구조 설계 필요

🧠 `epoll` 기반 이벤트 루프 고도화 (개요)

- `epoll_create`, `epoll_ctl`, `epoll_wait`를 이용한 확장 가능한 이벤트 감시
- 수천 개의 FD를 감시해도 높은 성능을 유지
- 비동기 I/O 서버의 기본 구조

```
// 구조화된 epoll 루프 (단순화)
epoll_fd = epoll_create1(0);
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &event);
while (1) {
    int n = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
    for (int i = 0; i < n; ++i) {
        int fd = events[i].data.fd;
        if (fd == listen_fd)
            accept_new_client();
        else
            handle_client_event(fd);
    }
}
```

📌 이벤트 루프와 파일 디스크립터 비교

모델	감시 방식	성능 특성	확장성
`select()`	비트맵 기반	FD 수 증가 시 성능 저하	낮음
`poll()`	배열 기반	선형 스캔	중간
`epoll()`	커널 레벨 이벤트 큐	이벤트 기반 알림	매우 높음

📎 정리 요약

- 이벤트 루프는 **서버에서 동시성 처리를 위한 핵심 구조**
- 파일 디스크립터를 감시하여 **비효율적인 블로킹 I/O를 방지**
- 구조를 잘 설계하면 수천 개 이상의 클라이언트 연결도 처리 가능
- 서버의 성능을 좌우하는 요소이므로 `epoll` 기반으로의 확장이 필수적