UNIX IO Multiplexing

`select`, `poll`, `epoll`, 그리고 `io-uring`은 모두 유닉스 계열 운영 체제에서 **입출력 멀티플렉싱**(I/O Multiplexing)을 처리하는 시스템 콜 또는 메커니즘임.

- `epoll`, `io-uring`은 리눅스에서만 지원됨

입출력 멀티플렉싱은 여러 파일 디스크립터(예: 네트워크 소켓, 파일, 파이프 등)를 동시에 모니터링하면서, 읽기나 쓰기 작업을 비동기적으로 처리할 수 있도록 도움. 이것은 서버가 많은 연결을 효율적으로 처리하거나 여러 파일에서 동시에 데이터를 읽고 쓸 때 매우 유용함.

각각의 메커니즘은 발전 과정에서 성능과 확장성을 개선해왔으며, 아래에서 각 방식에 대해 자세히 설명함.

1. `select`

`select`는 가장 오래된 I/O 멀티플렉싱 방식으로, **4.2BSD**에서 처음 도입되었으며, POSIX 표준에도 포함됨.

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

- 매개변수
  1. **`nfds`** : 감시할 파일 디스크립터 중 가장 큰 값에 1을 더한 값.
  2. **`readfds`** : 읽기 가능 상태를 감시할 파일 디스크립터의 집합.
  3. **`writefds`** : 쓰기 가능 상태를 감시할 파일 디스크립터의 집합.
  4. **`exceptfds`** : 예외 상태(에러)를 감시할 파일 디스크립터의 집합.
  5. **`timeout`** : 기다릴 최대 시간. 0이면 즉시 반환, `NULL`이면 무한 대기.
- 반환값
  - **양수**: 읽기, 쓰기, 예외 상태에 있는 파일 디스크립터 중 하나 이상의 상태가 변경된 개수
  - **0**: 타임아웃이 발생함.
  - **-1**: 오류 발생.

주요 특징:

- **작동 방식**:
  - 사용자는 읽기, 쓰기, 오류를 감시하고 싶은 파일 디스크립터 집합을 `select` 시스템 콜에 전달함.

    - 커널은 이 파일 디스크립터 집합을 순차적으로 검사하여 준비된(ready) 파일 디스크립터를 찾아냄.
- **파일 디스크립터 한계**:
    - `select`는 감시할 수 있는 파일 디스크립터 수에 제한이 있음.
    - 이 한도는 보통 `FD_SETSIZE`로 정의되며, 일반적으로 1024로 설정되어 있음.
        - 이 한도를 넘기면 추가적인 관리를 해야 함.
- **성능**:
    - `select`는 파일 디스크립터를 모두 순차적으로 검사하기 때문에 O(n) 복잡도를 가지며, 많은 파일 디스크립터를 처리하는 데 비효율적임.
- **블로킹**:
    - `select`는 지정한 시간 동안 대기할 수 있으며, 이 시간이 끝나거나 감시 중인 파일 디스크립터 중 하나가 준비되면 반환됨.

사용 예시:

```
fd_set readfds;  // 읽기 상태를 감시할 fd_set
FD_ZERO(&readfds);  // fd_set 초기화
FD_SET(server_fd, &readfds);  // 서버 소켓을 감시할 fd_set에 추가

int activity = select(server_fd + 1, &readfds, NULL, NULL, NULL);  // 블로킹 대기

if (activity > 0) {
    if (FD_ISSET(server_fd, &readfds)) {
        // 서버 소켓에 읽을 준비가 되었음을 의미 (새 클라이언트 연결)
    }
} else if (activity == 0) {
    // 타임아웃 발생
} else {
    // select() 오류 발생
}
```

2. `poll`

`poll`는 `select`의 한계를 개선하기 위해 도입된 방식으로, 주로 **System V UNIX**에서 사용됨. 파일 디스크립터 수에 제한이 없다는 점에서 `select`보다 유리함.

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

- 매개변수

1. `fds`: 감시할 파일 디스크립터들의 배열 (`pollfd` 구조체 배열).
    - 각 배열 항목은 감시할 파일 디스크립터와 이벤트 정보를 포함합니다.
    - **구조체**:

```
struct pollfd {
    int fd;         // 감시할 파일 디스크립터
    short events;   // 감시할 이벤트 (읽기, 쓰기, 예외 등)
    short revents;  // 이벤트 결과 (발생한 이벤트)
};
```

- `events`에 감시하고자 하는 이벤트를 설정할 수 있습니다:
    - `POLLIN`: 읽기 준비 완료.
    - `POLLOUT`: 쓰기 준비 완료.
    - `POLLERR`: 에러 발생 감시.
- `nfds`: 감시할 파일 디스크립터의 개수 (즉, `fds[]` 배열의 크기).
- `timeout`: 기다릴 최대 시간(밀리초 단위).
    - **0**: 즉시 반환.
    - **-1**: 무한 대기(이벤트가 발생할 때까지 기다림).
- 반환값
    - **양수**: 상태가 변경된 파일 디스크립터의 개수 (읽기, 쓰기, 예외 발생).
    - **0**: 타임아웃이 발생함 (변경된 파일 디스크립터가 없음).
    - **-1**: 오류 발생.

주요 특징:

- **작동 방식**:
    - `poll`은 파일 디스크립터를 배열 형태로 전달하며, 각 파일 디스크립터는 읽기, 쓰기, 오류 상태를 감시할 수 있음.
    - 커널은 이 배열을 순차적으로 검사하여 준비된 파일 디스크립터를 찾음.
- **파일 디스크립터 한계**:
    - `poll`은 `select`처럼 파일 디스크립터 수에 대한 고정된 제한은 없지만, 여전히 많은 파일 디스크립터를 다룰 때는 비효율적일 수 있음.
- **성능**:
    - 파일 디스크립터를 순차적으로 검사하는 방식 때문이며, O(n) 복잡도를 가짐
    - 파일 디스크립터의 수가 많아질수록 성능이 저하됨. 이는
- **블로킹**:
    - `poll` 역시 지정한 시간 동안 대기할 수 있으며, 준비된 파일 디스크립터가 있으면 바로 반환됨.

사용 예시:

```
...
    int server_fd, new_socket;
    struct pollfd fds[MAX_CLIENTS];

    // 서버 소켓 생성
    // 서버 주소 설정
    // 서버 소켓을 주소에 바인딩
    // 클라이언트 연결 대기 상태로 설정

    // 파일 디스크립터 초기화 (pollfd 구조체 배열 설정)
    for (int i = 0; i < MAX_CLIENTS; i++) {
        fds[i].fd = -1;  // 비어 있는 것으로 설정
        fds[i].events = POLLIN;  // 읽기 가능 상태 감시
    }

    fds[0].fd = server_fd;  // 서버 소켓 파일 디스크립터 추가
    fds[0].events = POLLIN;  // 서버 소켓에 대해 읽기 상태 감시

    while (1) {
        int activity = poll(fds, MAX_CLIENTS, -1);  // 이벤트 대기 (무한 대기)

        if (activity < 0) {
            perror("poll error");
            exit(EXIT_FAILURE);
        }

        // 서버 소켓에서 새 연결 요청 처리
        if (fds[0].revents & POLLIN) {
            new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
            if (new_socket < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
            }

            printf("New connection from %s:%d\n",
                   inet_ntoa(address.sin_addr), ntohs(address.sin_port));

            // 클라이언트 소켓을 감시할 pollfd에 추가
            for (int i = 1; i < MAX_CLIENTS; i++) {
                if (fds[i].fd == -1) {
                    fds[i].fd = new_socket;
                    fds[i].events = POLLIN;  // 읽기 이벤트 감시
                    break;
```

```
                }
            }
        }

        // 각 클라이언트 소켓에 대해 이벤트 처리
        for (int i = 1; i < MAX_CLIENTS; i++) {
            if (fds[i].fd != -1 && (fds[i].revents & POLLIN)) {
                int valread = read(fds[i].fd, buffer, BUFFER_SIZE);
                if (valread == 0) {
                    // 클라이언트 연결 종료
                    close(fds[i].fd);
                    fds[i].fd = -1;
                    printf("Client disconnected\n");
                } else {
                    // 받은 데이터를 클라이언트로 에코
                    buffer[valread] = '\0';
                    printf("Received: %s\n", buffer);
                    send(fds[i].fd, buffer, valread, 0);
                }
            }
        }
    }

    return 0;
}
```

3. `epoll`

`epoll`은 리눅스에서 도입된 멀티플렉싱 방식으로, **Linux 2.6** 버전에서 처음 소개됨.
`select`와 `poll`의 단점을 해결하기 위해 설계된 고성능 I/O 처리 방식임.

주요 특징:

- **작동 방식**:
  - `epoll`은 파일 디스크립터를 커널 내의 epoll 인스턴스라는 구조체에 등록하고, 사용자는 파일 디스크립터를 `epoll_ctl()` 시스템 콜을 통해 추가하거나 제거함.
    - epoll 인스턴스: 감시할 파일 디스크립터들을 등록하는 관심 목록과 읽기, 쓰기, 예외 상태에 있는 파일 디스크립터들이 추가되는 준비 목록을 관리
  - 이를 통해 커널이 내부적으로 파일 디스크립터의 준비 상태를 추적하므로, 매번 파일 디스크립터를 순차적으로 검사할 필요가 없음. 준비된 파일 디스크립터가 있을 때만 이벤트가 발생함.
- **성능**:

- `epoll`은 O(1) 복잡도를 가지며, 많은 수의 파일 디스크립터를 처리할 때도 성능이 뛰어남. 커널이 이벤트 기반으로 동작하므로, 파일 디스크립터의 상태가 변경될 때만 사용자에게 알려줌.
  - **Level-triggered (LT)**: 기본 모드로, 파일 디스크립터가 준비될 때마다 계속 알림을 줌.
  - **Edge-triggered (ET)**: 상태가 변할 때만 한 번 알림을 줌. 더 높은 성능을 제공하지만, 사용자 코드에서 더 신중한 관리가 필요함.
- **파일 디스크립터 한계**:
  - `epoll`은 이론적으로 감시할 수 있는 파일 디스크립터 수에 제한이 없으며, 확장성에 유리함.

---

1. `int epoll_create1(int flags);`

- **매개변수**:
  - `flags` : epoll 인스턴스를 생성할 때 설정하는 플래그.
    - `EPOLL_CLOEXEC` : 파일 디스크립터가 자식 프로세스에 상속되지 않도록 설정.
- **설명**: `epoll_create1()`은 epoll 인스턴스를 생성하고 epoll 파일 디스크립터를 반환합니다. 이는 epoll에서 사용할 파일 디스크립터를 생성하는 첫 단계입니다.
- **반환값**:
  - **양수**: epoll 인스턴스에 대한 파일 디스크립터.
  - **-1**: 오류 발생.

---

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

- **매개변수**:
  1. `epfd` : `epoll_create1()`을 통해 생성된 epoll 인스턴스의 파일 디스크립터.
  2. `op` : epoll 인스턴스에 대한 작업 종류 (감시할 파일 디스크립터에 대한 작업).
     - `EPOLL_CTL_ADD` : 새로운 파일 디스크립터를 추가.
     - `EPOLL_CTL_MOD` : 이미 등록된 파일 디스크립터를 수정.
     - `EPOLL_CTL_DEL` : 파일 디스크립터를 삭제.
  3. `fd` : 추가, 수정 또는 삭제할 파일 디스크립터.
  4. `event` : 감시할 이벤트를 정의하는 `epoll_event` 구조체 포인터.
     - **구조체**:

```
struct epoll_event {
    uint32_t events;    // 감시할 이벤트 (읽기, 쓰기, 예외 등)
    epoll_data_t data;  // 사용자 정의 데이터
};
```

    - `events` : 감시할 이벤트 (읽기, 쓰기, 예외 발생).
        - `EPOLLIN` : 읽기 준비 완료.
        - `EPOLLOUT` : 쓰기 준비 완료.
        - `EPOLLERR` : 오류 발생 감시.
        - `EPOLLET` : 엣지 트리거 방식으로 설정 (기본은 레벨 트리거).
- **설명**: `epoll_ctl()` 는 epoll 인스턴스에 파일 디스크립터를 추가, 수정, 삭제할 때 사용합니다.
- **반환값**:
    - **0**: 성공.
    - **-1**: 오류 발생.

---

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

- **매개변수**:
    1. `epfd` : epoll 인스턴스의 파일 디스크립터.
    2. `events` : 이벤트가 발생한 파일 디스크립터들을 저장할 `epoll_event` 구조체 배열.
        - 각 배열 항목에 이벤트가 발생한 파일 디스크립터와 이벤트 정보가 채워짐.
        - **구조체**:

```
struct epoll_event {
    uint32_t events;    // 발생한 이벤트 (읽기, 쓰기, 예외 등)
    epoll_data_t data;  // 사용자 정의 데이터 (예: fd)
};
```

    3. `maxevents` : `events[]` 배열의 크기, 즉 한 번에 처리할 수 있는 최대 이벤트 수.
    4. `timeout` : 기다릴 최대 시간(밀리초 단위).
        - **0**: 즉시 반환.
        - **-1**: 이벤트가 발생할 때까지 무한 대기.
        - **양수**: 지정된 시간 동안 이벤트가 발생할 때까지 대기.
- **설명**: `epoll_wait()` 는 epoll 인스턴스에 등록된 파일 디스크립터에서 발생한 이벤트들을 대기하고, 발생한 이벤트들을 처리합니다.
- **반환값**:
    - **양수**: 이벤트가 발생한 파일 디스크립터의 개수.
    - **0**: 타임아웃 발생 (변경된 파일 디스크립터가 없음).
    - **-1**: 오류 발생.

사용 예시:

```
#include <sys/epoll.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>

#define MAX_EVENTS 10
#define PORT 12345

int main() {
    int server_fd, new_socket, epfd;
    struct sockaddr_in address;
    struct epoll_event ev, events[MAX_EVENTS];
    int addrlen = sizeof(address);

    // 서버 소켓 생성
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 10);

    // epoll 인스턴스 생성
    epfd = epoll_create1(0);
    ev.events = EPOLLIN;  // 읽기 이벤트 감시
    ev.data.fd = server_fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);  // 서버 소켓을 epoll에 추가

    while (1) {
        int nfds = epoll_wait(epfd, events, MAX_EVENTS, -1);  // 이벤트 대기

        for (int i = 0; i < nfds; i++) {
            if (events[i].data.fd == server_fd) {
                // 서버 소켓에 클라이언트 연결 요청
                new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
                ev.events = EPOLLIN;  // 새 클라이언트 소켓에서 읽기 이벤트 감시
                ev.data.fd = new_socket;
                epoll_ctl(epfd, EPOLL_CTL_ADD, new_socket, &ev);  // 새 클라이
언트 소켓 추가
            } else {
```

```
                // 클라이언트로부터 데이터 수신
                char buffer[1024] = {0};
                int valread = read(events[i].data.fd, buffer, 1024);
                if (valread == 0) {
                    // 클라이언트가 연결을 끊음
                    close(events[i].data.fd);
                } else {
                    printf("Received: %s\n", buffer);
                    send(events[i].data.fd, buffer, strlen(buffer), 0);  // 에코 메시지 전송
                }
            }
        }
    }

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

4. `io-uring`

`io-uring`은 리눅스 커널 **5.1**에서 도입된 최신 I/O 시스템 콜로, **고성능 비동기 I/O**를 지원하는 방식임. 이는 전통적인 I/O 시스템 콜 방식보다 더 적은 시스템 콜과 메모리 복사로 I/O 작업을 처리할 수 있음.

`io-uring`은 사용자 공간과 커널 공간에 공유되는 두 개의 링 버퍼(SQ: Submission Queue, CQ: Completion Queue)를 사용하여 I/O 작업을 비동기적으로 처리함.

- **서브미션 큐(SQ)**: 사용자 공간에서 커널로 I/O 작업 요청을 보내기 위한 큐.
- **컴플리션 큐(CQ)**: 커널에서 처리한 I/O 작업 결과를 사용자에게 알리기 위한 큐.

사용자는 I/O 요청을 SQ에 제출하고, 완료된 작업은 CQ에서 확인할 수 있음. 이 방식은 시스템 콜(메모리 복사) 오버헤드를 줄여주며, 많은 수의 I/O 작업을 효율적으로 처리할 수 있음.

- 다중 CPU 환경에서는 메모리 일관성을 보장하기 위해 **메모리 장벽(memory barriers)**을 사용
- **메모리 장벽**: 커널과 사용자 공간 간의 데이터 일관성을 유지하는 메커니즘
    - **다중 코어 시스템**에서 메모리 작업이 재정렬되는 것을 방지하여 **데이터 손상** 방지

동작 과정

1. **큐 설정**:

- `io_uring_setup(2)`과 `mmap(2)`을 사용하여 서브미션 큐(SQ)와 컴플리션 큐(CQ)를 설정하고, 이를 사용자 공간에서 매핑합니다.

2. **I/O 요청 작성**:
   - I/O 요청을 수행하기 위해 **서브미션 큐 엔트리(SQE)**에 요청 내용을 작성합니다. 여기서 각 I/O 요청은 기존의 시스템 호출과 동일한 작업(예: 파일 읽기, 쓰기, 소켓 연결 수락 등)을 수행할 수 있습니다.
3. **서브미션 큐에 요청 추가**:
   - 여러 I/O 요청을 큐에 추가할 수 있으며, 요청이 추가되면 커널에 이를 알리기 위해 `io_uring_enter(2)`를 호출합니다.
4. **커널에서 I/O 처리**:
   - 커널은 서브미션 큐에서 I/O 요청을 처리하고, 처리 결과를 컴플리션 큐에 기록합니다. I/O가 완료되면 **컴플리션 큐 엔트리(CQE)**에 완료된 요청 정보를 반환합니다.
5. **결과 확인**:
   - CQ에서 CQE를 읽어 요청의 결과를 확인합니다. 각 CQE는 성공 또는 실패 여부를 나타내며, `res` 필드를 통해 시스템 호출의 반환 값이나 에러 코드를 확인할 수 있습니다.

주요 특징:

- **제로 복사 I/O**: `io-uring`은 네트워크 및 파일 I/O에서 제로 복사(zero-copy)를 지원하여 성능을 더욱 향상시킬 수 있음.
- **배칭과 확장성**: 여러 I/O 작업을 배치 처리할 수 있으며, 커널이 이를 병렬적으로 처리할 수 있어 대량의 동시 작업을 처리하는 데 매우 유리함. 이로 인해 `epoll`보다도 높은 성능을 발휘할 수 있음.
- **시스템 호출 감소**: `io_uring`은 여러 I/O 요청을 한 번에 처리할 수 있어 시스템 호출을 최소화합니다. 기존의 동기 I/O나 다른 비동기 I/O 방법에서는 각 요청마다 시스템 호출이 필요하지만, `io_uring`은 **한 번의 호출로 여러 요청을 처리**할 수 있습니다.
- **성능**: `io-uring`은 기존의 모든 멀티플렉싱 방식보다 뛰어난 성능을 제공하며, 특히 고성능 네트워크 및 파일 I/O에 적합함.

---

3. 핵심 API 설명

1. io_uring_setup

```
int io_uring_setup(unsigned entries, struct io_uring_params *params);
```

- **설명**: `io_uring` 인스턴스를 생성하고, 필요한 큐(서브미션 큐와 컴플리션 큐)를 설정합니다. 이 함수는 커널과 사용자 간의 링 버퍼를 구성합니다.

- **매개변수**:
  - `entries` : 큐에 사용할 최대 엔트리 수.
  - `params` : 큐 설정에 필요한 파라미터 정보가 포함된 구조체.
- **반환값**: 성공 시 `io_uring` 인스턴스에 대한 파일 디스크립터, 실패 시 -1.

2. io_uring_enter

```
int io_uring_enter(int ring_fd, unsigned int to_submit, unsigned int
min_complete, unsigned int flags);
```

- **설명**: 사용자 공간에서 서브미션 큐에 추가된 요청을 커널에 알리고, 커널에서 처리된 요청의 완료를 기다리는 함수입니다. 옵션에 따라 최소 몇 개의 작업이 완료될 때까지 대기할 수 있습니다.
- **매개변수**:
  - `ring_fd` : `io_uring` 인스턴스의 파일 디스크립터.
  - `to_submit` : 큐에 제출할 요청 수.
  - `min_complete` : 처리된 요청 중 최소 몇 개가 완료될 때까지 대기할지 설정.
  - `flags` : 큐에 설정할 플래그 (`IORING_ENTER_GETEVENTS` 등).
- **반환값**: 성공 시 0, 실패 시 -1.

3. io_uring_register

```
int io_uring_register(int ring_fd, unsigned opcode, const void *arg,
unsigned nr_args);
```

- **설명**: `io_uring` 인스턴스에 파일 디스크립터나 메모리와 같은 자원을 등록하는 데 사용됩니다. 이를 통해 파일 디스크립터나 메모리 버퍼를 고정시킬 수 있습니다.
- **매개변수**:
  - `ring_fd` : `io_uring` 인스턴스의 파일 디스크립터.
  - `opcode` : 등록할 작업의 종류 (`IORING_REGISTER_FILES` , `IORING_REGISTER_BUFFERS` 등).
  - `arg` : 등록할 자원의 주소.
  - `nr_args` : 등록할 자원의 개수.
- **반환값**: 성공 시 0, 실패 시 -1.

---

4. 서브미션 큐(SQ)와 컴플리션 큐(CQ) 구조

- **서브미션 큐(SQ)**: 사용자가 I/O 요청을 작성하고 이를 큐에 추가합니다. 이 큐는 **커널이 읽고 처리**할 수 있도록 제공됩니다.
- **컴플리션 큐(CQ)**: 커널이 완료된 I/O 작업의 결과를 기록하는 큐입니다. 사용자는 이 큐에서 **결과를 확인**할 수 있습니다.

각 큐는 **헤드와 테일 포인터**로 관리되며, **링 버퍼** 구조로 동작합니다.

서브미션 큐 엔트리(SQE) 구조

```
struct io_uring_sqe {
    __u8    opcode;         // I/O 작업 종류 (읽기, 쓰기 등)
    __s32   fd;             // 파일 디스크립터
    __u64   off;            // 파일 오프셋
    __u64   addr;           // 버퍼 주소
    __u32   len;            // 버퍼 크기
    __u64   user_data;      // 사용자 정의 데이터
    __u8    flags;          // 추가 플래그
};
```

- **opcode**: I/O 작업의 종류 (예: 읽기, 쓰기).
- **fd**: I/O 작업을 수행할 파일 디스크립터.
- **off**: 파일 읽기/쓰기의 오프셋.
- **addr**: 데이터가 저장될 버퍼의 주소.
- **len**: 버퍼의 크기.
- **user_data**: 사용자 정의 데이터로, 완료 시 CQE에서 확인할 수 있습니다.

컴플리션 큐 엔트리(CQE) 구조

```
struct io_uring_cqe {
    __u64     user_data;  // SQE에서 설정한 사용자 정의 데이터
    __s32     res;        // I/O 요청 결과 (성공 시 값, 실패 시 에러 코드)
    __u32     flags;
};
```

- **user_data**: 서브미션 큐에서 설정된 데이터가 그대로 반환됩니다.
- **res**: I/O 작업의 결과를 나타내는 값입니다. 시스템 콜 성공 시 처리된 바이트 수, 실패 시 음수의 에러 코드가 반환됩니다.

References

- https://smileostrich.tistory.com/entry/What-is-IOuring-Inside-IOuring
- https://man7.org/linux/man-pages/man7/epoll.7.html
- https://man7.org/linux/man-pages/man7/io_uring.7.html