Chapter 17

select보다 나은 epoll

select 기반의 IO 멀티플렉싱이 느린 이유

- select 함수의 단점

  - select 함수 호출 이후에 모든 파일 디스크립터를 대상으로 반복문을 이용하여 검사
  - select 함수를 호출할 때마다 인자로 관찰 대상에 대한 정보 전달

  – 운영체제의 커널에 의한 기능이 아닌, 순수한 함수에 의해 완성되는 기능
  – select 함수 호출을 통해 전달된 정보는 운영체제에 등록되지 않음
  – select 함수를 호출할 때마다 매번 관련 정보를 전달해야 됨

- select 함수 단점의 해결책

  - 운영체제에게 관찰 대상에 대한 정보를 딱 한번만 알려줌
  - 관찰 대상의 범위 또는 내용에 변경이 있을 때 변경 사항만 알려주는 방식

  – select 함수의 단점을 극복하기 위해서는 운영체제 레벨에서 멀티플렉싱 기능을 지원해야 됨
  – Linux의 epoll, Windows의 IOCP 사용

epoll 방식의 장점

- select 방식이 사용되기 위한 조건

  - 서버의 접속자 수가 많지 않은 경우
  - 다양한 운영체제에서 운영이 가능해야 됨

  – 운영체제 레벨이 아닌, 함수 레벨에서 완성되는 기능으로 호환성이 상대적으로 좋음
  – 리눅스의 epoll 기반 서버를 윈도우의 IOCP 기반으로 변경하는 것은 단순하지 않음
    ➢ 리눅스의 select 기반 서버를 윈도우의 select 기반 서버로 변경하는 것은 단순함

- epoll의 장점

  - 상태 변환의 확인을 위해 전체 파일 디스크립터를 검사하기 위한 반복문이 필요 없음
  - epoll_wait 함수 호출 시, 관찰 대상의 정보를 매번 전달할 필요가 없음

  – 상태 변화를 관찰하는데 더 나은 방법을 제공
  – 커널에서 상태 정보를 유지하기 때문에, 관찰 대상의 정보를 매번 전달하지 않아도 됨

  - 절대 우위를 점하는 서버 모델은 없음
  - 상황에 맞게 적절한 모델을 선택해서 사용해야 됨

epoll 구현에 필요한 함수와 구조체

- epoll 구현에 필요한 함수

```
• epoll_create(): epoll 파일 디스크립터 저장소 생성
• epoll_ctl(): 저장소에 파일 디스크립터 등록 및 삭제
• epoll_wait(): select() 함수와 마찬가지로 파일 디스크립터의 변화를 대기
```

- epoll 관련 구조체

```
struct epoll_event
{
    __uint32_t events;
    epoll_data_t data;
}
```

events: 이벤트 유형 등록

```
typedef union epoll_data
{
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

fd: 파일(소켓) 디스크립터 등록

epoll_create() 함수

- epoll_create(size) 함수

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

int epoll_create(int size);
-> 성공 시 epoll 파일 디스크립터, 실패 시 -1 반환
```

  - epoll 인스턴스 생성
  - size는 참고용: 커널 내부에서 epoll 인스턴스의 크기 조절
  - 운영체제가 관리하는 파일 디스크립터 저장소 생성

- 소멸시 close() 함수 호출을 통한 종료 과정이 필요

```
int epfd = epoll_create(100);
           . . .

close(epfd);
```

epoll_ctl() 함수

- epoll_ctl()
  - 생성된 epoll 인스턴스에 관찰 대상을 저장하고 삭제하는 함수
    - 두 번째 전달인자에 따라 등록, 삭제 및 변경이 이루어짐

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

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
-> 성공 시 0, 실패 시 -1 반환
```

      - epfd: 관찰 대상을 등록할 epoll 인스턴스의 파일 디스크립터
      - op: 관찰 대상의 추가, 삭제 또는 변경 여부 지정
      - fd: 등록할 관찰 대상의 파일 디스크립터
      - event: 관찰 대상의 관찰 이벤트 유형

  - 두 번째 전달인자(int op)
    - EPOLL_CTL_ADD: 파일 디스크립터를 epoll 인스턴스에 등록
    - EPOLL_CTL_DEL: 파일 디스크립터를 epoll 인스턴스에서 삭제
    - EPOLL_CTL_MOD: 등록된 파일 디스크립터의 이벤트 발생 상황을 변경

epoll_ctl() 함수 기반의 디스크립터 등록

- epoll_ctl() 인자

```
epoll_ctl(A, EPOLL_CTL_ADD, B, C);
```

- epoll 인스턴스 A에, 파일 디스크립터 B 등록
- C를 통해 전달된 이벤트 관찰 목적

```
epoll_ctl(A, EPOLL_CTL_DEL, B, NULL);
```

- epoll 인스턴스 A에서 파일 디스크립터 B 삭제

- epoll_event 구조체
  - 상태 변화가 발생한 파일 디스크립터를 묶는 용도로 사용

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

이벤트(EPOLLIN) 및 파일 디스크립터 등록 과정

  - 파일 디스크립터를 epoll 인스턴스에 등록
    - ➢ sockfd에 이벤트(수신 데이터 존재)를 등록

epoll event 유형

- event 유형
  - 비트 OR 연산을 사용해서 둘 이상의 event 유형을 함께 등록할 수 있음

event 유형	설명
• EPOLLIN	• 수신할 데이터가 존재하는 상황
• EPOLLOUT	• 출력 버퍼가 비워져서 당장 데이터를 전송할 수 있는 상황
• EPOLLPRI	• OOB 데이터가 수신된 상황
• EPOLLRDHUP	• 연결이 종료되거나 Half-close가 진행된 상황 • 이는 엣지 트리거 방식에서 유용하게 사용
• EPOLLERR	• 에러가 발생한 상황
• EPOLLET	• 이벤트 감지를 엣지 트리거 방식으로 동작 시킴
• EPOLLONESHOT	• 이벤트가 한 번 감지되면, 해당 파일 디스크립터에서는 더 이상 이벤트를 발생시키지 않음 • epoll_ctl() 함수의 두 번째 인자로 EPOLL_CTL_MOD을 전달해서 이벤트를 재설정해야 됨

이벤트 유형 더 많음!!

epoll.h (/usr/include/x86_64-linux-gnu/sys/epoll.h)

```
enum EPOLL_EVENTS
{
    EPOLLIN = 0x001,
#define EPOLLIN EPOLLIN
    EPOLLPRI = 0x002,
#define EPOLLPRI EPOLLPRI
    EPOLLOUT = 0x004,
#define EPOLLOUT EPOLLOUT
    EPOLLRDNORM = 0x040,
#define EPOLLRDNORM EPOLLRDNORM
    EPOLLRDBAND = 0x080,
#define EPOLLRDBAND EPOLLRDBAND
    EPOLLWRNORM = 0x100,
#define EPOLLWRNORM EPOLLWRNORM
    EPOLLWRBAND = 0x200,
#define EPOLLWRBAND EPOLLWRBAND
    EPOLLMSG = 0x400,
#define EPOLLMSG EPOLLMSG
    EPOLLERR = 0x008,
#define EPOLLERR EPOLLERR
    EPOLLHUP = 0x010,
#define EPOLLHUP EPOLLHUP
    EPOLLRDHUP = 0x2000,
#define EPOLLRDHUP EPOLLRDHUP
    EPOLLEXCLUSIVE = 1u << 28,
#define EPOLLEXCLUSIVE EPOLLEXCLUSIVE
    EPOLLWAKEUP = 1u << 29,
#define EPOLLWAKEUP EPOLLWAKEUP
    EPOLLONESHOT = 1u << 30,
#define EPOLLONESHOT EPOLLONESHOT
    EPOLLET = 1u << 31
#define EPOLLET EPOLLET
};
```

```
#define EPOLL_CTL_ADD 1
#define EPOLL_CTL_DEL 2
#define EPOLL_CTL_MOD 3

typedef union epoll_data
{
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

struct epoll_event
{
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
}
```

kalloc / malloc

32 bit events

int fd

https://stackoverflow.com/questions/62355395/what-does-it-mean-by-define-x-x

epoll_wait() 함수 ⇒ select()

- epoll_wait()
  - epoll 인스턴스에 등록된 파일 디스크립터를 대상으로 이벤트 발생 유무를 확인하는 함수

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

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
-> 성공 시 이벤트가 발생한 파일 디스크립터의 수, 실패 시 -1 반환
```

✓ ⇐ events의 크기

  - epfd: 이벤트 발생의 관찰 영역인 epoll 인스턴스의 파일 디스크립터
  - events: 이벤트가 발생한 파일 디스크립터가 저장될 버퍼의 주소
    - 이벤트가 발생한 파일 디스크립터가 저장되므로, 전체 파일 디스크립터 대상의 반복문은 필요 없음
  - maxevents: 두 번째 인자로 전달된 버퍼에 등록 가능한 최대 이벤트 수
  - timeout: 1/1000초 단위의 대기시간, -1 전달 시, 이벤트가 발생할 때까지 무한 대기

```
int event_cnt;
struct epoll_event *ep_events;
. . .
ep_events = malloc(sizeof(struct poll_event) * EPOLL_SIZE);
. . .
event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
```

고수님 추천, 배열 저장됨

epoll_wait() 반환값
- 이벤트가 발생한 파일 디스크립터의 수

epoll 기반의 에코 서버 #1 (기본 모델)

- 리스닝 소켓 등록과 연결 요청 대기

```
epfd = epoll_create(EPOLL_SIZE);
ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

event.events = EPOLLIN;
event.data.fd = serv_sock;
epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);

while(1)
{
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);

    if(event_cnt==-1)
    {
        puts("epoll_wait() error");
        break;
    }

    for(i=0; i < event_cnt; i++)
    {
        . . . // 뒷 페이지 코드 부분
    }
}
```

event.events = EPOLLIN
- 수신한 데이터가 존재하는지 체크

- epoll 서버의 기본 모델
  - 리스닝 소켓으로 전달되는 연결 요청도 수신 데이터의 일종
  - 리스닝 소켓을 epoll 인스턴스에 등록
  - EPOLLIN을 등록

epoll 기반의 에코 서버 #2

```
for(i=0; i < event_cnt; i++)
{
    if(ep_events[i].data.fd == serv_sock)
    {
        adr_sz = sizeof(clnt_adr);
        clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
        event.events = EPOLLIN;
        event.data.fd = clnt_sock;
        epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
        printf("connected client: %d \n", clnt_sock);
    }
    else
    {
        str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
        if(str_len == 0) // close request!
        {
            epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
            close(ep_events[i].data.fd);
            printf("closed client: %d \n", ep_events[i].data.fd);
        }
        else
        {
            write(ep_events[i].data.fd, buf, str_len); // echo!
        }
    }
}
```

클라이언트 소켓을 epoll 인스턴스에 추가

연결 종료 요청의 경우,
소켓 디스크립터의 해제 과정을 수행

메시지를 수신한 경우, 에코 처리함

epoll 기반 에코서버: echo_epollserv.c #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#define BUF_SIZE 100
#define EPOLL_SIZE 50
void error_handling(char *buf);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t adr_sz;
    int str_len, i;
    char buf[BUF_SIZE];

    struct epoll_event *ep_events;
    struct epoll_event event;
    int epfd, event_cnt;

    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
```

```c
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");

    if(listen(serv_sock, 5)==-1)
        error_handling("listen() error");

    epfd = epoll_create(EPOLL_SIZE);
    ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

    event.events = EPOLLIN;
    event.data.fd = serv_sock;

    epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
```

추가

- epoll 생성(50개) 및 serv_sock 추가
- serv_sock으로 수신되는 데이터를 확인: EPOLLIN 등록

epoll 기반 에코서버: echo_epollserv.c #2

```
while(1)
{
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);

    if(event_cnt==-1)
    {
        puts("epoll_wait() error");
        break;
    }

    for(i=0; i < event_cnt; i++)
    {
        if(ep_events[i].data.fd == serv_sock)
        {
            adr_sz = sizeof(clnt_adr);
            clnt_sock = accept(serv_sock,
                        (struct sockaddr*)&clnt_adr, &adr_sz);
            event.events = EPOLLIN;
            event.data.fd = clnt_sock;

            epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
            printf("connected client: %d \n", clnt_sock);
        }
        else
        {
            str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
            if(str_len == 0) // close request!
            {
                epoll_ctl(epfd, EPOLL_CTL_DEL,
                          ep_events[i].data.fd, NULL);
                close(ep_events[i].data.fd);
                printf("closed client: %d\n", ep_events[i].data.fd);
            }
            else
            {
                write(ep_events[i].data.fd, buf, str_len);// echo!
            }
        }
    }
    close(serv_sock);
    close(epfd);
    return 0;
}

void error_handling(char *buf)
{
    fputs(buf, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

client가 접속 종료한 경우, epoll 인스턴스에서 삭제

clnt_sock을 epoll에 추가

echo_epollserv.c

서버 실행 화면

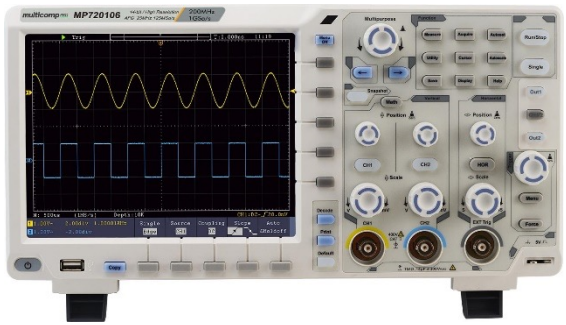
```
$ ./echo_epollserv 9190
connected client: 5
closed client: 5
```

파일 디스크립터
- 3: serv_sock
- 4: epfd
- 5: clnt_sock

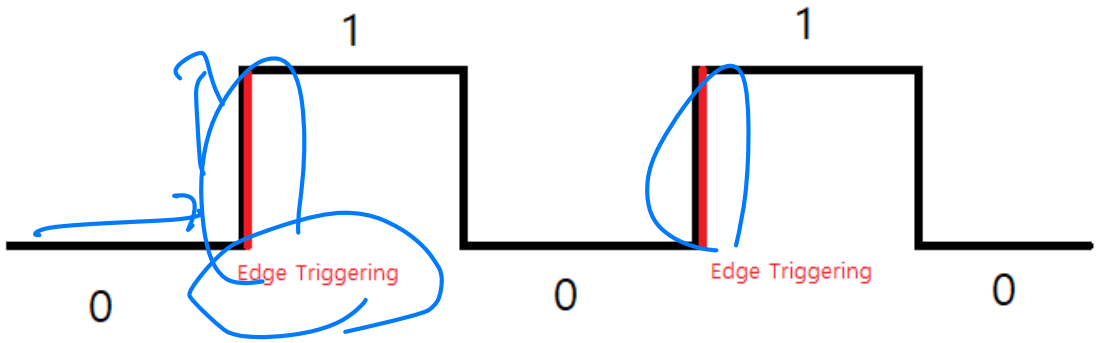
클라이언트 실행 화면

```
$ ./echo_client 127.0.0.1 9190
Connected..........
Input message(Q to quit): Hello
Message from server: Hello
Input message(Q to quit): Hi
Message from server: Hi
Input message(Q to quit): q
```

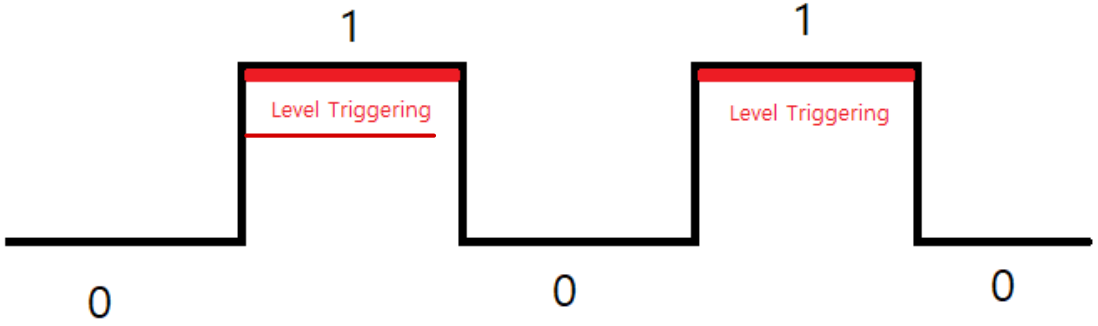
Level Trigger와 Edge Trigger (H/W)



오실로스코프



0->1 또는 1->0로 변하는 시점에 이벤트가 감지



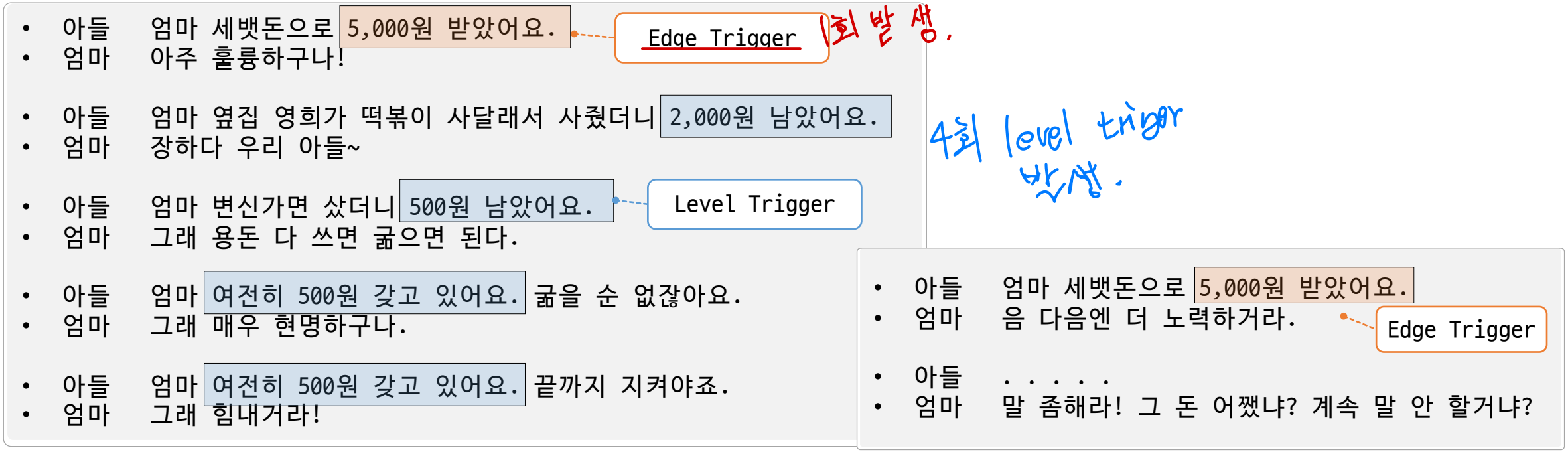
상태가 1 또는 0인 경우에 이벤트가 감지

Level Trigger와 Edge Trigger 차이

- Level trigger
  - 입력 버퍼에 데이터가 남아 있는 동안, 계속 이벤트를 발생 시킴
  - 데이터 양의 변화에 상관없음

- Edge trigger
  - 입력 버퍼에 데이터가 들어오는 순간 딱 한번 이벤트를 발생 시킴

변화 발생



Level Trigger의 이벤트 특성 파악하기

- echo_EPLTserv.c 일부

```
#define BUF_SIZE 4
#define EPOLL_SIZE 50
. . .
while(1)
{
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if(event_cnt == -1)
    {
        puts("epoll_wait() error");
        break;
    }
    puts("return epoll_wait");
    for(i=0; i < event_cnt; i++)
    {
        if(ep_events[i].data.fd==serv_sock)
        {
            . . .
        }
        else
        {
            . . .
        }
    }
}
```

버퍼의 크기를 4바이트로 줄여서 수신된 데이터를 한 번에 읽어 들이지 못하도록 수정함
- 입력 버퍼에 데이터가 남아 있음

- epoll은 기본적으로 Level trigger로 동작
  - 이벤트가 발생해서 epoll_wait() 함수가 반환할 때마다 문자열이 출력됨

```
$ ./echo_EPLTserv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
connected client: 6
return epoll_wait
return epoll_wait
closed client: 6
return epoll_wait
closed client: 5
```

Edge Trigger 기반의 서버 구현을 위해 필요한 것

- Edge Trigger 기반의 서버 구현
  - 1. Non-blocking IO로 소켓 속성 변경
    - fcntl 함수 호출을 통해, 소켓의 기본 설정 정보를 얻은 다음 O_NONBLOCK 속성을 더함

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

    - Edge trigger는 데이터 수신 시 딱 한번만 이벤트가 발생
      - ➢ 이벤트가 발생했을 때, 충분한 양의 버퍼를 마련하고 모든 데이터를 다 읽어 들여야 함
    - 데이터의 분량에 따라 IO로 인한 delay가 발생
    - Edge trigger는 non-blocking IO를 이용
      - ➢ 입력 함수의 호출과 다른 작업을 병행할 수 있음
  - 2. 입력 버퍼의 상태 확인
    - Non-blocking IO 기반으로 데이터 입력 시 데이터 수신이 완료되었는지 별도로 확인해야 됨
    - <error.h>를 포함하고 전역 변수 errno을 참조
      - ➢ errno에 EAGAIN이 저장되면 버퍼가 비어 있는 상태임

```
int errno;
```

← 전역 변수

errno.h 내용

```
#ifndef __ERRNO_H
#define __ERRNO_H
#include <compiler.h>

__BEGIN_CDECLS
extern int *__geterrno(void);

#define errno (*__geterrno())

#define     EPERM 1     /* Not super-user */
#define     ENOENT 2    /* No such file or directory */
#define     ESRCH 3     /* No such process */
#define     EINTR 4     /* Interrupted system call */
#define     EIO 5       /* I/O error */
#define     ENXIO 6     /* No such device or address */
#define     E2BIG 7     /* Arg list too long */
#define     ENOEXEC 8   /* Exec format error */
#define     EBADF 9     /* Bad file number */
#define     ECHILD 10   /* No children */
#define     EAGAIN 11   /* Resource temporarily unavailable */
#define     ENOMEM 12   /* Not enough core */
#define     EACCES 13   /* Permission denied */
...

#define EWOULDBLOCK EAGAIN          /* Operation would block */
#define __ELASTERROR 2000           /* Users can add values starting here */
__END_CDECLS
#endif
```

errno는 errno.h 파일에 전역으로 선언되어 있기 때문에 errno.h를 include하면 해당 변수를 사용 가능함

EAGAIN 정의

/usr/include/asm-generic/errno-base.h

errno 변수

- errno 변수
  - #include <errno.h>
  - 가장 최근 함수 호출의 error 리턴값을 가짐
  - 단순 함수의 리턴값만으로 오류의 원인을 파악하기 어려움
    - 오류 원인을 파악하기 위해 errno 변수를 사용
  - 다른 시스템 함수를 호출하면 errno의 값은 변경됨
    - errno를 사용하기 직전의 함수 호출에만 유효함

- errno 활용 함수
  - char* strerror(int errno)함수: errno에 대한 문자열을 리턴함

```
#include <string.h>
char *strerror(int errno);
```

  - void perror(const char* str): errno를 해석하여 시스템 오류 메시지를 화면에 출력함

```
#include <stdio.h>
void perror(const char *str);
```

errno 활용 예제: perror1.c

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main()
{
    FILE *fp;
    fp = fopen("aaa.txt", "rb");
    if(fp == NULL)
    {
        printf("strerror: %s\n", strerror(errno));
        perror("The following error occurred.");
    }
    else
        fclose(fp);

    return 0;
}
```

전역 변수

```
$ gcc perror1.c -o perror1
$ ./perror1
strerror: No such file or directory
The following error occurred.: No such file or directory
```

엣지 트리거 기반의 echo 서버 #1

- echo_EPETserv.c 일부

```c
#define BUF_SIZE 4
#define EPOLL_SIZE 50
   . . .
if(listen(serv_sock, 5)==-1)
    error_handling("listen() error");

epfd = epoll_create(EPOLL_SIZE);
ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

setnonblockingmode(serv_sock);
event.events = EPOLLIN;
event.data.fd = serv_sock;

epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);

while(1)
{
    event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    . . .
    puts("return epoll_wait");
```

버퍼의 크기를 4바이트로 줄여서 수신된 데이터를 한 번에 읽어 들이지 못하도록 수정함
- 입력 버퍼에 데이터가 남아 있음

```c
void setnonblockingmode(int fd)
{
    int flag=fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flag|O_NONBLOCK);
}
```

- 리스닝 소켓도 비동기 IO를 진행하도록 옵션을 설정하고 있음
- 연결 요청을 수락해서 생성된 소켓에 대해서도 비동기 IO 옵션을 설정해야 됨

엣지 트리거 기반의 echo 서버 #2

```
else
{
    while(1)
    {
        str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
        if(str_len == 0) // close request!
        {
            epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
            close(ep_events[i].data.fd);
            printf("closed client: %d \n", ep_events[i].data.fd);
            break;
        }
        else if(str_len < 0)
        {
            if(errno==EAGAIN)
                break;
        }
        else
        {
            write(ep_events[i].data.fd, buf, str_len); // echo!
        }
    }
}
```

- read() 함수가 -1을 리턴하고, errno가 EAGAIN인 경우
  - 입력 버퍼에 저장된 데이터를 모두 읽어 들인 상황임
  - 따라서 반복문을 빠져 나감

- 수신한 데이터가 있는 경우

Edge Trigger 기반의 echo 서버 실행 결과

서버 실행 결과: 서버의 BUF_SIZE는 1024로 변경함

```
$ ./echo_EPETserv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
```

- 서버는 클라이언트가 종료될 때까지 총 4회의 이벤트 발생

클라이언트 실행 결과

```
$ ./echo_client 127.0.0.1 9190
Connected.....
Input message(Q to quit): Hello Server!
Message from server: Hello Server!

Input message(Q to quit): Nice to meet you!
Message from server: Nice to meet you!

Input message(Q to quit): I like computer programming.
Message from server: I like computer programming.

Input message(Q to quit): q
```

- 클라이언트는 총 4회 메시지 전달

- Edge Trigger 기반에서는 데이터의 송수신 횟수와 이벤트의 발생수가 일치함

echo_EPETserv.c #1

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

#define BUF_SIZE 4
#define EPOLL_SIZE 50
void setnonblockingmode(int fd);
void error_handling(char *buf);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t adr_sz;
    int str_len, i;
    char buf[BUF_SIZE];

    struct epoll_event *ep_events;
    struct epoll_event event;
    int epfd, event_cnt;
```

```c
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");

    if(listen(serv_sock, 5)==-1)
        error_handling("listen() error");

    epfd=epoll_create(EPOLL_SIZE);
    ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

    setnonblockingmode(serv_sock);
    event.events=EPOLLIN;
    event.data.fd=serv_sock;
    epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
```

serv_sock을 non-blocking 모드로 설정

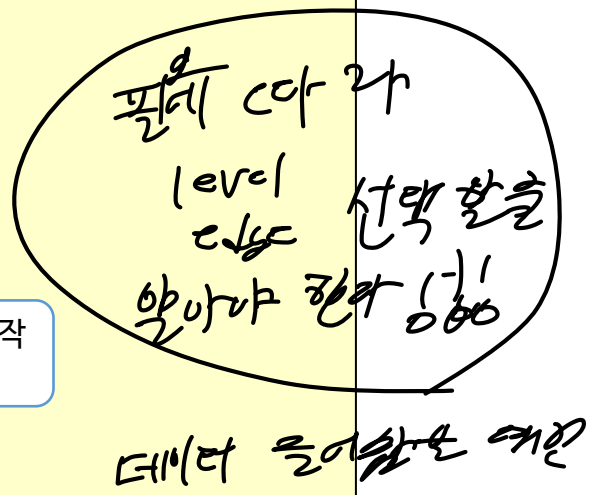
echo_EPETserv.c #2

```
while(1)
{
    event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if(event_cnt==-1)
    {
        puts("epoll_wait() error");
        break;
    }

    puts("return epoll_wait");
    for(i=0; i<event_cnt; i++)
    {
        if(ep_events[i].data.fd==serv_sock)
        {
            adr_sz=sizeof(clnt_adr);
            clnt_sock=accept(serv_sock,
                        (struct sockaddr*)&clnt_adr, &adr_sz);
            setnonblockingmode(clnt_sock); // Non-blocking IO 설정

            event.events=EPOLLIN|EPOLLET;
            event.data.fd=clnt_sock;
            epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
            printf("connected client: %d \n", clnt_sock);
        }
```

이벤트 감지를 위해 edge trigger 방식으로 동작
(EPOLLET 추가)



echo_EPETserv.c #3

```
            else
            {
                while(1)
                {
                    str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
                    if(str_len==0) // close request!
                    {
                        epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
                        close(ep_events[i].data.fd);
                        printf("closed client: %d \n", ep_events[i].data.fd);
                        break;
                    }
                    else if(str_len<0)
                    {
                        if(errno==EAGAIN)
                            break;
                    }
                    else
                    {
                        write(ep_events[i].data.fd, buf, str_len); // echo!
                    }
                }
            }
        }
    }
    close(serv_sock);
    close(epfd);
    return 0;
}
```

- EAGAIN인 경우
  - 버퍼에 데이터가 없는 경우이므로, 반복문을 빠져 나감

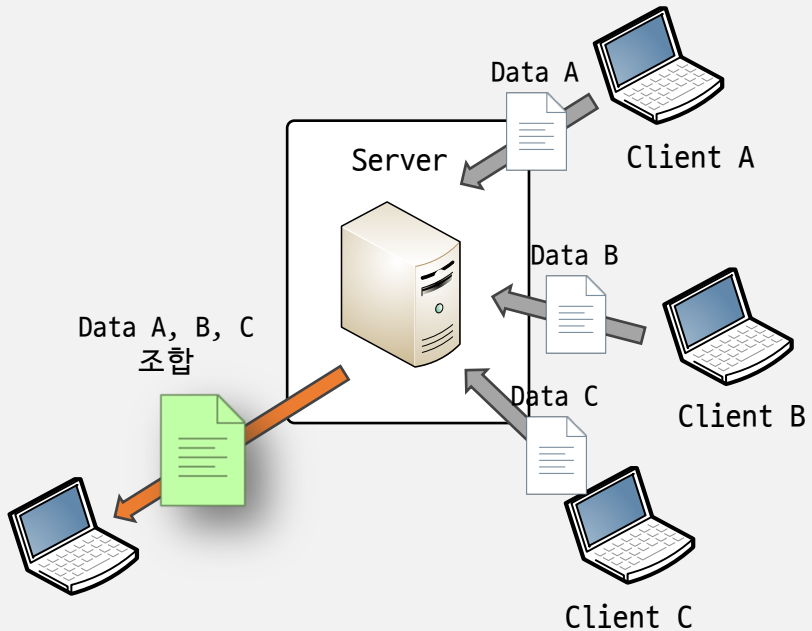
echo_EPETserv.c #4

```c
void setnonblockingmode(int fd)
{
    int flag=fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flag|O_NONBLOCK);
}

void error_handling(char *buf)
{
    fputs(buf, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

Edge Trigger와 Level Trigger 비교

- 서버 통신 시나리오
  - 서버는 클라이언트 A, B, C로부터 각각 데이터를 수신
  - 서버는 수신한 데이터를 A, B, C의 순으로 조합
  - 조합한 데이터는 임의의 호스트에게 전달



- 클라이언트가 서버에 접속 및 데이터를 전송하는 순서는 서버의 기대와 상관이 없음
- 서버에서 제어할 요소가 많은 경우에는 edge trigger가 유리함
- 서버의 역할이 상대적으로 단순하고 데이터 송수신 상황이 다양하지 않으면 level trigger 방식을 선택할 만함

Questions?