I/O Multiplexing-epoll()

→ Level trigger
→ Edge trigger

유닉스 CH17.

Instructor: Limei Peng
Dept. of CSE, KNU

Outlines

- **Level** trigger
- **Edge** trigger

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

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

➜ # of file descriptors generated by events on success; -1 on error

- epfd — File descriptors of epoll instances in the observation space
- events — Address of buffer that will be filled with file descriptors generated by events
- maxevent — Maximum # of events that can be saved in the buffer of the 2nd argument
- timeout — Waiting time with interval of 1/1000 sec; -1 on transmission; wait forever until any event occurs

레벨 트리거와 엣지 트리거의 차이

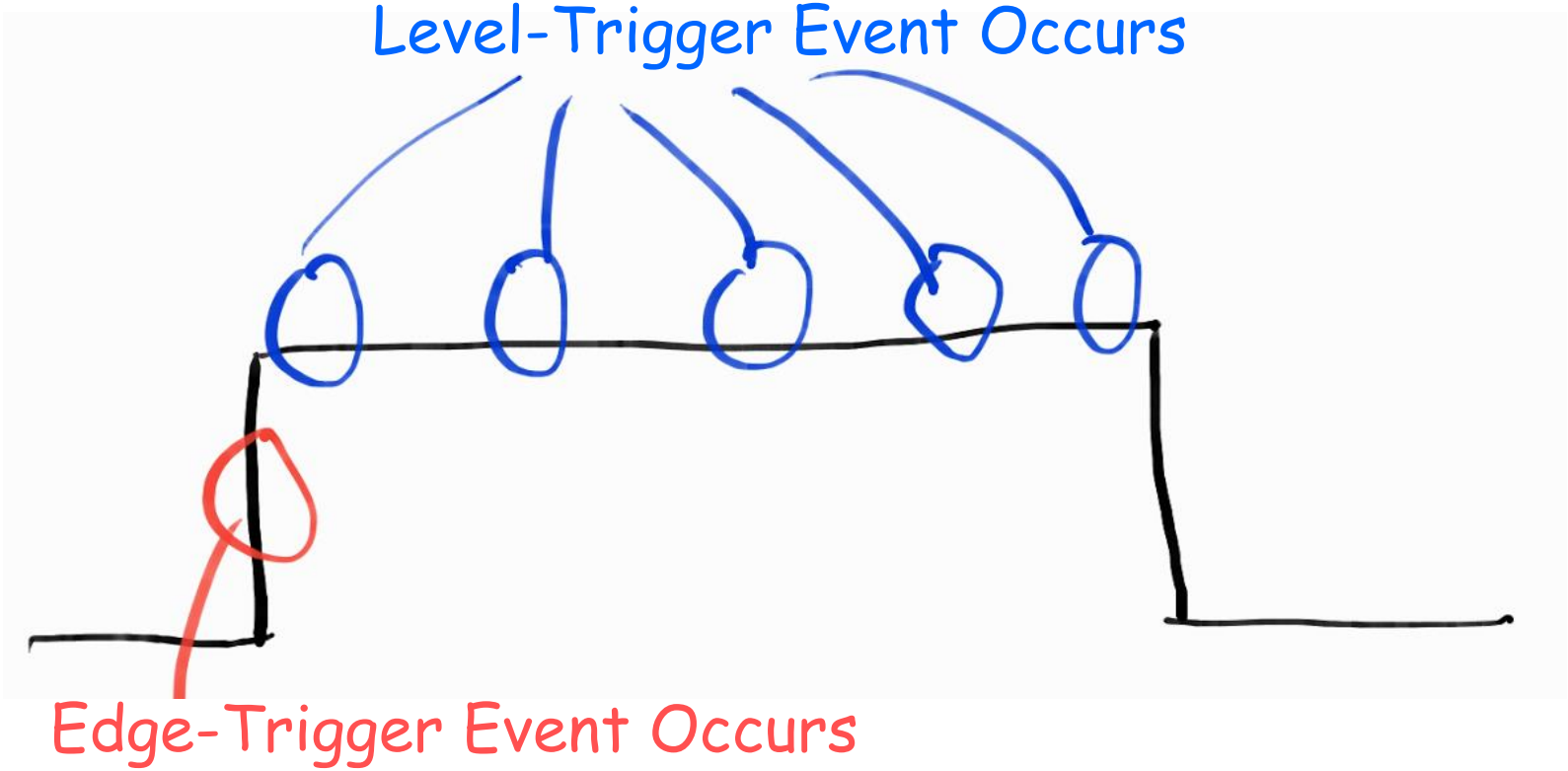
- 아들 엄마 세뱃돈으로 5,000원 받았어요.
- 엄마 아주 훌륭하구나!
- 아들 엄마 옆집 숙희가 떡볶이 사달래서 사줬더니 2,000원 남았어요.
- 엄마 장하다 우리아들~
- 아들 엄마 변신가면 샀더니 500원 남았어요.
- 엄마 그래 용돈 다 쓰면 굶으면 된다!
- 아들 엄마 여전히 500원 갖고 있어요. 굶을 순 없잖아요.
- 엄마 그래 매우 현명하구나!
- 아들 엄마 여전히 500원 갖고 있어요. 끝까지 지켜야지요.
- 엄마 그래 힘내거라!

레벨 트리거 방식은 입력 버퍼에 데이터가 남아있는 동안에 계속해서 이벤트를 발생시킨다. 데이터 양의 변화에 상관없이!

레벨 트리거와 엣지 트리거의 차이는 이벤트의 발생 방식에 있다.

엣지 트리거 방식은 입력 버퍼에 데이터가 들어오는 순간 딱 한번만 이벤트를 발생시킨다.

- 아들 엄마 세뱃돈으로 5,000원 받았어요.
- 엄마 음 다음엔 더 노력하거라.
- 아들 . . . . . .
- 엄마 말 좀 해라! 그 돈 어쨌냐? 계속 말 안 할거냐?


Level-Trigger Event Occurs
Edge-Trigger Event Occurs

레벨 트리거의 이벤트 특성 파악하기

예제 echo_EPLTserv.c의 일부

```
#define BUF_SIZE 4
#define EPOLL_SIZE 50
```

버퍼의 크기를 4바이트로 줄여서 수신된 메시지를 한번에 읽어 들이지 못하도록 하였다.

소켓은 기본적으로 레벨 트리거로 동작한다.

```
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)
        {
```

이벤트가 발생해서 epoll_wait 함수가 반환할 때마다 문자열이 출력되도록 하였다.

```
root@my_linux:/tcpip# gcc echo_EPLTserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
connected client: 6
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
return epoll_wait
closed client: 6
```

실행결과는 버퍼에 입력데이터가 남아있는 상황에서 이벤트가 발생함을 보이고 있다.

Example#5: echo_EPLTserv.c (1/3)

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

#define BUF_SIZE 4
#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);
        }

        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]));
```

Example#5: echo_EPLTserv.c(2/3)

```
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);

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);
                        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
```

Example#5: echo_EPLTserv.c(3/3)

```
                        {
                                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);
}
```

Example#5: echo_client.c(1/2)

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

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
        int sock;
        char message[BUF_SIZE];
        int str_len;
        struct sockaddr_in serv_adr;

        if(argc!=3) {
                printf("Usage : %s <IP> <port>\n", argv[0]);
                exit(1);
        }

        sock=socket(PF_INET, SOCK_STREAM, 0);
        if(sock==-1)
                error_handling("socket() error");

        memset(&serv_adr, 0, sizeof(serv_adr));
        serv_adr.sin_family=AF_INET;
        serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
        serv_adr.sin_port=htons(atoi(argv[2]));

        if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
                error_handling("connect() error!");
        else
                puts("Connected.....");
```

Example#5: echo_client.c(2/2)

```
        while(1)
        {
                fputs("Input message(Q to quit): ", stdout);
                fgets(message, BUF_SIZE, stdin);

                if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
                        break;

                write(sock, message, strlen(message));
                str_len=read(sock, message, BUF_SIZE-1);
                message[str_len]=0;
                printf("Message from server: %s", message);
        }

        close(sock);
        return 0;
}

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

Execution Results #1

```
zq@ubuntu:~/Desktop/Chapter17$ gcc echo_EPLTserv.c -o serv
zq@ubuntu:~/Desktop/Chapter17$ ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
connected client: 6
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 6
return epoll_wait
closed client: 5
```

Server

```
zq@ubuntu:~/Desktop/Chapter17$ gcc echo_client.c -o client
zq@ubuntu:~/Desktop/Chapter17$ ./client 127.0.0.1 9190
Connected.....
Input message(Q to quit): It's my life
Message from server: It's my life
Input message(Q to quit): Q
```

Client1

```
zq@ubuntu:~/Desktop/Chapter17$ gcc sep_clnt.c -o clnt
zq@ubuntu:~/Desktop/Chapter17$ ./client 127.0.0.1 9190
Connected.....
Input message(Q to quit): It's your life
Message from server: It's your life
Input message(Q to quit): Q
```

Client2

Execution Results #2

```
xiao@xiao-virtual-machine:~/Desktop$ gcc echo_EPLTserv.c -
o serv
xiao@xiao-virtual-machine:~/Desktop$ ./serv 9191
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
```

Server

```
xiao@xiao-virtual-machine:~/Desktop$ gcc echo_client.c -o
client
xiao@xiao-virtual-machine:~/Desktop$ ./client 127.0.0.1 91
91
Connected..........
Input message(Q to quit): I LIKE PROGRAMMING
Message from server: I LIKE PROGRAMMING
Input message(Q to quit): do you like programming
Message from server: do you like programming
Input message(Q to quit): good bye
Message from server: good bye
Input message(Q to quit): q
xiao@xiao-virtual-machine:~/Desktop$
```

Client

엣지 트리거 기반의 서버 구현을 위해 필요한 것

1. 넌-블로킹 IO로 소켓 속성 변경

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

fcntl 함수호출을 통해서 소켓의 기본 설정정보를 얻은 다음, 거기에 O_NONBLOCK 속성을 더해서 소켓의 특성을 재설정한다.

엣지 트리거는 데이터 수신 시 딱 한번만 이벤트가 발생하기 때문에 이벤트가 발생했을 때 충분한 양의 버퍼를 마련한 다음에 모든 데이터를 다 읽어 들여야 한다. 즉, 데이터의 분량에 따라서 IO로 인한 DELAY가 생길 수 있다. 그래서 엣지 트리거에서는 넌-블로킹 IO를 이용한다. 입력 함수의 호출과 다른 작업을 병행할 수 있기 때문이다.

2. 입력버퍼의 상태확인

```
int errno;
```

넌-블로킹 IO 기반으로 데이터 입력 시 데이터 수신이 완료되었는지 별도로 확인해야 한다. 헤더파일 <error.h>를 포함하고 변수 errno을 참조한다. errno에 EAGAIN이 저장되면 버퍼가 빈 상태이다.

엣지 트리거 기반의 echo 서버1

예제 echo_EPETserv.c의 일부

```
#define BUF_SIZE 4
#define EPOLL_SIZE 50
```

여전히 버퍼의 크기를 4바이트로 줄여서 수신된 메시지를 한번에 읽어 들이지 못하도록 하였다.

```
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);
```

리스닝 소켓도 비동기 IO를 진행하도록 옵션을 설정하고 있다.

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

accept() 함수 호출로 생성된 socket은 nonblocking IO로 설정

연결요청을 수락해서 생성된 소켓에 대해서도 비동기 IO의 옵션을 설정해야 한다.

엣지 트리거 기반의 echo 서버2

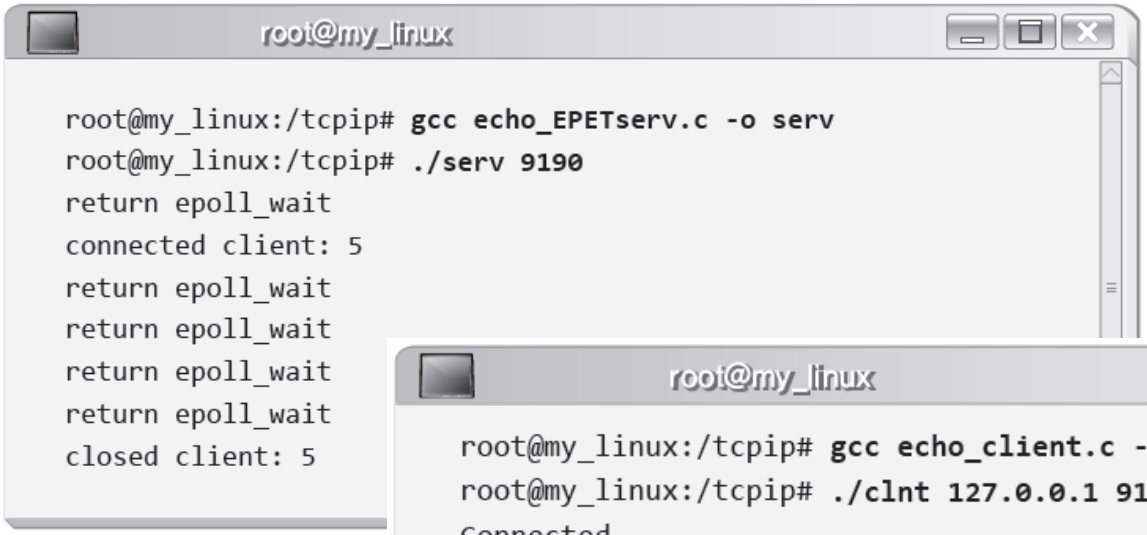
예제 echo_EPETserv.c의 일부(else... 수신할 데이터가 있는 경우)

```
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!
        }
    }
}
```

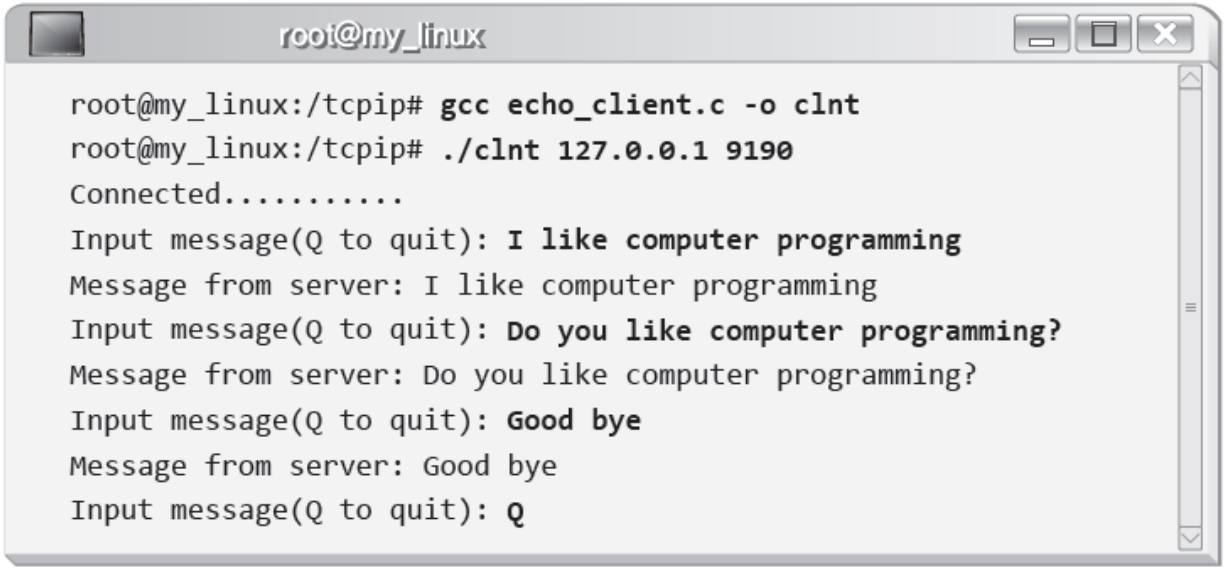
⊜

errno가 EAGAIN일 때까지, 즉 버퍼가 완전히 비워질 때까지 반복문을 계속 돌며 데이터를 수신하고 있다.

엣지 트리거 기반의 echo 서버의 실행결과



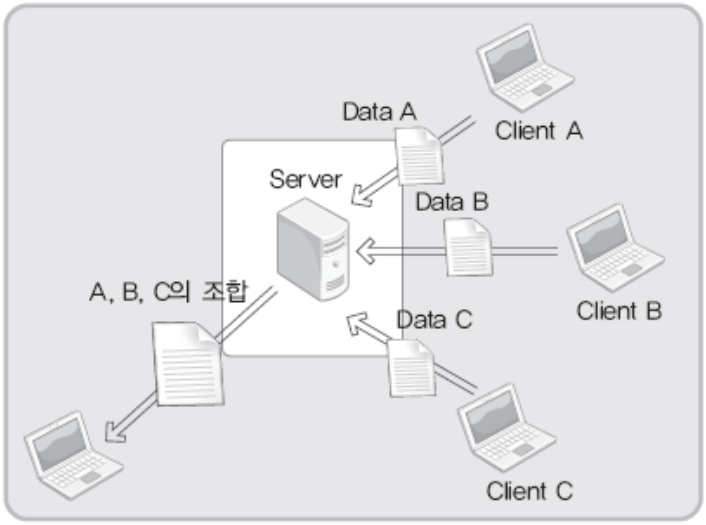
왼쪽은 서버의 실행결과이다. 클라이언트가 종료될 때까지 총 4회의 이벤트가 발생했음을 알 수 있다.



오른쪽은 클라이언트의 실행결과이다. 총 4회의 메시지를 전달했음을 알 수 있다.

이렇듯, 엣지 트리거 기반에서는 데이터의 송수신횟수와 이벤트의 발생수가 일치한다.

엣지 트리거와 레벨 트리거의 비교



- 서버는 클라이언트 A, B, C로부터 각각 데이터를 수신한다.
- 서버는 수신한 데이터를 A, B, C의 순으로 조합한다.
- 조합한 데이터는 임의의 호스트에게 전달한다.

위와 같은 시나리오 상에서 클라이언트가 서버에 접속 및 데이터를 전송하는 순서는 서버의 기대와 상관이 없다. 이처럼 서버측에서의 컨트롤 요소가 많은 경우에는 엣지 트리거가 유리하다. 반면, 서버의 역할이 상대적으로 단순하고 또 데이터 송수신의 상황이 다양하지 않다면, 레벨 트리거 방식을 선택할만하다.

Example#6:echo_EPETserv.c(1/3)

```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;

        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);
```

Example#6:echo_EPETserv.c(2/3)

```
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);
                        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);
                }
                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;
```

Just register the IO event for one time. When receiving data from client, just print "return epoll_wait" for one time

Example#6:echo_EPETserv.c(3/3)

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

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);
}
```

```
                }
                else
                {
                        write(ep_events[i].data.fd, buf, str_len);    // echo!
                }
```

Execution Results

- Echo_EPETserv.c

```
zq@ubuntu:~/Desktop/Chapter17$ gcc echo_EPETserv.c -o serv
zq@ubuntu:~/Desktop/Chapter17$ ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
```

- echo_client.c

```
zq@ubuntu:~/Desktop/Chapter17$ gcc echo_client.c -o clnt
zq@ubuntu:~/Desktop/Chapter17$ ./clnt 127.0.0.1 9190
Connected.....
Input message(Q to quit): i like computer programming
Message from server: i like computer programming
Input message(Q to quit): do you like programming?
Message from server: do you like programming?
Input message(Q to quit): good bye
Message from server: good bye
Input message(Q to quit): q
```