

# Ch17-I/O Multiplexing

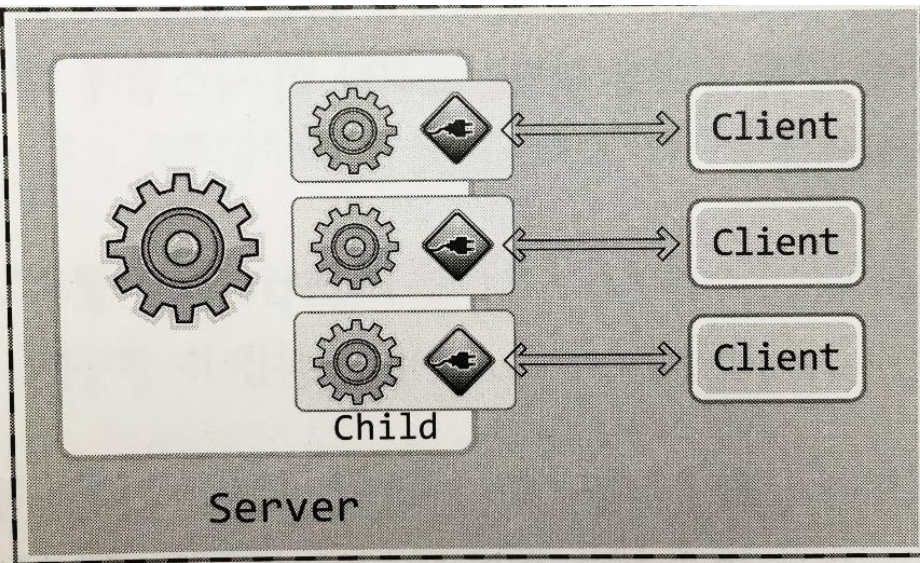
## -epoll() 윤영우 CH17.

Instructor: Limei Peng  
Dept. of CSE, KNU

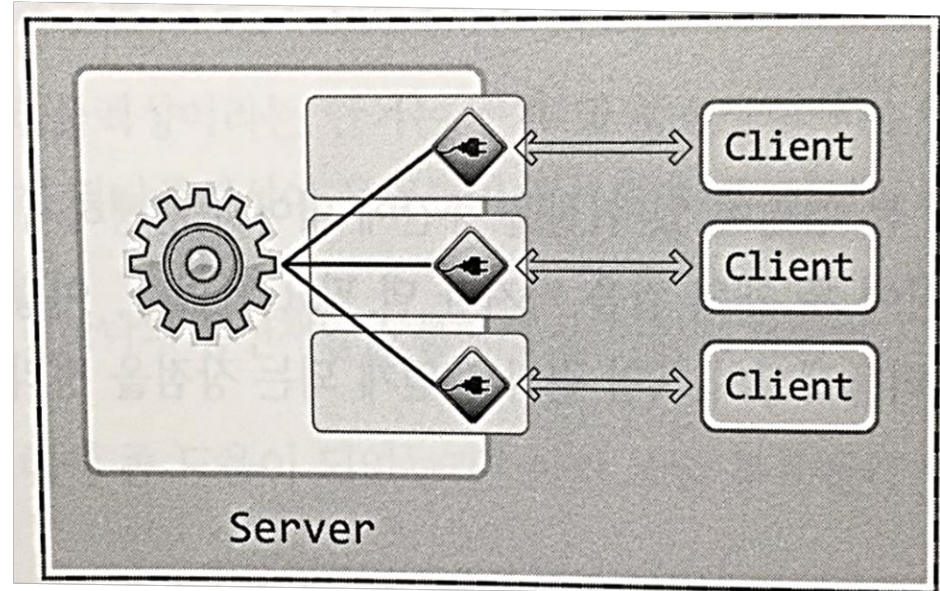
# Outlines

❖ Review on **select()**

❖ **epoll()**



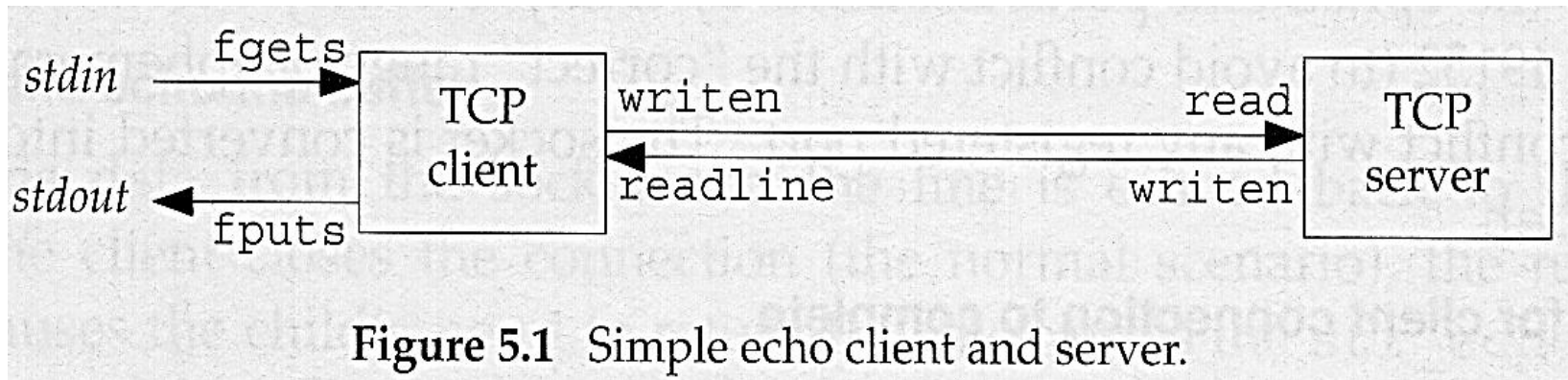
Multi-process based model



I/O Multiplexing based model

# Review on `select()`(1/3)

- ❖ Client could be blocked in `fgets` and miss data from `readline`
- ❖ Sending and receiving data should be independent



# Review on **select()**(2/3)

## ❖ **select** function prototype

- Instruct the kernel to wait for one of the multiple events to occur : multiple event 중 하나가 발생할 때까지 커널이 대기
- Wake up the process only when one or more of these events occurs or when a specified amount of time has passed

```
int select(int maxfdp1, fd_set *readset,  
           fd_set *writeset, fd_set *exceptset,  
           const struct timeval *timeout)
```

- Returns: positive count of ready descriptors, 0 on timeout, -1 on error

# Review on `select()` (3/3)

## ❖ Macros for `fd_set` datatype

- `void FD_ZERO(fd_set *fdset);`  
//clear all bits
- `void FD_SET(int fd, fd_set *fdset);`  
//turn on the bit for *fd*
- `void FD_CLR(int fd, fd_set *fdset);`  
// turn off the bit for *fd*
- `int FD_ISSET(int fd, fd_set *fdset);`  
// is the bit for *fd* on?

```
int main(void)
{
```

```
    fd_set set;
```

	fd0	fd1	fd2	fd3	
FD_ZERO(&set);	0	0	0	0	.....

	fd0	fd1	fd2	fd3	
FD_SET(1, &set);	0	1	0	0	.....

	fd0	fd1	fd2	fd3	
FD_SET(2, &set);	0	1	1	0	.....

	fd0	fd1	fd2	fd3	
FD_CLR(2, &set);	0	1	0	0	.....

```
}
```

# epoll()

- ❖ **epoll()** is the latest, greatest, newest polling method in Linux (and **only Linux**), added to kernel in 2002
- ❖ **select()** is supported by most of the OSs
- ❖ **epoll()** differs from **select()** in that
  - it **keeps** information/about the currently monitored descriptors and associated events **inside the kernel** and can add/remove/modify them

: 커널 내부에서 현재 모니터링되는 descriptor와 관련 이벤트들에 대한 정보를 유지하고,  
정보를 추가/삭제/변경할 수 있음.



# epoll()

- ❖ To use **epoll()**, much more preparation is needed
  - Create a storage space to store file descriptors by calling **epoll\_create()** : fd를 저장하기 위해 저장공간을 만든다.
  - Initialize the **epoll\_event** structure with the wanted events and the context data pointer : epoll\_event 구조체를 초기화한다.
  - Call **epoll\_ctl()** to register/delete descriptors in the monitoring set (i.e., storage space) : monitoring set에 fd를 등록/삭제
  - Call **epoll\_wait()** to react to the changes in events/file descriptors in the storage space : 저장공간에 event/fd의 변화에 반응함.
  - Iterate through the returned items

- **epoll\_create**      epoll 파일 디스크립터 저장소 생성
- **epoll\_ctl**      저장소에 파일 디스크립터 등록 및 삭제
- **epoll\_wait**      select 함수와 마찬가지로 파일 디스크립터의 변화를 대기한다

# epoll의 구현에 필요한 함수와 구조체

- ㉠• `epoll_create`    epoll 파일 디스크립터 저장소 생성
- ㉡• `epoll_ctl`        저장소에 파일 디스크립터 등록 및 삭제
- ㉢• `epoll_wait`       select 함수와 마찬가지로 파일 디스크립터의 변화를 대기한다.

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

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

위의 세 함수 호출을 통해서 epoll의 기능이 완성된다.

위의 세 함수와 왼쪽의 구조체가 어떻게 사용되는지 이해하면, epoll을 이해하는 셈이 된다.

왼쪽의 구조체는 소켓 디스크립터의 등록 및 이벤트 발생의 확인에 사용되는 구조체이다.



㉠

## epoll\_create



: epoll 특許 socket 파일디스크립터를 저장할 함수공간 생성.

→ socket 파일디스크립터를 저장하는데 사용.

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

```
int epoll_create(int size);
```

→ 성공 시 epoll 파일 디스크립터, 실패 시 -1 반환

● size      epoll 인스턴스의 크기정보.

운영체제가 관리하는, **epoll 인스턴스**라 불리는 **파일 디스크립터의 저장소**를 생성!

소멸 시 close 함수호출을 통한 종료의 과정이 필요하다.

위의 함수호출을 통해서 생성된 **epoll 인스턴스**에 관찰대상을 저장 및 삭제하는 함수가 **epoll\_ctl**이고, **epoll 인스턴스**에 등록된 파일 디스크립터를 대상으로 이벤트의 발생 유무를 확인하는 함수가 **epoll\_wait**이다.

④

# epoll\_ctl



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

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

→ 성공 시 0, 실패 시 -1 반환

두 번째 전달인자에 따라 등록, 삭제 및 변경이 이뤄진다.

두 번째 전달인자

- epfd      관찰대상을 등록할 epoll 인스턴스의 파일 디스크립터.
- op      관찰대상의 추가, 삭제 또는 변경여부 지정.
- fd      등록할 관찰대상의 파일 디스크립터.
- event      관찰대상의 관찰 이벤트 유형.

- EPOLL\_CTL\_ADD      파일 디스크립터를 epoll 인스턴스에 등록한다.
- EPOLL\_CTL\_DEL      파일 디스크립터를 epoll 인스턴스에서 삭제한다.
- EPOLL\_CTL\_MOD      등록된 파일 디스크립터의 이벤트 발생상황을 변경한다.

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

epoll 인스턴스 A에, 파일 디스크립터 B를 등록하되, C를 통해 전달된 이벤트의 관찰을 목적으로 등록을 진행한다.”

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

epoll 인스턴스 A에서 파일 디스크립터 B를 삭제한다.


# epoll\_ctl 함수 기반의 디스크립터 등록

epoll\_event 구조체는 이벤트의 유형

등록에 사용된다.

그리고 이벤트 발생시 발생한 이벤트의 정보로도 사용된다.

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



- **EPOLLIN** 수신할 데이터가 존재하는 상황
- **EPOLLOUT** 출력버퍼가 비워져서 당장 데이터를 전송할 수 있는 상황
- **EPOLLPRI** OOB 데이터가 수신된 상황
- **EPOLLRDHUP** 연결이 종료되거나 Half-close가 진행된 상황, 이는 엡지 트리거 방식에서 유용하게 사용될 수 있다.
- **EPOLLERR** 에러가 발생한 상황
- **EPOLLET** 이벤트의 감지를 엡지 트리거 방식으로 동작시킨다.
- **EPOLLONESHOT** 이벤트가 한번 감지되면, 해당 파일 디스크립터에서는 더 이상 이벤트를 발생시키지 않는다. 따라서 epoll\_ctl 함수의 두 번째 인자로 EPOLL\_CTL\_MOD을 전달해서 이벤트를 재설정해야 한다.

비트 OR 연산을 통해 둘 이상을 함께 등록할 수 있다.

㉔

# epoll\_wait



```
#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

- epfd 이벤트 발생의 관찰영역인 epoll 인스턴스의 파일 디스크립터.
- events 이벤트가 발생한 파일 디스크립터가 채워질 버퍼의 주소 값.
- maxevents 두 번째 인자로 전달된 주소 값의 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 epoll_event)*EPOLL_SIZE);
. . . . .
event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
. . . . .
```

epoll\_wait 함수 반환 후, 이벤트 발생한 파일 디스크립터의 수가 반환되고, 두 번째 인자로 전달된 주소의 메모리 공간에 이벤트 발생한 파일 디스크립터 별도로 묶인다.

epoll\_wait 함수의 두 번째 인자를 통해서 이벤트 발생한 디스크립터가 별도로 묶이므로,

전체 파일 디스크립터 대상의 반복문은 불필요 하다.

# Example #1-review on **select()**(1/2)

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

#define BUFSIZE 100
void errorHandling(char* buf)
{
    fputs(buf,stderr);
    fputc('\n',stderr);
    exit(1);
}

int main(int argc, char* argv[])
{
    int servSock, clntSock;
    struct sockaddr_in servAddr, clntAddr;
    struct timeval timeout;
    fd_set reads, cpyReads;

    socklen_t addrSz;
    int fdMax, strLen, fdNum, i;
    char buf[BUFSIZE];

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

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

    if( bind(servSock, (struct sockaddr*)&servAddr, sizeof(servAddr)) == -1 )
        errorHandling("bind() error");

    if(listen(servSock, 5) == -1 )
        errorHandling("listen() error");

    FD_ZERO(&reads);
    FD_SET(servSock, &reads);
    fdMax = servSock;
```

→Execute together  
with "echo\_client.c"

# Example #1-review on select()(2/2)

```
while(1)
{
    cpyReads = reads;
    timeout.tv_sec = 5;
    timeout.tv_usec = 5000;

    if((fdNum = select(fdMax+1, &cpyReads, 0, 0, &timeout)) == -1 )
        break;

    if(fdNum == 0 )
        continue;

    for(i = 0 ; i < fdMax+1 ; i++ )
    {
        if(FD_ISSET(i, &cpyReads) )
        {
            if(i == servSock )
            {
                addrSz = sizeof(clntAddr);
                clntSock = accept(servSock, (struct sockaddr*)&clntAddr, &addrSz);
                FD_SET(clntSock, &reads);

                if(fdMax < clntSock )
                    fdMax = clntSock;

                printf("connected client : %d\n", clntSock)
            }

            else
            {
                strLen = read(i, buf, BUFSIZE);

                if(strLen == 0 )
                {
                    FD_CLR(i, &reads);
                    close(i); // close request
                    printf("close client : %d\n", i);
                }

                else
                    write(i, buf, strLen); // echo
            }
        }
    }

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

```
socket@ubuntu:~/Desktop/code$ vi echo_selectserv.c
socket@ubuntu:~/Desktop/code$ gcc echo_selectserv.c -o eselects
socket@ubuntu:~/Desktop/code$ ./eselects 9190
connected client : 4
connected client : 5
close client : 4
close client : 5
```

Server

```
socket@ubuntu:~/Desktop/code$ gcc echo_client.c -o echoc
socket@ubuntu:~/Desktop/code$ ./echoc 127.0.0.1 9190
connected.....
Input message(Q to quit): Hi, I am the 1st client
Message from server:Hi, I am the 1st client
Input message(Q to quit): Goodbye
Message from server:Goodbye
Input message(Q to quit): ^C
```

Client1

```
socket@ubuntu:~/Desktop/code$ ./echoc 127.0.0.1 9190
connected.....
Input message(Q to quit): Hi, I am the 2nd client
Message from server:Hi, I am the 2nd client
Input message(Q to quit): Goodbye~
Message from server:Goodbye~
Input message(Q to quit): ^C
```

Client2



# Example #2 - **epoll()** (1/2)

```
/*echo_epollserv.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)
{
    fputs(buf,stderr);
    fputc('\n',stderr);
    exit(1);
}

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");
```

→Execute together  
with "echo\_client.c"

# Example #2- epoll() (2/2)

```
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++)
```

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

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

```
socket@ubuntu:~/Desktop/code$ vi echo_epollserv.c  
socket@ubuntu:~/Desktop/code$ gcc echo_epollserv.c -o epolls  
socket@ubuntu:~/Desktop/code$ ./epolls 9190  
connected client: 5  
connected client: 6  
closed client: 5  
closed client: 6
```

Server

```
socket@ubuntu:~/Desktop/code$ ./echoc 127.0.0.1 9190  
connected.....  
Input message(Q to quit): 1st epoll client  
Message from server:1st epoll client  
Input message(Q to quit): Bye~  
Message from server:Bye~  
Input message(Q to quit): ^C
```

Client1

```
socket@ubuntu:~/Desktop/code$ ./echoc 127.0.0.1 9190  
connected.....  
Input message(Q to quit): 2nd epoll client  
Message from server:2nd epoll client  
Input message(Q to quit): Byebye~  
Message from server:Byebye~  
Input message(Q to quit): ^C
```

Client2