Chapter 12

IO 멀티플렉싱 기반의 서버

멀티 프로세스 서버의 단점과 대안

- 멀티 프로세스 서버의 단점
  - 프로세스의 빈번한 생성은 성능 저하로 이어짐
  - 멀티 프로세스의 흐름을 고려한 구현이 쉽지 않음
  - 프로세스간 통신이 필요한 상황에서 서버의 구현이 복잡해짐

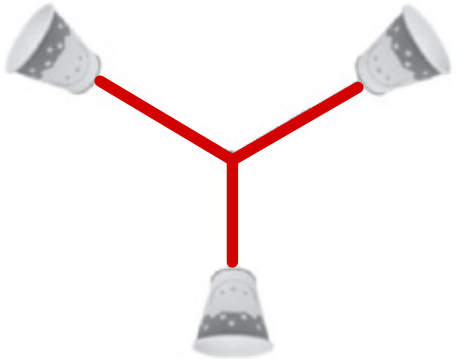
- 멀티 프로세스 서버의 대안
  - 하나의 프로세스가 다수의 클라이언트에게 서비스 제공
    - 하나의 프로세스가 여러 개의 소켓을 핸들링 할 수 있는 방법이 존재해야 됨
    - IO 멀티플렉싱(multiplexing)

- 멀티플렉싱 (Multiplexing)
  - 하나의 통신 채널을 통해서 둘 이상의 데이터를 전송하는 기술
    - TDM (Time-division Multiplexing)
    - FDM (Frequency-division Multiplexing)

단점
round robin 방식으로
client 처리
? 단점?

멀티플렉싱

멀티플렉싱

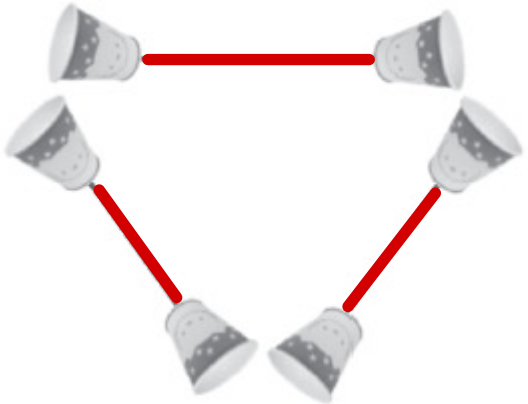


- 멀티플렉싱
  - 왼쪽 그림처럼 하나의 리소스를 둘 이상의 영역에서 공유

- 전화의 경우 말을 하는 순서가 일치하지 않고, 목소리 주파수도 다르기 때문에 멀티플렉싱이 가능
- 데이터의 송수신도 실시간으로 지연 시간이 없이 서비스를 해야 되는 것이 아니기 때문에 멀티플렉싱이 가능함
- 하나의 프로세스가 다수의 소켓을 관리

real time 처리에도 좋지않겠다.

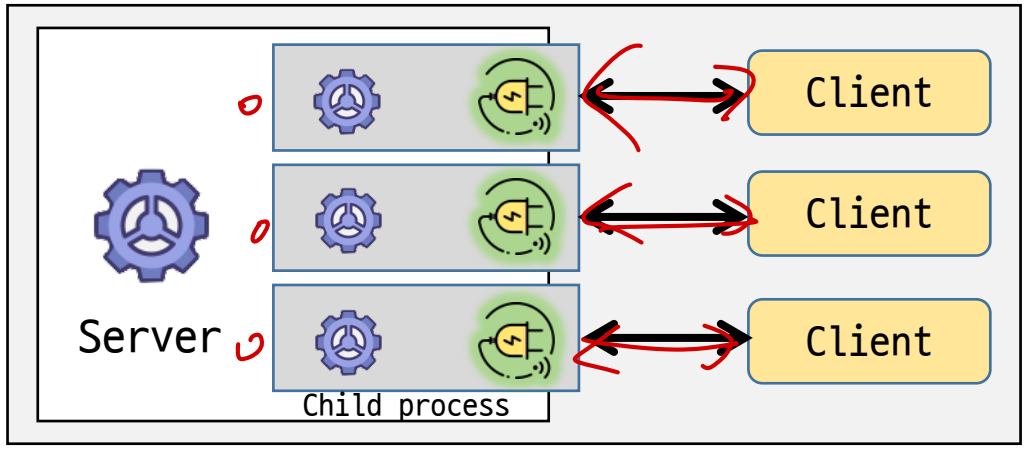
멀티 프로세스



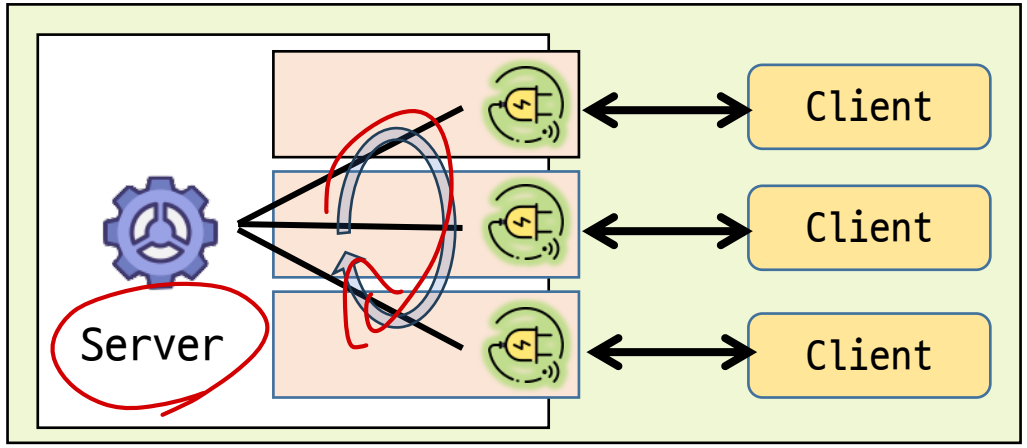
- 클라이언트가 접속하면 새로운 프로세스를 생성해서 통신
- 접속된 클라이언트의 수만큼 프로세스 생성

멀티플렉싱 서버 모델

멀티프로세스 서버 모델



멀티플렉싱 서버 모델
(select 함수 사용)



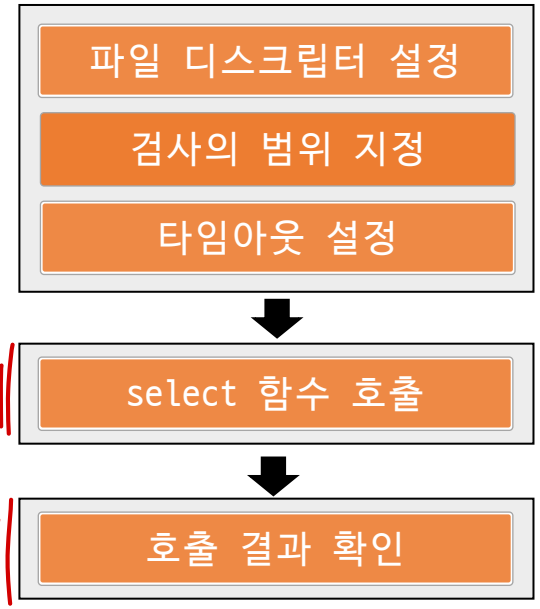
클라이언트의 수와 상관없이 하나의 서버 프로세스가 서비스를 제공

select 함수의 기능과 호출순서

- select() 함수
  - 여러 개의 파일 디스크립터를 모아 놓고 관찰할 수 있음
    - 배열에 저장된 다수의 파일 디스크립터를 대상으로 아래의 내용을 확인

3가지

- 수신한 데이터를 지니고 있는 소켓이 존재하는가?
- 블로킹되지 않고 데이터 전송이 가능한 소켓은 무엇인가?
- 예외 상황이 발생한 소켓은 무엇인가?

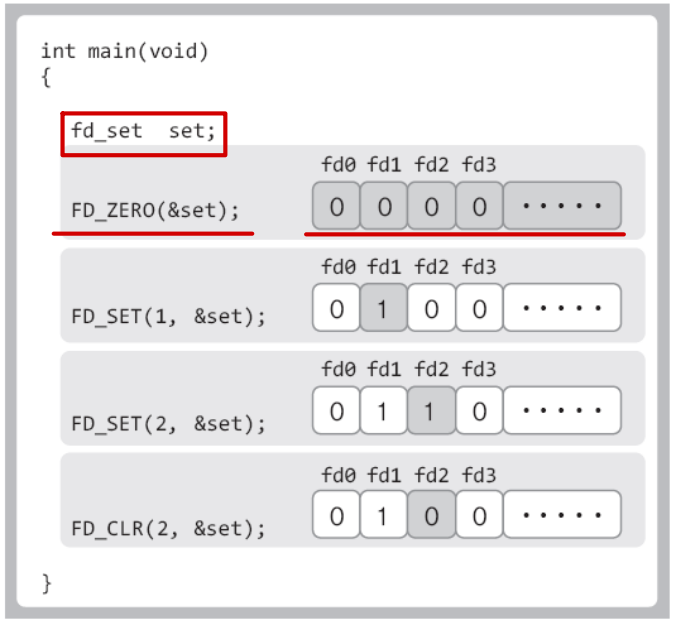


Step 01 관찰 대상을 묶고, 관찰 유형을 지정

Step 02 관찰 대상의 변화를 묻는다.

Step 03 물음에 대한 답을 듣는다.

파일 디스크립터의 설정



- select() 함수에 전달할 디스크립터의 정보를 fd_set 변수로 묶음
  - ➢ fd_set: 파일 디스크립터를 모아 놓은 배열
- 모두 0으로 초기화: FD_ZERO(&set)
- 디스크립터 1을 관찰 대상으로 추가: FD_SET(1, &set)
- 디스크립터 2를 관찰 대상으로 추가: FD_SET(2, &set)
- 디스크립터 2를 관찰 대상에서 제외: FD_CLR(2, &set)

파일 디스크립터 (소켓)

▪ fd_set 컨트롤 함수

- FD_ZERO(fd_set *fdset): 매개변수 fd_set 타입 변수의 모든 비트를 0으로 초기화
- FD_SET(int fd, fd_set *fdset): 매개변수 fdset에 파일 디스크립터(fd) 정보를 등록
- FD_CLR(int fd, fd_set *fdset): 매개변수 fdset에서 파일 디스크립터(fd) 정보를 삭제
- FD_ISSET(int fd, fd_set *fdset): 매개변수 fdset에서 파일 디스크립터(fd)의 정보가 있으면 양수를 반환

변화

fd_set의 크기: select.h

소스인사이트

소스분석 툴

이런거 분석하기 개 빡세...

궁금하면 찾아봐.

- fd_set의 크기: 운영체제에 따라 크기가 다름
  - /usr/include/x86_64-linux-gnu/sys/select.h 에 정의

```
/* The fd_set member is required to be an array of longs.  */
typedef long int __fd_mask;

/* Some versions of <linux/posix_types.h> define this macros.  */
#undef  __NFDBITS
/* It's easier to assume 8-bit bytes than to get CHAR_BIT.  */
#define __NFDBITS   (8 * (int) sizeof (__fd_mask))
#define __FD_ELT(d) ((d) / __NFDBITS)
#define __FD_MASK(d)    ((__fd_mask) (1UL << ((d) % __NFDBITS)))

/* fd_set for select and pselect.  */
typedef struct
  {
    /* XPG4.2 requires this member name.  Otherwise avoid the name
       from the global namespace.  */
#ifdef __USE_XOPEN
    __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->fds_bits)
#else
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->__fds_bits)
#endif
  } fd_set;

/* Maximum number of file descriptors in `fd_set'.  */
#define FD_SETSIZE      __FD_SETSIZE
```

디버깅 안에

__FD_SETSIZE 1024
- /usr/include/linux/posix_types.h 에 정의

```
 7 /*
 8  * This allows for 1024 file descriptors: if NR_OPEN is ever grown
 9  * beyond that you'll have to change this too. But 1024 fd's seem to be
10  * enough even for such "real" unices like OSF/1, so hopefully this is
11  * one limit that doesn't have to be changed [again].
12  *
13  * Note that POSIX wants the FD_CLEAR(fd,fdsetp) defines to be in
14  * <sys/time.h> (and thus <linux/time.h>) - but this is a more logical
15  * place for them. Solved by having dummy defines in <sys/time.h>.
16  */
17
18 /*
19  * This macro may have been defined in <gnu/types.h>. But we always
20  * use the one here.
21  */
22 #undef __FD_SETSIZE
23 #define __FD_SETSIZE    1024
24
```

fd_set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

해당하는 인덱스의 값을 1로 설정함

select 함수

- select() 함수

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfd, fd_set *readset, fd_set *writeset,
                      fd_set *exceptset, const struct timeval *timeout);
-> 타임 아웃시 0, 오류 발생시 -1 반환, 0보다 큰 값: 변화가 발생한 파일 디스크립터의 수
```

```
struct timeval
{
    long tv_sec; // seconds
    long tv_usec; // microseconds
}
```

- maxfd: 관찰 대상이 되는 디스크립터의 수 : (maxfd -1), [0][1]
- readset: 수신 데이터의 존재 여부에 관심 있는 파일 디스크립터 정보
- writeset: 블로킹이 없는 데이터 전송 여부에 관심 있는 파일 디스크립터 정보
- exceptset: 예외 상황 발생 여부에 관심 있는 파일 디스크립터 정보
- timeout: select 함수 호출 이후에 무한 블로킹 상태에 빠지지 않도록 타임아웃 설정 시간
- 반환 값
  - 등록된 파일 디스크립터에 변화가 발생하면 0보다 큰 값이 반환
  - 이 값은 변화가 발행한 파일 디스크립터의 수를 의미

or 0 : timeout

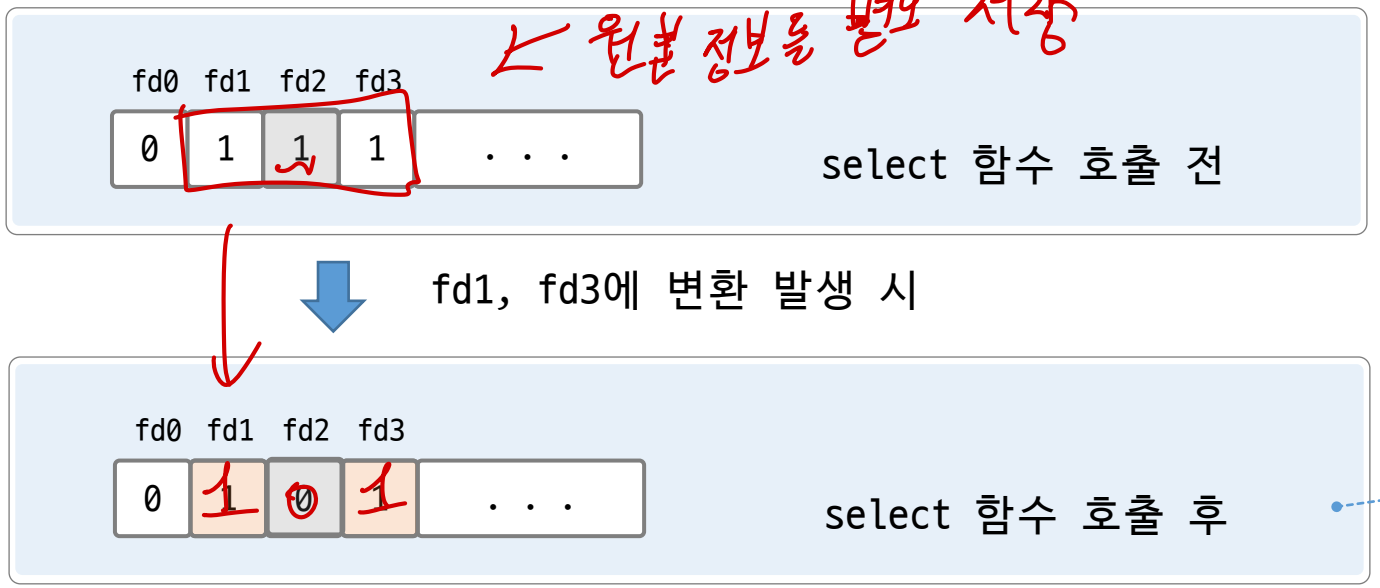
select 함수 호출 이후의 결과확인

- select() 함수 호출 이후
  - 변화가 발생한 소켓의 파일 디스크립터만 1로 설정
    - 입력 받은 데이터가 존재하는 경우
    - 출력이 가능한 상황 등
  - 변화가 없는 파일 디스크립터는 0으로 초기화 됨

그 외: 0으로 바뀜.

원본을 따로 둬야하는구나!!

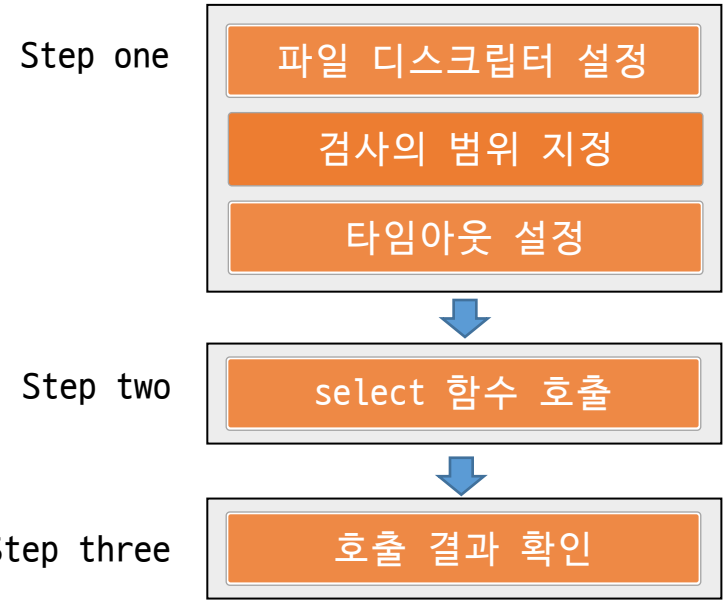
원본 정보를 별도 저장



- 변화가 발생한 파일 디스크립터만 1로 변경됨
- 기존 설정(fd_set)을 복사해야 됨

select 함수의 호출 단계

- fd_set 타입의 변수를 0으로 초기화
  - FD_ZERO(fd_set *fdset)
- 검사 범위 지정
  - FD_SET(int fd, fd_set *fdset)
  - ex) FD_SET(0, &reads):표준 입력 검사
- 타임아웃 설정
  - struct timeval timeout
  - timeout.tv_sec, timeout.tv_usec 설정
- select() 함수 호출
- 호출 결과 확인
  - FD_ISSET(int fd, fd_set *fdset)
    - 해당 파일 디스크립터에 변화가 있는지 확인



select.c 소스 파일

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

#define BUF_SIZE 30

int main(int argc, char *argv[])
{
    fd_set reads, temps;
    int result, str_len;
    char buf[BUF_SIZE];
    struct timeval timeout;

    FD_ZERO(&reads);
    FD_SET(0, &reads); //검사대상 지정 0: standard input

    while(1)
    {
        temps = reads;

        timeout.tv_sec = 5;
        timeout.tv_usec = 0;

        result = select(1, &temps, 0, 0, &timeout);

        if(result == -1)
        {
            puts("select() error!");
            break;
        }
        else if(result == 0)
        {
            puts("Time-out!");
        }
        else
        {
            if(FD_ISSET(0, &temps))
            {
                str_len = read(0, buf, BUF_SIZE);
                buf[str_len] = 0;
                printf("message from console: %s", buf);
            }
        }
    }
    return 0;
}
```

원본 fd_set

검사대상을 reads에 저장.

- select() 함수 호출 이후, 변화가 발생된 파일 디스크립터를 제외한 나머지 비트들은 0으로 초기화
- 원본 설정(reads)를 temps에 복사 후 사용

select() 함수 호출 직전에 타이머 초기화

maxfd -1 : 0번만 검사.

select(maxfd, …)
- 0 ~ maxfd-1 까지 검사함

fd=0(stdin)에서 이벤트 발생시, 화면 입력 내용을 읽어서 출력

```
$ ./select
Hi
message from console: Hi
Hello
message from console: Hello
Time-out! (5초 타임아웃 발생)
Time-out!
^C
```

멀티플렉싱 서버의 구현 1단계: echo_selectserv.c

```c
int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    struct timeval timeout;
    fd_set reads, cpy_reads;

    socklen_t adr_sz;
    int fd_max, str_len, fd_num, i;
    . . .

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    . . .

    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");

    FD_ZERO(&reads);
    FD_SET(serv_sock, &reads);
    fd_max = serv_sock;
```

serv_sock(리스닝 소켓)을 통한 연결 요청도 데이터 수신으로 구분 (관찰 대상에 추가)

1) 검사 대상: serv-sock 등록

1단계: 서버 소켓 생성과 관찰 대상 등록

- 연결 요청도 데이터 수신으로 구분되어 select 함수의 호출 결과를 통해서 확인이 가능함
- 따라서 리스닝 소켓도 관찰 대상에 포함 시킴

멀티플렉싱 서버의 구현 2단계

2단계: select() 함수의 반환 값 관찰

- 타임 아웃(0)인 경우, select() 함수를 다시 호출하기 위해 continue 문 실행
- select(fd_max+1, …)
  - ✓ select 함수가 체크하는 디스크립터는 0 ~ fd_max의 범위까지 검사를 수행하기 때문

```
while(1)
{
    cpy_reads = reads;

    timeout.tv_sec = 5;
    timeout.tv_usec = 5000;

    if((fd_num = select(fd_max+1, &cpy_reads, 0, 0, &timeout))== -1)
        break;

    if(fd_num == 0)
        continue;

    . . .
}
```

select() 함수를 호출할 때마다 타임 아웃 설정

sev_sock을 통해 수신된 데이터가 있는지 관찰

멀티플렉싱 서버의 구현 3단계

```
        for(i=0; i<fd_max+1; i++)
        {
            if(FD_ISSET(i, &cpy_reads))
            {
                if(i == serv_sock) // connection request
                {
                    adr_sz = sizeof(clnt_adr);
                    clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
                    printf("clnt_sock: %d\n", clnt_sock);
                    FD_SET(clnt_sock, &reads);
                    if(fd_max < clnt_sock)
                        fd_max = clnt_sock;
                    printf("connected client: %d\n", clnt_sock);
                }
                else // read message
                {
                    str_len = read(i, buf, BUF_SIZE);
                    if(str_len == 0)
                    {
                        FD_CLR(i, &reads);
                        close(i);
                        printf("closed client: %d\n", i);
                    }
                    else
                    {
                        write(i, buf, str_len); // echo
                    }
                }
            }
        }
    } // end of while(1)
```

등록된 fd_max 만큼 이벤트 발생 여부 확인

클라이언트 소켓을 select() 함수의 감시 대상에 추가

수신된 데이터가 serv_sock에 있으면, 연결 요청으로 이에 따른 처리를 진행함

수신된 데이터가 clnt_sock에 있으면, 에코 처리

3단계: 파일 디스크립터 확인 및 수신 데이터 처리
- serv_sock: 클라이언트의 연결 요청 처리
- clnt_sock: 클라이언트와 데이터 송수신

fd_set

serv_sock	clnt_sock	…

select 함수의 단점
- 이벤트가 발생하면, 이벤트 발생 대상을 찾기 위해 반복문을 구성해야 됨

실행 화면

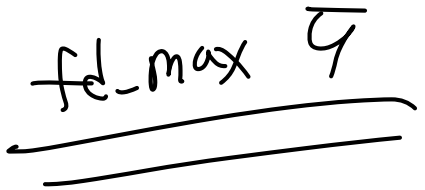
- 클라이언트: echo_client.c 사용

```
$ gcc echo_selectserv.c -o echo_selectserv
$ ./echo_selectserv 9190
connected client: 4
connected client: 5
```

fd_set reads

0	1	2	3	4	5
0	0	0	1	1	1

3: serv_sock
4: client #1
5: client #2




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

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

Questions?