Chapter 12-Select

Instructor: Limei Peng
Dept. of CSE, KNU

Problems in TCP echo client/server

- Client could be blocked in fgets and miss data from readline · Client는 fgets에서 block되고, readline에서 data를 못받을 수 O
- Sending and receiving data should be independent · 데이터를 주고받을 수 독립적이다.

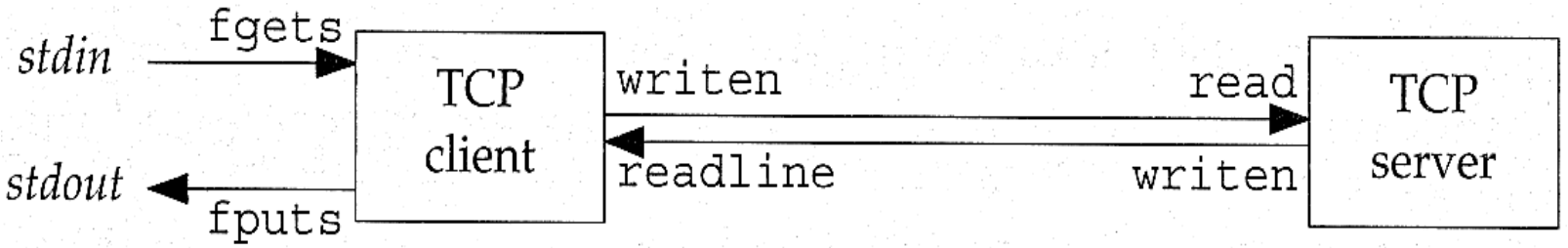
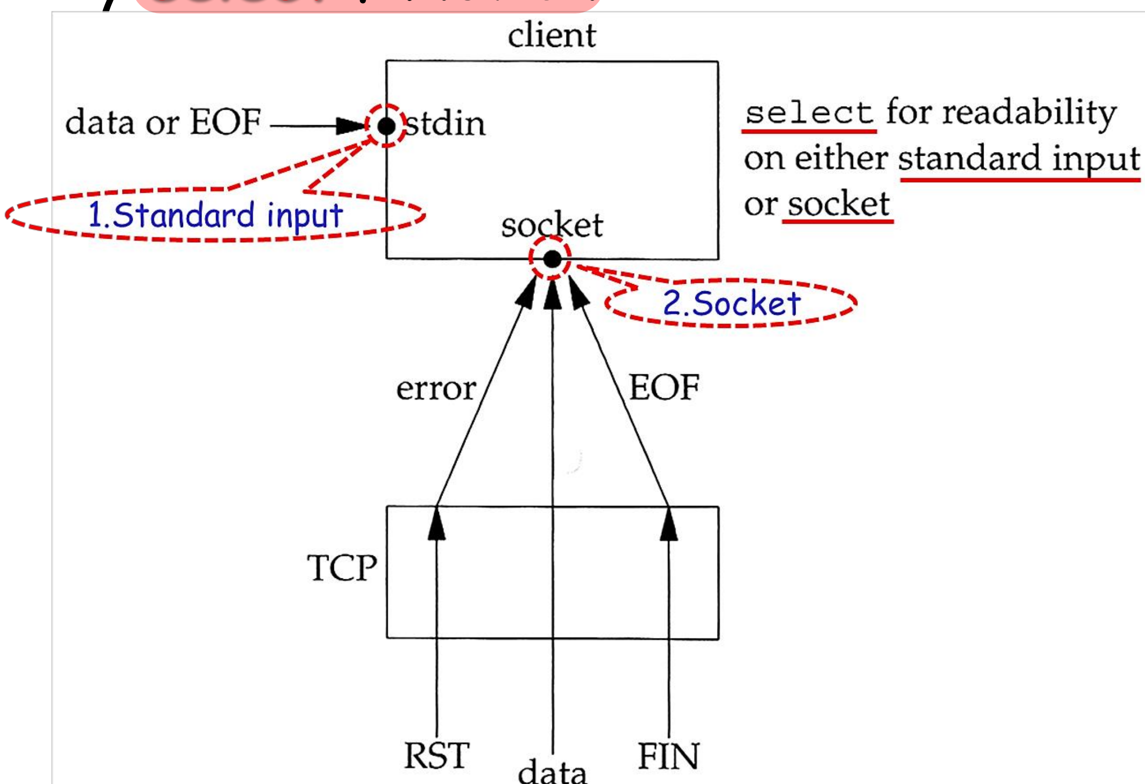


Figure 5.1 Simple echo client and server.

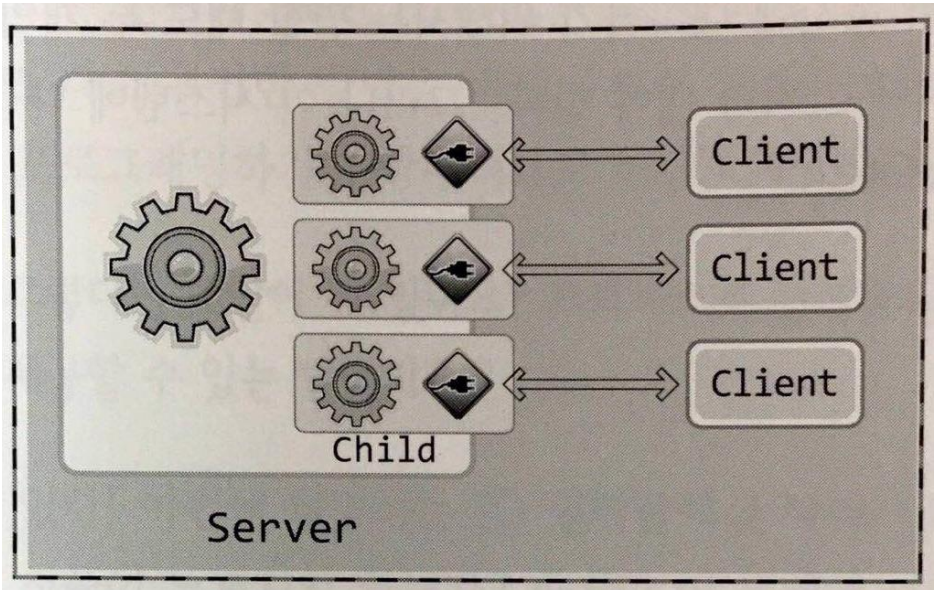
I/O Multiplexing

: 프로세스의 생성 없이 다수의 클라이언트에게 서비스를 제공할 수 있는 방법
- 하나의 통신채널을 통해 둘 이상의 데이터를 전송하는 데 사용
- 물리적 장치의 효율성을 높이기 위해, 최소한의 물리적 요소만 사용해서 최대한의 데이터를 전달해야 함.

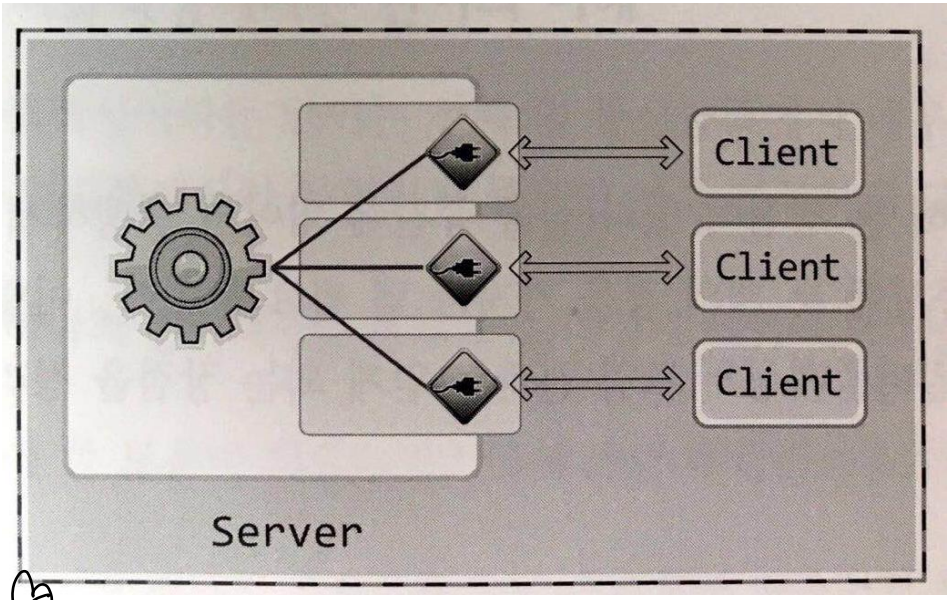
- What is I/O multiplexing? 하나 이상의 I/O 상태가 준비되었다면 알림을 받겠다고 커널에게 알릴 수 있는 능력
  - The capacity to tell the kernel that we want to be notified ***if one or more I/O conditions are ready***
    - e.g. input is ready to be read ex) Input이 read할 준비가 됨.
    - or the buffer is capable of taking more output ex) 버퍼가 more output을 받을 수 있음.
  - Provided by **select** function



I/O Multiplexing



Multi-process based server: one child process per client



Multiplexing based server: One process for several clients

select

select function

- Instruct the kernel to wait for any one of multiple events to occur : 여러 event 중 하나가 발생할 때까지 커널이 지시.
- Wake up the process only when one or more of these events occurs or when a specified amount of time has passed : 하나이상의 event 발생하거나, 특정 시간 지났을 때만 process가 wake up함.

```
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
- **maxfdp1**: specifies the **number of descriptors** to be handled. Its value is the maximum descriptor to be handled plus 1. 처리할 디스크립터의 NUM을 지정, 이 값은 처리할 max 디스크립터에 1을 더한 값.

select (Cont.)

```
struct timeval
{
  long tv_sec;  : second
  long tv_usec; : microsecond
}
```

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

- Three ways for **timeout**
  - **Wait forever**: return only when one of the specified descriptors is ready. The ***timeout*** argument is specified as NULL
  - **Wait up to a fixed time**: return when one of the specified descriptors is ready, but don't wait beyond the time specified by ***timeout***
  - **Don't wait at all**: return immediately after checking the descriptors. The two elements (i.e., tv_sec & tv_usec) of ***timeout*** is specified as both 0

select (Cont.)

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

- The middle three arguments specify the descriptors we want the kernel to handle
  - *readset*
  - *writeset*
  - *exceptset*
- Example: we can call select and tell the kernel to return only when
  - Any of the descriptors in the set {1, 4, 5} are ready for **reading**
  - Any of the descriptors in the set {2, 7} are ready for **writing**
  - Any of the descriptors in the set {1, 4} have an **exceptional** condition pending

select (Cont.)

- Macros for **fd_set** datatype

  - void **FD_ZERO**(fd_set ***fdset**);
    // **clear** all bits in ***fdset***

  - void **FD_SET**(int ***fd***, fd_set ***fdset**);
    // **turn on** the bit for ***fd*** in ***fdset***

  - void **FD_CLR**(int ***fd***, fd_set ***fdset**);
    // **turn off** the bit for ***fd*** in ***fdset***

  - int **FD_ISSET**(int ***fd***, fd_set ***fdset**);
    // is the bit for ***fd*** on in ***fdset***?

select (Cont.)

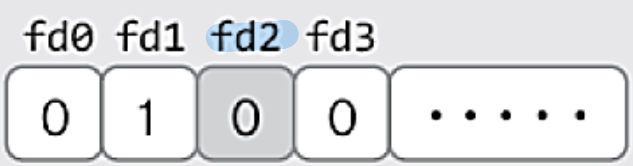
- Example of calling macros

```
int main(void)
{

   fd_set  set;

   FD_ZERO(&set);
```


```
   FD_SET(1, &set);
```


```
   FD_SET(2, &set);
```


```
   FD_CLR(2, &set);
```




```
}
```

Example#1: select.c

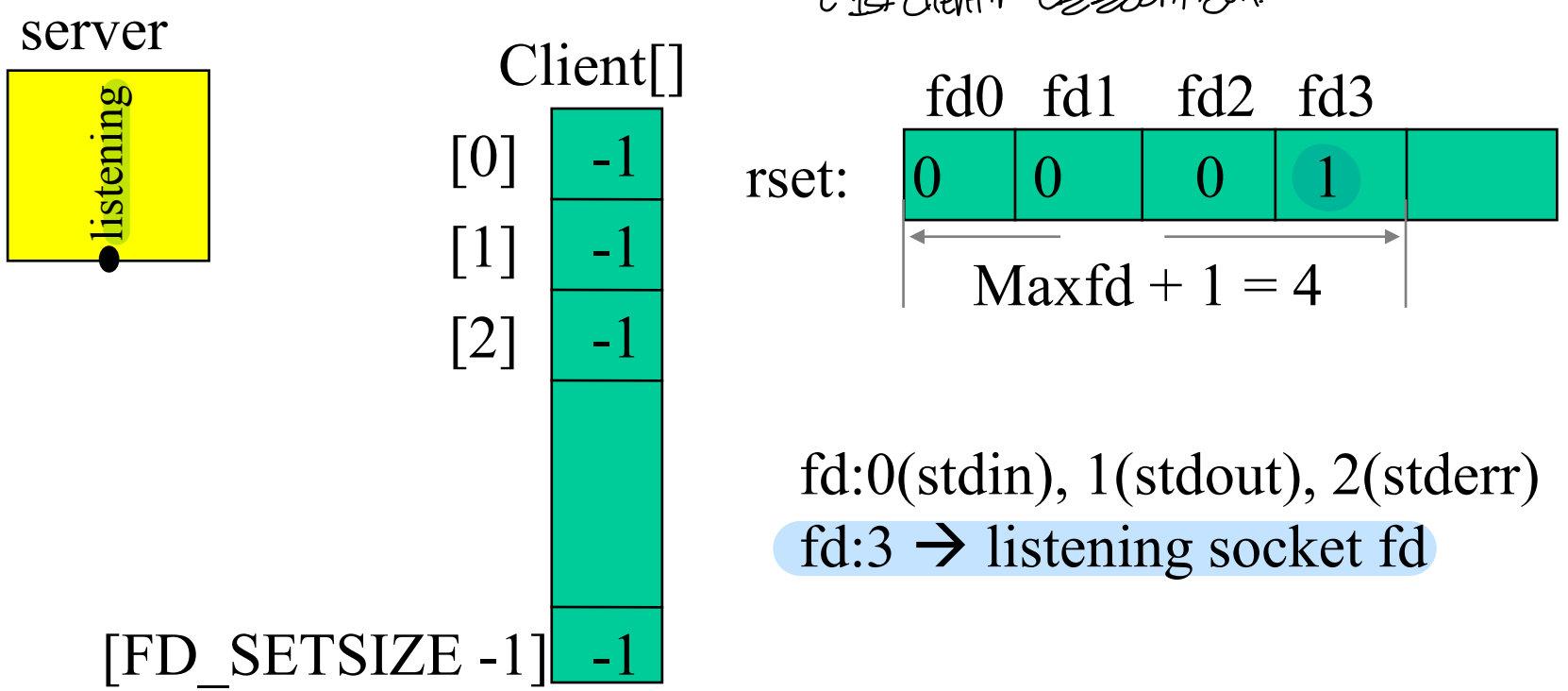
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/select.h>
#define BUF_SIZE 30
int main(int argc, char *argv[]){
        fd_set reads, temps;
        int result, str_len;
        char buf[BUF_SIZE];
        struct timeval timeout;
        FD_ZERO(&reads);
        FD_SET(0, &reads);

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

TCP Echo Server Using **select** 1/5

select를 사용하여 server를 single process로 rewrite함.

- Rewrite the server as a single process that uses **select** to handle any number of clients, instead of forking one child per client
- Before first client has established a connection

1st client가 연결되기 전이.

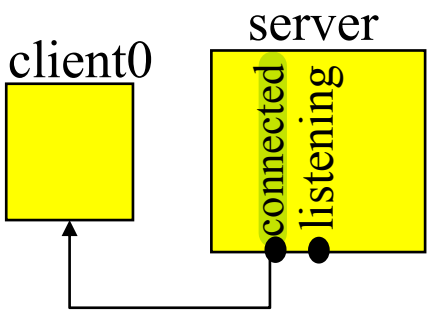


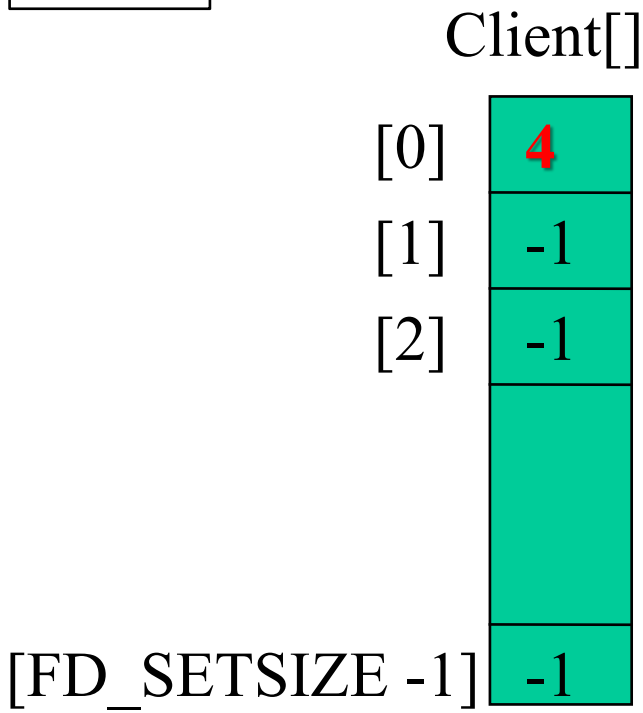
fd:0(stdin), 1(stdout), 2(stderr)

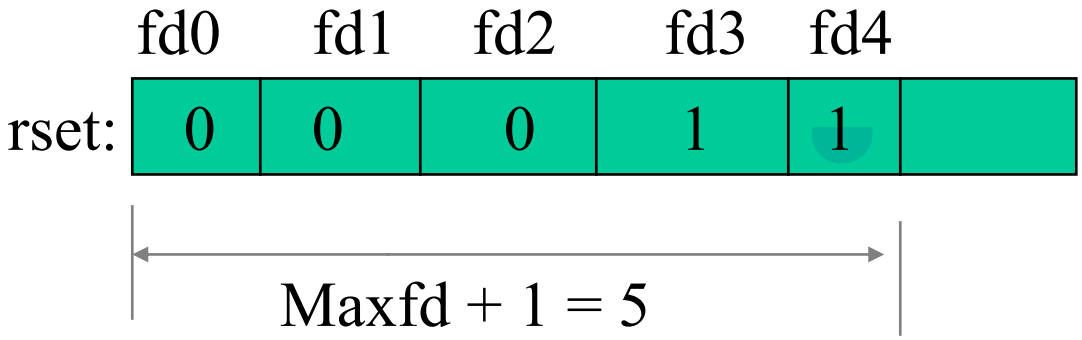
fd:3 → listening socket fd

TCP Echo Server Using **select** 2/5

- After first client connection is established (assuming connected descriptor returned by **accept** is 4)



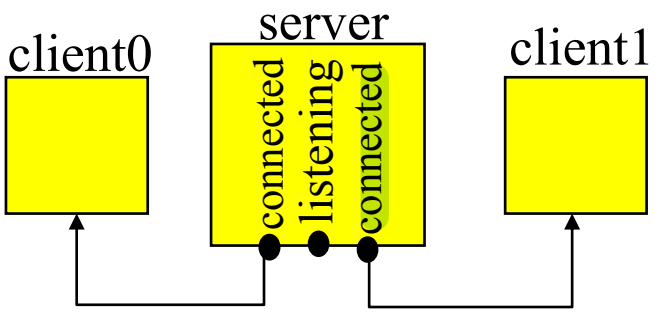


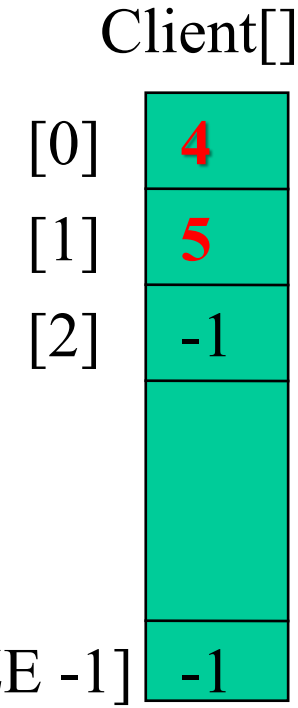


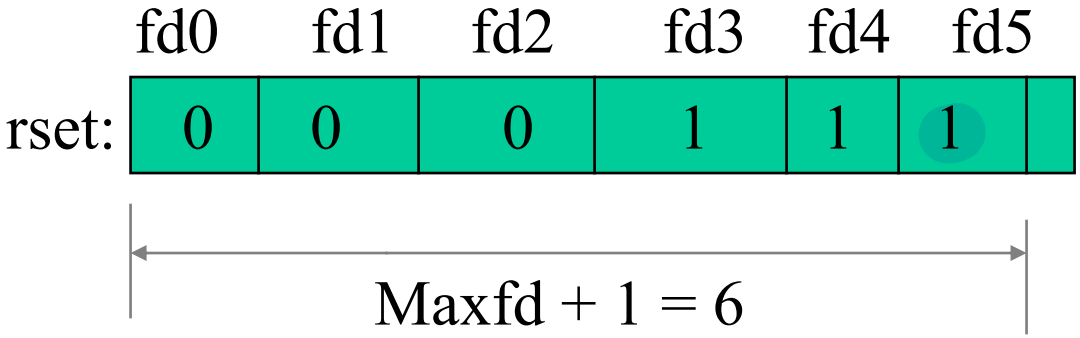
fd:0(stdin), 1(stdout), 2(stderr)
fd:3 → listening socket fd
fd:4 → first connected fd

TCP Echo Server Using **select** 3/5

- After second client connection is established (assuming connected descriptor returned by **accept** is 5)



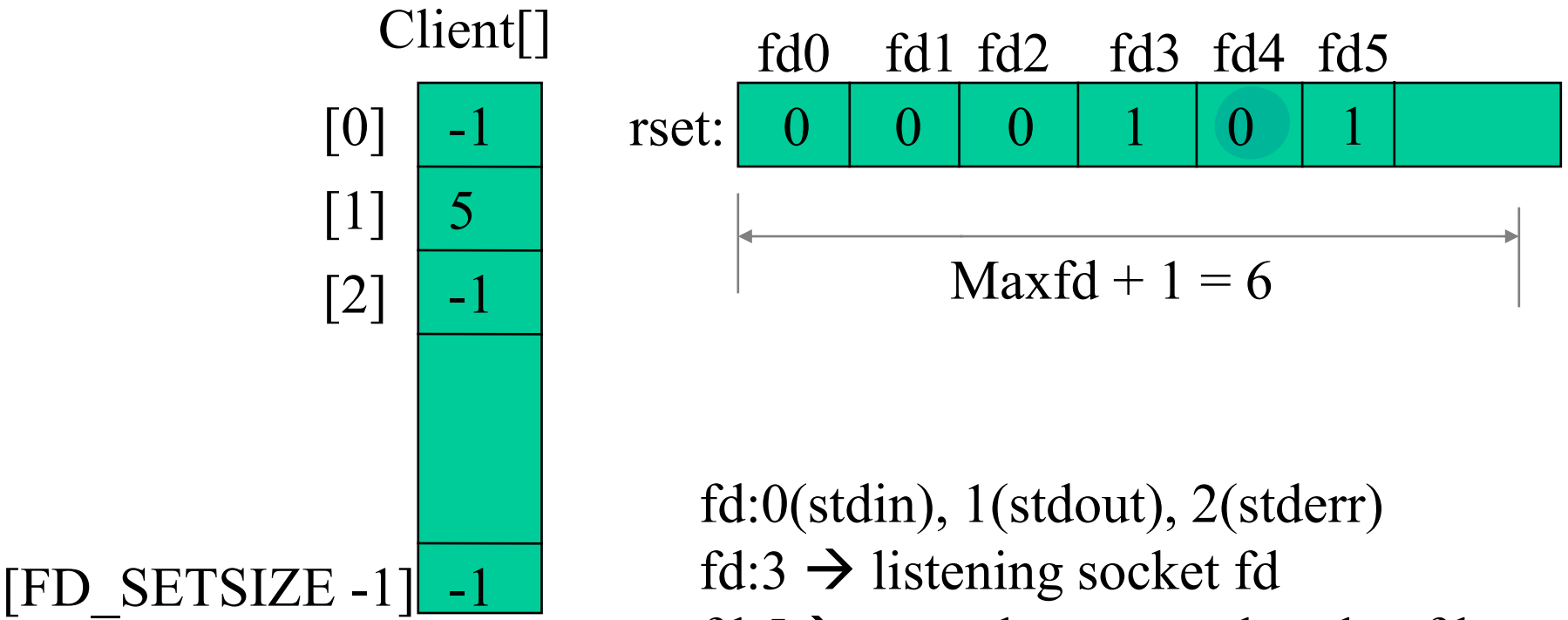




fd:0(stdin), 1(stdout), 2(stderr)
fd:3 → listening socket fd
fd:4 → first connected socket fd
fd:5 → second connected socket fd

TCP Echo Server Using select 4/5

- First client terminates its connection (fd 4 returns 0)



fd:0(stdin), 1(stdout), 2(stderr)
fd:3 → listening socket fd
fd:5→ second connected socket fd

TCP Echo Server Using **select** 5/5

- As clients arrive, record connected socket descriptor in first available entry in client array (first entry = -1)
- Add connected socket to read descriptor set
- Keep track of
  - Highest index in client array that is currently in use
  - **Maxfd** +1
- The limit on the number of clients to be served

  Min (FD_SETSIZE, **Max** (Number of descriptors allowed for this process by the kernel))

Example#2:Echo_selectserv.c(1/2)

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

#define BUF_SIZE 100
void error_handling(char *buf);

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;
        char buf[BUF_SIZE];
        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");
        FD_ZERO(&reads);
        FD_SET(serv_sock, &reads);
        fd_max = serv_sock;
```

```
        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;
                for(i = 0; i < fd_max + 1; i++){
                        if(FD_ISSET(i, &cpy_reads)){
                                if(i == serv_sock){
                                        adr_sz = sizeof(clnt_adr);
                                        clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr,
&adr_sz);

                                        FD_SET(clnt_sock, &reads);
                                        if(fd_max < clnt_sock)
                                                fd_max = clnt_sock;
                                        printf("connected client: %d \n", clnt_sock);
                                }else{
                                        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);
                                        }
                                }
                        }
                }
        }
        close(serv_sock);
        return 0;
}
void error_handling(char *buf){
         fputs(buf, stderr);
         fputc('\n', stderr);
         exit(1);
}
```

Example#2: Echo_selectserv.c (2/2)

Execution Results

Server

```
np2019@ubuntu:~/NP$ ./echo_selectserv 1234
connected client: 4
connected client: 5
closed client: 4
closed client: 5
^C
np2019@ubuntu:~/NP$
```

Client 1

```
np2019@ubuntu:~/NP/1st week$ ./echo_client 127.0.0.1 1234
Connected.......
Input message(Q to quit): Hi~
Message from server: Hi~
Input message(Q to quit): Good bye
Message from server: Good bye
Input message(Q to quit): Q
np2019@ubuntu:~/NP/1st week$
```

Client 2

```
np2019@ubuntu:~/NP/1st week$ ./echo_client 127.0.0.1 1234
Connected.......
Input message(Q to quit): Nice to meet you~
Message from server: Nice to meet you~
Input message(Q to quit): Bye~
Message from server: Bye~
Input message(Q to quit): Q
np2019@ubuntu:~/NP/1st week$
```