KWANGWOON UNIVERSITY

Network Programming

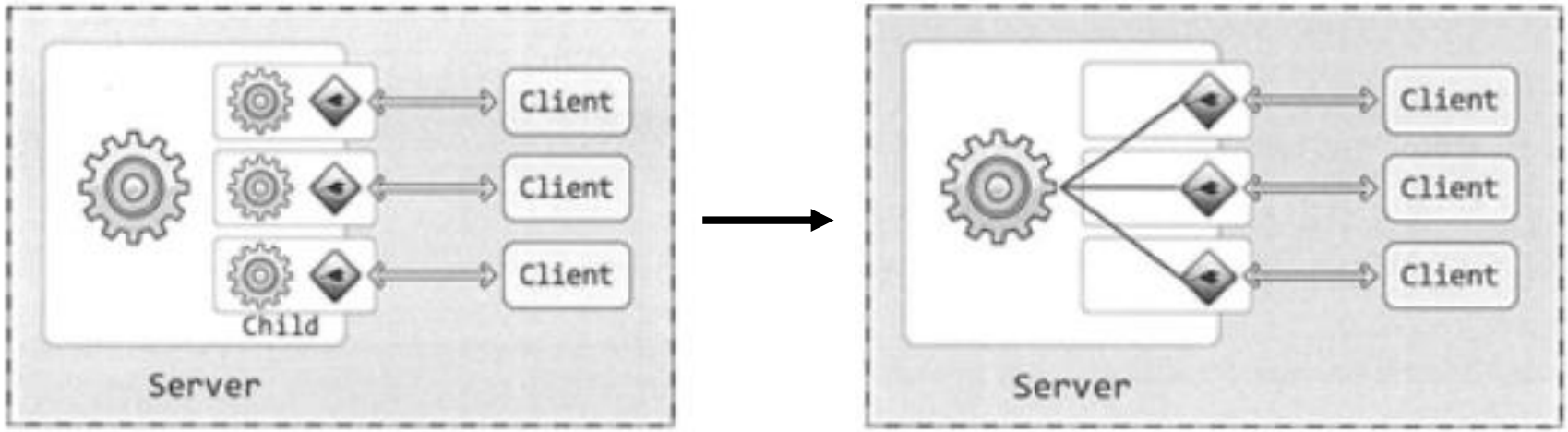
06. I/O Multiplexing
The select and poll Functions

section

- **6.1 Introduction**
- **6.2 I/O Models**
- **6.3** select **Function**
- **6.4** str_cli **Function (Revisited)**
- **6.5 Batch Input and Buffering**
- **6.6** shutdown **Function**
- **6.7** str_cli **Function (Revisited Again)**
- **6.8 TCP Echo Server (Revisited)**
- **6.9** pselect **Function**
- **6.10** poll **Function**
- **6.11 TCP Echo Server (Revisited Again)**

1. Introduction

- I/O Multiplexing이란?
  - 여러 소켓에 대해서 I/O를 병행적으로 하는 것
  - 다수의 프로세스나 쓰레드를 만들지 않고도 여러 파일을 처리



1. Introduction

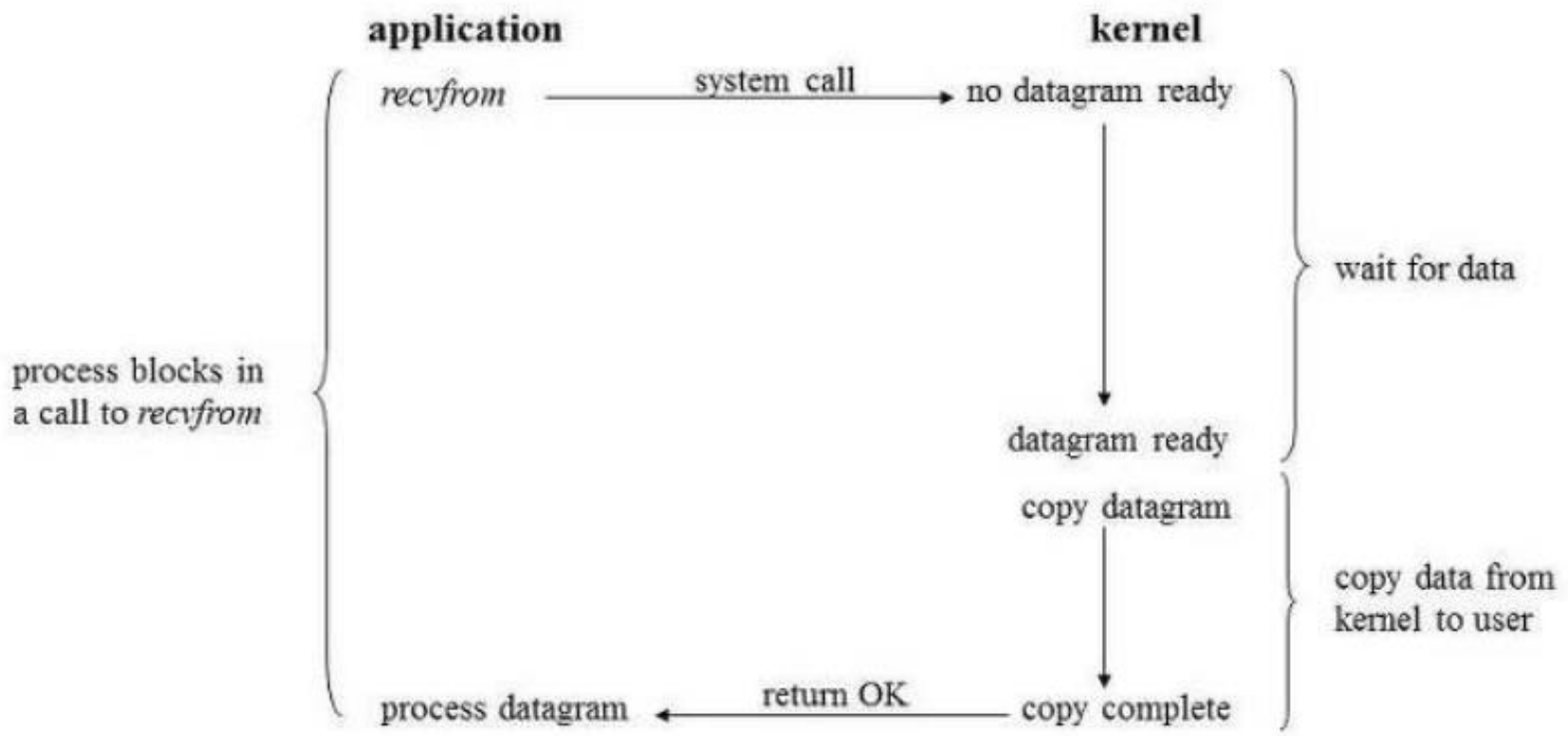
- When I/O Multiplexing is used as a network application
  1. When a client is handling multiple descriptors
  2. A client to handle multiple sockets at the same time (rare case)
  3. When a TCP server handles both a listening socket and its connected sockets
  4. When a server handles both TCP and UDP
  5. When a server handles multiple services and protocols

2. I/O Models

- Five I/O models that are available to us under Unix
  1. Blocking I/O
  2. Nonblocking I/O
  3. I/O multiplexing (select and poll)
  4. Signal driven I/O (SIGIO)
  5. Asynchronous I/O (the POSIX aio_functions)
- Two distinct phases for and input operation
  1. Waiting for the data to be ready
  2. Copying the data form the kernel to the process

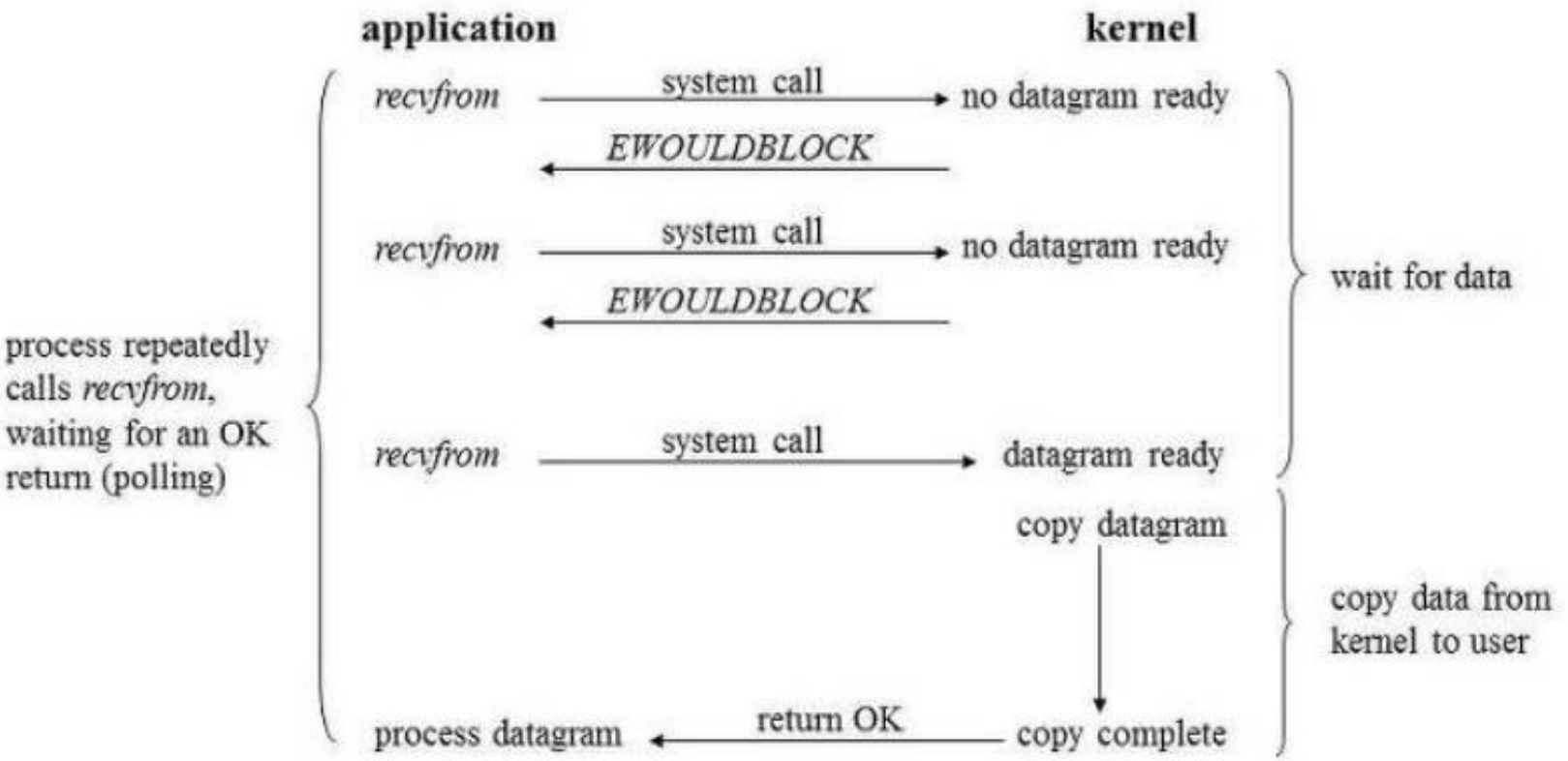
2. I/O Models

- **Blocking I/O Model**



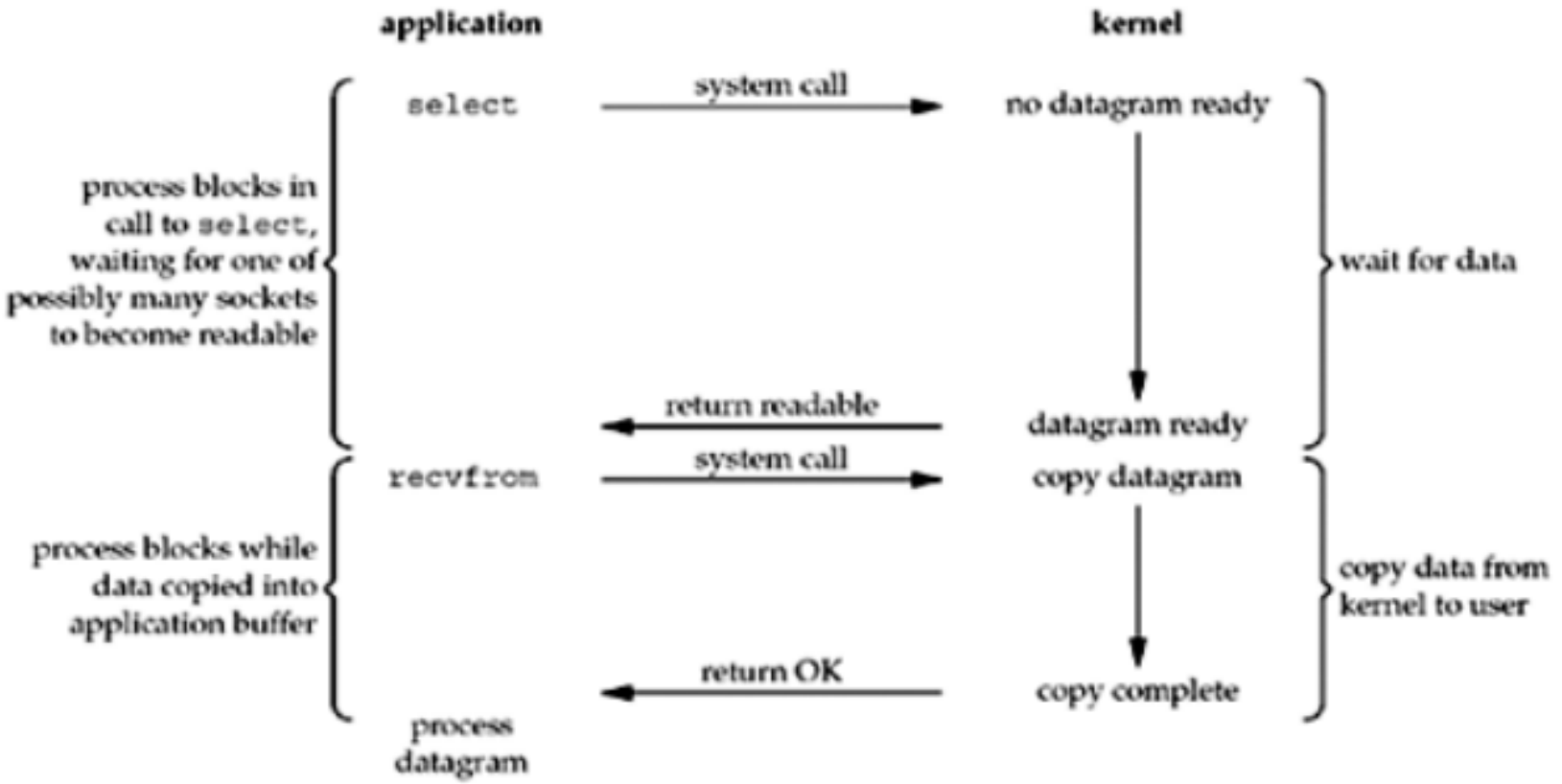
2. I/O Models

- **Nonblocking I/O Model**



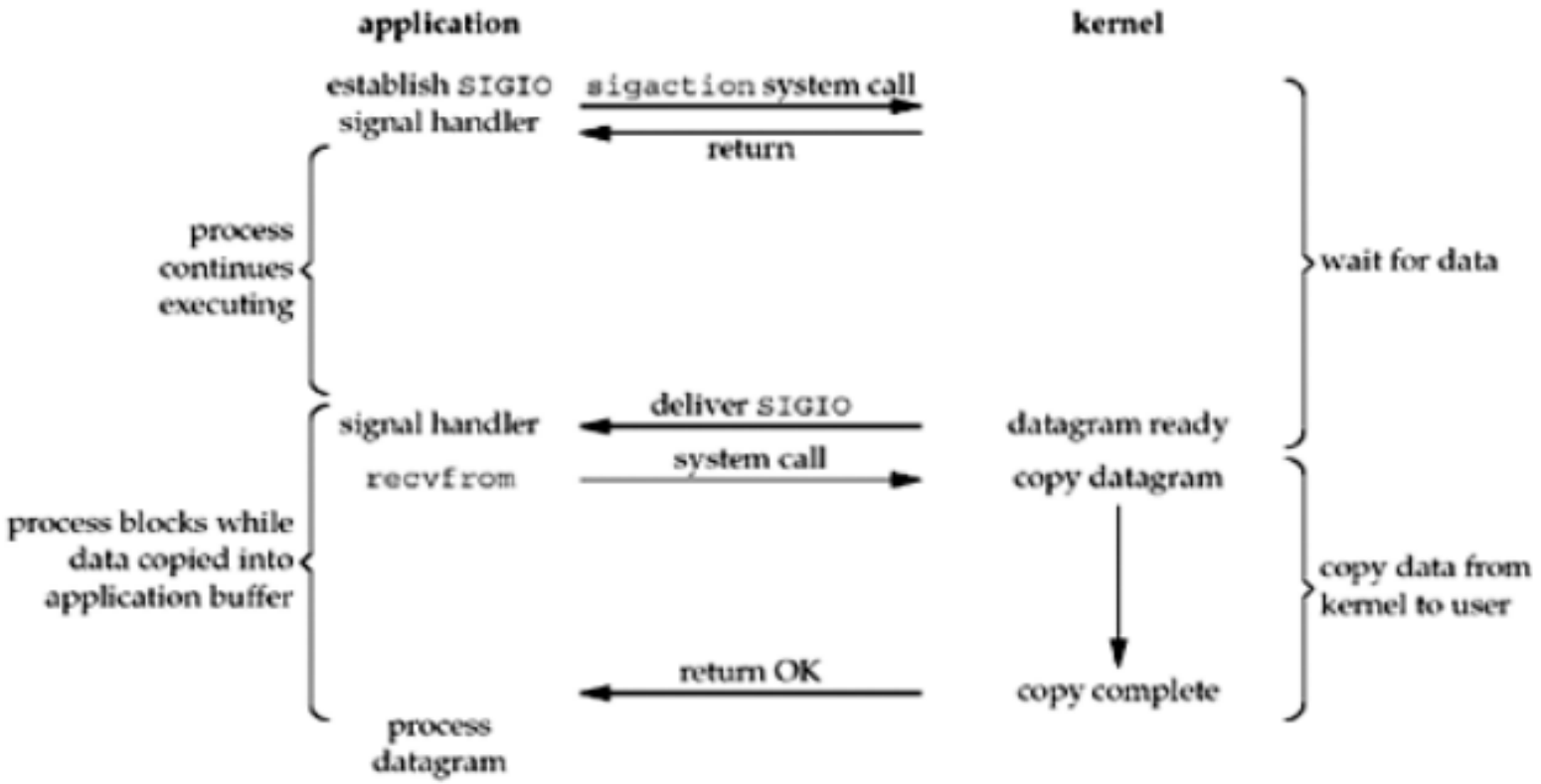
2. I/O Models

- I/O Multiplexing Model



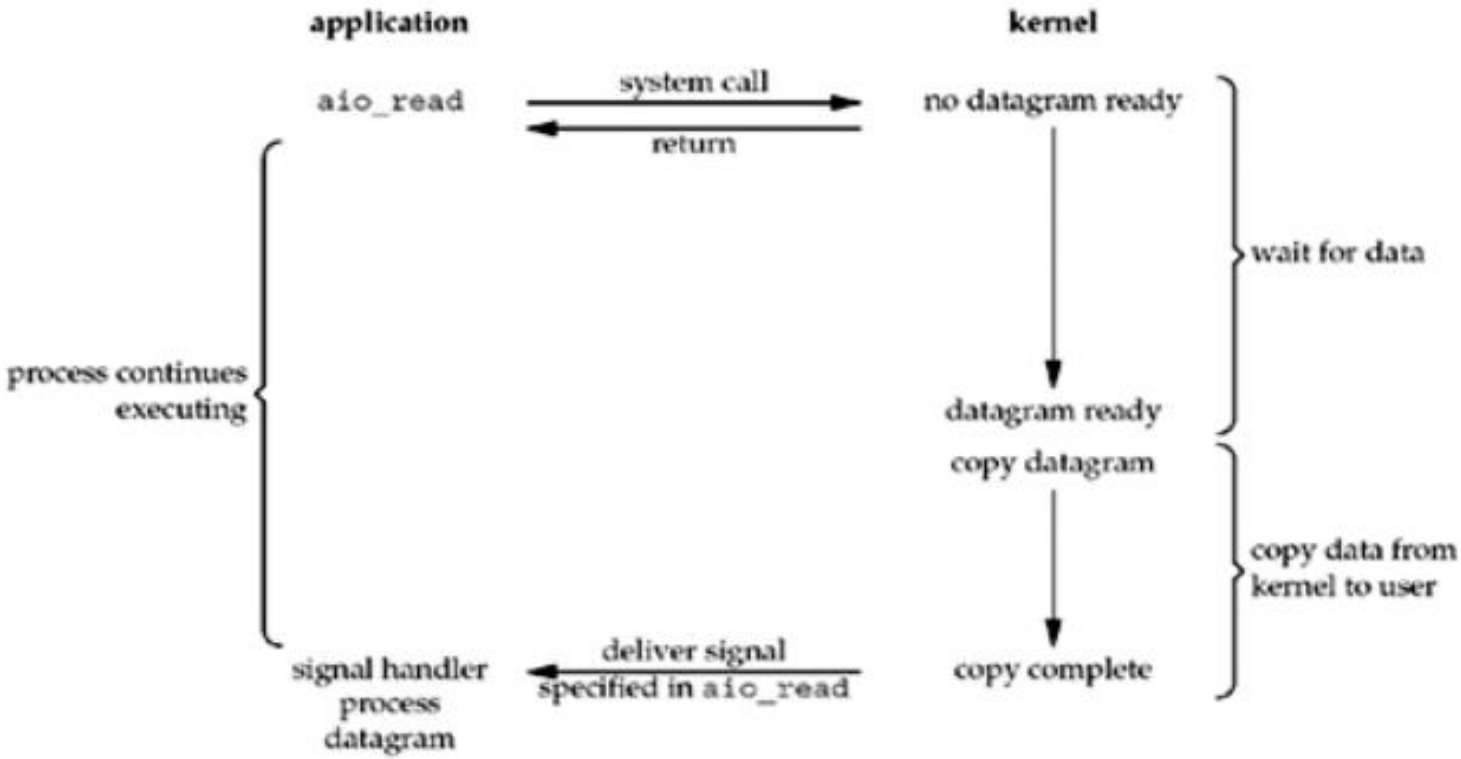
2. I/O Models

- Signal-Driven I/O Model



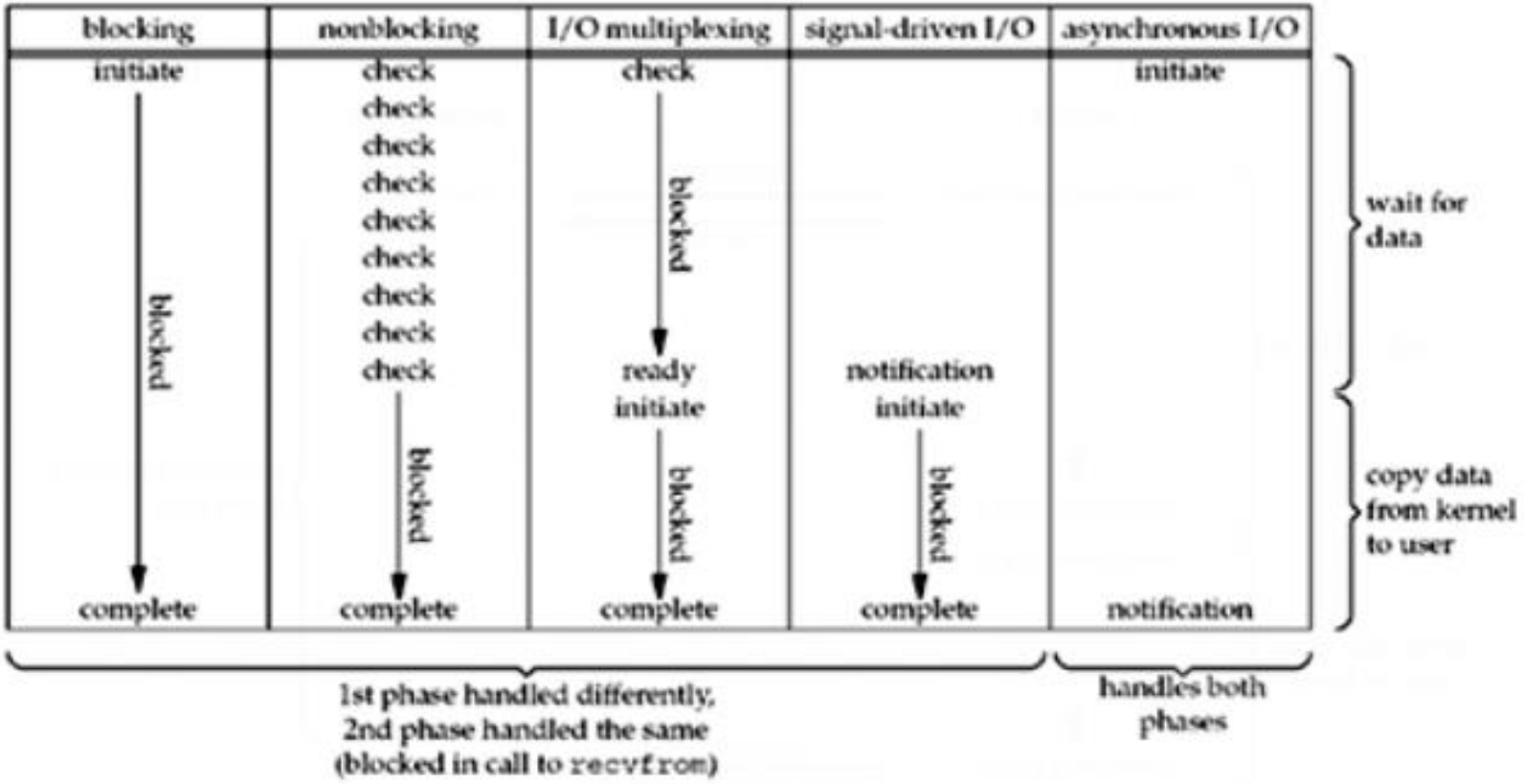
2. I/O Models

- Asynchronous I/O Model



2. I/O Models

- Comparison of the I/O Models



3. select **Function**

- 관심있는 descriptor (readable, writable, exception)이 무엇인지, 얼마나 시간이 지났는지 kernel에게 전달한다.

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

#include <sys/time.h>

int  select(int  maxfdp1,  fd_set  *readset,  fd_set  *writeset,  fd_set
*exceptest, const struct timeval *timeout);

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

```
struct timeval {

       long   tv_sec;       /* seconds */

       long   tv_usec;             /* microseconds */

};
```

3. select **Function**

- Timeout값에 따른 3가지 경우
  1. Wait Forever
     - Timeout값이 null pointer인 경우
     - Fd중 하나가 준비되거나 signal이 잡힐 때 까지 차단된다.
  2. Do not wait at all
     - timeout값이 0인 경우
     - Descriptor를 체크한 후 바로 return -> polling
  3. Wait up to a fixed amount of a time
     - timeout값이 0이 아닌 경우
     - Timeval 구조체에 저장된 시간 값만큼 기다린다.

3. select Function

- How to assign and verify fds on a fd_set

1. void FD_ZERO (fd_set *fdset);
   주어진 fd_set의 모든 비트를 0으로 만든다.
   -> select 호출하기전 매번 FD_ZERO 호출

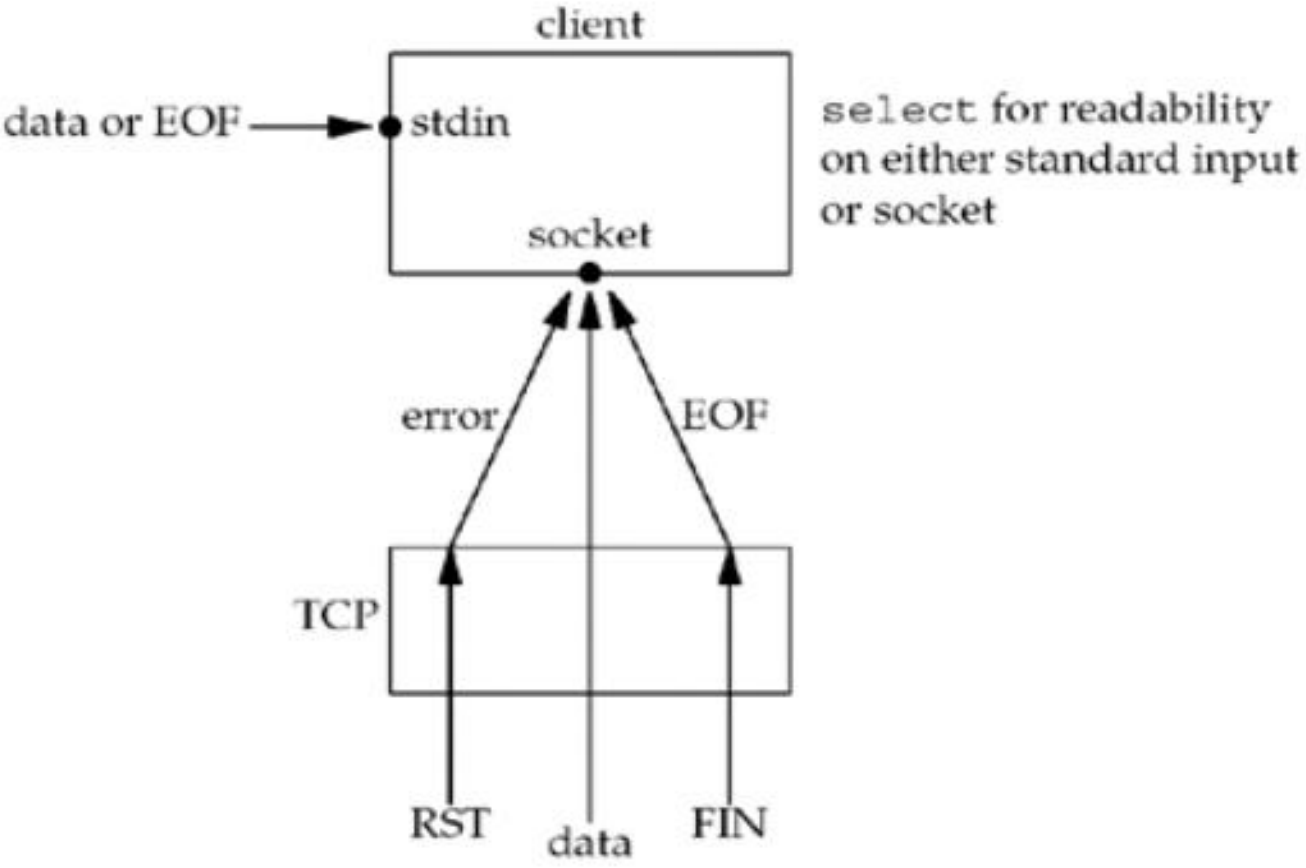
2. Void FD_SET (int fd, fd_set *fdset);
   집합의 특정 비트를 켤 때 사용
   -> fd를 fd_set에 추가할 때 사용

3. void FD_CLR (int fd, fd_set *fdset);
   집합의 특정 비트를 끌 때 사용
   -> fd를 fd_set에서 제거할 때 사용

4. Void FD_ISSET (int fd, fd_set *fdset);
   특정 비트가 켜져 있는지 확인할 경우 사용
   -> select() 호출이 반환된 후 fd가 입출력 준비가 끝났는지 점검하기 위해 사용

4. str_cli Function (Revisited)

- **Various conditions that are handled by our call to** select



4. str_cli Function (Revisited)

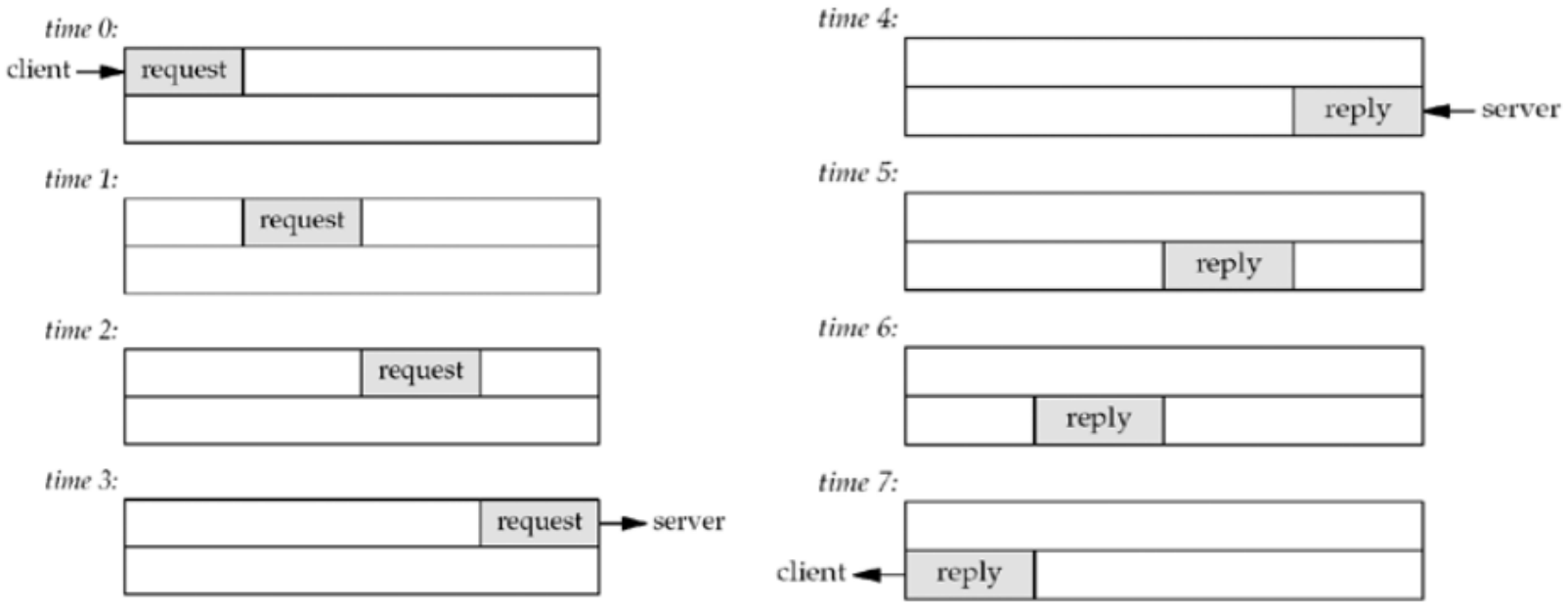
```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int     maxfdp1;
    fd_set  rset;
    char    sendline[MAXLINE], recvline[MAXLINE];

    FD_ZERO(&rset);
    for ( ; ; )  {
        FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd)  +  1;
        Select(maxfdp1,  &rset,  NULL,  NULL,  NULL);

        if (FD_ISSET(sockfd,  &rset))  {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0)
                err_quit("str_cli: server terminated prematurely");
            Fputs(recvline, stdout);
        }

        if (FD_ISSET(fileno(fp), &rset))  {  /*  input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL)
                return;           /* all done */
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

5. Batch Input and Buffering

- RTT (Round Trip Time)
  - 왕복지연시간
  - 상대측 호스트까지 패킷이 왕복하는데 걸리는 시간

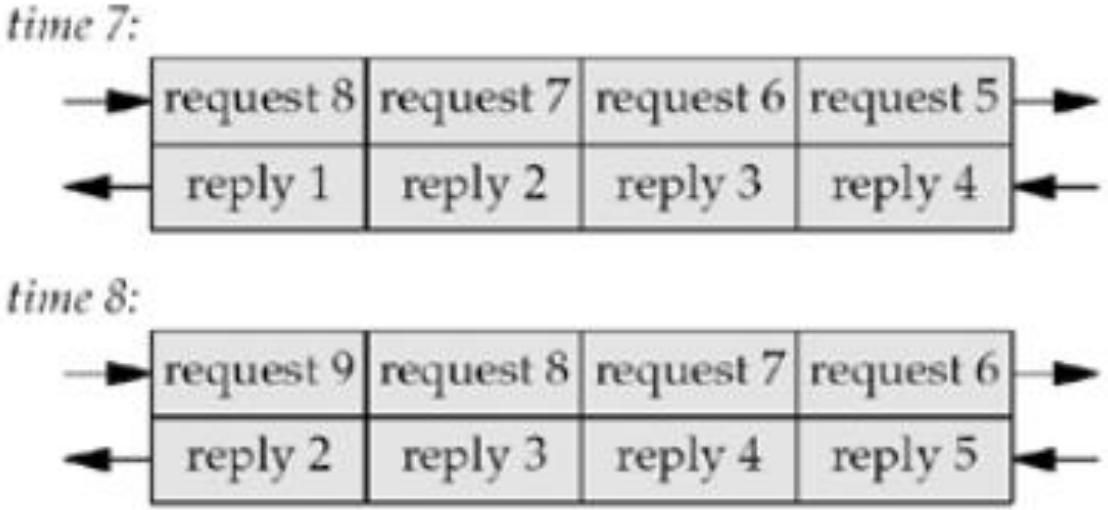
- IP 패킷의 RTT
  - Ping 명령어를 사용
  - RTT및 TTL(Time To Live)(Ip 패킷수명) 수치를 알 수 있다.

5. Batch Input and Buffering



- Stop-and wait mode
  - client측에서 한 개의 request를 보내고 server가 받고 나서 reply를 받은 후에 다시 requset를 보낼 수 있다.
  - 전송 효율이 떨어진다.

5. Batch Input and Buffering



- Batch mode (일괄 처리 형태)
  - request 1을 보내고 바로 request 2를 보낸다
  - stop-and-wait mode와 달리 reply가 오는 것을 기다리지 않는다.
  - 대량의 data를 처리한다
  - 일괄적으로 처리한다.

6. shutdown Function

- **Two limitations with** close **that can be avoided with** shutdown
  1. **Close() 사용시 descriptor의 reference count 감소**
     -> count가 0이 되면, socket closes
     => shutdown() 사용하면 reference수에 관계 없이
     TCP의 연결,종료 sequence 초기화 가능

  2. **Close() 사용시 소켓의 read,write 모두 종료**
     -> 전송할 data가 남아있는 경우 shutdown() 사용
     => half-close 상태를 구현할 때 shutdown()을 사용한다.

6. shutdown **Function**

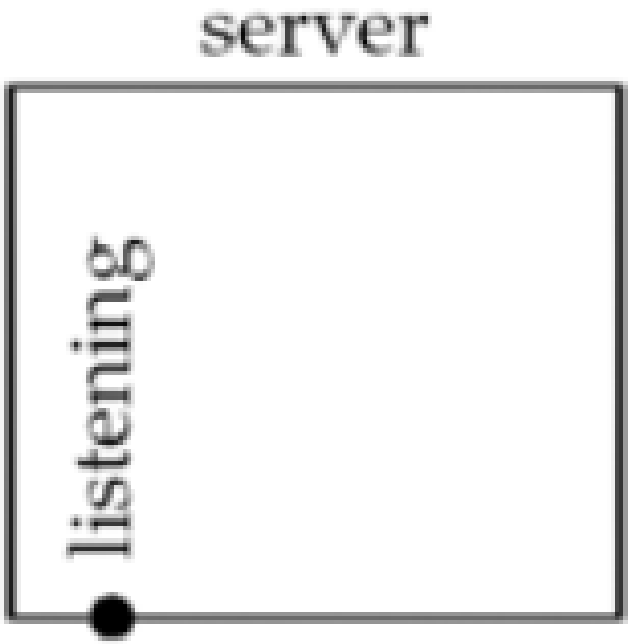
```
#include <sys/socket.h>


int shutdown(int sockfd, int howto);
```

- **Depneds on the value of the** howto **argument**
  1. **SHUT_RD**
     recv buffer만 차단한다.
     -> 해당 소켓으로부터 통신을 수신할 수 없다.
  2. **SHUT_WR**
     send buffer만 차단한다.
     -> 해당 소켓에게 송신할 수 없다.
  3. **SHUT_RDWR**
     두 버퍼 모두 차단한다.
     -> 해당 소켓과 송수신을 할 수 없다.

7. str_cli Function (Revisited Again)

```
#include            "lnp.h"
#incldue            <unistd.h>

void
str_cli(FILE *fp, int sockfd)
 {
       int    maxfdp1, stdineof;
       fd_set rset;
       char   buf[MAXLINE];
       int    n;
       stdineof = 0;
       FD_ZERO(&rset);
       for ( ; ; ) {
              if (stdineof == 0)
                     FD_SET(fileno(fp), &rset);
              FD_SET(sockfd, &rset);
              maxfdp1 = max(fileno(fp), sockfd) + 1;
              select(maxfdp1, &rset, NULL, NULL, NULL);
              if (FD_ISSET(sockfd, &rset) ) {
                     if ( (n = read (sockfd, buf, MAXLINE) == 0)
                            if (stdineof == 1)
                                   return;
                            else
                               err_quit("str_cli:server terminated ");
                     }
                     write (fileno (stdout), buf, n);
              }
              if (FD_ISSET (fileno (fp), &rset) ) {
                     if ( (n = read(fileno(fp), buf, MAXLINE) ) == 0)
       {
                            stdineof = 1;
                            shutdown (sockfd, SHUT_WR) ;/* FIN */
                            FD_CLR (fileno (fp), &rset);
                            continue;
                     }
                     write (sockfd, buf, n);
              }
       }
}
```

8. TCP Echo Server (Revisited)



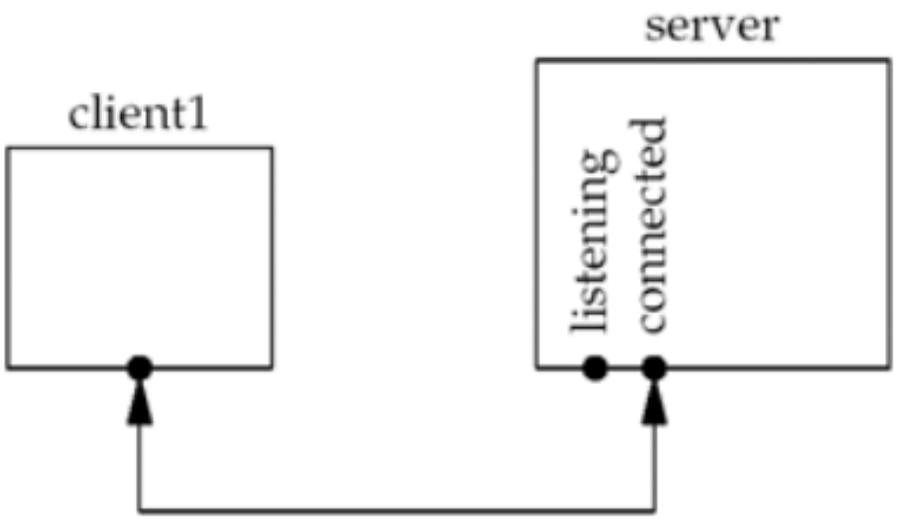
- The first client가 connection을 설정하기 전의 state of the server
- 하나의 listening descriptor만 가지고 있다.

8. TCP Echo Server (Revisited)

- Descriptor 0 : 표준 입력, 1 : 표준 출력, 2 : 표준 오류
- Listening socket이 사용가능한 first descriptor : 3

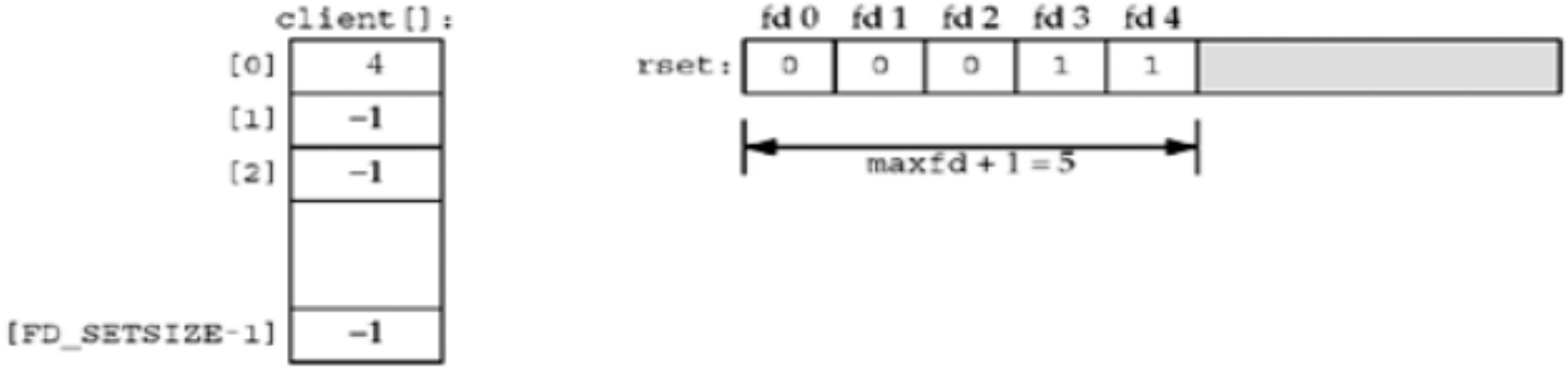
- 각 클라이언트에 연결된 socket descriptor를 포함하는 client라는 정수배열
  -> 배열의 요소는 -1로 초기화한다.

8. TCP Echo Server (Revisited)



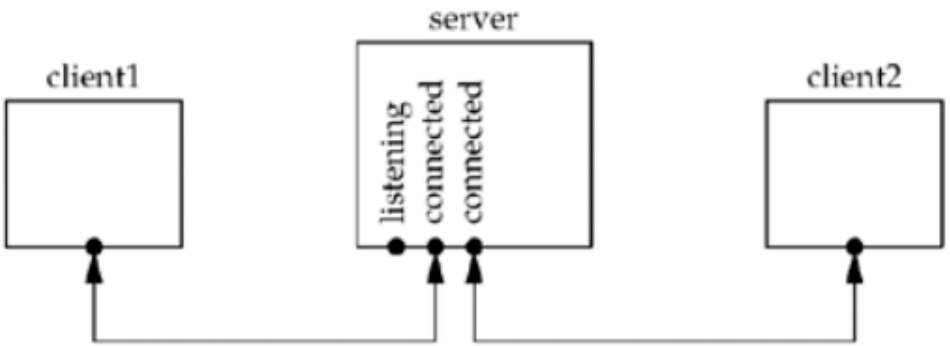
- Client에서 server로의 연결이 이루어진 상태
- Listening descriptor가 readable하게 된다.

8. TCP Echo Server (Revisited)

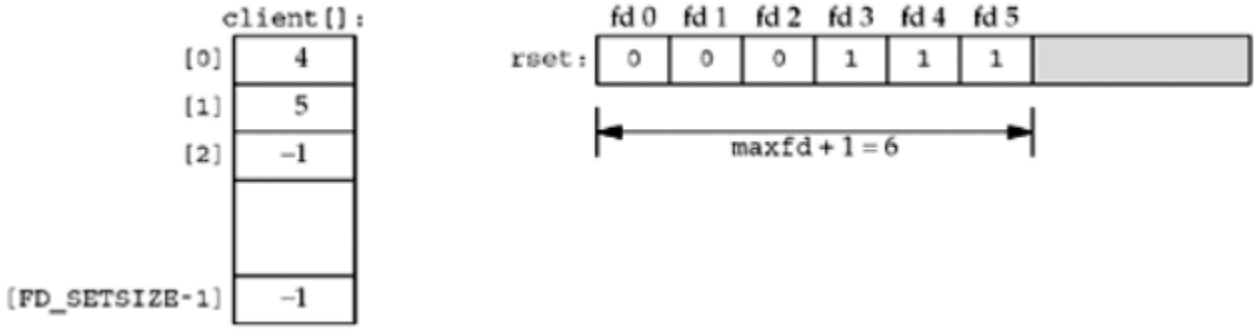


- **Server가 client 배열의 새로운 connection socket을 descriptor set에 추가해야 한다.**
- **Descriptor 4 : first connected client**
- **Server가** accept**를 call**
  -> accept**에 의해 반환된 새로운 connection descriptor : 4**

8. TCP Echo Server (Revisited)

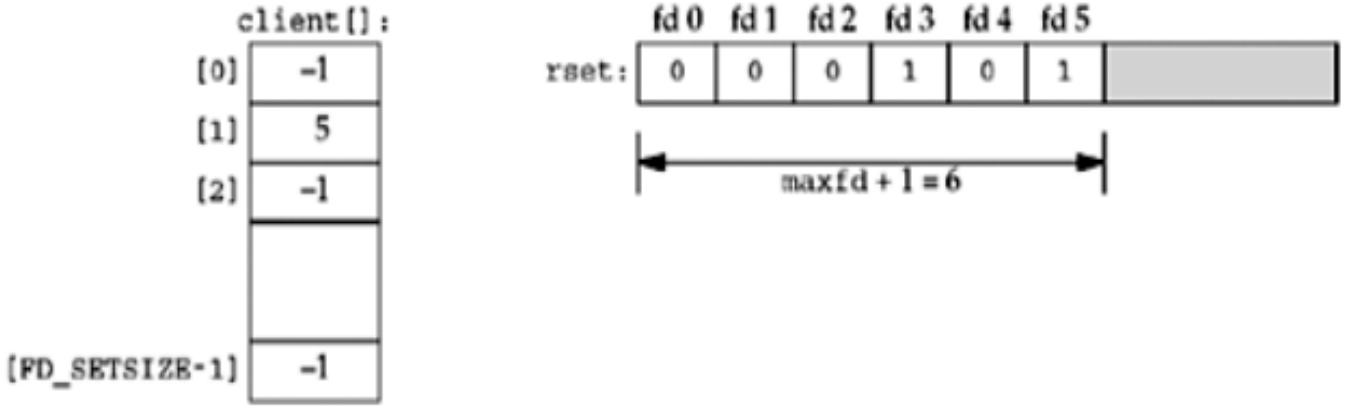


- Second client가 server와 connection이 이루어진 경우



- Descriptor 5 : second connected client

8. TCP Echo Server (Revisited)



- First client가 connection을 종료하는 경우
  -> client TCP가 server에서 descriptor 4를 읽을 수 있게 하는 FIN을 전송
  -> server가 connection된 socket을 읽을 때, read가 0 반환
  -> client 배열의 [0]이 -1로 설정
  -> descriptor set의 descriptor 4는 0으로 설정

9. pselect **Function**

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

#include <signal.h>

#include <time.h>

int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct
timespec *timeout, const sigset_t *sigmask);
```

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

9. pselect **Function**

- pselect **contains two changes from** select **function**
  1. pselect **는 timespec 구조체를 사용.**

```
struct timespec {
  time_t tv_sec;        /* seconds */
  long   tv_nsec;       /* nanoseconds */
};
```

  2. pselect**는 여섯 번째 인자(sigmask)가 추가된다.**
     : a pointer to a signal mask
     - sigmask가 NULL이면
     : select**와 동일한 기능**
     - sigmask가 NULL이 아니면
     : sigmask가 가리키는 signal mask가 자동으로 설정 되어 차단되고, pselect **호출이 반환될 때 signal mask가 복원되어 실행하게 된다.**

10. poll Function

```
#include <poll.h>


int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);

struct pollfd {
  int      fd;        /* descriptor to check */
  short    events;    /* events of interest on fd */
  short    revents;   /* events that occurred on fd */
};
```

- fd : file descriptor
- events : monitoring할 event 종류
- revents : 반환할 event
- nfds : 설정된 fds의 개수
- Timeout : ms단위의 timeout 설정

10. poll Function

- Events member
  - POLLIN : 읽을 자료가 존재한다.
  - POLLRDNORM : 읽을 일반 자료가 존재한다.
  - POLLRDBAND : 우선권이 있는 읽을 자료가 존재한다.
  - POLLPRI : 우선적으로 읽어야할 자료가 존재한다.
  - POLLOUT : 차단 없이 쓸 수 있다.
  - POLLWRNORM : 차단 없이 쓸 수 있다.
  - POLLWRBAND : 우선권이 있는 자료를 차단 없이 쓸 수 있다.
- Revents member
  - POLLERR : fd에 오류가 발생했다.
  - POLLHUP : fd에 연결이 끊겼다.
  - POLLNVAL : fd가 유효하지 않다.

10. poll Function

- **Timeout == -1**
  Wait forever

- **Timeout == 0**
  Return immediately, do not block

- **Timeout >0**
  Wait specified number of milliseconds

11. TCP Echo Server (Revisited Again)

```
#include     "unp.h"
#include     <limits.h>          /* for OPEN_MAX */

int
main(int argc, char **argv)
{
    int     i, maxi, listenfd, connfd, sockfd;
    int     nready;
    ssize_t n;
    char    buf[MAXLINE];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    client[0].fd = listenfd;
    client[0].events = POLLRDNORM;
    for (i = 1; i < OPEN_MAX; i++)
        client[i].fd = -1;      /* -1 indicates available entry */
    maxi = 0;                   /* max index into client[] array */
```

```
for ( ; ; ) {
    nready = Poll(client, maxi + 1, INFTIM);

    if (client[0].revents & POLLRDNORM) {  /* new client connection */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

        for (i = 1; i < OPEN_MAX; i++)
            if (client[i].fd < 0) {
                client[i].fd = connfd;  /* save descriptor */
                break;
            }
        if (i == OPEN_MAX)
            err_quit("too many clients");
        client[i].events = POLLRDNORM;
        if (i > maxi)
            maxi = i;       /* max index in client[] array */

        if (--nready <= 0)
            continue;       /* no more readable descriptors */
    }

    for (i = 1; i <= maxi; i++) {       /* check all clients for data */
        if ( (sockfd = client[i].fd) < 0)
            continue;
        if (client[i].revents & (POLLRDNORM | POLLERR)) {
            if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
                if (errno == ECONNRESET) {
                        /* connection reset by client */
                    Close(sockfd);
                    client[i].fd = -1;
                } else
                    err_sys("read error");
            } else if (n == 0) {
                    /* connection closed by client */
                Close(sockfd);
                client[i].fd = -1;
            } else
                Writen(sockfd, buf, n);
            if (--nready <= 0)
                    break;                /* no more readable descriptors */
        }
    }
}
```