Lecture 9: I/O Multiplexing

(11/18/2009)

<u>Lecture Outline</u>

1. Introduction: I/O model and I/O multiplexing
2. The select function
3. Revised str_cli function (using select function)
4. Batch input and its effects
5. Shutdown function
6. Re-revised client-server example implementation (TCP & UDP)
7. pselect function
8. poll function

1. Introduction: I/O models and I/O multiplexing

(1) Motivations: a process may have the need to deal with more than one descriptor simultaneously.

a. Example 1: A line printer capable of accepting external requests through LANs.

(a) Two sockets are opened - a Unix domain socket for local requests and a TCP socket for external requests.

(b) Problems: which socket should the print daemon examine? Possible solutions:

· Use “nonblocking” I/O by calling *fcntl* to set both socket as nonblocking. *Polling* has to be performed – waste CPU cycles.

· *fork* one child process to handle each of the two channels. Data read by the child processes must be returned to the parent process through some form of IPC is needed.

· Use asynchronous I/O. Three problems: signals are expensive to catch; if more than one descriptor is enabled for asynchronous I/O, the occurrence of an I/O signal does not tell which descriptor is ready for I/O; also asynchronous I/O is only supported for terminals and socket under 4.3BSD.

· The *select* function: to be discussed in this subsection.

b. Example 2: the client process in our client-server example in Chapter 5. It has two handle two descriptors simultaneously: an file input descriptor and a socket descriptor.

c. Example 3: a service that is available with two different protocols (TCP and UDP). Two sockets, one for TCP and another for UDP, have to be monitored by the server.

(2) I/O Models

a. Two distinct phases of an input operation:

   (a) Waiting for the data to be ready;

   (b) Copying the data from the kernel to the process.

b. Blocking I/O Model

   (a) Most commonly used. A process will be blocked (put in sleep) at a socket descriptor until data is ready at the descriptor;

   (b) Illustration diagram: Fig.6.1, p.155. UDP is used in the diagram for simplicity.

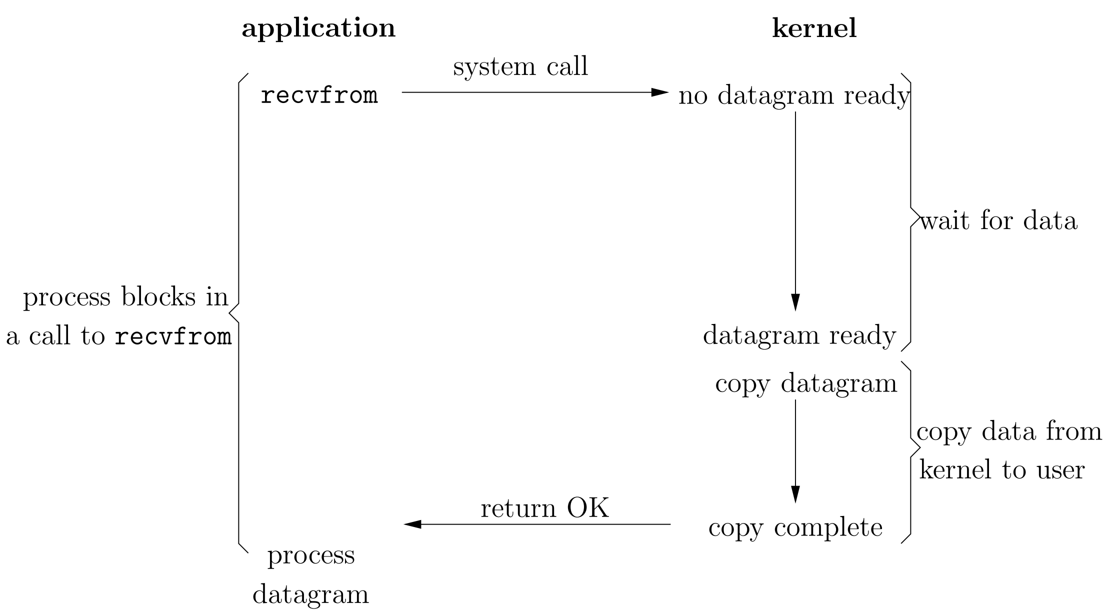

Figure 93: Blocking I/O model. (Fig.6.1,p.155)

c. Nonblocking I/O Model

   (a) A process will not be blocked even if the requested data is not ready at the socket descriptor. Instead, an error value (EWOULDBLOCK) will be returned to the process to inform the non-availability of the data;

   (b) Testing a socket descriptor to verify if data is ready or not is called *polling*;

   (c) The process can choose keep polling (busy waiting), or test for availability of data later on;

(d) Illustration diagram: Fig.6.2, p.156

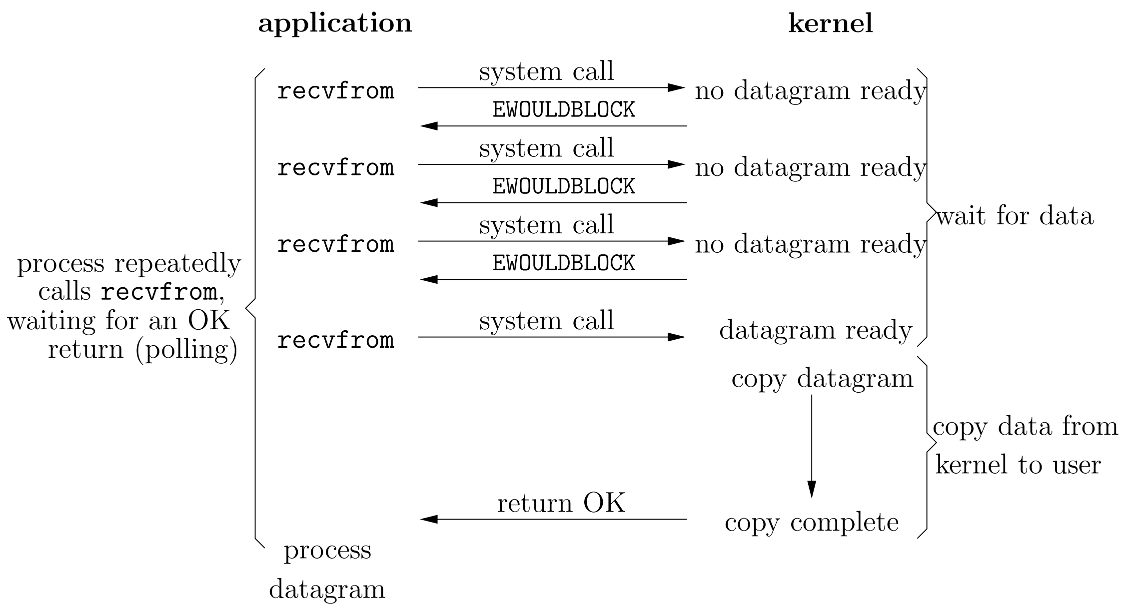

Figure 94: Nonblocking I/O model. (Fig.6.2,p.156)

d. I/O Multiplexing Model

(a) One of the two functions *select* (BSD) or *poll* (SVR4) is called. Both functions allow a process to specify a collection of descriptors to wait for I/O ready. A process can specify how long to wait. Both functions also provide options so that a process can poll, i.e. return immediately if none of the descriptors is ready;

(b) Illustration diagram: Fig.6.3, p.157

e. Signal driven I/O model

(a) A signal handler for SIGIO signal is first established (via the *signal* function or the *sigaction* function);

(b) The process will continue as usual after that;

(c) The kernel will interrupt the process (i.e. the process is blocked) by calling the signal handler when a SIGIO signal occurs;

(d) After the signal handler finishes, the original process will resume its execution;

(e) Illustration diagram: Fig.6.4, p.158

f. Asynchronous I/O Model

(a) A new I/O model introduced in 1993 in Posix.1;

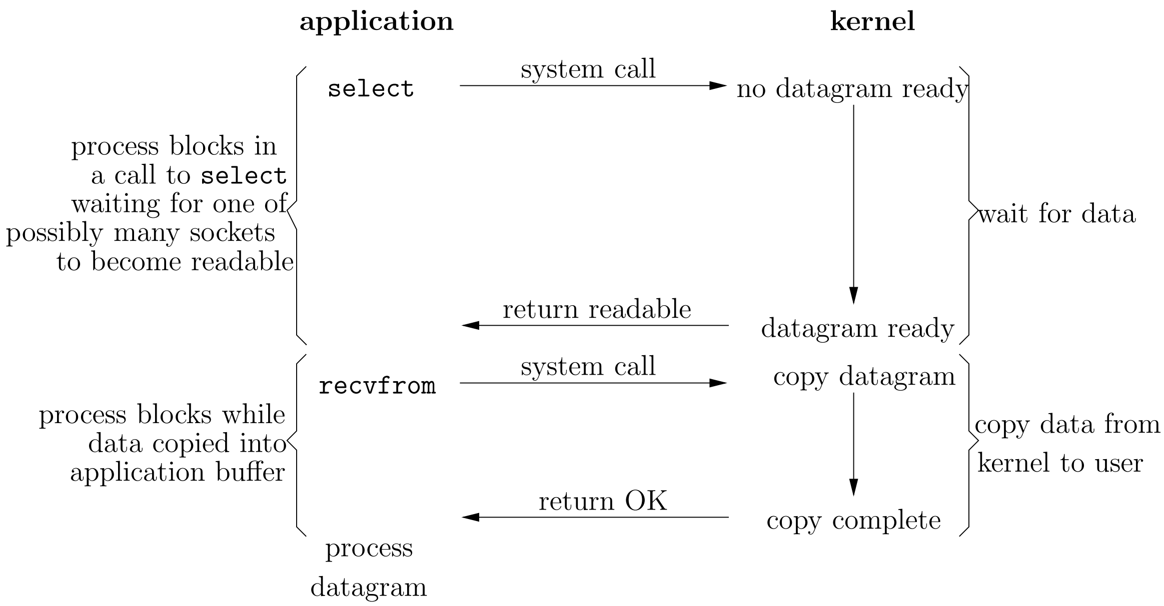

Figure 95: I/O multiplexing model. (Fig.6.3,p.157)

(b) A process must call the *aio_read* function to initiate asynchronous I/O. Namely, it informs the kernel to start I/O and to inform the calling process after I/O completes;

(c) Illustration diagram: Fig.6.5, p.159;

(d) Difference between signal driven I/O and asynchronous I/O: the former will notify us when an I/O can be initiated, the latter informs us that an I/O has completed.

g. Summary and Comparison of I/O Models: Fig.6.6, p.160.

h. Synchronous I/O vs Asynchronous I/O: defined by Posix.1

(a) A synchonous I/O operation causes the requesting process to be blocked until that I/O operation completes.

(b) An asynchonous I/O operation does not cause the requesting process to be blocked.

According to the above definition, the first four models are synchronous I/O models, while the last one is an asynchronous I/O model.

2. The `select` function

(1) Syntax and semantics

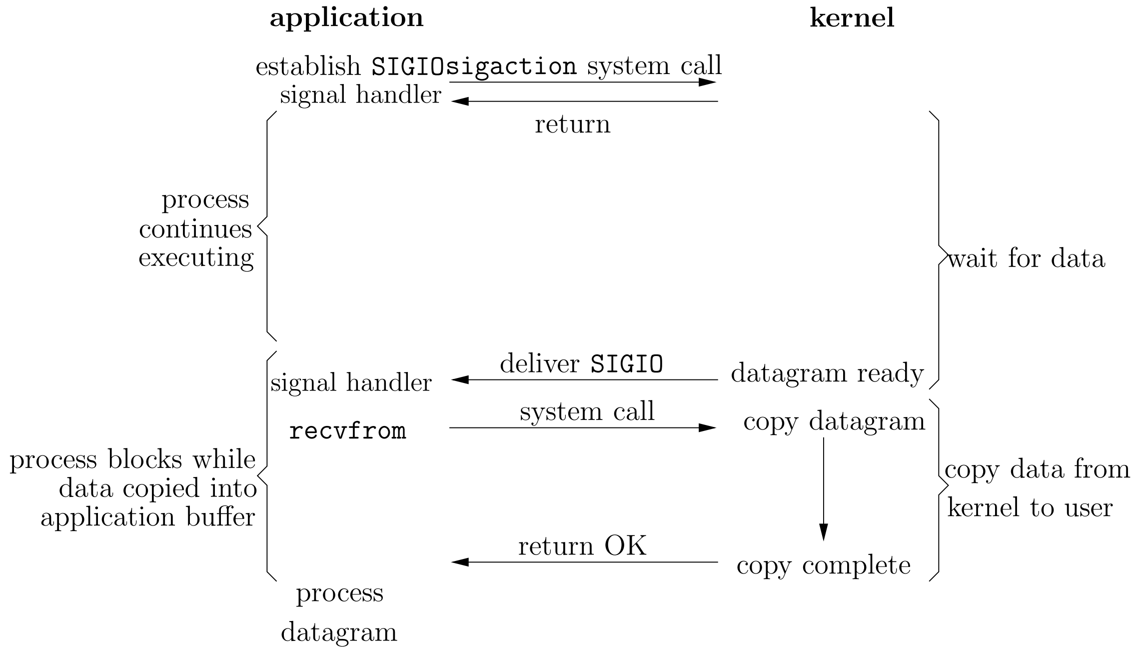

Figure 96: Signal Driven I/O model. (Fig.6.4,p.158)

a. This function allows a process to specify a collection of descriptors to be monitored for I/O or error events. A process can specify how long to wait or just poll (non-blocking);

b. Syntax and return values

(a) Syntax

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

The structure *timeval*, defined in <sys/time.h>:

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

(b) Return values

i. A postive integer which is the number of ready descriptors if there are descriptors ready before timeout reaches;

ii. 0 if timeout occurs;

iii. -1 on error.

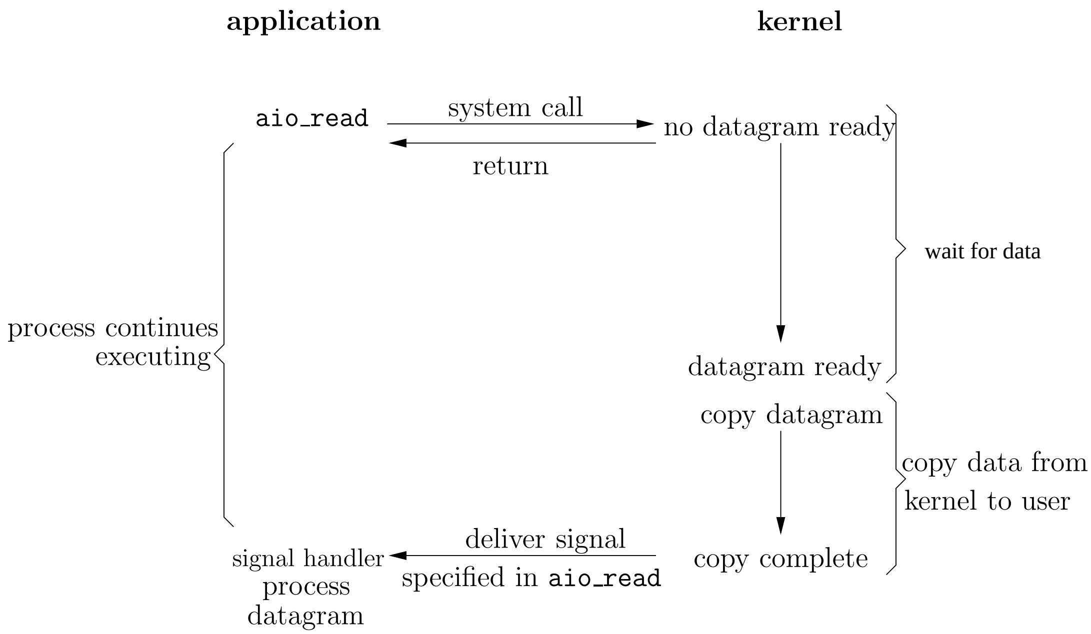

Figure 97: Asynchronous I/O model. (Fig.6.5,p.159)

c. Semantics: a call to *select* can tell if any of the file descriptors in the *readfds*, *writefds*, and *exceptfds* are ready for reading, writing, or having exception conditions raised, respectively.

(a) In addition there are three possible semantics, depending upon the last parameter *timeout*:

i. Wait indefinitely and return when one of the specified descriptors is ready for I/O. In this case *timeout* must be set to NULL.

ii. Return when one of the specified descriptors is ready for I/O within the specified *timeout* amount of time. In this case *timeout* points to a *timeval* structure with timer value nonzero;

iii. Poll: return immediately after checking the descriptors. In this case *timeout* points to a *timeval* structure with timer value set to 0;

In the first two cases, the wait will be interrupted if the process catches a signal and returns from a signal handler.

(b) The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum number of descriptors to be tested, plus 1. Descriptors 0, 1, 2, upto *maxdfp1* - 1 will be tested.

(c) The three middle parameters *readset, writeset*, and *exceptset* specify the set of descriptors that should be monitored for reading, writing, and error exceptions, respectively.

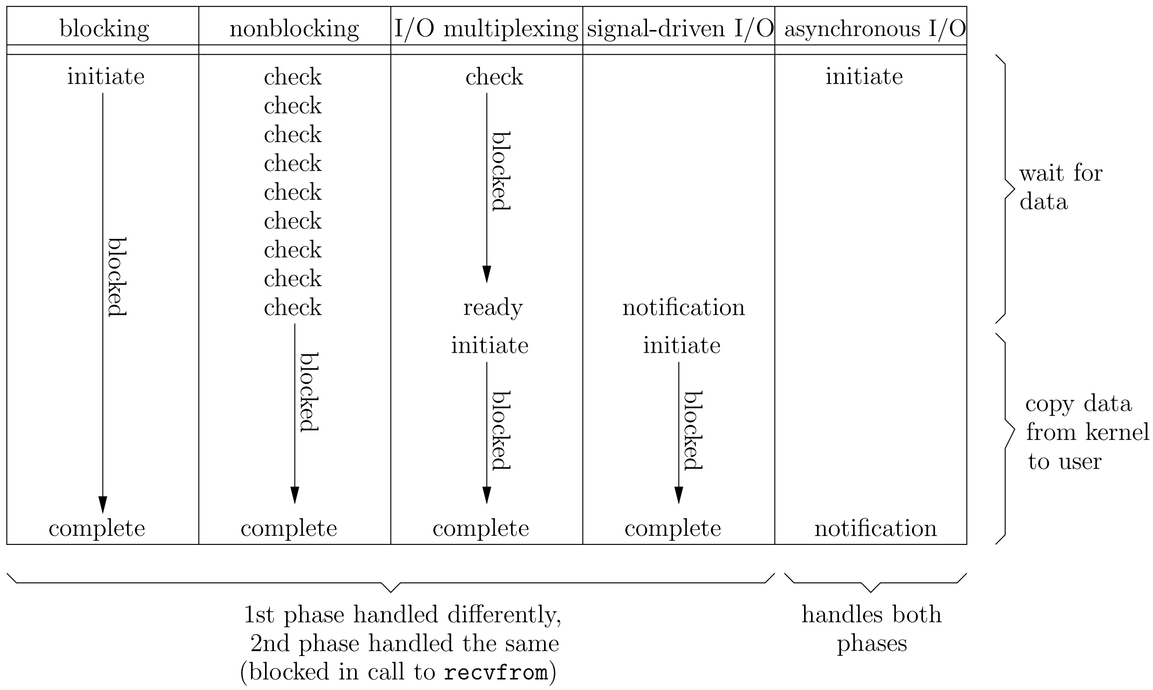

Figure 98: Comparison of the five I/O models. (Fig.6.6,p.160)

(d) Currently, only two exception conditions are supported:

   i. The arrival of out-of-band data for a socket (Chapter 21).

   ii. The presence of control status information to be read fom the master side of a pseudo-terminal that has been put into packet mode (not covered in this volume)

(e) Macros for testing staus of descriptors. Macros FD_XX and the data type *fd_set* are defined in <sys/types.h>:

FD_ZERO(fd_set **fdset*);	/* clear all bits in *fdset* */
FD_SET(int *fd*, fd_set **fdset*);	/* turn the bit for *fd* on in *fdset* */
FD_CLR(int *fd*, fd_set **fdset*);	/* turn the bit for *fd* off in *fdset* */
FD_ISSET(int *fd*, fd_set **fdset*);	/* test the bit for *fd* on in *fdset* */

(3) Specifying the descriptor sets:

a. Each descriptor is represented by a single bit in a vector (e.g. a 32-bit integer). Details of the implementation is hidden behind the FD_xxx macros.

b. **Example**: define a variable of type *fd_set* and then turn on the indicators for descriptors 1, 4, and 5:

```
fd_set fdvar;
FD_ZERO(&fdvar);    /* initialize the set (turn all bits off) */
```

FD_SET(1, &fdvar); /* turn on bit for fd 1 */
FD_SET(4, &fdvar); /* turn on bit for fd 4 */
FD_SET(5, &fdvar); /* turn on bit for fd 5 */

c. The maximum number of files that can be opened by a single process is limited by system. Usually less than 64.

d. The three *fds* arguments are all value-result parameters. An fd will be set by the system if the I/O is ready or exception is raised on that device. *FD_ISSET* macro can e used to test if an fd is set.

e. The *select* call can also be used to await a connection on a socket. A client wanting to making multiple connections can use the *select* call to test if a connection is ready.

(4) Implement an accurate timer: (the example on p.330 of the 1st ed. of the second textbook).

a. More accurate than the *sleep* function.

b. Notice that all three *fds* sets are set to NULL.

(5) Conditions under which a socket descriptor is ready (can be returned by select function as one of the ready descriptors)

a. Conditions for a ready read socket descriptor:

(a) The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. In this case, a read operation will not block and will return the actual number of bytes read. The low-water mark value can be changed by using *SO_RCVLOWAT* option;

(b) The read-half of the connection is closed (i.e. a FIN segment has be received in TCP). An zero (eof) will be returned;

(c) A listening socket has connection requests waiting in the completed queue. An *accept* call will normally not block;

(d) A socket error is pending. A -1 will be returned and the value of the variable *errno* indicates specific error conditions. The pending errors can be fetched and cleared by calling *getsockopt* function specifying the *SO_ERROR* socket option.

b. Conditions for a ready write socket descriptor:

(a) The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer and either (1) the socket is connected; or (2) the socket does not require a connection (UDP). In this case, a write operation will not block and will return the actual number of bytes written. The low-water mark is normally defaulted to 2048;

Note: ready to write means that the kernel allows an application's data to be copied into its corresponding send buffer. It does not mean that the data will be actually sent out yet.

(b) The write-half of the connection is closed. A write operation in this case will produce a SIGPIPE signal;

(c) A socket error is pending. A -1 will be returned immediately.

c. Conditions for an exception socket descriptor: arrival of an out-of-band data (discussed in Chapter 21).

d. Summary of socket descriptor conditions: Fig.6.7, p.166

Condition	readable?	writable	exception?
data to read	•		
read-half of the connection closed	•		
new connection ready for listing socket	•		
space available for writing		•	
write-half of the connection closed		•	
pending error	•	•	
TCP out-of-band data			•

Figure 6.7 Summary of conditions that cause a socket to be ready for `select`

(6) Maximum number of descriptors that can be used with `select` function

a. For many implementations, that value (the constant `FD_SETSIZE`) is defined in the file <sys/types.h>. A typical value is 256;

b. Other implementations allow a process to define their own `FD_SETSIZE` constant.

c. Note: the exact limit is implementation dependent. Just increasing the value of the constant `FD_SETSIZE` may not always increase the actual size of the descriptor sets.

3. Revised *str_cli* function

(1) The *select* function is used by the *str_cli* function to monitor two descriptors: the file descriptor stdin and the socket descriptor

(2) Conditions handled by the `select` function: Fig.6.8, p.167

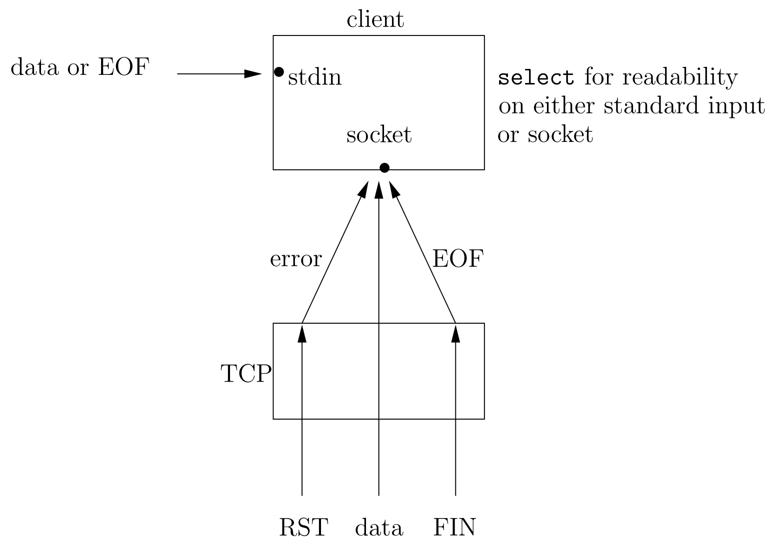

Figure 99: Conditions handled by `select` in `str_cli`. (Fig.6.8,p.167)

a. For the file descriptor stdin, two conditions: data or EOF;

b. For the socket descriptors, three conditions: data from the peer, FIN from the peer, and RST from the peer.

(3) The revised function: Fig.6.9, p.168

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

Figure 6.9 Implementation of `str_cli` function using `select`.

4. Batch Input: sending requests continuously from client to server

   (1) Measuring the RTT (round-trip-time) between client and server: Using the *ping* command to estimate RTT. Taking the average of 30 measurements produces 175 ms;

   (2) Estimating the total time required to send 2000 input lines

      a. Total number bytes: 98,349. Total number of lines: 2,000. Average number of bytes per line: 49;

      b. TCP and IP headers each have 20 bytes. So the average size of a TCP segment is 89 bytes. This is approximately equal to the size of a ping packet;

      c. Therefore sending the 2,000 line will take an estimated $2000 * 0.175 = 350$ sec. The actual time is 354 seconds, very close to the estimates.

   (3) Time line of the stop-and-wait mode client-server communications: Fig.6.10, p.170

   (4) Problems of stop-and-wait mode: wastes of bandwidth.

   (5) Filling the pipes - batch mode input: Fig.6.11, p.171.

      a. Problems with the current echo client-server implementations: total number of output is always less than the total number of input in batch mode;

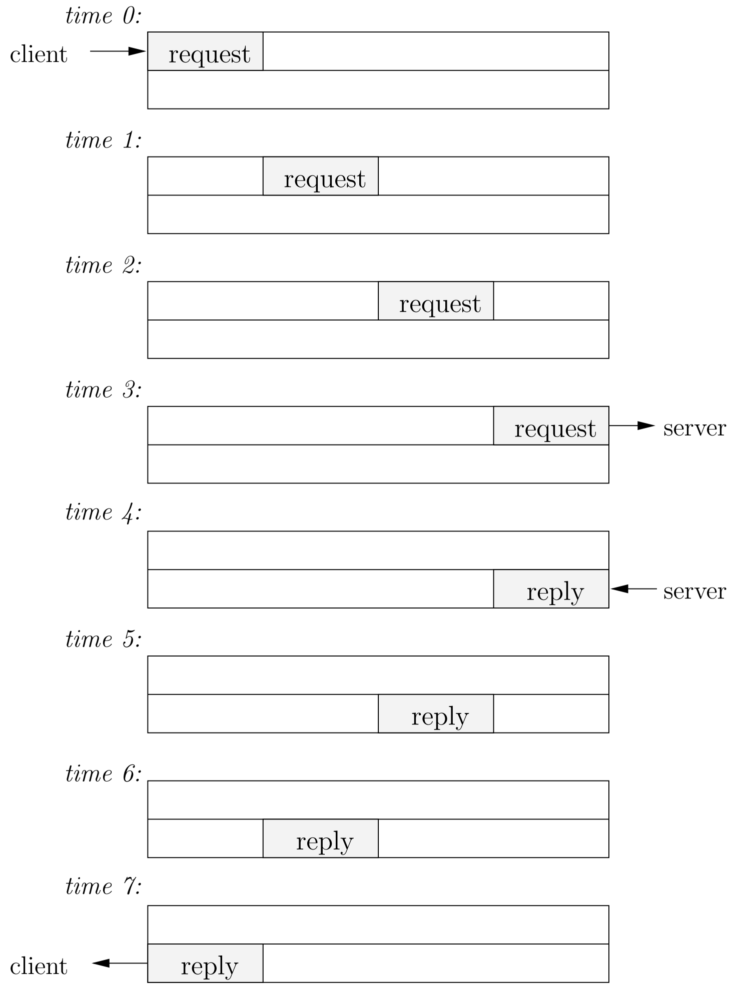

Figure 100: Time line of stop-and-wait mode: interactive input. (Fig.6.10,p.170)

b. Cause of the problem: the client terminates as soon as EOF is reached at the file descriptor stdin. It does not wait for replies for those requests that haven't been acknowledged by the server yet!

c. Solution: when EOF is reached on stdin, we have to only close one-half of the connection: we no longer send new requests to the server. However, we should still be able to receive requests. But the *close* function will not allow us to do this.

5. `shutdown` function

(1) Limitations of *close* function

a. It decrements the reference counter of a socket descriptor and closes the socket only when the counter reaches zero. The *shutdown* function can close a connection

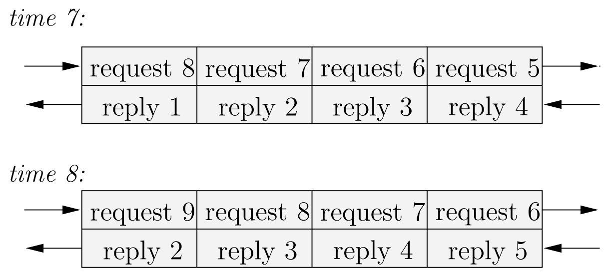

Figure 101: Filling the pipe between the client and server: batch mode. (Fig.6.11,p.171)

regardless the reference counter;

b. *close* terminates both directions of data transmissions. *shutdown* allows a process have more control such as closing one direction of data transfer.

(2) Illustration of use of *shutdown* function: Fig.6.12, p.172

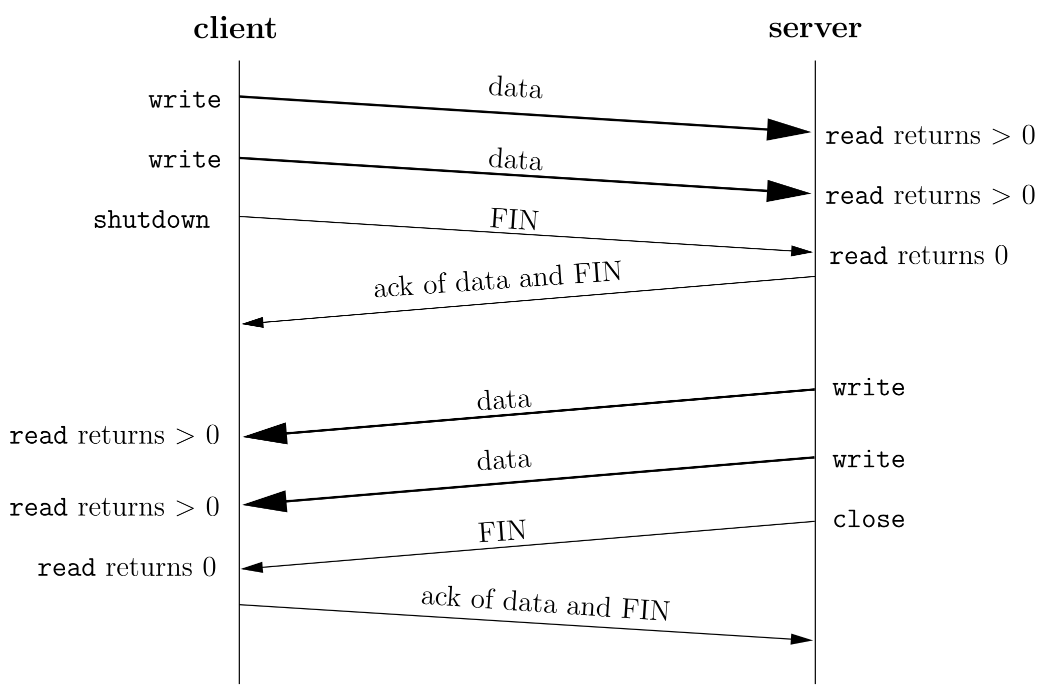

Figure 102: Calling `shutdown` to close half of a TCP connection. (Fig.6.12,p.172)

(3) Syntax of *shutdown* function.

#include <sys/socket.h>

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

The *howto* argument:

a. SHUT_RD: (value 0) no more data can be received on the socket;

b. SHUT_WR: (value 1) no more output will be allowed on the socket;

c. SHUT_RDWR: (value 2) both send and received are disallowd.

6. Re-revised client-server example implementation

(1) Re-revised *str_cli* function

a. The revised function in Fig.6.9, p.168 is revised again using the shutdown function to fix the problem of batch output;

b. The new version: Fig.6.13, p.174

```
#include   "unp.h"

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

    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)) {   /* socket is readable */
           if (Readline(sockfd, recvline, MAXLINE) == 0) {
              if (stdineof == 1)
                 return;      /* normal termination */
              else
                 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) {
```

```
                stdineof = 1;
                Shutdown(sockfd, SHUT_WR);   /* send FIN */
                FD_CLR(fileno(fp), &rset);
                continue;
            }
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

Figure 6.13 `str_cli` function using `select` that handles end-of-file correctly.

(2) Revised TCP echo server

a. The new version of the server is an iterative server. It will use *select* function, instead of the *fork* function to create child processes.

b. Main idea: The server will monitor, in a loop, a listening socket and a set of client socket descriptors. All these descriptors will be placed in a read set that will be monitored by calling `select` function. Each client connection will result in an active read client descriptor placed in the read set and each client termination request will remove the corresponding entry from the set of active read descriptors.

c. Main data structures used

   (a) An array *client*[] that contains a connected socket descriptor for each client. All elements are initialized to -1 to indicated that it is unused;

   (b) Only a read set is used. Write set and except set are both set to NULL;

   (c) The listening socket is always in the variable *rset*. The variable *allset* and *rset* are initiated to be the same at the beginning of each iteration of the for-loop. The variable *allset* will be adjusted everytime a client terminates a connection.

d. Illustration of main ideas: Fig.6.14, Fig.6.15 (p.175), Fig.16, Fig.6.17,Fig.6.18 (p.176), Fig.6.19 and Fig.6.20 (p.177).

e. The source code: Fig.6.21, 6.22, p.178 and p.179

```
#include "unp.h"

int
main(int argc, char **argv)
{
    int             i, maxi, maxfd, listenfd, connfd, sockfd;
    int             nready, client[FD_SETSIZE];
```

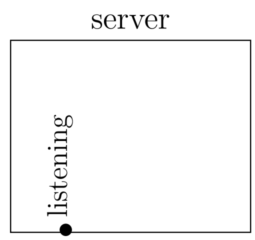

Figure 103: TCP server before first client has established connection. (Fig.6.14,p.175)

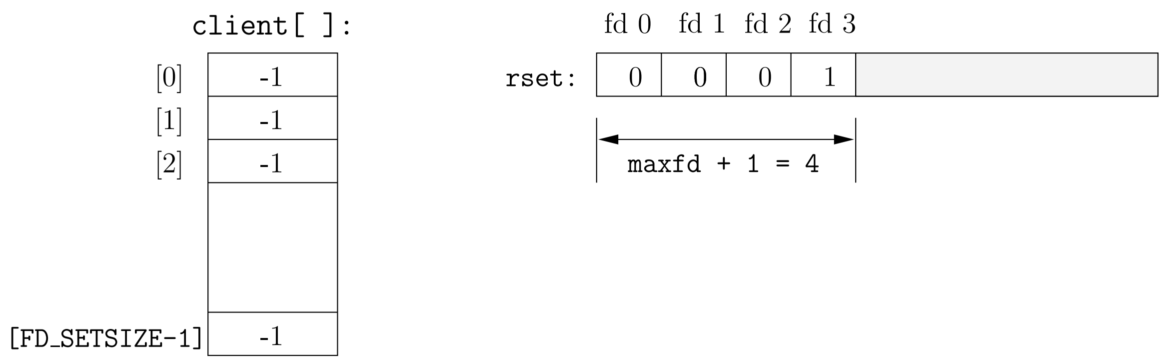

Figure 104: Data structures for TCP server with just listening socket. (Fig.6.15,p.175)

```
    ssize_t             n;
    fd_set              rset, allset;
    char                line[MAXLINE];
    socklen_t           clilen;
    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);

    maxfd = listenfd;           /* initialize */
    maxi = -1;                  /* index into client[] array */
    for (i = 0; i < FD_SETSIZE; i++)
        client[i] = -1;         /* -1 indicates available entry */
    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);
```

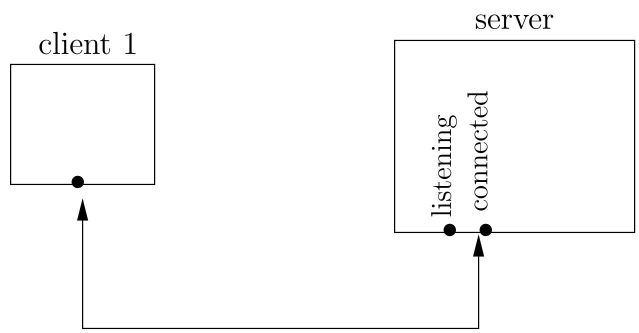

Figure 105: TCP server after first client establishes connection. (Fig.6.16,p.176)

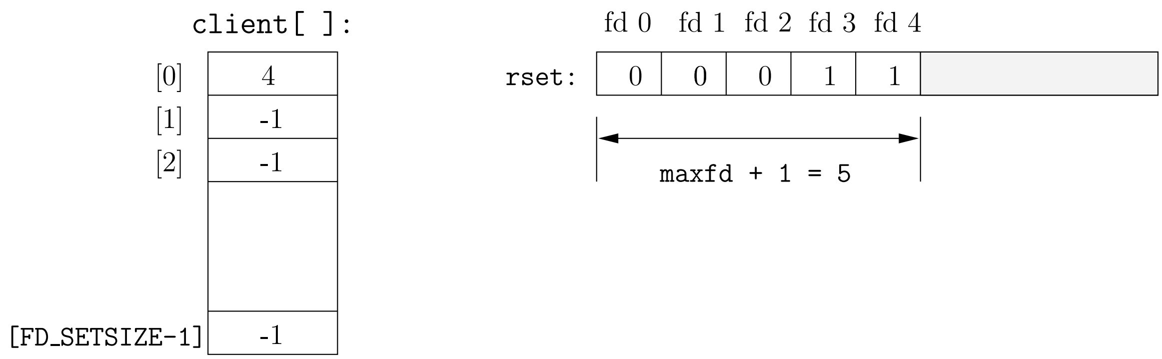

Figure 106: Data structures after first client connection is established. (Fig.6.17,p.176)

Figure 6.21 TCP server using a single process and `select`: initialization.

```
    for ( ; ; ) {
        rset = allset;      /* structure assignment */
        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &rset)) {   /* new client connection */
           clilen = sizeof(cliaddr);
           connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

           for (i = 0; i < FD_SETSIZE; i++)
               if (client[i] < 0) {
                  client[i] = connfd;   /* save descriptor */
                  break;
               }
           if (i == FD_SETSIZE)
              err_quit("too many clients");

           FD_SET(connfd, &allset);   /* add new descriptor to set */
           if (connfd > maxfd)
              maxfd = connfd;         /* for select */
```

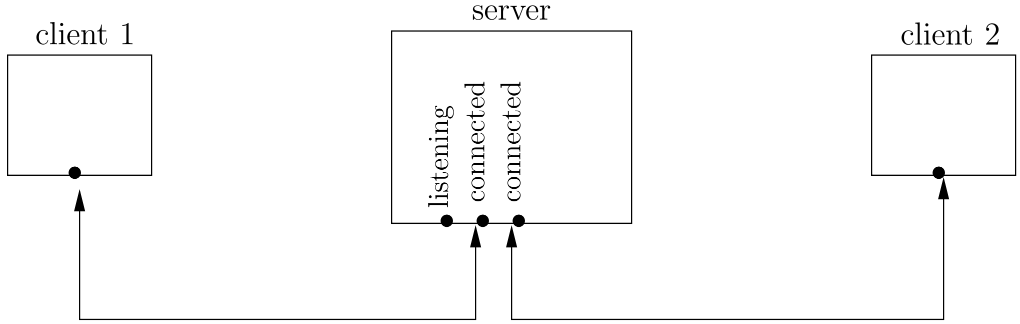

Figure 107: TCP server after second client connection is established. (Fig.6.18,p.176)

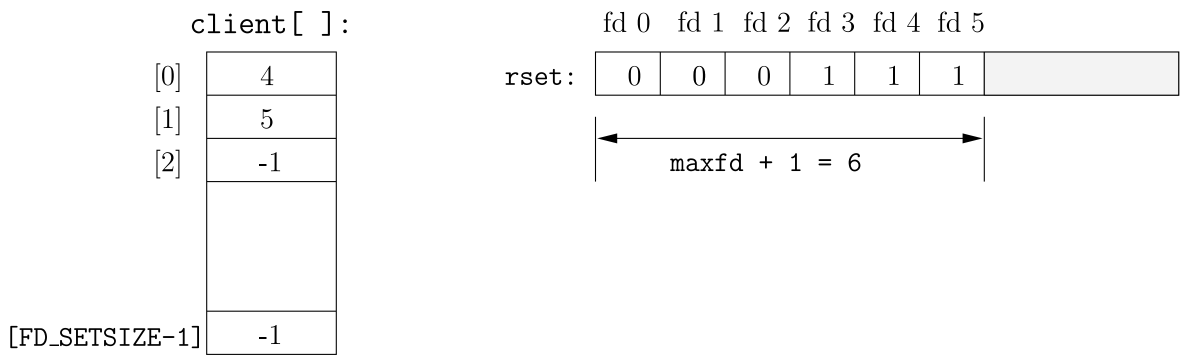

Figure 108: Data structures after second client connection is established. (Fig.6.19,p.177)

```
            if (i > maxi)
               maxi = i;            /* max index in client[] array */

            if (--nready <= 0)
               continue;            /* no more readable descriptors */
         }
         for (i = 0; i <= maxi; i++) {   /* check all clients for data */
             if ( (sockfd = client[i]) < 0)
                continue;
             if (FD_ISSET(sockfd, &rset)) {
                if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
                         /*4connection closed by client */
                    Close(sockfd);
                    FD_CLR(sockfd, &allset);
                    client[i] = -1;
                } else
                    Writen(sockfd, line, n);

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

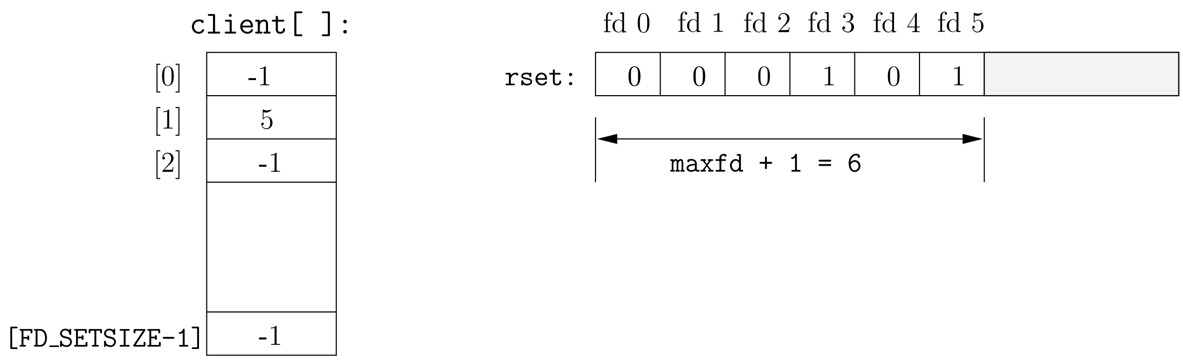

Figure 109: Data structures after firstond client terminates its connection. (Fig.6.20,p.177)

```
        }
    }
}
```

Figure 6.22 TCP server using a single process and `select`: loop.

f. Denial of service attacks:

(a) Definition of *denial of server attack*: a server that handles multiple clients is blocked in a function call while servicing a single client. Hence services to other clients are be *denied* while the server is being blocked.

(b) Recall: the function `readline` (two versions in Chapter 3) reads a line of characters. It breaks out of the `for` loop after reading a newline char '\n'. A client for this new version of the server can just send one byte (other than the newline char), then goes in sleep. The server will be blocked at the readline function, waiting for the rest of the line.

(c) Solutions of denial of service attack: (a) Non-blocking I/O; (2) Separate threads or processes; (3) time-out.

7. *pselect* function

(1) Syntax

```
#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);
```

(2) Differences between *select* and *pselect*:

a. *pselect* uses the *timespec* structure, which can be more accurate than than the *timeval* structure (specified in <time.h>):

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

b. *pselect* adds one more argument. The sixth argument allows a process to disable the delivery of certain signals.

8. `poll` function

(1) Syntax

a. This is a SVR4 function and is very similar to the *select* function. It allows a process to specify a collection of descriptors to be monitored for I/O or error events. A process can specify how long to wait or just poll (non-blocking);

b. Syntax and return values

(a) Syntax

```
#include <poll.h>
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
```

The structure *pollfd*, defined in <poll.h>:

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

The second argument specifies the number of array elements in the first argument. The last parameter is the amount of timeout in units of milliseconds.

(b) Return values (similar to *select*):

i. A postive integer which is the number of ready descriptors if there are descriptors ready before timeout reaches;

ii. 0 if timeout occurs;

iii. -1 on error.

c. Input events and occured events: Fig.6.23, p.183

(a) Four input events;

(b) Three output events;

(c) Three error events. They cannot be set in the `events` field of the structure but are always returned in `revents` when the corresponding conditions exist.

Constant	Input to *events?*	Result from *revents?*	Description
`POLLIN`	•	•	normal or priority band data can be read
`POLLRDNORM`	•	•	normal data can be read
`POLLRDBAND`	•	•	priority band data can be read
`POLLPRI`	•	•	high-priority data can be read
`POLLOUT`	•	•	normal or priority band data can be written
`POLLWRNORM`	•	•	normal data can be written
`POLLWRBAND`	•	•	priority band data can be written
`POLLERR`		•	an error has occurred
`POLLHUP`		•	hangup has occurred
`POLLNVAL`		•	descriptor is not an open file

Figure 6.23 Input *events* and returned *revents* for `poll`

d. Classes of data: three classes of data identified by `poll`:

(a) *normal*

(b) *priority band*

(c) *high priority*

This classification come from the stream-based implementations (Cf. Fig.33.5).

(2) Semantics: a call to `poll` can tell if any of the file descriptors in the *fdarray* are ready for reading, writing, or having exception conditions, depending upon the nature of event that actually occurred. Like the `select` function, there are three possible semantics, depending upon the last parameter *timeout* (Fig.6.24):

a. Wait indefinitely and return when one of the specified descriptors is ready for I/O. In this case *timeout* argument must be set to the INFTIM constant, which is defined to be a negative integer.

b. *timeout* > 0. Return when one of the specified descriptors is ready for I/O within the specified *timeout* amount of time (in the unit of millisecond).

c. *timeout* $= 0$. This is a polling. Return immediately after checking the descriptors specified in the first argument.

In the first two cases, the wait will be interrupted if the process catches a signal and returns from a signal handler.

timeout value	Description
`INFTIM`	wait forever
0	return immediately, do not block
> 0	wait specified number of milliseconds

Figure 6.24 *timeout* values for `poll`

(3) Notes:

a. Compared with `select` function, `poll` does not need macros such as `FD_SETSIZE`. An application must explicit declare an array of type `pollfd` and allocate memory for them if necessary. The data structure is explicit (unlike the `fd_set` type).

b. If an application is not interested in a particular descriptor, it can just set the `fd` member to -1. Then the function will ignore `events` member and `revents` is set to 0 on return.

(4) Revised TCP echo server using the `poll` function: Fig.6.25, p.186, Fig.6.26, p.187.