I/O Multiplexing

Giảng viên: TS. Trần Hải Anh
Bộ môn Truyền Thông & Mạng máy tính
Khoa CNTT & TT

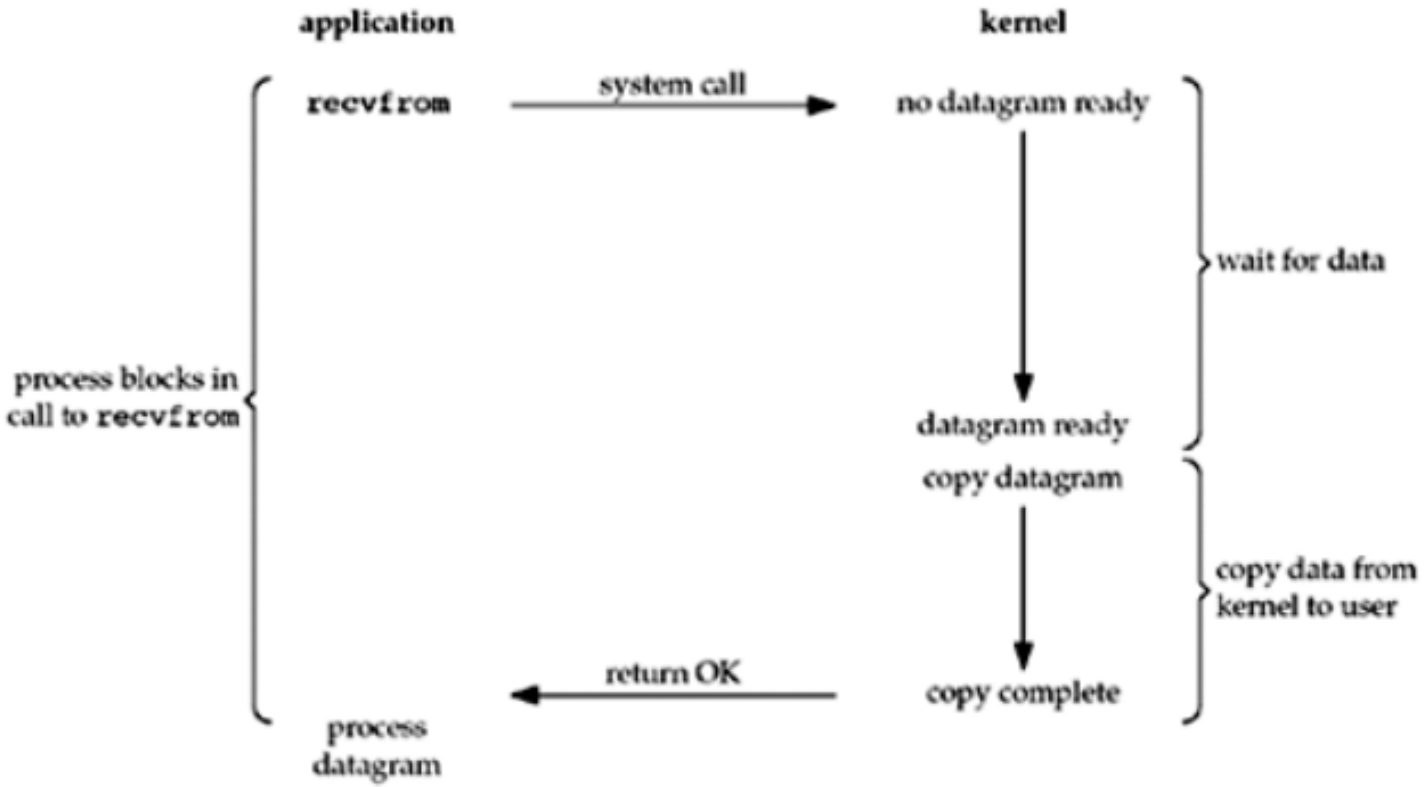
Intro

- The client has to handle two inputs at the same time: **standard input** and a **TCP socket.**
- Problem: when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input => it never saw that.
- Need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready
- **I/O Multiplexing** (**select** & **poll** functions)

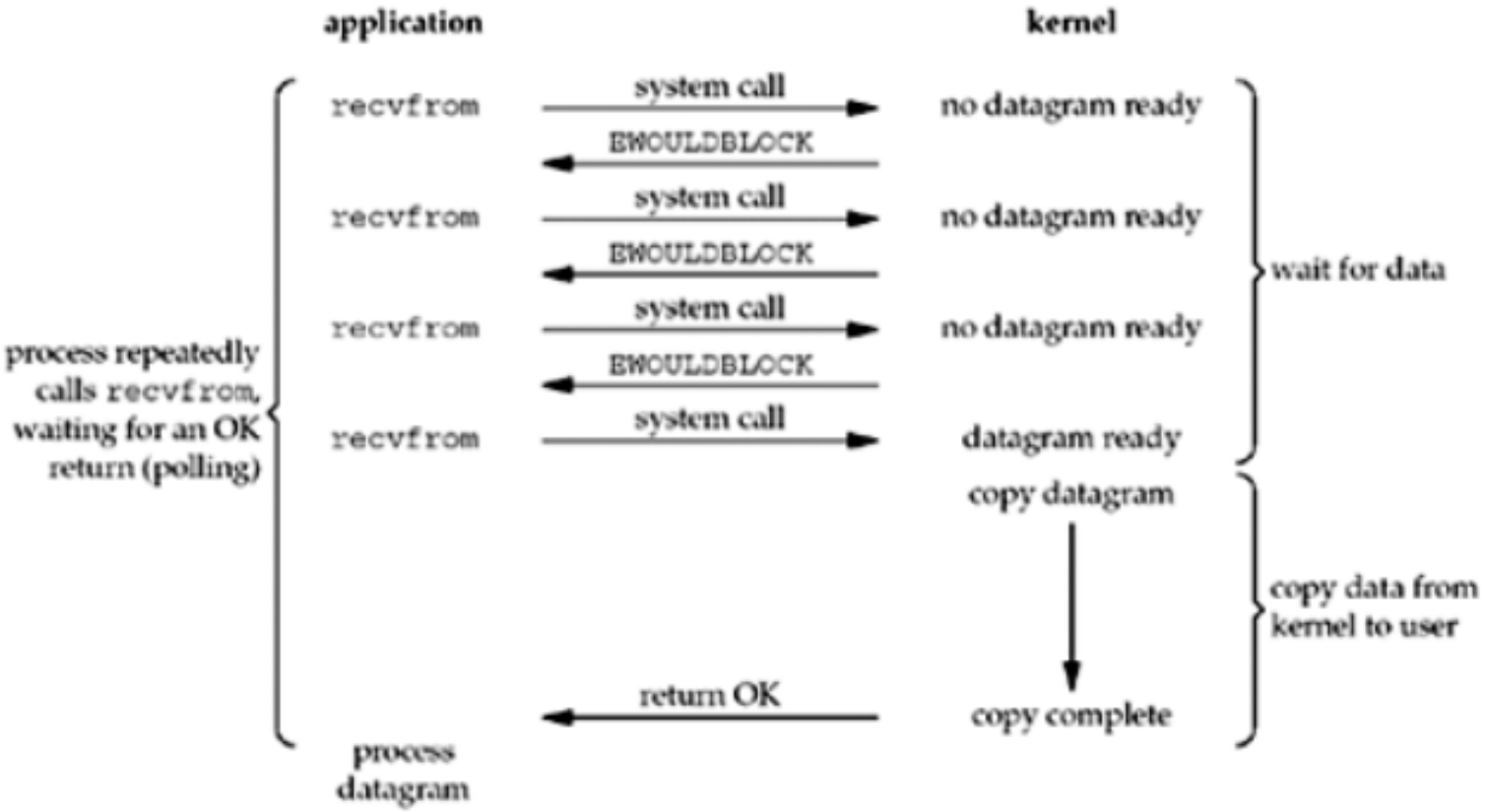
I/O Models

- blocking I/O
- nonblocking I/O
- I/O multiplexing (*select* and *poll*)
- signal driven I/O (*SIGIO*)
- asynchronous I/O (the POSIX *aio*_functions)

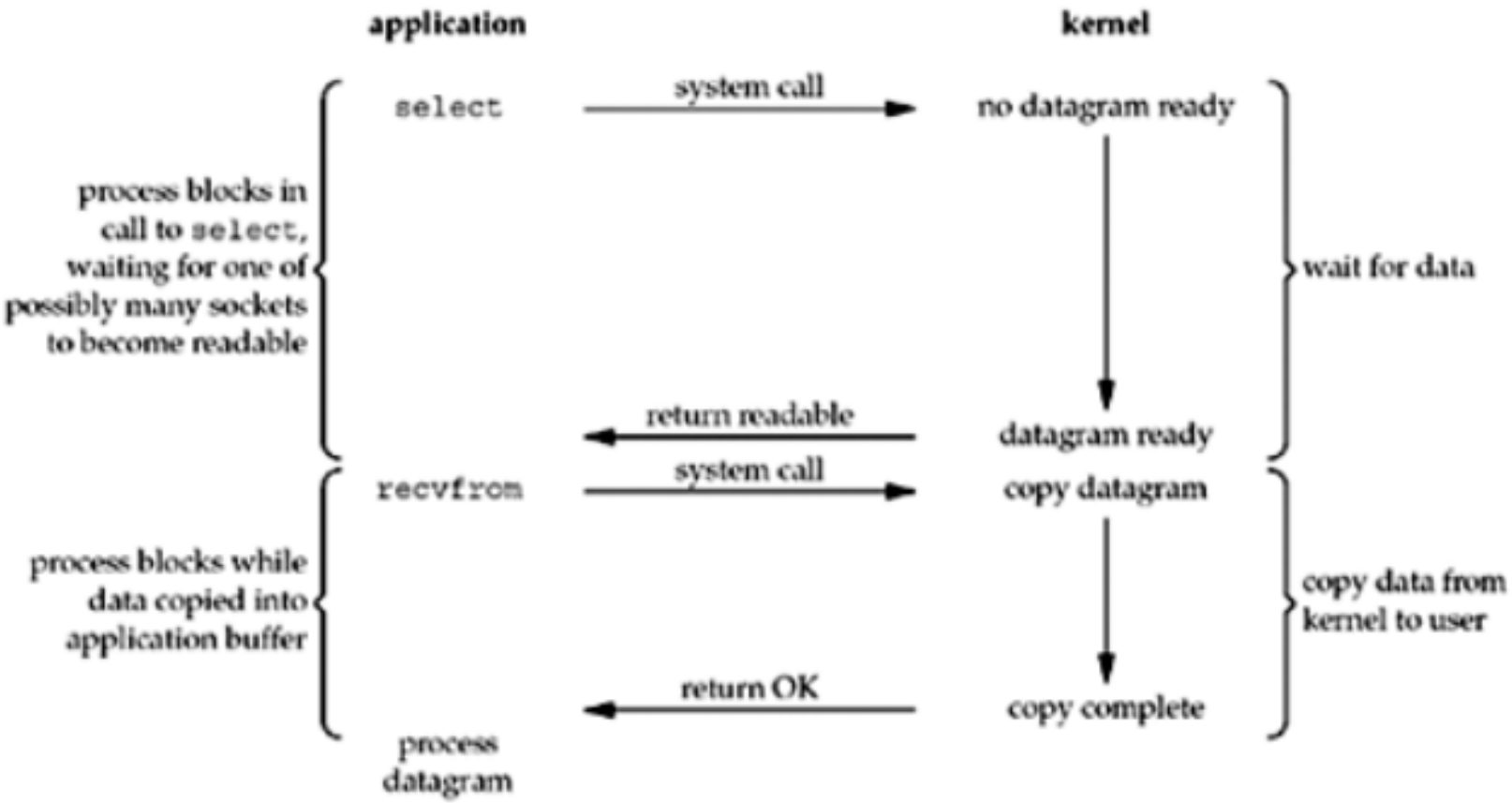
Blocking I/O Model



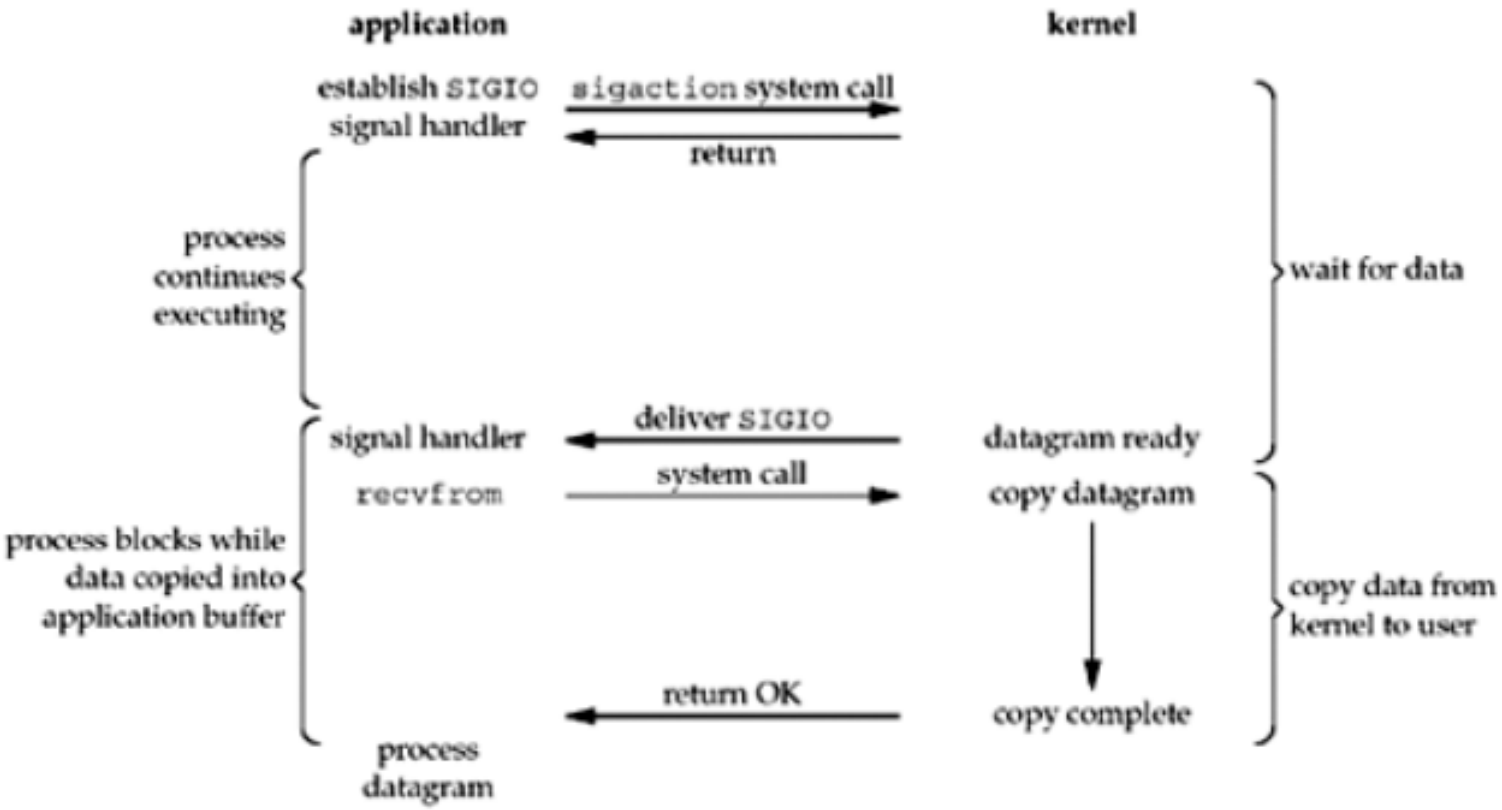
Nonblocking I/O Model



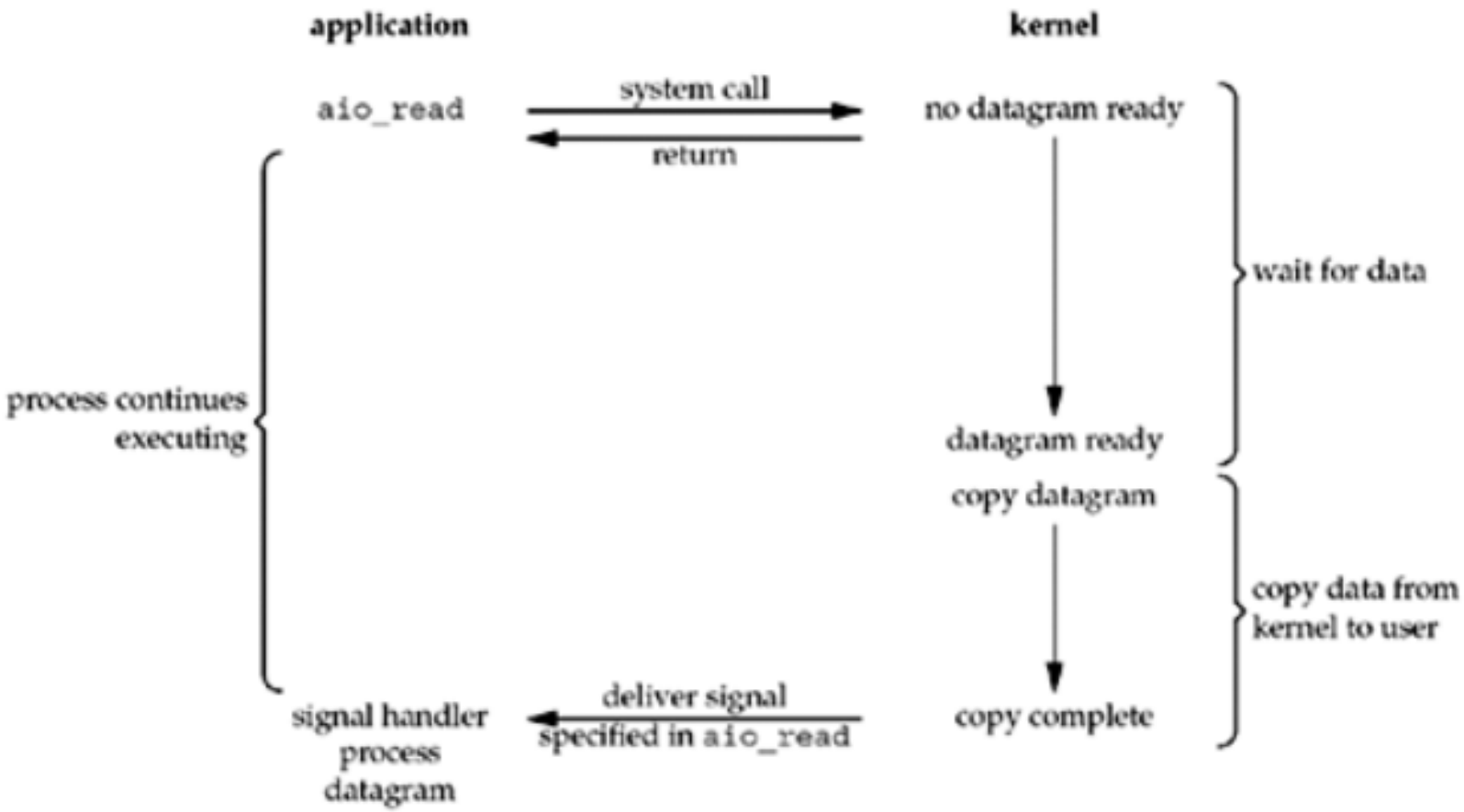
I/O Multiplexing Model



Signal-Driven I/O Model



Asynchronous I/O Model



select function

- This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.
- Example:
  - 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 exception condition pending
  - 10.2 seconds have elapsed

select function

```
#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);
```

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

Arguments of *select* function

- *const struct timeval *timeout* argument
- Structure:

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

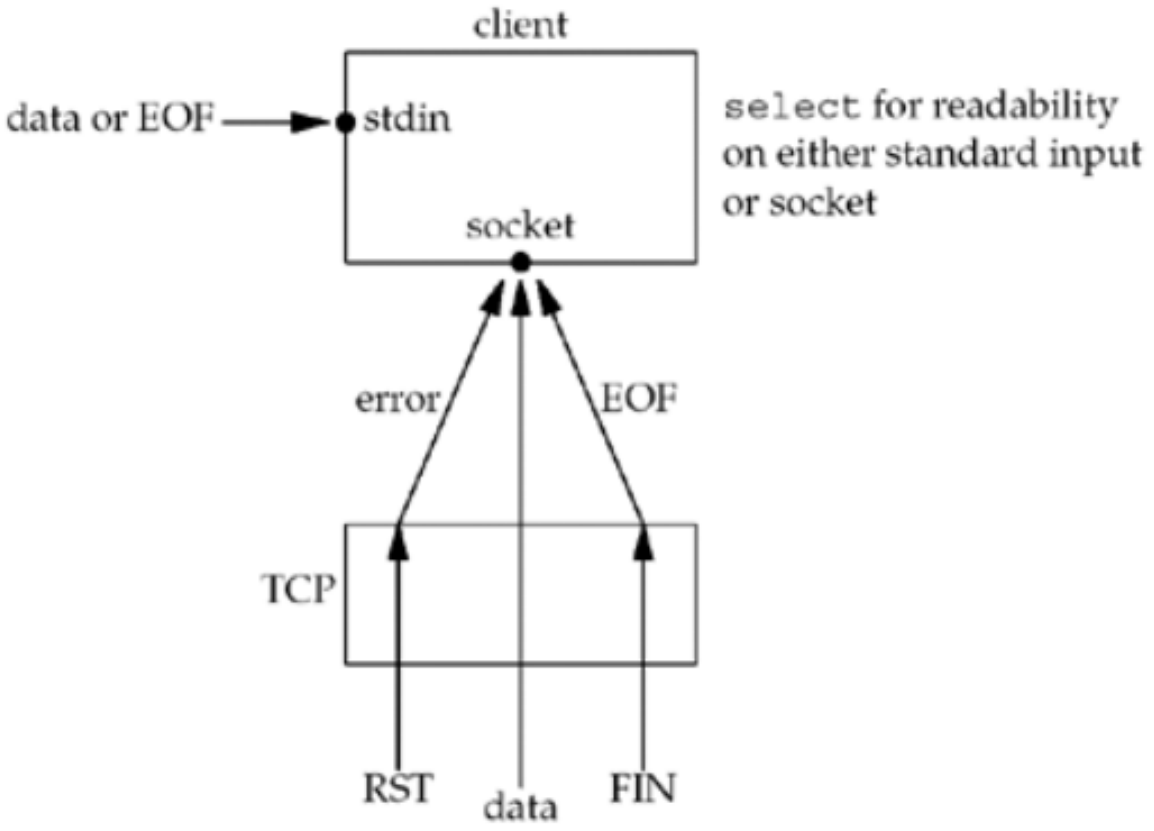
- 3 possibilities:
  - Wait forever => null pointer.
  - Wait up to a fixed amount of time
  - Do not wait at all: This is called polling. => assign to 0

Arguments of *select* function

- The three middle arguments, *readset*, *writeset*, and *exceptset*, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions.

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

Re-work with *str_cli* function



Re-work with *str_cli* function

- If the peer TCP sends data, the socket becomes readable and read returns greater than 0 (i.e., the number of bytes of data).
- If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and read returns 0 (EOF).
- If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable, read returns –1, and errno contains the specific error code.

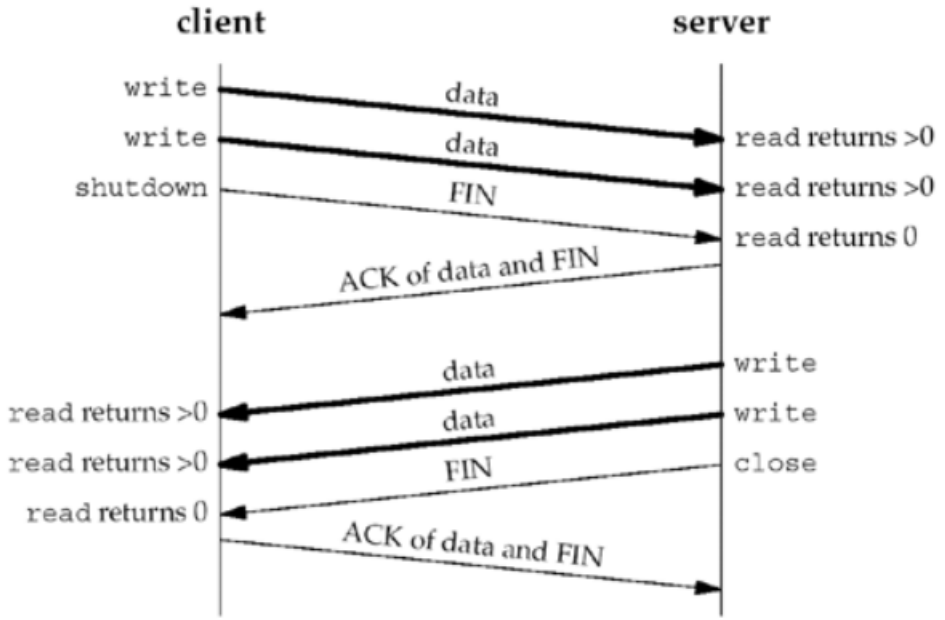
Re-work with *str_cli* function

select/strcliselect01.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1;
 6     fd_set  rset;
 7     char    sendline[MAXLINE], recvline[MAXLINE];

 8     FD_ZERO(&rset);
 9     for ( ; ; )  {
10         FD_SET(fileno(fp), &rset);
11         FD_SET(sockfd, &rset);
12         maxfdp1 = max(fileno(fp), sockfd)  +  1;
13         Select(maxfdp1,  &rset,  NULL,  NULL,  NULL);

14         if (FD_ISSET(sockfd,  &rset))  {  /* socket is readable */
15             if (Readline(sockfd, recvline, MAXLINE) == 0)
16                 err_quit("str_cli: server terminated prematurely");
17             Fputs(recvline, stdout);
18         }

19         if (FD_ISSET(fileno(fp), &rset))  {  /*  input is readable */
20             if (Fgets(sendline, MAXLINE, fp) == NULL)
21                 return;          /* all done */
22             Writen(sockfd, sendline, strlen(sendline));
23         }
24     }
25 }
```

shutdown Function

- two limitations with close function:
  - close decrements the descriptor's reference count and closes the socket only if the count reaches 0.
  - close terminates both directions of data transfer, reading and writing, even though the other end might have more data to send us.



shutdown Function

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

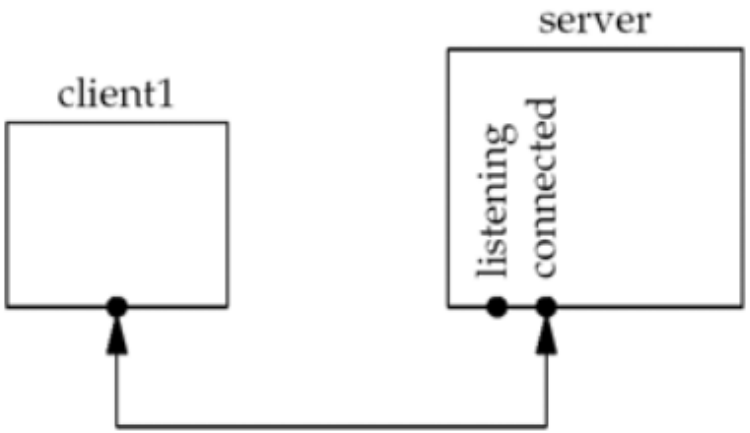
- Returns: 0 if OK, –1 on error
- howto argument
  - *SHUT_RD*: The read half of the connection is closed— No more data can be received on the socket and any data currently in the socket receive buffer is discarded.
  - *SHUT_WR*: The write half of the connection is closed. Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence.
  - *SHUT_RDWR*: The read half and the write half of the connection are both closed.

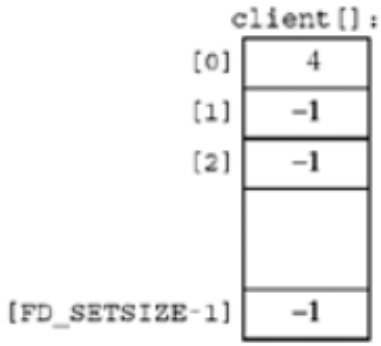
Re-work with str_cli function

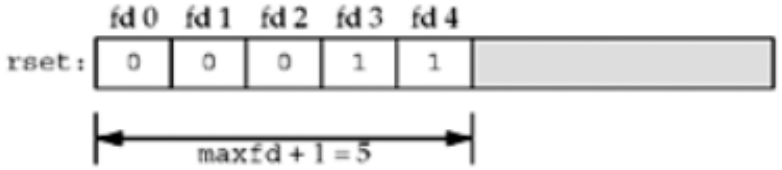
select/strcliselect02.c

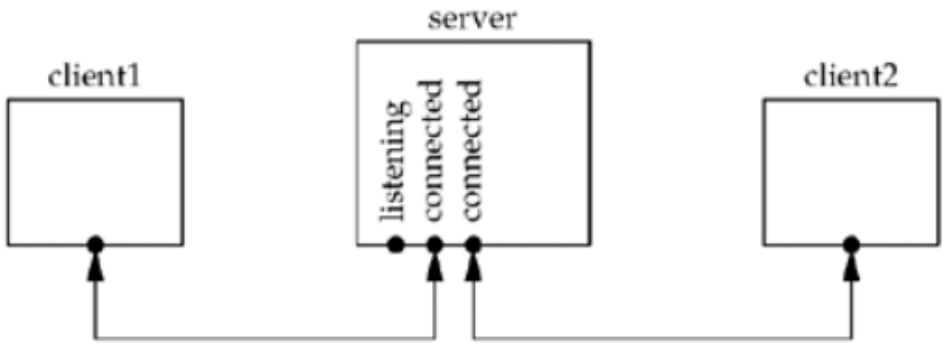
```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1, stdineof;
 6     fd_set  rset;
 7     char    buf[MAXLINE];
 8     int     n;

 9     stdineof = 0;
10     FD_ZERO(&rset);
11     for ( ; ; ) {
12         if (stdineof == 0)
13             FD_SET(fileno(fp), &rset);
14         FD_SET(sockfd, &rset);
15         maxfdp1 = max(fileno(fp), sockfd) + 1;
16         Select(maxfdp1, &rset, NULL, NULL, NULL);

17         if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
18             if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
19                 if (stdineof == 1)
20                     return;       /* normal termination */
21                 else
22                     err_quit("str_cli: server terminated prematurely");
23                 }
24                 Write(fileno(stdout), buf, n);
25         }
26         if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
27             if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28                 stdineof = 1;
29                 Shutdown(sockfd, SHUT_WR);  /* send FIN */
30                 FD_CLR(fileno(fp), &rset);
31                 continue;
32             }
33             Writen(sockfd, buf, n);
34         }
35     }
36 }
```

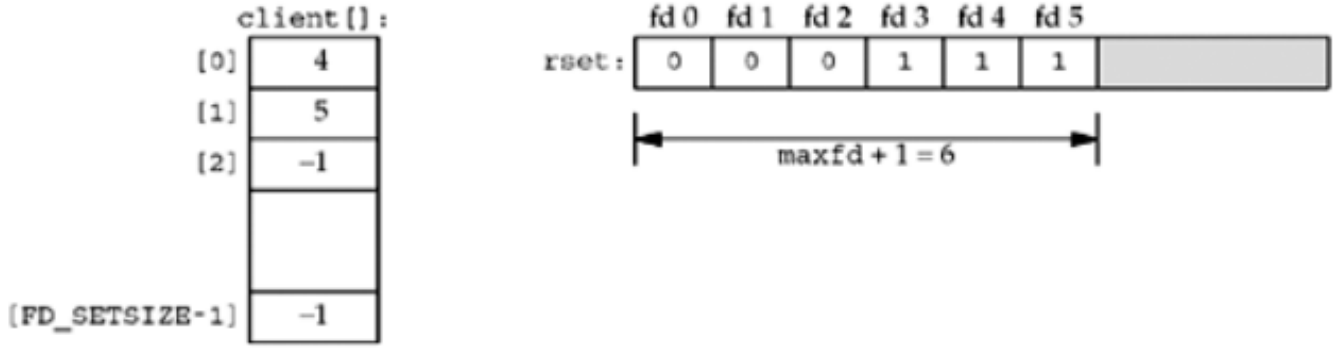
Re-write TCP server

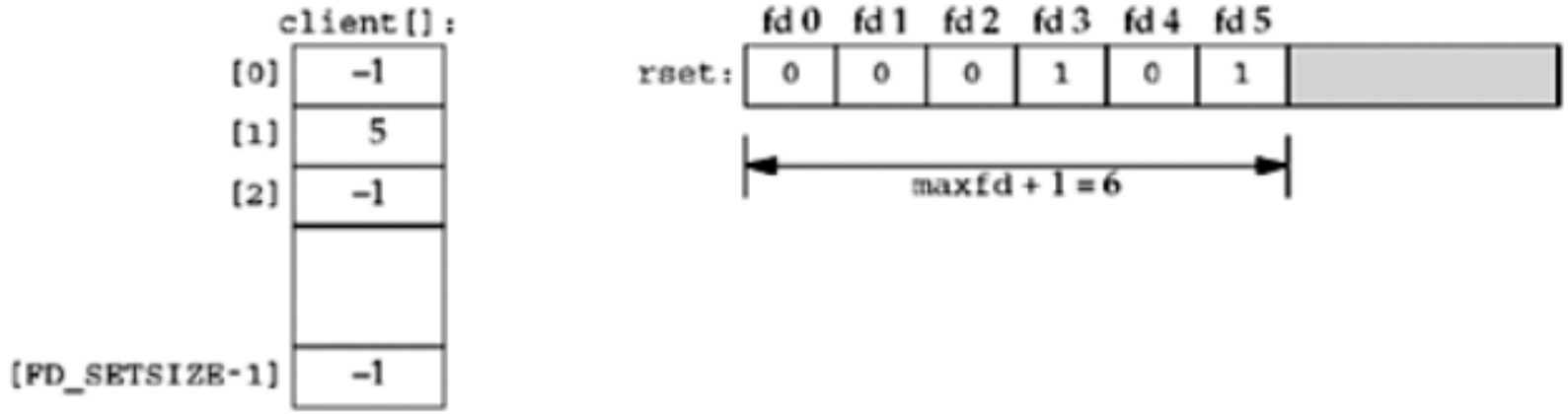

server
client1
listening
connected


client[]:
[0]
4
[1]
-1
[2]
-1
[FD_SETSIZE-1]
-1


fd 0
fd 1
fd 2
fd 3
fd 4
rset:
0
0
0
1
1
maxfd + 1 = 5


server
client1
client2
listening
connected
connected


client[]:
[0]
4
[1]
5
[2]
-1
[FD_SETSIZE-1]
-1
fd 0
fd 1
fd 2
fd 3
fd 4
fd 5
rset:
0
0
0
1
1
1
maxfd + 1 = 6



Data structures after first client terminates its connection.

tcpcliserv/tcpservselect01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, maxi, maxfd, listenfd, connfd, sockfd;
 6     int     nready, client[FD_SETSIZE];
 7     ssize_t n;
 8     fd_set  rset, allset;
 9     char    buf[MAXLINE];
10     socklen_t  clilen;
11     struct sockaddr_in cliaddr, servaddr;

12     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

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

18     Listen(listenfd, LISTENQ);

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

tcpcliserv/tcpservselect01.c

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

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

31              for (i = 0; i < FD_SETSIZE; i++)
32                  if (client[i] < 0) {
33                      client[i] = connfd; /* save descriptor */
34                      break;
35                  }
36              if (i == FD_SETSIZE)
37                  err_quit("too many clients");
38              FD_SET(connfd, &allset);       /* add new descriptor to set */
39              if (connfd > maxfd)
40                  maxfd = connfd; /* for select */
41              if (i > maxi)
42                  maxi = i;           /* max index in client[] array */

43              if (--nready <= 0)
44                  continue;           /* no more readable descriptors */
45          }
46          for (i = 0; i <= maxi; i++) {       /* check all clients for data */
47              if ( (sockfd = client[i]) < 0)
48                  continue;
49              if (FD_ISSET(sockfd, &rset)) {
50                  if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
51                          /* connection closed by client */
52                      Close(sockfd);
53                      FD_CLR(sockfd, &allset);
54                      client[i] = -1;
55                  } else
56                      Writen(sockfd, buf, n);

57                  if (--nready <= 0)
58                      break;          /* no more readable descriptors */
59              }
60          }
61      }
62  }
```

poll function

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

- Returns: count of ready descriptors, 0 on timeout, –1 on error
- The first argument is a pointer to the first element of an array of structures

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

- Input events and returned revents for poll:

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 data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

- Timeout value:

timeout value	Description
INFTIM	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

TCP server using poll

tcpcliserv/tcpservpoll01.c

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

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i, maxi, listenfd, connfd, sockfd;
 7     int     nready;
 8     ssize_t n;
 9     char    buf[MAXLINE];
10     socklen_t clilen;
11     struct pollfd client[OPEN_MAX];
12     struct sockaddr_in cliaddr, servaddr;

13     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

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

19     Listen(listenfd, LISTENQ);

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

tcpcliserv/tcpservpoll01.c

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

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

30              for (i = 1; i < OPEN_MAX; i++)
31                  if (client[i].fd < 0) {
32                      client[i].fd = connfd;  /* save descriptor */
33                      break;
34                  }
35              if (i == OPEN_MAX)
36                  err_quit("too many clients");
37              client[i].events = POLLRDNORM;
38              if (i > maxi)
39                  maxi = i;       /* max index in client[] array */

40              if (--nready <= 0)
41                  continue;       /* no more readable descriptors */
42          }

43          for (i = 1; i <= maxi; i++) {       /* check all clients for data */
44              if ( (sockfd = client[i].fd) < 0)
45                  continue;
46              if (client[i].revents & (POLLRDNORM | POLLERR)) {
47                  if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
48                      if (errno == ECONNRESET) {
49                              /* connection reset by client */
50                          Close(sockfd);
51                          client[i].fd = -1;
52                      } else
53                          err_sys("read error");
54                  } else if (n == 0) {
55                          /* connection closed by client */
56                      Close(sockfd);
57                      client[i].fd = -1;
58                  } else
59                     Writen(sockfd, buf, n);
60                  if (--nready <= 0)
61                          break;               /* no more readable descriptors */
62              }
63          }
64      }
65  }
```