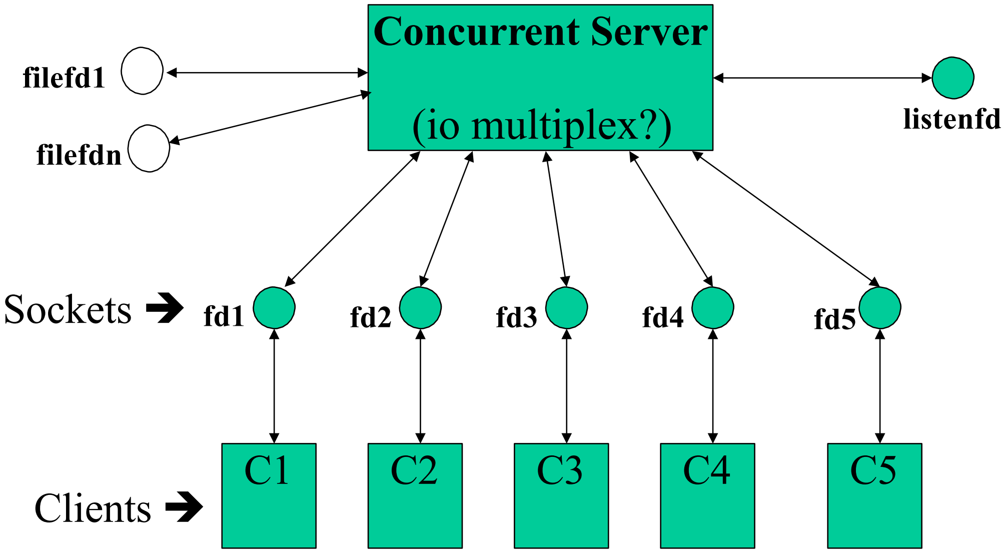
I/O Multiplexing

What is I/O Multiplexing?

- When an application needs to handle multiple I/O descriptors at the same time
  - E.g., file and socket descriptors, or multiple socket descriptors
- Application processes both *interactive* input and *network* socket
- Server handles multiple ports and protocols
  - Server handles both TCP listen and connected sockets
  - Server handles both TCP and UDP
- When I/O on any one descriptor can result in *blocking*

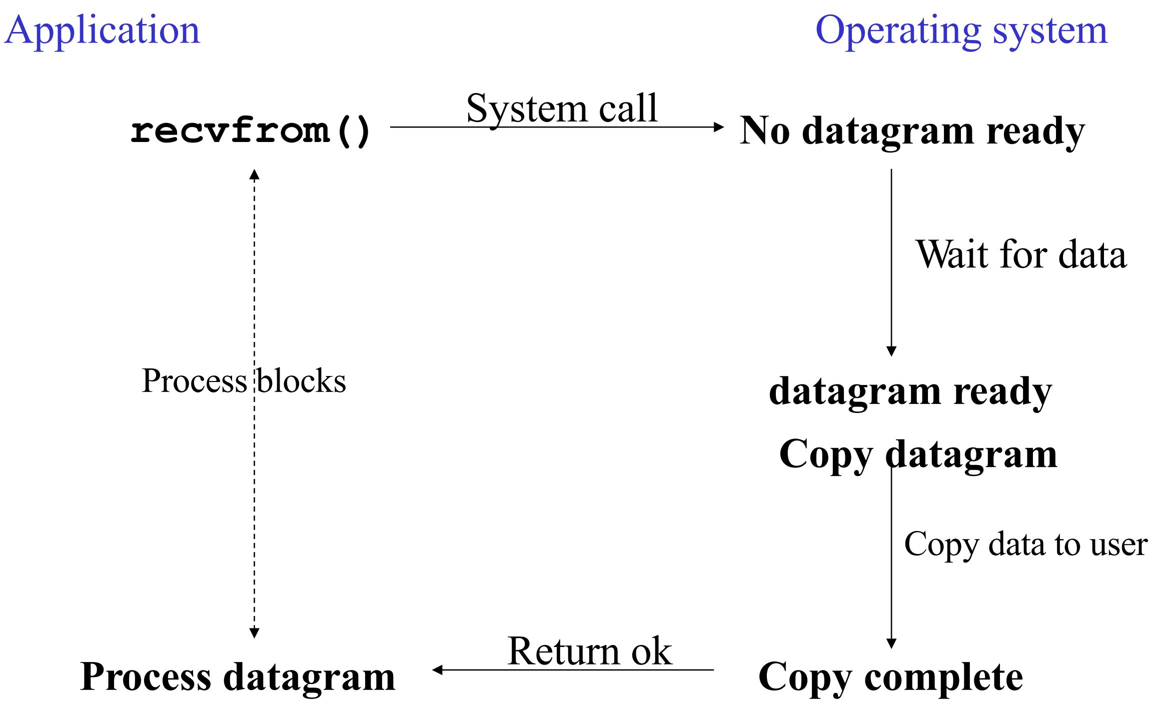
Non-Forking Concurrent Server



I/O models

1. Blocking I/O
2. Non-blocking I/O
3. I/O multiplexing – select(), poll()
4. Signal driven I/O
5. Asynchronous I/O

Blocking I/O



Application

Operating system

`recvfrom()` → System call → **No datagram ready**

Wait for data

datagram ready

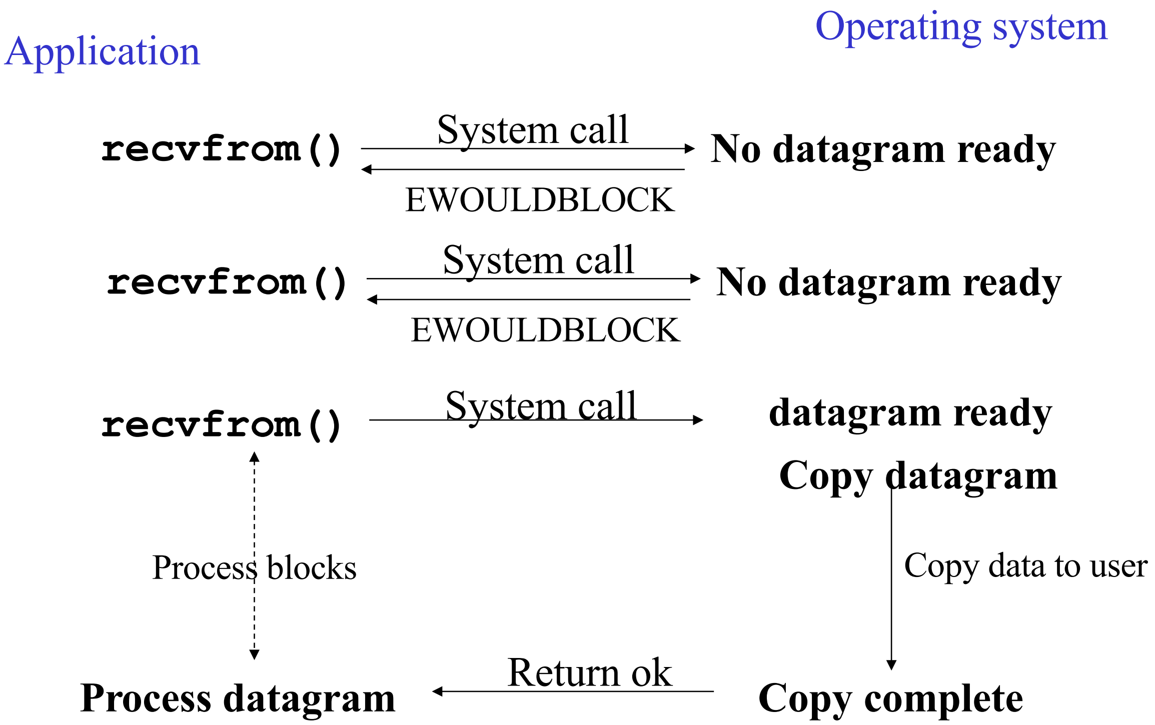
Copy datagram

Copy data to user

Process blocks

Process datagram ← Return ok ← **Copy complete**

Non-Blocking I/O



Application

Operating system

`recvfrom()` → System call → **No datagram ready**

EWOULDBLOCK

`recvfrom()` → System call → **No datagram ready**

EWOULDBLOCK

`recvfrom()` → System call → **datagram ready**

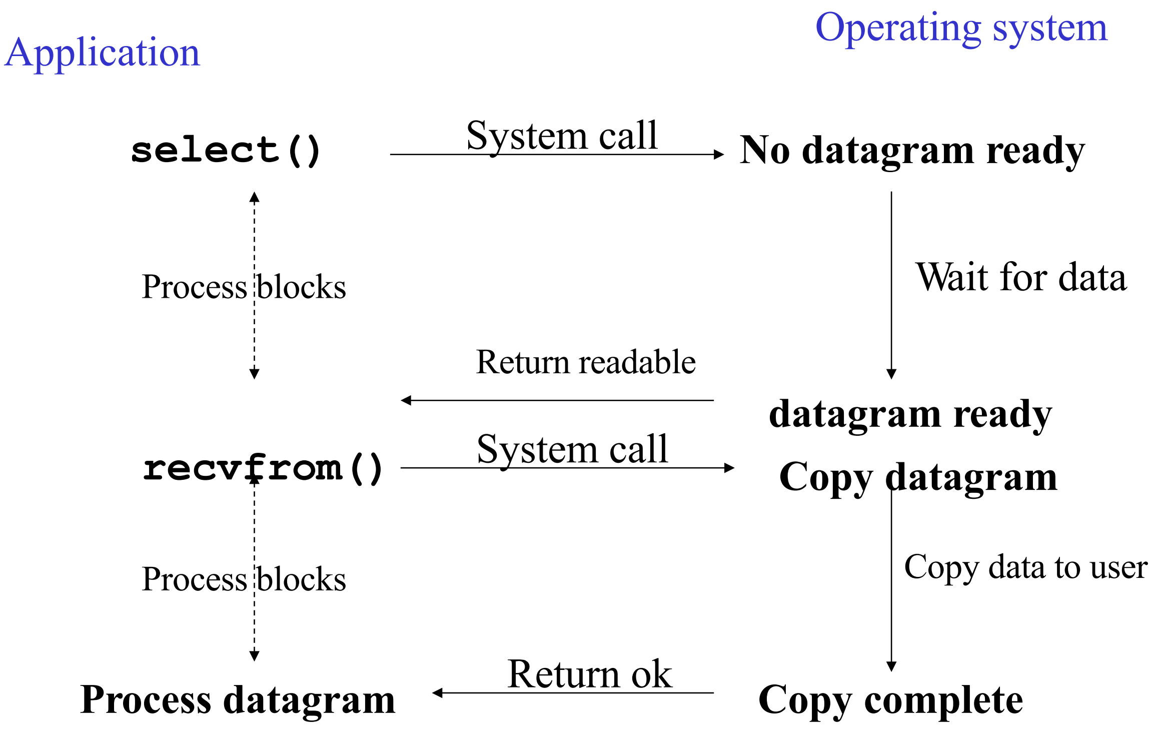
Copy datagram

Copy data to user

Process blocks

Process datagram ← Return ok ← **Copy complete**

I/O Multiplexing



Application

Operating system

`select()`

System call

No datagram ready

Process blocks

Wait for data

Return readable

datagram ready

`recvfrom()`

System call

Copy datagram

Process blocks

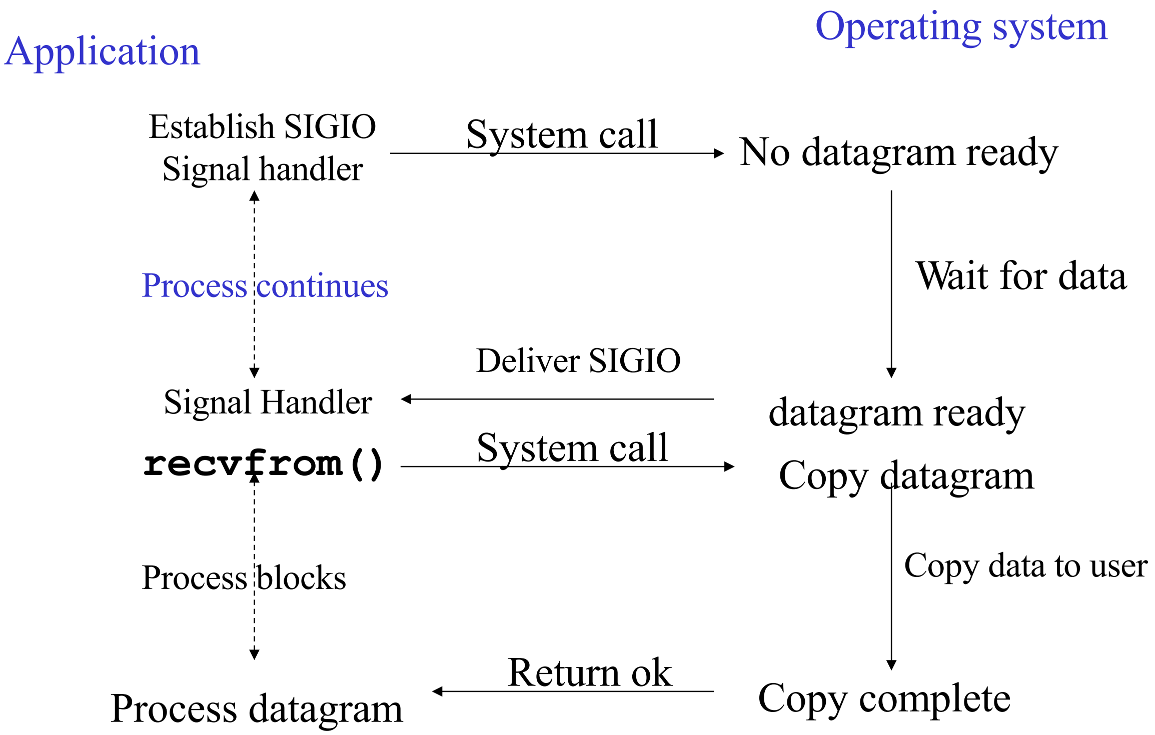
Copy data to user

Process datagram

Return ok

Copy complete

Signal driven I/O



Application

Operating system

Establish SIGIO
Signal handler

System call

No datagram ready

Process continues

Wait for data

Deliver SIGIO

Signal Handler

datagram ready

`recvfrom()`

System call

Copy datagram

Process blocks

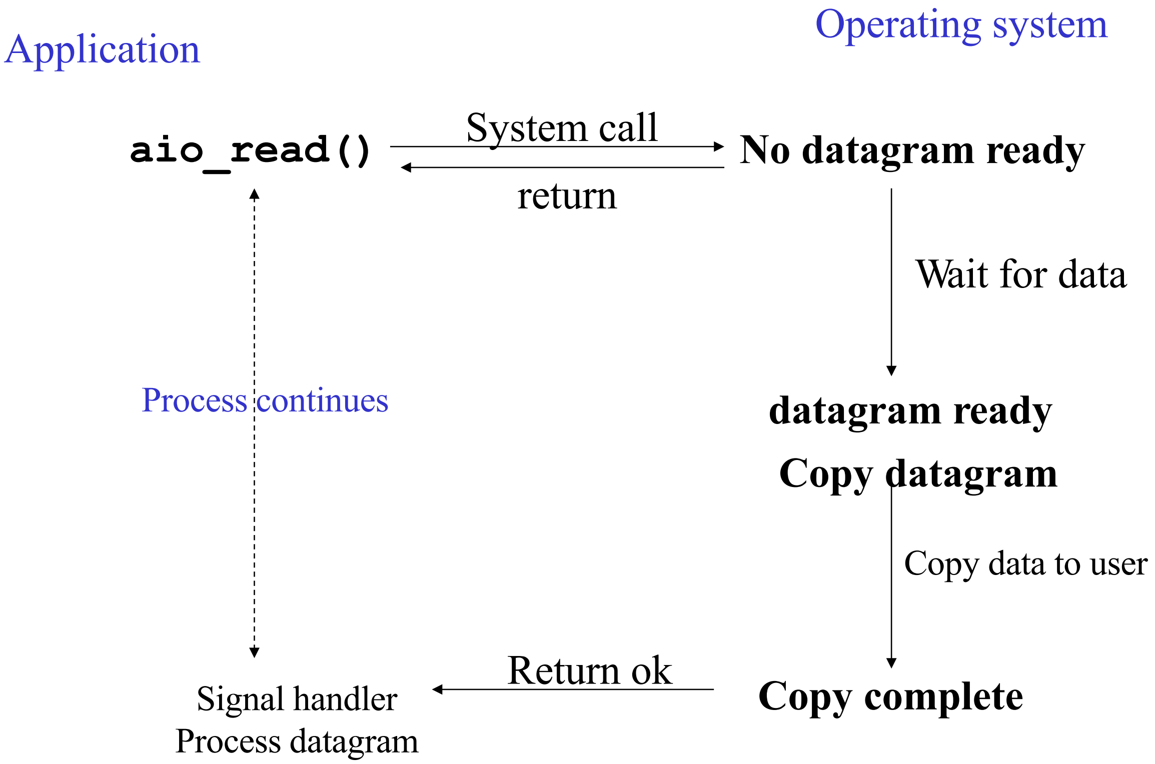
Copy data to user

Process datagram

Return ok

Copy complete

Asynchronous I/O



Application

Operating system

aio_read()

System call

return

No datagram ready

Wait for data

Process continues

datagram ready

Copy datagram

Copy data to user

Return ok

Copy complete

Signal handler
Process datagram

select() call

- Allows a process to *wait* for an event to occur on any one of its descriptors:
  - Wait forever
  - Wait for a fixed amount of time
  - Do not wait at all! This is called *polling*

- Types of *events*
  - Ready for read
  - Ready for write
  - Exception condition

select() call

```
int  select(int maxfdp1,        /* max. fd + 1 */
            fd_set *readfds,     /* read ready? */
            fd_set *writefds,    /* write ready? */
            fd_set *exceptfds,   /* exception? */
            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 */
};
```

fd_set

- Describes a set of descriptors that we want to wait on for events
- Traditionally held 1024 descriptors, but should be configurable for newer kernels
- Defines manipulation macros

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

Value-result arguments

- Select modifies descriptor sets pointed to by *readset*, *writeset*, and *exceptset* pointers
- On function call
  - Specify value of descriptors that we are interested in
- On function return
  - Result indicates which descriptors are ready
- Use FD_ISSET macro on return to test a specific descriptor in an fd_set structure
  - Any descriptor not ready will have its bit cleared
  - You need to turn on all the bits in which you are interested on all the descriptor sets each time you call select

maxfdp1 argument

- Specifies the maximum number of descriptors to be tested
- Its value is the maximum descriptor plus one
  - Descriptors 0, 1, 2, up through and including *maxfdp1*-1 are tested
  - Calculate the maxfdp1 value based on the current descriptor set *for efficiency* of processing fd_set
- Constant FD_SETSIZE defined by including header file <sys/select.h>
  - It is the number of descriptors in the fd_set datatype (traditionally 1024)

Socket conditions

- Handling conditions where peer can send data, RST or FIN

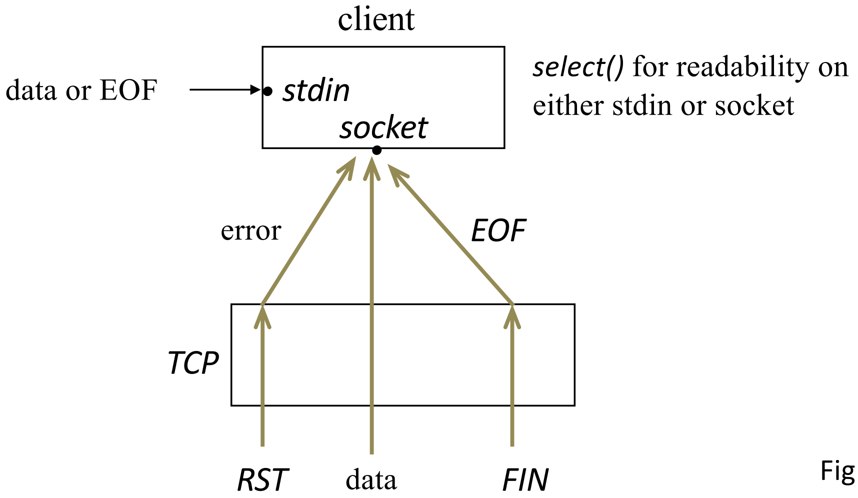


Fig. 6.8

Socket conditions (2)

- Can handle conditions where peer can send data, RST or FIN
- Peer TCP sends data
  - The socket becomes readable and read returns greater than 0 (number of bytes of data)
- Peer TCP sends a FIN (terminate or close)
  - The socket becomes readable and read returns 0 (i.e., EOF)
- Peer TCP sends a RST (crashed and rebooted)
  - The socket becomes readable and returns -1
  - **errno** contains the specific error code

Ready conditions

Summary of conditions that cause a socket to be ready for select()

Condition	Readable?	Writable?	Exception?
Data to read Read-half of the connection closed New connection ready for listening socket	● ● ●		
Space available for writing Write-half of the connection closed		● ●	
Pending error	●	●	
TCP out-of-band data			●

Non-forking concurrent server

```
fdset rdset, wrset;
int listenfd, connfd1, connfd2;
int maxfdp1;

/* do connection establishment etc */
    ……
/* initialize */
FD_ZERO(&rdset);
FD_ZERO(&wrset);
```

```
for ( ;; ) {
  FD_SET(connfd1, &rdset);
  FD_SET(connfd2, &wrset);
  FD_SET(listenfd, &rdset);

  maxfdp1 = max(connfd1, connfd2, listenfd) + 1;
  /* wait for some event */
  Select(maxfdp1, &rdset, &wrset, NULL, NULL);

  if (FD_ISSET(connfd1, &rdset)) {
       /* read data from connfd */
  }
  if (FD_ISSET(connfd2, &wrset)) {
       /* write data to connfd */
  }
  if (FD_ISSET(listenfd, &rdset)) {
       /* process a new connection */
  }
}
```

High resolution timer!

- Select can be used as a millisecond resolution timer.

```
select(0, NULL, NULL, NULL, &timeval);
```

- Usual **sleep**() call has resolution of seconds.

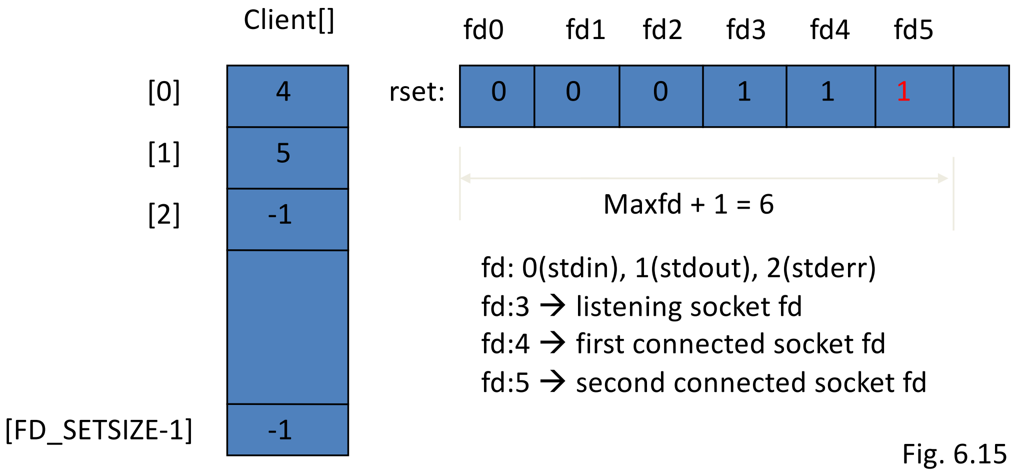
Fig 6.13 (strcliselect02.c)

```
#include    "unp.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)) {  /* socket is readable */
            if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
                if (stdineof == 1)
                    return;       /* normal termination */
                else
                    err_quit("str_cli: server terminated prematurely");
            }
            Write(fileno(stdout), buf, n);
        }
        if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
            if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
                stdineof = 1;
                Shutdown(sockfd, SHUT_WR);  /* send FIN */
                FD_CLR(fileno(fp), &rset);
                continue;
            }
            Writen(sockfd, buf, n);
        }
    }
}
```

Multiple client descriptors (e.g.)

- Instead of fork() per client, use select()
- After the second client connection is established (assuming connected descriptor returned by accept is 5)



Client[]

[0]	4
[1]	5
[2]	-1
[FD_SETSIZE-1]	-1

	fd0	fd1	fd2	fd3	fd4	fd5	
rset:	0	0	0	1	1	1	

Maxfd + 1 = 6

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

Fig. 6.15

Fig. 6.22 (tcpservselect01.c)

```
  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 */
      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 = Read(sockfd, buf, MAXLINE)) == 0) {
            /* connection closed by client */
          Close(sockfd);
          FD_CLR(sockfd, &allset);
          client[i] = -1;
        } else
          Writen(sockfd, buf, n);

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

pselect()

- Const nanosec timespec for timeout parameter
- Set a blocking mask while waiting, restore original mask when returning

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

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

pselect()

Race condition can exist between the call to select() and signal delivery

```
if (intr_flag)
   handle_intr();      /* handle the signal */
                                                         ←      ??
if ( (nready = select( ... )) < 0) {
   if (errno == EINTR) {
      if (intr_flag)
         handle_intr();
   }
   ...
}
```

pselect()

Signal mask argument overrides the existing signal mask

```
sigset_t newmask, oldmask, zeromask;

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask);        /* block SIGINT */
if (intr_flag)
   handle_intr();     /* handle the signal */
if ( (nready = pselect ( ... , &zeromask)) < 0 ) {
   if (errno == EINTR)  {
      if (intr_flag)
         handle_intr ();
   }
   ...
}
```

poll()

- Provides similar functionality to select()
- Events encoded in bits specifying certain conditions
- Wait forever INFTIM (-1 or < 0); or don’t block 0; or wait for > 0

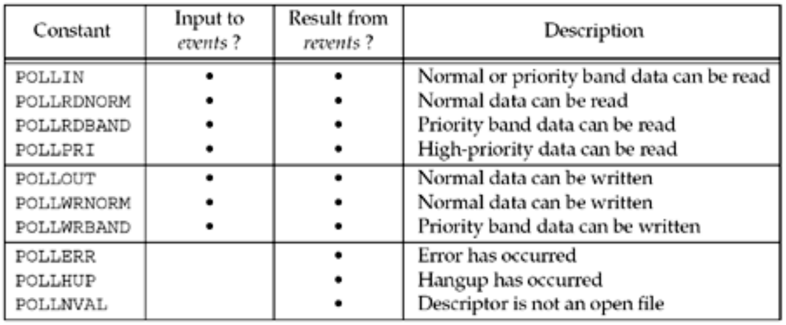
```
#include <poll.h>

int  poll(struct pollfd *fdarray,
          unsigned long nfds, /* array size */
          int timeout);       /* msec to wait */

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

Poll() Parameters

- Have two variables per descriptor
  - Input value events member **event**
  - Returns status for a descriptor in the corresponding **revent**s member



	Constant	Input to *events* ?	Result from *revents* ?	Description
inputs	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
outputs	POLLOUT	•	•	Normal data can be written
	POLLWRNORM	•	•	Normal data can be written
	POLLWRBAND	•	•	Priority band data can be written
errors	POLLERR		•	Error has occurred
	POLLHUP		•	Hangup has occurred
	POLLNVAL		•	Descriptor is not an open file

- Set **fd** member < 0 to ignore it
- Note fdarray allocated by caller; no fixed-size data type similar to fd_set

Poll Conditions (e.g.)

- Three classes of data identified:
  - Normal, priority, high-priority
- Some events for backward compatibility, e.g., POLLIN and POLLOUT
  - POLLIN: data other than high-priority data may be read without blocking
  - POLLRDNORM: normal data (priority band equals 0) may be read without blocking
  - POLLWRNORM: same as POLLOUT
- Regular TCP/UDP considered normal
- TCP out-of-band is priority band
- TCP read half-close is normal
- TCP error can be either normal or error (POLLERR)

***