I/O Models

- There are five I/O models under Unix
  - Blocking I/O
  - Nonblocking I/O
  - I/O multiplexing (*select* and *poll*)
  - Signal driven I/O (SIGIO)
  - Asynchronous I/O
- Two distinct phases for an input operation
  - Waiting for the data to be ready
  - Copying the data from the kernel to the process
- Definition of “data being ready”
  - For UDP, an entire datagram has been received
  - For TCP, data received passed the low-water mark

Blocking I/O

- Process is put to sleep if blocked

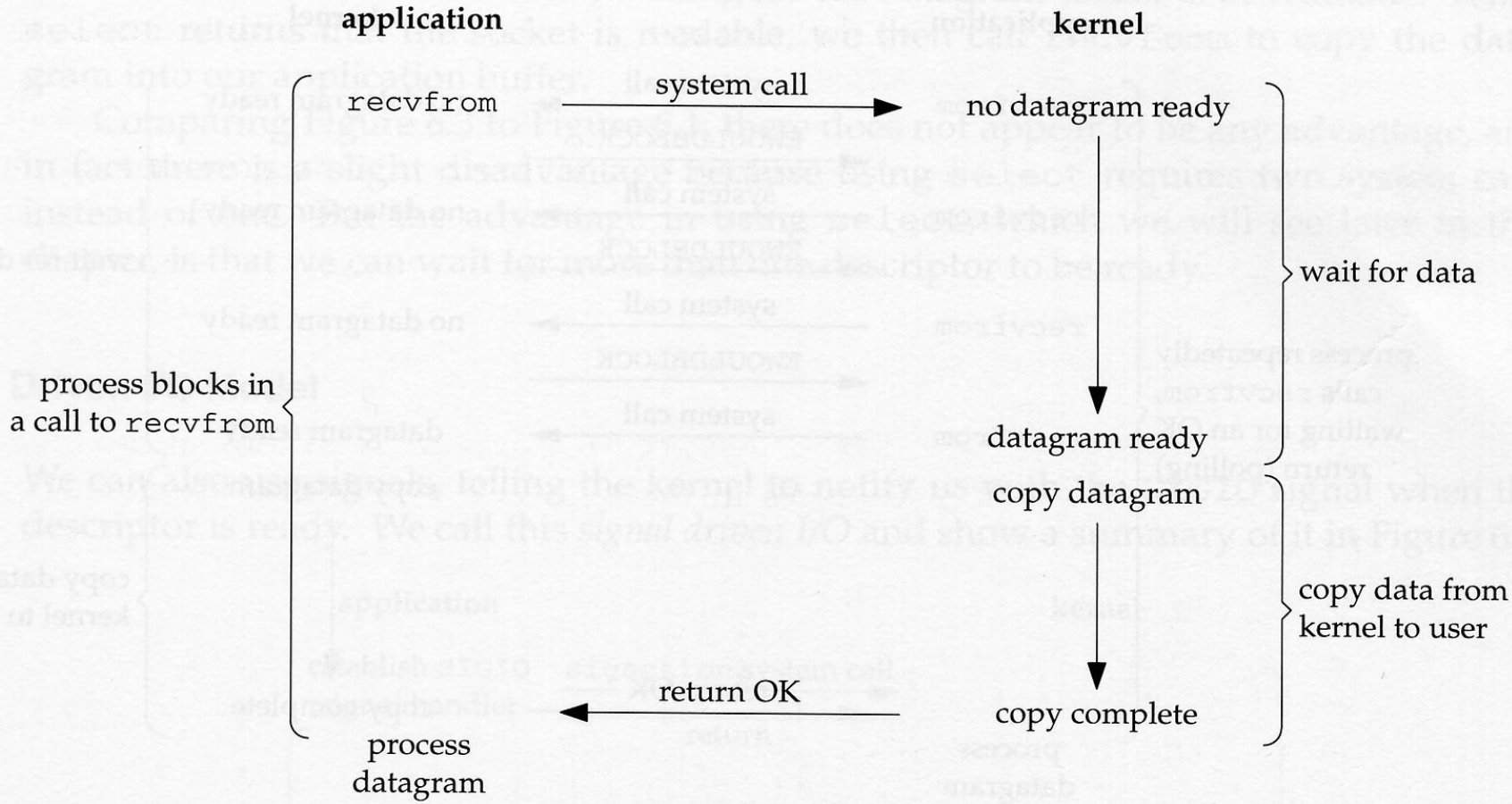


Figure 6.1 Blocking I/O model.

Nonblocking I/O

- When an I/O cannot be completed, the process is not put to sleep, but returns with an error (EWOULDBLOCK)
- Waste of CPU time

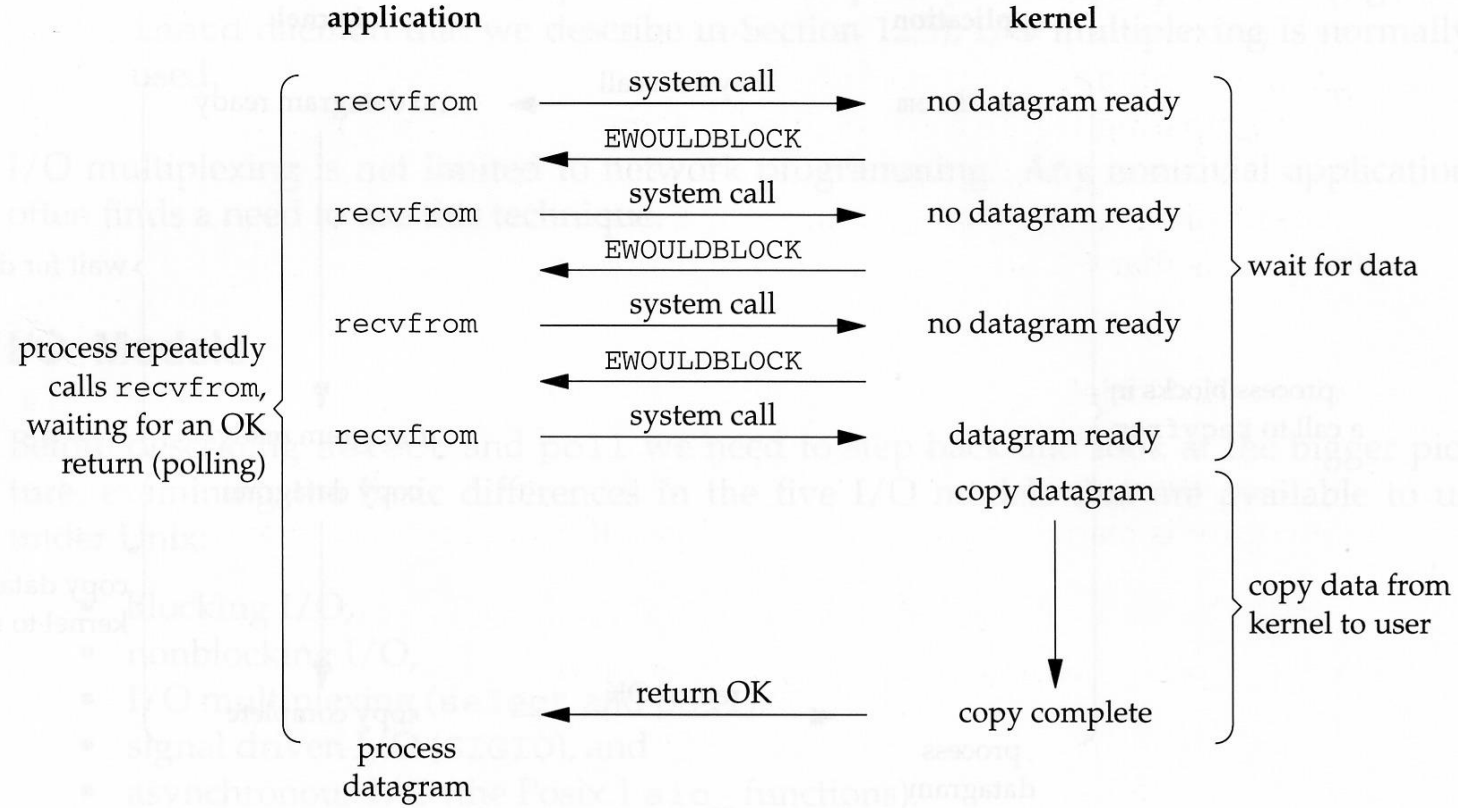


Figure 6.2 Nonblocking I/O model.

I/O Multiplexing

- Use *select* or *poll* to report if some descriptor is readable or writable. *select* may be blocked if no descriptor is readable or writable.

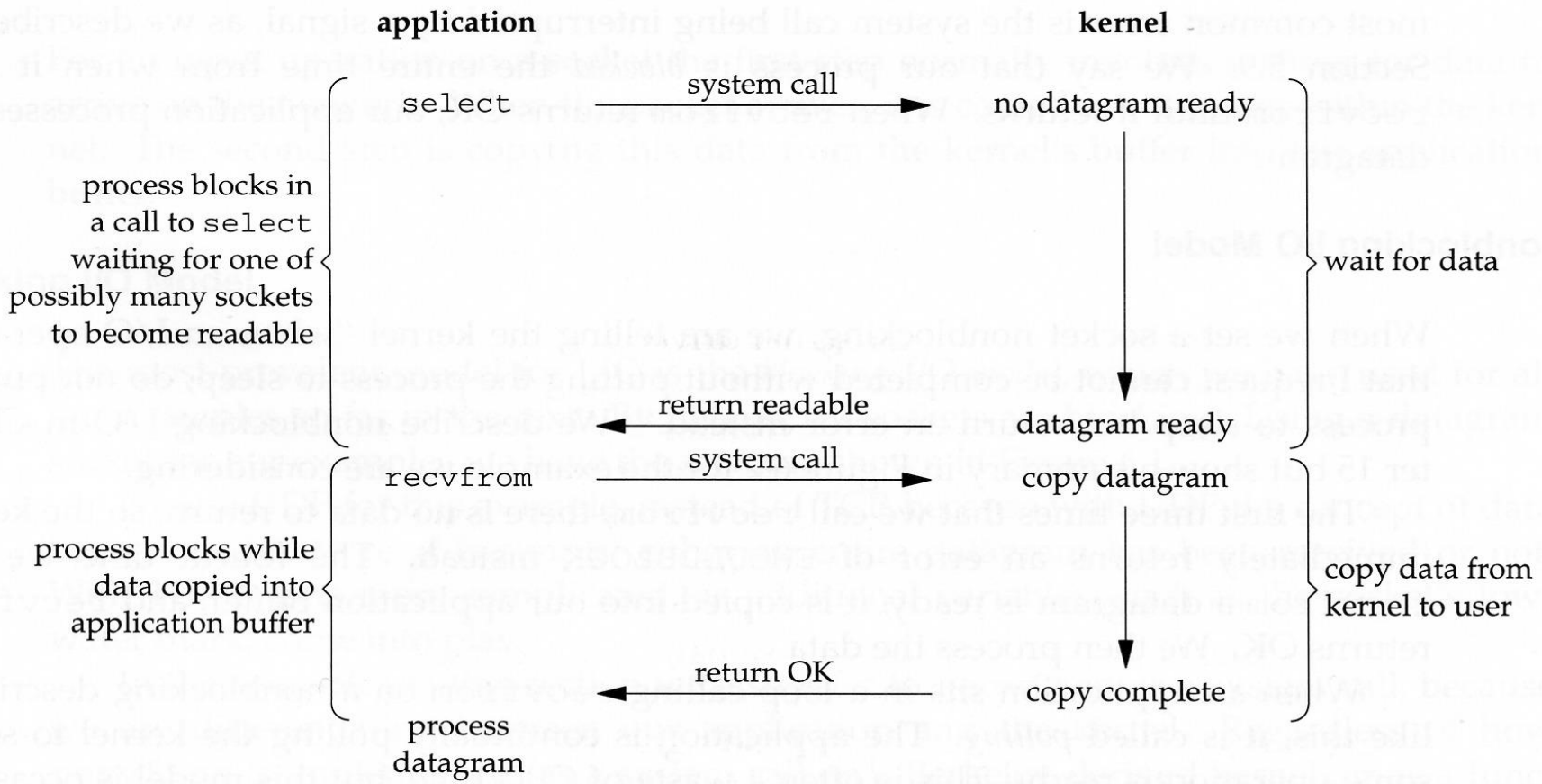


Figure 6.3 I/O multiplexing model.

Signal driven I/O

- If a descriptor is ready, notify the process with the SIGIO signal

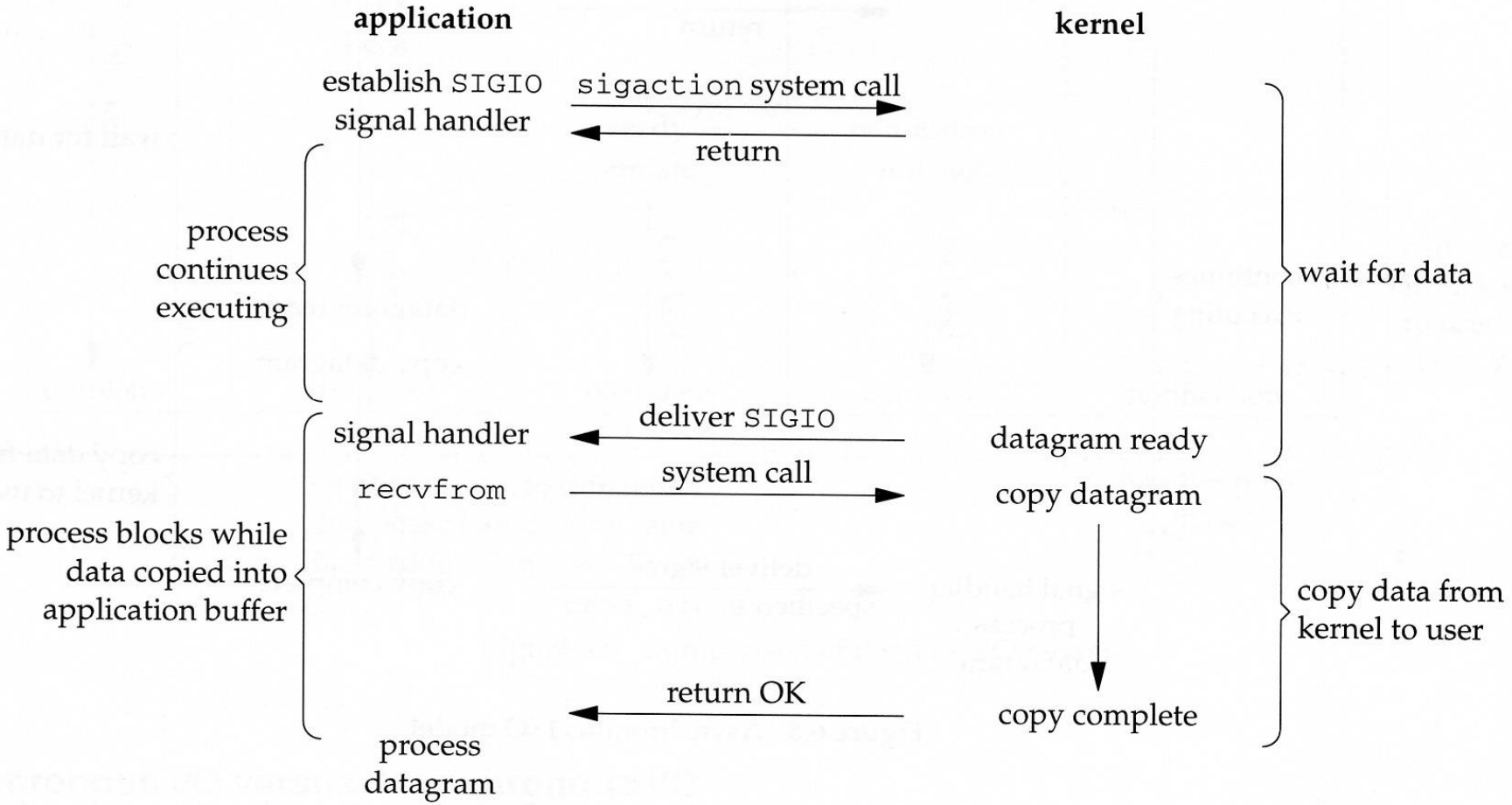


Figure 6.4 Signal Driven I/O model.

Asynchronous I/O

- The process *initiates* an I/O operation. When it is *complete*, the process is notified.

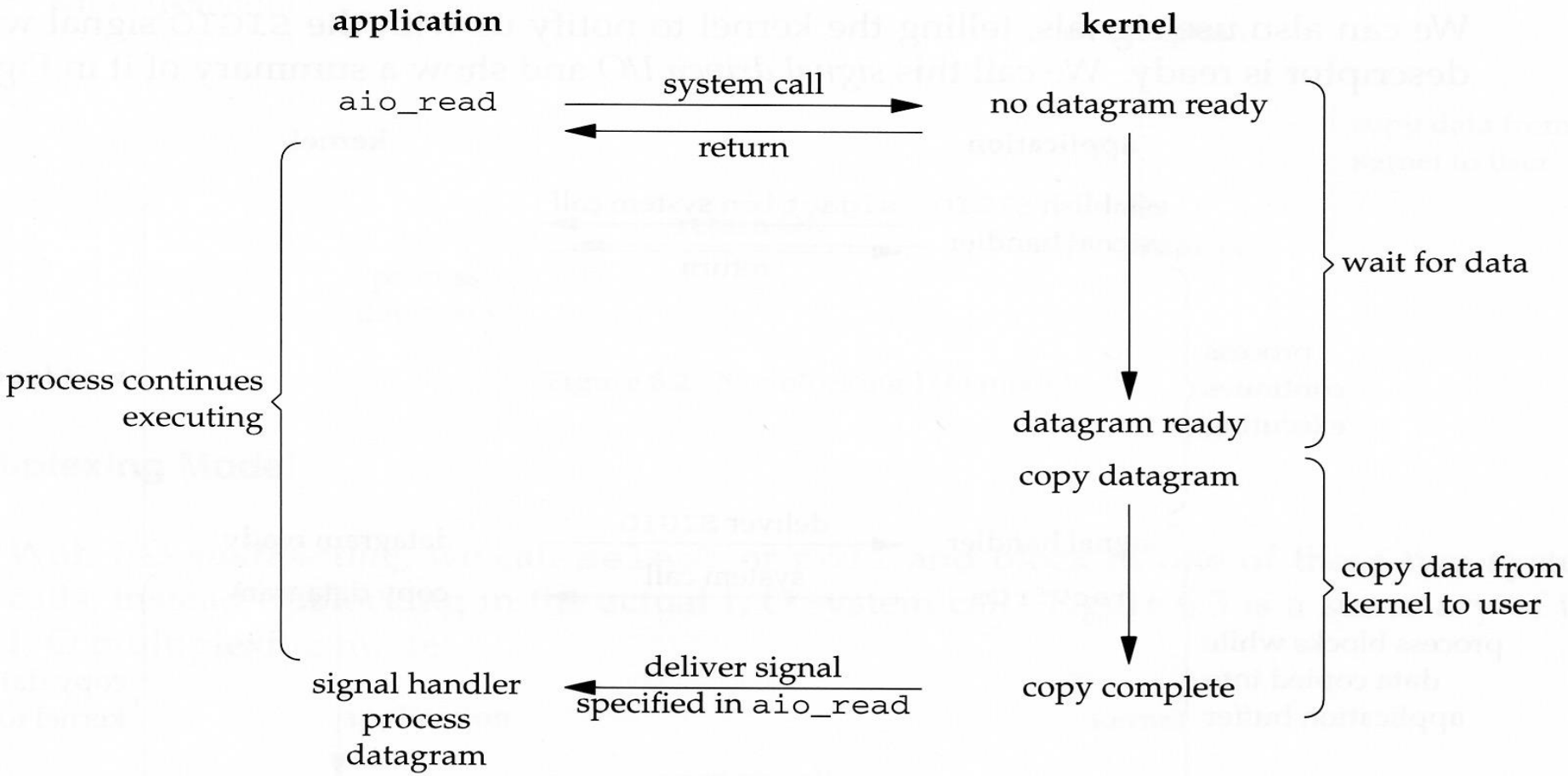


Figure 6.5 Asynchronous I/O model.

Comparison of I/O models

- The first four are synchronous I/O.

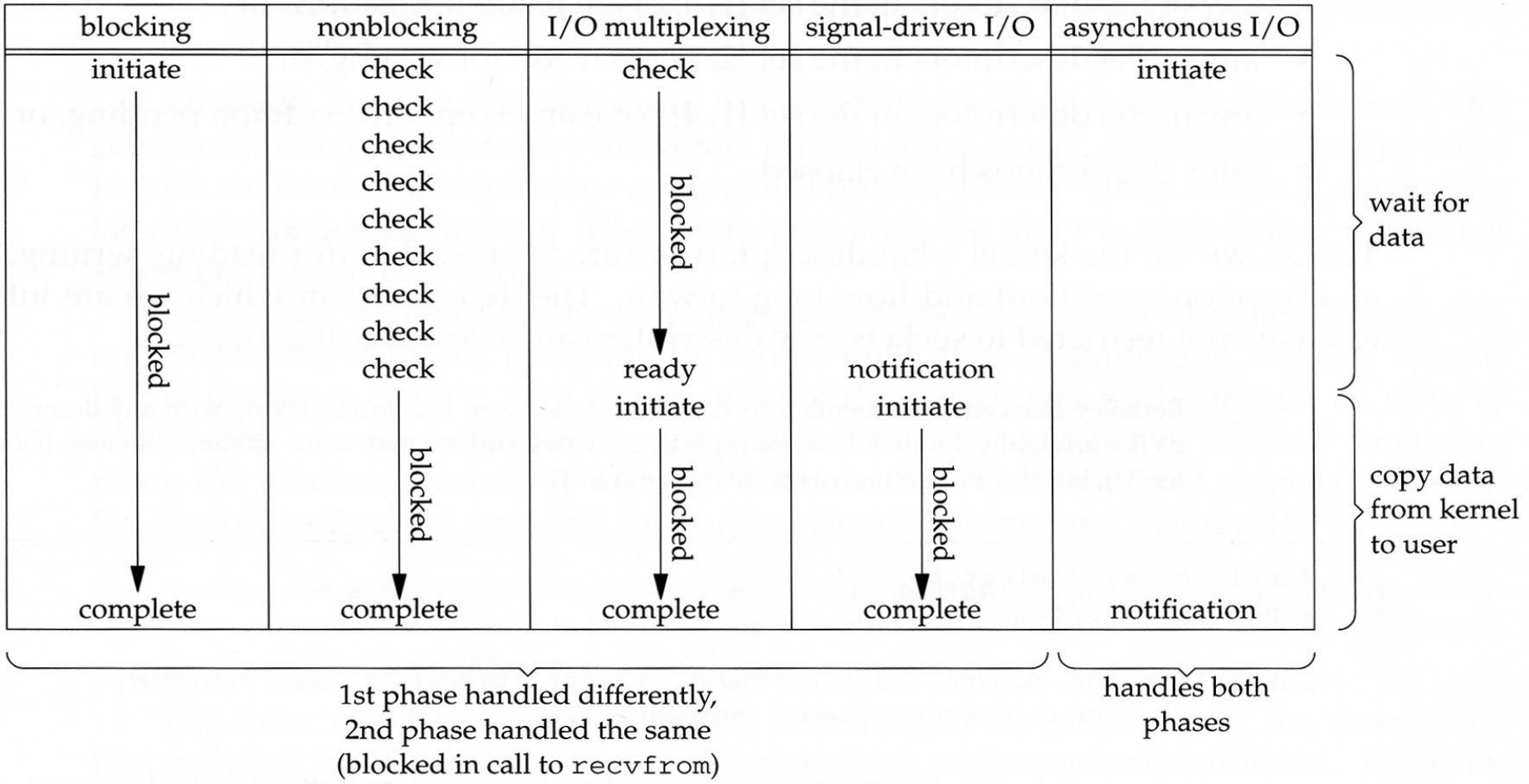


Figure 6.6 Comparison of the five I/O models.

Synchronous I/O vs Asynchronous I/O

- POSIX definition:
  - A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
  - An asynchronous I/O operation does not cause the requesting process to be blocked.

I/O Multiplexing 1

- What is I/O multiplexing?
  - 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, or the buffer is capable of taking more output)
  - Provided by *select* and *poll* functions
  - Or *pselect/poll*

I/O Multiplexing 2

- Scenarios for I/O multiplexing in C/S
  - A client handles multiple descriptors, or sockets
  - A server handles both a listening socket and its connected sockets
  - A server handles both TCP and UDP
  - A server handles multiple services and protocols (e.g. the *inetd* daemon)
  - It is possible, but rare, for a client to handle multiple sockets at the same time.

- I/O multiplexing is not limited to network programming.

Usage of I/O Multiplexing

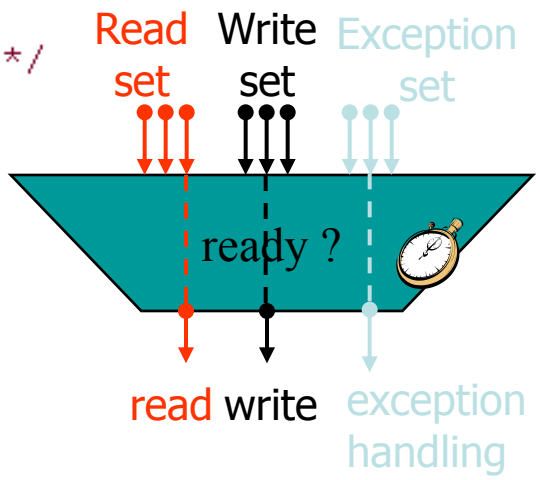
- Client
  - handles an interactive input and a socket
  - handles multiple sockets at the same time
- Server
  - handles both a listening socket and its connected socket
  - handles both TCP and UDP
  - handles multiple services and perhaps multiple protocols (e.g., inetd daemon)

Select Functions

```
#include <sys/time.h>                    /* UNIX */
#include <sys/select.h>                  /* UNIX */
#include <unistd.h>                      /* UNIX */
#include <winsock2.h>                    /* Windows */
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timeval *timeout);
```

Returns: count of read desciptors if positive, 0 on timeout, -1 on error

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



- Wait for any one of multiple events to occur and wake up the process only when
  - one or more of these events occurs
  - or, a specified amount of time has passed
    - wait forever: timeout = NULL
    - wait up to a fixed amount of time
    - polling: do not wait at all: timer value = 0

❖ *readset, writeset, exceptset* after select returns may be changed
  - Need to set them again for testing file descriptors ready

How to Manipulate Descriptor Sets

- Descriptor sets(fd_set): array of integers(FD_SETSIZE)
  - if fdset == NULL, no interest on the condition
  - caution: value result arguments

- Macros

```
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 1

- *select* function
  - 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
  -
```
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

select 2

Examples of when to use select

- Wait for descriptors {1,4,5} are ready for reading
- Wait for descriptors {2,7} are ready for writing
- Wait for descriptors {1,4} have an exception condition pending
- Wait for 10.2 seconds

- We tell the kernel what descriptors we are interested in and how long to wait.
- The descriptors are not restricted to sockets.

select 3

```
struct timeval { long tv_sec;
                 long tv_usec; }
```

- 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 of *timeout* is specified as both 0. (polling)

select 4

- The wait during *select* can be interrupted by signals (first two ways)
- Exception conditions
  - The arrival of out-of-band data

select 5

- The middle three arguments specify the descriptors we want the kernel to test.
- They are:
  - *readset*
  - *writeset*
  - *exceptset*
- They are value-result arguments. (most common error)
- On return, the result indicates the descriptors that are ready.
  - The bits should be re-set each time we call select

select 6

- Macros for *fd_set* datatype
  - `FD_ZERO(fd_set *fdset);`
    `// clear all bits in fdset`
  - `FD_SET(int fd, fd_set *fdset);`
    `// turn on the bit for fd in fdset`
  - `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 7

- *maxfdp1* specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested, plus 1. (most common error)
- Maximum number of descriptors: 256?, 1024 (Linux)?

Descriptor sets

- Array of integers : each bit in each integer correspond to a descriptor.

- fd_set: an array of integers, with each bit in each integer corresponding to a descriptor.

- 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 ? */

Example of Descriptor sets function

```
fd_set  rset;

FD_ZERO(&rset);/*all bits off : initiate*/
FD_SET(1, &rset);/*turn on bit fd 1*/
FD_SET(4, &rset); /*turn on bit fd 4*/
FD_SET(5, &rset); /*turn on bit fd 5*/
```

Maxfdp1 argument

- specifies the number of descriptors to be tested.
- Its value is the maximum descriptor to be tested, plus one.(hence our name of maxfdp1)(example:fd1,2,5 => maxfdp1: 6)
- constant FD_SETSIZE defined by including <sys/select.h>, is the number of descriptors in the fd_set datatype.(1024)

Conditions for Readiness 1

- A socket is ready for reading if any of the following conditions is true:
  - Data received in buffer greater than or equal to the low-water mark
  - Read-half of the connection is closed
  - A listening socket with nonzero number of connections
  - A socket error is pending
    - getsocket with SO_ERROR

Conditions for Readiness 2

- A socket is ready for writing if any of the following conditions is true:
  - Available space in the socket send buffer is greater than the low-water mark and the socket is connected or does not require a connection (UDP)
  - The write-half of the connection is closed (SIGPIPE)
  - A socket using a non-blocking connect has completed the connection, or the connect has failed
  - A socket error is pending
- A socket has an exception condition pending if there exists out-of-band data for the socket.

Summary of conditions

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			•

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

select-based Server

- Need to keep track of each client and its descriptor (array)
- Need to keep track of the highest used descriptor
- Good for many short lived clients

TCP Echo Iterative Server

Figure 6.14 TCP server before first client has established a connection.

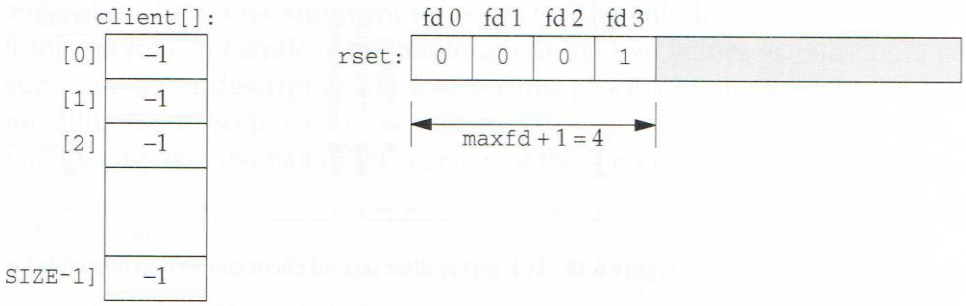


Figure 6.15 Data structures for TCP server with just listening socket.

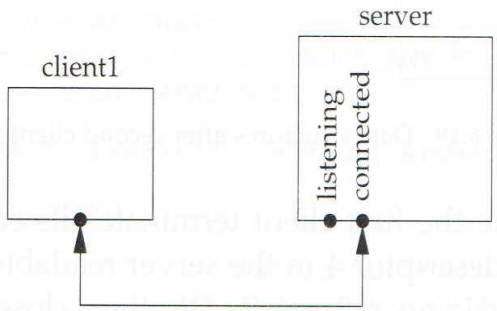


Figure 6.16 TCP server after first client establishes connection.

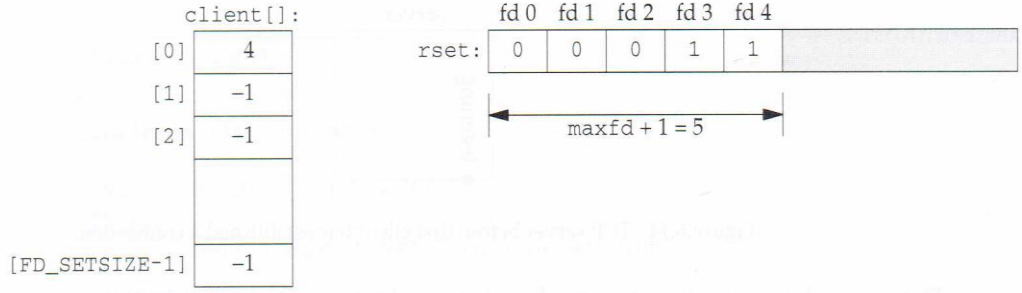


Figure 6.17 Data structures after first client connection is established.

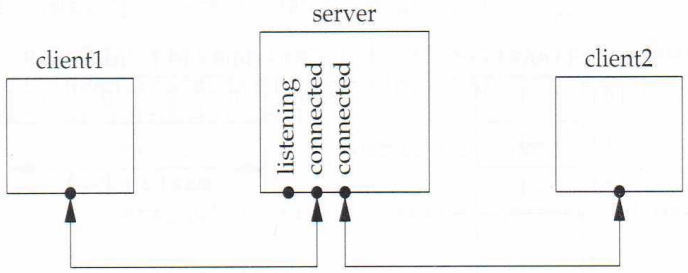


Figure 6.18 TCP server after second client connection is established.

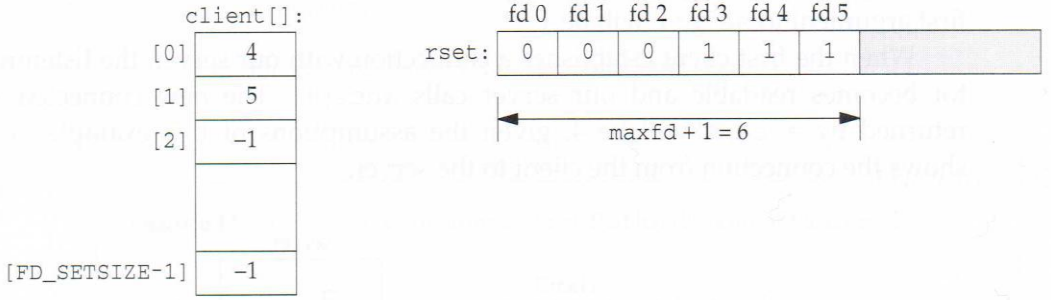


Figure 6.19 Data structures after second client connection is established.

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)
```

pselect function was invented by Posix.1g.

pselect function

- struct timespec{

  time_t tv_sec;   /*seconds*/

  long   tv_nsec; /* nanoseconds */
- sigmask => pointer to a signal mask.

Poll function

- Similar to select, but provide additional information when dealing with streams devices
- #include <poll.h>

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

/*return : count of ready descriptors, 0 on timeout, -1 on error*/

- Struct pollfd{

  int fd; /* descriptor to check */

  short events; /* events of interest on fd */

  short revents;/*events that occurred on fd*/

  }

  specifies the conditions to be tested for a given descriptor fd

  events: the conditions to be tested

  revents:the status of that descriptor

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		•	An error has occurred
POLLHUP		•	hangup has occurred
POLLNVAL		•	descriptor is not an open file

Timeout value for ***poll***

Specifies how long the function is to wait before returning

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

If we are no longer interested in particular descriptor, just set the fd member of the pollfd structure

pselect function

- struct timespec{

  time_t tv_sec;   /*seconds*/

  long   tv_nsec; /* nanoseconds */
- sigmask => pointer to a signal mask.

Parameter Passing

- **call by value:** copy going into the procedure
- **call by result:** copy going out of the procedure
- **call by value result:** copy going in, and again going out
- **call by reference:** pass a pointer to the actual parameter, and indirect through the pointer
- **call by name:** re-evaluate the actual parameter on every use. For actual parameters that are simple variables, this is the same as call by reference. For actual parameters that are expressions, the expression is re-evaluated on each access. It should be a runtime error to assign into a formal parameter passed by name, if the actual parameter is an expression. Implementation: use anonymous function ("thunk") for call by name expressions

Example 1

```
begin
   integer n;
   procedure p(k: integer);
      begin
      n := n+1;
      k := k+4;
      print(n);
      end;
   n := 0;
   p(n);
   print(n);
   end;
```

- Output:

- call by value:
- call by value-result:
- call by reference:

- Note that when using call by reference, n and k are aliased.

Example 2

```
begin
   integer n;
   procedure p(k: integer);
      begin
      print(k);
      n := n+1;
      print(k);
      end;
   n := 0;
   p(n+10);
   end;
```

- Output:

- call by value:
- call by name:

- Note that when using call by reference, n and k are aliased.

Example 3

```
begin
   integer n;
   procedure p(k: integer);
      begin
      print(k);
      end;
   n := 5;
   p(n/0);
   end;
```

- Output:

- call by value:
- call by name:

- Note that when using call by reference, n and k are aliased.

Example 4 **Non-local references**

```
procedure clam(n: integer);
  begin

    procedure squid;
    begin
      print("in procedure squid -- n="); print(n);
    end;

    if n<10 then clam(n+1) else squid;

  end;

clam(1);
```

Example 5

```
begin
  procedure whale(n: integer, p: procedure);
  begin
    procedure barnacle;
    begin
      print("in procedure barnacle -- n="); print(n);
    end;
    print("in procedure whale -- n="); print(n);
    p;
    if n<10 then
      begin
        if n=3 then whale(n+1,barnacle) else whale(n+1,p)
      end
  end;
    procedure limpet;
    begin
      print("in procedure limpet");
    end;

  whale(1,limpet);

  end;
```