Socket Programming
IO Models

References:

Stevens, Fenner, Rudoff, *UNIX Network Programming, vol. 1*, Chapters 5, and Appendix A.

https://beej.us/guide/bgnet/

Topics:

Solution to the echo client blocked in read (0,
Five different models for I/O provided by Unix.
select() function used for I/O multiplexing.

shutdown() function.
Concurrent server approach 2 using I/O Multiplexing.
DDoS Attackes.

OS I/O SYSTEM

2

Is socket an I/O device

- ***socket*** interface acts like a cable or pipeline connecting two networked entities.
- Data can be put into the socket at one end, and read out sequentially at the other end.
- Sockets are normally full-duplex, allowing for bi-directional data transfer
- Ref: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html

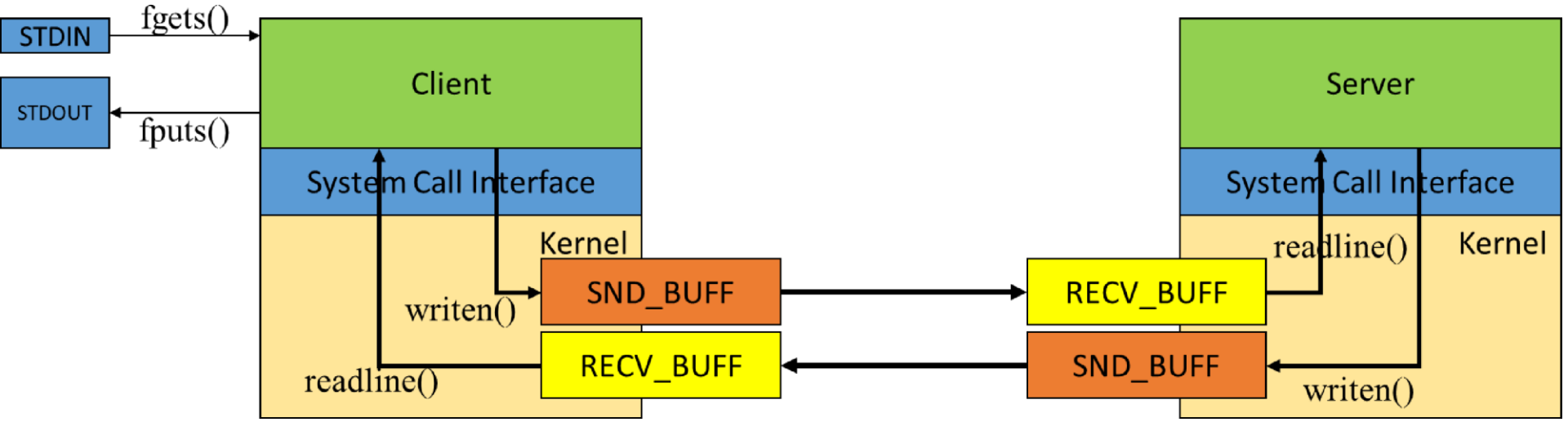
Network Communication

- In general, there are two phases for an input operation:

1. Waiting for the data to arrive on the network.

- When the packet arrive, copy it into a buffer within the kernel.

2. Copy this data from the kernel's buffer into the application buffer



4

Unix I/O Models

- Under Unix, there are five available I/O models:

  - Blocking I/O (blocking).
  - Nonblocking I/O (polling).
  - I/O Multiplexing (select() and poll()).
  - Signal driven I/O (SIGIO).
  - Asynchronous I/O (Posix.1).

- When reading from I/O devices we should follow up the models.

Synchronous I/O vs. Asynchronous I/O

- **Posix.1 definitions:**
- A *synchronous I/O operation* causes the requesting process to be blocked until that I/O operation completes.

➢ Four I/O models belong to this category:

- blocking.
- nonblocking.
- I/O multiplexing.
- signal-driven I/O.
- An *asynchronous I/O operation* does not cause the requesting process to be blocked**.**

➢ The OS notifies the process when I/O completes.

6

1. Blocking I/O Model

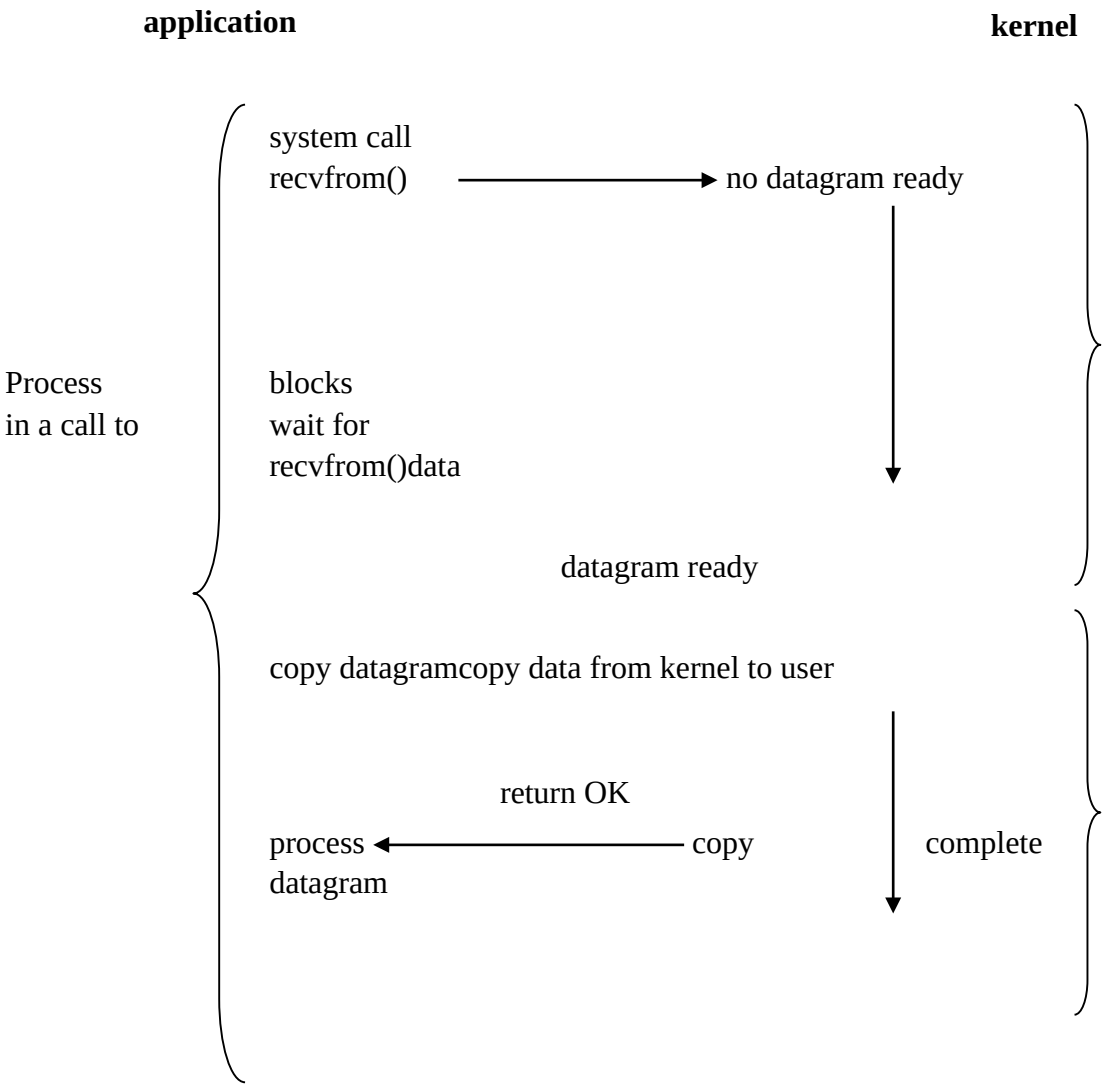
- The most prevalent model.
  - By default, all sockets are blocking.

- Blocking occurs under the following circumstances:
  - Read: When no data has arrived yet; example a block in read() or recvfrom().
  - Write: When the internal buffers are full and waiting for transmission, the program requests for more data to be sent.
  - Connection: when accept() find no pending connection in the listening queue.

Example:

- An application calls recvfrom()
  - The system call does not return until the datagram arrives and is copied into application buffer, or an error occurs.
  - The application is blocked the entire time from when it calls recvfrom() until it returns.
  - When recvfrom() returns OK, the application processes the datagram.
- The most common error is the system call being interrupted by a signal.

1. Blocking I/O Model (Cont.)



8

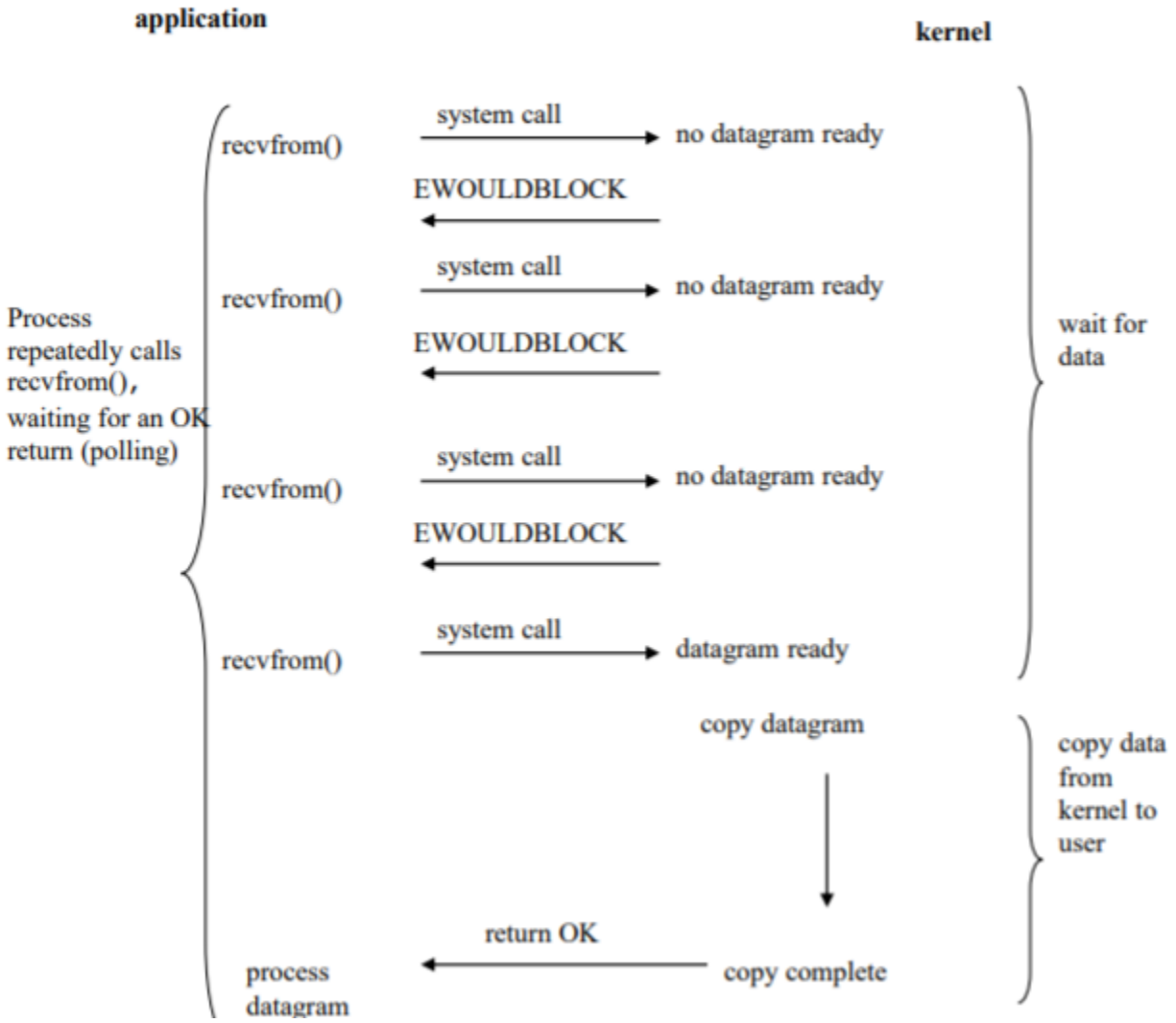
2. Nonblocking I/O Model

- When an I/O operation requested by a process that cannot be completed, the kernel does not put the process to sleep, but returns an error EWOULDBLOCK.
  - Can be used for a process that is monitoring more than one I/O.
  - Use fcntl() system call to set O_NONBLOCK of the file descriptor.

Example:

- An application calls recvfrom().
  - Assume there is no data to return → the kernel immediately returns an EWOULDBLOCK error.
  - When a datagram is ready, it is copied into the application buffer, and recvfrom() returns OK.
- In nonblocking, the application runs a loop calling recvfrom() on a nonblocking descriptor (*polling*) ➢ This can be a waste of CPU time.

2. Nonblocking I/O Model (Cont.)



3. I/O Multiplexing Model

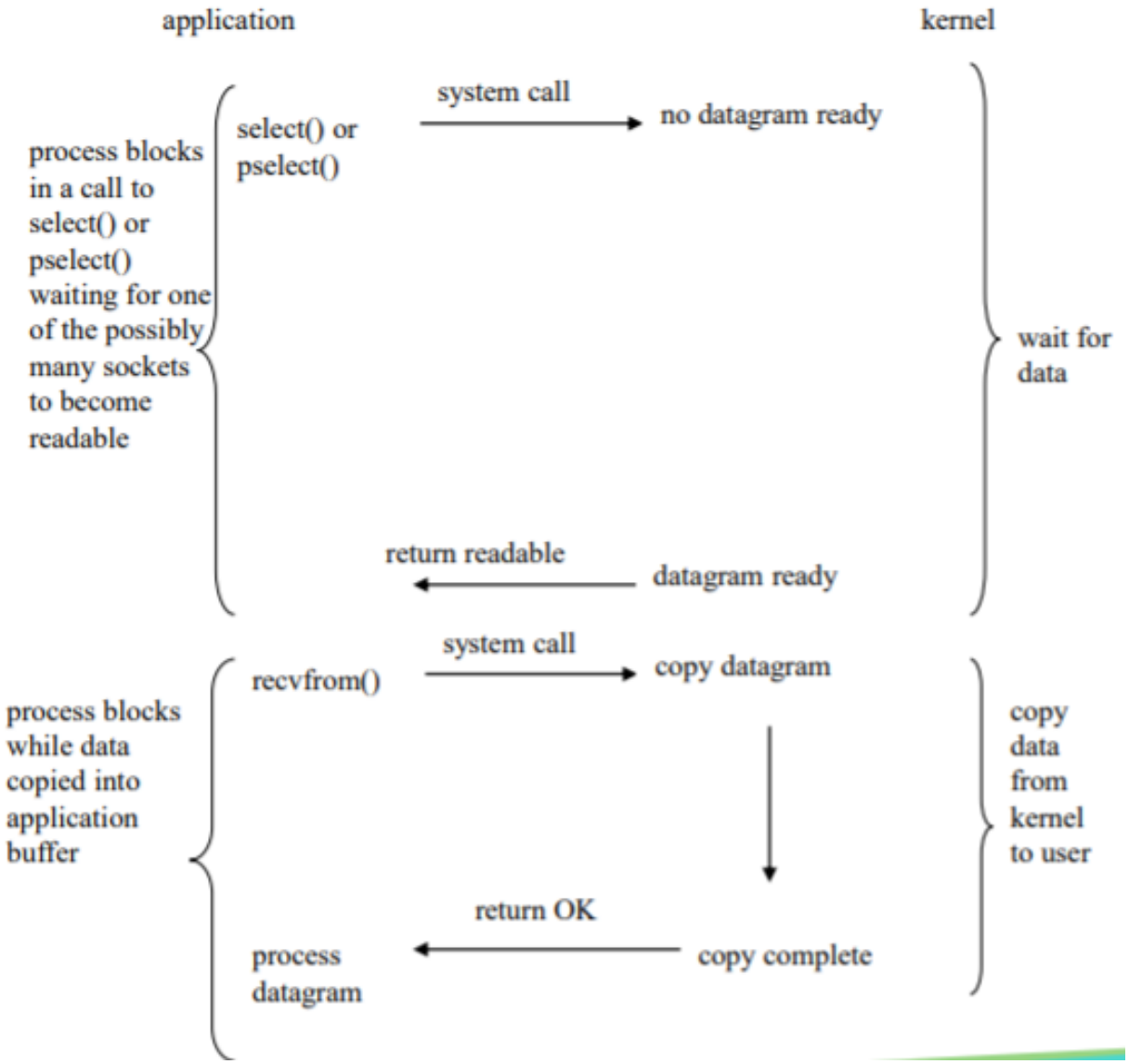
- I/O multiplexing is an alternative to nonblocking
  - The process is blocked until I/O is available from any of a set of file descriptors.
- I/O multiplexing has the power of telling the kernel the program has to be notified if one or more I/O conditions are ready.

Example:

A client is handling two inputs at the same time: a standard input and a TCP Socket.

- The client is blocked in a call to fgets() (on standard input).
- What happened when the server TCP process is killed?
  - The server process correctly sends a FIN, but, the client never reads it because it is blocked in fgets().
- In I/O multiplexing, a call to select(), pselect(), or poll() is made and it blocks in one of these two systems calls, instead of blocking in the actual I/O system call.
- Advantage: we can wait for more than one descriptor to be ready.
- Disadvantage: Similar to blocking I/O but requires two system calls.

3. I/O Multiplexing Model (Cont.)



4. Signal Driven I/O Model

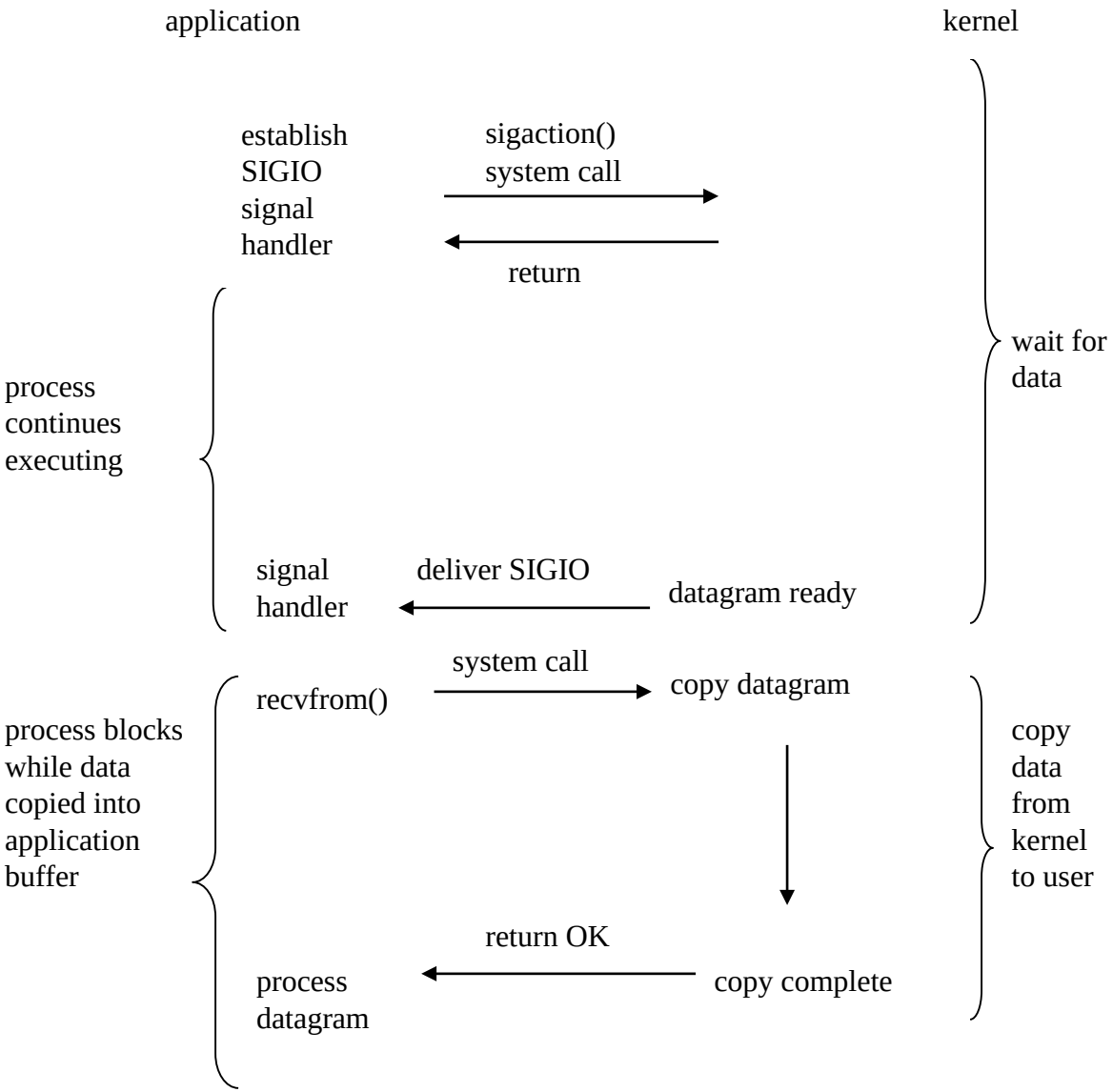
- The kernel notifies the app with the SIGIO signal when the descriptor is ready
  - This is called *signal driven* I/O.

Steps:

1. Enable the socket for signal driven I/O.
2. **Install a signal handler** using sigaction() or signal () system call.
   - **It will return immediately; process is not blocked**.
3. When the datagram is ready to be read, the SIGIO signal is generated for the process.
4. Process reads the datagram.

- The advantage of this model is that the process is not blocked while waiting for the datagram to arrive.
- The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or that the datagram is ready to be read.

4. Signal Driven I/O Model (Cont.)

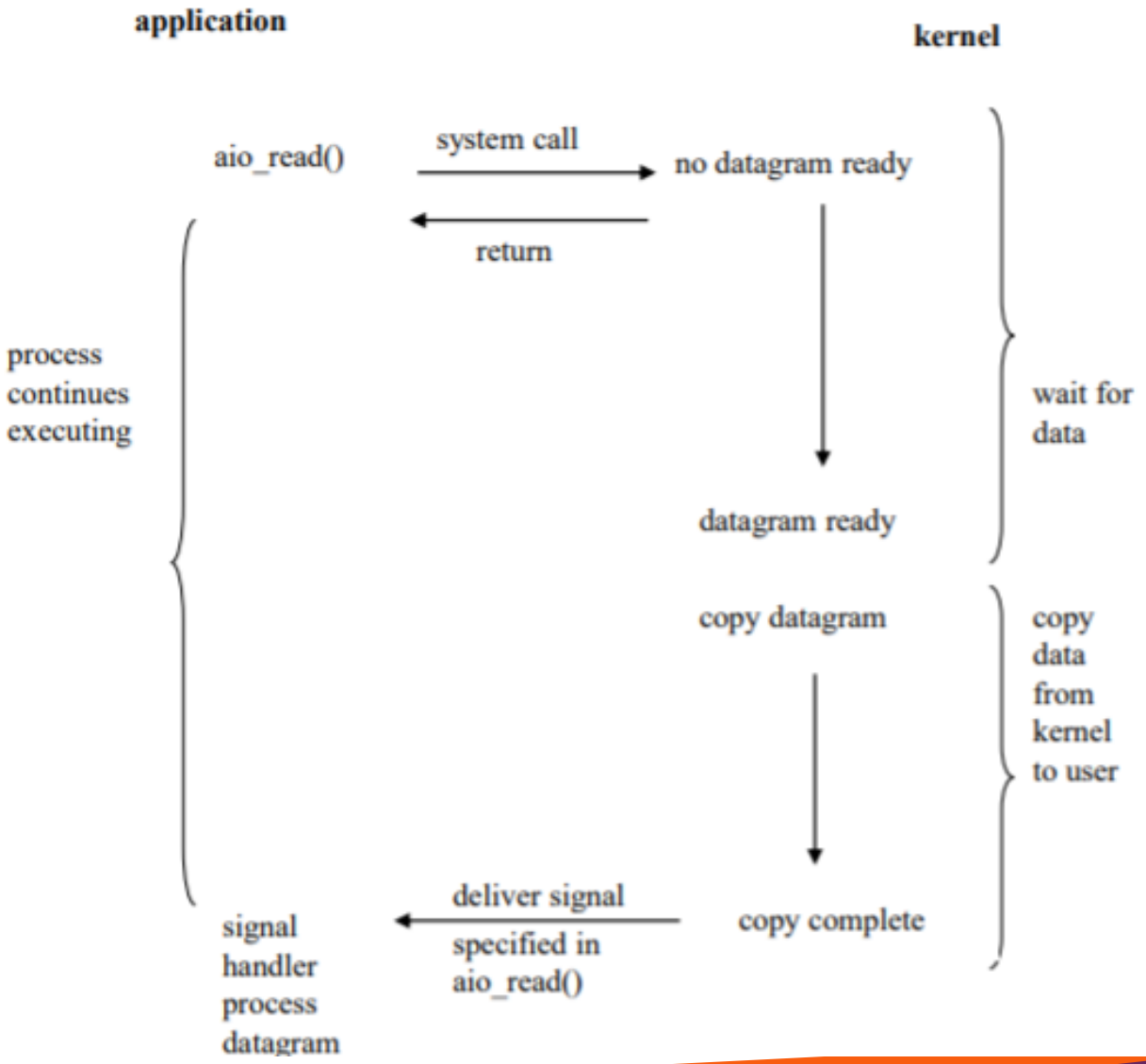


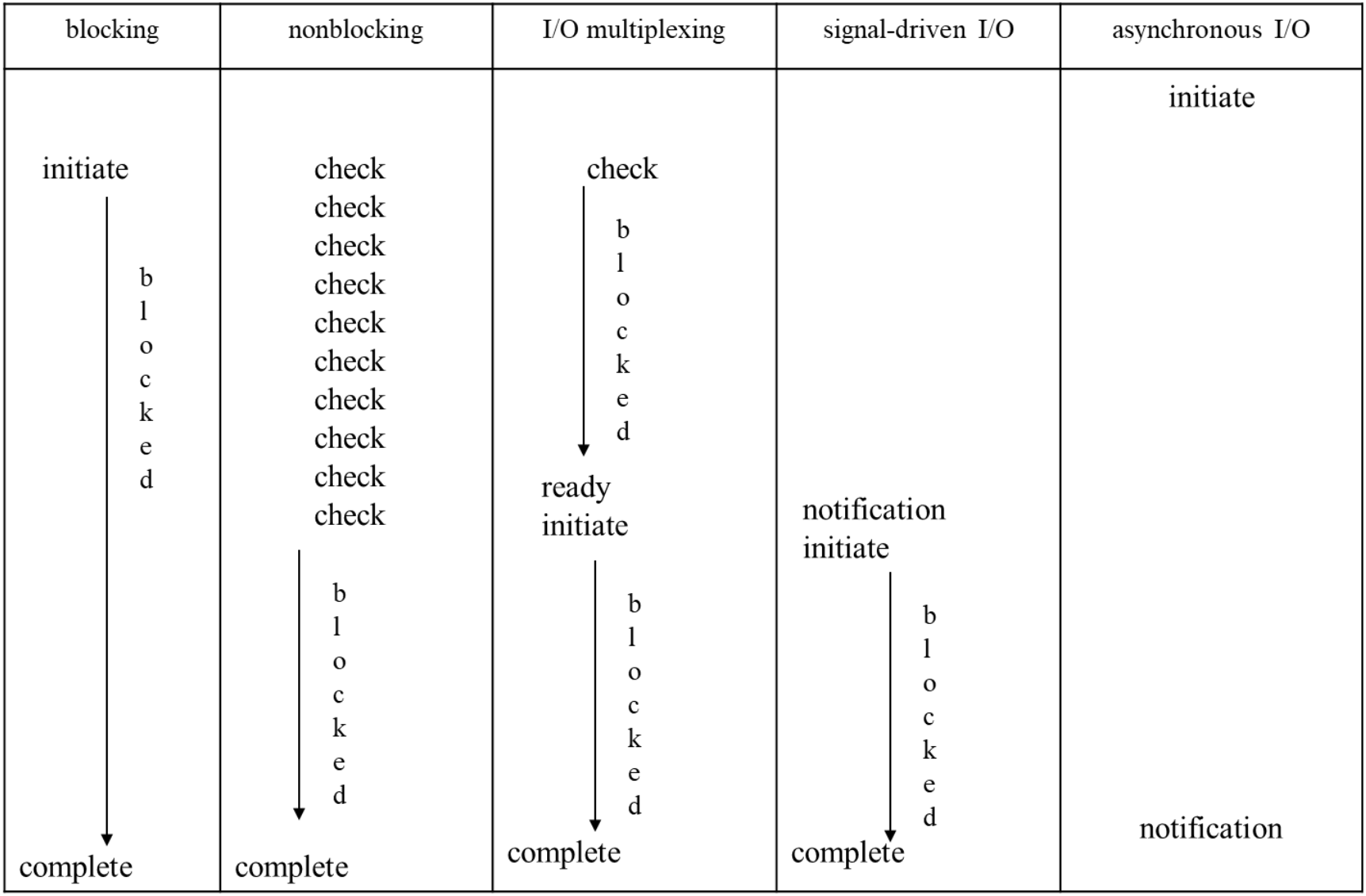
5. Asynchronous I/O Model

- The kernel starts the operation and notifies the process when the entire operation (including copying the data from the kernel to the buffer) is complete.

- The main difference between this model and the signal driven I/O model:
  - In signal driven I/O the kernel tells us when an I/O operation can be *initiated*.
  - In asynchronous I/O, the kernel tells us when an I/O operation is *complete*.

Asynchronous I/O Model (Cont.)





Comparison of the I/O Models

SOLUTION TO THE CLIENT PROGRAM USING I/O MULTIPLEXING

18

Revisit the client program

- A client handles multiple descriptors.
  - For Reading
  - For Writing

- What are the file descriptors the client program handle to read the data?
- What are the file descriptors the client program handle to write the data?
- In such situations we should give a chance to each descriptor to get the data to it's buffer.

select() Function

```
#include <sys/select.h>
#include <sys/time.h>
#include <signal.h> // for pselect()

/* returns positive count of ready descriptors, 0 on timeout, -1 on error */
int select(int maxfd1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
```

- The following slides use select() when they also apply to

  pselect() • Calling the select() function:

  - The process is blocked on select().
  - The kernel waits for any of multiple events (specified when select() was called).
  - The kernel wakes up the process:
- when one or more of the events occur; or
- when a specified amount of time (timeout) has passed.

select() Function (Cont.)

maxfd1: specifies the number of descriptors to be tested → its value is the maximum descriptor (number) to be tested, plus one;

Example: if we use descriptors 2, 5 and 7 the maxfd is 8 (7+1).

- *readset*, *writeset*, and *exceptset* are value-result arguments:

  - On calling: specify the values of the descriptors we are interested in .
  - On return: result indicates which descriptors are ready → use FD_ISSET macro to test. • select() returns:
  - the total number of bits that are ready across all the descriptor sets; or
    - 0 if the timer value expires before any of the descriptors are ready; or
    - –1 on error.

select() Function (Cont.)

- *timeout*: specifies how long the kernel must wait for one of the specified descriptors to become ready → three possibilities:
  - **Wait forever:** returns only when one of the specified descriptors is ready for I/O. • Set the *timeout* to a null pointer.
  - **Waits up to a fixed amount of time:**
- Specify the amount of time in timeval pointed to by *timeout*.
- **Do not wait at all:** returns immediately after checking the descriptors (called polling).

- timeval structure for select():

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

select() Function (Cont.)

On Linux: when select() returns, timeout reflects the amount of time not slept.

Select descriptors

- *readset*: specifies the descriptors that we want the kernel to test for reading; NULL if no condition.
- *writeset*: specifies the descriptors that we want the kernel to test for writing. NULL if no condition.
- *exceptset*: specifies the descriptors that we want the kernel to test for exception conditions. NULL if no condition.

How to specify the descriptors?

- select() uses descriptor set.

select() Function (Cont.)

- Typically an array of integers with each bit in each integer corresponding to a descriptor → descriptors 0 – 31, 32- 63, etc.

Use the following four 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

select() Function (Cont.)

FD_ISSET (int *fd*, fd_set **fdset*); /* is the bit for *fd* on in *fdset* ? */

Example for setting FDs to select function

Example: turns on bits for descriptors 2, 5 and 7.

```
#include <…>
int maxfd;
fd_set rset; /* Create a variable to sore the FD set */
FD_ZERO(&rset); /* Make the set zero */
```

select() Function (Cont.)

```
FD_SET(2,&rset); /* add FD 2 to the set */
FD_SET(5,&rset); /* add FD 5 to the set */
FD_SET(7,&rset); /* add FD 7 to the set */
maxfd = 7+1;
Select (<       >,<       >,<       >,<       >) // Fill the blanks;
```

Str_cli () with select function

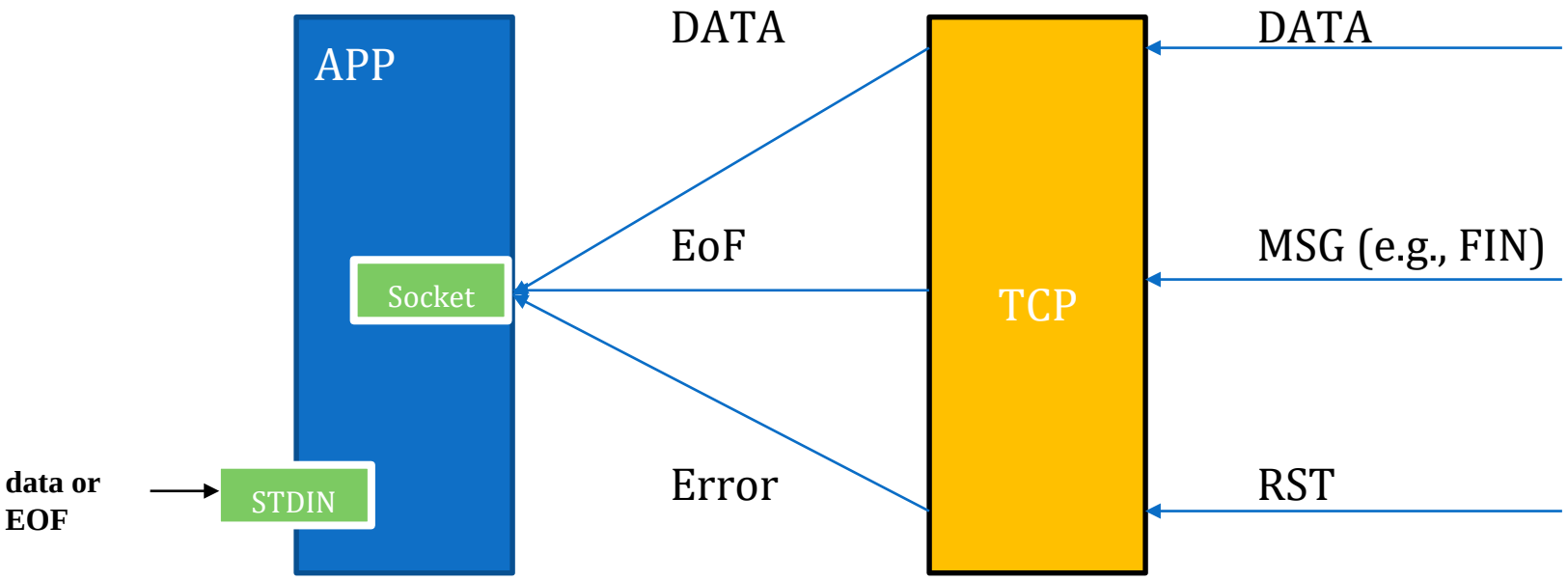
```
void
str_cli(FILE *fp, int sockfd)
{ int maxfdp1; /* Store the maximum file descriptor */
      fd_set    rset; /* reading set */
      char      sendline[MAXLINE], recvline[MAXLINE];

      FD_ZERO(&rset); /* zero the read set */
      for ( ; ; ) {
            FD_SET(fileno(fp), &rset); /* add STDIN to the reading set */
            FD_SET(sockfd, &rset); /* add SOCKET to the reading set */
            maxfdp1 = max(fileno(fp), sockfd) + 1; /* Calculate the MaxFD */
            Select(maxfdp1, &rset, NULL, NULL, NULL); /*call the select function with the reading set and time as forever*/

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

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

}

Three cases the application handles

Three conditions are handled with the socket now:

1. read returns > 0: data. If the peer TCP sends data, the socket becomes readable and read returns greater than 0 (the number of bytes of data)
2. read returns = 0: EOF (FIN received). If the peer TCP sends a FIN (if the peer process terminates), the socket becomes readable and read returns 0 (end-of-file)
3. read returns = -1: error code in errno. If the peer TCP sends an RST (the peer host has crashed and been rebooted), the socket becomes readable and read returns –1.



select({0,3}, {}, {}, NULL) for readability on either standard input or socket

When is a Descriptor Ready?

Conditions for a socket ready for reading (any of these four):

- 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** of the socket receive buffer; or
- The read-half of the connection is closed (i.e., a TCP connection that has received a FIN); or
- The socket is a listening socket and the number of completed connections has become nonzero (accept has occurred); or
- A socket error is pending.

When is a Descriptor Ready?

Conditions for a socket ready for writing (any of these four):

- The number of bytes of available space in the socket send buffer is greater or equal to the current size of the **low-water mark** for the socket send buffer (and the socket is connected if it is a TCP socket.); or
- The write-half of the connection is closed; or
- A socket error is pending; or
- A socket using nonblocking connect() has completed the connection, or connect() has failed.

Low-Watermarks

low-water marks

- The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading and writing.
- As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

shutdown() Function

/* returns 0 if OK, -1 on error */ #include <sys/socket.h> int shutdown(int *sockfd*, int *howto*);

- shutdown() is an alternative to terminating a network connection using close().

➢ The action of the function depends on the value of the *howto* argument:

- SHUT_RD: The read-half of the connection is closed.
- SHUT_WR: The write-half of the connection is closed.
- SHUT_RDWR: The read-half and the write-half of the connections are both closed.

- shutdown() gives us access to TCP's half-close.

➢ Set *howto* to 1 (SHUT_WR).

➢ Causes FIN to be sent.

➢ But data can be read from the socket.

➢ As normal, read () returns 0 when other end closes (e.g., we receive FIN).

str_cli() function using select() + shutdown()

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