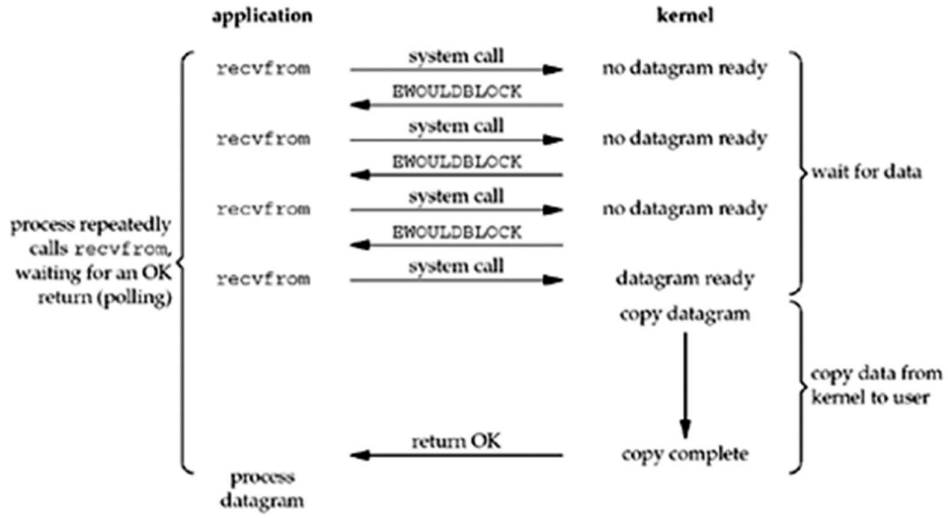
Zadanie 5.
Wyjaśnij jak przy pomocy multipleksowania wejścia-wyjścia (ang. I/O multiplexing) i nieblokujących wywołań systemowych zbudować jednowątkowy serwer TCP obsługujący współbieżnie wiele połączeń sieciowych. Zapoznaj się notatkami do §6 książki „Unix Network programming, Volume 1". Przeanalizuj kod serwera usługi `echo` wykorzystującego wywołanie `poll(2)`. Opisz znaczenie flag zawartych w polach «`events`» i «`revents`» struktury `pollfd`.

.

Nonblocking I/O Model

Nieblokujące wywołania systemowe na przykładzie serwera:

When a socket is set to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead". The figure is below:



Czyli pollujemy pojedyncze sockety, aż natrafimy na jakiś z którego możemy czytać.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

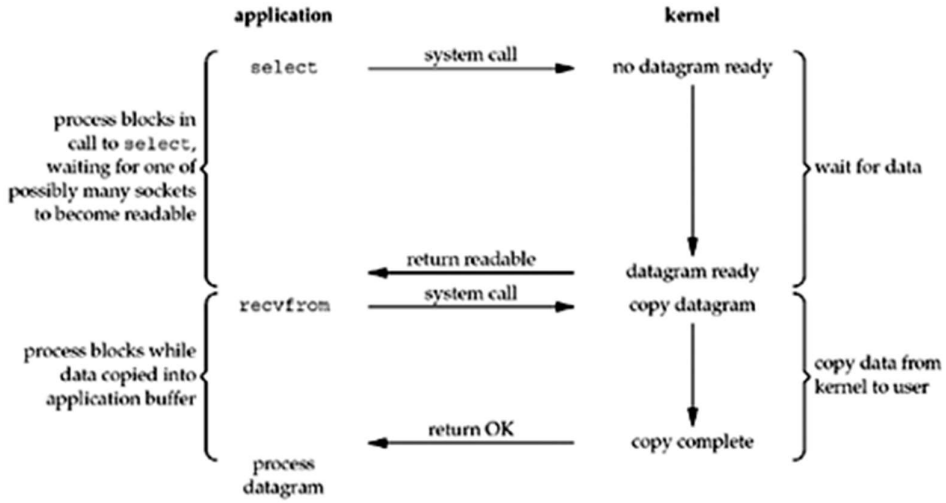
Jeśli nie ma żadnych danych w gnieździe sieciowym *recvfrom* czeka na otrzymanie danych chyba, że gniazdo jest nieblokujące, wtedy *recvfrom* zwraca -1 i ustawia errno na EWOULDBLOCK albo EAGAIN.

Czyli różnica między wywołaniem blokującym a nieblokującym jest taka, że wywołanie blokujące usypia proces do mementu aż dane będą dostępne, a nieblokujące zwraca błąd, jeśli dane nie są jeszcze dostępne.

I/O Multiplexing Model

Multipleksowanie I/O – blokowanie i wykorzystanie wielu strumieni I/O jednocześnie.

With **I/O multiplexing**, we call `select` or `poll` and block in one of these two system calls, instead of blocking in the actual I/O system call. The figure is a summary of the I/O multiplexing model:




```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

Select pozwala programowi monitorować wiele deskryptorów jednocześnie, w oczekiwaniu aż przynajmniej jeden z nich będzie gotowy na wykonanie pewnej operacji I/O (np. odczyt z gniazda przy pomocy *recvfrom* bez blokowania).

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Poll ma takie samo zastosowanie jak select. Struct pollfd działa tak, że na wejściu (events) dajemy bity eventów, które nas interesują, a na wyjściu (revents) dostajemy informację (też pod postacią maski) czy te eventy się wydarzyły.

```
struct pollfd {
               int   fd;         /* file descriptor */
               short events;     /* requested events */
               short revents;    /* returned events */
           };

The bits that may be set/returned in events and revents are defined
       in <poll.h>:

       POLLIN There is data to read.

       POLLPRI
              There is some exceptional condition on the file descriptor.
              Possibilities include:

              *  There is out-of-band data on a TCP socket (see tcp(7)).
```

* A pseudoterminal master in packet mode has seen a state change on the slave (see ioctl_tty(2)).

* A *cgroup.events* file has been modified (see cgroups(7)).

POLLOUT
Writing is now possible, though a write larger that the available space in a socket or pipe will still block (unless **O_NONBLOCK** is set).

POLLRDHUP (since Linux 2.6.17)
Stream socket peer closed connection, or shut down writing half of connection. The **_GNU_SOURCE** feature test macro must be defined (before including *any* header files) in order to obtain this definition.

POLLERR
Error condition (only returned in *revents*; ignored in *events*).
This bit is also set for a file descriptor referring to the write end of a pipe when the read end has been closed.

POLLHUP
Hang up (only returned in *revents*; ignored in *events*). Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer closed its end of the channel. Subsequent reads from the channel will return 0 (end of file) only after all outstanding data in the channel has been consumed.

POLLNVAL
Invalid request: *fd* not open (only returned in *revents*; ignored in *events*).

Jak działa ten serwerek z linka?
https://github.com/shichao-an/unpv13e/blob/master/tcpcliserv/tcpservpoll01.c

1. Inicjalizacja zmiennych
    a. Szczególnie tablicy client[] do której na początku dodany jest deskryptor listenera -> stąd będzie wiadomo czy ktoś nowy się połączył
2. W nieskończonej pętli polling eventów
    a. Sprawdzanie czy nie ma nowego klienta. Jeśli jest to dodanie deskryptora jego gniazda do tablicy klientów.
    b. Sprawdzenie w pętli dla każdego klienta czy przesłał nam jakieś dane.
        i. Jeśli tak, to następuje synchroniczny read a potem synchroniczny write do socketa.
    c. Obsługa błędów
        i. Przerwanie połączenia