MULTI-PROCESS TCP SERVER(CONT)

How to handle multiple inputs

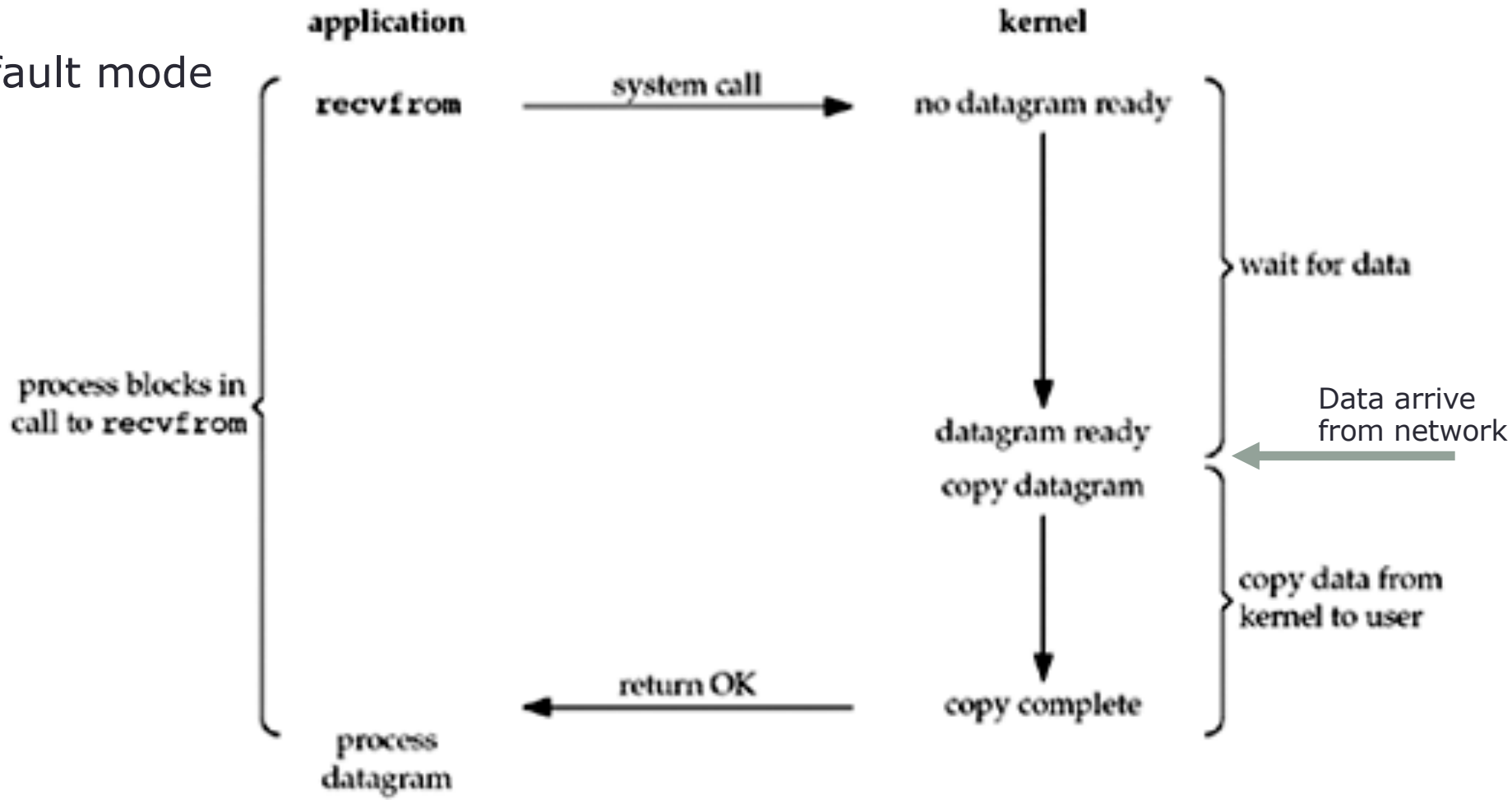
- Client hay server đơn tiến trình có thể phải xử lý vào ra dữ lieu từ nhiều nguồn khác nhau, ví dụ
  - Vừa từ đầu vào chuẩn (e.g., keyboard, khi gọi hàm fgets()) và vừa từ một socket
  - Từ nhiều sockets (đọc gửi dữ lieu từ nhiều socket khác nhau)
- Vấn đề: dữ liệu đến không đồng bộ
  - Chương trình (tiến trình) không biết khi nào dữ liệu từ một nguồn đến.
  - Nếu không có dữ liệu đến từ một nguồn, việc đọc dữ liệu đến bang `recv()` ở chế độ blocking sẽ dừng chương trình tại đó.
  - Nếu không có dữ liệu được nhập từ bàn phím, hàm `fgets()` block.
  - ➔ Nếu chương trình bị block do đọc dữ liệu từ một nguồn, CT không thể xử lý dữ liệu từ nguồn khác
- Giải pháp có thể
  - Polling sử dụng non-blocking socket ➔ Không hiệu quả
  - I/O multiplexing sử dụng select() ➔ đơn giản.

I/O Models

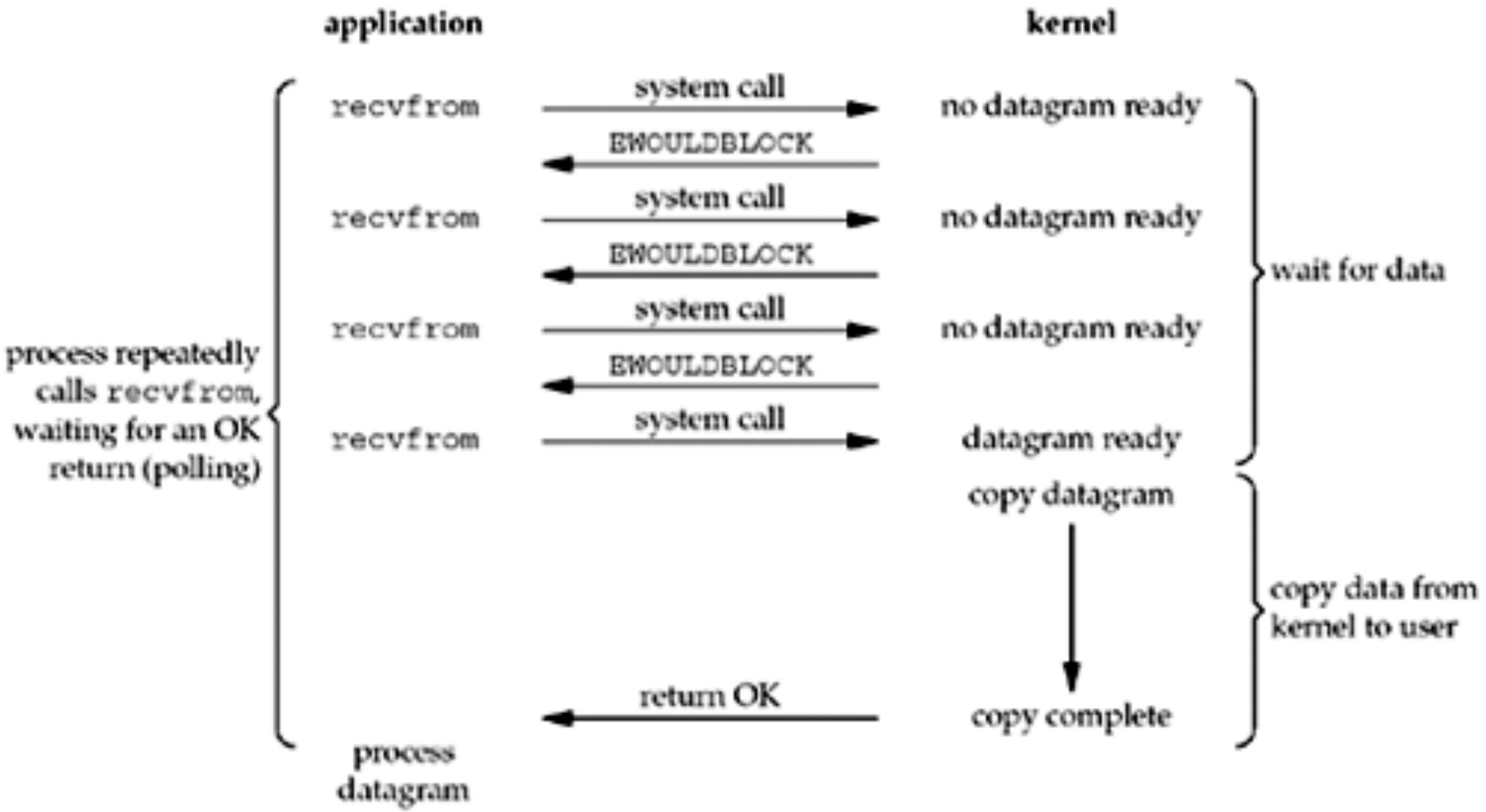
- Năm mô hình vào ra
  - blocking I/O
  - nonblocking I/O
  - I/O multiplexing (select and poll)
  - signal driven I/O (SIGIO)
  - asynchronous I/O (the POSIX aio_functions)
- Mỗi quá trình đọc dữ liệu từ một socket gồm 2 giai đoạn:
  1. Chờ có dữ liếu đến: Khi dữ liệu đến, nó được copy vào vùng đệm trong hạt nhân Hệ điều hành
  2. Copy dữ lieu từ hạt nhân HDH vào vùng đệm của ứng dung (tiến trình)
-

Blocking I/O Model

Default mode



Non-blocking I/O Model



Non-blocking I/O model

- Để hoạt động ở chế độ này, socket phải được thiết lập kiểu non-blocking.
- Khi đó, mỗi lần gọi recv() hoặc recvfrom(), nếu chưa có dữ lieu đến socket, hạt nhân sẽ trả ngay về kết quả lỗi EWOULDBLOCK
- Nếu không, hạt nhân sẽ trả kết quả OK
- Như vậy muốn đọc được dữ liệu, cần polling liên tục đến khi nhận được OK
  - Lặp với recv(), recvfrom()

Thiết lập socket ở chế độ non-blocking: fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int sockfd, int cmd, long arg)
```

- fcntl = Control socket descriptors
- Performs the operations *cmd* with argument *arg* on the file descriptor *sockfd*
- Parameter:
  - `sockfd`: socket descriptor
  - `cmd`: operation (set hoặc get)
  - `arg`: required argument
- Return: depends on `cmd`

fcntl(): operations

- cmd =F_SETFL: yêu cầu thiết lập kiểu hoạt động theo tham số trong `arg`
  - `arg = O_NONBLOCK`
    - Recv hoặc send hoặc recvfrom hoặc sendto không block kể cả khi không có dữ liệu
  - `arg = O_ASYNC`
    - Một tín hiệu SIGIO được sinh ra mỗi khi socket thay đổi trạng thái.
  - Trả về:
    - Khác -1 nếu thiết lập thành công
    - -1 nếu có lỗi
- cmd = F_GETFL: yêu cầu đọc kiểu hoạt động của socket
  - `arg = 0`
  - Return: Giá trị phản ánh kiểu hoạt động
  - O_NONBLOCK: Non-blocking mode.
  - O_RDONLY: Open for reading only.
  - O_RDWR: Open for reading and writing.
  - O_WRONLY: Open for writing only.
  - O_ASYNC: Asynchronous mode with SIGIO signal generated whenever socket change status

Non-blocking send(), recv()

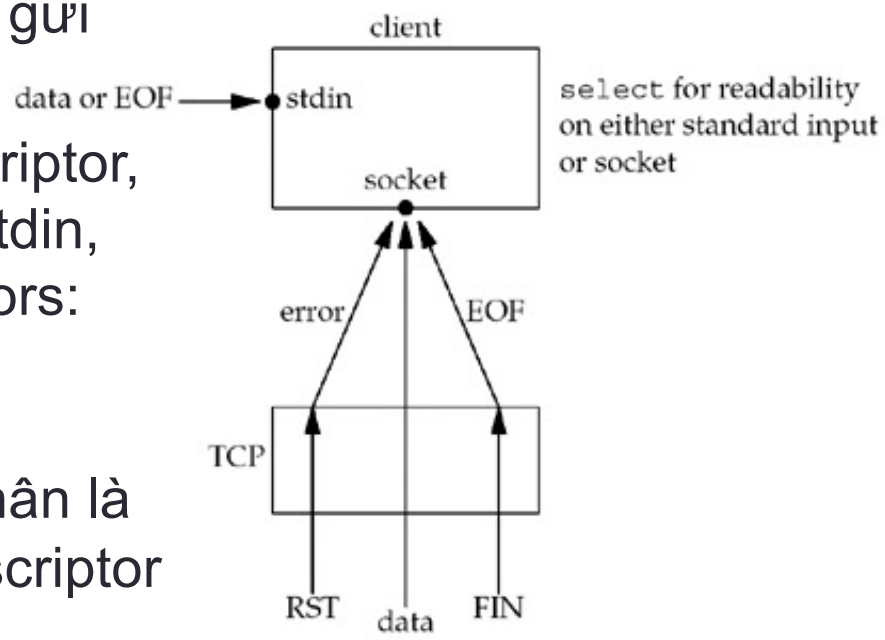
- Hàm hoàn thành ngay
- Nếu không có dữ liệu sẵn sàng ở socket:
  - Return giá trị -1
  - Đặt trạng thái biến ngoài *errno* =EWOULDBLOCK.
    - Cần khai báo #include <errno.h>
- Nếu có dữ liệu, trả về dữ liệu

Non-blocking send(), recv()

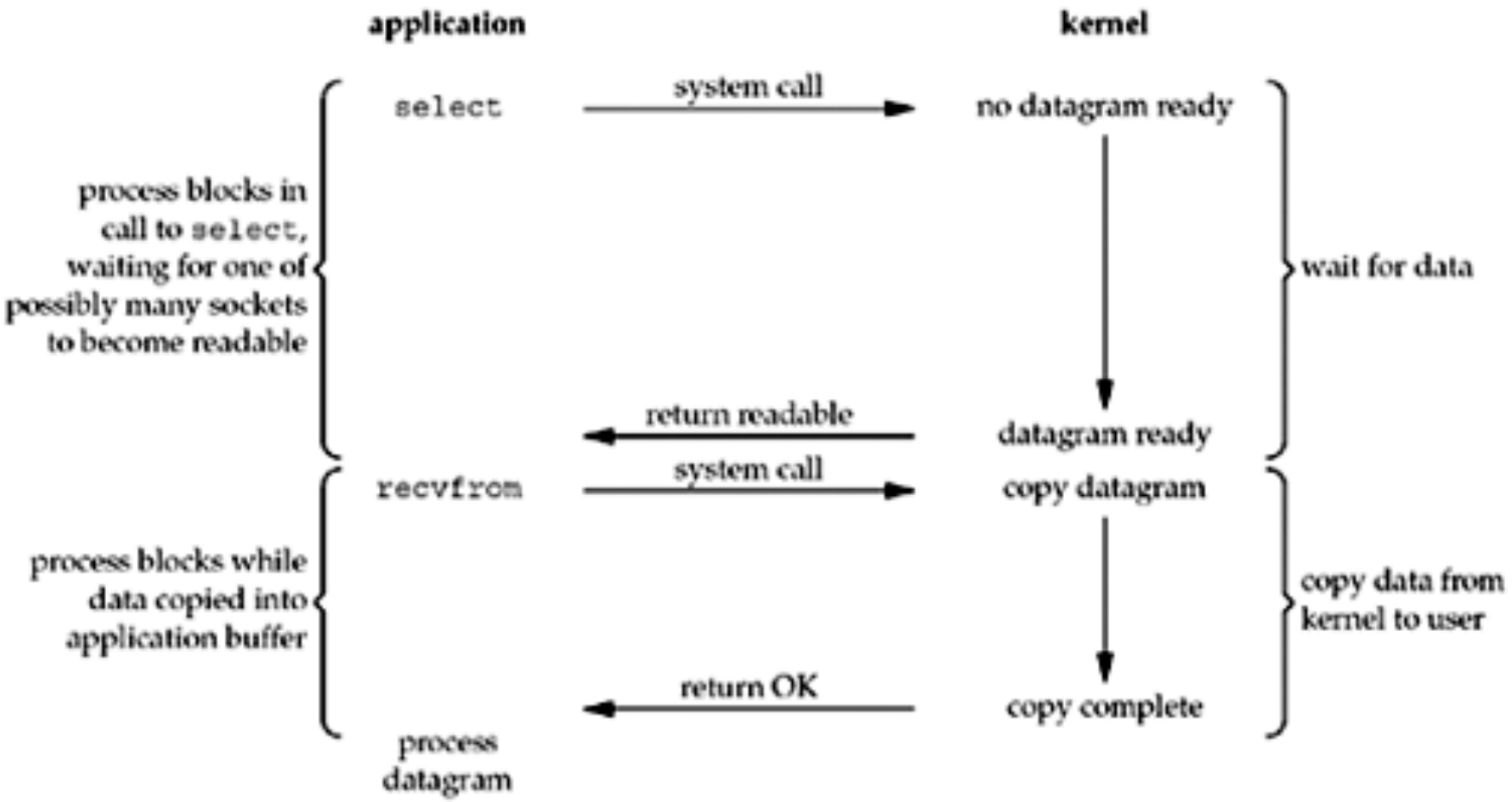
```
int val, sockfd;
char buff[1024];
fcntl(sockfd, F_SETFL, O_NONBLOCK);
while ((n = recv(sockfd, buff, sizeof(buff), 0) < 0)
{
      printf("read error on socket");
}
send(sockfd, buff, sizeof buff, 0));
```

I/O Multiplexing Model

- Cho phép 1 tiến trình làm việc với nhiều file descriptors (nguồn vào ra, sockets) để gửi hoặc nhận dữ liệu đồng thời.
- Mỗi tiến trình gán một số hiệu file descriptor, số ID, cho mỗi file hoặc luồng vào/ra(stdin, stdout) hoặc socket. Ví dụ file descriptors:
  - stdin: 0, stdout: 1, stderr:2.
  - Một socket: ví dụ có số hiệu 7
- Sử dụng select() để đăng ký với hạt nhân là tiến trình trờ chữ lieu từ một số file descriptor (socket, nguồn)
  - Hàm select hoạt động kiểu blocking.
- Khi dữ liệu sẵn sàng ở một file descriptor (socket), hàm select() thoát.
  - Khi đó có thể gọi recvfrom() để đọc dữ liệu chỉ từ socket đang có dữ liệu đến.



I/O Multiplexing Model: using select



select() function

- Hàm select cho phép một tiến trình (có gọi hàm select) đăng ký với hạt nhân đánh thức nó khi có một hoặc một số sự kiện vào ra xảy ra trên một số file descriptors hoặc thời gian chờ đợi (timeout) kết thúc
- Ví dụ, hạt nhân đánh thức tiến trình khi
  - File descriptors {0, 4, 5} có dữ liệu sẵn sàng để đọc
  - File descriptors {2, 7} có dữ liệu sẵn sàng để ghi ra
  - File descriptors {1, 4} có exception chờ xử lý
  - Timeout 10.2 giây đã hết hạn
- Hàm **select()** cung cấp một cách để kiểm tra đồng thời nhiều sockets (hay nói chung là file descriptor) để xem chúng có dữ liệu cần được **recv()**, hay **send()** hay có exception mà không cần dừng chờ như ở chế độ blocking.

select() function (2)

```
#include <sys/select.h>
int select(int maxfd, fd_set *readfds, fd_set *writefds,
                      fd_set *exceptfds, struct timeval *timeout);
```

- *maxfd* chỉ số file descriptor lớn nhất được theo dõi +1. Tương đương kích thước lớn nhất trong các mảng readfds, writefds, exceptfds.
- *readfds: mảng bit đánh dấu các FD cần được đọc dữ liệu*
- *writefds: mảng bit đánh dấu các FD cần được đợi để ghi dữ liệu*
- *exceptfds: mảng bit đánh dấu các FD đang chờ exception*
- *timeout:* thời gian hạt nhân chờ để để có một trong các FD có dữ liệu sẵn sàng. Có 3 cách đặc tả giá trị *timeout*
  - *Chờ mãi mãi*: timeout =NULL
  - *Chờ một khoảng cố định: timeout khác 0*
  - *Không chờ*: timeout =0
- Giá trị trả về (select) :
  - Số number of descriptors có dữ liệu,
  - 0 nếu timeout xảy ra
  - -1 nếu có lỗi
- Khi kết thúc hàm select, giá trị các mảng FD *readfds, writefds, exceptfds* sẽ được thay đổi đánh dấu các socket có dữ liệu.

fd_set

index	0	1	2	3	4	5
	0	0	0	0	1	0

- 3 *fd_set* are used to specify the descriptors that we want the kernel to test for reading, writing, and exception conditions.
- *A descriptor set* is a bit array with each bit corresponds to a FD. Ex: bit 5 corresponds to FD 4.
- All the implementation details are irrelevant to the application and are hidden in the *fd_set* datatype and 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 FD_ISSET(int fd, fd_set *fdset); /* Return true if fd is in the fdset */
```

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

Examples

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];
// pretend we've connected both to a server at this point s1 = socket(...); s2 = socket(...);
//connect(s1, ...)...   connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);
// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);
// since we got s2 second, it's the "greater", so we use that for the n param in select()
n = s2 + 1;
// wait until either socket has data ready to be recv()d (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);
if (rv == -1) {
        perror("select"); // error occurred in select() }
else if (rv == 0)
        {
                printf("Timeout occurred! No data after 10.5 seconds.\n");
        }
        else {
                // one or both of the descriptors have data
                if (FD_ISSET(s1, &readfds)) {
                        recv(s1, buf1, sizeof buf1, 0); }
                if (FD_ISSET(s2, &readfds)) {
                        recv(s1, buf2, sizeof buf2, 0); }
}
```

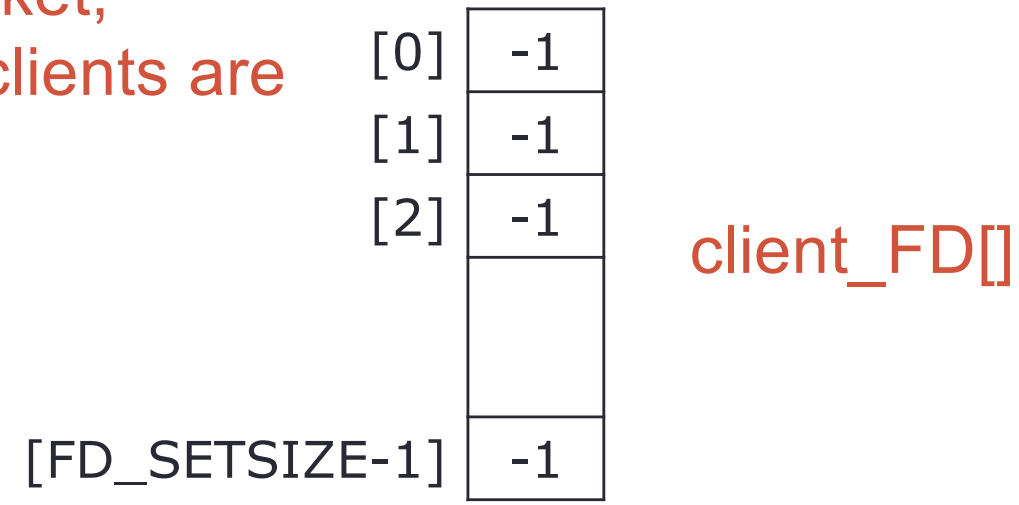
List all FD for watching in readfds.

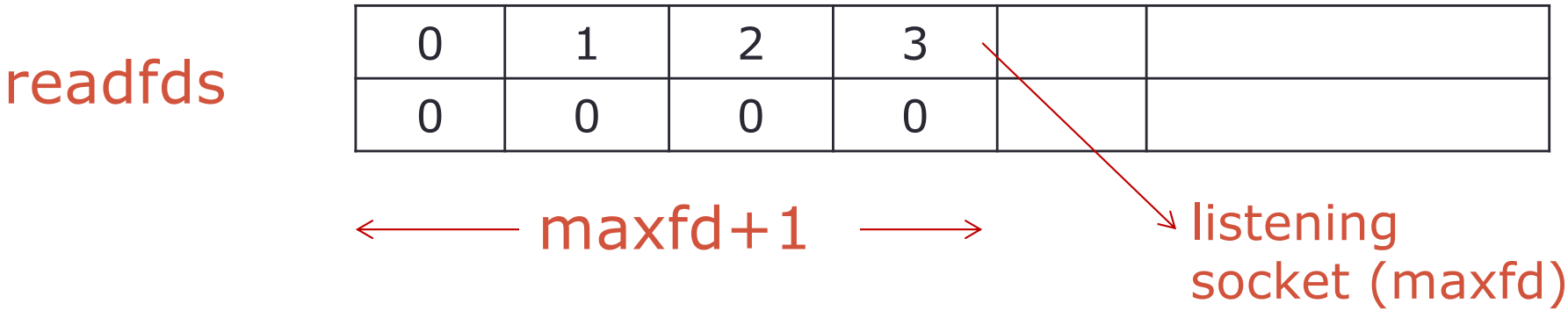
Call select to wait for FDs ready.

Browse all the FDs and read

How to use select() in TCP server

Data structures for TCP server with just a listening socket, connection sockets to clients are stored in client_FD[]





A single server process can handle all connections

How to use select() in TCP server

Data structures after first client connection is established

client_FD[]

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

1st accepting socket

readfds
(mark all sockets to be watched)

0	1	2	3	4	
0	0	0	1	1	

maxfd = 4+1

Chi so cao nhat

A single server process can handle all connections

How to use select() in TCP server

Data structures after first client connection is established

client_FD[]

[0]	4
[1]	7
[2]	-1
[FD_SETSIZE-1]	-1

2nd accepting socket

Chi so cao nhat

readfds
(mark all sockets to be watched)

0	1	2	3	4			7	
0	0	0	1	1			1	

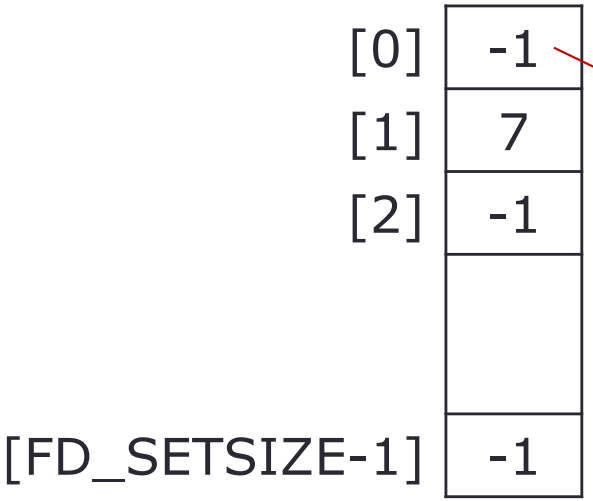
Maxfd = 7+1

A single server process can handle all connections

How to use select() in TCP server

Data structures after first client terminates its connection

client_FD[]



1st client terminates

readfds

0	1	2	3	4			7	
0	0	0	1	1			1	

Maxfd=7+1

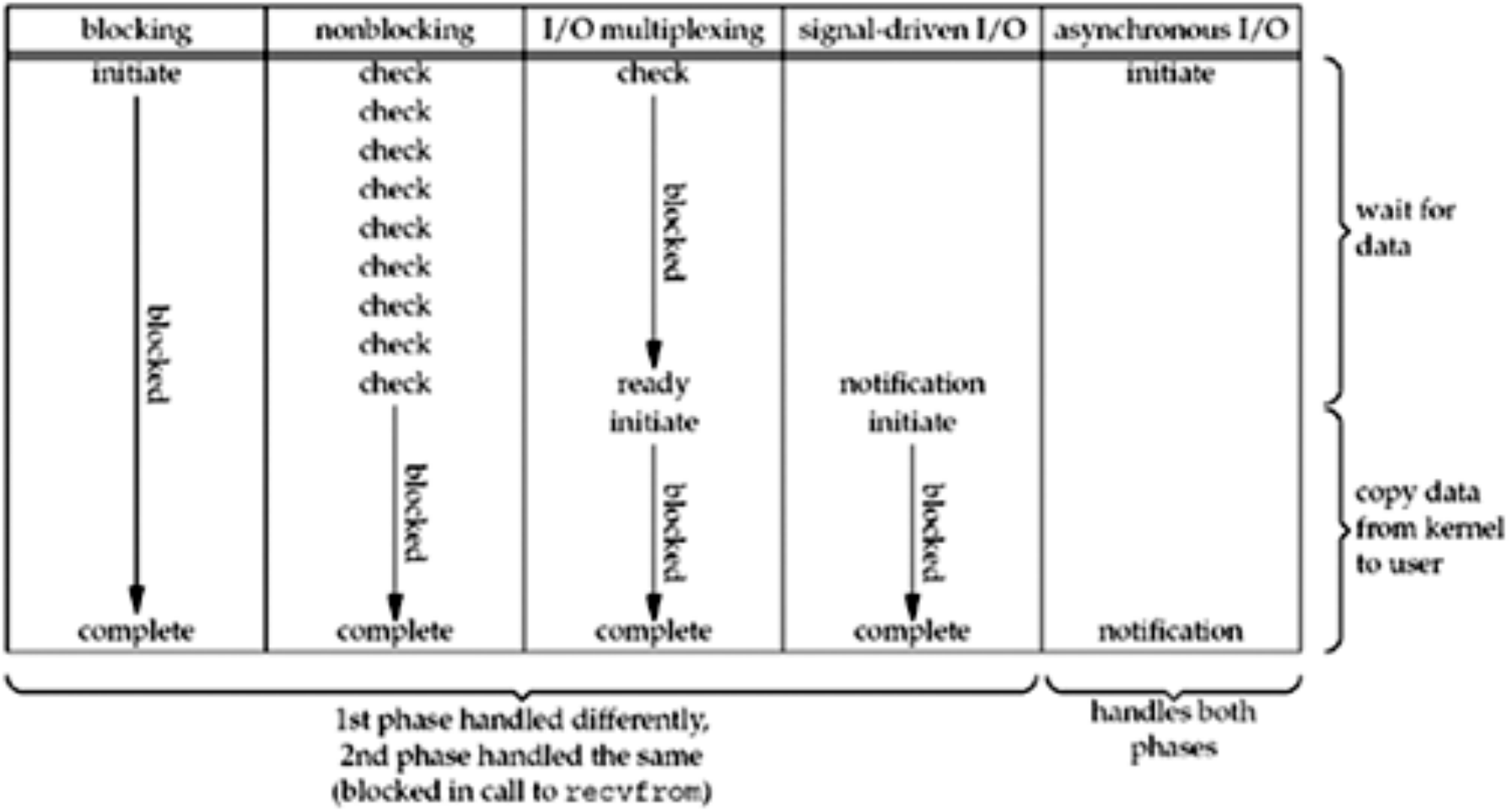
A single server process can handle all connections

Ex: handling both input/output source and socket

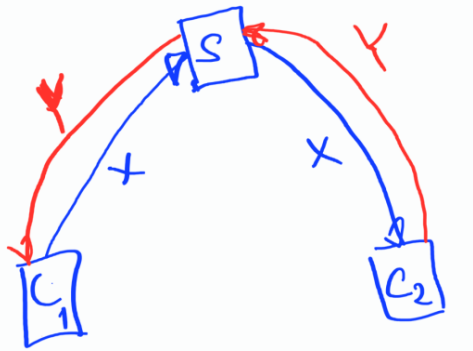
- fileno(File *f) return file descriptor associated with stream f.
  - fileno(stdin)
- Next program handes both reading from stdin and a socket.

```
36  int stdin_fd = fileno(stdin);
37   if (stdin_fd< sockfd)
38      n = sockfd+1;
39    else
40      n= stdin_fd+1;
41  while (1)
42  {
43     // clear the set ahead of time
44    FD_ZERO(&readfds);
45    // add our descriptors to the set
46    FD_SET(sockfd, &readfds); // from socket
47    FD_SET(stdin_fd, &readfds); //from stdio
48
49    int rv = select(n, &readfds, NULL, NULL, &tv);
50
51    if (rv == -1) {
52        perror("select"); // error occurred in select()
53    }
54    else if (rv == 0)
55        printf("Timeout occurred! No data after 10.5 seconds.\n");
56      else {
57        // one or both of the descriptors have data
58        if (FD_ISSET(stdin_fd, &readfds)) {
59            gets(str1);
60            sendto(sockfd, str1, strlen(str1), 0, (struct sockaddr *) &servaddr, sizeof(servad
61         }
62        if (FD_ISSET(sockfd, &readfds)) {
63            buf2[0]='\0';
64            recv(sockfd, buf2, sizeof buf2, 0);
65            printf("Receive from socket (remote client) :%s\n",buf2);
66          }
67      }
68   }
```

Comparison of the I/O Models



Exercise

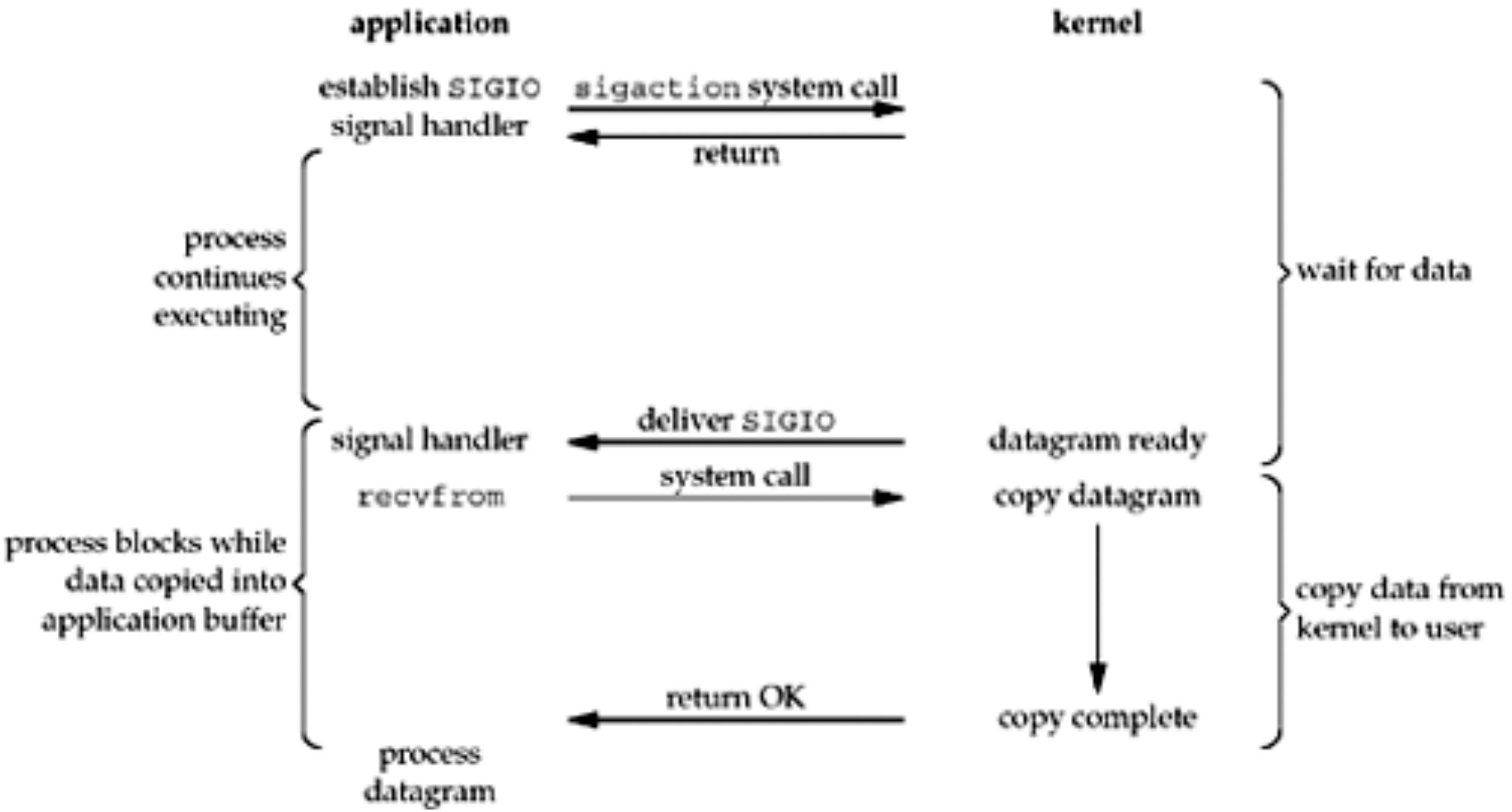


- Sửa ứng dung chat sử dụng UDP/TCP phía Client để client có thể gửi dữ liệu người dùng nhập vào lên server và nhận dữ liệu từ phía server một cách không đồng bộ, tức là có thể gửi nhiều tin nhắn khi không nhận tin nhắn nào hoặc ngược lại, nhận nhiều tin nhắn khi không gửi tin nhắn nào:
- Ví dụ:
- Client: Hôm nay
- Client: Trời mưa
- Server: abc
- Client:

Signal-Driven I/O Model

- Đặt socket ở chế độ Signal-Driven I/O.
- Mỗi khi có dữ lieu đến, một tín hiệu SIGIO sẽ sinh ra.
- Đăng ký một hàm bắt và xử lý tín hiệu SIGIO sinh ra.
  - Trong hàm thực hiện đọc dữ liệu từ socket, ví dụ recvfrom().
  - → No blocking

Signal-Driven I/O Model



Signal-Driven I/O Model

1. Đặt socket ở chế độ Signal-Driven I/O.

```
fcntl(sockfd, F_SETFL, O_ASYNC);
```

Mỗi khi trạng thái socket thay đổi (ví dụ do dữ lieu đến, đi), một tín hiệu SIGIO được sinh ra.

2. Đăng ký tiến trình sẽ nhận tín hiệu SIGIO sinh ra

- `fcntl(sockfd, F_SETOWN, pid);`
  - pid = process ID

3. Đăng ký một hàm (signal handler) trong chương trình của tiến trình (trong bước 2) sẽ xử lý tín hiệu SIGIO, hàm signal handler sẽ gọi recv(), recvfrom(), send(), sendto().

- → No blocking

Signal-Driven I/O Model

- Vấn đề: Khi sử dung giao thức TCP, tín hiệu SIGIO có thể được sinh ra bởi rất nhiều sự kiện:
  - Một kết nối được thực hiện tại socket nghe (listening socket)
  - Một yêu cầu hủy kết nối được gửi đến
  - Một yêu cầu hủy kết nối được hoàn thành
  - Một chiều của kết nối đã được đóng
  - Có dữ liệu đến một socket
  - Có dữ liệu được gửi ra khỏi socket (i.e., vùng đệm luồng dữ liệu ra của socket đã được giải phóng)
  - Một lỗi không đồng bộ xảy ra.

  - ➔ Chỉ có sự kiện mầu đỏ là sự kiện ta muốn xử lý.

Ví dụ: trong chương trình chính

```
// Signal driven I/O mode and NONBlOCK mode so that recv will not b
if(fcntl(client_sock_fd, F_SETFL, O_NONBLOCK|O_ASYNC))
    printf("Error in setting socket to async, nonblock mode");

signal(SIGIO, signio_handler); // assign SIGIO to the handler

//set this process to be the process owner for SIGIO signal
if (fcntl(client_sock_fd, F_SETOWN, getpid()) <0)
    printf("Error in setting own to socket");

char str[50];
while (1)
    {
      printf("Client: ");
      gets(str);
      send(client_sock_fd, str, sizeof(str),0);
    }
```

Ví dụ: hàm xử lý tín hiệu SIGIO

```
int client_sock_fd;
¶
void signio_handler(int signo)¶
{¶
  char buff[1024];¶
  int n = recv(client_sock_fd, buff, sizeof buff, 0);¶
  if (n>0) // if SIGIO is generated by a data arrival¶
    printf("Received from server (%d bytes), content: %s\n",n, buff);¶
}¶
```

Socket options

- Có nhiều cách để thay đổi tùy chọn cho một socket
  - fcntl()
  - ioctl()
  - getsockopt(), setsockopt()

Socket options

#include <sys/types.h>

#include <sys/socket.h>

*int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen)*

*int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);*

- Arguments
  - *sockfd* : socket number
  - *level*: socket code or protocol code: socket level, transport level, ip level
  - *optname*: specifics option
  - *optval:* a pointer to a variable storing the option value. (new value for setsockopt; current value for getsockopt)
  - *optlen*: size of optval
- Return:
  - 0: succesful
  - -1: error

level	*optname*	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Join or leave multicast group		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Block or unblock multicast source		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Join or leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6{}
	IPV6_PATHMTU	•		Retrieve current path MTU		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_*XXX*	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq{}
IPPROTO_IP *or* IPPROTO_IPV6	MCAST_JOIN_GROUP		•	Join multicast group		group_req{}
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req{}

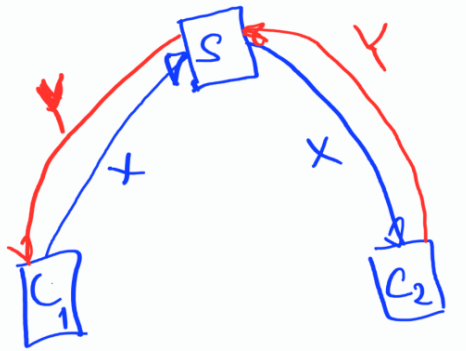
Socket option at transport layer

level	*optname*	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 addresses	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

Example

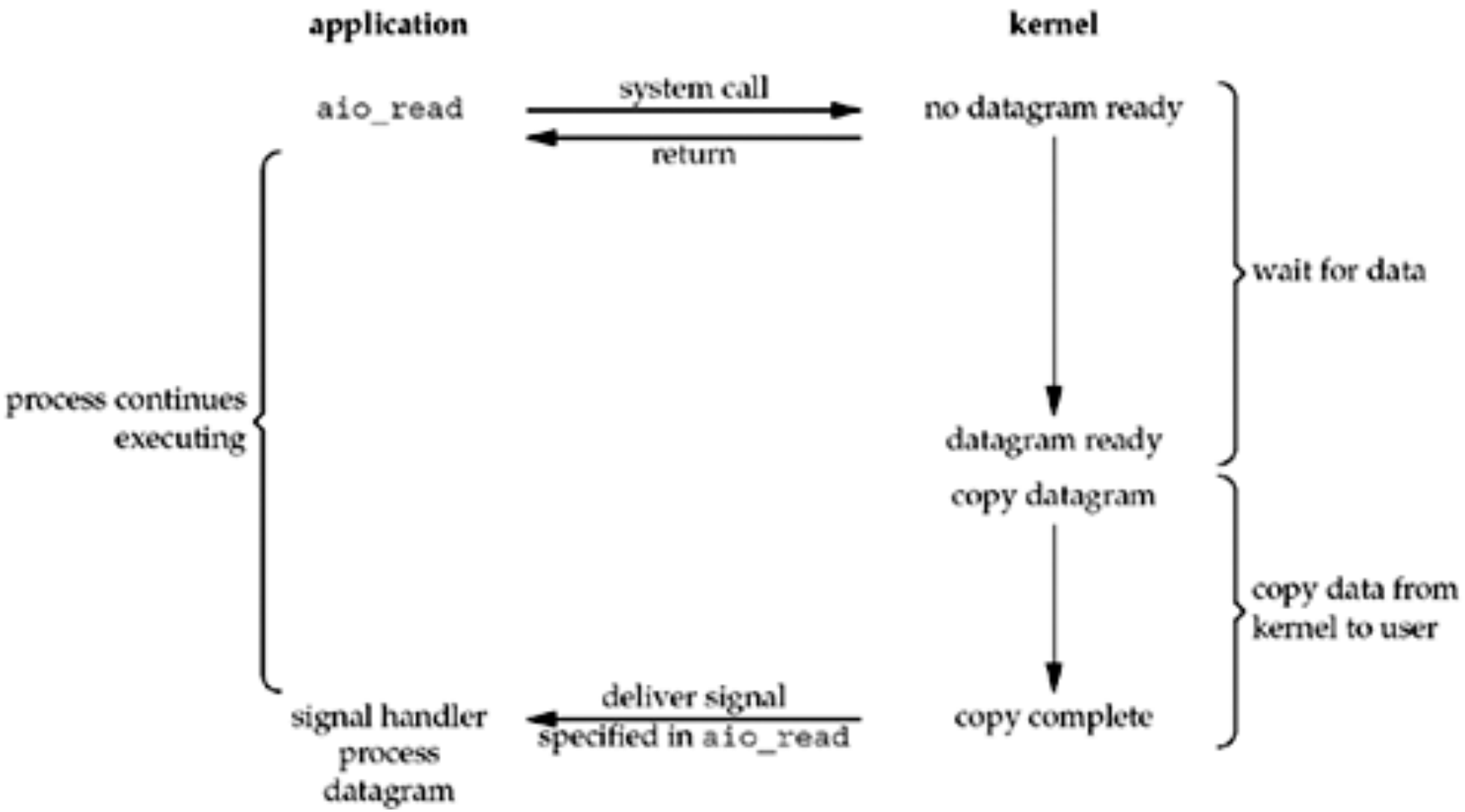
```
int optval;
int optlen;
char *optval2;
// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval); //
bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);
// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0)
        print("SO_BROADCAST enabled on s3!\n");
```

Exercise



- Revise echoServer and the echoClient so that if there are only 2 clients then:
  - When client 1 sends something to server, server forwards this information to client 2
  - When client 2 sends something to server, server forwards this information to client 1
  - One client can send a string to server anytime independently with the reception of incoming data. (similar to chat)
- Hint:
  - Option 1: doing Non-blocking model, one process on client check alternatively data to send and receive. May still have problem with reading data from user in the same time.
  - Option 2: IO multiplexing with select()
  - Option 3: On the client side, associate SIGIO with a function which calls recvfrom() to receiving data uniquely when data arrives.

Asynchronous I/O Model



Asynchronous I/O Model

- Asynchronous I/O is defined by the POSIX specification

- These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
  - Different to signal-driven I/O model
  - In signal-driven I/O model, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete

Asynchronous I/O Model (2)

- Call aio_read
  - POSIX asynchronous I/O functions begin with aio_
- Function asks kernel to start waiting for data and notifies when data is ready in buffer.
- Pass the kernel
  - the descriptor
  - buffer pointer
  - buffer size (the same three arguments for read)
  - buffer offset (similar to lseek)
  - how to notify us when the entire operation is complete
- This system call returns immediately
  - No-blocking while waiting for the I/O to complete.