MULTI-PROCESS TCP SERVER(CONT)

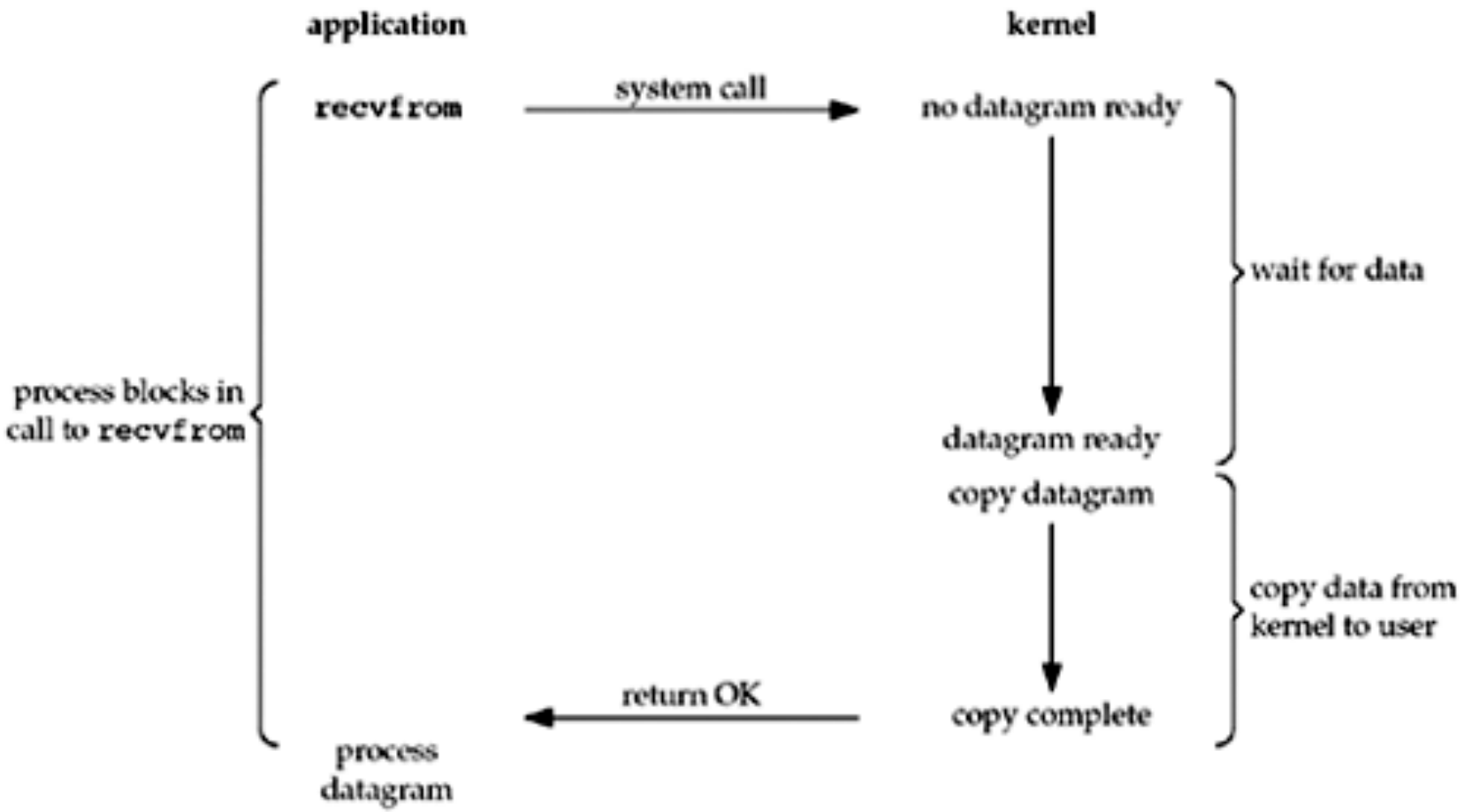
Content

- I/O Models
  - Non-blocking I/O model
  - I/O Multiplexing using select ()
  - Signal driven I/O model (SIGIO)
- Socket options

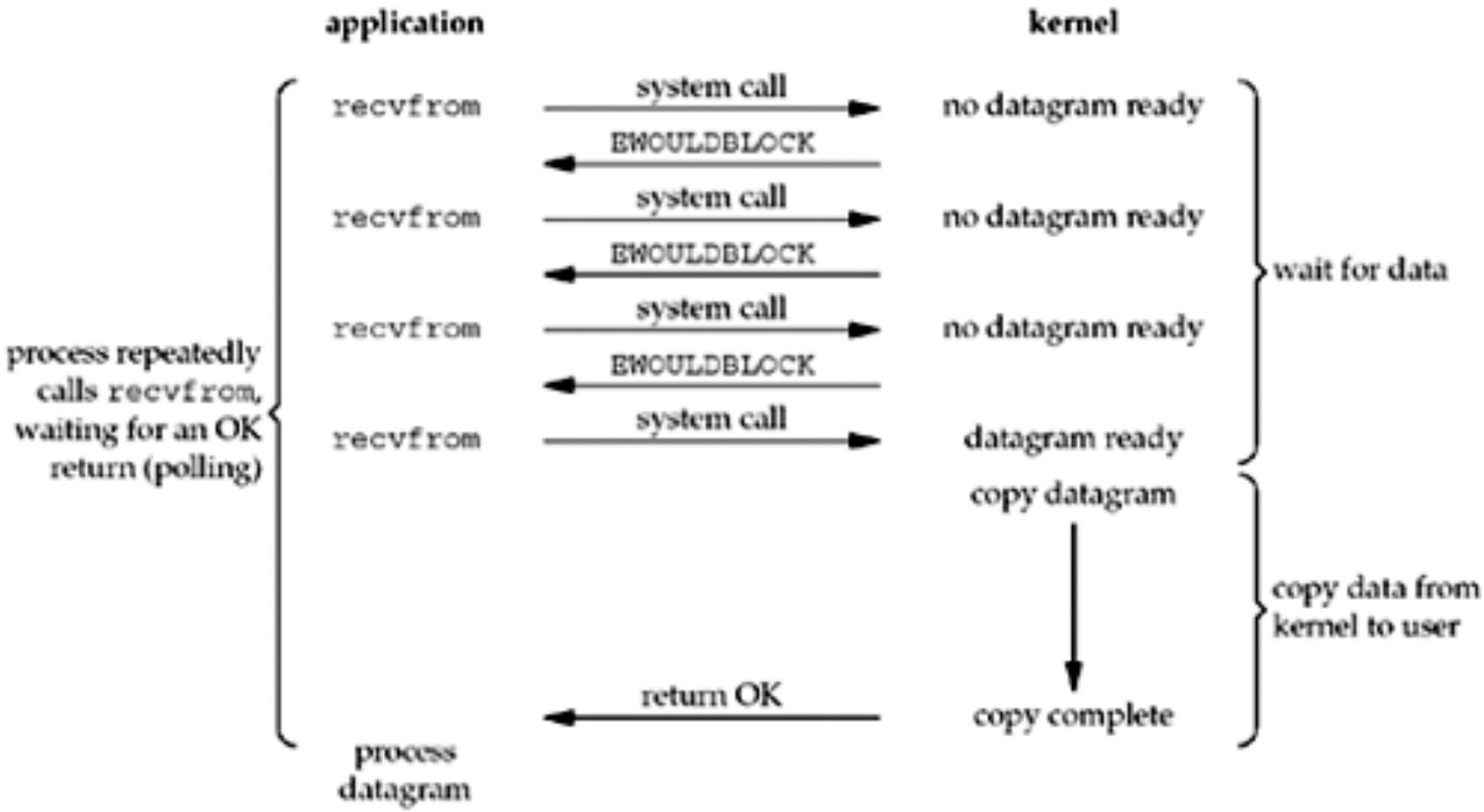
I/O Models

- The basic differences in the five I/O models that are available
  - blocking I/O
  - nonblocking I/O
  - I/O multiplexing (select and poll)
  - signal driven I/O (SIGIO)
  - asynchronous I/O (the POSIX aio_functions)

Blocking I/O Model



Non-blocking I/O Model



Non-blocking I/O model

- Need to set socket to non-blocking type.
- When we call recv() or recvfrom() there is no data and the system return immediately with error EWOULDBLOCK
- Otherwise it returns OK
- We can poll to read data
  - Loop with recv(), recvfrom()

Non-blocking socket

- By default, sockets are blocking:
  - Input operations: `recv()`
  - Output operations: `send()`
  - Accepting incoming connection: `accept()`
    - Solved by `select`
  - Initiating outgoing connections: `connect()`

Make socket 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
  - `arg`: required argument
- Return: depends on `cmd`

fcntl(): operations

- cmd =F_SETFL: Set the file status flags to the value specified by `arg`
  - `arg = O_NONBLOCK`
    - Recv or send or recvfrom, sendto will not block even if data are not ready
  - `arg = O_ASYNC`
    - A signal SIGIO is generated whenever socket change status.
  - Return:
    - other than -1 on success,
    - -1 on error
- cmd = F_GETFL: Get the file status flags and file access modes
  - `arg = 0`
  - Return: Value of file status flags
- File status flags:
  - 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()

- Functions return immediately
- If no messages are available at the socket:
  - Return value -1
  - External variable *errno* is set to EAGAIN or EWOULDBLOCK.
    - Need to #include <errno.h>
- Otherwise return any data available, up to the requested amount

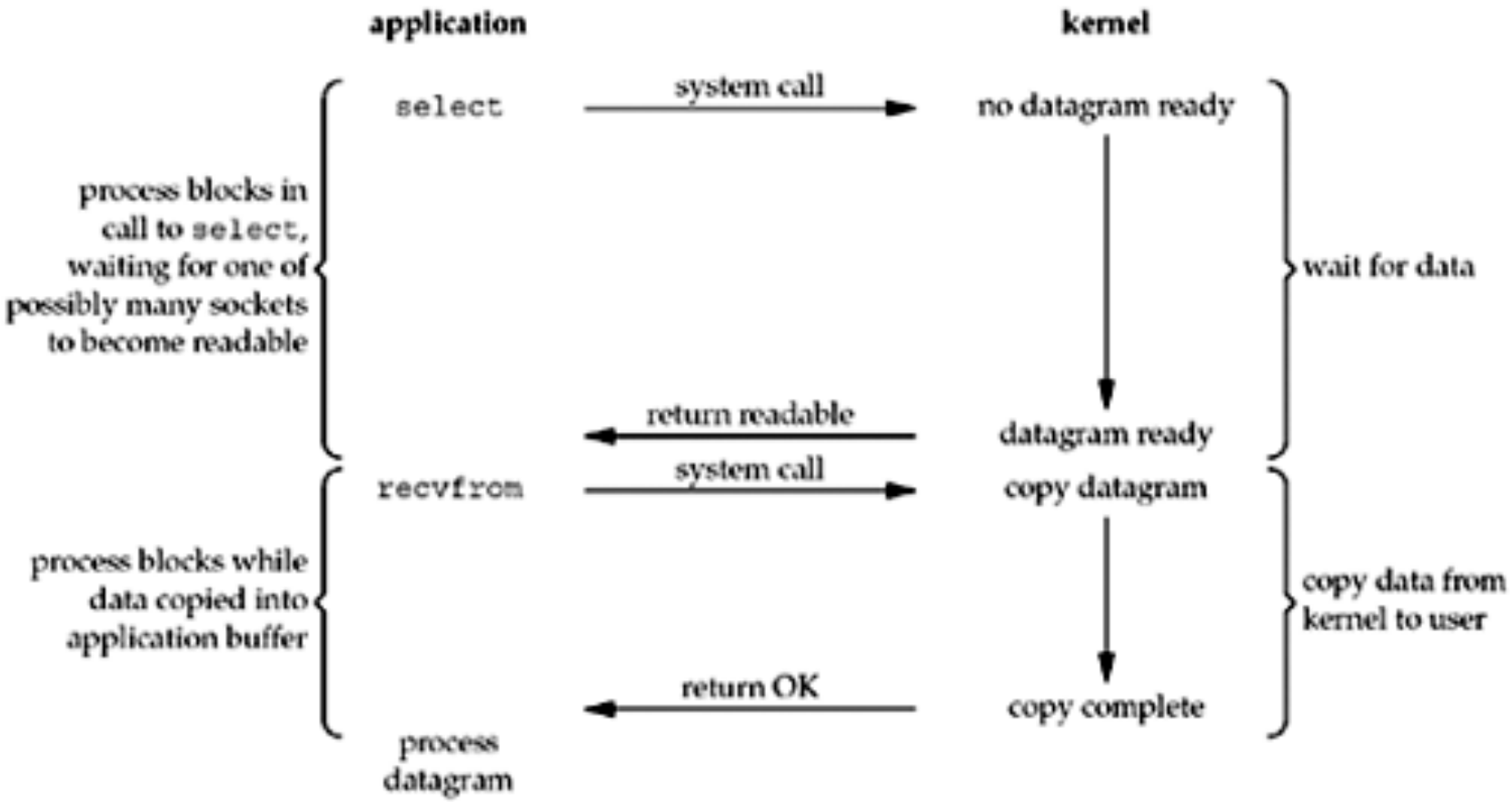
Non-blocking send(), recv()

```
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

- When the TCP client is handling two inputs at the same time: standard input and a TCP socket, we encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed.
- We want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called I/O multiplexing and is provided by the select and poll functions ..
- Use select() to wait for data from several sockets
  - It is a blocking function.
- When data is ready in one socket then select returns
- We can then use recvfrom() to read from the chosen socket.

I/O Multiplexing Model: using select



select() function

- This function allows the process to instruct the kernel to wake up the process only when one or more of events occurs or when a specified amount of time has passed.
- Exp : kernel to return only when
  - {1, 4, 5} are ready for reading
  - {2, 7} are ready for writing
  - {1, 4} have an exception condition pending
  - 10.2 seconds have elapsed
- The **select()** function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be **recv()**, or if you can **send()** data to them without blocking, or if some exception has occurred.

select() function (2)

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

- *maxfd* is the highest-numbered file descriptor in any of the three sets, plus 1.
- *readfds: set of FD to wait to read from*
- *writefds: set of FD to wait to write to*
- *exceptfds: set of FD to wait for exception*
- *timeout:* how long kernel need to wait for one of the specified descriptors to become ready. There are three types of using *timeout*
  - *Wait forever* : timeout =NULL
  - *Wait up to a fixed amount of time*
  - *Do not wait at all* : timeout =0
- Return value (select) :
  - the number of descriptors in the set on success,
  - 0 if the timeout was reached
  - -1 on error
- On exit, the FD sets are modified in place to indicate which file descriptors actually changed status

fd_set

- 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

client_FD[]

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

readfds

0	1	2	3		
0	0	0	0		

← maxfd+1 →

listening socket (maxfd)

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

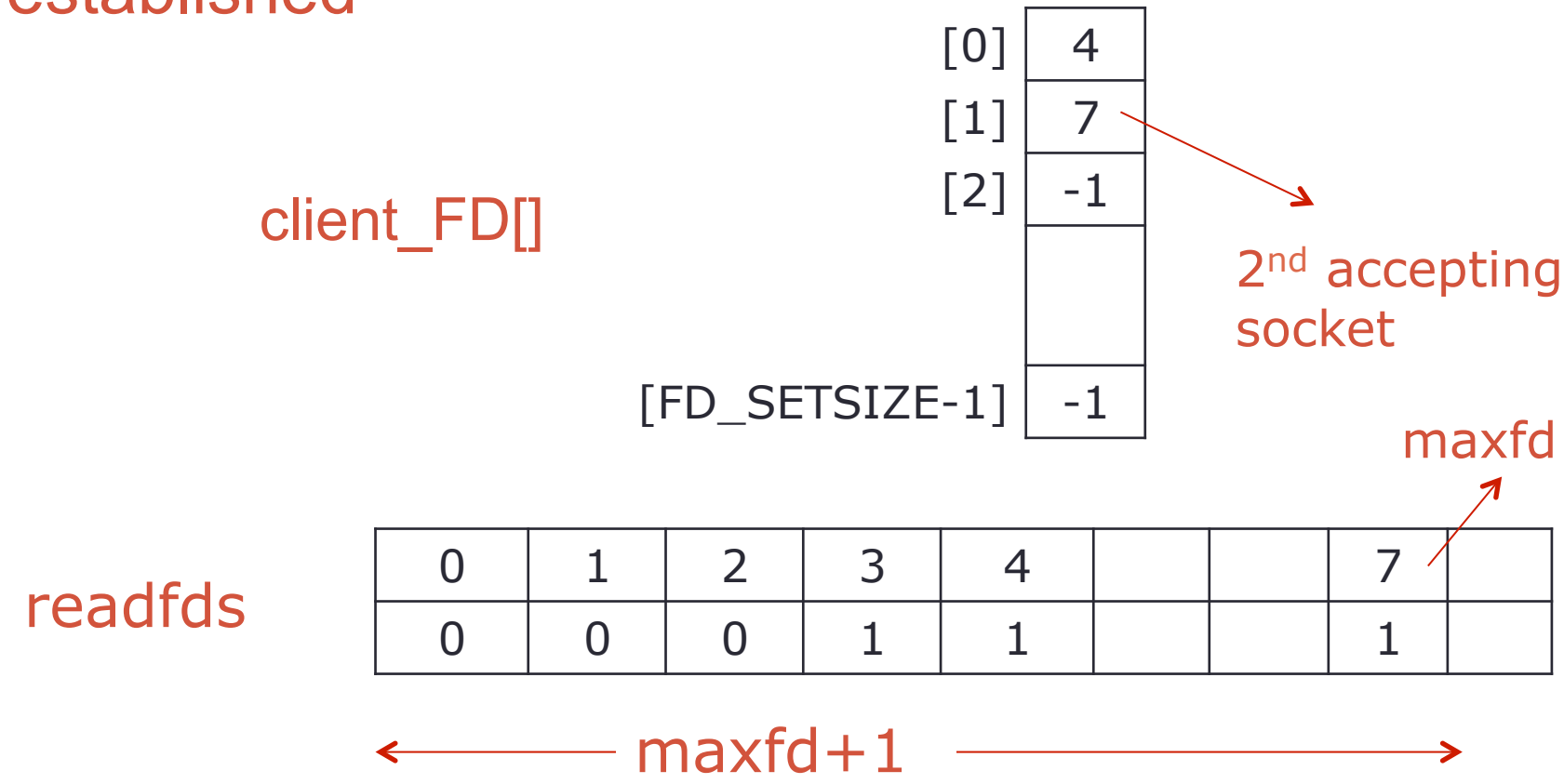
0	1	2	3	4	
0	0	0	1	1	

maxfd+1

maxfd

How to use select() in TCP server

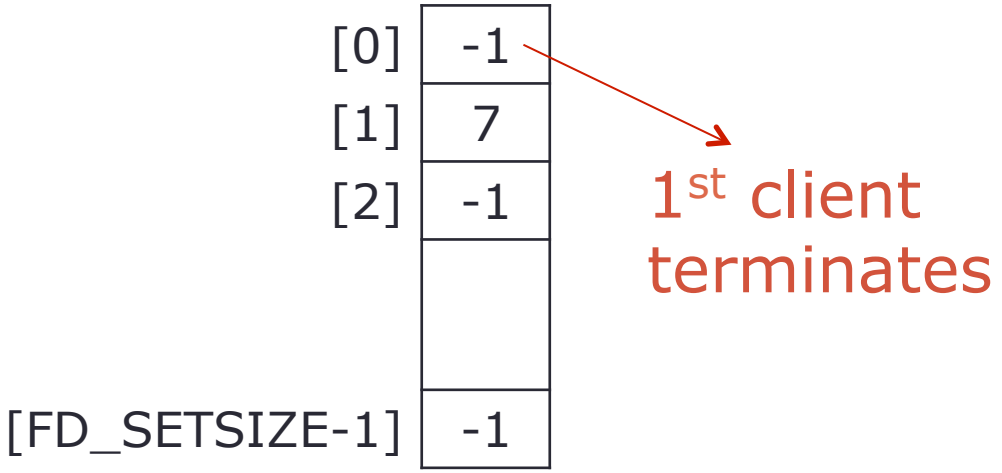
Data structures after first client connection is established



How to use select() in TCP server

Data structures after first client terminates its connection

client_FD[]



readfds

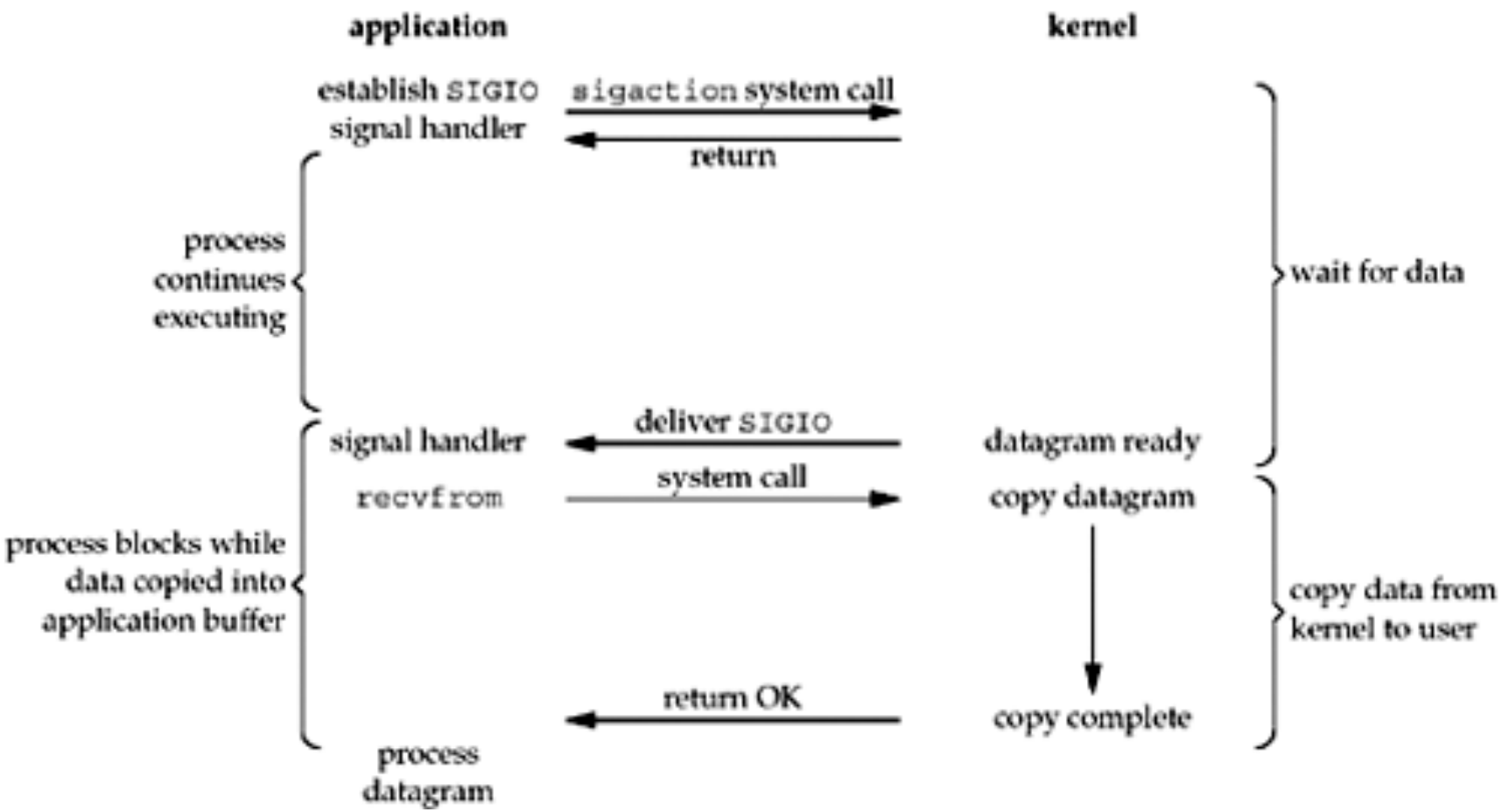
0	1	2	3	4			7	
0	0	0	1	1			1	

maxfd+1

Signal-Driven I/O Model

- Set socket to Signal-Driven I/O mode.
- When data arrive, a SIGIO occurs.
- Solution:
  - Associate SIGIO with a signal handler
  - When SIGIO occurs read data by recvfrom().
  - → No blocking

Signal-Driven I/O Model



Signal-Driven I/O Model

- Must set socket to Signal-Driven I/O mode.
  - `fcntl(sockfd, F_SETFL, O_ASYNC);`
- Whenever the socket change status a signal SIGIO is generated
- Must assign a process to receive the SIGIO signal
  - `fcntl(sockfd, F_SETOWN, pid);`
    - pid is the process ID
- Must associate SIGIO with a signal handler which can call recv(), recvfrom(), send(), sendto().
- → No blocking

Signal-Driven I/O Model

- Issue: In TCP, SIGIO signal can be generated by many different events:
  - A connection request has completed on a listening socket
  - A disconnect request has been initiated
  - A disconnect request has completed
  - Half of a connection has been shut down
  - Data has arrived on a socket
  - Data has been sent from a socket (i.e., the output buffer has free space)
  - An asynchronous error occurred

Example: main function

```
¶
// 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);¶
    }¶
¶
```

Example: SIGIO handling function

```
#include <stdlib.h>¶
#include <stdio.h>¶
#include <netinet/in.h>¶
#include <sys/types.h>¶
#include <sys/socket.h>¶
#include <arpa/inet.h>¶
#include <string.h>¶
#include <unistd.h>¶
#include <fcntl.h>¶
#include <errno.h>¶
#include <signal.h>¶
¶
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

- There are various ways to set socket option
  - 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{}

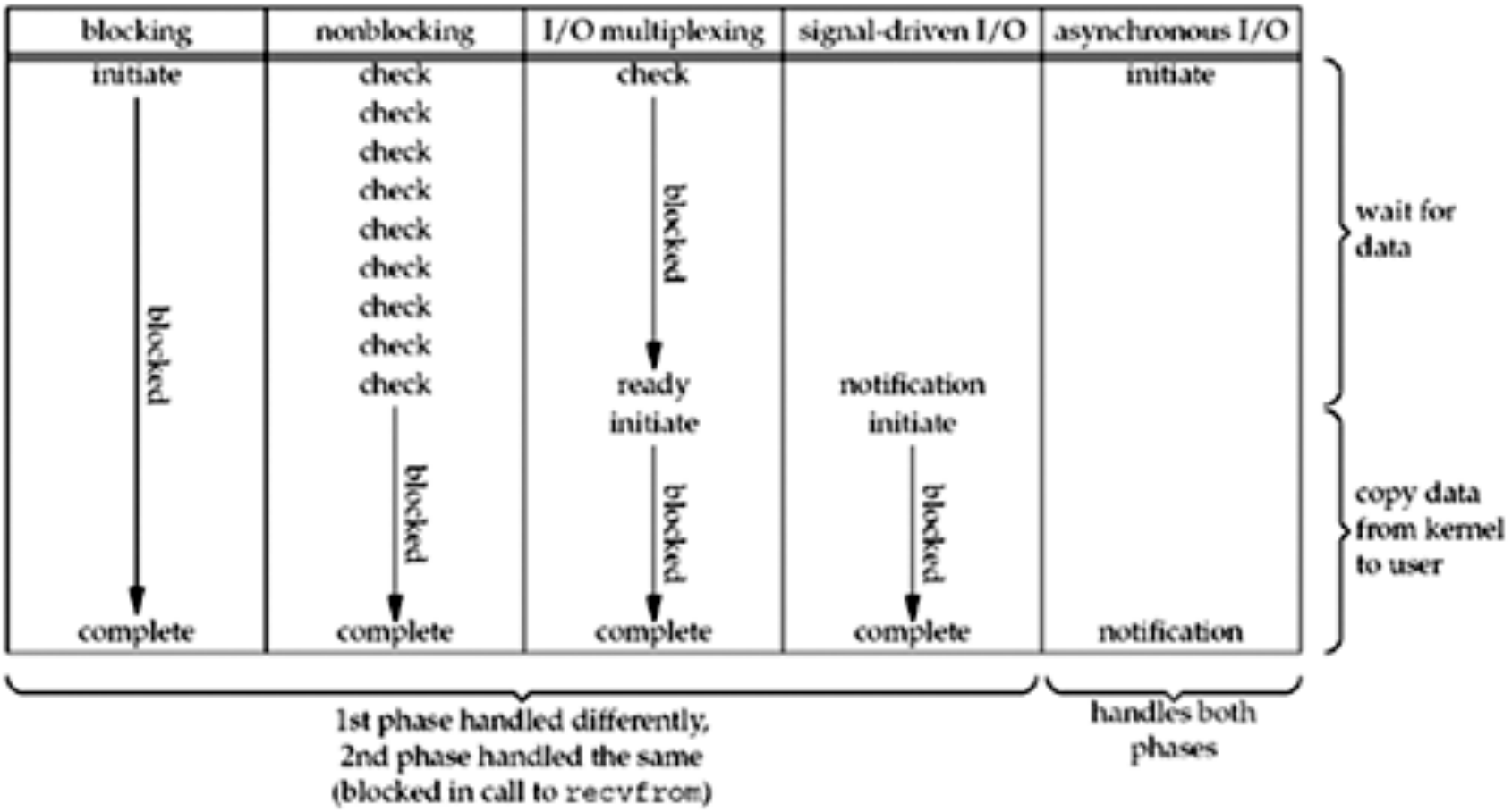
Example

```
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");
```

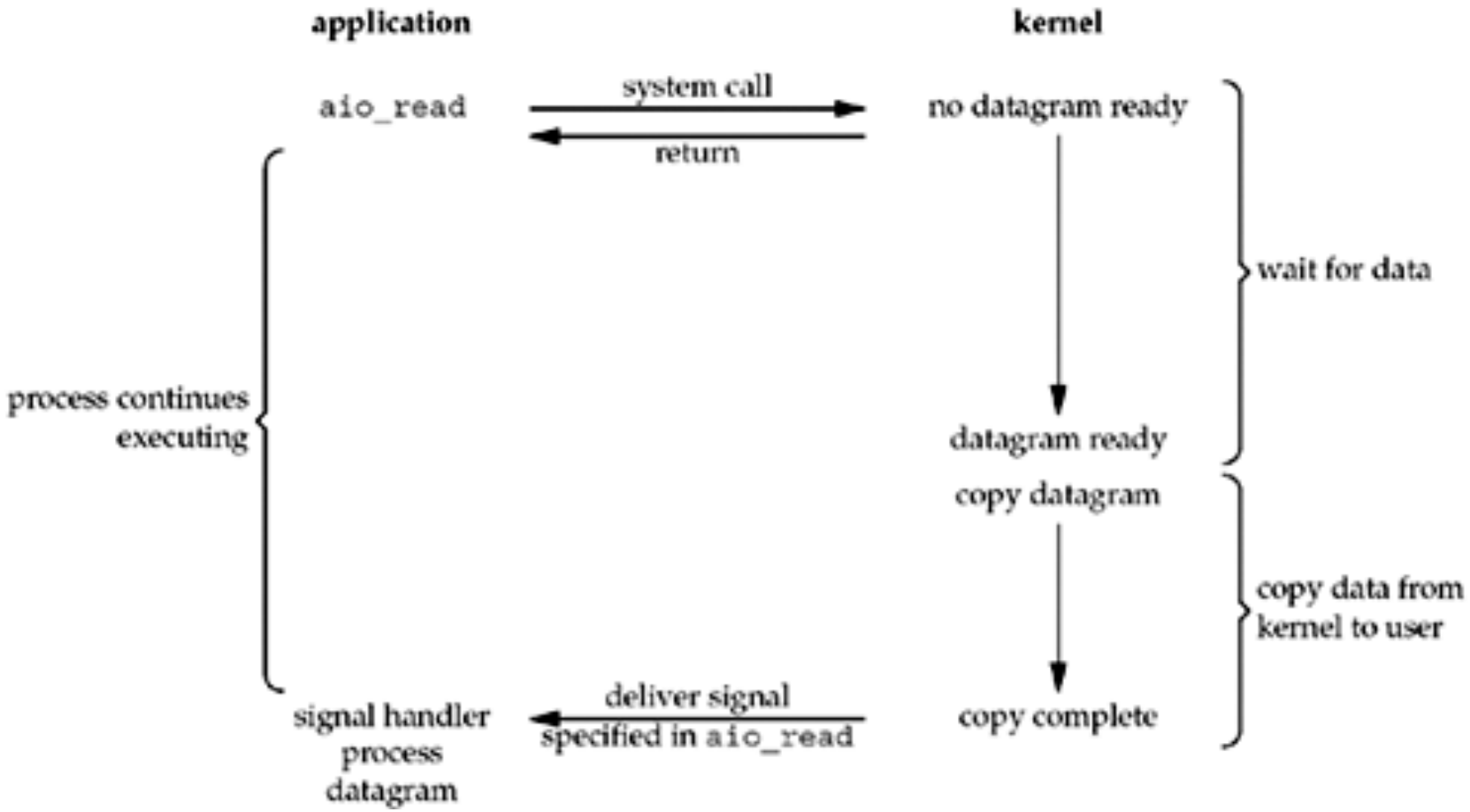
Exercise

- 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
  - Option 2: IO multiplexing
  - Option 3: On the client side, associate SIGIO with a function which calls recvfrom() to receiving data uniquely when data arrives.

Comparison of the I/O Models



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
  - 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.