Lecture 3 Overview

- Last Lecture
  - TCP socket and Client-Server example
  - Source: Chapters 4&5
- This Lecture
  - I/O multiplexing and Socket Options
  - Source: Chapters 6 & 7
- Next Lecture
  - Name Address Conversion & IPv6
  - Source: Chapters 11&12

Problems from Last Time

- Client could be blocked in fgets and miss data from readline.
- Sending and receiving data should be independent.

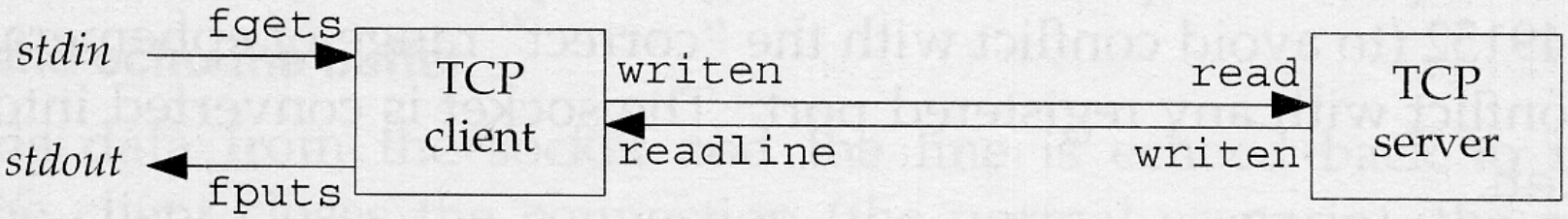


Figure 5.1 Simple echo client and server.

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

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 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

Blocking I/O

- Process is put to sleep if blocked

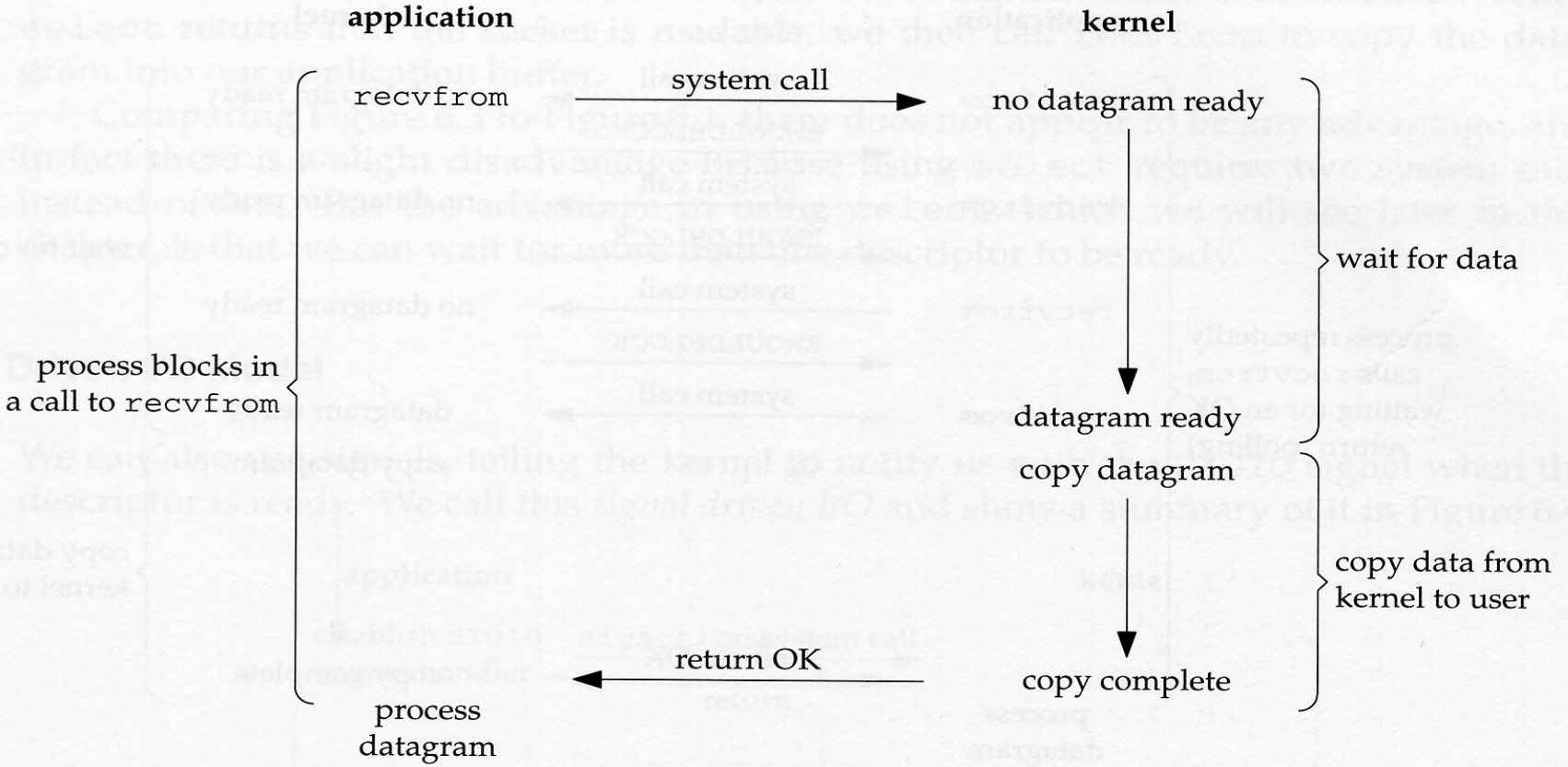


Figure 6.1 Blocking I/O model.

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

Polling

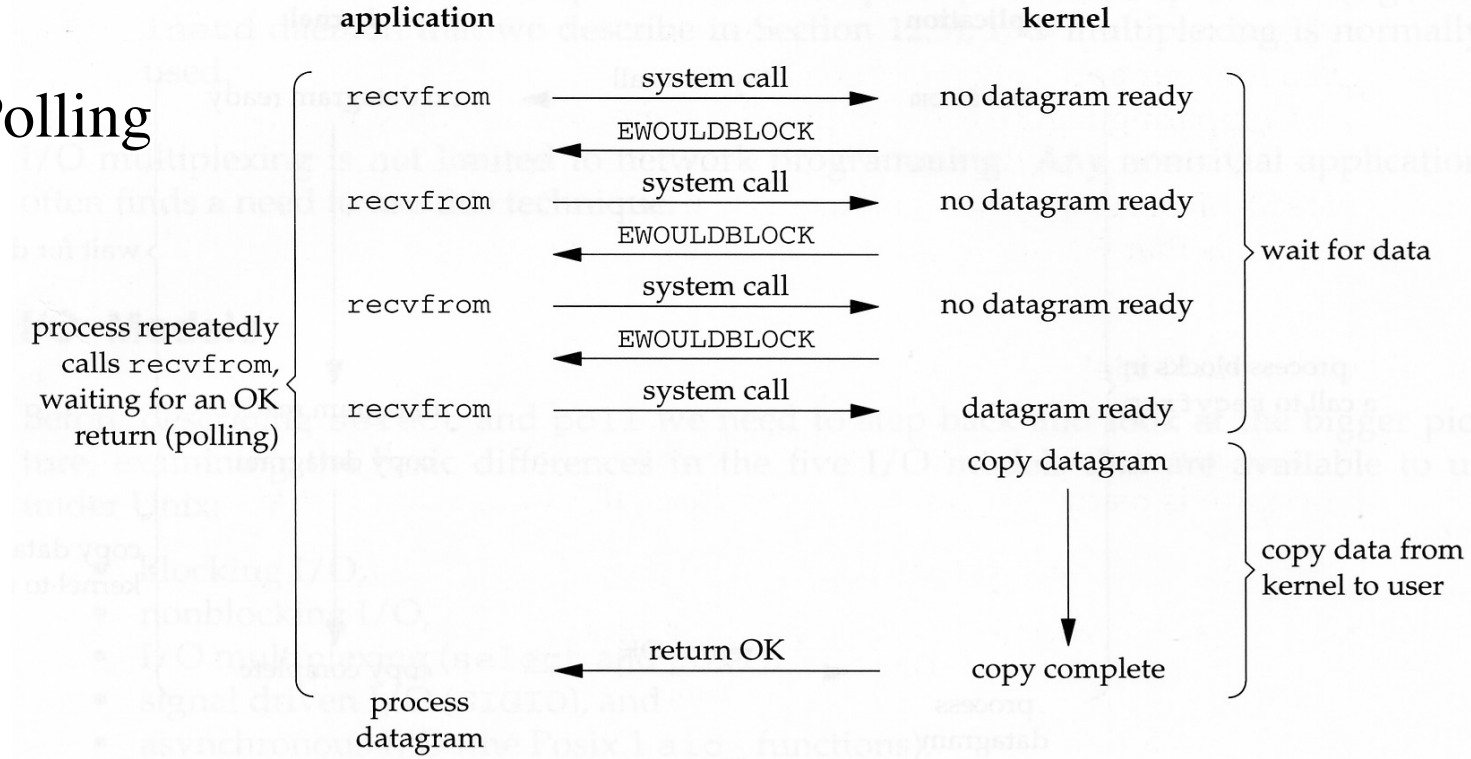


Figure 6.2 Nonblocking I/O model.

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.

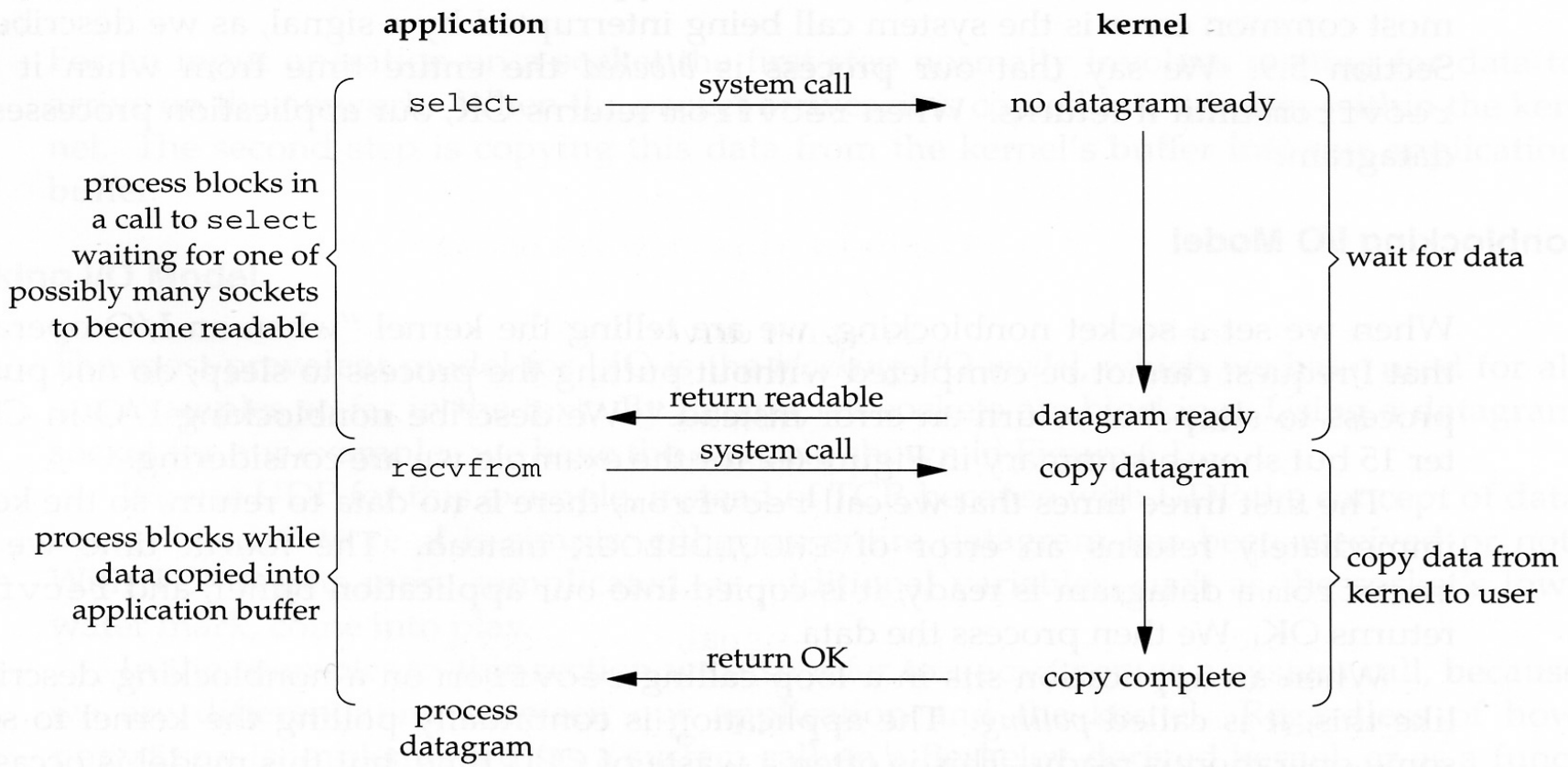


Figure 6.3 I/O multiplexing model.

Signal driven I/O

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

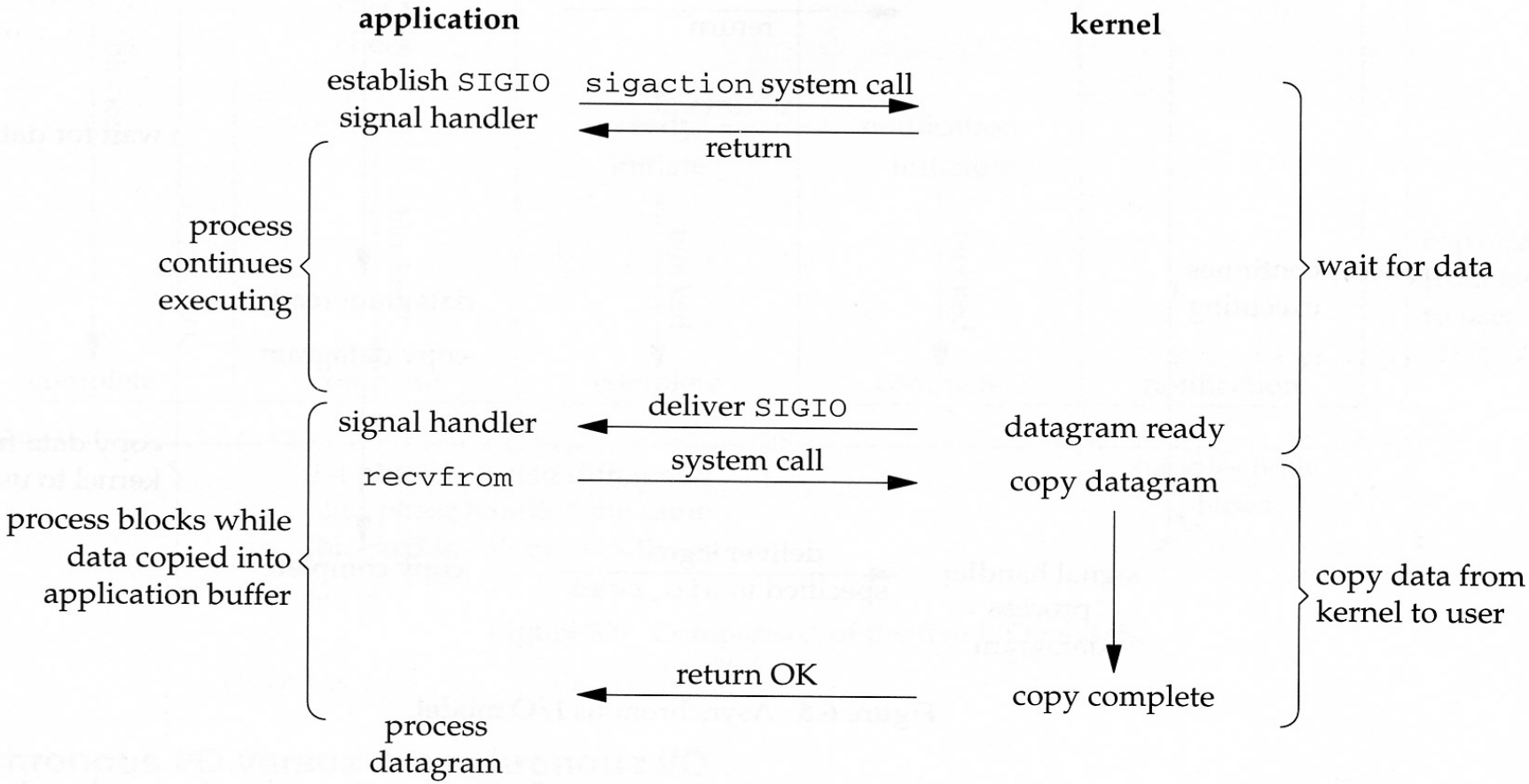


Figure 6.4 Signal Driven I/O model.

Asynchronous I/O

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

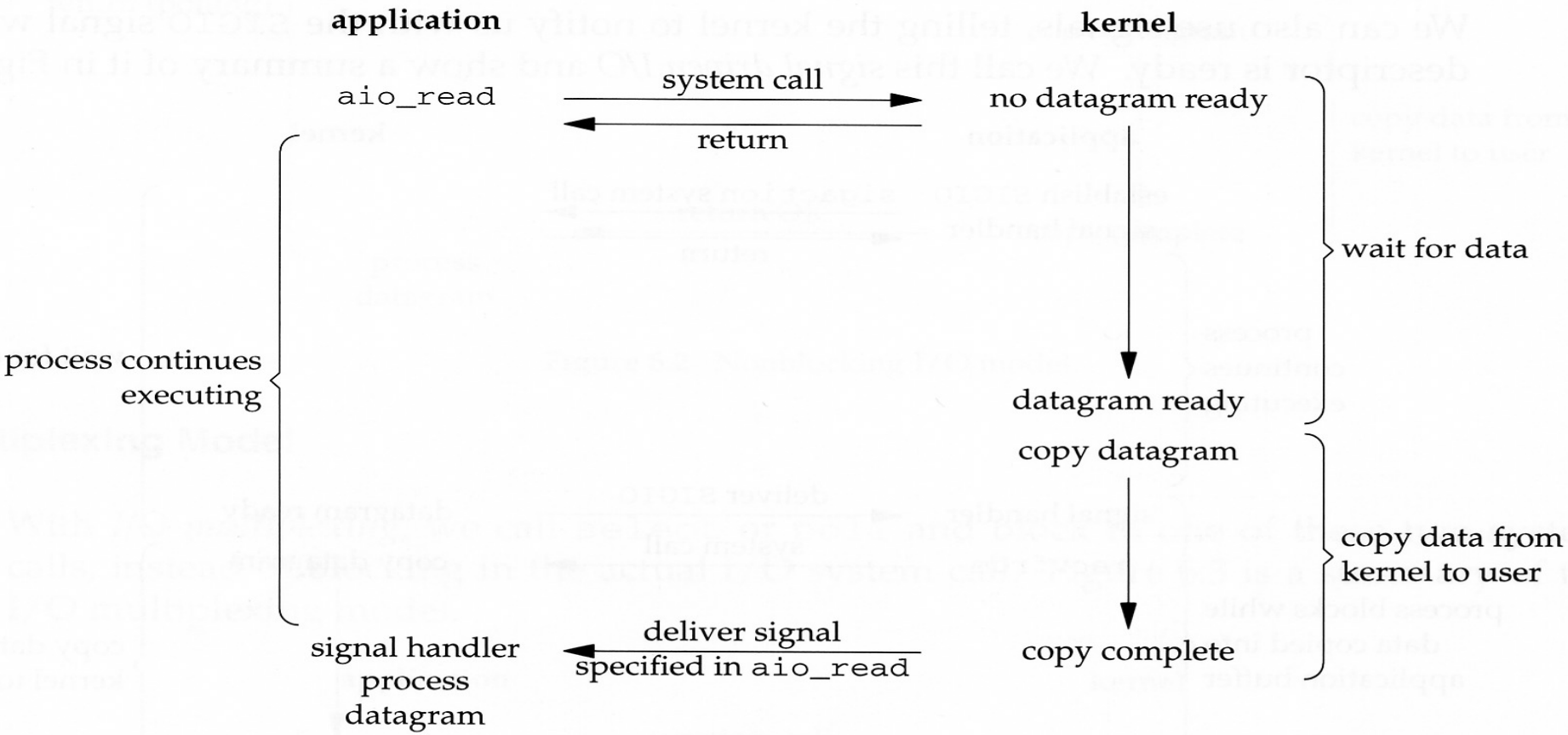


Figure 6.5 Asynchronous I/O model.

Comparison of I/O models

- The first four are synchronous I/O.

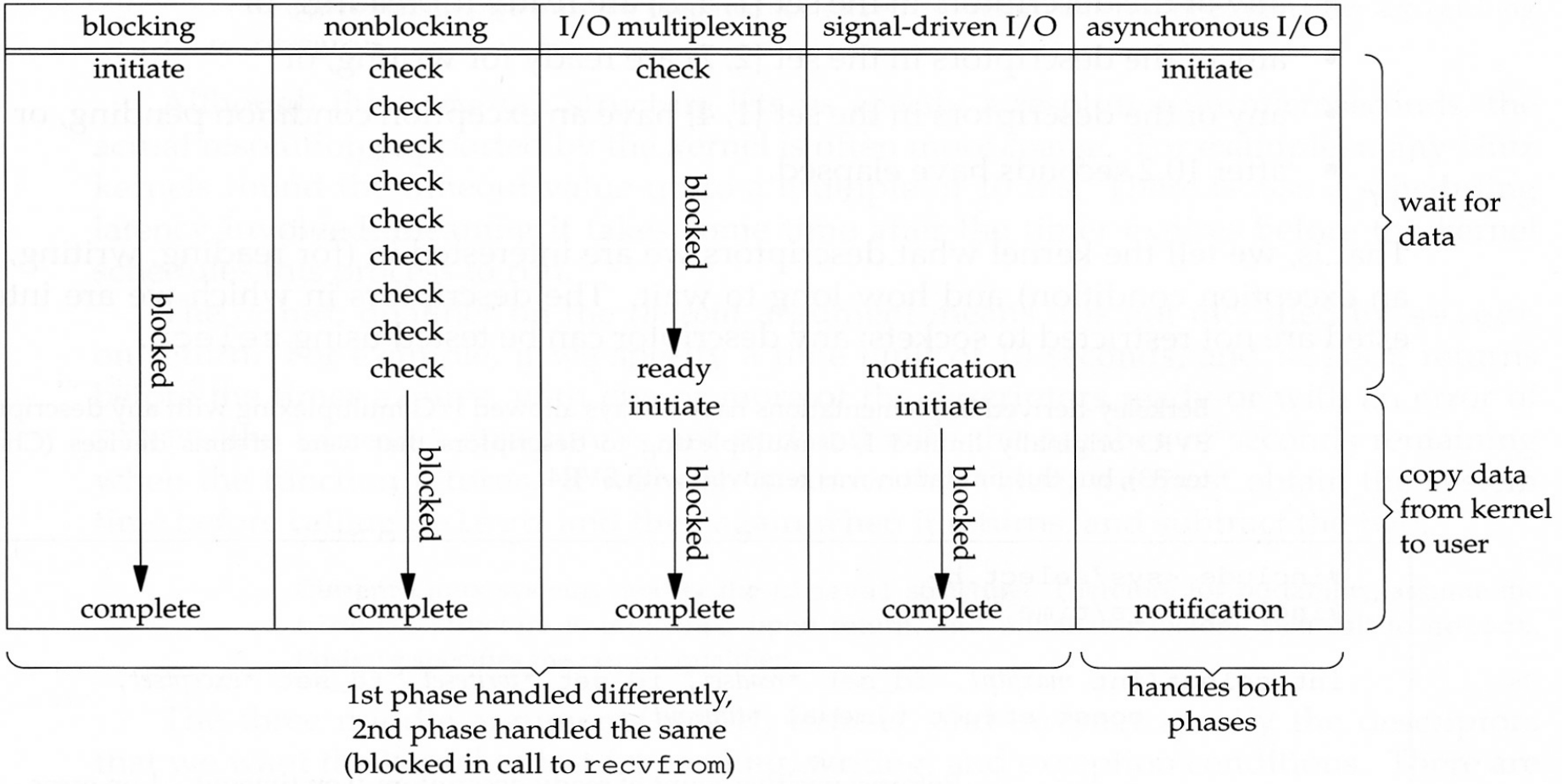


Figure 6.6 Comparison of the five I/O models.

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.

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
- 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

Select (2)

```
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 (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. This is called 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.

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)?

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 (receives a FIN)
  - A listening socket with nonzero number of connections
  - A socket error is pending

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.

Revised str_cli (1)

- Three conditions for socket
  - If peer TCP sends data, socket becomes readable and read returns greater than 0
  - If peer TCP sends a FIN, the socket becomes readable and read returns 0 (EOF)
  - If peer TCP sends RST, socket becomes readable and read returns -1, and errno contains specific error code.

Revised str_cli (2)

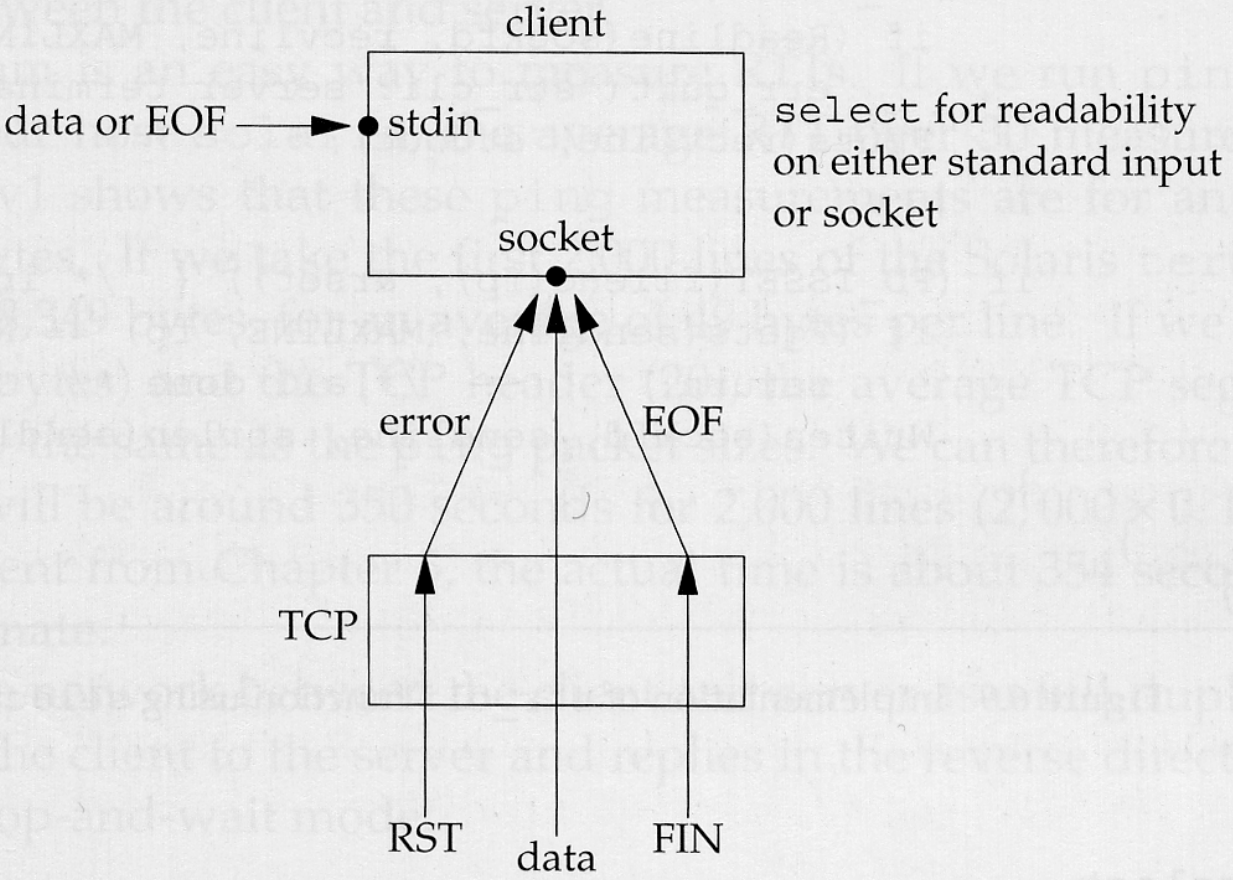


Figure 6.8 Conditions handled by `select` in `str_cli`.

Revised str_cli (3)

```
int maxfdp1;
fd_set rset;

FD_ZERO(&rset);
for ( ; ; ){
  FD_SET(fileno(fp), &rset);
  FD_SET(sockfd, &rset);
  maxfdp1=max(fileno(fp),sockfd) + 1;
  select(maxfdp1, &rset, NULL, NULL, NULL);
```

Revised str_cli (4)

```
  if (FD_ISSET(sockfd, &rset)) {
    if (readline(sockfd, recvline, MAXLINE)==0)
      err_quit("str_cli-server term premature");
    fputs(recvline, stdout);
  }
  if (FD_ISSET(fileno(fp), &rset)) {
    if (fgets(fileno(fp), MAXLINE, fp)==NULL)
      return;
    writen(sockfd, sendline, strlen(sendline));
  }
}
```

Shutdown (1)

- Normal way to terminate a network connection is to call *close*
- There are two limitations with *close*
  - *close* decrements the descriptor's reference count and closes the socket only if count reaches 0.
  - *close* terminates both directions of data transfer, reading and writing.

Shutdown (2)

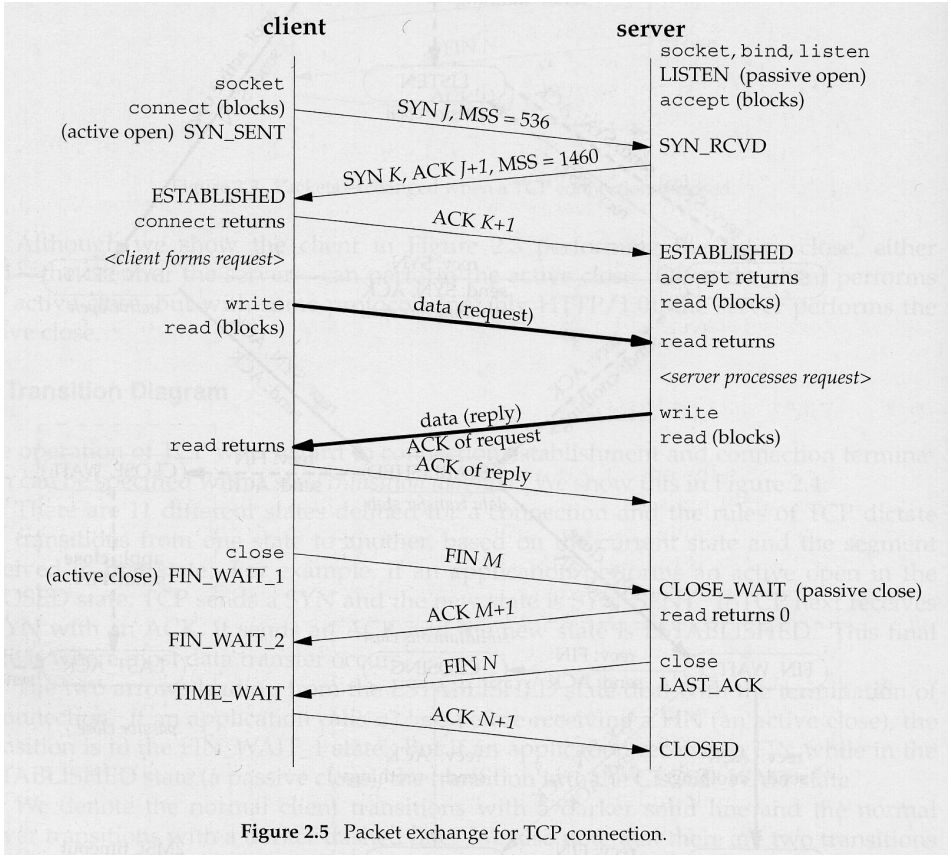


Figure 2.5 Packet exchange for TCP connection.

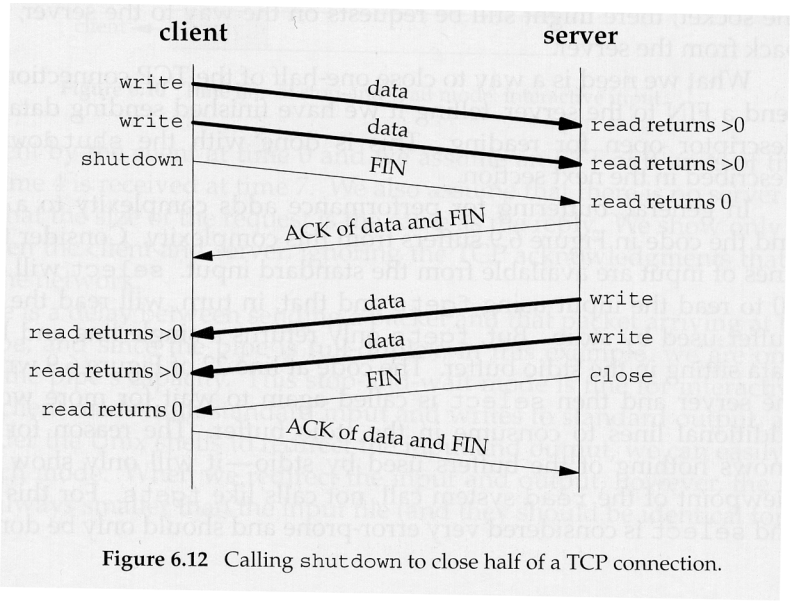


Figure 6.12 Calling `shutdown` to close half of a TCP connection.

Shutdown (3)

- *shutdown* function
  - Can initiate connection termination sequence regardless of the reference count
  - Can only terminate one direction of data transfer
  - `int shutdown(int sockfd,`
    `int howto)`
  - Returns: 0 if OK, -1 on error
  - *howto*: SHUT_RD, SHUT_WR, SHUT_RDWR

str_cli (again) 1

```
void str_cli (FILE *fp, int sockfd) {
  int maxfdp1, stdlineof;
  fd_set rset;
  char buf[MAXLINE];
  int n;

  stdlineof = 0;
  FD_ZERO(&rset);
```

str_cli (again) 2

```
    for ( ; ; ) {
      if (stdlineof == 0)
        FD_SET(fileno(fp), &rset);
      FD_SET(sockfd, &rset);
      maxfdp1 = max(fileno(fp), sockdf) + 1;
      select(maxfdp1, &rset, NULL, NULL, NULL);

      // deal with socket

      // deal with file

    }
}   // end of str_cli
```

str_cli (again) 3

```
// deal with socket

    if (FD_ISSET(sockdf, &rset) {
      if ( (n=read(sockfd, buf, MAXLINE)) == 0) {
        if (stdlineof == 1)
           return;
        else
           err_quit("str_cli: server
                     terminated prematurely");
      }
      write(fileno(stdout), buf, n);
    }
```

str_cli (again) 4

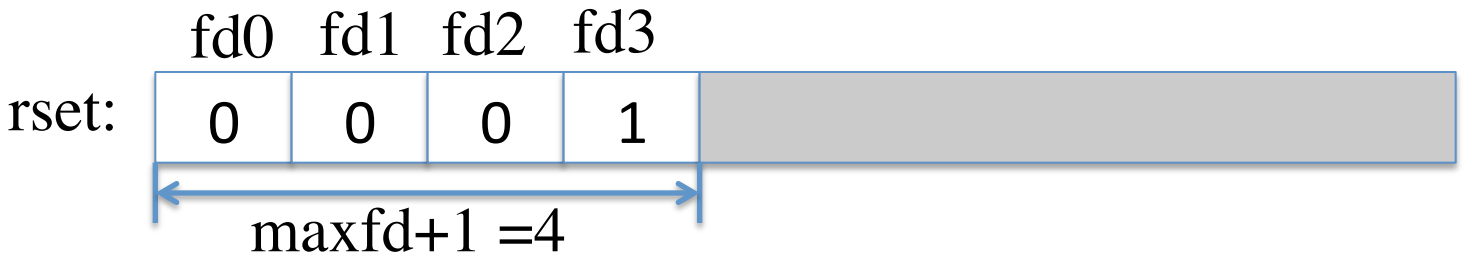
```
// deal with file

    if (FD_SET(fileno(fp), &rset)) {
      if ( (n=read(fileno(fp), buf, MAXLINE)) == 0) {
        stdlineof = 1;
        shutdown(sockfd, SHUT_WR); // send FIN
        FD_CLR(fileno(fp), &rset);
        continue;
      }
      writen(sockfd, buf, n);
    }
```

select-based Server (1)

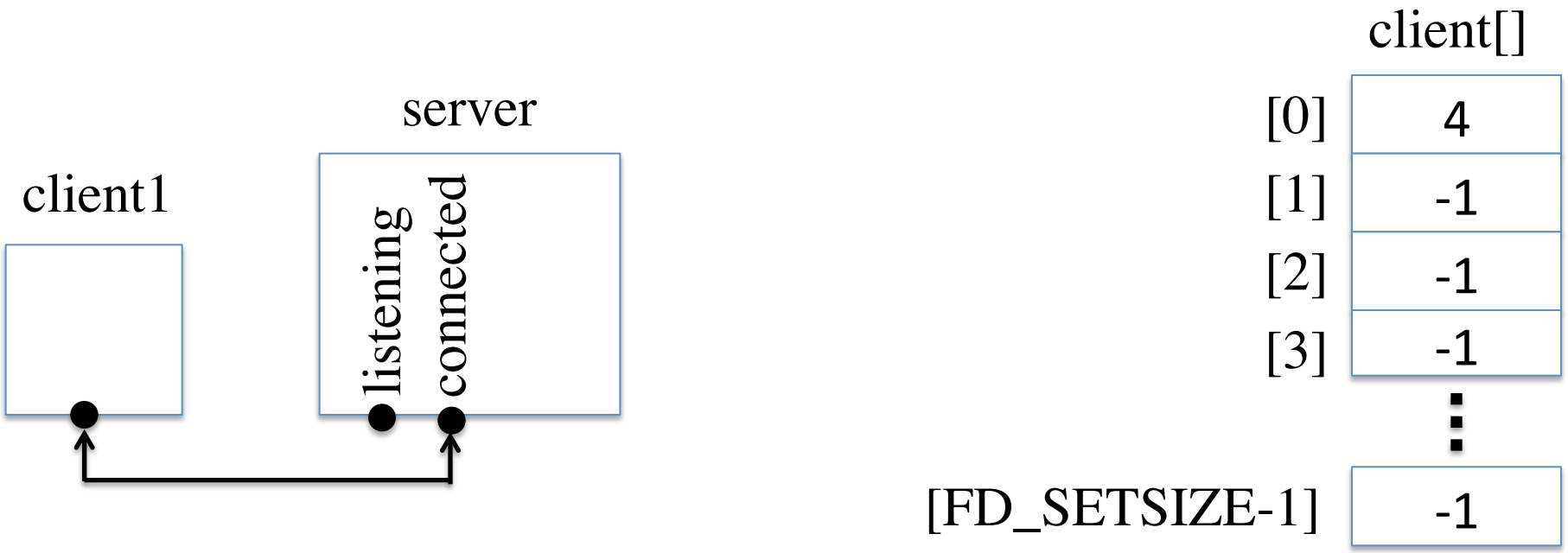
- Use *select* to handle multiple clients instead of forking one child per client.
  - Need to keep track of each client and its descriptor (array)
  - Need to keep track of the highest used descriptor
  - Good for many short lived clients
- See the attached source code for the TCP echo server

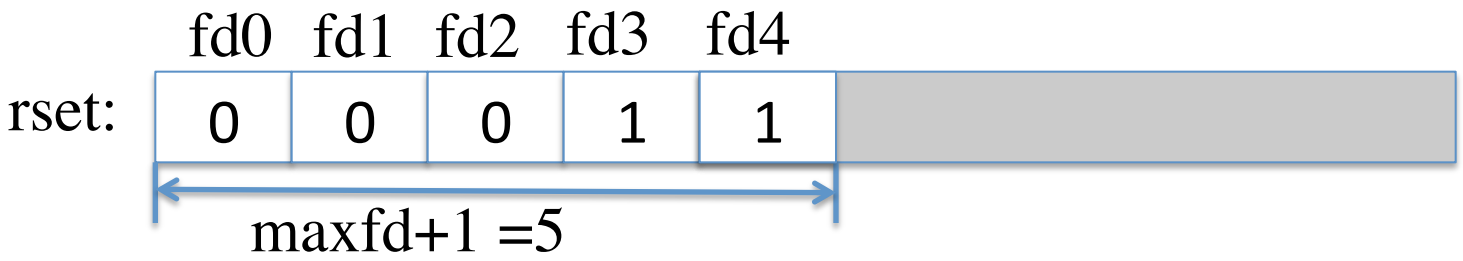
select-based Server (2)

FD_SETSIZE: the number of descriptors in the *fd_set* data type.

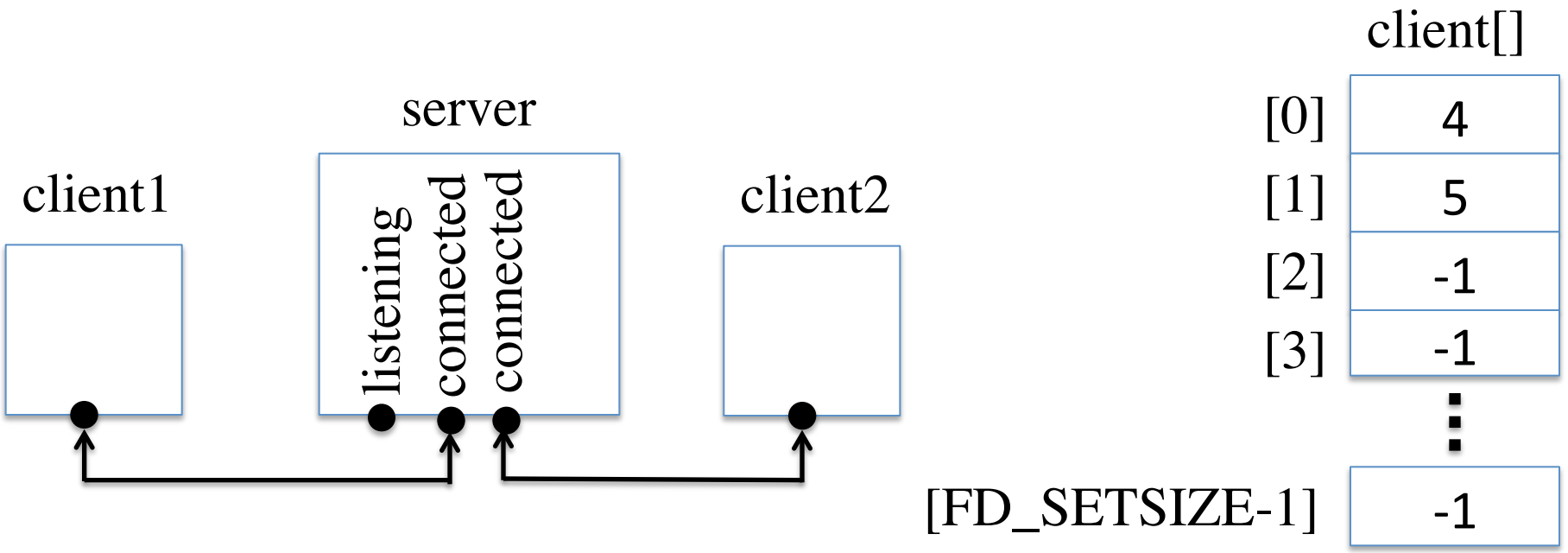


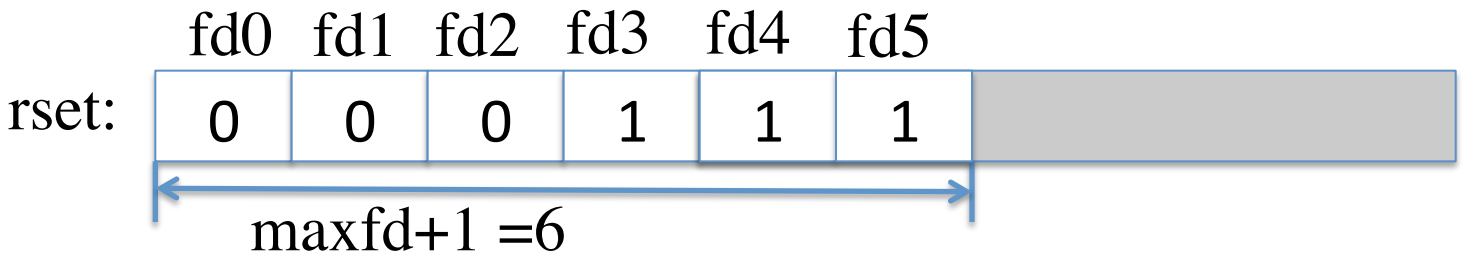
select-based Server (3)



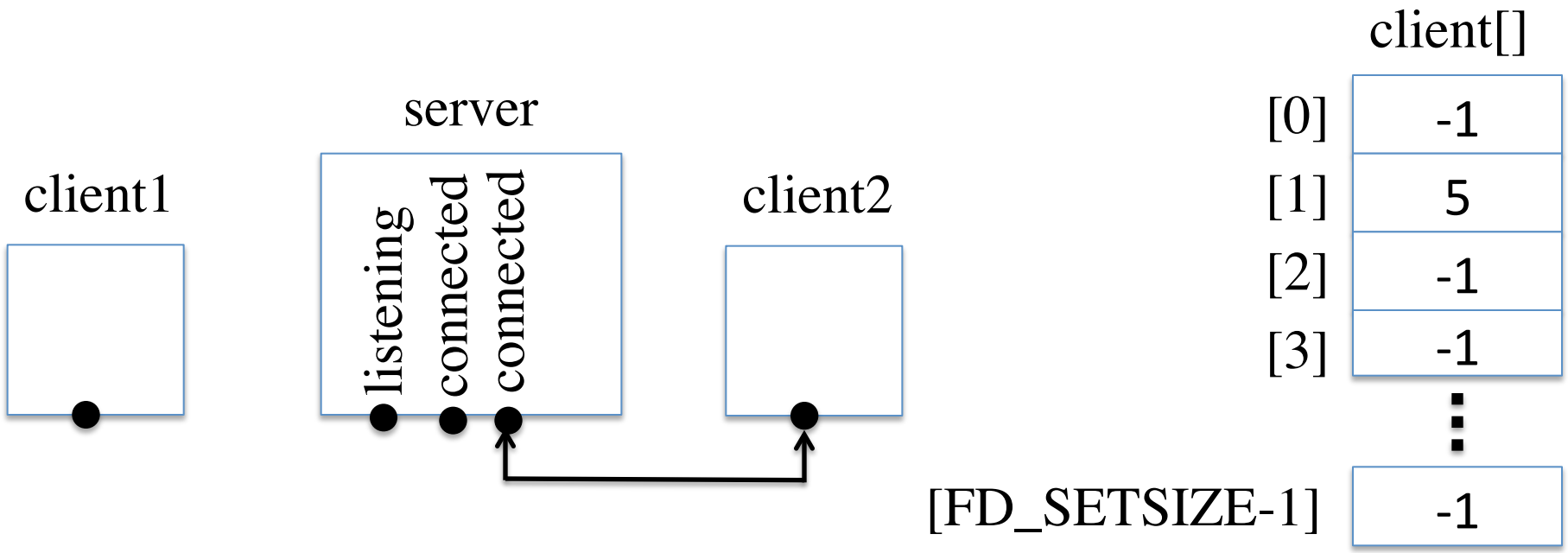


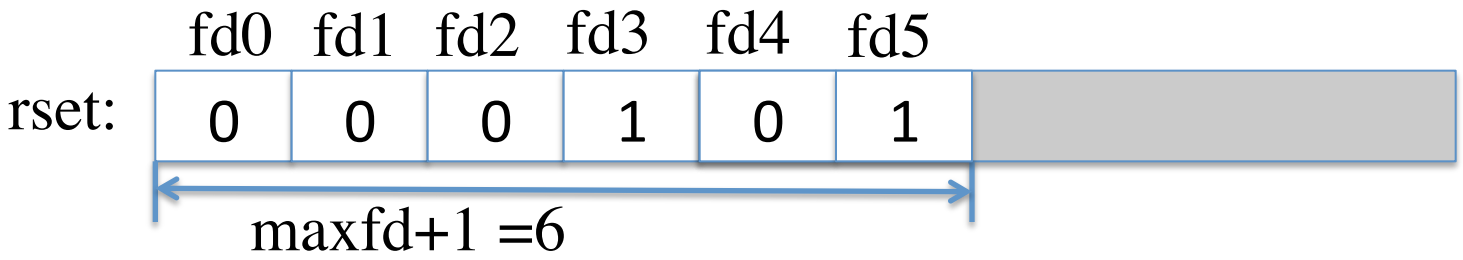
select-based Server (4)





select-based Server (5)





Socket Options

- There are many socket options for programmers to set for fine control of the underlying system and protocols
  - Generic socket options
  - IPv4 socket option
  - TCP socket options
  - IPv6 socket options
  - ICMPv6 socket options
- Options can be manipulated with the following functions
  - *getsockopt*
  - *setsockopt*

getsockopt and setsockopt (1)

- These two functions apply only to sockets

```
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);
```

getsockopt and setsockopt (2)

- Both return: 0 if OK, -1 on error
- *sockfd* must refer to an open socket descriptor
- *level* specifies the code in the system to interpret the option (the general code or protocol-specific code).
- *optname* is an integer representing the specific option.
- *optval* is a pointer to a variable storing the option value. (new value for setsockopt; current value for getsockopt)
- *Optlen* is a result-value parameter referring to the size of *optval*

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{}

Figure 7.1 Summary of socket and IP-layer socket options for getsockopt and setsockopt.

Summary of IP-layer socket options

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()

Figure 7.2 Summary of transport-layer socket options.

Option Values

- There are two types of option values
  - Binary options that enable or disable a certain feature (flags with • in the tables)
    - 0 for *disable*
    - nonzero for *enable*.
  - Options that fetch and return specific values that we can either set or examine (values). The actual values are passed between the kernel and the user spaces.
- Data types for values
  - Most option values are integer
  - Some are structures, such as *timeval*, *linger*, and character array

Socket States

- Some socket options have timing considerations about when to set or fetch the option due to the state of the socket
- The following options are inherited by a connected TCP socket from the listening socket
  - SO_DEBUG, SO_DONTROUTE, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF, SO_SNDBUF, SO_RCVLOWAT, SO_SNDLOWAT, TCP_MAXSEG, AND TCP_NODELAY
  - For TCP, the connected socket is not returned to a server by *accept* until the three-way handshake is completed by the TCP layer.
  - To ensure that one of the above options is set for the connected socket when the three-way handshake completes, we must set that option for the listening socket.

Generic Socket Options (1)

- SO_BROADCAST
  - Enable or disable the ability of the socket to send broadcast messages
- SO_DEBUG
  - Supported only by TCP. When enabled, the kernel keeps track of detailed info about all the packets sent or received by the socket
- SO_DONTROUTE
  - Bypass the normal routing mechanism of the underlying protocol

Generic Socket Options (2)

- SO_ERROR
  - Get pending error and clear
  - *so_error* is set to a Exxxx value
  - Called a pending error
  - Two ways for process to be immediately notified
    - If blocked in a call to select, select returns
    - If using signal-driven I/O, SIGIO signal is generated
  - Process can obtain so_error by fetching SO_ERROR option.
  - If so_error is nonzero
    - If read is called and no data to return, -1 is returned and errno=so_error
    - If read is called and data is queued, data is returned instead of the error condition
    - If write is called, -1 is returned and errno=so_error

Generic Socket Options (3)

- SO_KEEPALIVE
  - If there is no data exchanged in either direction for 2 hours, a probe is sent to the peer if this option is set. One of three scenarios exists:
    - Peer responds with an ACK
    - Peer responds with a RST
    - No response

Ways to detect TCP conditions

Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
Our TCP is actively sending data	Peer TCP sends a FIN, which we can detect immediately using `select` for readability. If TCP sends another segment, peer TCP responds with RST. If TCP sends yet another segment, our TCP sends us `SIGPIPE`.	Our TCP will time out and our socket's pending error is set to `ETIMEDOUT`.	Our TCP will time out and our socket's pending error is set to `EHOSTUNREACH`.
Our TCP is actively receiving data	Peer TCP will send a FIN, which we will read as a (possibly premature) end-of-file.	We will stop receiving data.	We will stop receiving data.
Connection is idle, keepalive set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to `ETIMEDOUT`.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to `EHOSTUNREACH`.
Connection is idle, keepalive not set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	(Nothing.)	(Nothing.)

Figure 7.5 Ways to detect various TCP conditions.

Linger (1)

- SO_LINGER
  - Specifies how *close* operates for a connection-oriented protocol
  - The following structure is used:

```
struct linger {
    int l_onoff;
    int l_linger;
}
// l_onoff - 0=off; nonzero=on
// l_linger specifies seconds
```

  - Three scenarios:
    - If *l_onoff* is 0, *close* returns immediately. If there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The value of *l_linger* is ignored.

Linger (2)

- If *l_onoff* is nonzero and linger is 0, TCP aborts the connection when close is called. TCP discards data in the send buffer and sends RST to the peer.
- If *l_onoff* is nonzero and *linger* is nonzero, the kernel will linger when close is called.
  - If there is any data in the send buffer, the process is put to sleep until either:
    - the data is sent and acknowledged

    Or
    - the linger time expires (for a nonblocking socket the process will not wait for *close* to complete)
  - When using this feature, the return value of *close* must be checked. If the linger time expires before the remaining data is send and acknowledged, close returns EWOULDBLOCK and any remaining data in the buffer is ignored.

close scenarios (1)

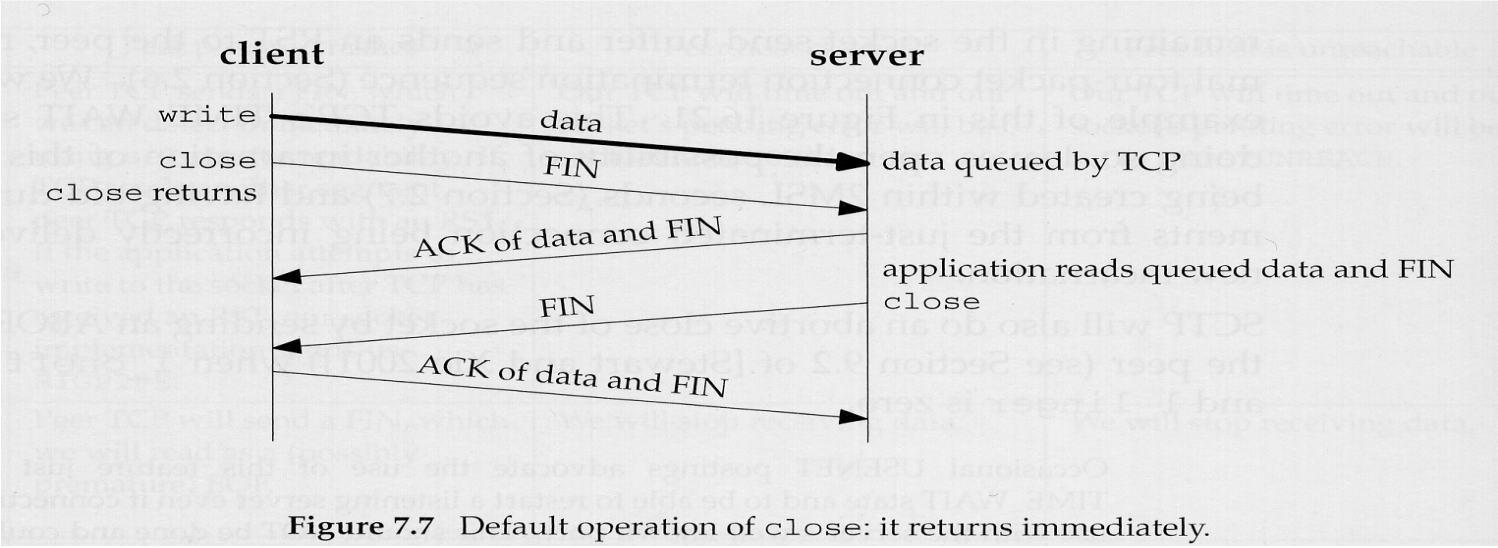


Figure 7.7 Default operation of `close`: it returns immediately.

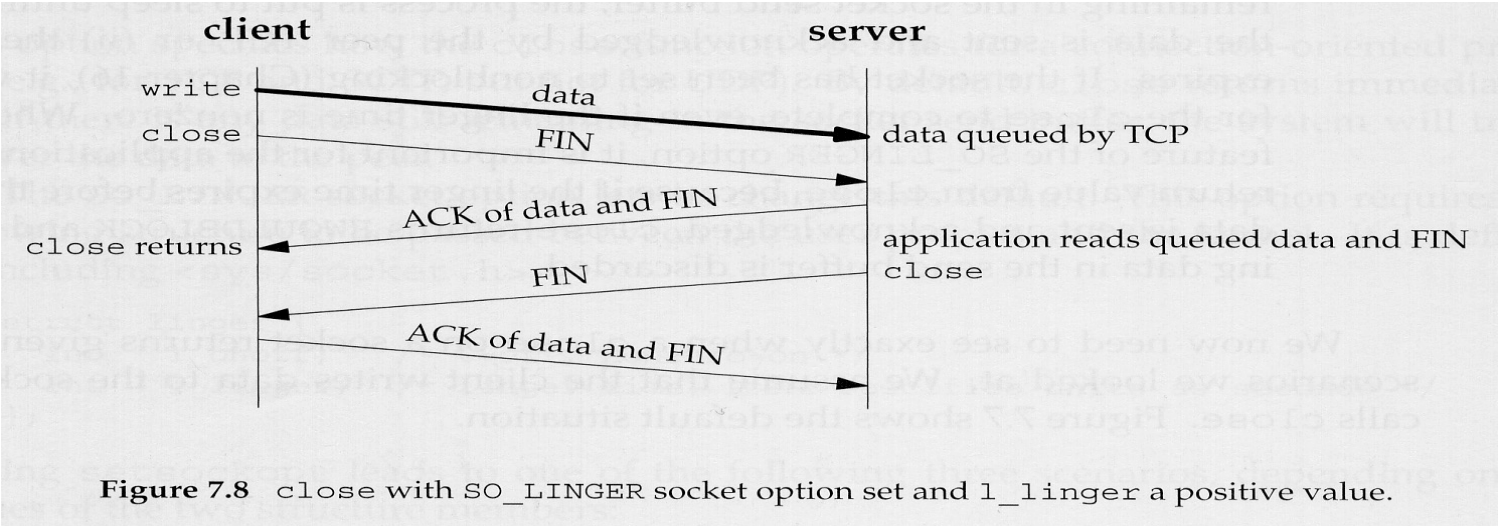


Figure 7.8 `close` with `SO_LINGER` socket option set and `l_linger` a positive value.

close scenarios (2)

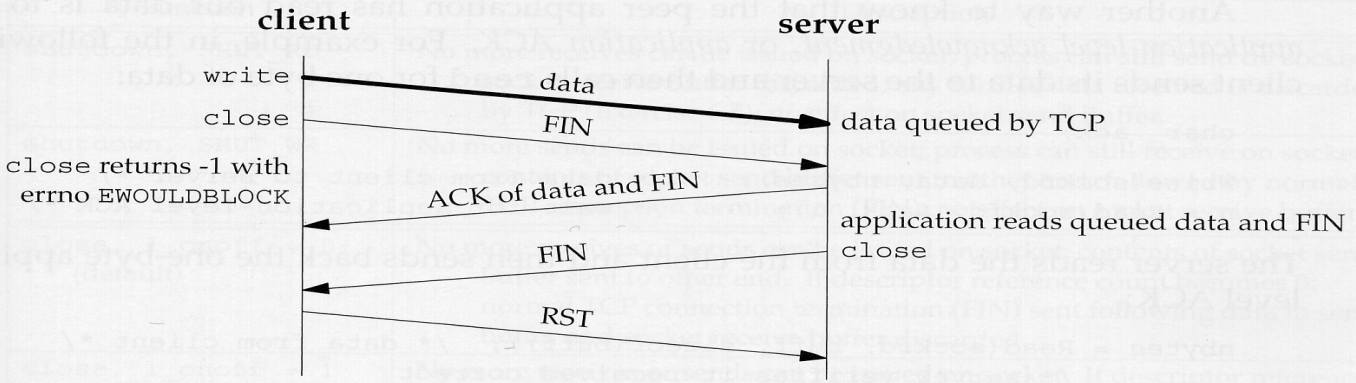


Figure 7.9 `close` with `SO_LINGER` socket option set and `l_linger` a small positive value.

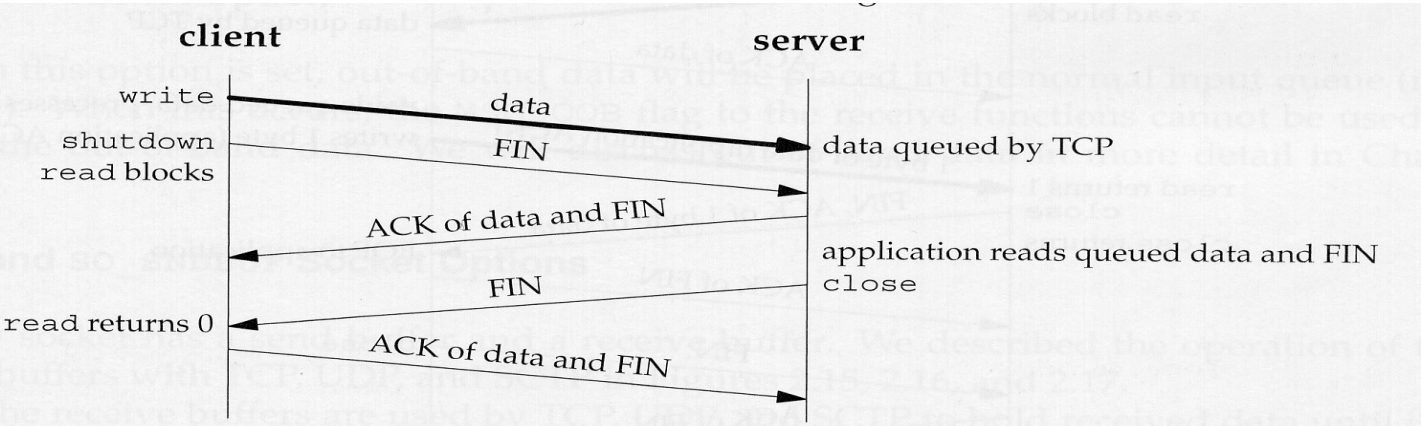


Figure 7.10 Using `shutdown` to know that peer has received our data.

close scenarios (3)

- Application ACK to confirm the receipt of data

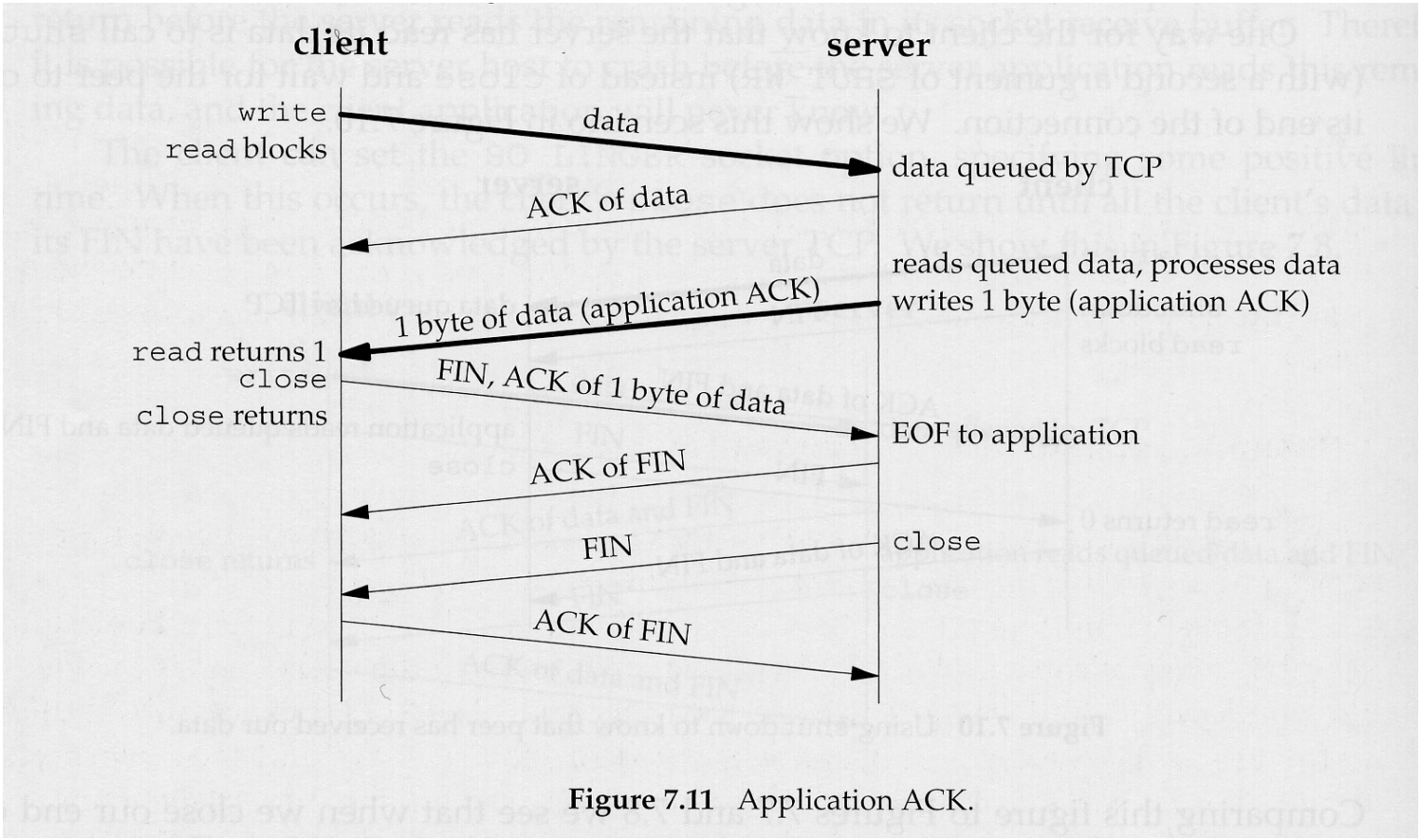


Figure 7.11 Application ACK.

close and shutdown

Function	Description
`shutdown, SHUT_RD`	No more receives can be issued on socket; process can still send on socket; socket receive buffer discarded; any further data received is discarded by TCP (Exercise 6.5); no effect on socket send buffer.
`shutdown, SHUT_WR`	No more sends can be issued on socket; process can still receive on socket; contents of socket send buffer sent to other end, followed by normal TCP connection termination (FIN); no effect on socket receive buffer.
`close, l_onoff = 0` (default)	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger = 0`	No more receives or sends can be issued on socket. If descriptor reference count becomes 0: RST sent to other end, connection state set to CLOSED (no TIME_WAIT state), socket send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger != 0`	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer, socket receive buffer discarded, and if linger time expires before connection CLOSED, `close` returns `EWOULDBLOCK`.

Figure 7.10 Summary of `shutdown` and `SO_LINGER` scenarios.

Generic Socket Options (4)

- SO_OOBINLINE
  - Out-of-band data can be placed in the normal input queue
- SO_RCVBUF and SO_SNDBUF
  - Get/set the send buffer size and receive buffer size
  - These sizes are related to capacity of the connection
- SO_RCVLOWAT and SO_SNDLOWAT
  - Decide the conditions for *readable* and *writable*

Generic Socket Options (5)

- SO_RCVTIMEO and SO_SNDTIMEO
  - Place a timeout on socket receives and sends
  - They affect read and write function families
- SO_TYPE
  - Returns the socket type such as SOCK_STREAM and SOCK_DGRAM
- SO_USELOOPBACK
  - Applies only to routing sockets

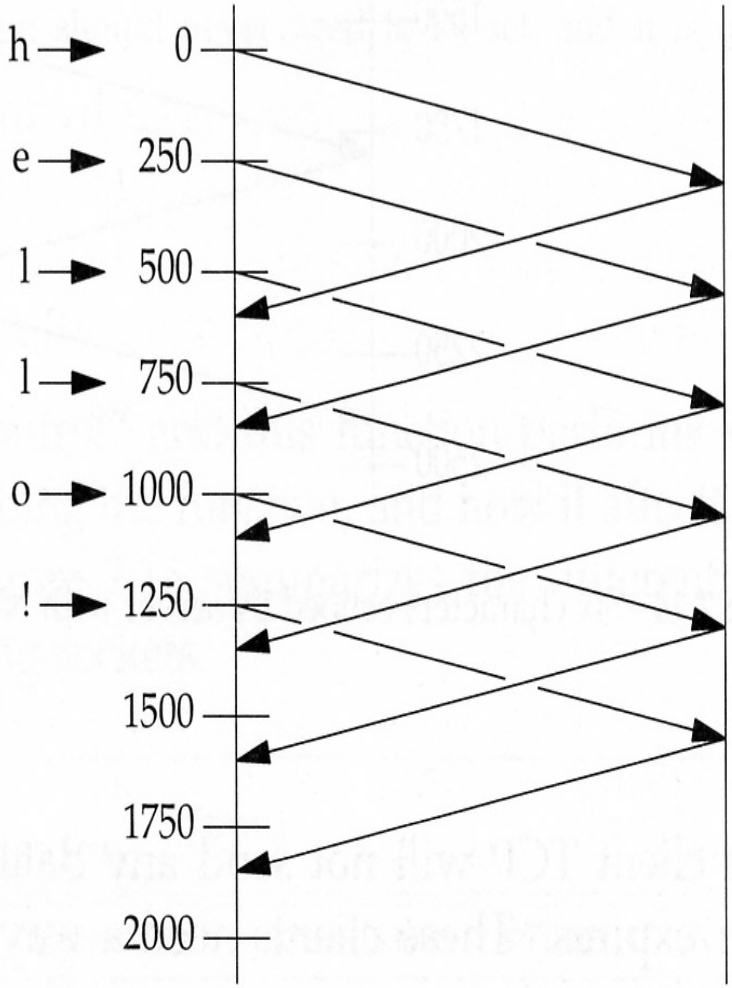
Generic Socket Options (6)

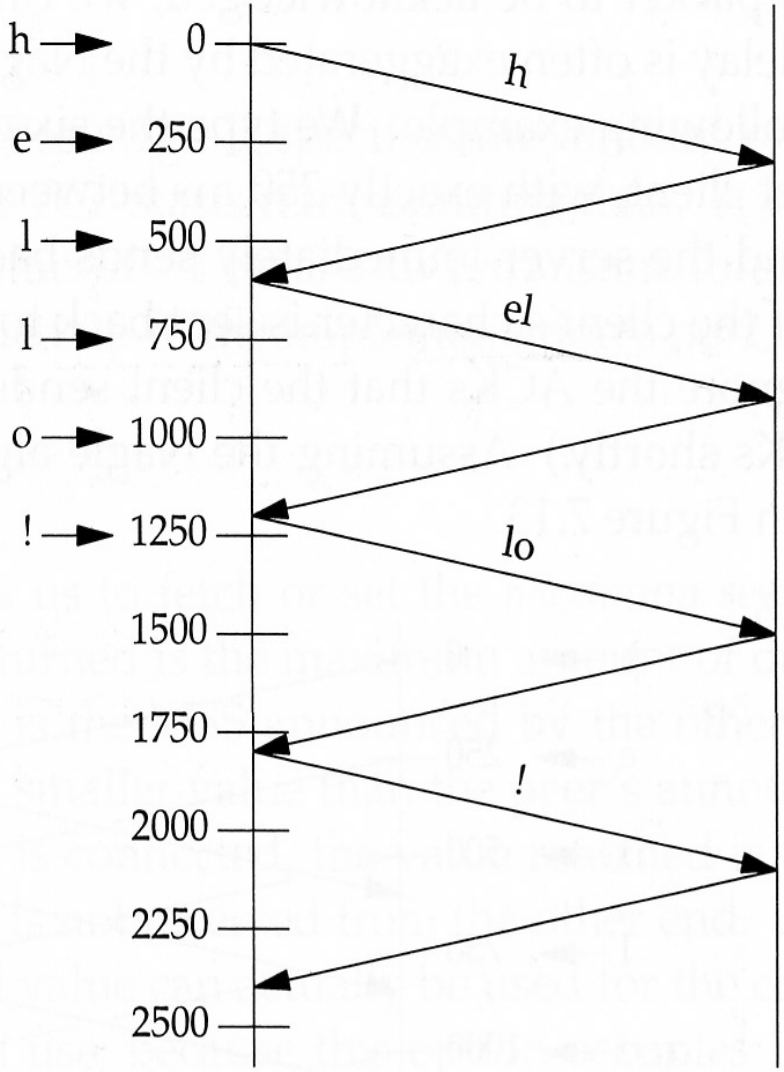
- SO_REUSEADDR
  - Allows a listening server to start and bind its well-known port even if previously established connections exist that use this port as their local port
  - Allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.
  - Allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address
  - Allows completely duplicate bindings: a bind of an IP address and port, when that same IP address and port are already bound to another socket (normally for support of multicasting).

IPv4 Socket Options (2)

- IP_TOS
  - Get/set the type-of-service field in the IP header
- IP_TTL
  - Get/set the default TTL
- TCP_MAXSEG
  - Get/set the maximum segment size for a TCP connection
- TCP_NODELAY
  - Disable the delay algorithm (Nagle algorithm)

Nagle Algorithm





TCP client/server

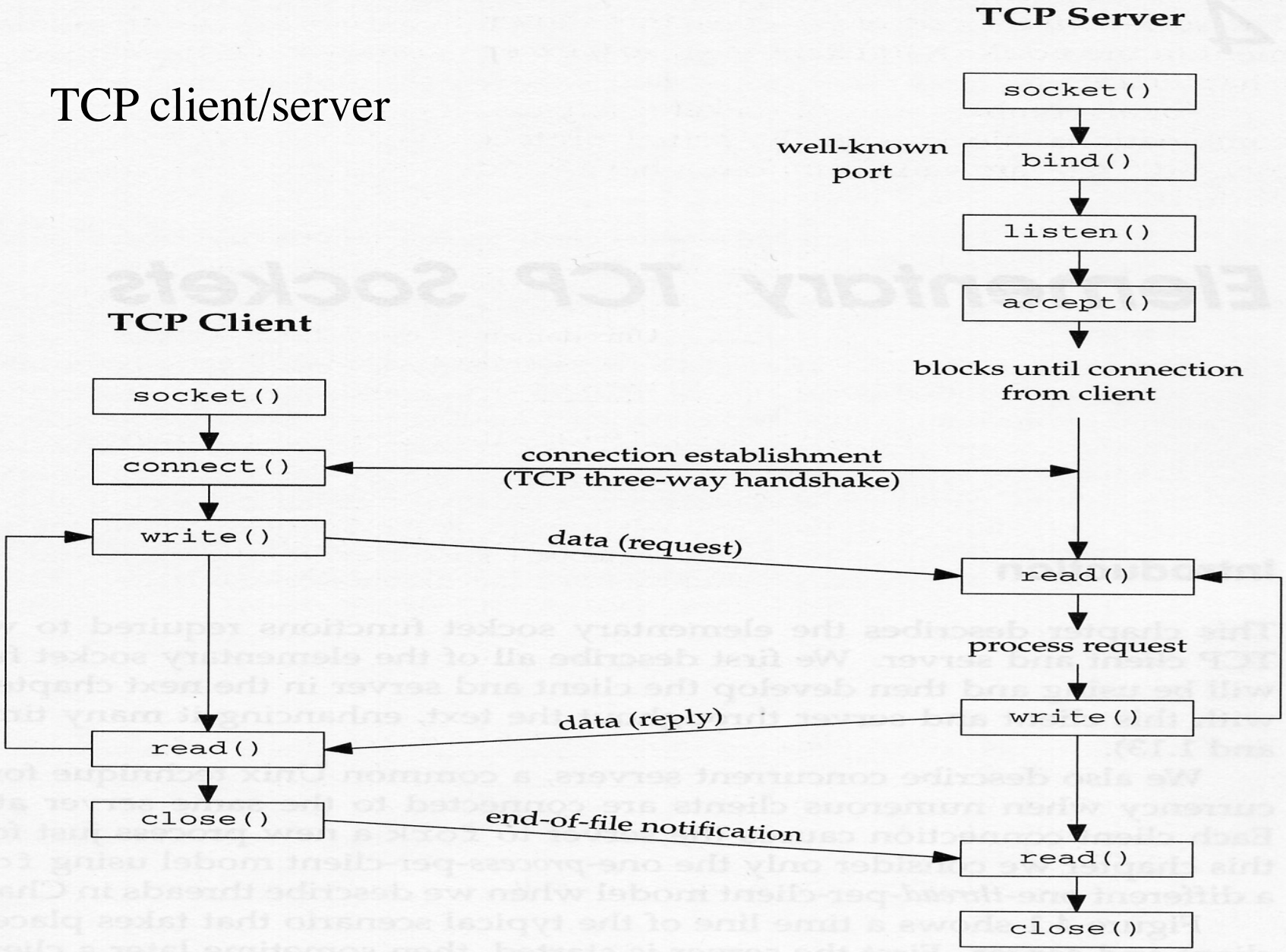


Figure 4.1 Socket functions for elementary TCP client–server.

UDP client/server

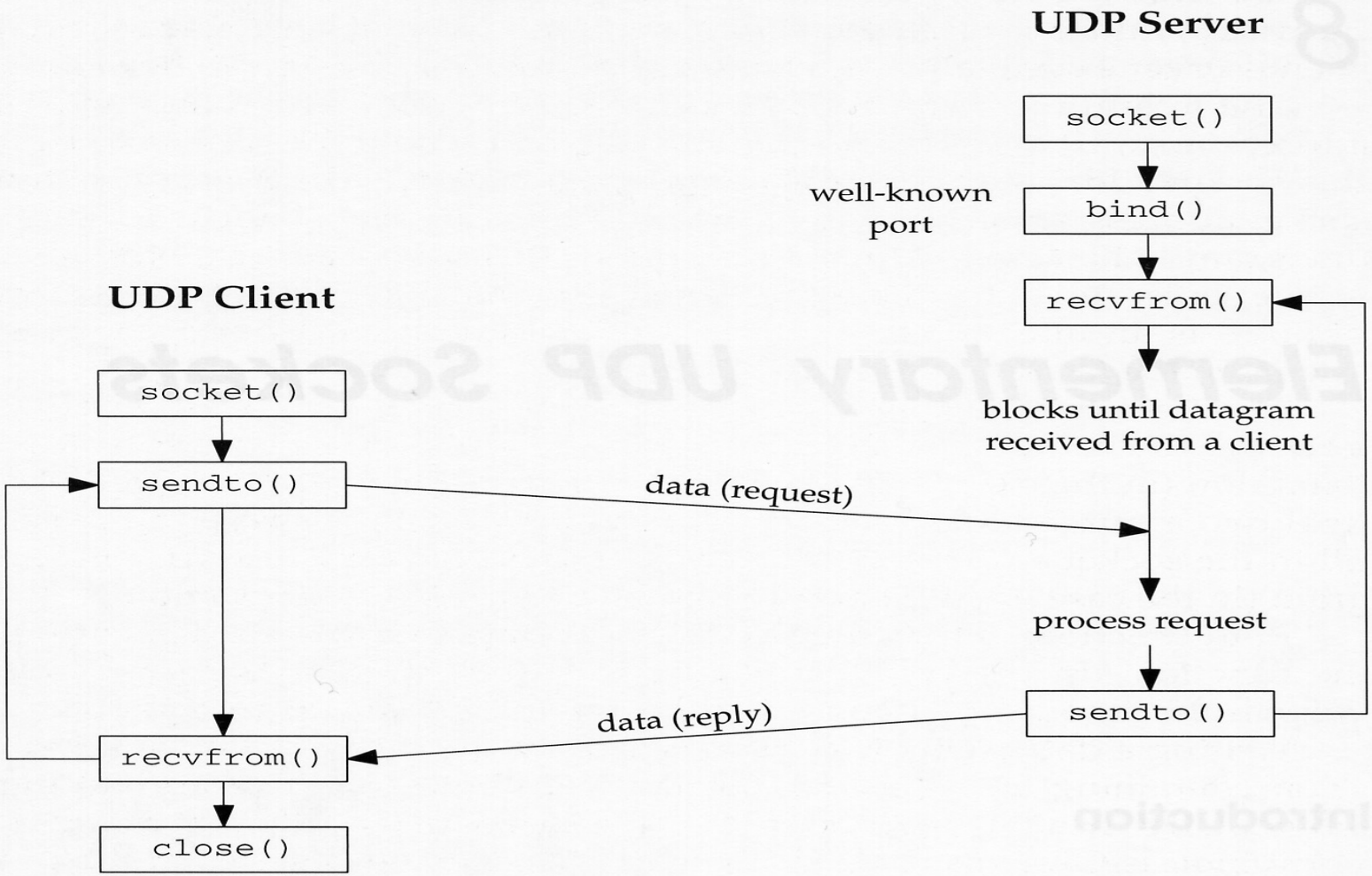


Figure 8.1 Socket functions for UDP client–server.

recvfrom and sendto (1)

```
ssize_t recvfrom(int sockfd,
    void *buff, size_t nbytes,
    int flags, struct sockaddr *from,
    socklen_t *addrlen);

ssize_t sendto(int sockfd,
    const void *buff, size_t nbytes,
    int flags, const struct sockaddr *to,
    socklen_t addrlen);
```

- Both return: number of bytes read or written if OK, -1 on error

recvfrom and sendto (2)

- *sockfd*, *buff*, and *nbytes* are identical to read/write
- f*lags* is normally set 0, but can be set for advanced functions
- The final two arguments to *recvfrom* are similar to the final two arguments to *accept*. They can be NULL.
- The final two arguments to *sendto* are similar to the final two arguments to *connect*
- Send 0 bytes is ok; likewise receive 0 bytes is ok

Simple UDP C/S

- Refer to *udpcliserv/udpserv01.c*, *lib/dg_echo.c*, *udpcliserv/udpcli01.c*, and *lib/dg_cli.c* for details
- Sockets are created with type SOCK_DGRAM

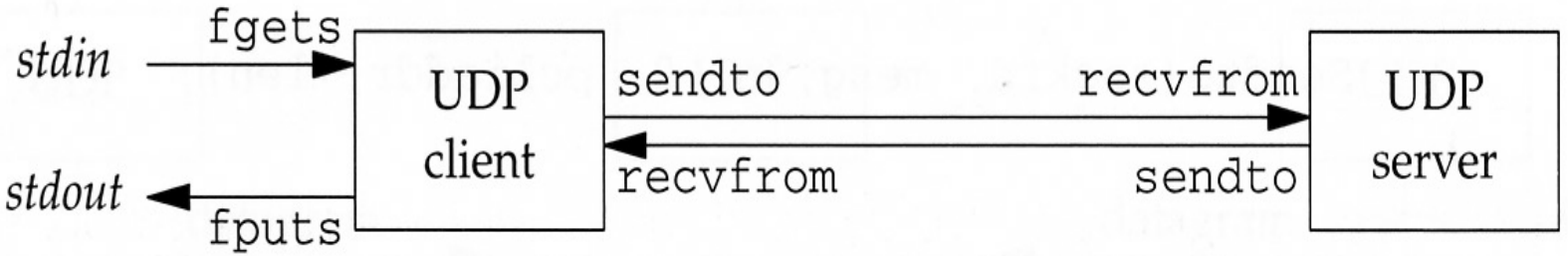


Figure 8.2 Simple echo client–server using UDP.

- If a datagram is lost, the client will wait forever
- Timeout is needed, but not enough (duplicate problem)