Socket Options and Signal-Driven I/O

- *getsockopt* and *setsockopt* functions
- Check options and obtain default values
- Generic socket options
- IPv4 socket options
- IPv6 socket options
- TCP socket options
- *fcntl* function
- Signal-driven I/O
- Summary

getsockopt and *setsockopt* Functions

```
#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);
```

Both return: 0 if OK, -1 if error

Types of options: flag and value

Levels:

Generic --	SOL_SOCKET
IP --	IPPROTO_IP
ICMPv6 --	IPPROTO_ICMPV6
IPv6 --	IPPROTO_IPV6
TCP --	IPPROTO_TCP

Check Options and Obtain Default Values

- Program declaration:
  - declare union of possible option values
  - define printing function prototypes
  - define sock_opt structure, initialize sock_opt[ ] array
- Check and print options:
  - create TCP socket, go through all options
  - call getsockopt
  - print option’s default value

sockopt/checkopts.c

```
 1 #include    "unp.h"
 2 #include    <netinet/tcp.h>     /* for TCP_xxx defines */
 3 union val {
 4   int               i_val;
 5   long              l_val;
 6   struct linger     linger_val;
 7   struct timeval    timeval_val;
 8 } val;
 9 static char *sock_str_flag(union val *, int);
10 static char *sock_str_int(union val *, int);
11 static char *sock_str_linger(union val *, int);
12 static char *sock_str_timeval(union val *, int);
13 struct sock_opts {
14   const char      *opt_str;
15   int        opt_level;
16   int        opt_name;
17   char   *(*opt_val_str)(union val *, int);
18 } sock_opts[] = {
19     { "SO_BROADCAST",     SOL_SOCKET, SO_BROADCAST,    sock_str_flag },
20     { "SO_DEBUG",         SOL_SOCKET, SO_DEBUG,        sock_str_flag },
21     { "SO_DONTROUTE",     SOL_SOCKET, SO_DONTROUTE,    sock_str_flag },
22     { "SO_ERROR",         SOL_SOCKET, SO_ERROR,        sock_str_int },
23     { "SO_KEEPALIVE",     SOL_SOCKET, SO_KEEPALIVE,    sock_str_flag },
24     { "SO_LINGER",        SOL_SOCKET, SO_LINGER,       sock_str_linger },
25     { "SO_OOBINLINE",     SOL_SOCKET, SO_OOBINLINE,    sock_str_flag },
```

```
  { "SO_RCVTIMEO",      SOL_SOCKET, SO_RCVTIMEO,    sock_str_timeval },
  { "SO_SNDTIMEO",      SOL_SOCKET, SO_SNDTIMEO,    sock_str_timeval },
  { "SO_REUSEADDR",     SOL_SOCKET, SO_REUSEADDR,   sock_str_flag },
#ifdef  SO_REUSEPORT
  { "SO_REUSEPORT",     SOL_SOCKET, SO_REUSEPORT,   sock_str_flag },
#else
  { "SO_REUSEPORT",     0,          0,              NULL },
#endif
  { "SO_TYPE",          SOL_SOCKET, SO_TYPE,        sock_str_int },
  { "SO_USELOOPBACK",   SOL_SOCKET, SO_USELOOPBACK, sock_str_flag },
  { "IP_TOS",           IPPROTO_IP, IP_TOS,         sock_str_int },
  { "IP_TTL",           IPPROTO_IP, IP_TTL,         sock_str_int },
  { "IPV6_DONTFRAG",    IPPROTO_IPV6,IPV6_DONTFRAG, sock_str_flag },
  { "IPV6_UNICAST_HOPS", IPPROTO_IPV6,IPV6_UNICAST_HOPS,sock_str_int },
  { "IPV6_V6ONLY",      IPPROTO_IPV6,IPV6_V6ONLY,   sock_str_flag },
  { "TCP_MAXSEG",       IPPROTO_TCP,TCP_MAXSEG,     sock_str_int },
  { "TCP_NODELAY",      IPPROTO_TCP,TCP_NODELAY,    sock_str_flag },
  { "SCTP_AUTOCLOSE",   IPPROTO_SCTP,SCTP_AUTOCLOSE,sock_str_int },
  { "SCTP_MAXBURST",    IPPROTO_SCTP,SCTP_MAXBURST, sock_str_int },
  { "SCTP_MAXSEG",      IPPROTO_SCTP,SCTP_MAXSEG,   sock_str_int },
  { "SCTP_NODELAY",     IPPROTO_SCTP,SCTP_NODELAY,  sock_str_flag },
  { NULL,               0,          0,              NULL }
```

Figure 7.4 shows our `main` function.

sockopt/checkopts.c

```
53 int
54 main(int argc, char **argv)
55 {
56     int     fd;
57     socklen_t len;
58     struct sock_opts *ptr;

59     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
60         printf("%s: ", ptr->opt_str);
61         if (ptr->opt_val_str == NULL)
62             printf("(undefined)\n");
63         else {
64             switch (ptr->opt_level) {
65             case SOL_SOCKET:
66             case IPPROTO_IP:
67             case IPPROTO_TCP:
68                 fd = Socket(AF_INET, SOCK_STREAM, 0);
69                 break;
70 #ifdef  IPV6
71             case IPPROTO_IPV6:
72                 fd = Socket(AF_INET6, SOCK_STREAM, 0);
73                 break;
74 #endif
75 #ifdef  IPPROTO_SCTP
76             case IPPROTO_SCTP:
77                 fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                 break;
79 #endif
80             default:
81                 err_quit("Can't create fd for level %d\n", ptr->opt_level);
```

```
70 #ifdef  IPV6
71             case IPPROTO_IPV6:
72                 fd = Socket(AF_INET6, SOCK_STREAM, 0);
73                 break;
74 #endif
75 #ifdef  IPPROTO_SCTP
76             case IPPROTO_SCTP:
77                 fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                 break;
79 #endif
80             default:
81                 err_quit("Can't create fd for level %d\n", ptr->opt_level);
82             }

83             len = sizeof(val);
84             if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
85                            &val, &len) == -1) {
86                 err_ret("getsockopt error");
87             } else {
88                 printf("default = %s\n", (*ptr->opt_val_str) (&val, len));
89             }
90             close(fd);
91         }
92     }
93     exit(0);
94 }
```

sockopt/checkopts.c

sockopt/checkopts.c

```
 95 static char strres[128];

 96 static char *
 97 sock_str_flag(union val *ptr, int len)
 98 {
 99     if (len != sizeof(int))
100         snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)", len);
101     else
102         snprintf(strres, sizeof(strres),
103                  "%s", (ptr->i_val == 0) ? "off" : "on");
104     return(strres);
105 }
```

sockopt/checkopts.c

```
freebsd % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 57344
SO_SNDBUF: default = 32768
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 2048
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: default = off
SO_TYPE: default = 1
SO_USELOOPBACK: default = off
IP_TOS: default = 0
IP_TTL: default = 64
IPV6_DONTFRAG: default = off
IPV6_UNICAST_HOPS: default = -1
IPV6_V6ONLY: default = off
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
SCTP_AUTOCLOSE: default = 0
SCTP_MAXBURST: default = 4
```

Generic Socket Options

- SO_BROADCAST: permit sending of broadcast datagram, only on broadcast links
- SO_DONTROUTE: bypass routing table lookup, used by routing daemons (routed and gated) in case the routing table is incorrect
- SO_ERROR: get pending error and clear
- SO_KEEPALIVE: test if TCP connection still alive periodically (after a 2-hour idle time, send a packet that will provoke the receiver to send back an ACK if the receiver is still alive.)

SO_KEEPALIVE option

- The purpose of this option is to detect if the peer host crashes or becomes unreachable.
- When this option is set and no data has been exchanged in either direction for two hours, TCP automatically sends a keep-alive probe to the peer. Three possible outcomes:
  - The peer responds with an ACK. This shows that the peer is still alive and the TCP will send another probe two hours later.
  - The peer responds with an RST. This shows that the peer has crashed and rebooted. The socket's pending error is set to ECONNRESET and the socket is closed.
  - There is no response from the peer. In this case, eight additional probes will be sent, 75 seconds apart, and then the socket's pending error is set to ETIMEOUT and the socket is closed.

SO_LINGER option

- This option specifies how the close() function operates for a connection-oriented protocol.
- The default, the close() returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.
- This option can change the default processing by passing the struct linger{} to the kernel.

Struct Linger{}

- Struct linger {
  int l_onoff; (flag)
  int l_linger; (lingering time in seconds)
  }
- If l_onoff is 0, this option is turned off and the value of l_linger is ignored and the default processing is used.
- If l_onoff is 1 and l_linger is 0, then TCP aborts the connection when it is closed. That is, TCP discards any data still remaining in the socket send buffer and send a RST to the peer. The close() returns immediately.

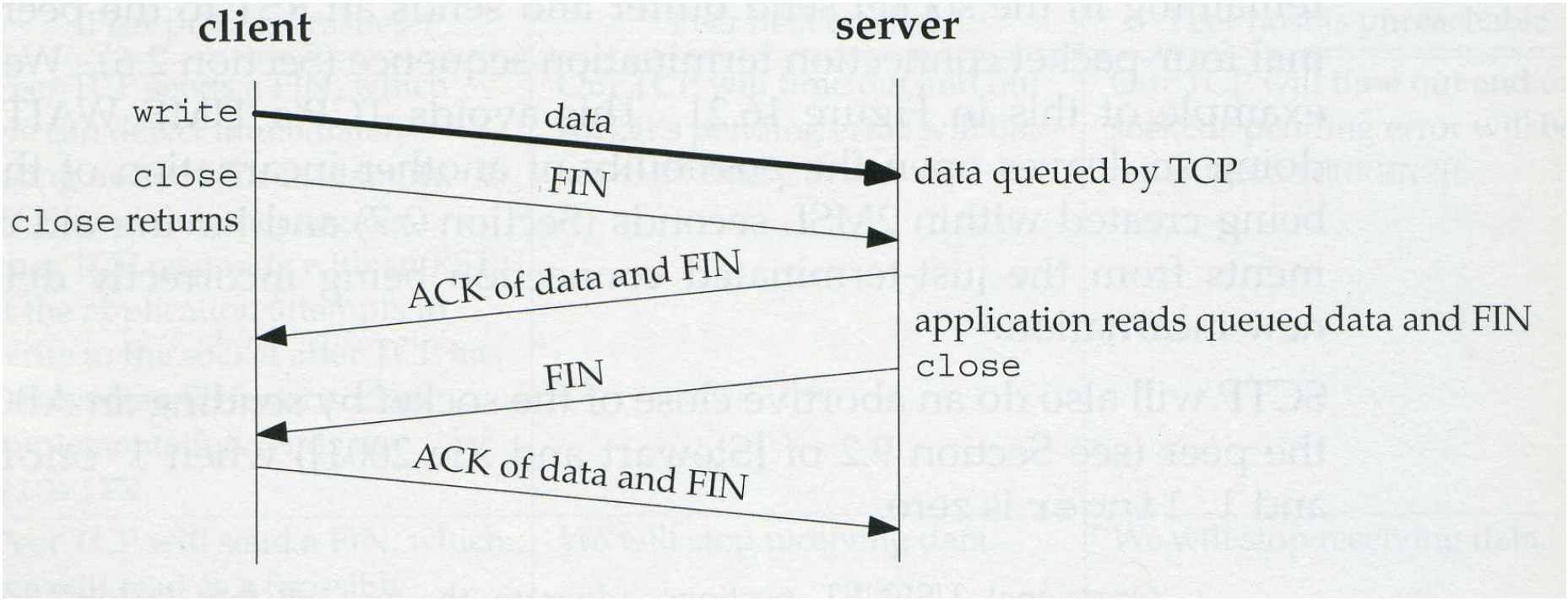
Linger Example

```
ling.l_onoff = 1;           /* cause RST to be sent on close() */
ling.l_linger = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
Close(sockfd);
```

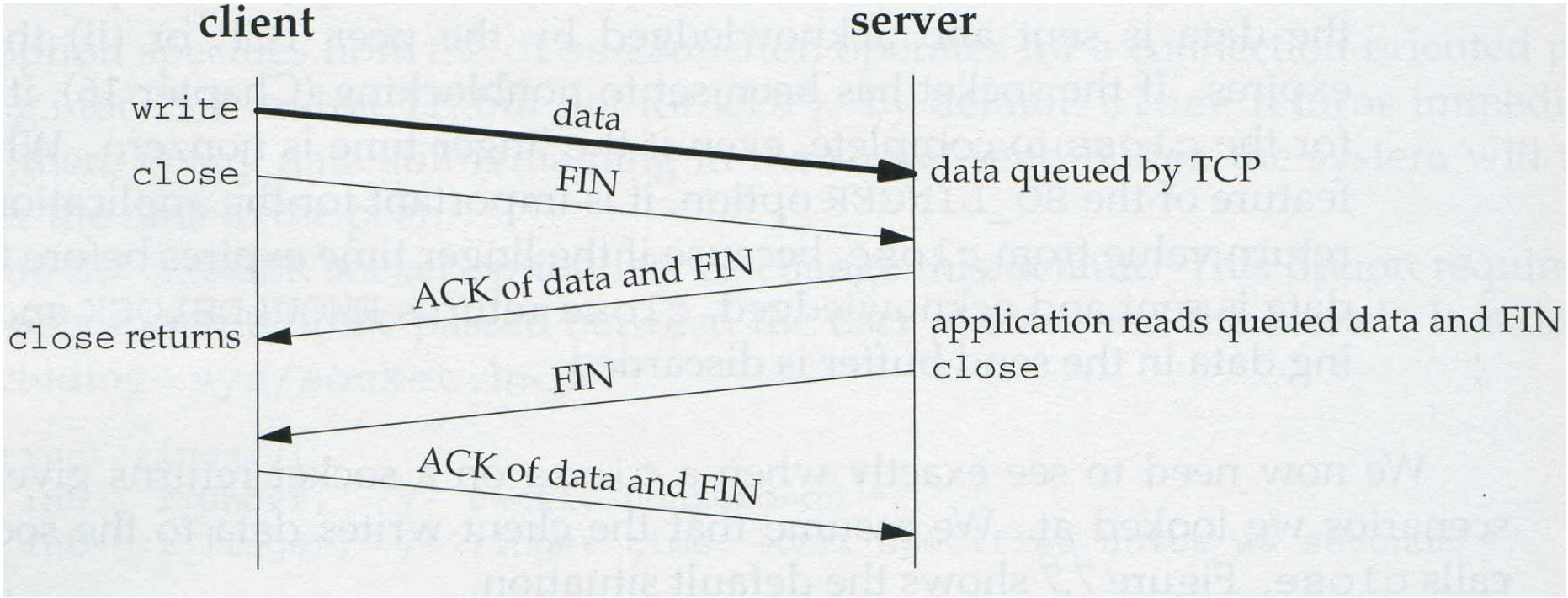
Struct Linger{} (cont'd)

- If l-onoff is nonzero and l_linger is nozero, then the kernel will linger when the socket is closed. That is, if there is any data remaining in the socket send buffer, the process is put to sleep until (1) all the data is sent and acknowledged by the peer TCP, or (2) the linger time expires (in this case, the close() will return -1 with errno set to EWOULDBLOCK).

Default operation of close()

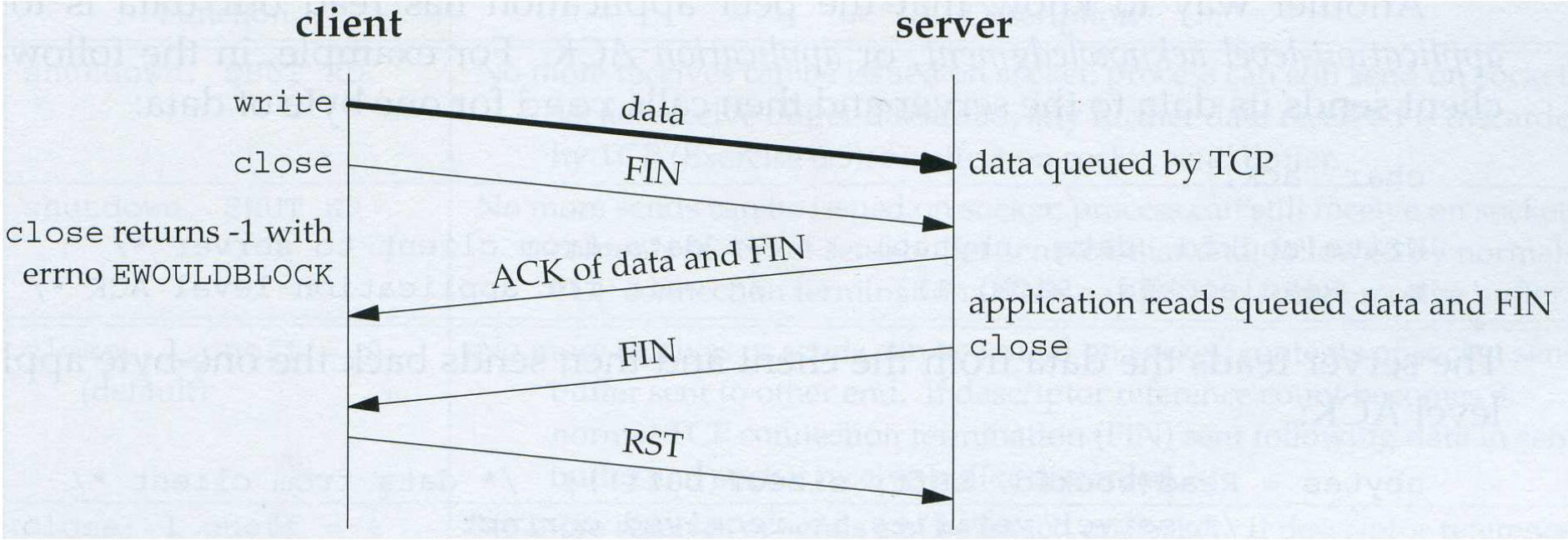


Close() with l_linger set to a positive value

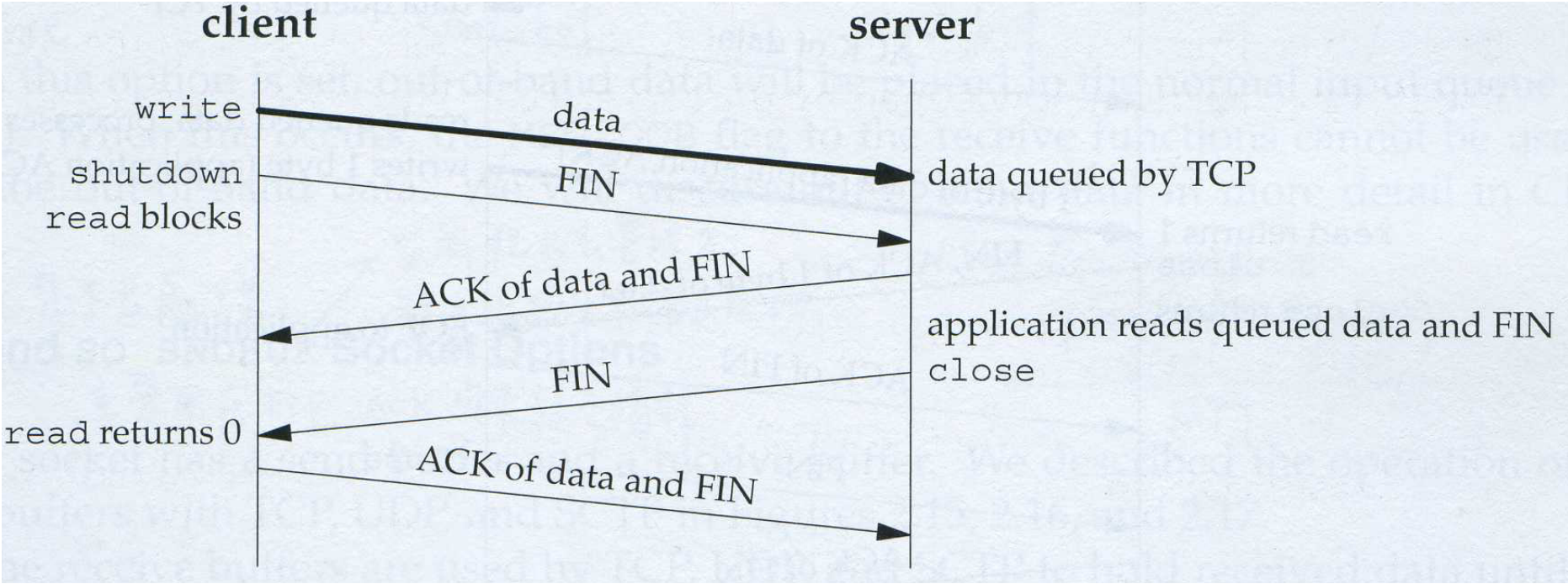


Important fact: A successful return from close() with this option set only tell us that the data have been ACKed by the peer TCP. This does not tell us whether the peer application has read the data!

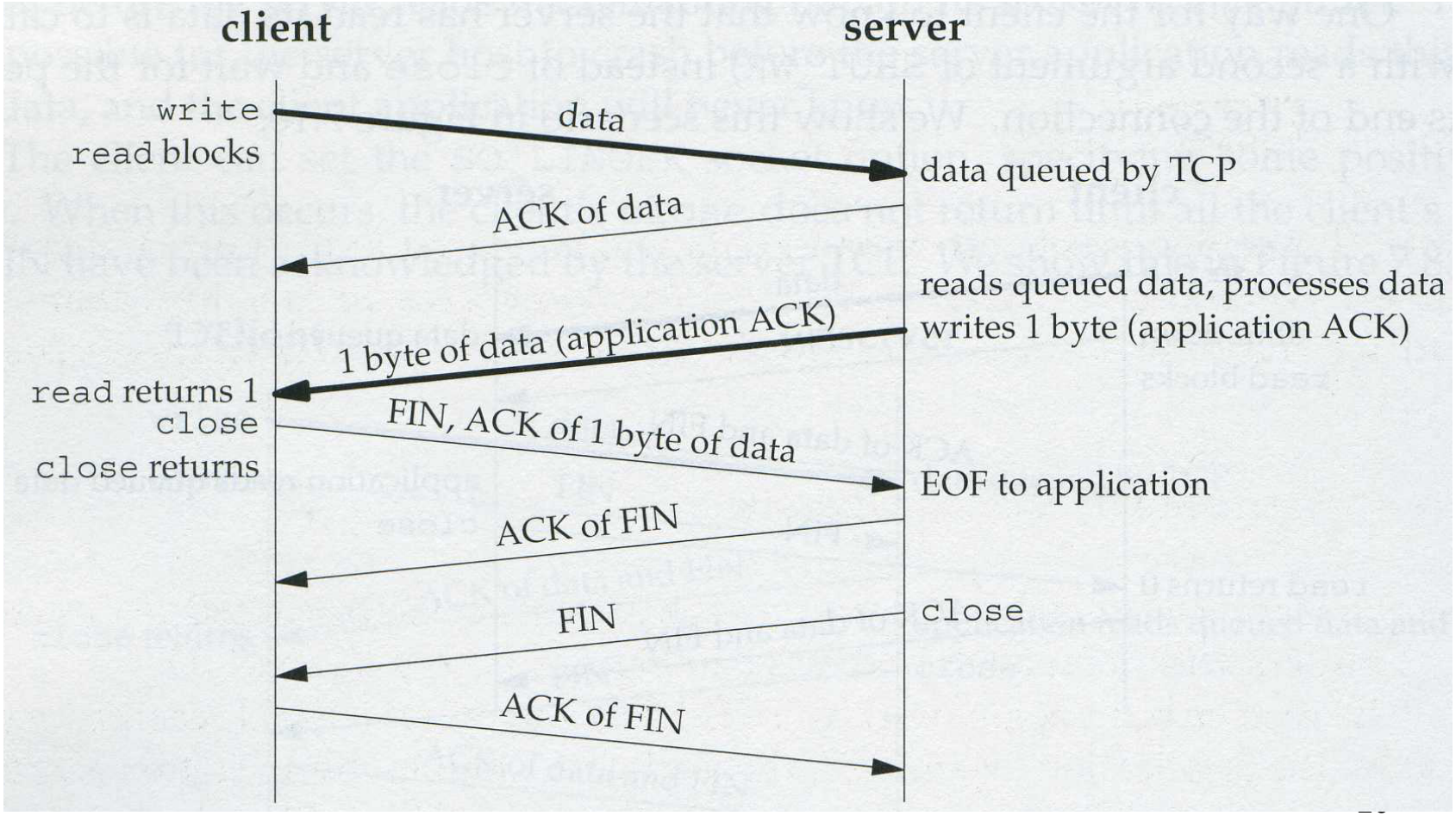
Close() with l_linger set to a small positive value



Using shutdown() to know the peer has read our data



Using application ACK to know the peer has read our data



Generic Socket Options (Cont.)

- SO_RCVBUF/SO_SNDBUF: socket receive / send buffer size, TCP default: 8192-61440, UDP default: 40000/9000

```
size = 4096;
Setsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
```

- SO_RCVLOWAT/SO_SNDLOWAT: receive / send buffer low water mark for *select* to return

Generic Socket Options (Cont.)

- SO_RCVTIMEO/SO_SNDTIMEO: receive / send timeout for socket read/write

```
tv.tv_sec = 5;
tv.tv_usec = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

- SO_TYPE: get socket type, SOCK_STREAM or SOCK_DGRAM

Generic Socket Options (Cont.)

- SO_REUSEADDR/SO_REUSEPORT: allow local address reuse for TCP server restart (Ctrl+C, 2MSL timeout), IP alias, UDP duplicate binding for multicasting

```
const int on = 1;
Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
```

- SO_TYPE: get socket type, SOCK_STREAM or SOCK_DGRAM

IPv4 Socket Options

- IP_HDRINCL: IP header included with data, e.g. *traceroute* builds own IP header on a raw socket
- IP_OPTIONS: specify socket options like source route, timestamp, record route, etc.
- IP_RECDSTADDR: return destination IP address of a received UDP datagram by recvmsg
- IP_RECVIF: return received interface index for a received UDP datagram by recvmsg

IP Socket Options (Cont.)

- IP_TOS: set IP TOS field of outgoing packets for TCP/UDP socket, TOS: IPTOS_LOWDELAY / THROUGHPUT, RELIABILITY / LOWCOST
- IP_TTL: set and fetch the default TTL for outgoing packets, 64 for TCP/UDP sockets, 255 for raw sockets, used in *traceroute*

IPv6 Socket Options

- ICMP6_FILTER: fetch and set icmp6_filter structure specifying message types to pass
- IPV6_ADDFORM: change address format of socket between IPv4 and IPv6
- IPV6_CHECKSUM: offset of checksum field for raw socket
- IPV6_DSTOPTS: return destination options of received datagram by recvmsg
- IPV6_HOPLIMIT: return hop limit of received datagrams by recvmsg

IPv6 Socket Options (Cont.)

- IPV6_HOPOPS: return hop-by-hop options of received datagrams by recvmsg
- IPV6_NEXTHOP: specify next hop address as a socket address structure for a datagram
- IPV6_PKTINFO: return packet info, dest IPv6 and arriving interface, of received datagrams

IPv6 Socket Options (Cont.)

- IPV6_PKTOPTIONS: specify socket options of TCP socket
- IPV6_RTHDR: receive source route
- IPV6_UNICAST_HOPS: ~ IP_TTL
- IPV6_MULTICAST_IF/HOPS/LOOP, IPV6_ADD/DROP_MEMBERSHIP

TCP Socket Options

- TCP_MAXSEG: set TCP max segment size
- TCP_NODELAY: disable Nagle algorithm. The algorithm is to reduce the number of small packets.

fcntl Function

```
#include <fcntl.h>
int fcntl (int fd, int cmd, ... /* int arg */ );
                                    returns: depends on cmd if OK, -1 on error
```

Two flags that affect a socket:
(fetch by F_GETFL, set by F_SETFL)

- O_NONBLOCK (nonblocking I/O)
- O_ASYNC (signal-driven I/O notification)

To set a flag : fetch, OR/AND~, set

```
int flags;
if ( (flags = fcntl (fd, F_GETFL, 0) ) < 0) error_sys ( “F_GETFL error” );
flags |= O_NONBLOCK; /* turn on */
( or flags &= ~O_NONBLOCK; /* turn off */ )
if (fcntl(fd, F_SETFL, flags) < 0) error_sys ( “F_SETFL error” );
```

fcntl Function (Cont.)

Two signals, SIGIO and SIGURG, are sent to the socket owner (process ID or process group ID) if the socket is assigned an owner by F_SETOWN command.

fcntl commands to set or get socket owner: F_SETOWN, F_GETOWN

```
/* allow the process to receive SIGIO */
fcntl(fd, F_SETOWN, getpid());
/* Make the file descriptor asynchronous (the manual page says only
   O_APPEND and O_NONBLOCK, will work with F_SETFL...) */
fcntl(fd, F_SETFL, FASYNC);
```

```
/* open the device to be non-blocking (read will return immediatly) */
fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
if (fd <0) {perror(MODEMDEVICE); exit(-1); }

/* install the signal handler before making the device asynchronous */
saio.sa_handler = signal_handler_IO;
saio.sa_mask = 0;
saio.sa_flags = 0;
saio.sa_restorer = NULL;
sigaction(SIGIO,&saio,NULL);
```

```
/***************************************************************************
* signal handler. sets wait_flag to FALSE, to indicate above loop that     *
* characters have been received.                                           *
***************************************************************************/

void signal_handler_IO (int status)
{
  printf("received SIGIO signal.\n");
  wait_flag = FALSE;
}
```

```
/* loop while waiting for input. normally we would do something
   useful here */
while (STOP==FALSE) {
  printf(".\n");usleep(100000);
  /* after receiving SIGIO, wait_flag = FALSE, input is available
     and can be read */
  if (wait_flag==FALSE) {
    res = read(fd,buf,255);
    buf[res]=0;
    printf(":%s:%d\n", buf, res);
    if (res==1) STOP=TRUE; /* stop loop if only a CR was input */
    wait_flag = TRUE;      /* wait for new input */
  }
}
```

- One limitation of this FASYNC approach is that if the application has multiple descriptors, the delivery of SIGIO dost not indicate which descriptor is readable.
- To solve this problem, after receiving SIGIO, the application still need to use select() to test and check which descriptor(s) are readable.

Signal-Driven I/O

- If the socket is a UDP socket, an SIGIO will be sent only when a datagram arrives for the UDP socket.
- However, there are two many conditions that may cause SIGIO to be generated for a TCP socket, making signal-driven I/O of little use to TCP sockets.
- Because signals are not queued, normally a signal-driven I/O handler needs to set the socket to the nonblocking mode and read() it in a loop to receive all UDP datagrams that have arrived. Otherwise, the read() may be blocked!

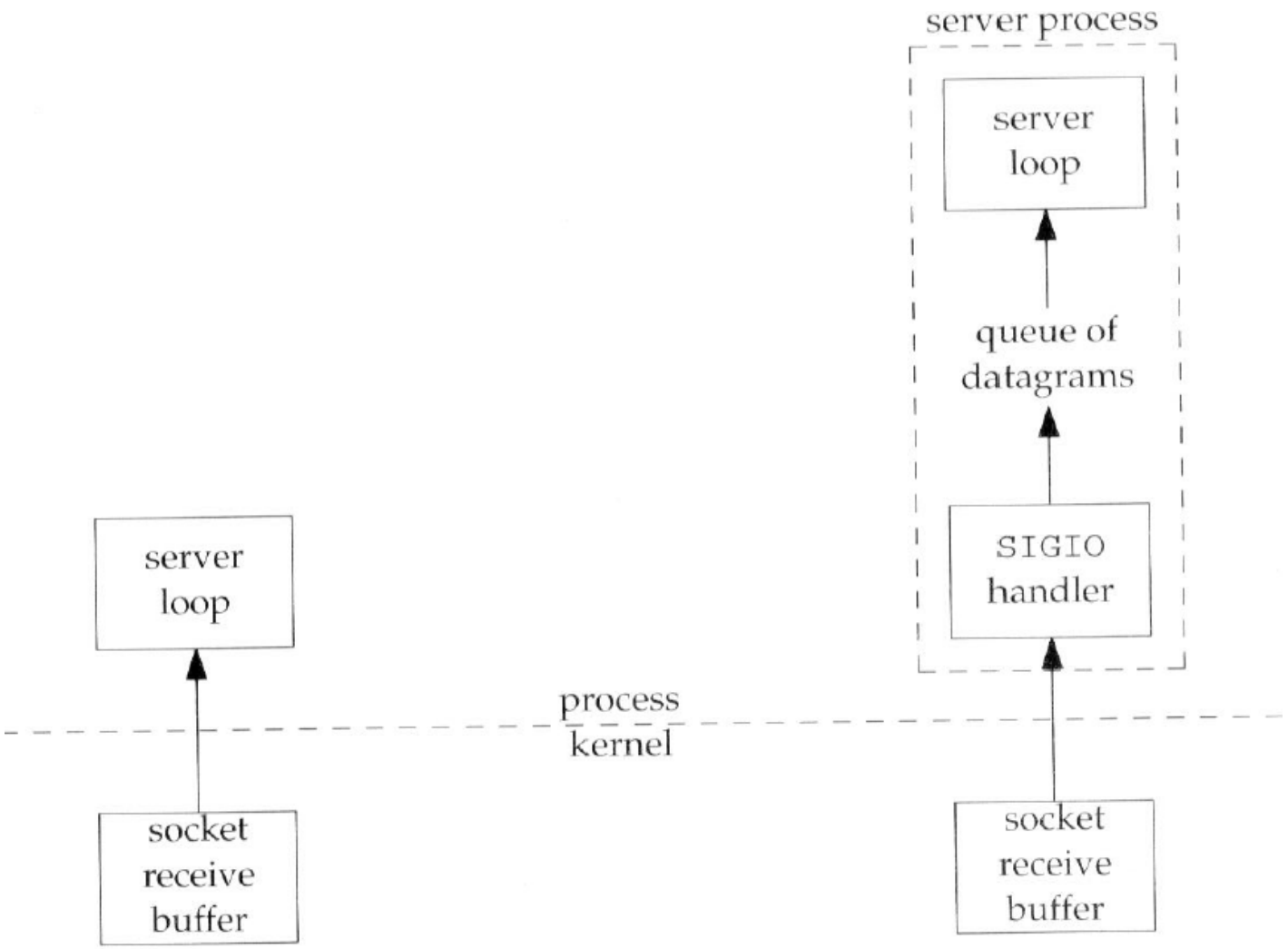


Figure 25.1 Two different ways to build a UDP server.

sigio/dgecho01.c

```
 1 #include    "unp.h"

 2 static int sockfd;

 3 #define QSIZE      8              /* size of input queue */
 4 #define MAXDG   4096              /* max datagram size */

 5 typedef struct {
 6     void   *dg_data;              /* ptr to actual datagram */
 7     size_t  dg_len;               /* length of datagram */
 8     struct sockaddr *dg_sa;       /* ptr to sockaddr{} w/client's address */
 9     socklen_t dg_salen;           /* length of sockaddr{} */
10 } DG;
11 static DG dg[QSIZE];             /* queue of datagrams to process */
12 static long cntread[QSIZE + 1]; /* diagnostic counter */

13 static int iget;                 /* next one for main loop to process */
14 static int iput;                 /* next one for signal handler to read into */
15 static int nqueue;               /* # on queue for main loop to process */
16 static socklen_t clilen;         /* max length of sockaddr{} */

17 static void sig_io(int);
18 static void sig_hup(int);
```

sigio/dgecho01.c

```
19 void
20 dg_echo(int sockfd_arg, SA *pcliaddr, socklen_t clilen_arg)
21 {
22     int     i;
23     const int on = 1;
24     sigset_t zeromask, newmask, oldmask;

25     sockfd = sockfd_arg;
26     clilen = clilen_arg;

27     for (i = 0; i < QSIZE; i++) {   /* init queue of buffers */
28         dg[i].dg_data = Malloc(MAXDG);
29         dg[i].dg_sa = Malloc(clilen);
30         dg[i].dg_salen = clilen;
31     }
32     iget = iput = nqueue = 0;

33     Signal(SIGHUP, sig_hup);
34     Signal(SIGIO, sig_io);
35     Fcntl(sockfd, F_SETOWN, getpid());
36     Ioctl(sockfd, FIOASYNC, &on);
37     Ioctl(sockfd, FIONBIO, &on);

38     Sigemptyset(&zeromask);      /* init three signal sets */
39     Sigemptyset(&oldmask);
40     Sigemptyset(&newmask);
41     Sigaddset(&newmask, SIGIO); /* signal we want to block */

42     Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
43     for ( ; ; ) {
44         while (nqueue == 0)
45             sigsuspend(&zeromask);  /* wait for datagram to process */

46             /* unblock SIGIO */
47         Sigprocmask(SIG_SETMASK, &oldmask, NULL);

48         Sendto(sockfd, dg[iget].dg_data, dg[iget].dg_len, 0,
49                dg[iget].dg_sa, dg[iget].dg_salen);

50         if (++iget >= QSIZE)
51             iget = 0;

52             /* block SIGIO */
53         Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
54         nqueue--;
55     }
56 }
```

sigio/dgecho01.c

```
57 static void
58 sig_io(int signo)
59 {
60     ssize_t len;
61     int     nread;
62     DG     *ptr;

63     for (nread = 0;;) {
64         if (nqueue >= QSIZE)
65             err_quit("receive overflow");

66         ptr = &dg[iput];
67         ptr->dg_salen = clilen;
68         len = recvfrom(sockfd, ptr->dg_data, MAXDG, 0,
69                        ptr->dg_sa, &ptr->dg_salen);
70         if (len < 0) {
71             if (errno == EWOULDBLOCK)
72                 break;          /* all done; no more queued to read */
73             else
74                 err_sys("recvfrom error");
75         }
76         ptr->dg_len = len;

77         nread++;
78         nqueue++;
79         if (++iput >= QSIZE)
80             iput = 0;

81     }
82     cntread[nread]++;           /* histogram of # datagrams read per signal */
83 }
```

sigio/dgecho01.c

Summary

- Commonly used socket options: SO_KEEPALIVE, SO_RCVBUF, SO_SNDBUF, SO_REUSEADDR
- SO_REUSEADDR set in TCP server before calling bind
- SO_LINGER gives more control over when close returns and forces RST to be sent
- SO_RCVBUF/SO_SNDBUF for bulk data transfer across long fat pipes