I/O Multiplexing: *select* and *poll*

- Introduction
- I/O models
- *select* function
- Rewrite *str_cli* function
- Supporting batch input with *shutdown* function
- Rewrite concurrent TCP echo server with *select*
- *pselect* function: avoiding signal loss in race condition
- *poll* function: polling more specific conditions than *select*
- Rewrite concurrent TCP echo server with *poll*

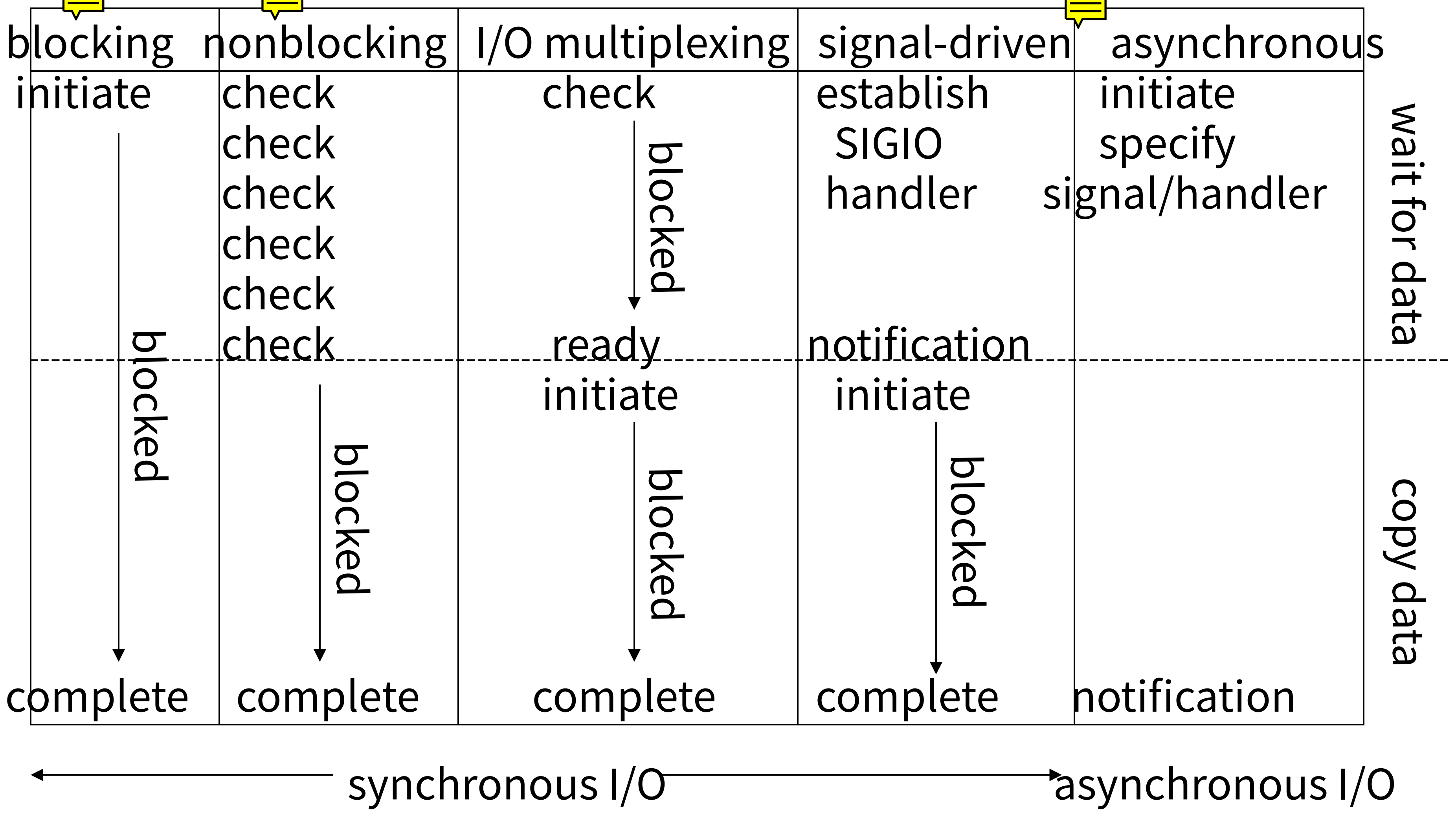
Introduction

- I/O multiplexing: to be notified, by kernel, if one or more I/O conditions are ready.
- Scenarios in networking applications:
  - a client handling multiple descriptors (stdio/socket)
  - a client handling multiple sockets
  - a TCP server handling a listening socket and its connected sockets
  - a server handling both TCP and UDP
  - a server handling multiple services and protocols

I/O Models

- Two distinct phases for an input operation:
  - **wait** for data; **copy** data from kernel to user
- Five I/O models:
  - blocking I/O: blocked all the way
  - nonblocking I/O: if no data, immediate returns EWOULDBLOCK
  - I/O multiplexing (*select* and *poll*): blocked separately in wait and copy
  - signal driven I/O (SIGIO): nonblocked in wait but blocked in copy (signaled when I/O can be initiated)
  - asynchronous I/O (*aio_*): nonblocked all the way (signaled when I/O is complete)

Comparison of Five I/O Models



select Function

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
            const struct timeval *timeout);
          returns: postive count of ready descriptors, 0 on timeout, -1 on error

struct timeval {              (null: wait forever; 0: do not wait)
          long tv_sec; /*second */
          long tv_usec; /* microsecond */ };

fd_set --- implementation dependent
            four macros:  void FD_ZERO(fd_set *fdset);
                          void FD_SET(int fd, fd_set *fdset);
                          void FD_CLR(int fd, fd_set *fdset);
                          int FD_ISSET(int fd, fd_set *fdset);
```

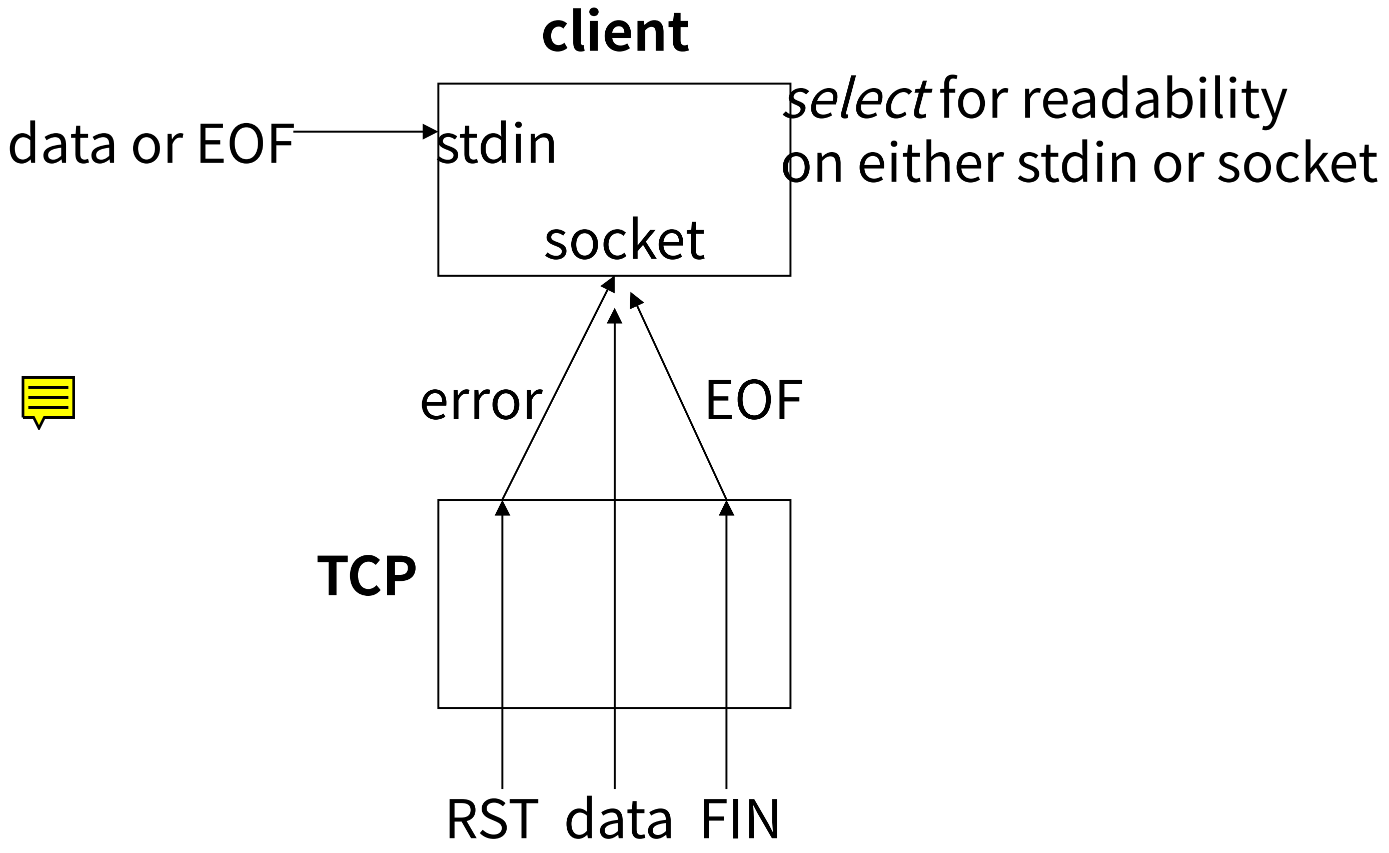
Socket Ready Conditions for *select*

Condition	readable?	writeable?	Exception?
enough data to read	x		
read-half closed	x		
new connection ready	x		
writing space available		x	
write-half closed		x	
pending error	x	x	
TCP out-of-band data			x

Low-water mark (enough data/space to read/write in
socket receive/send buffer): default is 1/2048, may be set by
SO_RCVLOWAT/SO_SNDLOWAT socket option

Maximum number of descriptors for select?
Redefine FD_SETSIZE and recompile kernel

Rewrite *str_cli* Function with *select*



Rewrite *str_cli* Function with *select*

```
#include      "unp.h"                                   select/strcliselect01.c

void
str_cli(FILE *fp, int sockfd)
{
      int              maxfdp1;
      fd_set        rset;
      char          sendline[MAXLINE], recvline[MAXLINE];

      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);
```

select/strcliselect01.c

```
 if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
                        if (Readline(sockfd, recvline, MAXLINE) == 0)
                                err_quit("str_cli: server terminated prematurely
");
                        Fputs(recvline, stdout);
                }

                if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
                        if (Fgets(sendline, MAXLINE, fp) == NULL)
                                return;         /* all done */
                        Writen(sockfd, sendline, strlen(sendline));
                }
        }
}
```

Supporting Batch Input with *shutdown*

- Stop-and-wait mode (interactive) vs batch mode (redirected stdin/stdout)
- In batch mode, *str_cli* returns right after EOF on input and *main* returns immediately, while leaving server replies unprocessed.
- Solution: In *str_cli*, close write-half of TCP connection, by *shutdown*, while leaving read-half open.

include <sys/socket.h>
nt shutdown (int *sockfd*, int *howto*); returns: 0 if OK, -1 on error
owto: SHUT_RD, SHUT_WR, SHUT_RDWR
nititate TCP normal termination regardless of descriptor's reference count

Rewrite *str_cli* with *select* and *shutdown*

select/strcliselect02.c

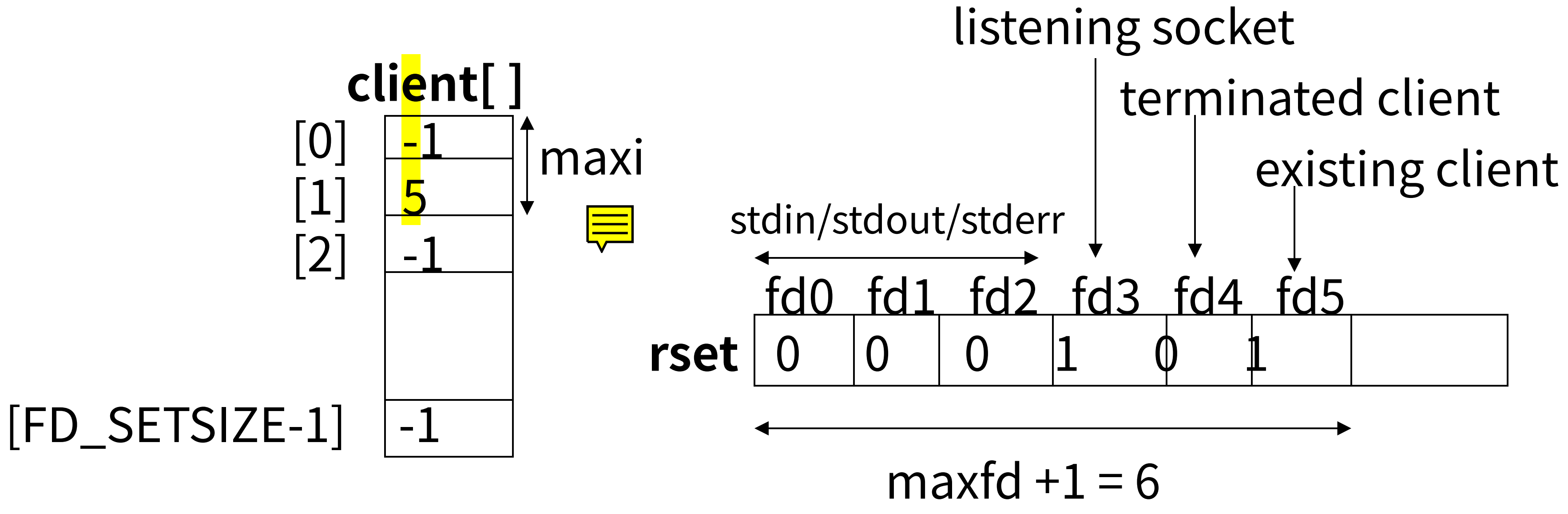
```
#include      "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int             maxfdp1, stdineof;
    fd_set          rset;
    char            sendline[MAXLINE], recvline[MAXLINE];

    stdineof = 0;
    FD_ZERO(&rset);
    for ( ; ; ) {
        if (stdineof == 0)
            FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd) + 1;
        Select(maxfdp1, &rset, NULL, NULL, NULL);
```

select/strcliselect02.c

```
        if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0) {
                if (stdineof == 1)
                    return;       /* normal termination */
                else
                    err_quit("str_cli: server terminated prematurely");
            }
            Fputs(recvline, stdout);
        }
        if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL) {
                stdineof = 1;
                Shutdown(sockfd, SHUT_WR);    /* send FIN */
                FD_CLR(fileno(fp), &rset);
                continue;
            }
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

Rewrite TCP Echo Server with *select*

- A single server process using *select* to handle any number of clients
- Need to keep track of the clients by *client[ ]* (client descriptor array) and *rset* (read descriptor set)



Rewrite TCP Echo Server with *select*

tcpcliserv/tcpservselect01.c

Initialization

```
#include	"unp.h"
int main(int argc, char **argv)
{
	int			i, maxi, maxfd, listenfd, connfd, sockfd;
	int			nready, client[FD_SETSIZE];
	ssize_t			n;
	fd_set			rset, allset;
	char			line[MAXLINE];
	socklen_t		clilen;
	struct sockaddr_in	cliaddr, servaddr;

	listenfd = Socket(AF_INET, SOCK_STREAM, 0);

	bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family      = AF_INET;
	servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servaddr.sin_port        = htons(SERV_PORT);
```

tcpcliserv/tcpservselect01.c

Initialization (cont.)

```
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

 Listen(listenfd, LISTENQ);

 maxfd = listenfd;/* initialize */
 maxi = -1;/* index into client[] array */
 for (i = 0; i < FD_SETSIZE; i++)
 client[i] = -1;/* -1 indicates available entry */
 FD_ZERO(&allset);
 FD_SET(listenfd, &allset);
```

Loop

tcpcliserv/tcpservselect01.c

```
for ( ; ; ) {
        rset = allset;          /* structure assignment */
        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
        if (FD_ISSET(listenfd, &rset)) {        /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
             #ifdef  NOTDEF
            printf("new client: %s, port %d\n",
                    Inet_ntop(AF_INET, &cliaddr.sin_addr, 4, NULL),
                    ntohs(cliaddr.sin_port));
             #endif
            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0) {
                    client[i] = connfd;     /* save descriptor */
                    break;
                }
            if (i == FD_SETSIZE)
                err_quit("too many clients");
             FD_SET(connfd, &allset);       /* add new descriptor to set */
            if (connfd > maxfd)
                maxfd = connfd;             /* for select */
            if (i > maxi)
                maxi = i;                   /* max index in client[] array */
            if (--nready <= 0)
                continue;                   /* no more readable descriptors */
        }
```

Loop (cont.)

tcpcliserv/tcpservselect01.c

```
                    for (i = 0; i <= maxi; i++) {   /* check all clients for data */
                     if ( (sockfd = client[i]) < 0)
                         continue;
                     if (FD_ISSET(sockfd, &rset)) {
                         if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
                                 /* connection closed by client */
                             Close(sockfd);
                             FD_CLR(sockfd, &allset);
                             client[i] = -1;
                         } else
                             Writen(sockfd, line, n);
                         if (--nready <= 0)
                             break;              /* no more readable descriptors */
                    }
                }
            }
        }
```

Denial of Service Attacks

- Problem of concurrent TCP echo server with *select* : blocks in a function call, *read* in *readline*, related to a single client
- Attack scenario:
  - a malicious client sends 1 byte of data (other than a newline) and sleep
  - server hangs until the malicious client either sends a newline or terminates
- Solutions:
  - nonblocking I/O for the listening socket
  - separate thread/process for each client
  - timeout on I/O operations

pselect Function: Avoiding Signal Loss in Race Conditio

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
             const struct timespec *timeout, const sigset_t *sigmask);
          returns: count of ready descriptors, 0 on timeout, -1 on error
struct timespec {                    time_t tv_sec;  /* seconds */
                             long tv_nsec;  /* nanosecond */ };
```

```
if (intr_flag)
   handle_intr( ); /* handle signal */
if ((nready = select ( ... ))< 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr( );
     }
     ....
}
```

signal lost if *select* blocks forever

```
sigemptyset (&zeromask);
sigemptyset (&newmask);
sigaddset (&newmask, SIGINT);
sigprocmask (SIG_BLOCK, &newmask, &oldmask);
if (intr_flag)
    handle_intr( );
if ( (nready = pselect ( ... , &zeromask)) < 0) {
      if (errno == EINTR) {
          if (intr_flag)
               handle_intr( );
      }
   ....}
```

poll Function: polling more specific conditions than *select*

```
#include <poll.h>
int poll (struct pollfd  *fdarray, unsigned long ndfs, int timeout);
          returns: count of ready descriptors, 0 on timeout, -1 on error
struct pollfd {
          int fd;             /* a descriptor to poll */
          short events;       /* events of intereston fd, value argument */
          short revents;      /* events that occurred on fd, result argument */
};
```

Constant	events	revents	Description
POLLIN	x	x	normal or priority band to read
POLLRDNORM	x	x	normal data to read
POLLRDBAND	x	x	priority band data to read
POLLPRI	x	x	high-priority data to read
POLLOUTx	x	normal data to write	
POLLWRNORM	x	x	normal data to write
POLLWRBAND	x	x	priority band data to write
POLLERR	x	error occurred	
POLLHUP		x	hangup occurred
POLLNVAL		x	descriptor is not an open file

Concurrent TCP Echo Server with *poll*

- When using *select*, the server maintains array *client[ ]* and descriptor set *rset*. When using poll, the server maintains arrary *client* of pollfd structured.
- Program flow:
  - allocate array of pollfd structures
  - initialize (listening socket: first entry in *client*)
    (set POLLRDNORM in *events*)
  - call poll; check for new connection
    (check, in *revents*, and set, in *events*, POLLRDNORM)
  - check for data on an existing connection
    (check POLLRDNORM or POLLERR in *revents*)

Rewrite Concurrent TCP Echo Server with *poll*

Initialization

tcpcliserv/tcpservpoll01.c

```
#include      "unp.h"
#include      <limits.h>          /* for OPEN_MAX */
int main(int argc, char **argv)
{
     int                      i, maxi, listenfd, connfd, sockfd;
     int                      nready;
     ssize_t                n;
     char                   line[MAXLINE];
     socklen_t                clilen;
     struct pollfd         client[OPEN_MAX];
     struct sockaddr_in     cliaddr, servaddr;
     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
     bzero(&servaddr, sizeof(servaddr));
     servaddr.sin_family      = AF_INET;
     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
     servaddr.sin_port        = htons(SERV_PORT);
```

tcpcliserv/tcpservpoll01.c

Initialization (cont.)

```
    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

        Listen(listenfd, LISTENQ);

        client[0].fd = listenfd;
        client[0].events = POLLRDNORM;
        for (i = 1; i < OPEN_MAX; i++)
            client[i].fd = -1;          /* -1 indicates available entry */
        maxi = 0;                      /* max index into client[] array */
```

Loop

tcpcliserv/tcpservpoll01.c

```
for ( ; ; ) {
        nready = Poll(client, maxi+1, INFTIM);

        if (client[0].revents & POLLRDNORM) {   /* new client connection */
             clilen = sizeof(cliaddr);
             connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
              #ifdef  NOTDEF
             printf("new client: %s\n", Sock_ntop((SA *) &cliaddr, clilen));
              #endif
             for (i = 1; i < OPEN_MAX; i++)
                  if (client[i].fd < 0) {
                       client[i].fd = connfd;  /* save descriptor */
                       break;
                  }
             if (i == OPEN_MAX)
                  err_quit("too many clients");
             client[i].events = POLLRDNORM;
             if (i > maxi)
                    maxi = i;                    /* max index in client[] array */
             if (--nready <= 0)
                  continue;                    /* no more readable descriptors */
        }
```

Loop (cont.)

tcpcliserv/tcpservpoll01.c

```
for (i = 1; i <= maxi; i++) {   /* check all clients for data */
            if ( (sockfd = client[i].fd) < 0)
                continue;
            if (client[i].revents & (POLLRDNORM | POLLERR)) {
                if ( (n = readline(sockfd, line, MAXLINE)) < 0) {
                    if (errno == ECONNRESET) {
                                /* connection reset by client */
                               #ifdef  NOTDEF
                        printf("client[%d] aborted connection\n", i);
                               #endif
                        Close(sockfd);
                        client[i].fd = -1;
                    } else
                        err_sys("readline error");
                } else if (n == 0) {
                        /*4connection closed by client */
                           #ifdef  NOTDEF
                    printf("client[%d] closed connection\n", i);
                           #endif
                    Close(sockfd);
                    client[i].fd = -1;
                } else
                    Writen(sockfd, line, n);
                if (--nready <= 0)
                    break;                  /* no more readable descriptors */
 } } } }
```