BITS Pilani
Pilani Campus

Network Programming

K Hari Babu
Department of Computer Science & Information Systems

Outline

Outline

- I/O Multiplexing (cont..)
  - Client using select()
  - shutdown()
  - Concurrent server using select()
- Non-blocking sockets
  - ○ Nonblocking read(), write()
  - ○ Nonblocking connect()

innovate achieve lead

BITS Pilani

Pilani Campus

I/O Multiplexing

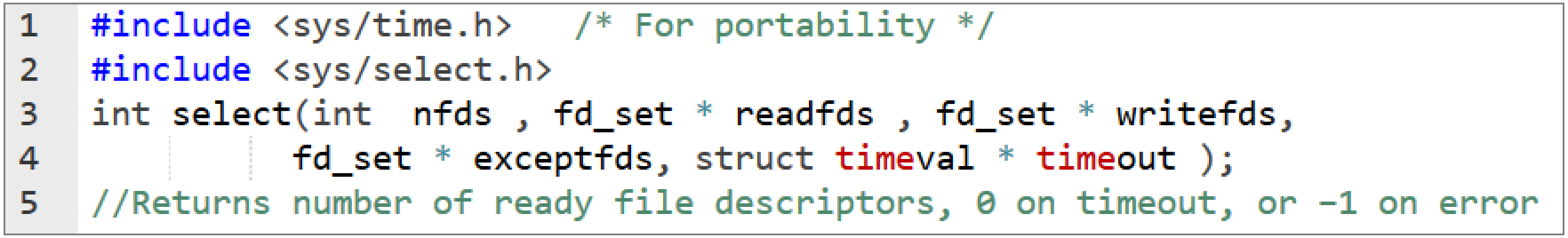
T1: Ch6

I/O Multiplexing

- I/O multiplexing allows us to simultaneously monitor multiple file descriptors to see if I/O is possible on any of them.
- select(), appeared along with the sockets API in BSD. This was historically the more widespread of the two system calls. The other system call, poll(), appeared in System V.
- We can use select() and poll() to monitor file descriptors for regular files, terminals, pseudoterminals, pipes, FIFOs, sockets, and some types of character devices.
- Both system calls allow a process either to block indefinitely waiting for file descriptors to become ready or to specify a timeout on the call.

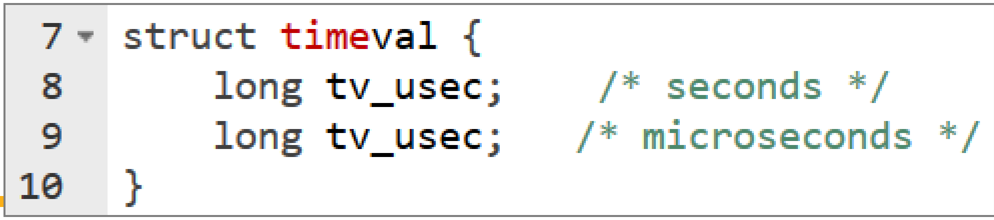
select()

- The select() system call blocks until one or more of a set of file descriptors becomes ready.



```c
#include <sys/time.h>    /* For portability */
#include <sys/select.h>
int select(int  nfds , fd_set * readfds , fd_set * writefds,
          fd_set * exceptfds, struct timeval * timeout );
//Returns number of ready file descriptors, 0 on timeout, or -1 on error
```

- ○ *nfds*: highest number assigned to a descriptor +1.
- ○ *readfds*: set of descriptors we want to read from.
- ○ *writefds*: set of descriptors we want to write to.
- ○ *exceptfds*: set of descriptors to watch for exceptions.
- ○ *timeout*: maximum time select should wait



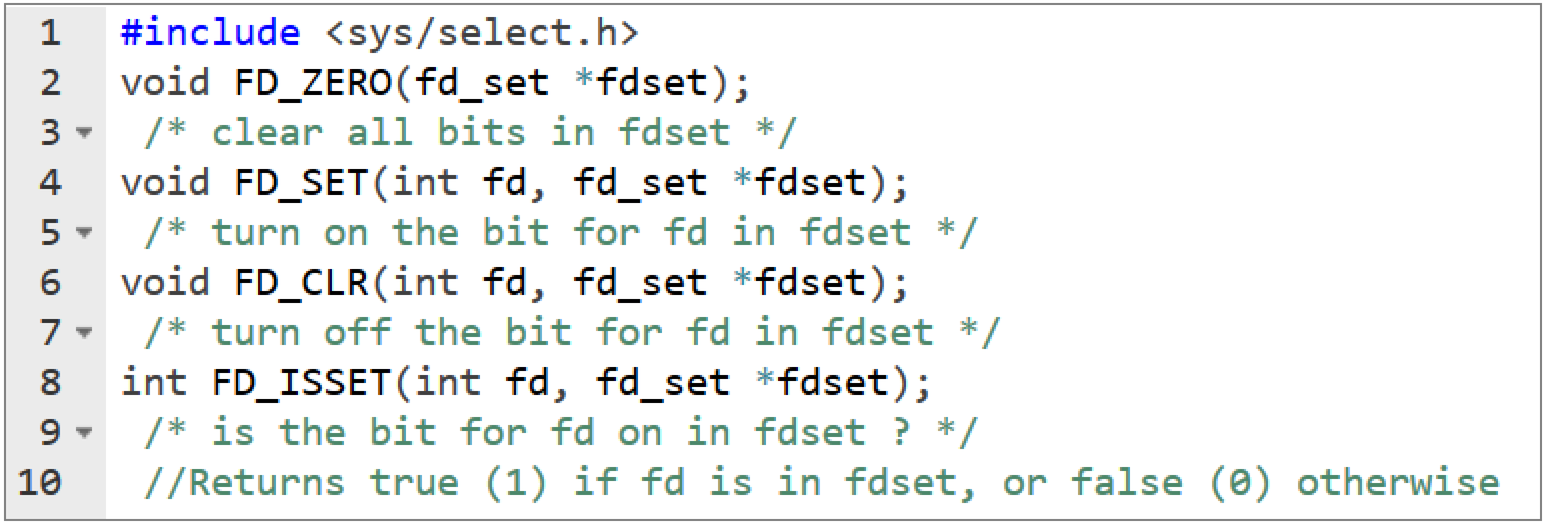
```c
struct timeval {
    long tv_usec;     /* seconds */
    long tv_usec;   /* microseconds */
}
```

select()

- timeval==NULL
  - Wait forever : return only when descriptor is ready
- timeval != NULL: wait up to a fixed amount of time
  - timeval = 0
    - Do not wait at all : return immediately after checking the descriptors
  - Timeval>0
    - Return only if descriptor is ready or timeval expires.

File descriptor sets

- The readfds, writefds, and exceptfds arguments are pointers to file descriptor sets, represented using the data type *fd_set*.
- the fd_set data type is implemented as a bit mask.



```
#include <sys/select.h>
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);
 /* is the bit for fd on in fdset ? */
 //Returns true (1) if fd is in fdset, or false (0) otherwise
```

- A file descriptor set has a maximum size, defined by the constant FD_SETSIZE . On Linux, this constant has the value 1024.

select()

- *nfds*
  - Its value is the maximum descriptor to be tested, plus one
    - example: fds 1,2,5 => nfds: 6
- *readset*
  - descriptor set for checking readable
- *writeset*
  - descriptor set for checking writable
- *exceptset*
  - descriptor set for checking two exception conditions
    - arrival of out of band data for a socket
    - he presence of control status information to be read from the master side of a pseudo terminal
- When select returns value>1, these sets have been modified by kernel. Now they contain the fds which are ready.

When is the descriptor ready for reading?

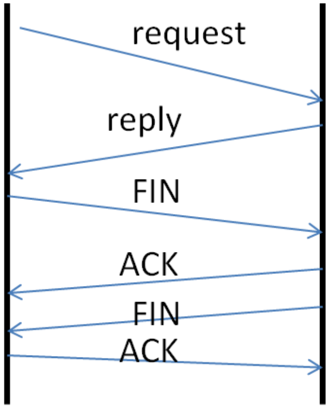
- The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. SO_RCVLOWAT socket option. It defaults to 1 for TCP and UDP sockets
- The read half of the connection is closed (i.e., a TCP connection that has received a FIN)
- The socket is a listening socket and the number of completed connections is nonzero.
- A socket error is pending. A read operation on the socket will not block and will return an error (–1) with errno set to the specific error condition.

When the socket is ready for writing?

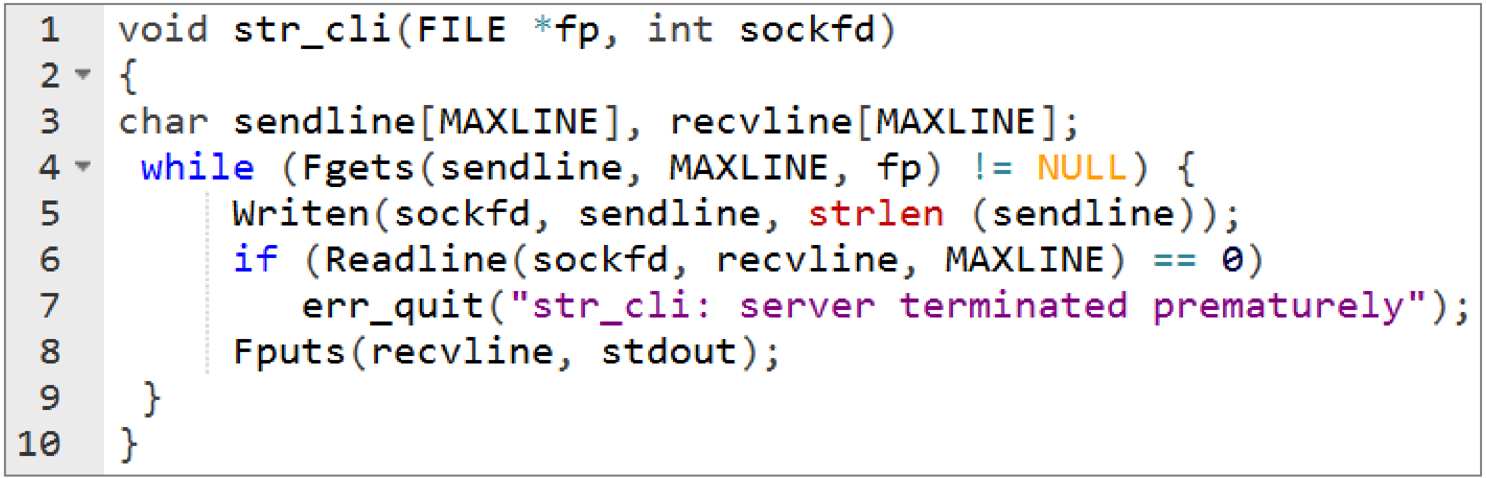
- The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer. 2048 bytes.
- The write half of the connection is closed. A write operation on the socket will generate SIGPIPE.
- A socket using a non-blocking connect has completed the connection, or the connect has failed
- A socket error is pending. A write operation on the socket will not block and will return an error (–1) with errno set to the specific error condition.
- These pending errors can also be fetched and cleared by calling getsockopt with the SO_ERROR socket option.

Client Handling Multiple Descriptors

- A client is handling two descriptors.
  - ○ *stdin*
  - ○ *socket*
- Sequential handling:
  - ○ First wait on *stdin.*
  - ○ Write to *socket*
  - ○ Read from *socket*.
  - ○ Write to *stdout*.



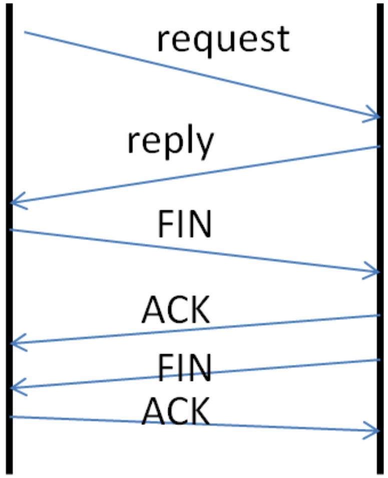
Normal course of actions



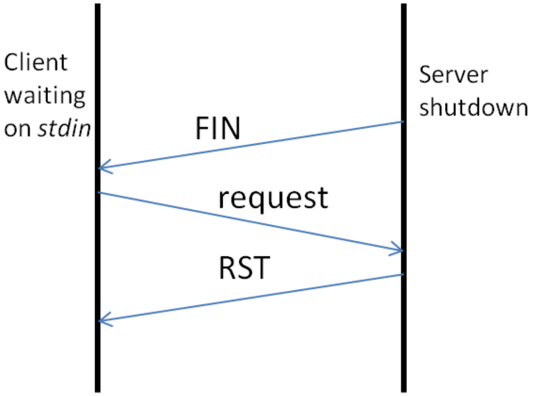
```
void str_cli(FILE *fp, int sockfd)
{
char sendline[MAXLINE], recvline[MAXLINE];
 while (Fgets(sendline, MAXLINE, fp) != NULL) {
     Writen(sockfd, sendline, strlen (sendline));
     if (Readline(sockfd, recvline, MAXLINE) == 0)
        err_quit("str_cli: server terminated prematurely");
     Fputs(recvline, stdout);
 }
}
```

Client Handling Multiple Descriptors

- read() call on both stdin and socket will block until data is available.
- Consider a case:
  - If client is blocked in waiting for user to enter data, meanwhile TCP receives FIN from server.
    - Server is down. So sending request is meaningless.
- How to handle uncertainty of availability of data on descriptors?



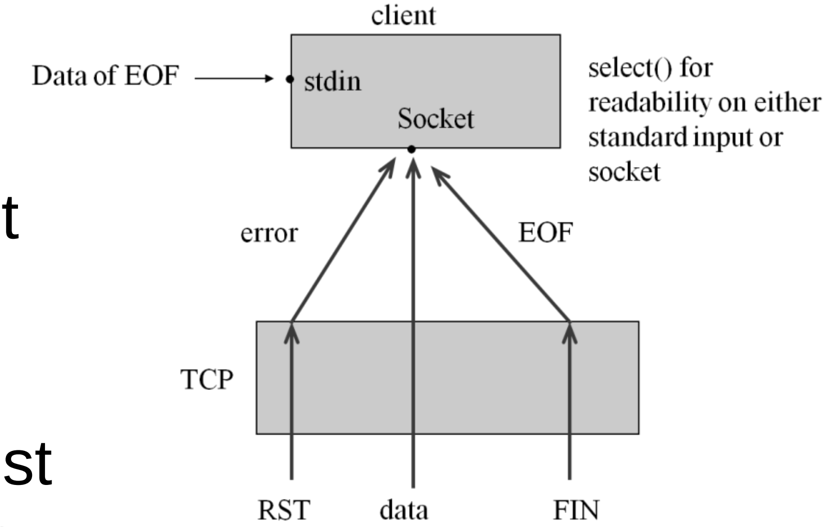
Normal course of actions



Unexpected Server shutdown

read() on socket

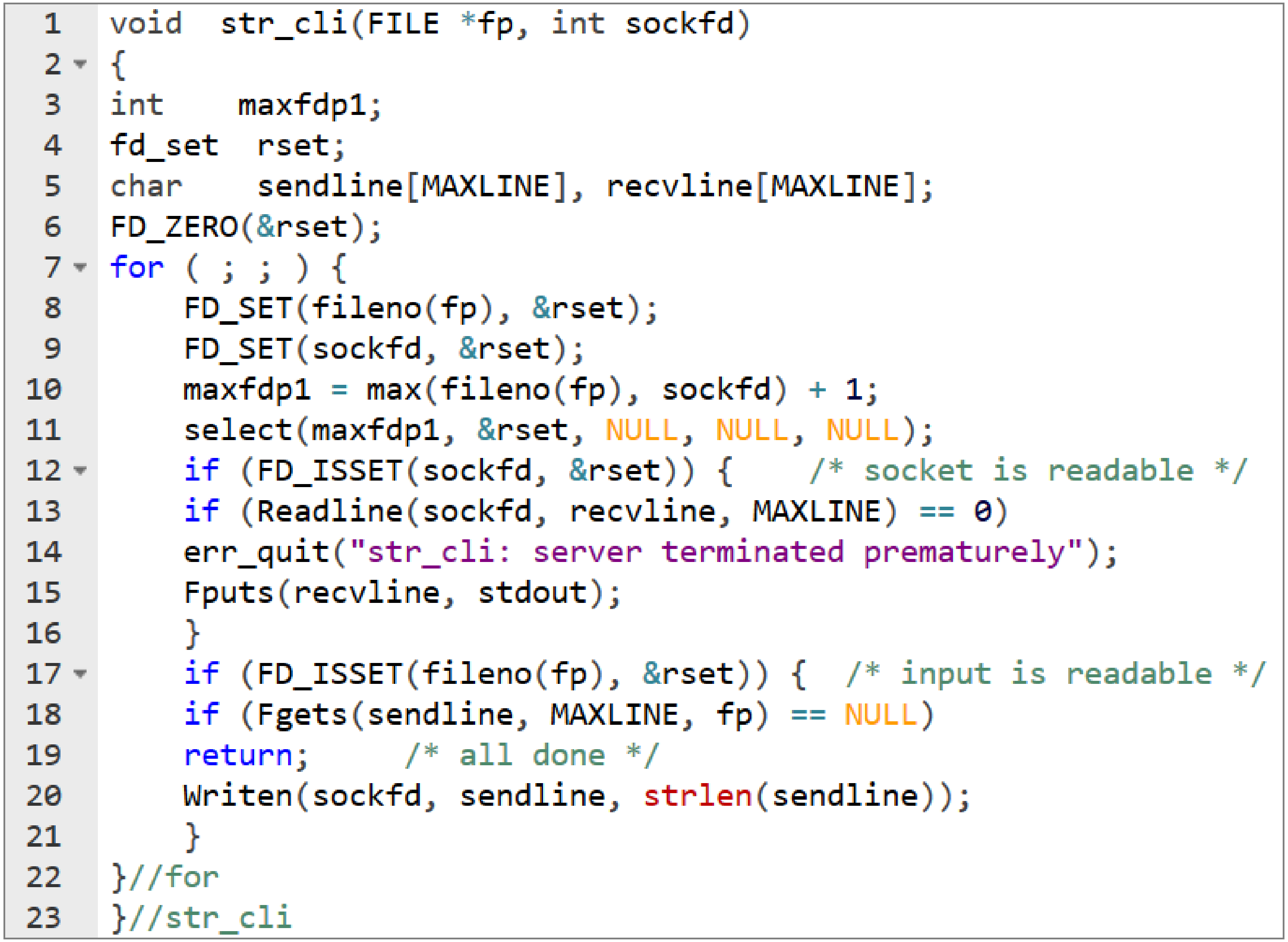
- Peer TCP sends data, the socket becomes readable and *read* returns greater than 0.
- Peer TCP send a FIN(peer process terminates), the socket become readable and *read* returns 0(end-of-file)
- Peer TCP send a RST(peer host has crashed and rebooted), the socket become readable and returns -1 and *errno* contains the specific error code



Client Handling Multiple Descriptors

- To avoid a situation where data has arrived from socket but client is unable to take note of it, use I/O Multiplexing.
- Client can wait on select().
  - Add stdin, socket to fd_set.
  - Call select () with fd_set for readabliity
  - When select() returns, find out which descriptor is ready with data.
  - Call read() on that fd.
- This will enable client to give timely response and avoid error situations.

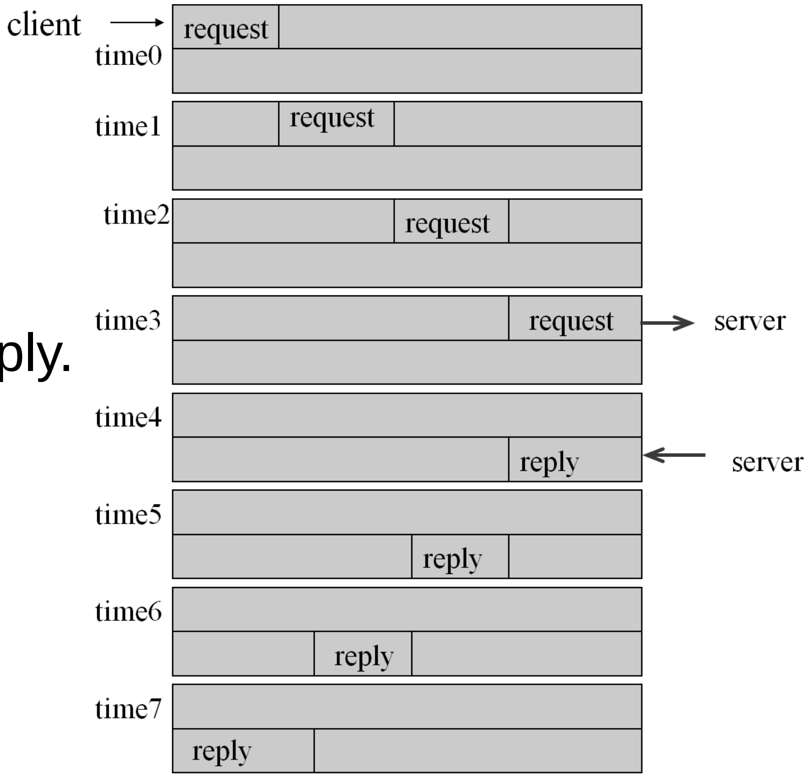
Client Handling Multiple Descriptors



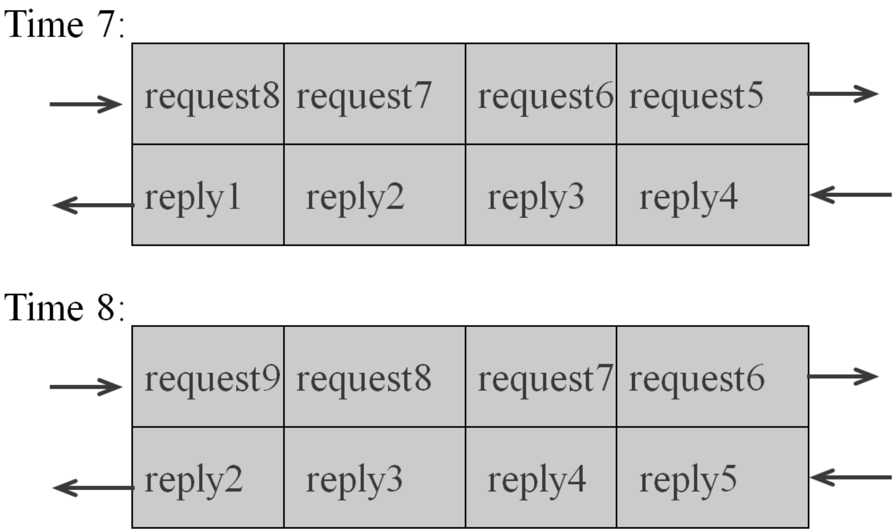
```c
void  str_cli(FILE *fp, int sockfd)
{
int    maxfdp1;
fd_set  rset;
char    sendline[MAXLINE], recvline[MAXLINE];
FD_ZERO(&rset);
for ( ; ; ) {
    FD_SET(fileno(fp), &rset);
    FD_SET(sockfd, &rset);
    maxfdp1 = max(fileno(fp), sockfd) + 1;
    select(maxfdp1, &rset, NULL, NULL, NULL);
    if (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));
    }
}//for
}//str_cli
```

Batch Mode Client

- Stop and Wait client
  - ○ Send one request and wait for reply.
  - ○ Usual in interactive mode.
- Batch Mode clients
  - ○ Send requests without waiting for reply.
  - ○ Better bandwidth utilization.



client → time0: request
time1: request
time2: request
time3: request → server
time4: reply ← server
time5: reply
time6: reply
time7: reply



Time 7:

request8	request7	request6	request5
reply1	reply2	reply3	reply4

Time 8:

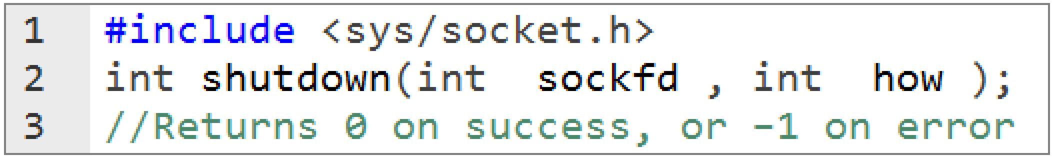
request9	request8	request7	request6
reply2	reply3	reply4	reply5

Batch Mode Client

- Need for closing a socket partially:
  - We tell server that we have sent all requests by closing socket.
    - It will send FIN to server.
  - But in batch mode, by closing socket, we send FIN but we can't read replies which are yet to reach the client.
- close() vs shutdown() sys calls
  - closes the socket partially (either read end or write end) unlike close() sys call.
    - close() closes completely.
  - Irrespective of reference count it closes the socket.
    - close() will initiate FIN only if reference count for the fd reaches 0.

shutdown() sys call

- Sometimes, it is useful to close one half of the connection, so that data can be transmitted in just one direction through the socket.

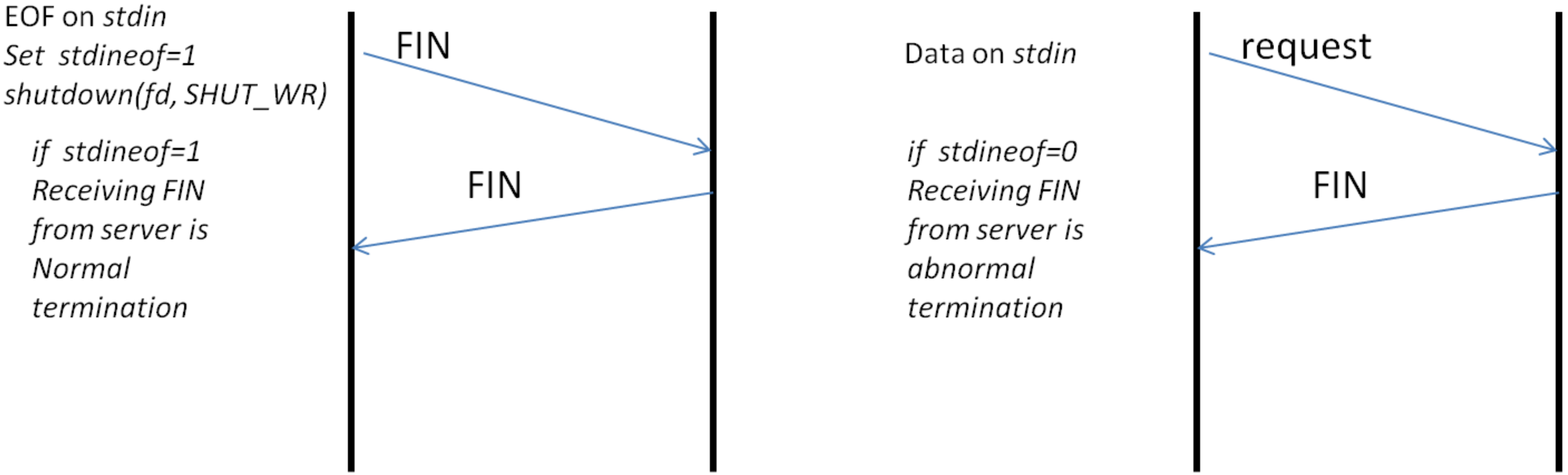


```c
#include <sys/socket.h>
int shutdown(int  sockfd , int  how );
//Returns 0 on success, or -1 on error
```

  - SHUT_RD : read-half of the connection closed. Subsequent reads will return end-of-file (0).
    - SHUT_RD can't be used meaningfully for TCP sockets.
  - SHUT_WR : write-half of the connection closed. Also called *socket half-close*. Buffered data will be sent followed by termination sequence.
    - Common use of shutdown()
    - Subsequent writes to the local socket yield the SIGPIPE signal and an EPIPE error.
  - SHUT_RDWR : both closed
    - Note that shutdown() doesn't close the file descriptor, even if how is specified as SHUT_RDWR . To close the file descriptor, we must additionally call close().

Batch Mode Client

- After user presses, Ctrl-D (EOF), close write half of the socket.
- Also set *stdineof* variable to 1.
  - This will help in inferring the FIN received from server as normal or abnormal termination.
  - In case of normal termination, we received all the replies.

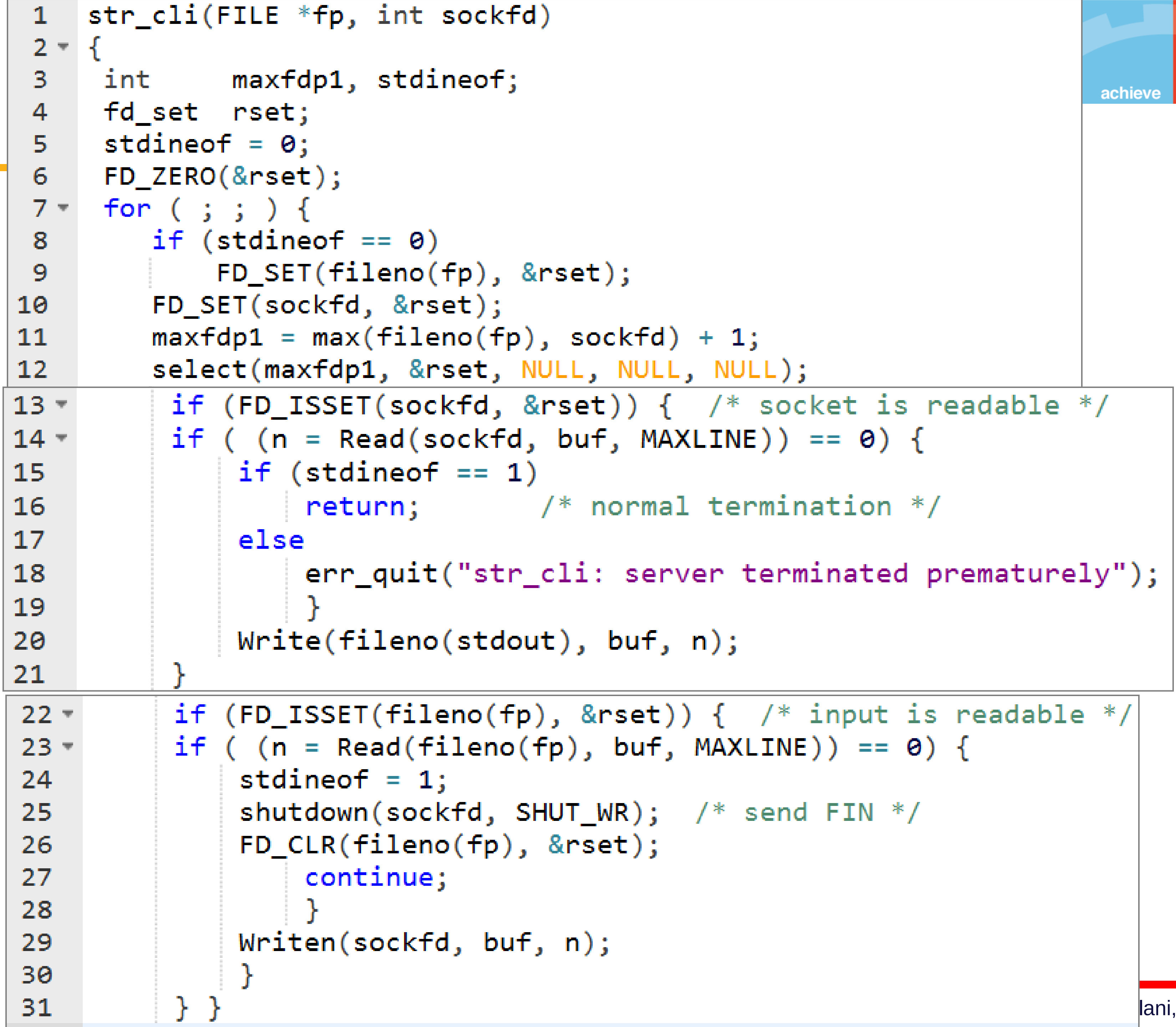


EOF on *stdin*
Set stdineof=1
shutdown(fd, SHUT_WR)

if stdineof=1
Receiving FIN
from server is
Normal
termination

Data on *stdin*

if stdineof=0
Receiving FIN
from server is
abnormal
termination



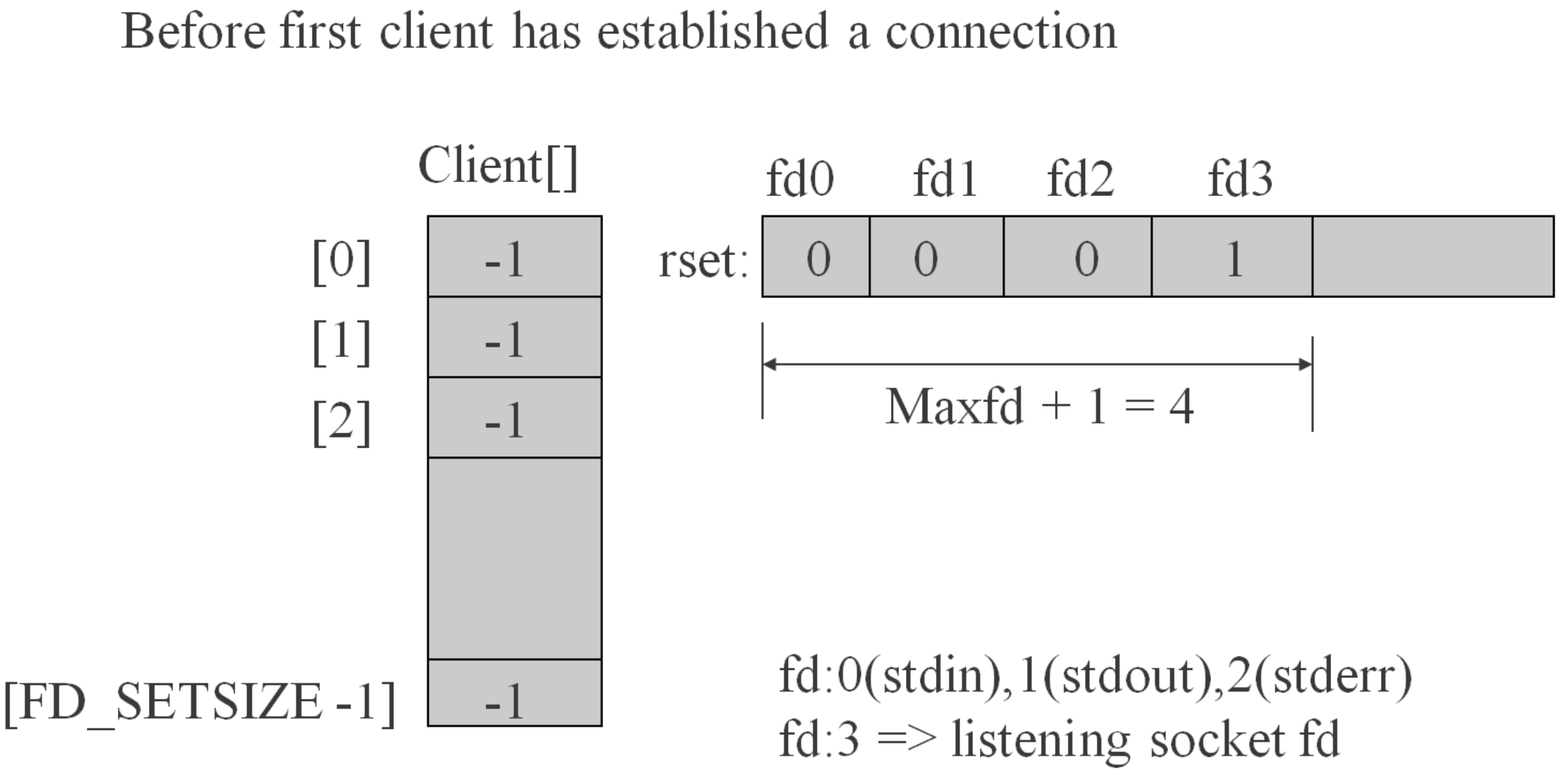
```
str_cli(FILE *fp, int sockfd)
{
 int      maxfdp1, stdineof;
 fd_set   rset;
 stdineof = 0;
 FD_ZERO(&rset);
 for ( ; ; ) {
    if (stdineof == 0)
        FD_SET(fileno(fp), &rset);
    FD_SET(sockfd, &rset);
    maxfdp1 = max(fileno(fp), sockfd) + 1;
    select(maxfdp1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
    if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
        if (stdineof == 1)
            return;        /* normal termination */
        else
            err_quit("str_cli: server terminated prematurely");
            }
        Write(fileno(stdout), buf, n);
    }
    if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
    if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
        stdineof = 1;
        shutdown(sockfd, SHUT_WR);  /* send FIN */
        FD_CLR(fileno(fp), &rset);
            continue;
            }
        Writen(sockfd, buf, n);
        }
    } }
```

TCP Server Using select()

- Single process server that uses select to handle any number of clients, instead of forking one child per client.
- Protocol: echo
- Two data structures:
  - Client array
    - Keeps list of client sockets connected currently
  - fd_set *allset*
    - Keeps list of fds for checking against readability.

TCP Server Using select()

- There are three fds: 0,1,2
- One more fd after creating listening socket.



Before first client has established a connection

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

	fd0	fd1	fd2	fd3	
rset:	0	0	0	1	

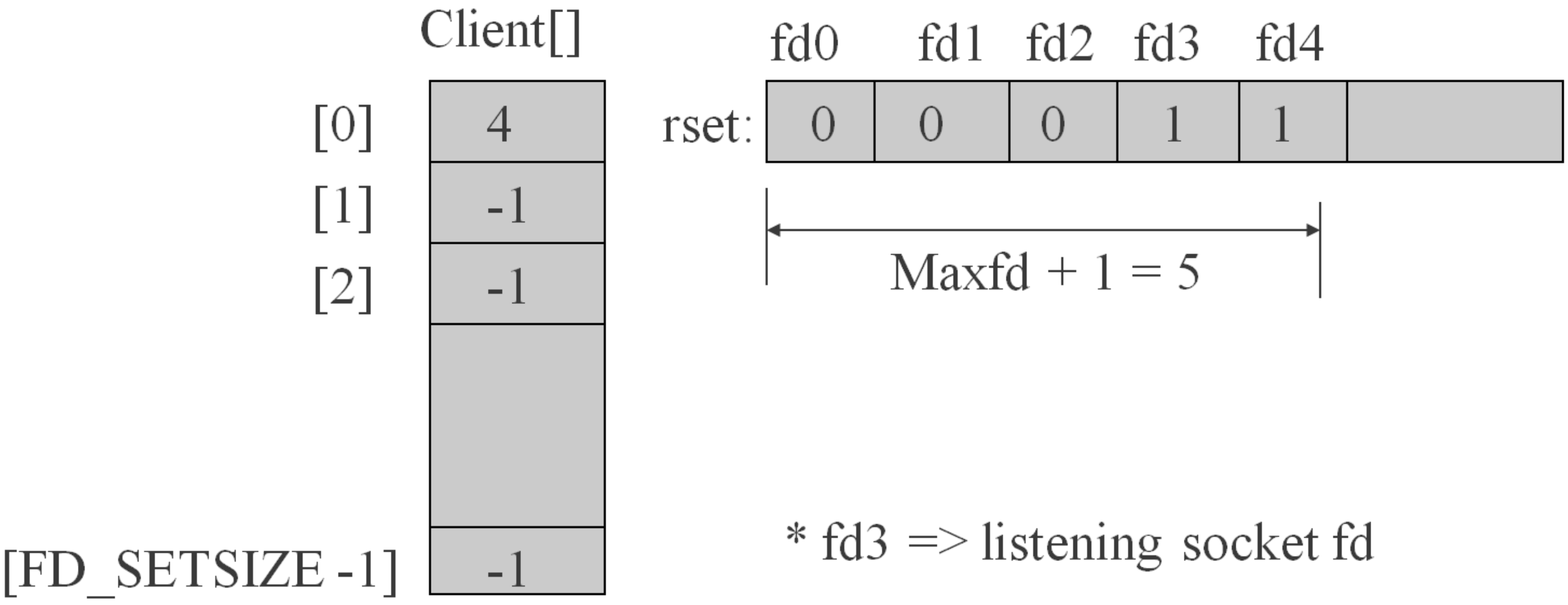
Maxfd + 1 = 4

fd:0(stdin),1(stdout),2(stderr)
fd:3 => listening socket fd

TCP Server Using select()

- When a new client is accepted through accept()
  - A connected socket is added

After first client connection is established



Client[]

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

	fd0	fd1	fd2	fd3	fd4	
rset:	0	0	0	1	1	

Maxfd + 1 = 5

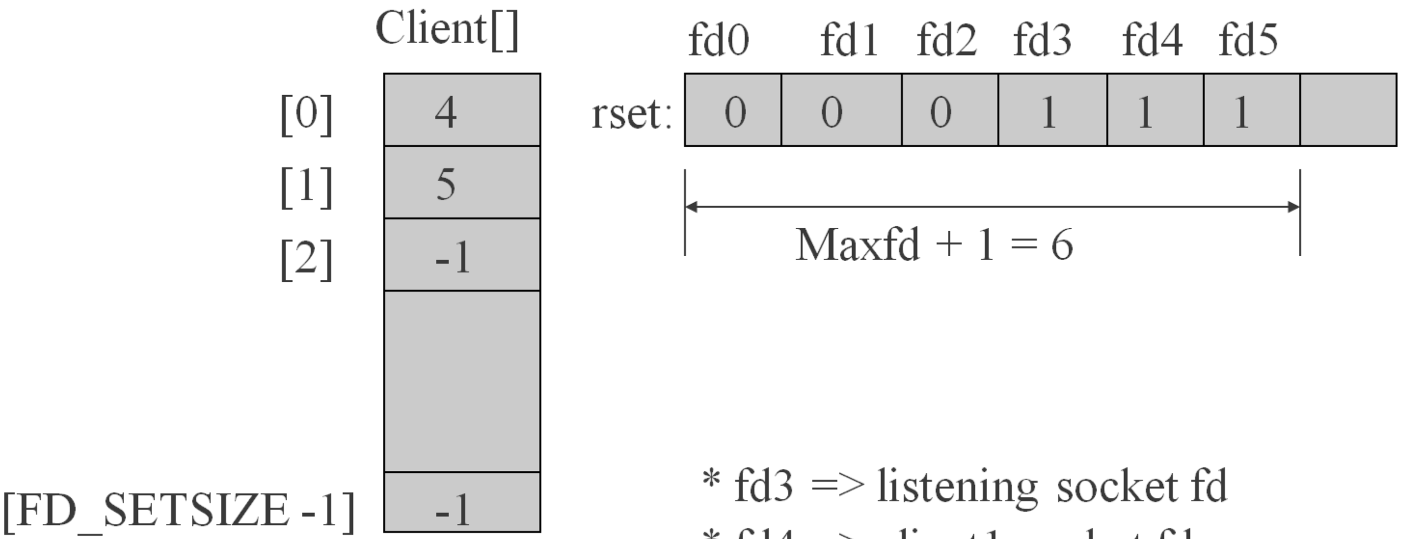
* fd3 => listening socket fd

*fd4 => client socket fd

TCP Server Using select()

- When second client is accepted

After second client connection is established



	Client[]
[0]	4
[1]	5
[2]	-1
[FD_SETSIZE -1]	-1

	fd0	fd1	fd2	fd3	fd4	fd5	
rset:	0	0	0	1	1	1	

Maxfd + 1 = 6

* fd3 => listening socket fd

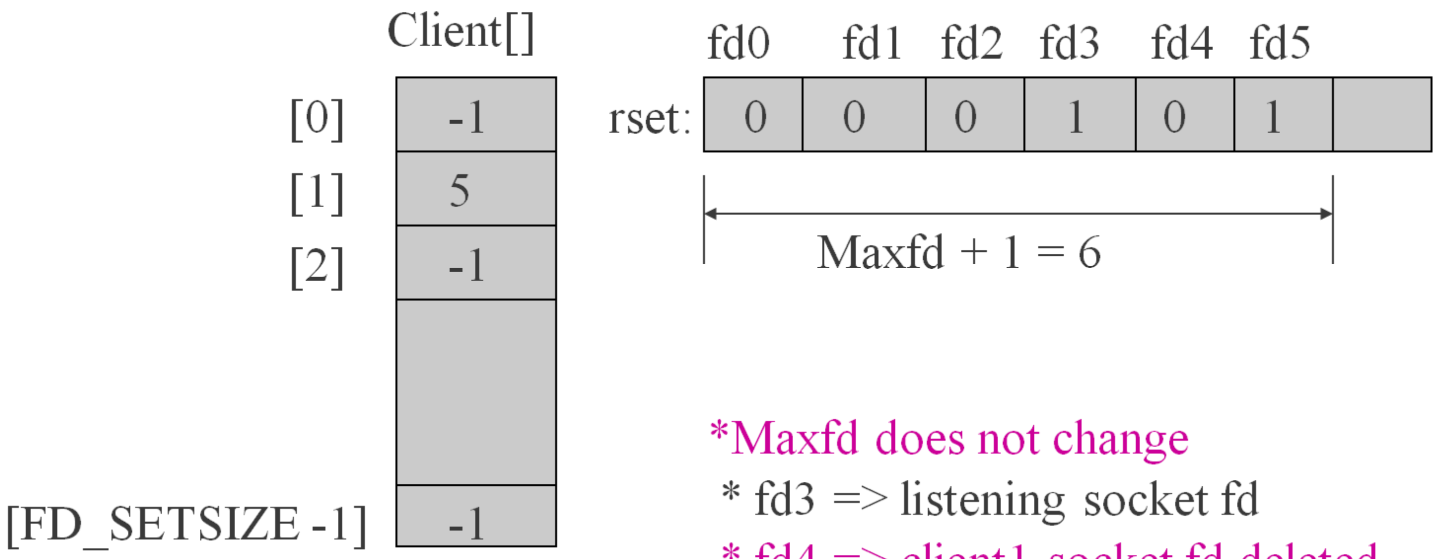
* fd4 => client1 socket fd

* fd5 => client2 socket fd

TCP Server Using select()

- When the first client terminates connection
  - This is known when read() returns zero.

After first client terminates its connection



Client[]	
[0]	-1
[1]	5
[2]	-1
[FD_SETSIZE -1]	-1

	fd0	fd1	fd2	fd3	fd4	fd5	
rset:	0	0	0	1	0	1	

Maxfd + 1 = 6

*Maxfd does not change

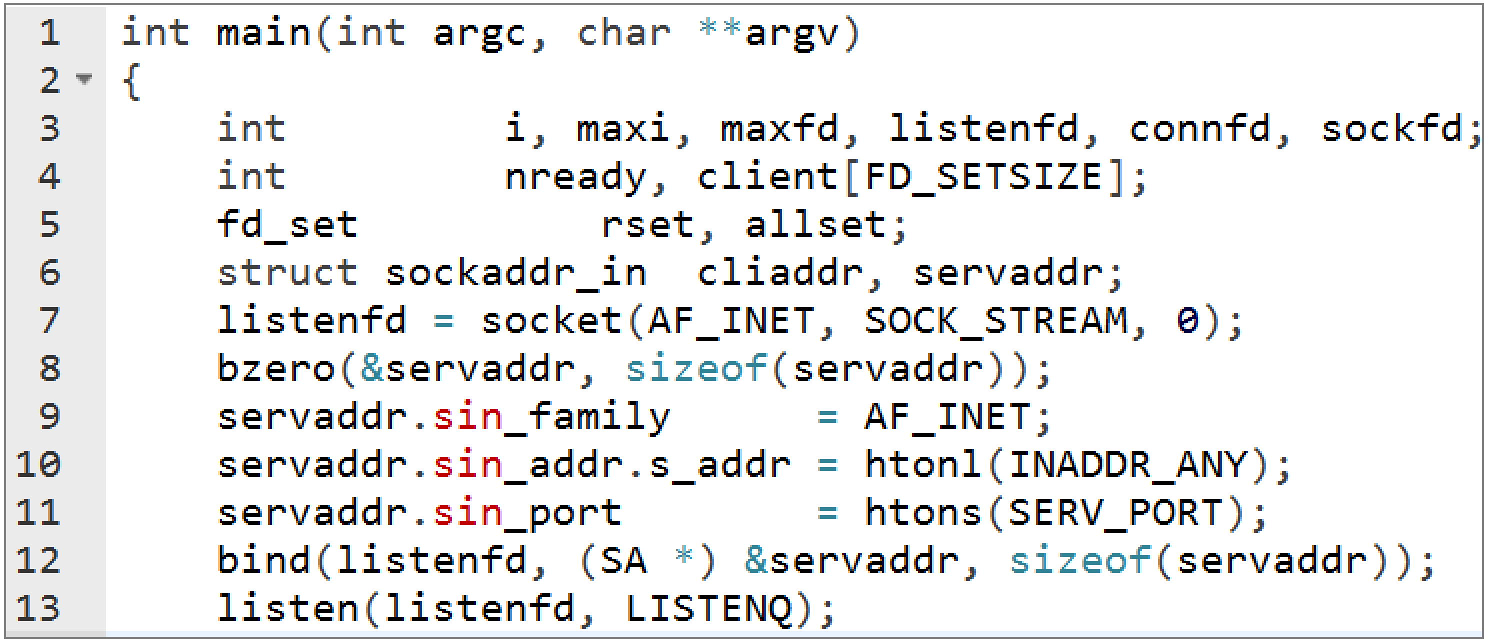
* fd3 => listening socket fd

* fd4 => client1 socket fd deleted

* fd5 => client2 socket fd

TCP Server Using select()

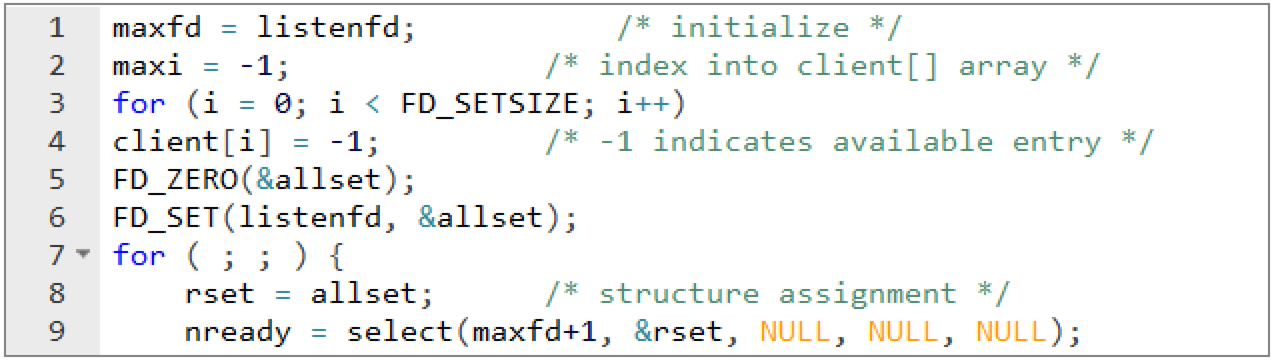
- Create a passive socket.



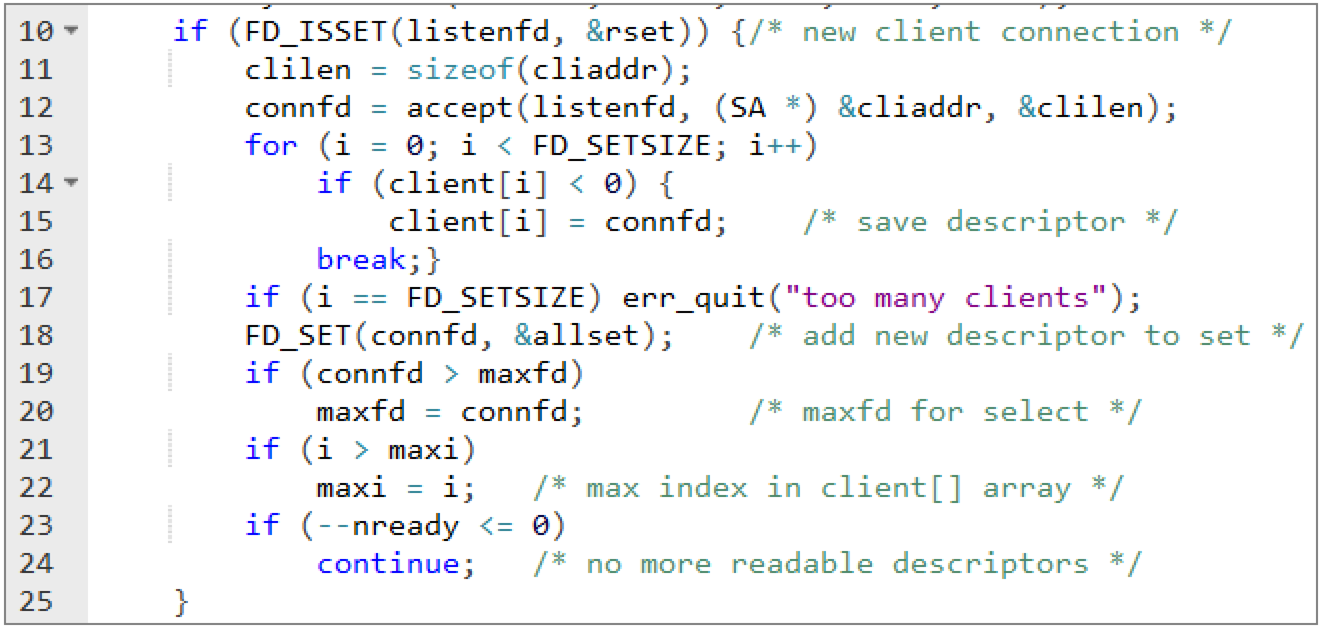
```
int main(int argc, char **argv)
{
    int         i, maxi, maxfd, listenfd, connfd, sockfd;
    int         nready, client[FD_SETSIZE];
    fd_set          rset, allset;
    struct sockaddr_in  cliaddr, servaddr;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
    bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
    listen(listenfd, LISTENQ);
```

TCP Server Using select()

- Handling when listening socket is readable



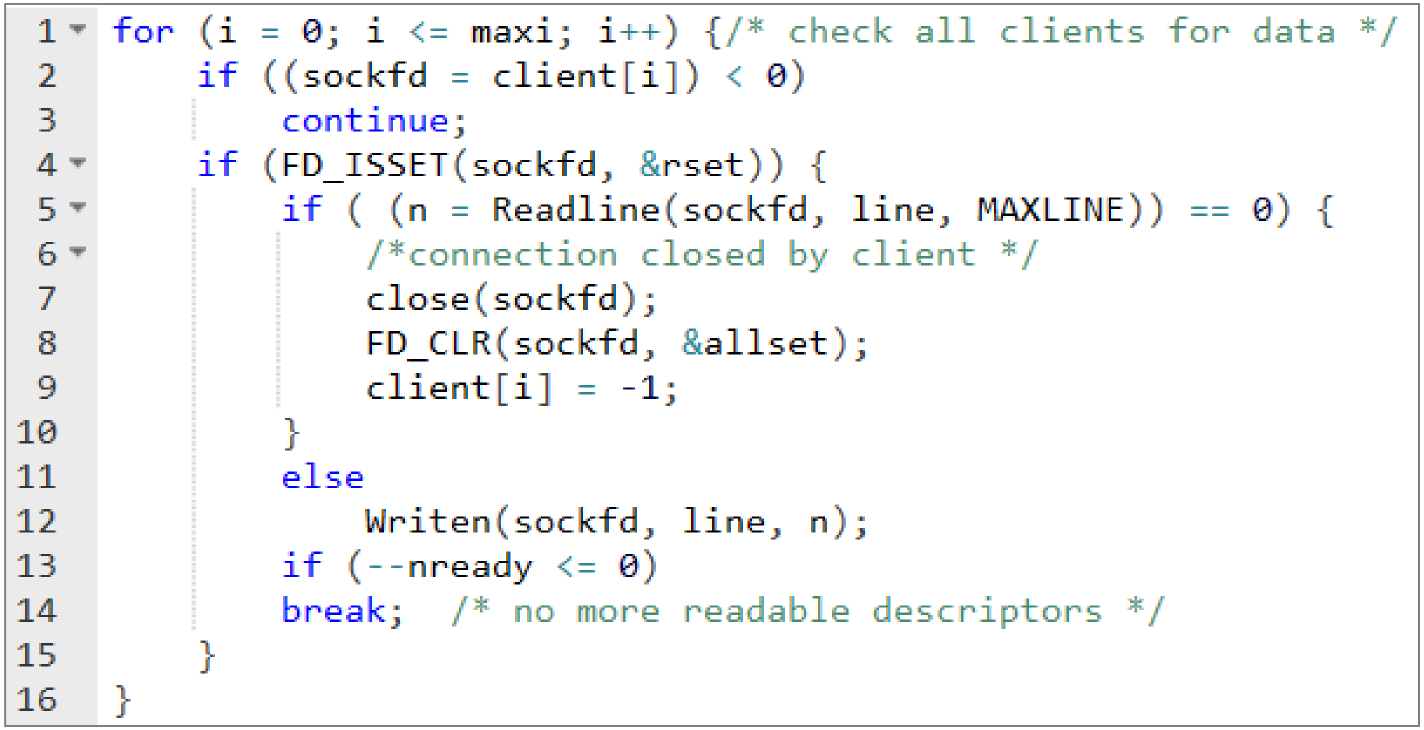
```
maxfd = listenfd;           /* initialize */
maxi = -1;              /* index into client[] array */
for (i = 0; i < FD_SETSIZE; i++)
client[i] = -1;         /* -1 indicates available entry */
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
for ( ; ; ) {
    rset = allset;      /* structure assignment */
    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);
        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;         /* maxfd for select */
        if (i > maxi)
            maxi = i;   /* max index in client[] array */
        if (--nready <= 0)
            continue;   /* no more readable descriptors */
    }
```

TCP Server Using select()

- When a connected socket is readable



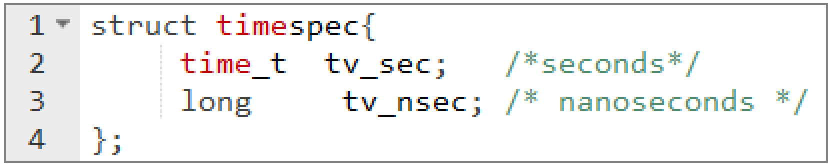
```
for (i = 0; i <= maxi; i++) {/* check all clients for data */
    if ((sockfd = client[i]) < 0)
        continue;
    if (FD_ISSET(sockfd, &rset)) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
            /*connection closed by client */
            close(sockfd);
            FD_CLR(sockfd,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

- This code looks complicated when compared to fork-per-client model. But this design avoids overhead of fork().

Denial-of-Service Attacks

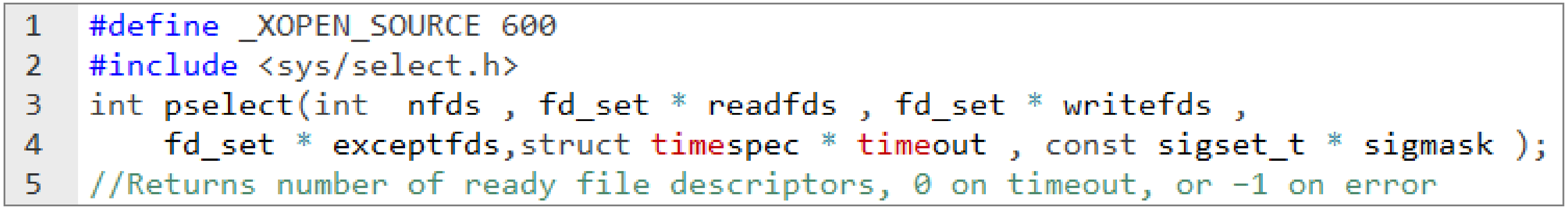
- If malicious client connect to the server, send 1 byte of data (other than a newline), and then goes to sleep.
  - in readline(), server is blocked.
- Solution
  - use nonblocking I/O
  - have each client serviced by a separate thread of control (spawn a process or a thread to service each client)
  - place a timeout on the I/O operation

pselect()

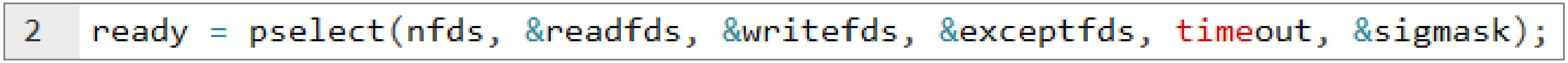


```c
struct timespec{
    time_t  tv_sec;    /*seconds*/
    long     tv_nsec; /* nanoseconds */
};
```

- pselect contains two changes from the normal select function:
  - ○ pselect uses the timespec structure instead of the timeval structure.
  - ○ Accepts signal mask.

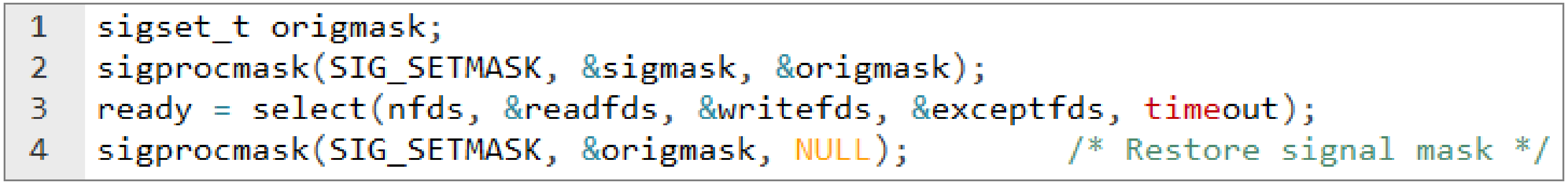


```c
#define _XOPEN_SOURCE 600
#include <sys/select.h>
int pselect(int  nfds , fd_set * readfds , fd_set * writefds ,
    fd_set * exceptfds,struct timespec * timeout , const sigset_t * sigmask );
//Returns number of ready file descriptors, 0 on timeout, or -1 on error
```



```c
ready = pselect(nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask);
```

- ○ This call is equivalent to



```c
sigset_t origmask;
sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);        /* Restore signal mask */
```

Problems with select() and poll()

- The select() and poll() system calls are the portable, long-standing, and widely used methods of monitoring multiple file descriptors for readiness.
- Suffer from some probems
  - Kernel must check all the fds to check if they are ready.
  - Each time select() passes data structures which kernel modifies and returns.
  - Once select() returns, the program must inspect the data structure to see which fds are ready.
- Select() scales poorly with the increase of *fds*.
- Signal driven I/O or *epoll* (event poll) provide a scalable solution.

innovate achieve lead

BITS Pilani
Pilani Campus

Non-blocking I/O on Sockets

Socket Operations

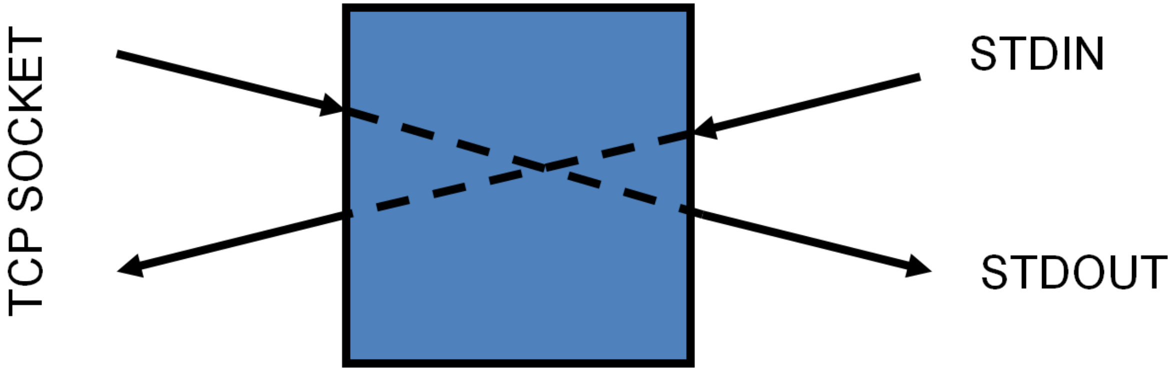
- Input operations: read, recv, readv, recvfrom, recvmsg
  - Blocking operations
  - TCP: until a byte arrives.
  - UDP: until a datagram arrives.
  - With non-blocking socket, if no data, return with EWOULDBLOCK error.
- Output Operations: write, send, writev, sendto, sendmsg
  - Blocks if there is no room in socket send buffer.
  - TCP: until all the data is written.
  - UDP: no send buffer present.
  - With non-blocking socket, TCP write will write whatever it can and returns no. of bytes written. If no room at all, it returns with error EWOULDBLOCK.

Socket Operations

- Accepting incoming connections: accept
  - Blocks if no incoming connection.
  - With non-blocking socket, it would return with an error.
- Initiating Connections: connect
  - Blocks until client TCP receives ACK.
  - With non-blocking socket, it returns *errno* EINPROGRESS, and continues to establish connection.

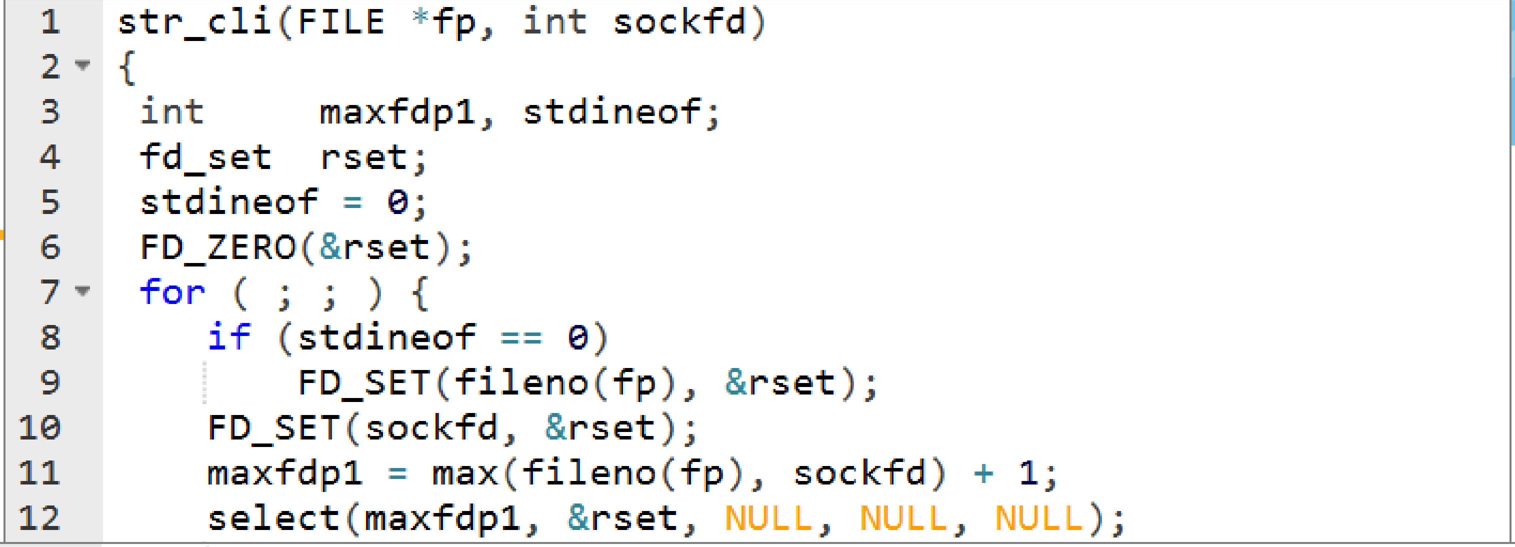
Client Handling a socket, stdin, stdout

- A client usually deals with
  - Stdin
  - Stdout
  - Socket

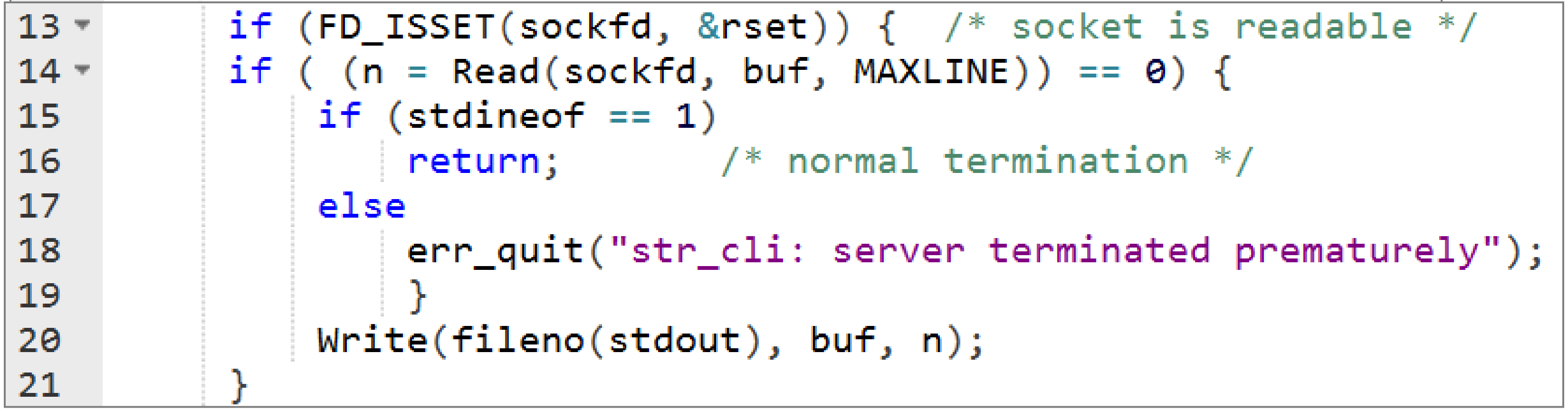


Client Handling a socket, stdin, stdout

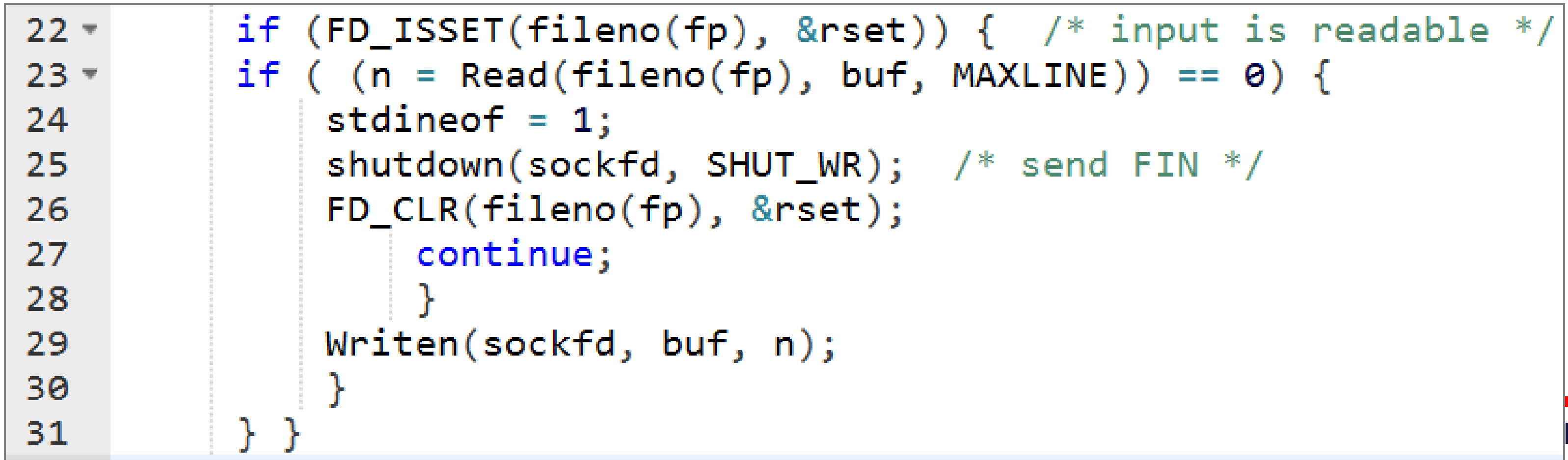
- We looked at
  - Stop and wait client
  - Batch mode client - select with blocking I/O
    - Once select() returns, and if socket is readable, read() is called on socket.
    - readline() call gets blocked on socket till it gets required data.
    - During this time, other clients have to wait.
- Now we look at select with non-blocking I/O
  - In this, read() will be a non-blocking operation. It will read whatever data available on socket. It returns.
  - When this fd is readable next time, further data is read.
    - This requires that we track the number of bytes read and the pointers in the buffer.



```
str_cli(FILE *fp, int sockfd)
{
 int      maxfdp1, stdineof;
 fd_set   rset;
 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);
```



```
     if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
     if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
         if (stdineof == 1)
             return;         /* normal termination */
         else
             err_quit("str_cli: server terminated prematurely");
             }
         Write(fileno(stdout), buf, n);
     }
```



```
     if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
     if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
         stdineof = 1;
         shutdown(sockfd, SHUT_WR);  /* send FIN */
         FD_CLR(fileno(fp), &rset);
             continue;
             }
         Writen(sockfd, buf, n);
         }
     } }
```

Select with Non-Blocking IO

- Non-blocking IO complicates buffer management.
  - We have to keep track of how much is read and how much is written.
- Two buffers:
  - *to*: reading from standard input and write to socket.
  - *from*: read from socket and write to stdout.

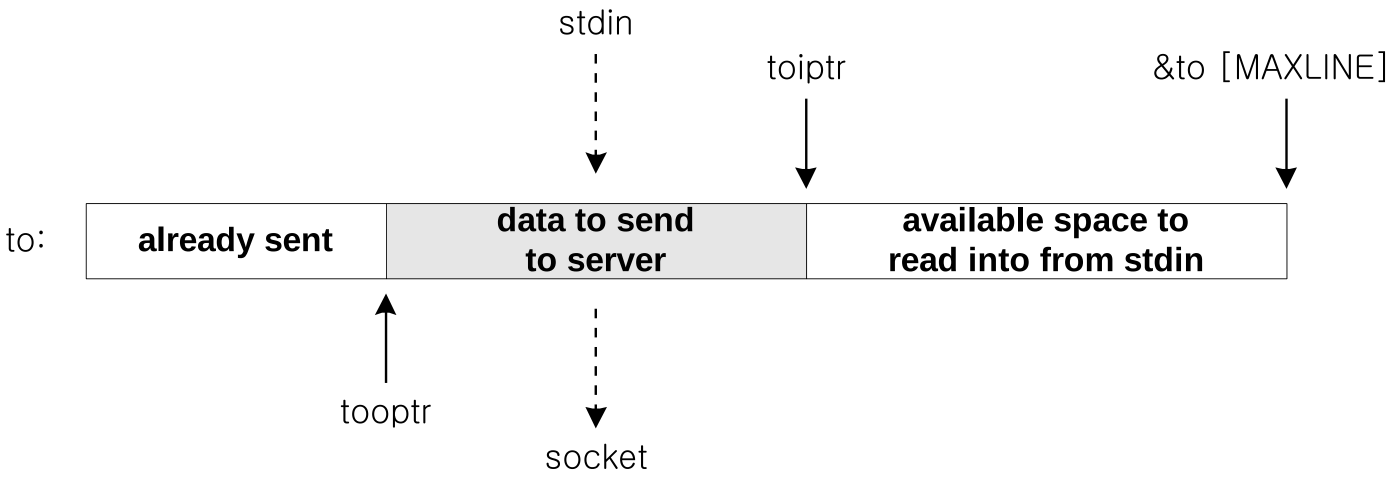


Figure 15.1 Buffer containing data from standard input going to the socket.

Select with Non-Blocking IO

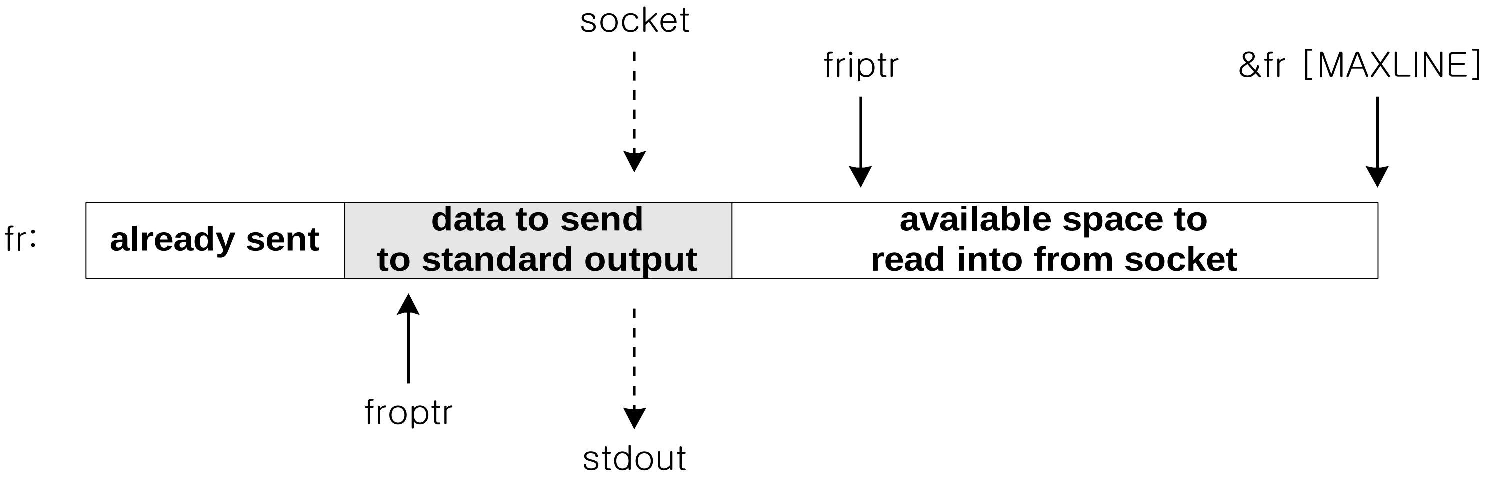
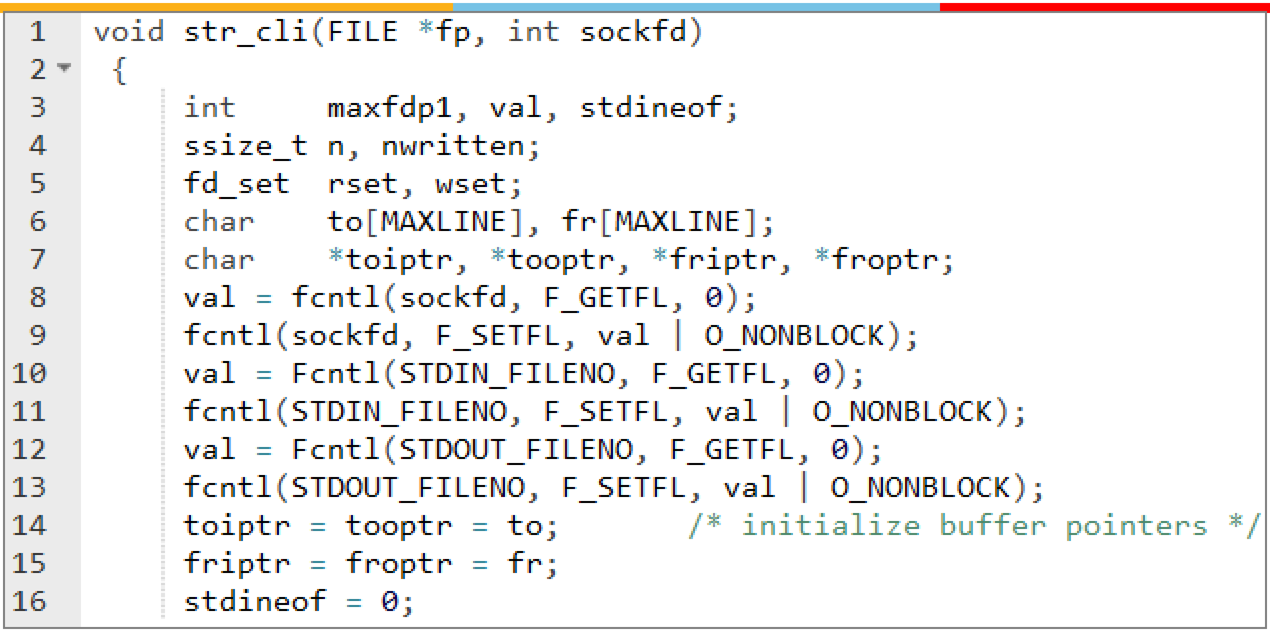


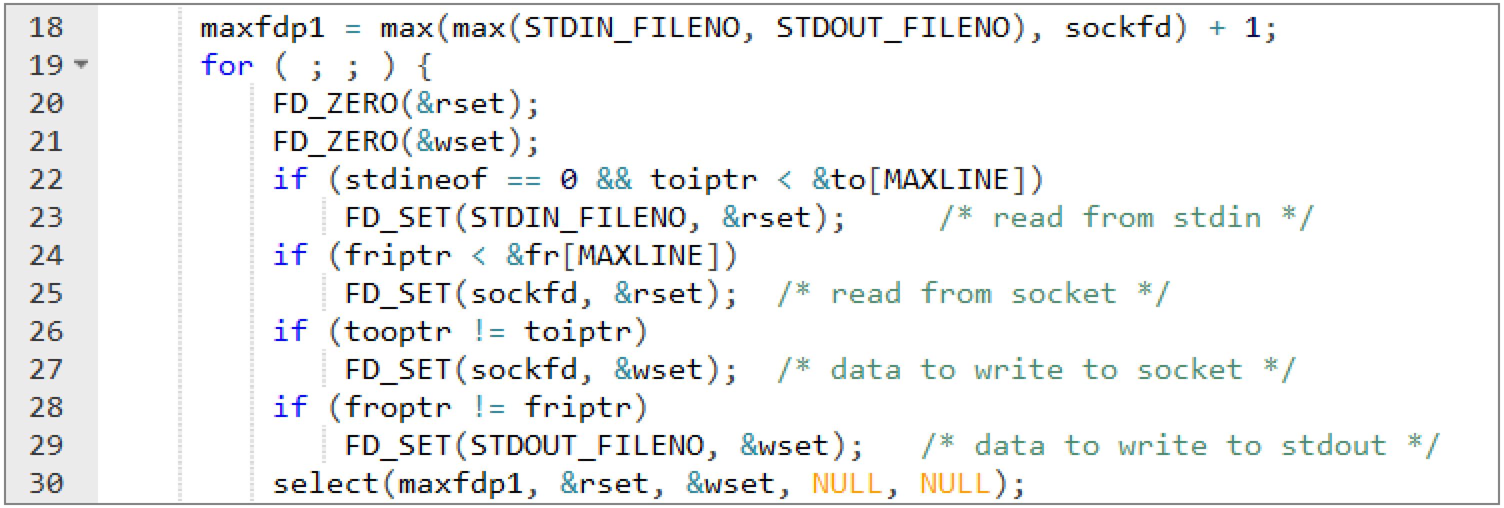
Figure 15.2 Buffer containing data from the socket going to standard output.

- *froptr*: points to the next byte to be sent to stdout.
- *froiptr*: points to the next byte into which next byte can be read..

Select with Non-Blocking IO

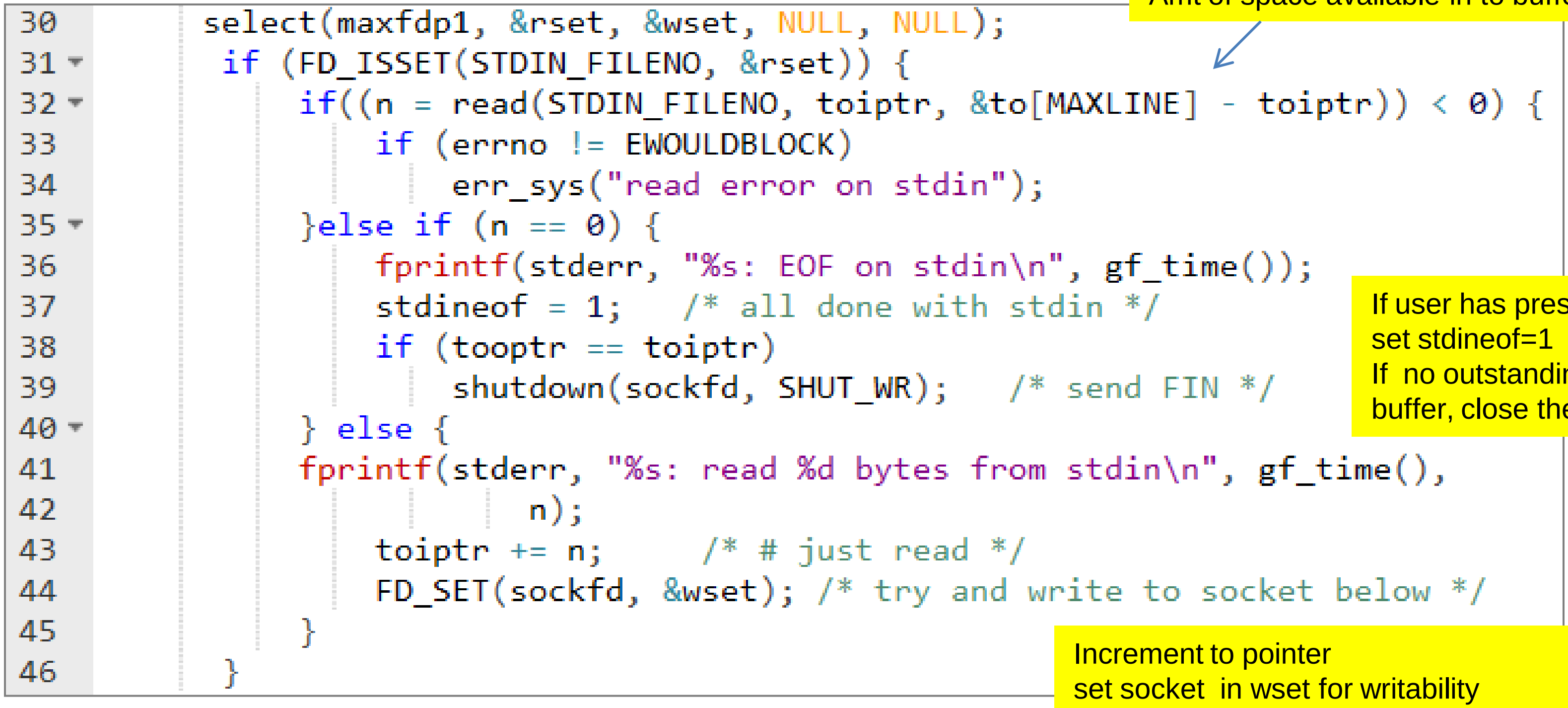


```
void str_cli(FILE *fp, int sockfd)
 {
    int     maxfdp1, val, stdineof;
    ssize_t n, nwritten;
    fd_set  rset, wset;
    char    to[MAXLINE], fr[MAXLINE];
    char    *toiptr, *tooptr, *friptr, *froptr;
    val = fcntl(sockfd, F_GETFL, 0);
    fcntl(sockfd, F_SETFL, val | O_NONBLOCK);
    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);
    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);
    toiptr = tooptr = to;       /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;
```



```
    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
    for ( ; ; ) {
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        if (stdineof == 0 && toiptr < &to[MAXLINE])
            FD_SET(STDIN_FILENO, &rset);     /* read from stdin */
        if (friptr < &fr[MAXLINE])
            FD_SET(sockfd, &rset);  /* read from socket */
        if (tooptr != toiptr)
            FD_SET(sockfd, &wset);  /* data to write to socket */
        if (froptr != friptr)
            FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */
        select(maxfdp1, &rset, &wset, NULL, NULL);
```

reads from standard input



```
    select(maxfdp1, &rset, &wset, NULL, NULL);
     if (FD_ISSET(STDIN_FILENO, &rset)) {
         if((n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
             if (errno != EWOULDBLOCK)
                 err_sys("read error on stdin");
         }else if (n == 0) {
             fprintf(stderr, "%s: EOF on stdin\n", gf_time());
             stdineof = 1;   /* all done with stdin */
             if (tooptr == toiptr)
                 shutdown(sockfd, SHUT_WR);   /* send FIN */
         } else {
         fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(),
                     n);
             toiptr += n;      /* # just read */
             FD_SET(sockfd, &wset); /* try and write to socket below */
         }
     }
```

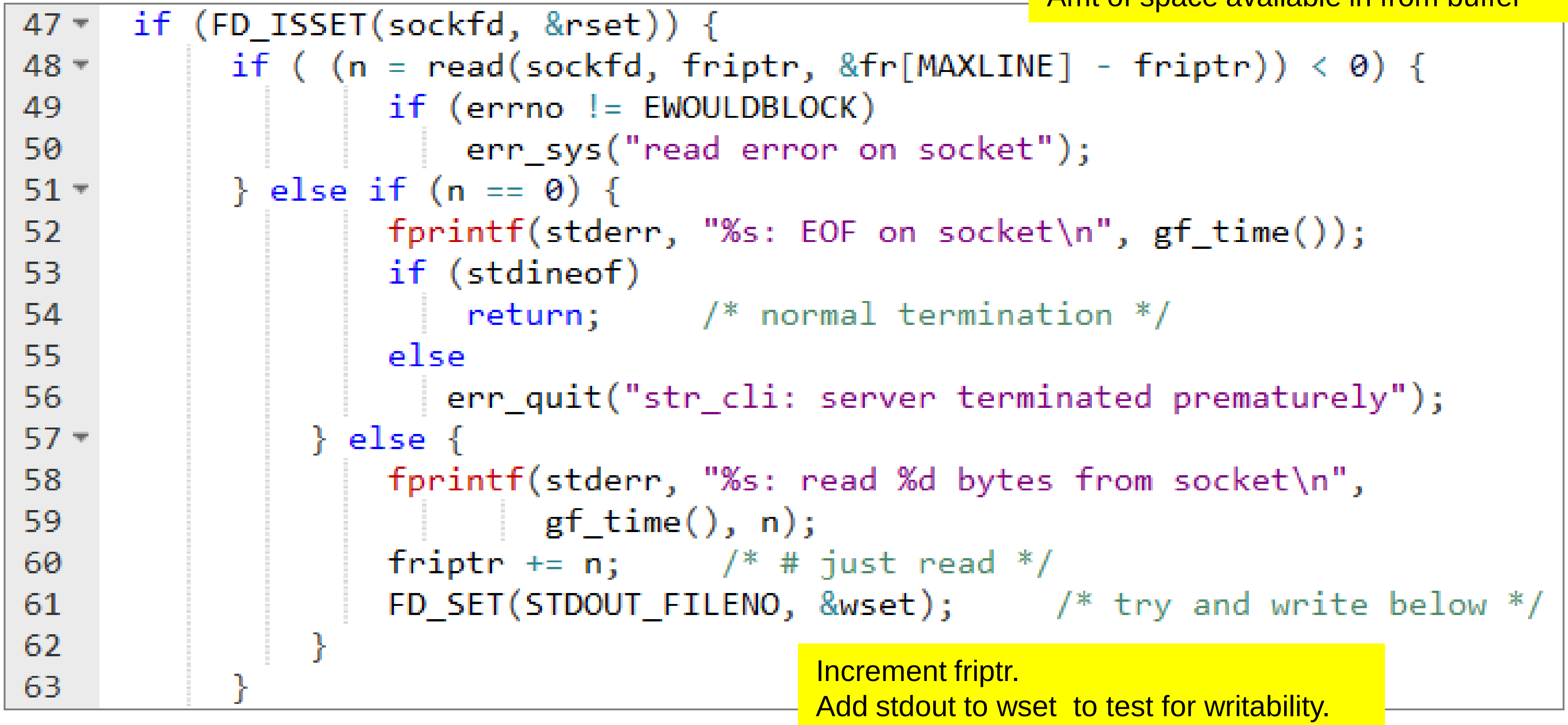
Amt of space available in to buffer

If user has pressed Ctrl-D,
set stdineof=1
If no outstanding data on
buffer, close the write end.

Increment to pointer
set socket in wset for writability

reads from socket

Amt of space available in from buffer

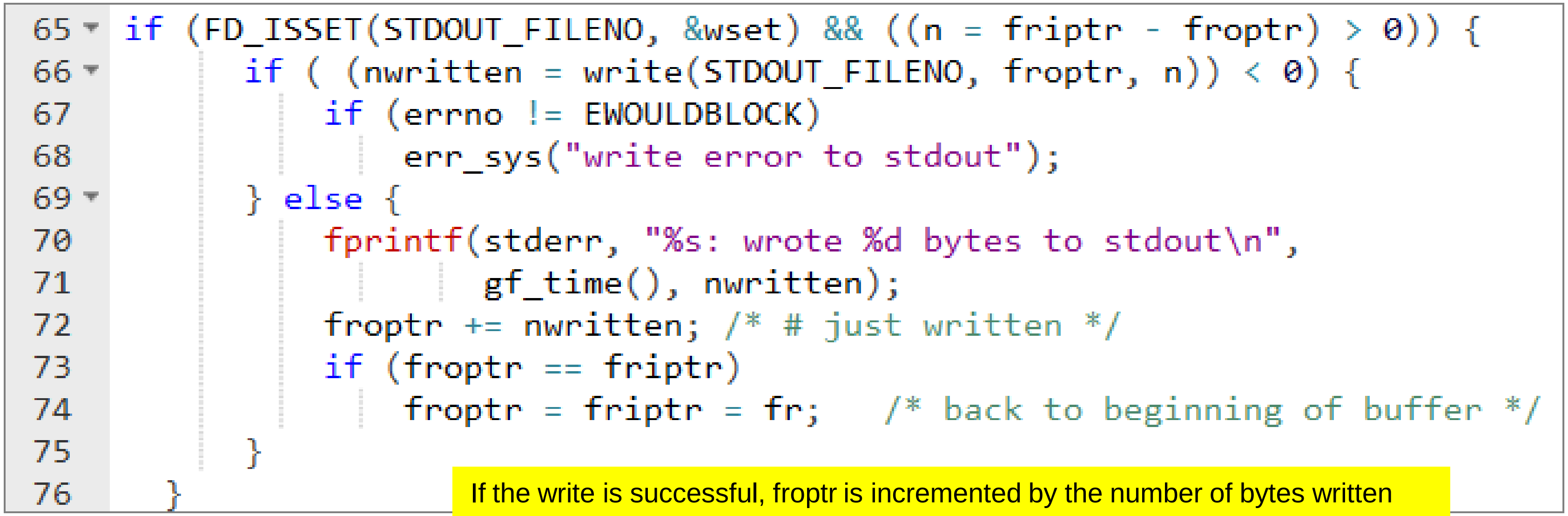


```
if (FD_ISSET(sockfd, &rset)) {
    if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
            if (errno != EWOULDBLOCK)
                err_sys("read error on socket");
    } else if (n == 0) {
            fprintf(stderr, "%s: EOF on socket\n", gf_time());
            if (stdineof)
                return;     /* normal termination */
            else
               err_quit("str_cli: server terminated prematurely");
        } else {
            fprintf(stderr, "%s: read %d bytes from socket\n",
                    gf_time(), n);
            friptr += n;     /* # just read */
            FD_SET(STDOUT_FILENO, &wset);     /* try and write below */
        }
    }
```

Increment friptr.
Add stdout to wset to test for writability.

writes to standard output

No of bytes to write >0

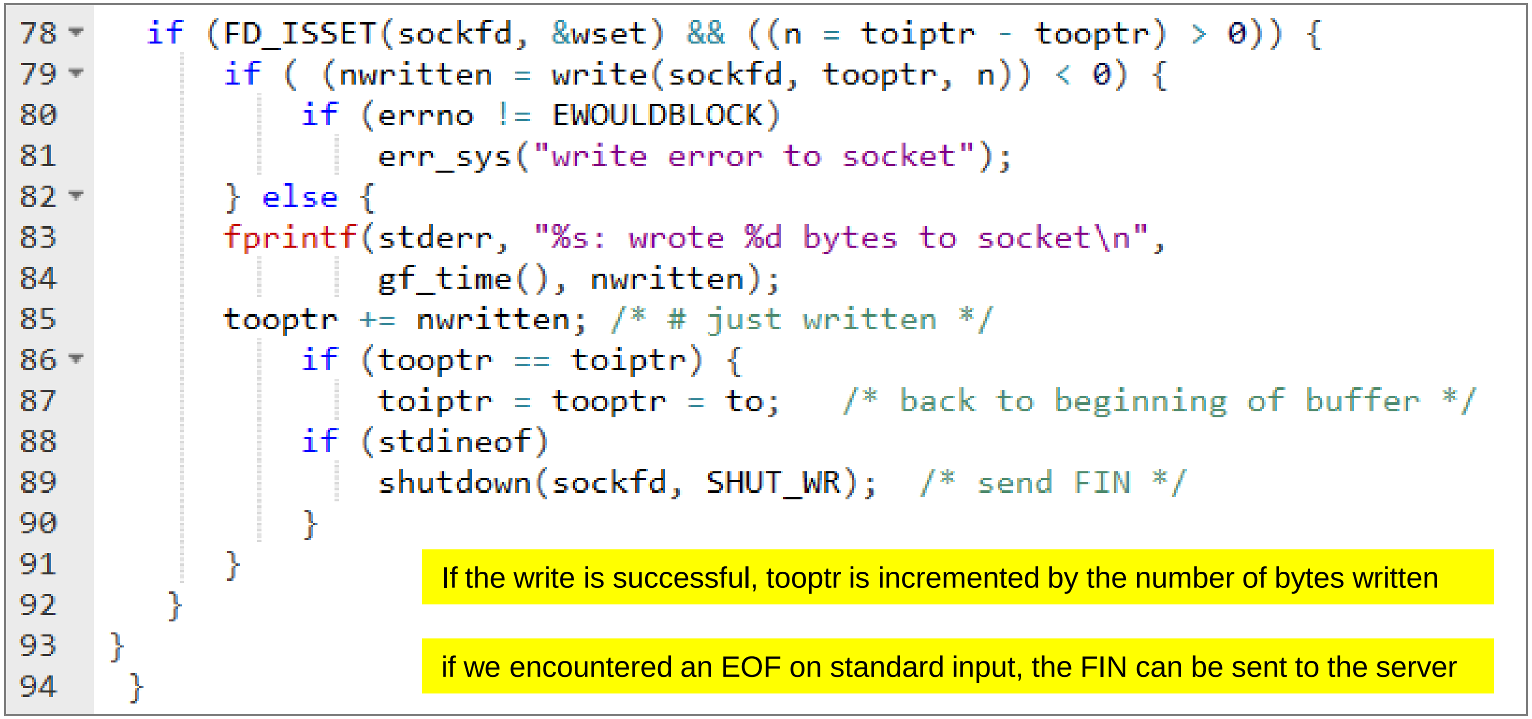


```
if (FD_ISSET(STDOUT_FILENO, &wset) && ((n = friptr - froptr) > 0)) {
      if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
          if (errno != EWOULDBLOCK)
              err_sys("write error to stdout");
      } else {
          fprintf(stderr, "%s: wrote %d bytes to stdout\n",
                  gf_time(), nwritten);
          froptr += nwritten; /* # just written */
          if (froptr == friptr)
              froptr = friptr = fr;    /* back to beginning of buffer */
      }
  }
```

If the write is successful, froptr is incremented by the number of bytes written

Writes to socket

No of bytes to write >0



```
 if (FD_ISSET(sockfd, &wset) && ((n = toiptr - tooptr) > 0)) {
   if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
     if (errno != EWOULDBLOCK)
       err_sys("write error to socket");
   } else {
   fprintf(stderr, "%s: wrote %d bytes to socket\n",
       gf_time(), nwritten);
   tooptr += nwritten; /* # just written */
     if (tooptr == toiptr) {
       toiptr = tooptr = to;   /* back to beginning of buffer */
     if (stdineof)
       shutdown(sockfd, SHUT_WR);  /* send FIN */
     }
   }
  }
}
 }
```

If the write is successful, tooptr is incremented by the number of bytes written

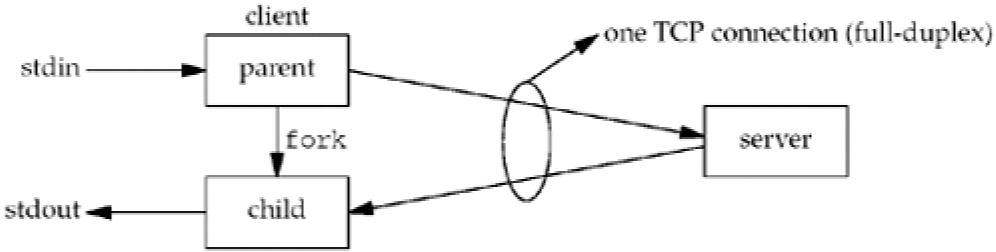
if we encountered an EOF on standard input, the FIN can be sent to the server innovate achieve lead

BITS Pilani
Pilani Campus

Client using Multiple Processes

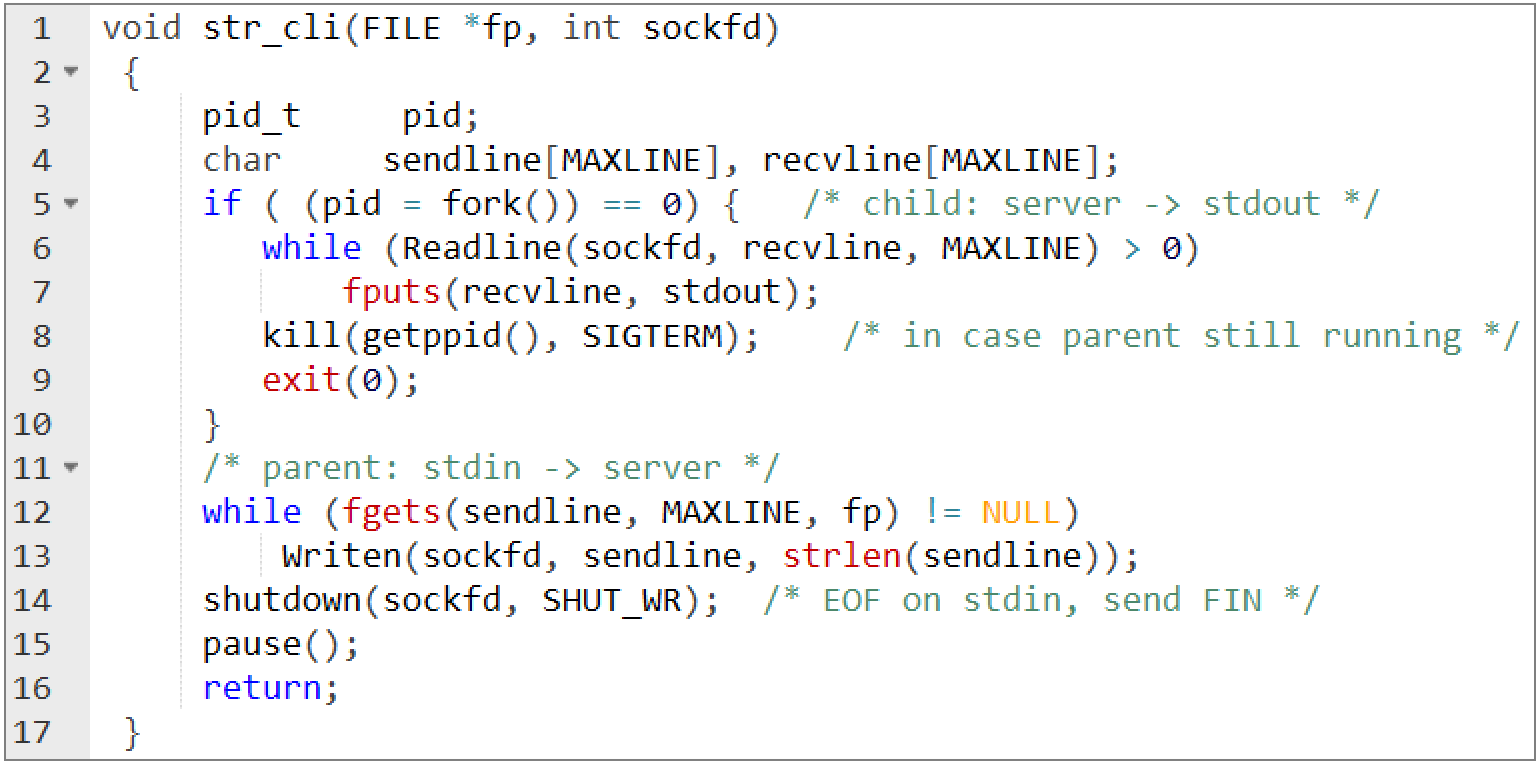
Client using Multiple Processes

- Whenever we find the need to use nonblocking I/O, it will usually be simpler to split the application into either processes (using fork) or threads.
  - Parent reads from stdin and writes to socket
  - Child reads from socket and writes to stdout.



  - Normal termination occurs when the EOF on standard input is encountered. The parent reads this EOF and calls shutdown to send a FIN.
  - If abnormal occurs, the child will read an EOF on the socket. If this happens, the child must tell the parent to stop copying from the standard input to the socket
    - the child sends a signal (e.g. SIGTERM) to the parent.

Client using Multiple Processes



```
void str_cli(FILE *fp, int sockfd)
 {
     pid_t     pid;
     char     sendline[MAXLINE], recvline[MAXLINE];
     if ( (pid = fork()) == 0) {   /* child: server -> stdout */
        while (Readline(sockfd, recvline, MAXLINE) > 0)
            fputs(recvline, stdout);
        kill(getppid(), SIGTERM);    /* in case parent still running */
        exit(0);
     }
     /* parent: stdin -> server */
     while (fgets(sendline, MAXLINE, fp) != NULL)
         Writen(sockfd, sendline, strlen(sendline));
     shutdown(sockfd, SHUT_WR);  /* EOF on stdin, send FIN */
     pause();
     return;
 }
```

Comparing Cleint Designs

- when copying 2,000 lines from a client to a server with an RTT of 175 ms:

Client	Time taken for sending and receiving
stop-and-wait	354.0 sec
select and blocking I/O	12.3 sec
nonblocking I/O	6.9 sec
fork	8.7 sec
threaded version	8.5 sec

  - nonblocking I/O version is almost twice as fast as version using blocking I/O with select.
  - Version using fork is slower than nonblocking I/O version.
  - Nevertheless, given the complexity of the nonblocking I/O code versus the fork code, fork version is simple approach.

Non-blocking Connect

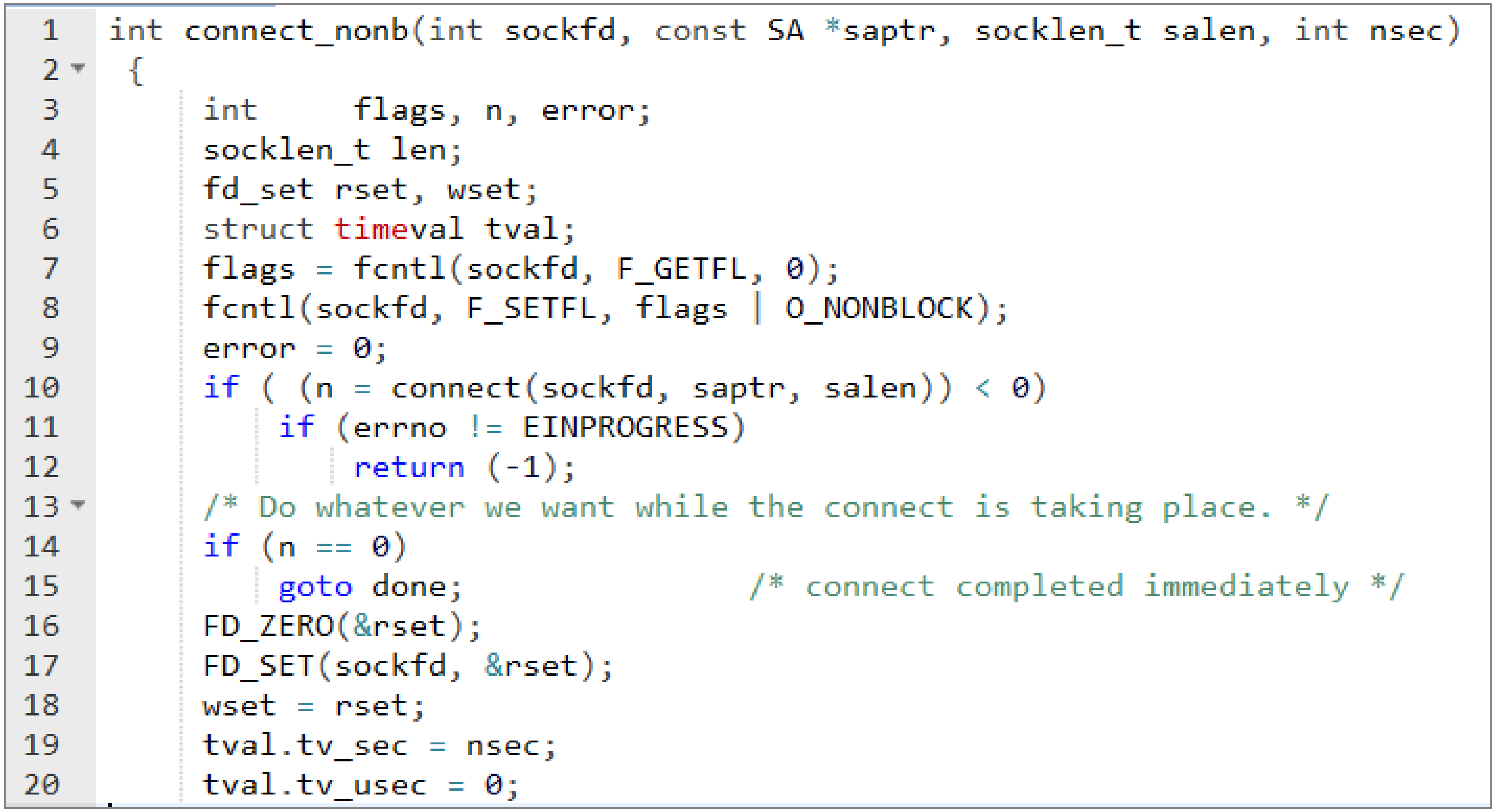
Nonblocking connect()

- TCP socket nonblocking connect
  - return: an error of EINPROGRESS
  - TCP three-way handshake continues
  - check the connection establishment using select
- There are three uses for a nonblocking connect.
  - We can overlap other processing with the three-way handshake.
  - We can establish multiple connections at the same time using this technique.
    - popular with Web browsers
  - Since we wait for the connection establishment to complete using select, we can specify time limit for select, allowing us to shorten the timeout for the connect.

Nonblocking connect()

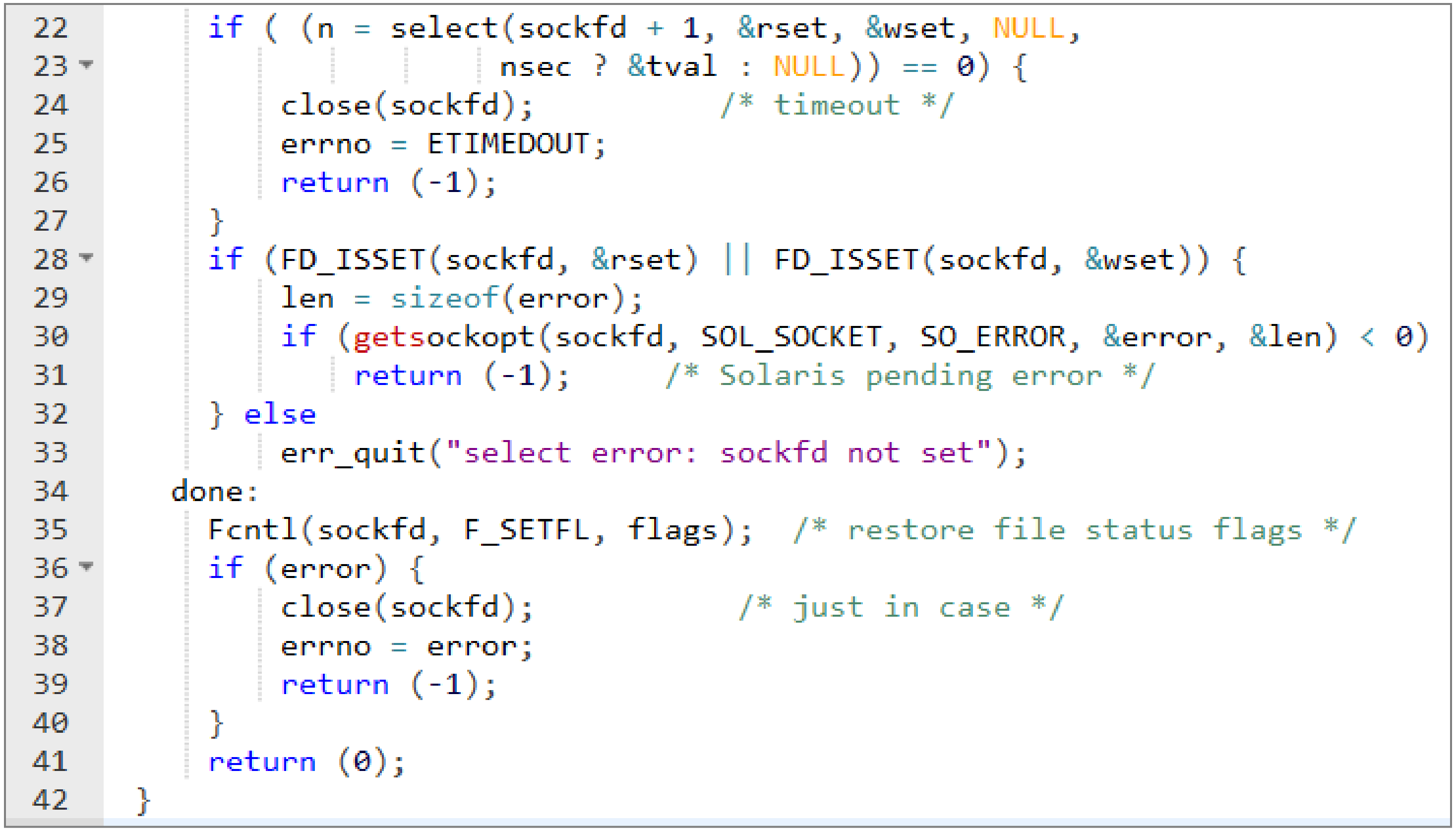
- Set the socket to non-blocking.
- Call connect(). It will return immediately with error EINPROGRESS.
- We use *select*() to check what has happened to connect().
- If the descriptor is readable or writable, we call getsockopt() to fetch the socket's pending error (SO_ERROR). If the connection completed successfully, this value will be 0.

Nonblocking connect()



```
int connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
 {
     int     flags, n, error;
     socklen_t len;
     fd_set rset, wset;
     struct timeval tval;
     flags = fcntl(sockfd, F_GETFL, 0);
     fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
     error = 0;
     if ( (n = connect(sockfd, saptr, salen)) < 0)
         if (errno != EINPROGRESS)
             return (-1);
     /* Do whatever we want while the connect is taking place. */
     if (n == 0)
         goto done;              /* connect completed immediately */
     FD_ZERO(&rset);
     FD_SET(sockfd, &rset);
     wset = rset;
     tval.tv_sec = nsec;
     tval.tv_usec = 0;
```

Nonblocking connect()



```
    if ( (n = select(sockfd + 1, &rset, &wset, NULL,
                    nsec ? &tval : NULL)) == 0) {
        close(sockfd);          /* timeout */
        errno = ETIMEDOUT;
        return (-1);
    }
    if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
        len = sizeof(error);
        if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
            return (-1);     /* Solaris pending error */
    } else
        err_quit("select error: sockfd not set");
  done:
    Fcntl(sockfd, F_SETFL, flags);  /* restore file status flags */
    if (error) {
        close(sockfd);           /* just in case */
        errno = error;
        return (-1);
    }
    return (0);
}
```

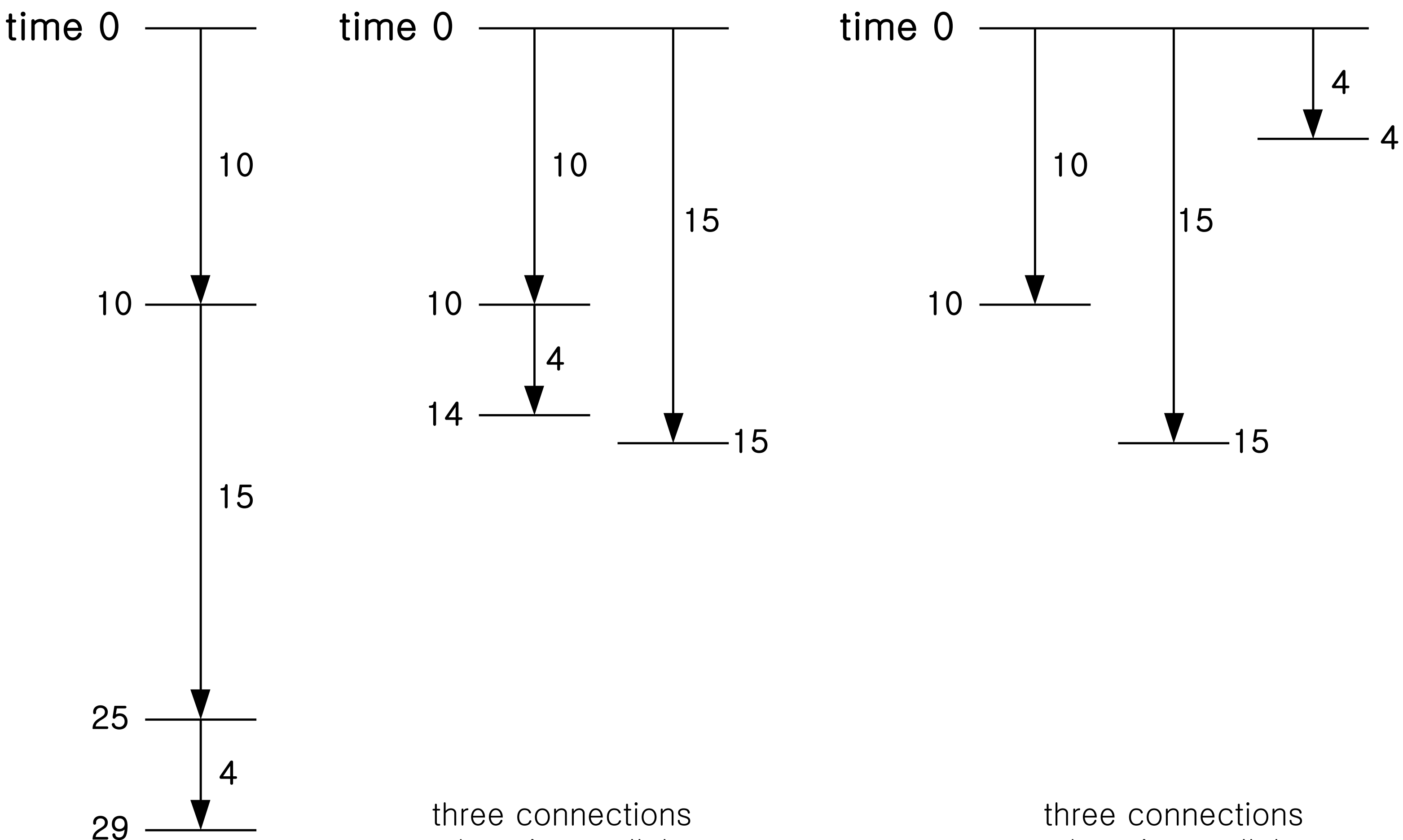
BITS Pilani
Pilani Campus

Web client: Non-blocking Connect

nonblocking `connect:` web client

- A real-world example of nonblocking connects started with Netscape Web Client
- The client establishes an HTTP connection with a Web server and fetches a home page.
- On that page are often numerous references to other Web pages.
- Instead of fetching these other pages serially, one at a time, the client can fetch more than one at the same time, using nonblocking connects.

nonblocking `connect:` web client (cont.)



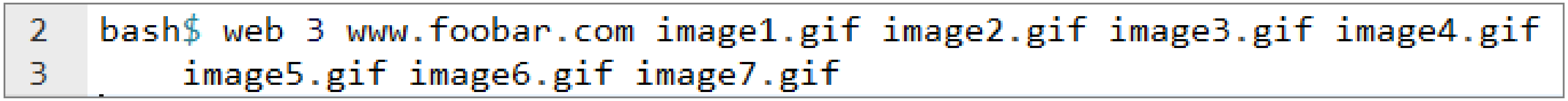
three connections done serially

three connections done in parallel; maximum of two connections at a time

three connections done in parallel; maximum of three connections at a time

Non-blocking Connect

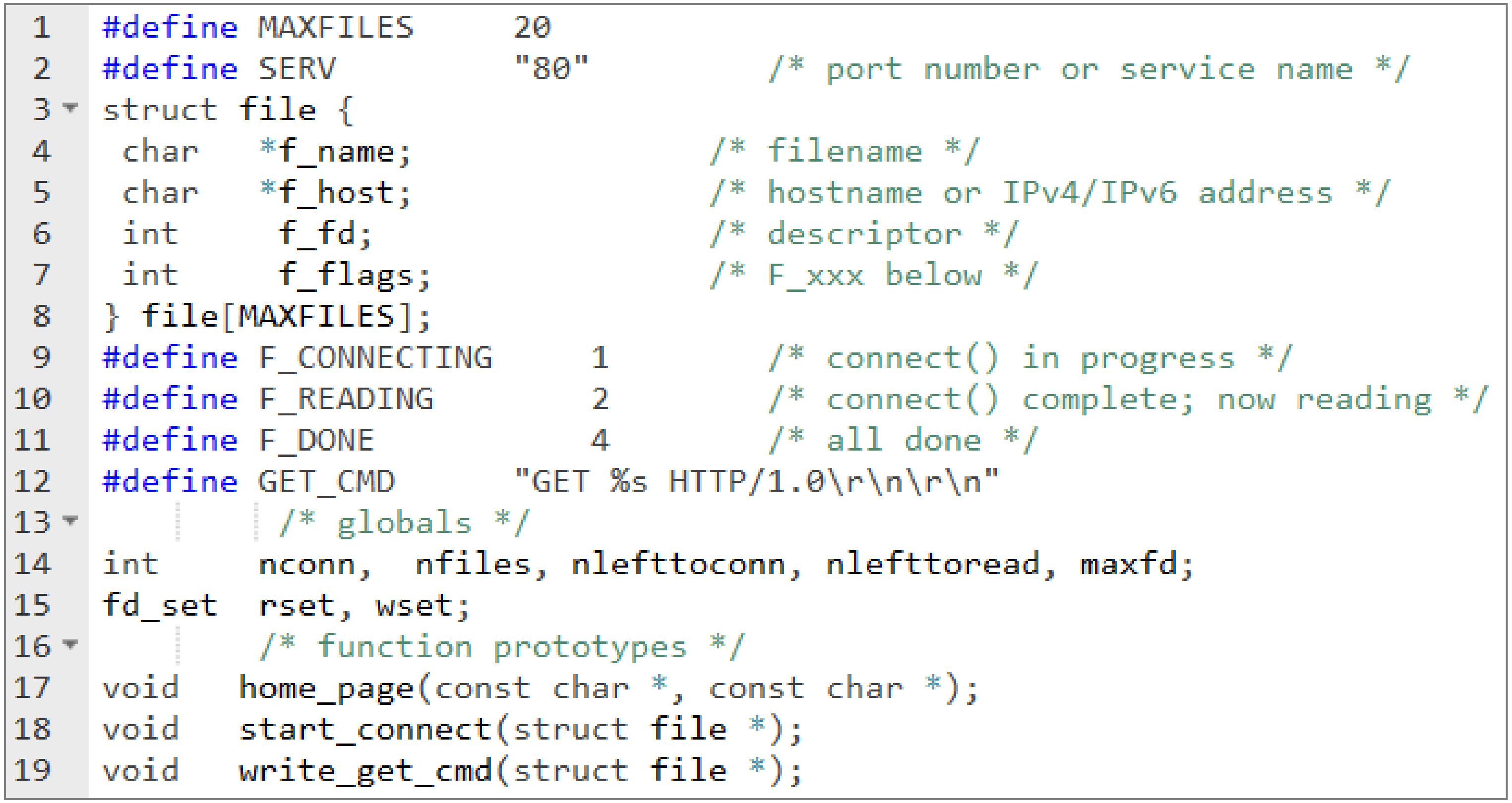
- This program will read up to 20 files from a Web server.
- We specify as command-line arguments
  - the maximum number of parallel connections,
  - the server's hostname, and
  - each of the filenames to fetch from the server.



```
bash$ web 3 www.foobar.com image1.gif image2.gif image3.gif image4.gif
    image5.gif image6.gif image7.gif
```

  - It means
    - three simultaneous connection
    - server’s hostname
    - filename for the home page
    - the files to be read

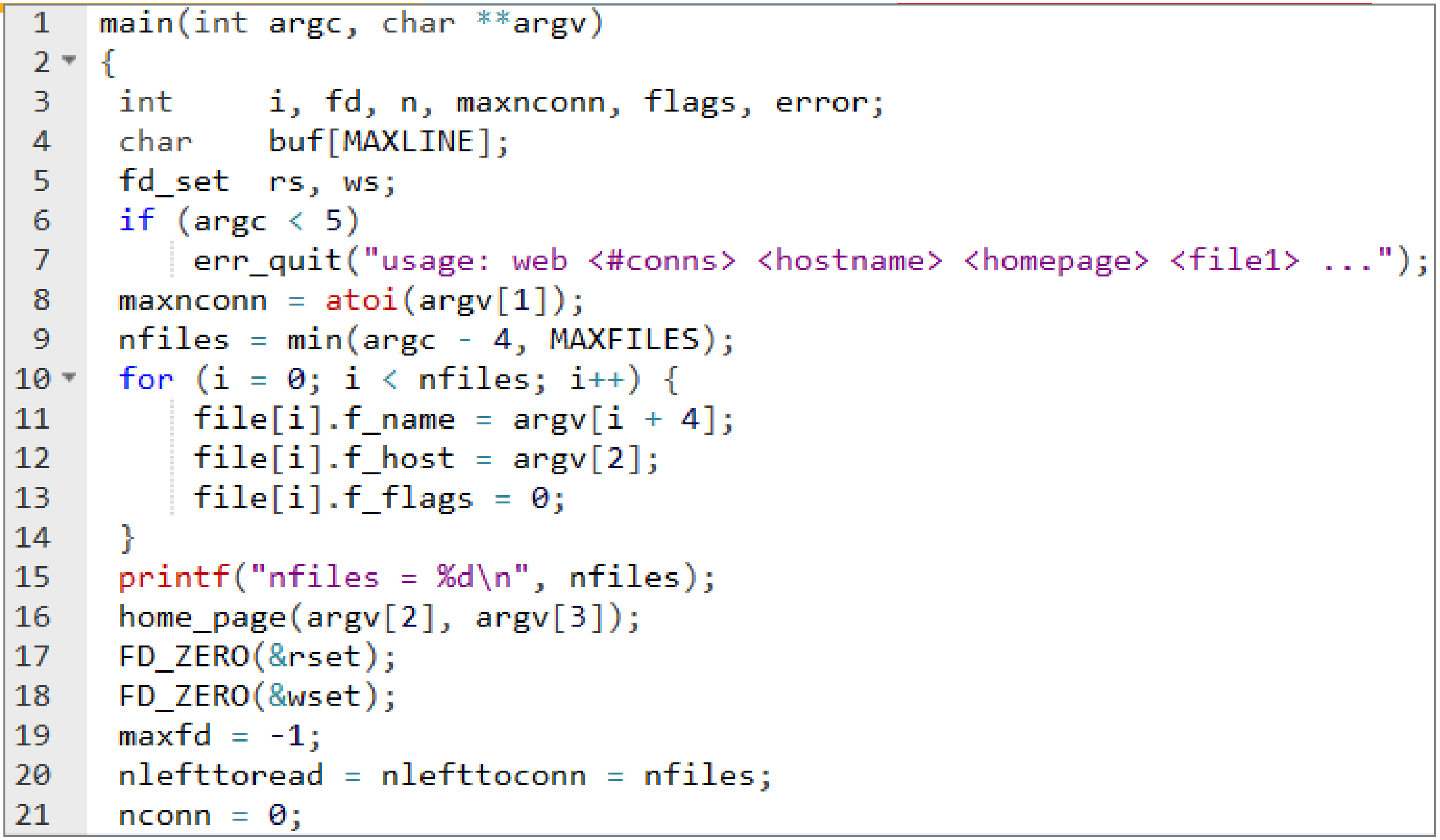
nonblock/web.h



```c
#define MAXFILES     20
#define SERV         "80"         /* port number or service name */
struct file {
 char   *f_name;               /* filename */
 char   *f_host;               /* hostname or IPv4/IPv6 address */
 int     f_fd;                 /* descriptor */
 int     f_flags;              /* F_xxx below */
} file[MAXFILES];
#define F_CONNECTING     1        /* connect() in progress */
#define F_READING        2        /* connect() complete; now reading */
#define F_DONE           4        /* all done */
#define GET_CMD      "GET %s HTTP/1.0\r\n\r\n"
                 /* globals */
int     nconn,  nfiles, nlefttoconn, nlefttoread, maxfd;
fd_set  rset, wset;
                /* function prototypes */
void   home_page(const char *, const char *);
void   start_connect(struct file *);
void   write_get_cmd(struct file *);
```

- Each file has a state and fd.

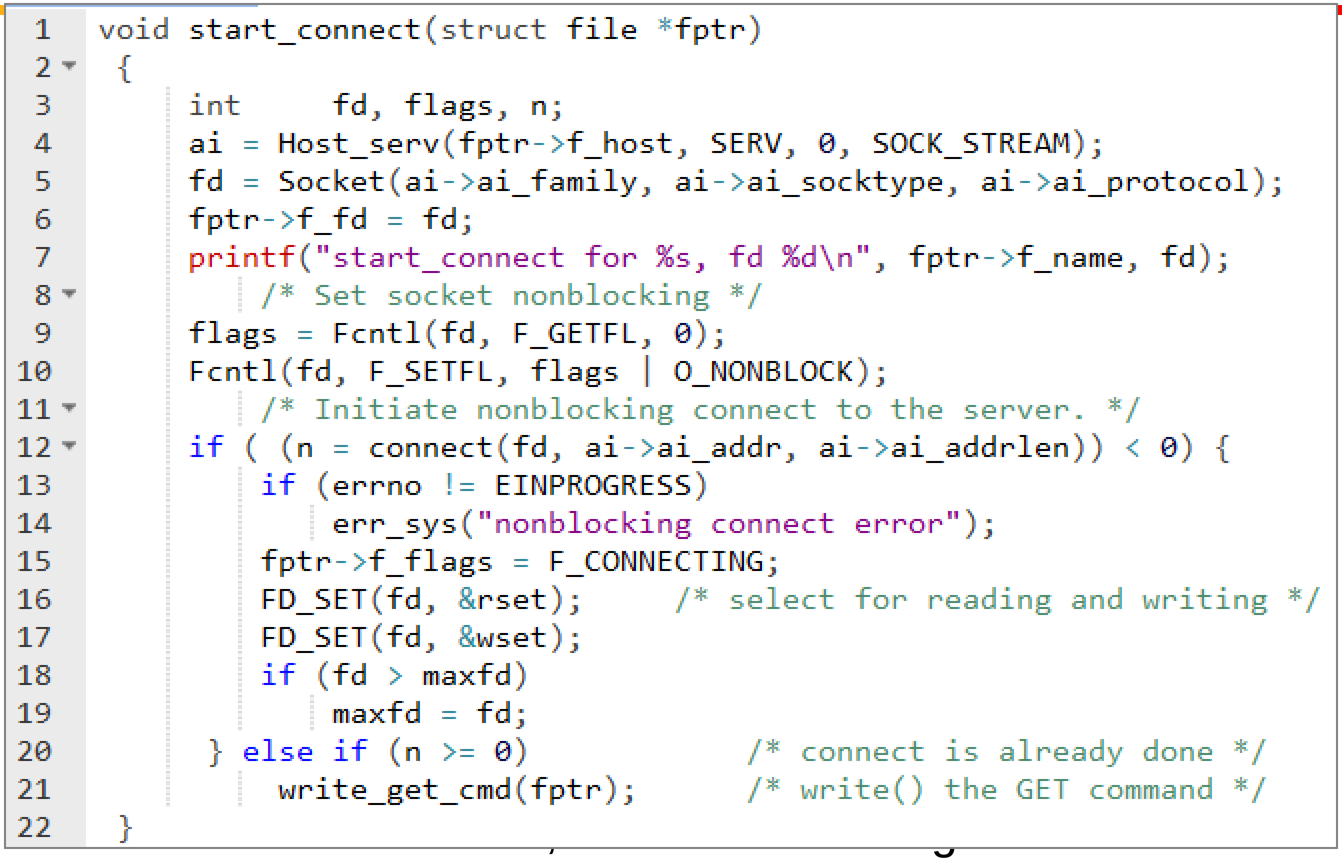
nonblock/web.c



```
main(int argc, char **argv)
{
 int     i, fd, n, maxnconn, flags, error;
 char    buf[MAXLINE];
 fd_set  rs, ws;
 if (argc < 5)
     err_quit("usage: web <#conns> <hostname> <homepage> <file1> ...");
 maxnconn = atoi(argv[1]);
 nfiles = min(argc - 4, MAXFILES);
 for (i = 0; i < nfiles; i++) {
     file[i].f_name = argv[i + 4];
     file[i].f_host = argv[2];
     file[i].f_flags = 0;
 }
 printf("nfiles = %d\n", nfiles);
 home_page(argv[2], argv[3]);
 FD_ZERO(&rset);
 FD_ZERO(&wset);
 maxfd = -1;
 nlefttoread = nlefttoconn = nfiles;
 nconn = 0;
```

- Process command-line arguments
- Read home page
- Initialize globals
  - – Fd sets, nconn is current number of connections.

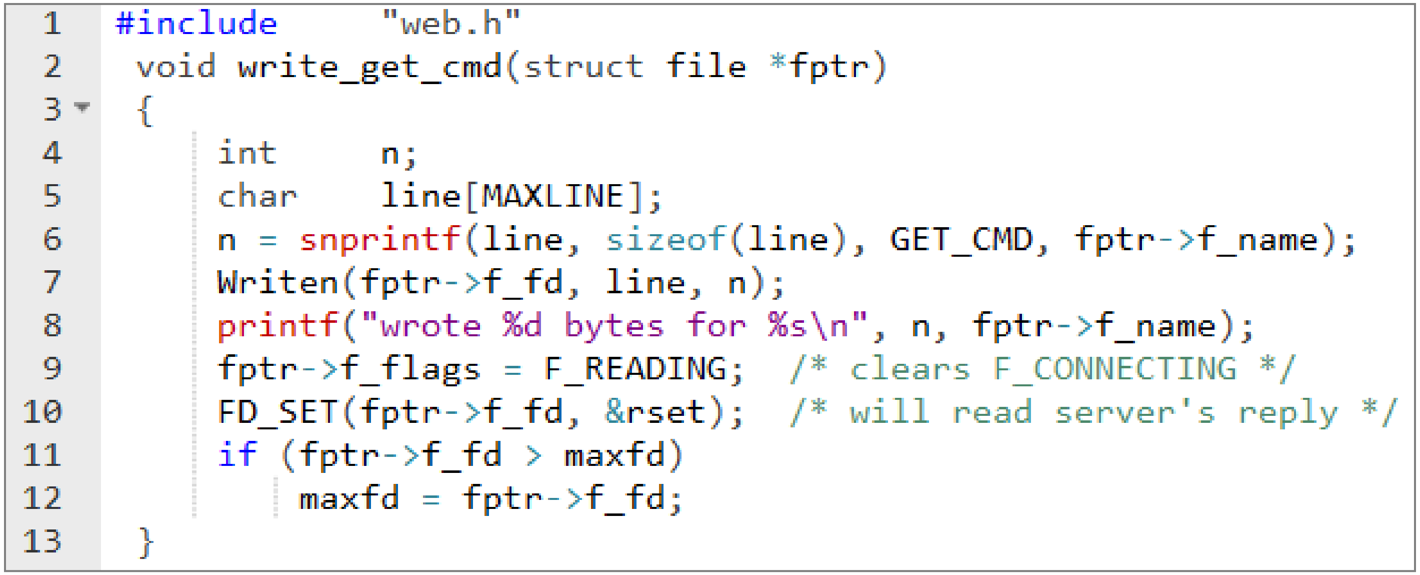
nonblock/start_connect.c



```c
void start_connect(struct file *fptr)
 {
     int     fd, flags, n;
     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);
     fd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
     fptr->f_fd = fd;
     printf("start_connect for %s, fd %d\n", fptr->f_name, fd);
         /* Set socket nonblocking */
     flags = Fcntl(fd, F_GETFL, 0);
     Fcntl(fd, F_SETFL, flags | O_NONBLOCK);
         /* Initiate nonblocking connect to the server. */
     if ( (n = connect(fd, ai->ai_addr, ai->ai_addrlen)) < 0) {
         if (errno != EINPROGRESS)
             err_sys("nonblocking connect error");
         fptr->f_flags = F_CONNECTING;
         FD_SET(fd, &rset);     /* select for reading and writing */
         FD_SET(fd, &wset);
         if (fd > maxfd)
             maxfd = fd;
      } else if (n >= 0)           /* connect is already done */
          write_get_cmd(fptr);     /* write() the GET command */
 }
```

- Initiate nonblocking connect
- Handle connection complete
- If connect returns successfully, the connection is already complete and the function write_get_cmd ends a command to the server.

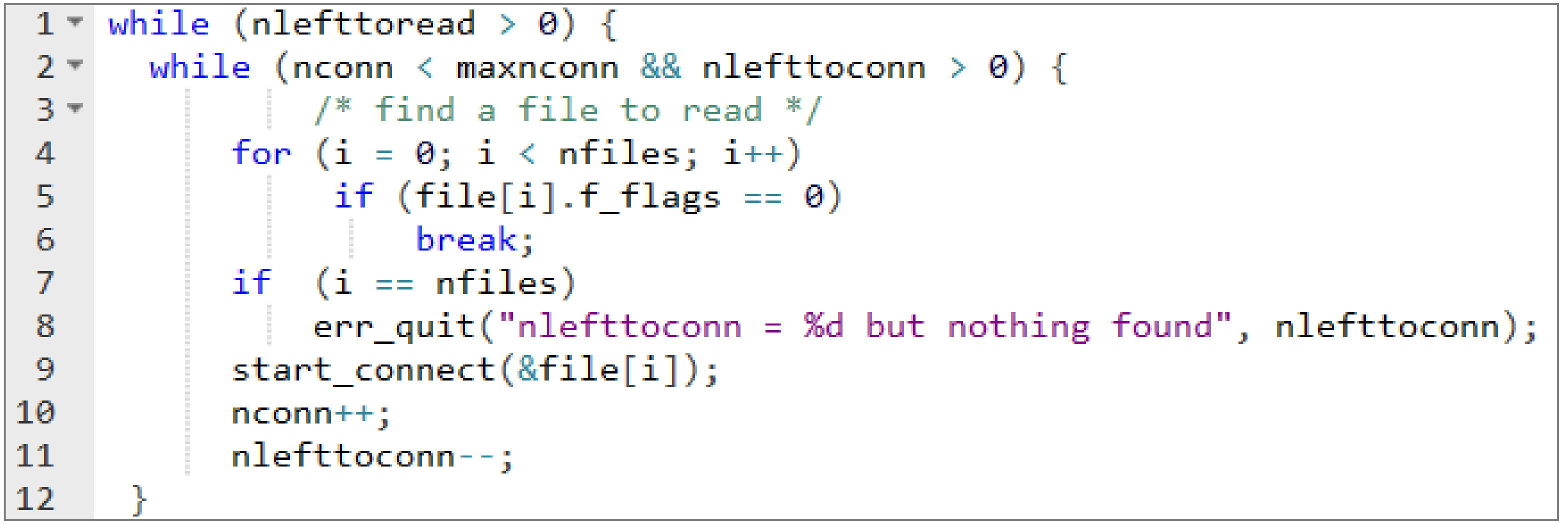
nonblock/write_get_cmd.c



```c
#include      "web.h"
 void write_get_cmd(struct file *fptr)
 {
     int      n;
     char     line[MAXLINE];
     n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
     Writen(fptr->f_fd, line, n);
     printf("wrote %d bytes for %s\n", n, fptr->f_name);
     fptr->f_flags = F_READING;  /* clears F_CONNECTING */
     FD_SET(fptr->f_fd, &rset);  /* will read server's reply */
     if (fptr->f_fd > maxfd)
         maxfd = fptr->f_fd;
 }
```

- Build command and send it
- Set flags

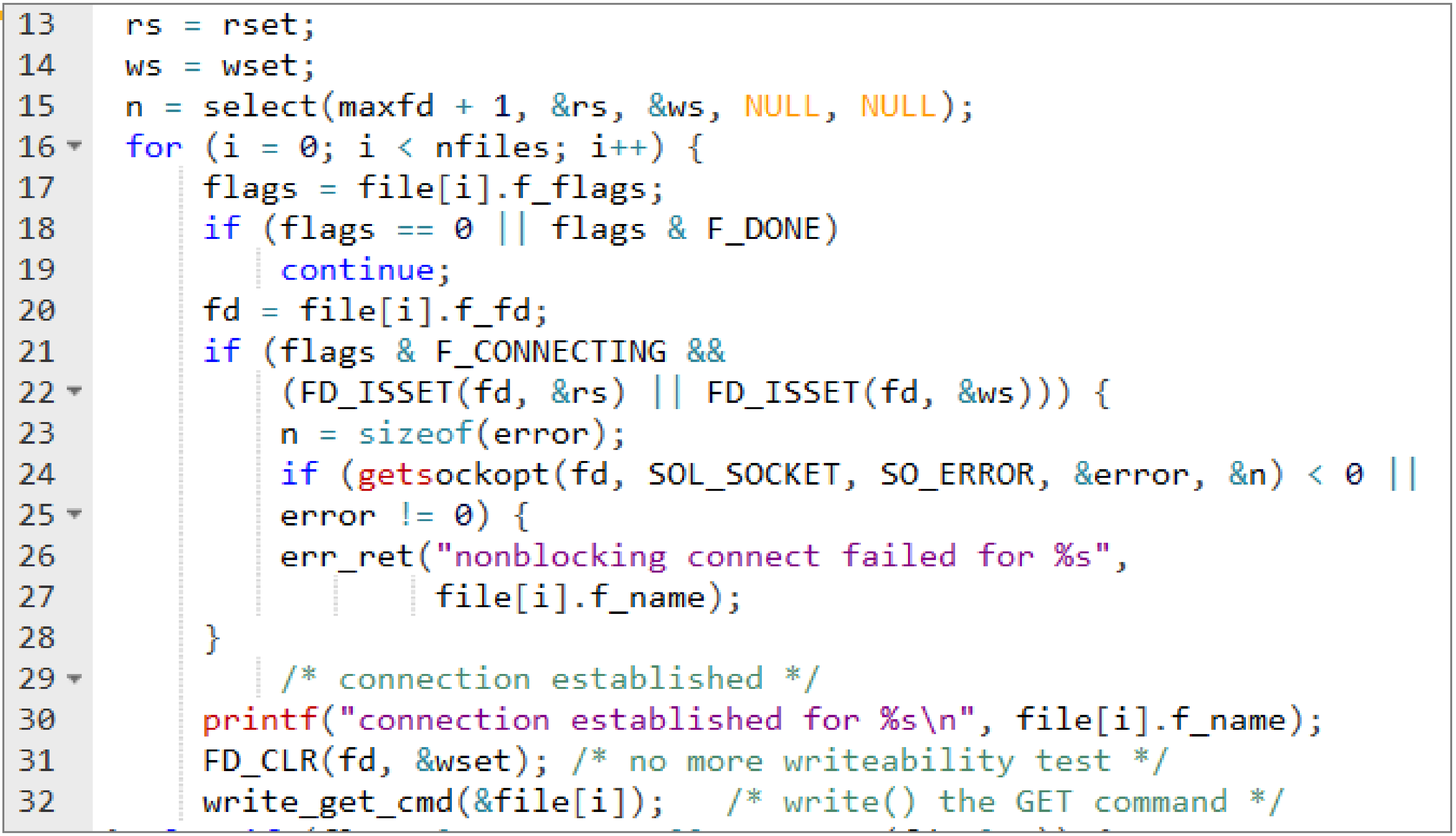
Main function: web.c



```
while (nlefttoread > 0) {
  while (nconn < maxnconn && nlefttoconn > 0) {
          /* find a file to read */
      for (i = 0; i < nfiles; i++)
           if (file[i].f_flags == 0)
               break;
      if  (i == nfiles)
          err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
      start_connect(&file[i]);
      nconn++;
      nlefttoconn--;
 }
```

- Initiate another connection, if possible

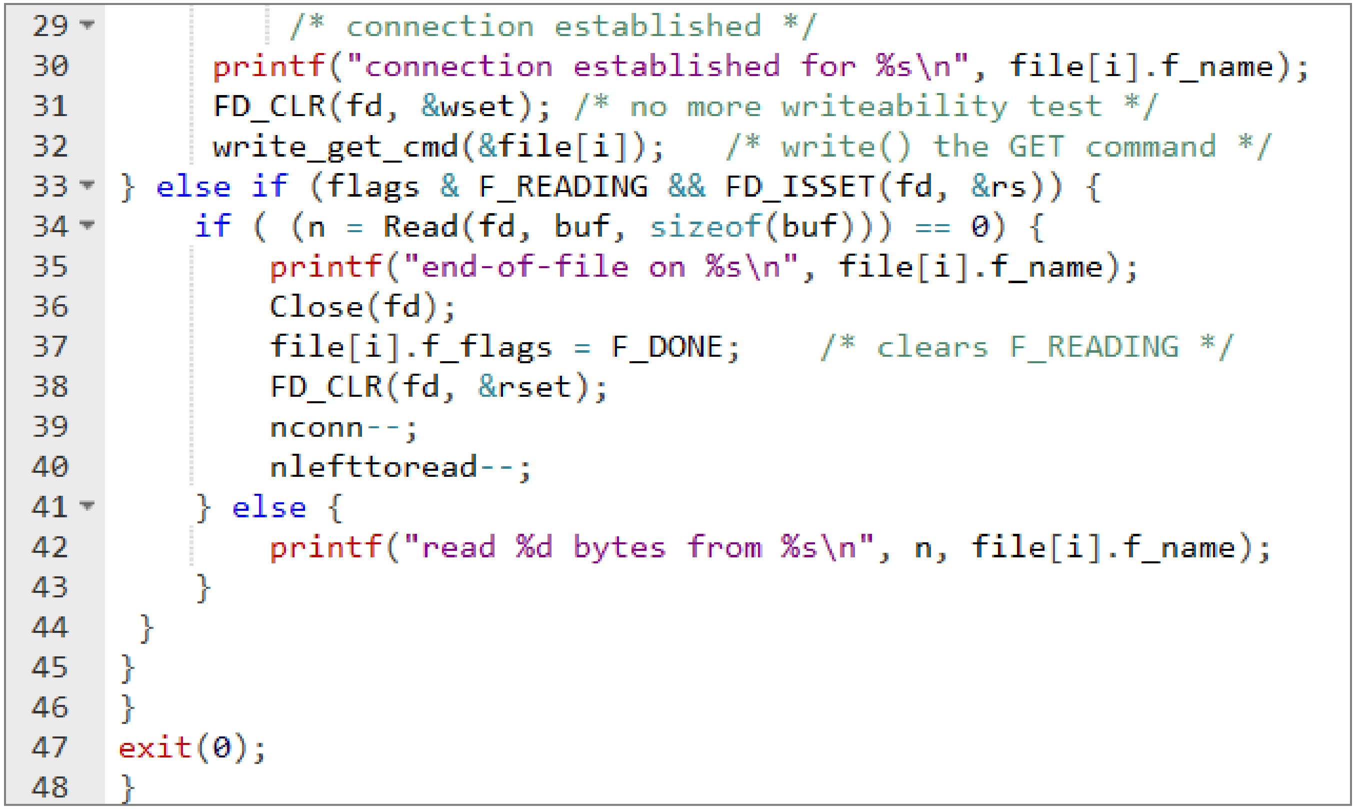
Main function: web.c



```
rs = rset;
ws = wset;
n = select(maxfd + 1, &rs, &ws, NULL, NULL);
for (i = 0; i < nfiles; i++) {
    flags = file[i].f_flags;
    if (flags == 0 || flags & F_DONE)
        continue;
    fd = file[i].f_fd;
    if (flags & F_CONNECTING &&
        (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
        n = sizeof(error);
        if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
        error != 0) {
        err_ret("nonblocking connect failed for %s",
                file[i].f_name);
    }
        /* connection established */
    printf("connection established for %s\n", file[i].f_name);
    FD_CLR(fd, &wset); /* no more writeability test */
    write_get_cmd(&file[i]);   /* write() the GET command */
```

- select waits for either readability or writability.
    - Descriptors that have a nonblocking connect in progress will be enabled in both sets, while descriptors with a completed connection that are waiting for data from the server will be enabled in just the read set.

Main function: web.c



```
         /* connection established */
     printf("connection established for %s\n", file[i].f_name);
     FD_CLR(fd, &wset); /* no more writeability test */
     write_get_cmd(&file[i]);   /* write() the GET command */
} else if (flags & F_READING && FD_ISSET(fd, &rs)) {
    if ( (n = Read(fd, buf, sizeof(buf))) == 0) {
        printf("end-of-file on %s\n", file[i].f_name);
        Close(fd);
        file[i].f_flags = F_DONE;    /* clears F_READING */
        FD_CLR(fd, &rset);
        nconn--;
        nlefttoread--;
    } else {
        printf("read %d bytes from %s\n", n, file[i].f_name);
    }
 }
}
}
exit(0);
}
```

- If the F_READING flag is set and the descriptor is ready for reading, we call read.

Performance of Nonblocking Connect

- Table shows the clock time required to fetch a Web server's home page, followed by nine image files from that server.
  - The RTT to the server is about 150 ms.
  - The home page size was 4,017 bytes and the average size of the 9 image files was 1,621 bytes.
  - TCP's segment size was 512 bytes.
- Most of the improvement is obtained with three simultaneous connections.

# simultaneous connections	Clock time (seconds), non blocking	Clock time(secs) Threads
1	6.0	6.3
2	4.1	4.2
3	3.0	3.1
4	2.8	3.0
5	2.5	2.7
6	2.4	2.5
7	2.3	2.3
8	2.2	2.3
9	2.0	2.2

Acknowledgements

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You