Nonblocking I/O

- Nonblocking reads and writes
  - Default mode is blocking
  - TCP v.s. UDP
- Nonblocking accept
- Nonblocking connect

Nonblocking Reads and Writes
overlapping I/O in str_cli of echo client

- Set descriptors nonblocking using fcntl
- initialize buffer pointers of “to” and “fr” buffers
- main loop to call select
  - specify interested descriptors
  - call select
  - read from standard input
  - handle nonblocking error
  - read returns end-of-file
  - read returns data
  - read from socket
  - write OK
  - write to socket

Buffers Enabling Overlapped Nonblocking I/O in str_cli

Figure 16.1. Buffer containing data from standard input going to the socket.

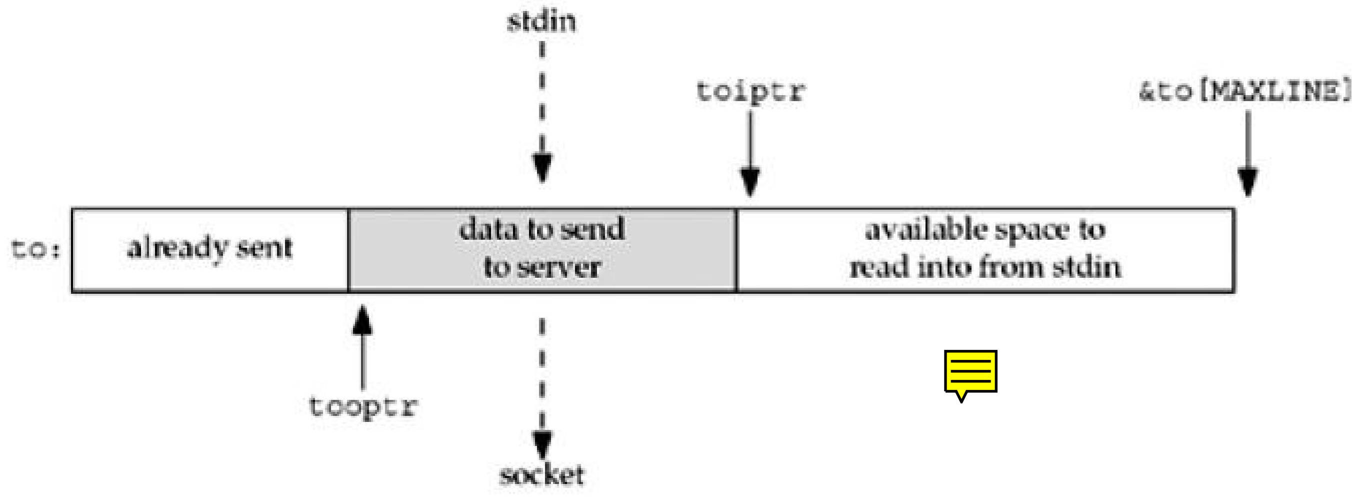


Figure 16.2. Buffer containing data from the socket going to standard output.

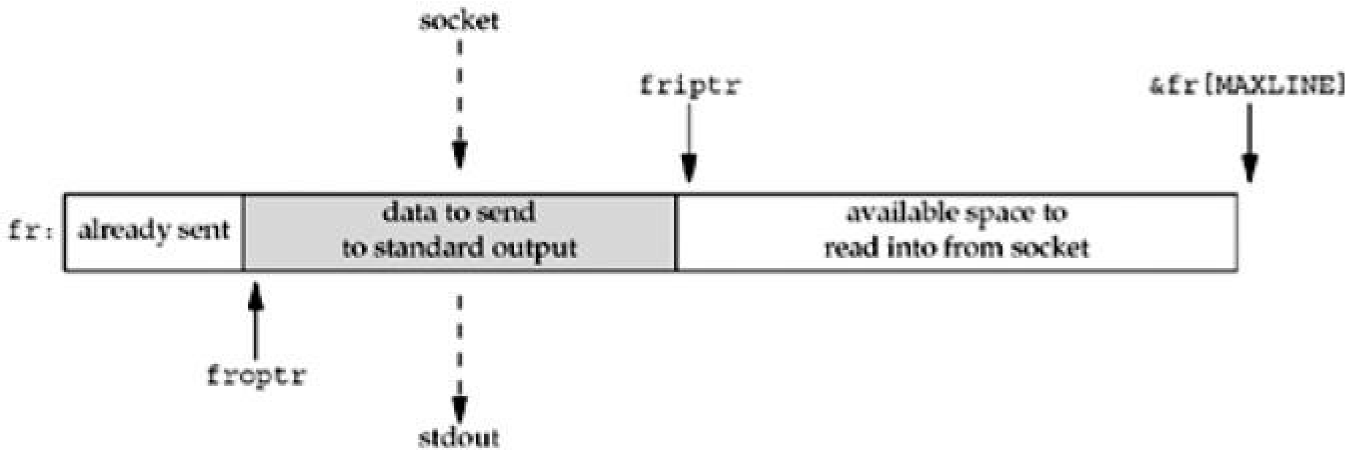


Figure 16.3 `str_cli` function, first part: initializes and calls `select`.

nonblock/strclinonb.c

```
#include    "unp.h"

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);
```

```c
    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);
```

Figure 16.4 `str_cli` function, second part: reads from standard input or socket.

nonblock/strclinonb.c

```
    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 */
        }
    }

    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) {
            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 */
        }
    }

    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 */
        }
    }
```

Figure 16.5 `str_cli` **function, third part: writes to standard output or socket.**

nonblock/strclinonb.c

```
    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 (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) {
```

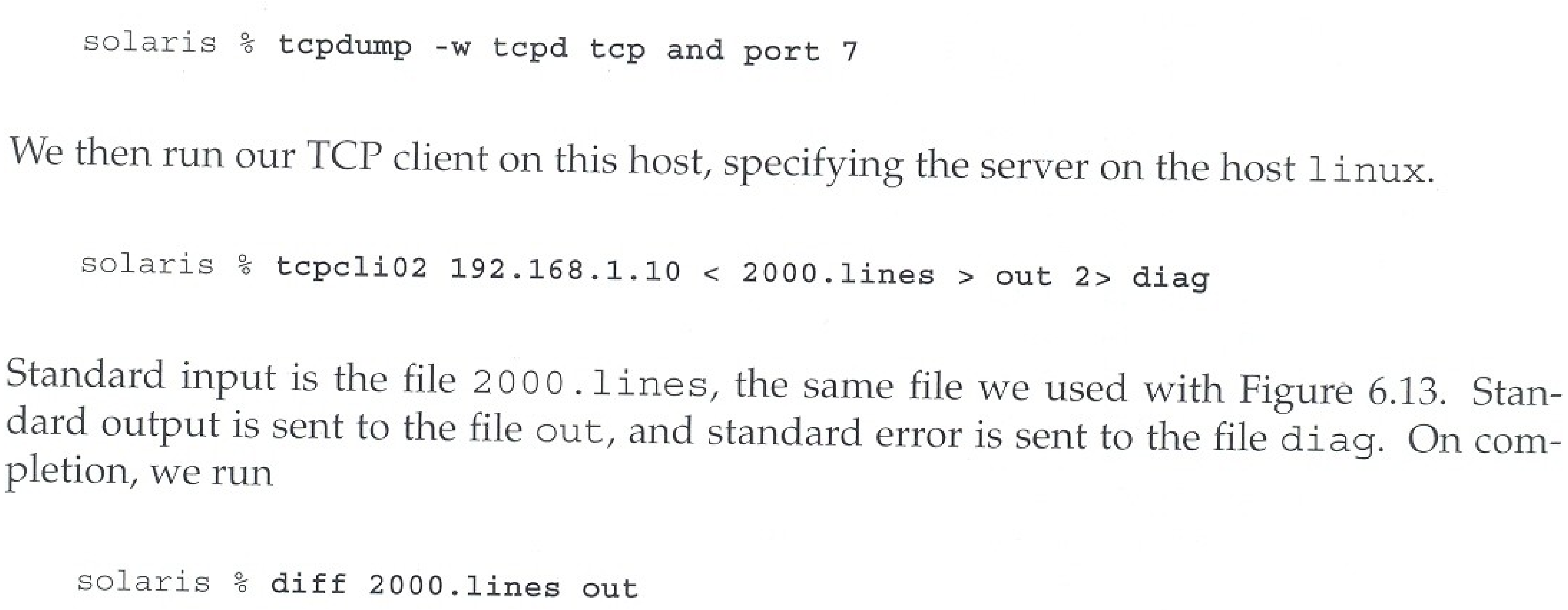
```
                     gf_time(), nwritten);
             froptr += nwritten; /* # just written */
             if (froptr == friptr)
                 froptr = friptr = fr;   /* back to beginning of buffer */
         }
     }

     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 */
             }
         }
      }
   }
}
```

Figure 16.6 `gf_time` function: returns pointer to time string.

lib/gf_time.c

```
#include     "unp.h"
#include     <time.h>

char *
gf_time(void)
{
    struct timeval tv;
    static char str[30];
    char  *ptr;

    if (gettimeofday(&tv, NULL) < 0)
        err_sys("gettimeofday error");

    ptr = ctime(&tv.tv_sec);
    strcpy(str, &ptr[11]);
    /* Fri Sep 13 00:00:00 1986\n\0 */
    /* 0123456789012345678901234 5 */
    snprintf(str + 8, sizeof(str) - 8, ".%06ld", tv.tv_usec);

    return (str);
}
```

An Usage Example



```
solaris % tcpdump -w tcpd tcp and port 7
```

We then run our TCP client on this host, specifying the server on the host `linux`.

```
solaris % tcpcli02 192.168.1.10 < 2000.lines > out 2> diag
```

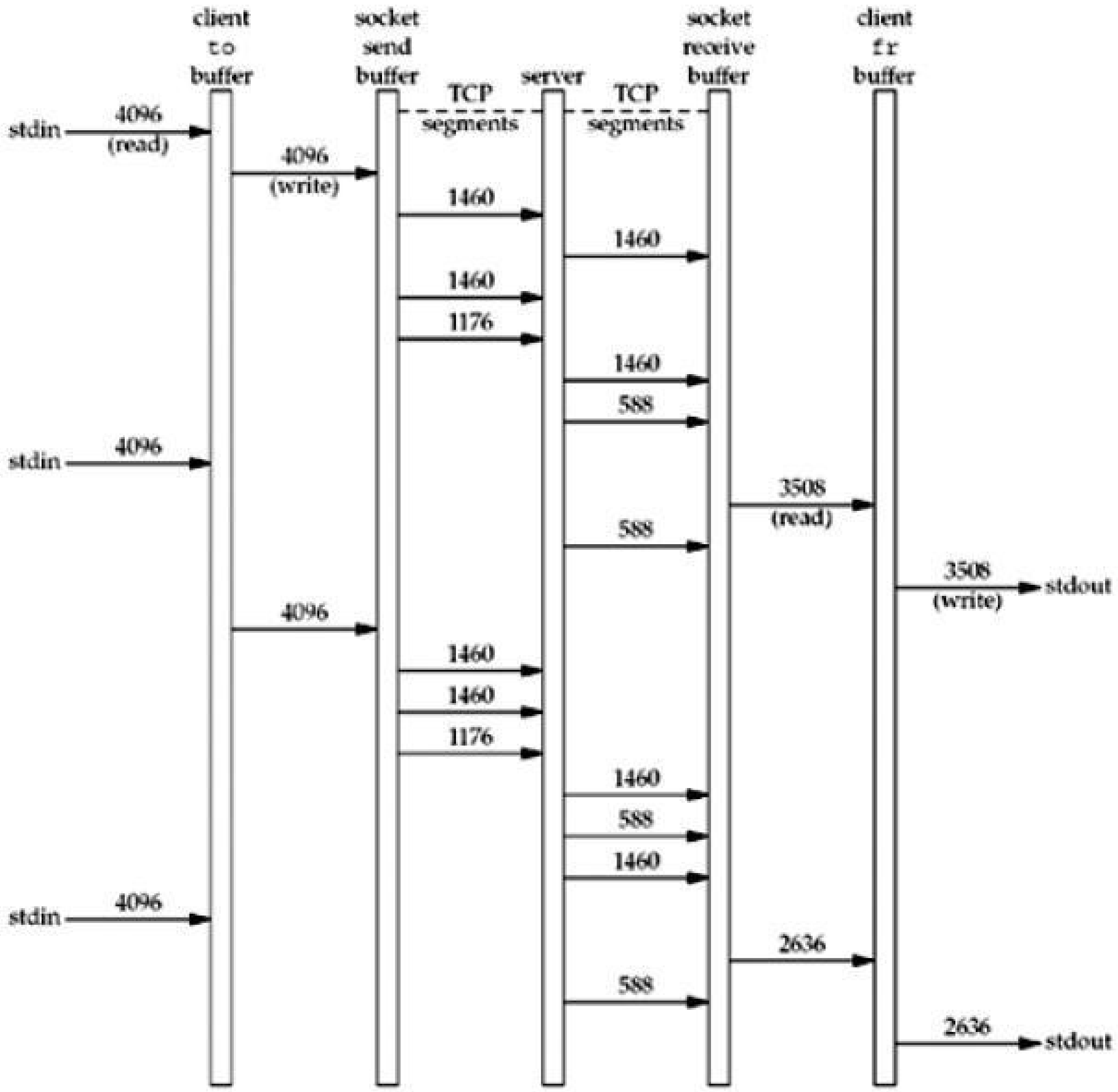
Standard input is the file `2000.lines`, the same file we used with Figure 6.13. Standard output is sent to the file `out`, and standard error is sent to the file `diag`. On completion, we run

```
solaris % diff 2000.lines out
```

Figure 16.7 Sorted output from `tcpdump` and diagnostic output.

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > linux.echo: S 1802738644:1802738644(0)
                                            win 8760 <mss 1460>
10:18:34.488278 linux.echo > solaris.33621: S 3212986316:3212986316(0)
                                            ack 1802738645 win 8760 <mss 1460>
10:18:34.488490 solaris.33621 > linux.echo: . ack 1 win 8760

10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > linux.echo: P 1:1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 linux.echo > solaris.33621: P 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > linux.echo: . 1461:2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > linux.echo: P 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris.33621 > linux.echo: . ack 1461 win 8760
10:18:34.536193 linux.echo > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
10:18:34.536697 linux.echo > solaris.33621: P 2921:3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin
10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris.33621 > linux.echo: . ack 3509 win 8760
10:18:34.582244 linux.echo > solaris.33621: P 3509:4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > linux.echo: P 4097:5557(1460) ack 4097 win 8760
10:18:34.617610 solaris.33621 > linux.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > linux.echo: P 7017:8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 linux.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 linux.echo > solaris.33621: . 4097:5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > linux.echo: . ack 5557 win 8760
10:18:34.626611 linux.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 linux.echo > solaris.33621: . 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin
10:18:34.667305: read 2636 bytes from socket
10:18:34.670324 solaris.33621 > linux.echo: . ack 7605 win 8760
10:18:34.672221 linux.echo > solaris.33621: P 7605:8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

Figure 16.8. Timeline of nonblocking example.



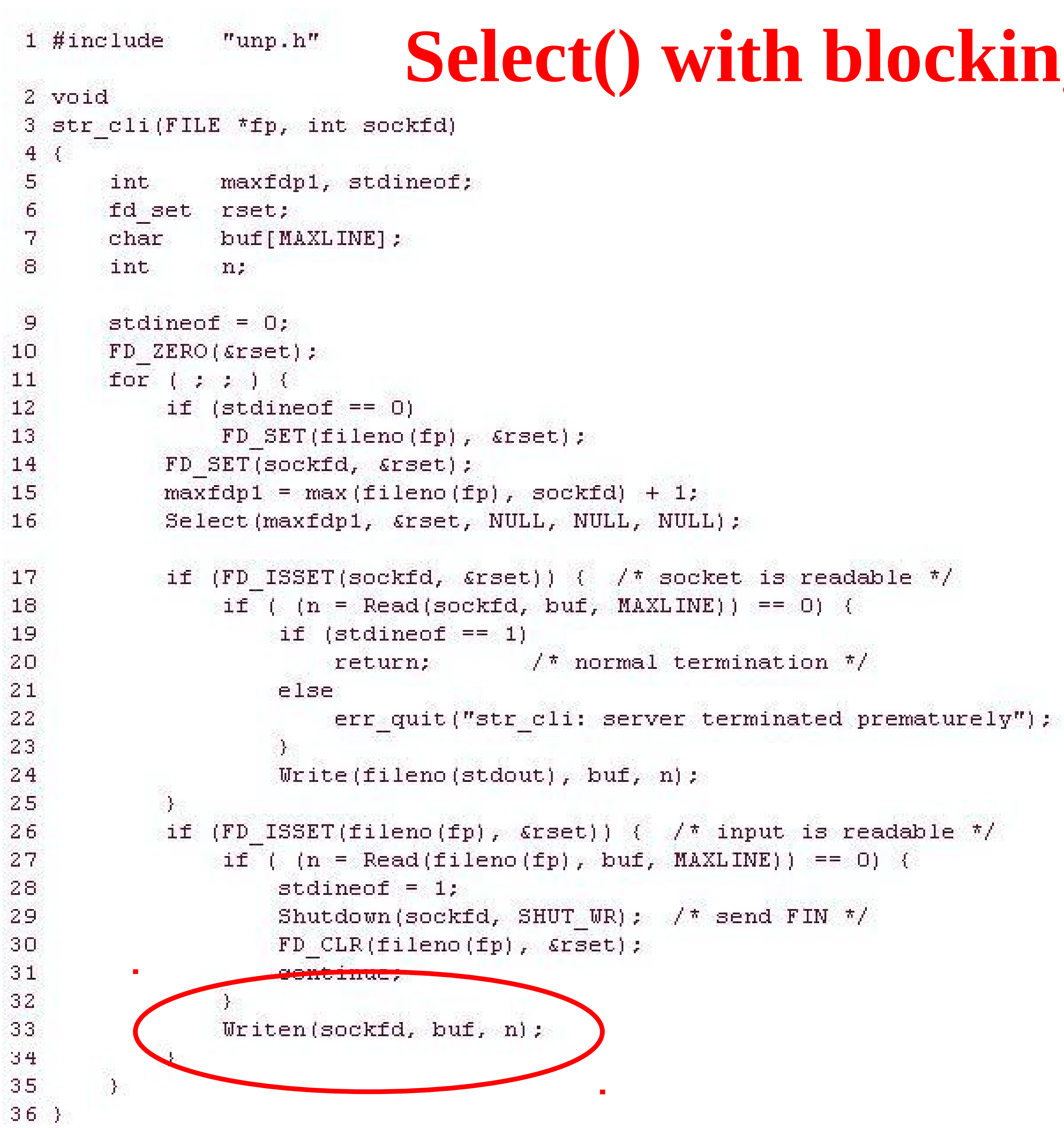
Is the effort worth it at all?

- The nonblocking version is nontrivial: about 135 lines of code
- The version using select with blocking I/O in Fig. 6.13 has only 36 lines of code.
- The original stop-and-wait version in Fig. 5.5 has only 12 lines of code.

Select() with blocking I/O (40 lines)

Figure 6.13 `str_cli` function using `select` that handles EOF correctly.

select/strcliselect02.c



```c
#include    "unp.h"

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

    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);
        }
    }
}
```

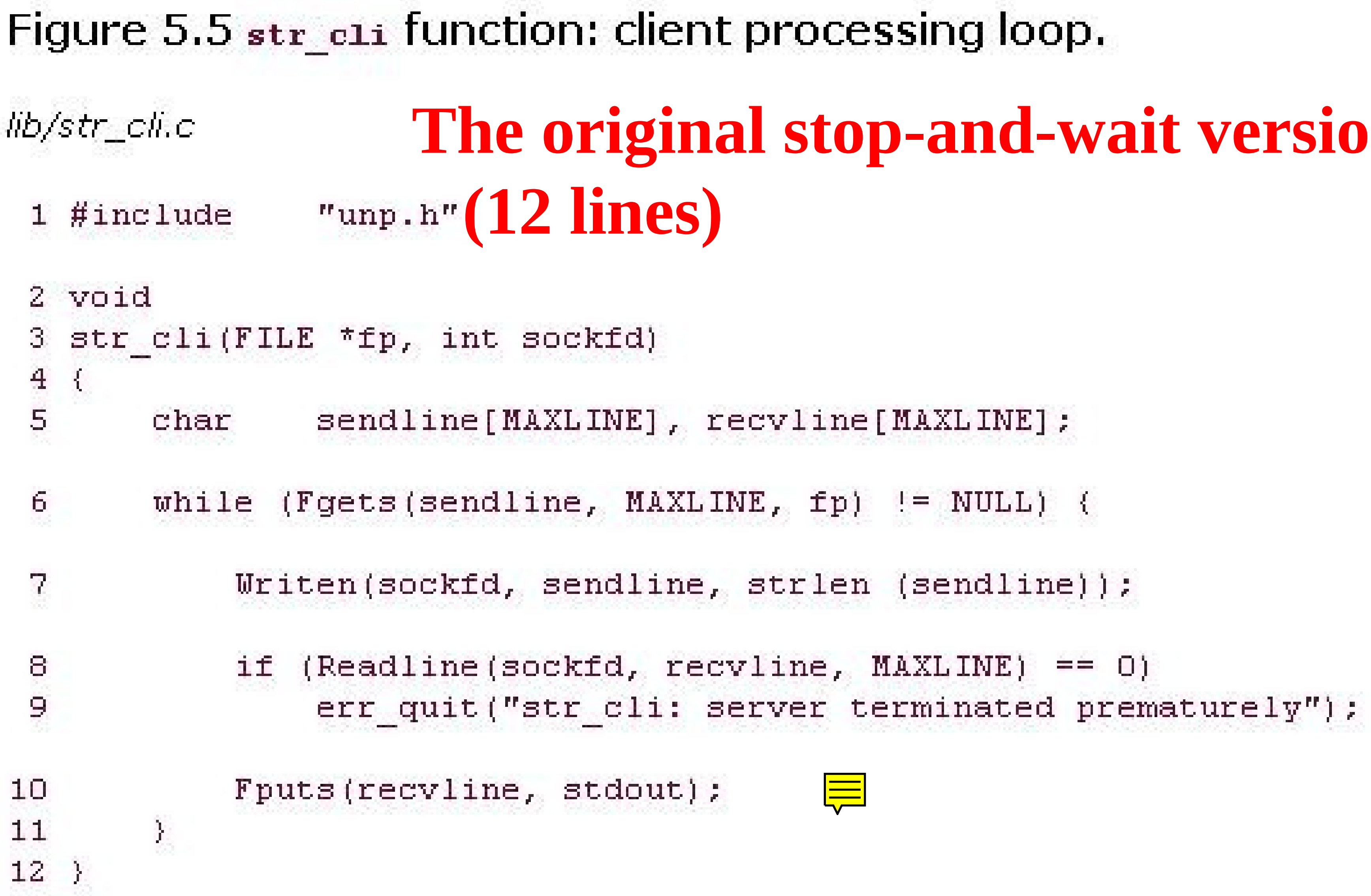


Figure 5.5 `str_cli` function: client processing loop.

lib/str_cli.c

The original stop-and-wait version (12 lines)

```c
#include    "unp.h"

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);
    }
}
```

A Simpler Version Using Two Processes

Figure 16.9. `str_cli` **function using two processes.**

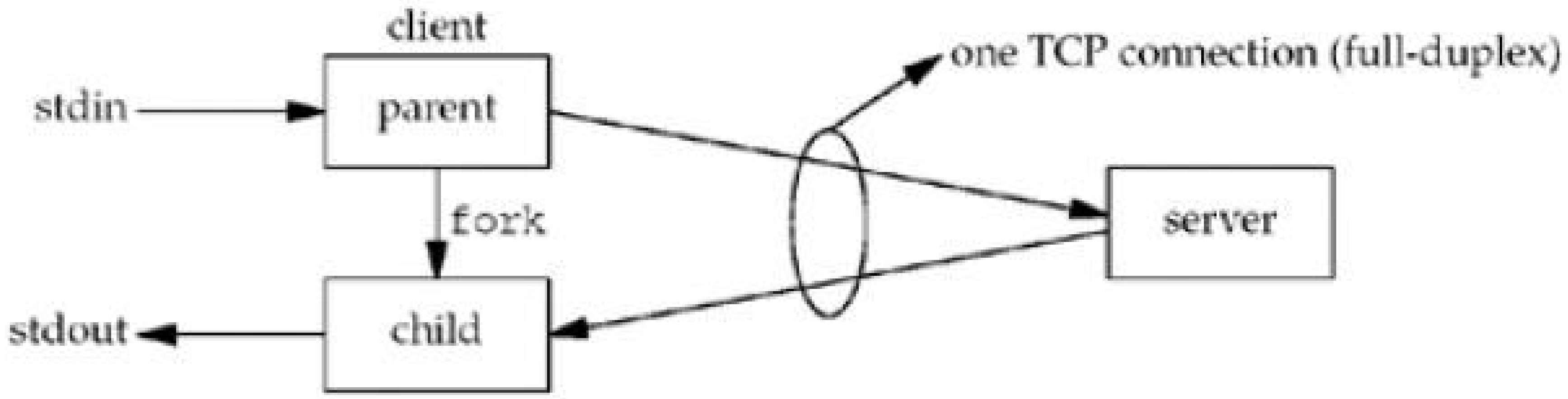
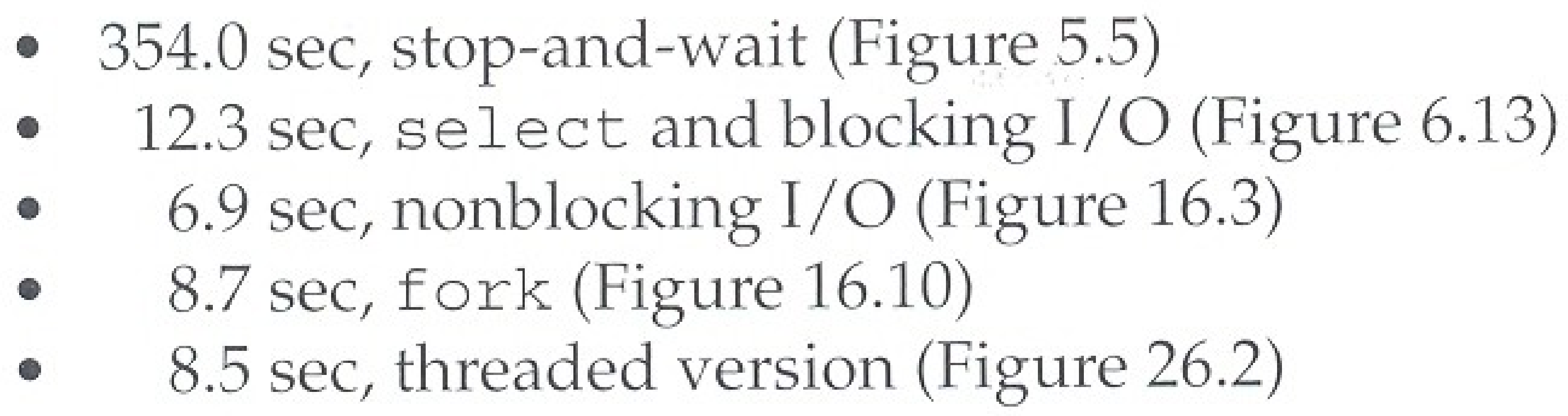


Figure 16.10 Version of `str_cli` function that uses `fork`.

nonblock/strclifork.c

```
#include    "unp.h"

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;
}
```

Performance Comparison



- 354.0 sec, stop-and-wait (Figure 5.5)
- 12.3 sec, `select` and blocking I/O (Figure 6.13)
- 6.9 sec, nonblocking I/O (Figure 16.3)
- 8.7 sec, `fork` (Figure 16.10)
- 8.5 sec, threaded version (Figure 26.2)

Your choice?

Nonblocking connect: Daytime Client

- Set socket nonblocking by *fcntl*
- Overlap processing with connection establishment
- Check for immediate completion
- Call *select*
- Handle timeouts
- Check for readability or writability
- Turn off nonblocking and return

Figure 16.11 Issue a nonblocking `connect`**.**

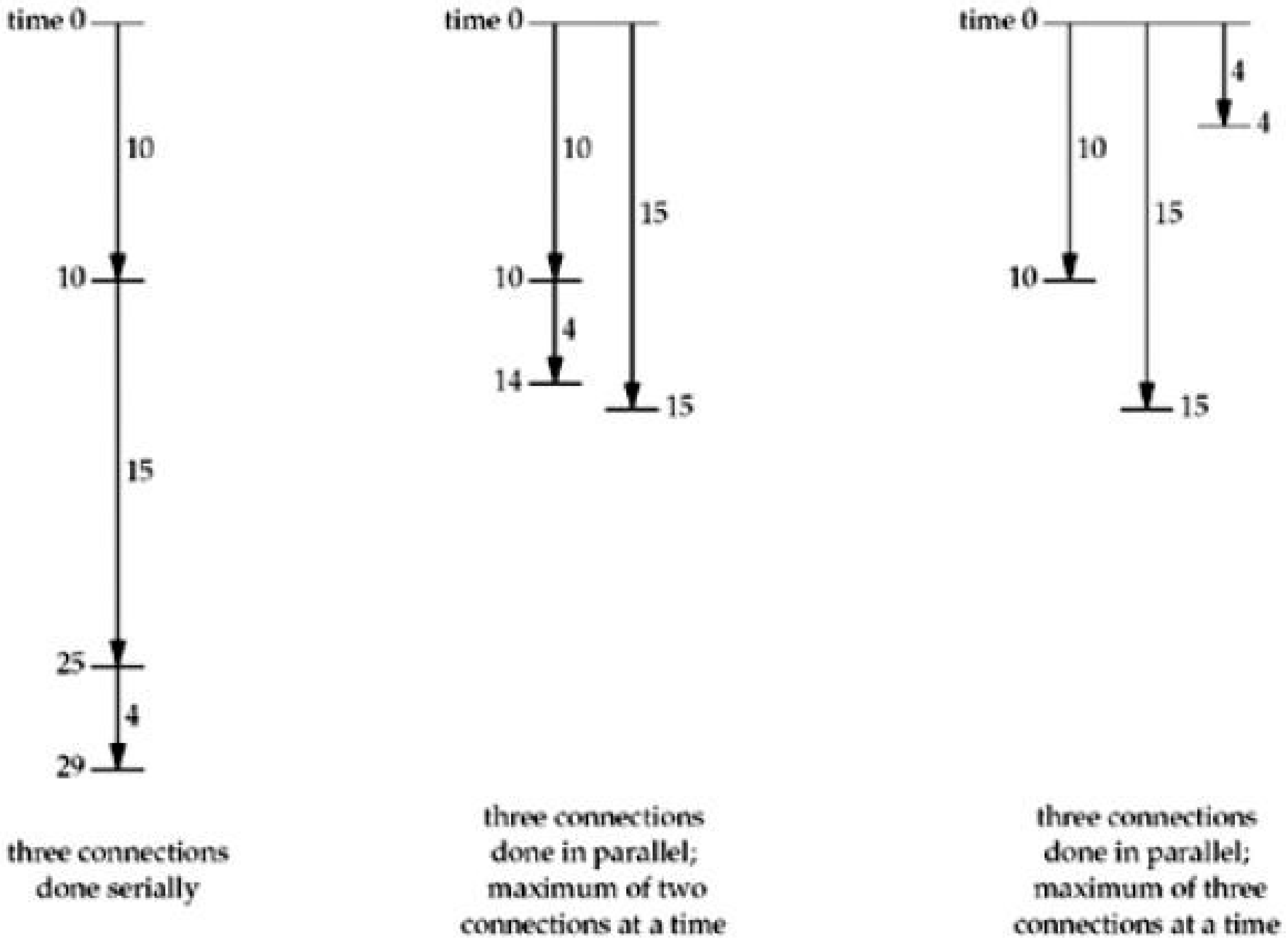
lib/connect_nonb.c

```
#include    "unp.h"

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;

    if ( (n = Select(sockfd + 1, &rset, &wset, NULL,
                     nsec ? &tval : NULL)) == 0) {
        close(sockfd);          /* timeout */
        errno = ETIMEDOUT;
        return (-1);
    }
    tval.tv_sec = nsec;
    tval.tv_usec = 0;

    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);
}
```

Nonblocking connect: Web Client

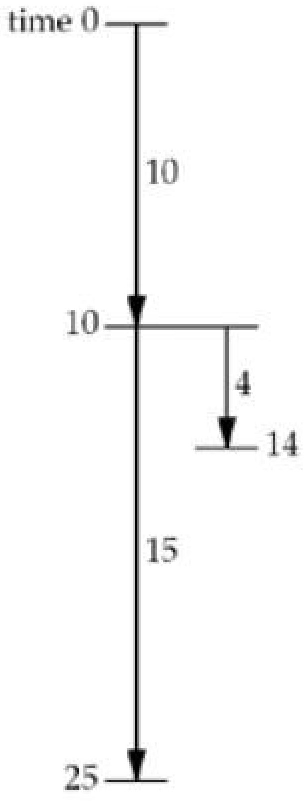
- Do you know why Netscape web browser beat Mosaic web browser in 1995?
- It uses the parallel download technique.

Figure 16.12. Establishing multiple connections in parallel.


time 0
10
10
15
25
4
29
three connections done serially
time 0
10
15
10
4
14
15
three connections done in parallel; maximum of two connections at a time
time 0
4
4
10
15
10
15
three connections done in parallel; maximum of three connections at a time

The first web page download can only be done serially

Figure 16.13. Complete first connection, then multiple connections in parallel.



An Usage Example of a Parallel-Download Web Browser

```
solaris % web  3  www.foobar.com  /  image1.gif  image2.gif \
image3.gif  image4.gif  image5.gif \
image6.gif  image7.gif
```

Figure 16.14 `web.h` header.

nonblock/web.h

```
#include     "unp.h"

#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 *);
```

Figure 16.15 First part of simultaneous `connect`: globals and start of `main`.

nonblock/web.c

```
#include     "web.h"

int
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;
```

Figure 16.16 `home_page` **function.**

nonblock/home_page.c

```
#include     "web.h"

void
home_page(const char *host, const char *fname)
{
    int     fd, n;
    char    line[MAXLINE];
    fd = Tcp_connect(host, SERV);   /* blocking connect() */

    n = snprintf(line, sizeof(line), GET_CMD, fname);
    Writen(fd, line, n);

    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;              /* server closed connection */

        printf("read %d bytes of home page\n", n);
        /* do whatever with data */
    }
    printf("end-of-file on home page\n");
    Close(fd);
}
```

Figure 16.17 Initiate nonblocking `connect`.

nonblock/start_connect.c

```c
#include    "web.h"

void
start_connect(struct file *fptr)
{
    int     fd, flags, n;
    struct addrinfo *ai;

    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");
    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 */
}
```

Figure 16.18 Send an HTTP GET command to the server.

nonblock/write_get_cmd.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;
}
```

Figure 16.19 Main loop of `main` function.

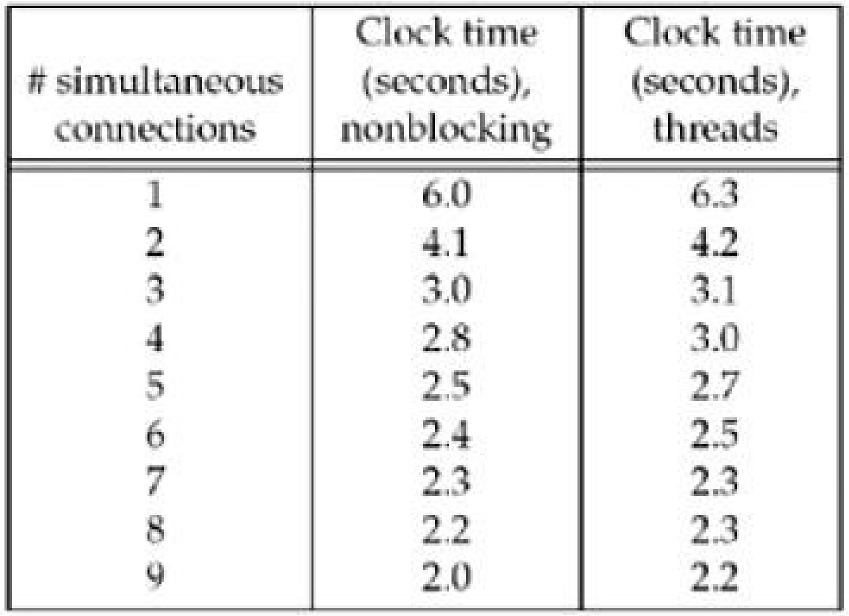
nonblock/web.c

```
    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--;
        }

        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);
```

```
   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 */

  } 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);
      }
   }
}
```

Using Three Parallel Download Is Good Enough

Figure 16.20. Clock time for various numbers of simultaneous connections.



# simultaneous connections	Clock time (seconds), nonblocking	Clock time (seconds), 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

Blocking Accept with Select() May Cause a Problem

- Suppose that a client initiates a connection and after the connection is set up, the client immediately disconnects it.
  - Why? The server may be too busy to “accept” the connection immediately.
- When the server finally has time to “accept” the connection, the aborted connection has been removed from the kernel, causing the server to block!

```
if (FD_ISSET(listenfd, &rset)) {     /* new client connection *
    printf("listening socket readable\n");
    sleep(5);
    clilen = sizeof(cliaddr);
    connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
```

A Client That Issues RST Immediately After Getting Connected

Figure 16.21 TCP echo client that creates connection and sends an RST.

nonblock/tcpcli03.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct linger ling;
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    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);

    exit(0);
}
```

How to Handle This Problem?

- Always set a listening socket to nonblocking when you use select() to indicate when a connection is ready to be accepted.
- Ignore the following error on the subsequent call to accept(): EWOULDBLOCK (for BSD), etc.