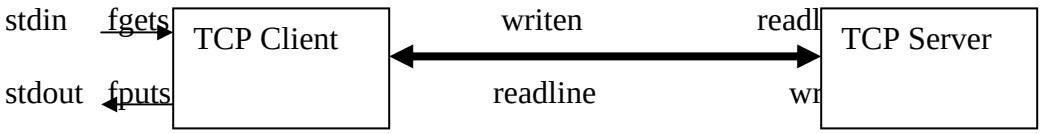
TCP CLIENT - SERVER COMMUNICATION

1. Client – Server communication involves reading of text (character or text) from the client and writing the same into the server and server reading text and client writing the same. Following picture depicts the same.



Simple Echo client server.

Functions ***fgets() and fputs()*** are from standard I/O library. And ***writen() and readline()*** are function created by the W Richard Stevans (WRS) (code given in the section 3.9)

The communication between client and server is understood by Echo client and Server. The Following code corresponds to Server side program.

tcpcliserv/tcpserv01.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;
 9     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13     servaddr.sin_port = htons(SERV_PORT);
14     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
15     Listen(listenfd, LISTENQ);
16     for ( ; ; ) {
17         clilen = sizeof(cliaddr);
18         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
19         if ( (childpid = Fork()) == 0) {     /* child process */
20             Close(listenfd);    /* close listening socket */
21             str_echo(connfd);   /* process the request */
22             exit(0);
23         }
24         Close(connfd);          /* parent closes connected socket */
25     }
26 }
```

tcpcliserv/tcpserv01.c

Figure 5.2 TCP echo server.

Line 1: It is the header created by the WRS which encapsulates a large number of header that are required for the functions that are referred.
Line 2 – 3: This the definition of the ***main()*** with command line arguments.
Line 5-8 : These are variable declarations of types that are used.
Line 9 : It is the system call to the **socket** function that returns a descriptor of type int.- in this case it is named as listenfd. The arguments are family type, stream type and protocol argument – normally 0)
Line 10: the function ***bzero()*** sets the address space to zero.

Line 11-12: Sets the internet socket address to wild card address and the server port to the number defined in **SERV_PORT** which is 9877 (specified by WRS). It is an intimation that the server is ready to accept a connection destined for any local interface in case the system is multi homed.
Line 14 : ***bind ()*** function binds the address specified by the address structure to the socket.
Line 15: The socket is converted into listening socket by the call to the listen()function
Line 17-18: The server blocks in the call to accept, waiting for a client connection to complete.
Line 19 – 24: For each client, ***fork()*** spawns a child and the child handles the new client. The child closes the listening socket and the parent closes the connected socket The child then calls ***str_echo ()*** to handle the client

2. TCP Echo Server : **str_echo** function

This is shown in the following figure. It shows the server processing for each client: reading the lines from the client and echoing them back to the client.

lib/str_echo.c

```
 1 #include   "unp.h"
 2 void
 3 str_echo(int sockfd)
 4 {
 5     ssize_t n;
 6     char    line[MAXLINE];
 7     for ( ; ; ) {
 8         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
 9             return;          /* connection closed by other end */
10         Writen(sockfd, line, n);
11     }
12 }
```

lib/str_echo.c

Figure 5.3 str_echo function: echo lines on a socket.

Line 6: MAXLINE is specified as constant of 4096 characters.
Line 7-11: ***readline*** reads the next line from the socket and the line is echoed back to the client by ***writen*** If the client closes the connection, the recept of client's FIN causes the child's readline to return 0. This causes the str_echo function to return which terminates the child.
TCP Echo Client : Main Function : Following code shows the TCP client main function.

tcpcliserv/tcpcli01

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr;
 7     if (argc != 2)
 8         err_quit("usage: tcpcli <IPaddress>");
 9     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
13     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15     str_cli(stdin, sockfd);     /* do it all */
16     exit(0);
17 }
```

tcpcliserv/tcpcli01.

Figure 5.4 TCP echo client.

Line 9 – 13: A TCP socket is created and an Internal socket address structure is filled in with the server's IP address and port number. We take the server's IP address from the command line argument and the server's well known port (SERV_PORT) from the header.

Line 13: The function inet_pton () converts the argument received at the command line from presentation to numeric value and stores the same at the address specified as the third arguments.

Line 14 –1 5: Connection function establishes the connection with the server. The function str_cli () than handles the client processing.

TCP Echo Client : str _ cli function:

lib/str_cli.c

```
1 #include    "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     char    sendline[MAXLINE], recvline[MAXLINE];

6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

7         Writen(sockfd, sendline, strlen(sendline));

8         if (Readline(sockfd, recvline, MAXLINE) == 0)
9             err_quit("str_cli: server terminated prematurely");

10        Fputs(recvline, stdout);
11    }
12 }
```

lib/str_cli.c

Figure 5.5 `str_cli` function: client processing loop.

Above function handles the client processing loop. That is read a line of text from standard input, write it to the server, read back the server's echo of the line and output the echoed line to standard input.

Line 6-7 : ***fgets*** reads a line of text and ***writen*** sends the line to the server.
Line 8 – 10: readline reads the line echoed back from the server and fputs writes it to the standard output.

Signals – Introduction.

A signal is a message (an integer) sent to a process. The receiving process can try to ignore the signal or call a routine (a so-called *signal handler*). After returning from the *signal handler*, the receiving process will resume execution at the point at which it was interrupted. The system-calls to deal with signals vary between one Unix version and another

The following conditions can generate a signal(1):

- When user presses terminal keys, the terminal will generate a signal. For example, when the user breaks a program by CTRL + C key pair.
- Hardware exceptions can generate signals, too: Division by 0, invalid memory reference. Inexperienced programmers often get SIGSEGV (Segmentation Violation signal) because of an invalid address in a pointer.
- Processes can send signals to themselves by using kill(2) system call (If permissions allow).
- Kernel can generate signal to inform processes when something happens. For example, SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader.

Signals can be sent using the `kill()` routine. The `signal()` and `sigaction()` routines are used to control how a process should respond to a signal.

Posix Signal Handling :

- Every signal has a ***disposition***, which is called the ***action*** associated with the signal. The disposition is set by calling the ***sigaction* function**.
- We have three choices for the disposition:
  a) Whenever a specific signal occurs, a specific function can be provided. This function is called ***signal handler*** and the action is called ***catching*** the signal.
     - The two signal SIGKILL and SIGSTOP cannot be caught – this is an exception.
     - The function is called with a single integer argument that is the signal number and the function returns nothing as shown below:

       const struct sigaction act;
       sigaction (SIGCHLD, &act, NULL)
     - Calling *sigaction* and specifying a function to be called when the signal occurs is all that is required to catch the signal.
     - For few signal like SIGIO, SIGPOLL, and SIGURG etc additional actions on the part of the process is required to catch the signal.

  b) A signal a can be **ignore**d by setting its disposition to **SIG_IGN.** Again the two signals SIGKILL and SIGSTOP are exceptions.

  c) We can set the **default** disposition for a signal by setting its disposition to **SIG_DFL.** The default is normally to terminate a process on the receipt of a signal, with certain signal also generating a core image of the process in its current working directory. The signals whose default disposition is to be ignored are : SIGCHLD AND SIGURG(sent on arrival of out of band data.)

with appropriate settings in the `sigaction` structure you can control the current process's response to receiving a `SIGCHLD` signal. As well as setting a signal handler, other behaviours can be set. If

- `act.sa_handler` is `SIG_DFL` then the default behaviour will be restored
- `act.sa_handler` is `SIG_IGN` then the signal will be ignored if possible (`SIGSTOP` and `SIGKILL` can't be ignored)
- `act.sa_flags` is `SA_NOCLDSTOP` - `SIGCHLD` won't be generated when children stop.
- `act.sa_flags` is `SA_NOCLDWAIT` - child processes of the calling process will not be transformed into zombie processes when they terminate.

signal Function :

Posix way to establish the disposition of a signal is to call the **sigaction** function. However this is complicated at one argument to the function is a structure that we must allocate and fill in. An easier way to set the disposition for a signal is to call the **signal function.** The first argument is the **signal name** and the second arguments is the **pointer to a function** or one of the constants **SIG_IGN** or **SIG_DFE.** Normally, the define our own **signal function** that just calls the **Posix sigaction.**

lib/signal.c

```
 1 #include    "unp.h"
 2 Sigfunc *
 3 signal(int signo, Sigfunc *func)
 4 {
 5     struct sigaction act, oact;
 6     act.sa_handler = func;
 7     sigemptyset(&act.sa_mask);
 8     act.sa_flags = 0;
 9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART;     /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return (SIG_ERR);
20     return (oact.sa_handler);
21 }
```

lib/signal.c

Figure 5.6 `signal` function that calls the Posix `sigaction` function.

line 2-3 call to the function when a signal occurs. It has pointer to signal handling function as the second argument

Line 6: Set Handler : The **sa_handler** member of the sigaction structure is set to the func argument
Line 7: Set signal mask to handler: POSIX allows us to specify a set of signals that will be blocked when our signal handler is called. Any signal that is blocked cannot be delivered to the process. We set the **sa_mask** member to the empty set, which means that no additional signals are blocked while our signal handler is running Posix guarantees that he signal being caught is always blocked while its handler is executing

Line 8 – 17: An optional flag SA_RESTART, if it is set, a system call interrupted by this signal will automatically restarted by the kernal.

Line 18 – 20: The function sigaction is called and then return the old action for the signal as the return value of the signal function.

Posix Signal semantics:

1. Once a signal handler is installed, it remains installed.
2. While a signal handler is executing, the signal being delivered is blocked. Further any additional signals that were specified in the sa_mask signal set passed to sigaction when the handler was installed are also blocked . When the **sa_mask** is set to empty set, no additional signals are blocked other than the signal being caught.
3. If a signal is being generated more times while it is blocked, it is normally delivered only one time after the signal is unblocked.
4. It is possible to selectively block and unblock a set of signals using the ***sigprocmask*** function. This protects a critical region of code by providing certain signals from being caught while the region of code is executing.

Handling SIGCHLD signals:

A zombie is a process that ha terminated whose parent is still running, but has not yet waited for its child processes. This will result in the resources occupied by the terminated process not returned to the system. If there are a lot of zombies in the system the system resources may run out.

The purpose of the zombie state is to maintain information about the child for the parent to fetch at some time later. The information are : the process ID of the child, its termination status and information on the resource utilization of the child (CPU time, memory etc). If a process terminates, and the process has children in the zombie state. The parent process ID of all the zombie children is set to 1 (the init process), which will inherit the children and clean them up.

The zombie are not desired to be left. They take up space in the kernal and eventually we can run out of process. Whenever we fork children, we must wait, for them to prevent them from becoming zombies.

When a child terminates, the kernal generates a SGICHLD signal for the parent. The parent process should catch the signal and take an appropriate action. If the signal is not caught, the default action of this signal is to be ignored.

To handle a zombie, the parent process establishes a signal handler to catch SIGCHLD signal. The handler then invokes the function wait() or waitpid () to wait for a terminated child process and free all the system resources occupied by the process.

To do this we establish a signal handler called SIGCHLD and within the handler, we call wait. The signal handler is established by adding function call

Signal(SIGCHLD, sig_chld)

tcpcliserv/sigchldwait.c

```
1 #include    "unp.h"
2 void
3 sig_chld(int signo)
4 {
5     pid_t   pid;
6     int     stat;
7     pid = wait(&stat);
8     printf("child %d terminated\n", pid);
9     return;
10 }
```

tcpcliserv/sigchldwait.c

Figure 5.7 Version of SIGCHLD signal handler that calls wait.

Interrupted System Calls:

There are certain system calls in the Client Server communication that are blocked while waiting for input. It can be seen that in the case of echo client server programme, in the server programme, the function ***accept ()*** is blocked while waiting for the call from a client. This condition is called ***slow system call*** . This can also occur in case of functional calls such as read(), write (), select(), open () etc. Under such condition, when a SIGCHLD is delivered on termination of a process, the sig_chld function executes (the signal handler for SIGCHLD), wait function fetches the child's pid and termination status. The signal handler then returns.

But as the signal was caught by the parent while parent was blocked in a *slow system call (*accept*)*, the kernal causes the accept to return an error of EINTR (interrupted system call). The parent does not handle this error, so it aborts. This is a potential problem which all slow system call (read, write, open, select etc) face whenever they catch any signal. This is undesirable.

In case of Solaris 2.5, the *signal function* provided in the C library does not cause an interrupted system call to be automatically restarted by the kernal. That is SA_RESTART flag that is set in signal

function is not set by the signal function in the system library. Where as some other systems like, 4.4 BSD using the library version of signal function, the kernal restarts the interrupted system call and accept does not return an error.

Handling the Interrupted System Calls:

The basic rule that applies here is that when a process is blocked in a slow system call and the process catches a signal and the signal handler returns, the system call can return an error of EINTR. (Error interruption). For portability, when the programmes are written, one must be prepared for slow system calls to return EINTR. Even if some system supports the SA_RESTART flag, not all interrupted system calls may automatically be started. Most Berkeley-derived implementation never automatically restart.

To handle interrupted accept, we change the call to accept (in the server programme) in the following way:

```
for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if  ( ( connfd = accept (listenfd, (SA) & cliaddr, & clilen)) < 0) {
                if (errno == EINTR
                        continue;
                else
                        err_sys (" accept error");
                }
        }
```

IN this code, the interrupted system call is restarted. This method works for the functions read, write, select and open etc. But there is one function that cannot be restarted by itself. – connect. If this function returns EINTR, we cannot call it again, as doing so will return an immediate error. In this case we must call select to wait for the connection to complete.

Wait () and waitpid () functions:

pid_t wait(int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);

wait and waitpid both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (integer) is returned through statloc pointer. (Termination status are determined when three macros are called that examine the termination status and tell if the child terminated normally, was killed by a signal or is just the job control stopped. Additional macros tell more information on the reason.)

If there are no terminated children for the calling wait, but the process has one or more children that are still executing, then wait blocks until the first of the existing children terminate.

w***aitpid*** gives us more control over which process to wait for and whether or not to block. pid argument specify the process id that we want to wait for. a value of -1 says to wait for the first of our children to terminate. The option argument lets us specify additional options. The most common option is WNOHANG This tells the kernel not to block if there are no terminated children; it blocks only if there are children still executing.

The **wait** *pid* argument specifies a set of child processes for which status is requested. The waitpid() function shall only return the status of a child process from this set.

- If pid is equal to (pid_t) -1, status is requested for any child process. IN this respect, waitpid() is similar to wait().
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process groups.

Difference between wait and waitpid:

To understand the difference, TCP / IP client programme is modified as follows:

tcpcliserv/tcpcli04.c

```
1  #include     "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int     i, sockfd[5];
6      struct sockaddr_in servaddr;
7      if (argc != 2)
8          err_quit("usage: tcpcli <IPaddress>");
9      for (i = 0; i < 5; i++) {
10         sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);
11         bzero(&servaddr, sizeof(servaddr));
12         servaddr.sin_family = AF_INET;
13         servaddr.sin_port = htons(SERV_PORT);
14         Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15         Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
16     }
17     str_cli(stdin, sockfd[0]);   /* do it all */
18     exit(0);
19 }
```

tcpcliserv/tcpcli04.c

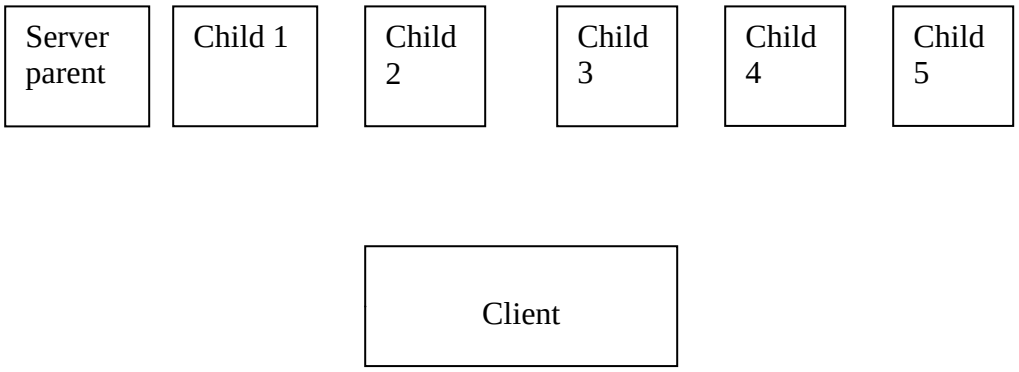
Figure 5.8 TCP client that establishes five connections with server.

The client establishes five connections with the server and then uses only the first one **(sockfd[0])** in the call to **str_cli**. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server as shown below:

When the client terminates, all open descriptors are closed automatically by the kernel and all the five serve children terminate at about the same time. This causes five SIGCHLD signals to be delivered to the parent at about the same time, which we show in Figure below:

It is this delivery of multiple occurrences of the same signal that causes the problem that we are talking about. By executing ps, one can see that other children still exist as zombies.

Establishing a signal handler and calling *wait* form the signal handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are not normally queued. Also this problem is non deterministic.



(If the client and server are on the same host, one child is terminated leaving four zombies. If we run the client and server on different hosts, the handler is executed twice once as a result of the first signal being generated and since the other four signals occur while the signal handling is executing, the handler is called once more This leaves three zombies. However depending on the timing of the FIN arriving at the server host, the signal handler is executed three or even four times).

The correct solution is to call waitpid instead of wait. Following code shows the server version of our *sig_ chld* function that handles SIGCHLD. Correctly. This version works because we call waitpid within the loop, fetching the status of any of our children that have terminated. We must specify the WNOHNG option; This tells waitpid not to block if there exists running children that not yet terminated. . In the code for wait , we cannot call wait in a loop, because there is no to prevent wait from blocking if there exits running children that have not yet terminated. Following version correctly handles a return EINTR from accept and it establishes a signal handler that called ***waitpid*** for all terminated children.

The following programme shows the implementation of waitpid(). The second part is the implementation of complete server programme incorporating the signal handler.

The three scenarios that we encounter with networking programme are :

We must catch SIGCHLD signal when forking child processes.

We must handle interrupted system calls when we catch signal.

A SIGCHLD handler must be coded correctly using waitpid to prevent any zombies form being left around

tcpcliserv/sigchldwaitpid.c

```
 1 #include    "unp.h"
 2 void
 3 sig_chld(int signo)
 4 {
 5     pid_t   pid;
 6     int     stat;

 7     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
 8         printf("child %d terminated\n", pid);
 9     return;
10 }
```

tcpcliserv/sigchldwaitpid.c

Figure 5.11 Final (correct) version of `sig_chld` function that calls `waitpid`.

tcpcliserv/tcpserv04.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;
 9     void    sig_chld(int);
10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(SERV_PORT);
15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16     Listen(listenfd, LISTENQ);
17     Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
18     for ( ; ; ) {
19         clilen = sizeof(cliaddr);
20         if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21             if (errno == EINTR)
22                 continue;       /* back to for() */
23             else
24                 err_sys("accept error");
25         }
26         if ( (childpid = Fork()) == 0) {    /* child process */
27             Close(listenfd);    /* close listening socket */
28             str_echo(connfd);   /* process the request */
29             exit(0);
30         }
31         Close(connfd);          /* parent closes connected socket */
32     }
33 }
```

tcpcliserv/tcpserv04.c

Figure 5.12 Final (correct) version of TCP server that handles an error of `EINTR` from `accept`.

Connection abort before *accept* returns :

Similar to interrupted system call, a non fatal error occurs when the client is reset after three handshake without transfer of any packet. The figure is shown below:

After the three way handshake, the connection is established and then the client TCP sends an RST (reset). ON the server side the connection is queued by its TCP waiting for the server process to call accept when the RST arrives. Some time later the server process calls accept.

What happens to aborted connection depends on the type of implementation . The Berkley derived implementation handles the aborted connection completely within the kernal and the server process never sees it. Most of the SVR4 (System V release 4) implementation returns an error to the process as the return from ***accept*** and the type of error depends on the implementation. Most implementation returns an ***errono*** EPROTO (protocol error) but posix .1g specifies that the return must be ECONNABORTED ("software caused connection abort"). The reason for this is that EPROTO is also returned when some fatal protocol related event occurs on the bytestream. Receiving the same error EPROTO by the server makes it difficult to decide whether to call ***accept*** again or not. IN case of ECONNABORTED error, the4 server ignores the error and just call accept again.

Termination of Server Process:

After starting client – server, the child process is killed at the server (kill the child process based on the its ID). This simulates the crashing of the server process, then what happens to client: The following steps take place.

1. We start the server and client on different hosts and type one line to the client to verify that all is OK. That line is echoed normally by the server child.
2. Identify the process ID of the server child and kill it. As part of the process termination, all open descriptors in the child are closed. This causes the FIN to by sent to the client and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The SIGCHLD signal is sent to the server parent and handled correctly.
4. Nothing happens at the client. The client receives the FIN from the sever TCP and responds with an ACK. But the problem is that the client process is blocked in the call to the ***fgets*** waiting for a line form the terminal.
5. When we type another line, str_cli calls written and the client TCP sends the data to the sever. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not send any more data. The receipt of FIN does not tell the client that the server process has terminated (which in this case it has).
6. When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST is sent by watching the packets with ***tcpdump***
7. But the client process will not see the RST because it calls ***readline*** immediately a after the call to ***writen*** and ***readline*** returns 0 (end of line) immediately because of the FIN that was received in

step 2. Our client is not expecting to receive an end of line at this point so it quits with an error message server terminated prematurely."

8. So when the client terminates (by calling err_quit), all its open descriptors are closed.

The problem in this example is that the client blocked in the call to fgets when the FIN arrives on the socket. The client is really working with the two descriptors - the socket and the user input - and instead of blocking on input from any one of the two sources, it should block on input from either source. This is the function of ***select*** and ***poll*** function .

SIGPIPE signal.:

When the client has more than one to write, what happens to it? That is when the FIN is received, the readline returns RST, but the second one is also written. The rule applied here is, when a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process. The default action of this signal is to terminate the process so the process must catch the signal to avoid involuntarily terminated.

If the process catches the signal and returns from the signal handler, or ignores the signal, the write operation returns EPIPE (error pipe)

```
        #include "unp.h"
void    str_cli (FILE *fp, int sockfd)
        {
                char sendline[MAXLINE], recvline[MAXLINE];
        while (fgets(sendline, MAXLINE, fp)!=null)
        {
                writen(sockfd, sendline, 1);
                sleep(1);
                        writen (sockfd, sendline +1, strlen (sendline)-1);
                if (readline()sockfd, recvline, MAXLINE==0)
                        err_quit ("str_cli: server terminated prematurely");
                fputs (recvline, stdout);
        }
}
```

in the above str_cli(), the writen is called two times: the first time the first byte of data is writen to the socket, followed by a pause of 1 sec, followed by the remainder of the line. The intention is for the first writen to elicit the RST and then for the second writen to generate SIGPIPE.

We start with the client, type in one line, see that line is echoed correctly, and then terminates the server child on the server host, we then type another line, but nothing is echoed and we just get a shell Prompt. Since the default action of the SIGPIPE is to terminate the process without generating a core file, nothing is printed by the Kornshell.

The recommended way to handle SIGPIPE depends on what the application what to do when this occurs. IF there is nothing special to do, then setting the signal disposition to SIG_IGN is easy, assuming that subsequent output operations will catch the error of EPIPE and terminate.

IF special actions are needed when the signal occurs, then the signal should be caught and any desired actions can be performed in the signal handler.

If multiple sockets are in use, the delivery of the signal does not tell us which socket encountered the error. IF we need to know which write caused the error, then we must either ignore the signal or return from the signal handler and handle **EPIPE** from ***write***

Crashing of Server Host:

Next scenario is to see what happens when the server host crashes. To simulate this we must run the client and server on different hosts. We then start server, start the client, type in a line to the client to verify

that the connection is up, disconnect the server host from the network, and type in another line at the client. This also covers the scenario of the server host being unreachable when the client sends data (some immediate router is down after the connection has been established).

The following steps take place.

1. When the server host crashes, nothing is sent out on the existing network connections. That is we are assuming the host crashes, and is not shut down by the operator
2. We type a line of input to the client, it is written by writen and is sent by the client TCP as a data segment. The client then blocks in the call to readline waiting for the echoed reply.
3. If we watch the network with tcpdump, we will see the client TCP continually retransmit the data segment, trying to receive ACK from the server. Berkley derived implementations transmit the date segments 12 times, waiting around 9 minutes before giving up. When the client finally gives up, an error is returned to the client process. Since the client is blocked in the call to readline, it returns an error. Assuming the server host had crashed and there were no responses at all to the client's data segments, the error is ETIMEDOUT. But if some intermediate router determine that the server was unreachable and responded with ICMP destination unreachable message, then error is either EHOSTUNREACH or ENETUNREACH.

   To detect that the server is unreachable even before 9 minutes, place a time out call to readline.
   To find the crash of server even if client is not sending data actively, another technique is used which used SO_KEEPALIVE socket option

Shutdown of Server

When a Unix system is shutdown, the init process normally sends the SIGTERM signal to all processes (this signal can be caught), waits some fixed amount of time (often between 5 and 20 seconds), and then sends SIGKILL signal (which we cannot catch) to any process still running. This gives all running processes a short amount of time to clean up and terminate. If we do not catch SIGTERM and terminate, our server will be terminated by SIGKILL signal. When the process terminates, all the open descriptors are closed, and we then follow the same sequence of steps discussed under "termination of server process". We need to select the select or poll function in the client to have the client detect the termination of the server process as soon it occurs.

I / O MULTIPLEXING : THE *select* AND *poll* FUNCTIONS

It is seen that the TCP client is handling two inputs at the same time: **standard input and a TCP socket.** It was found that when the client was blocked in a call to read(by calling readline function), and the server process was killed. The server TCP correctly, correctly sends a FIN to the client TCP, but since the client process is blocked reading from the standard input, it never sees the end – of file until it reads from the socket . What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e. input is ready to be read, or the descriptors is capable of taking more outputs). This capability is called I /O Multiplexing and is provided by the ***select*** and ***poll*** functions . There is one more Posix .1g variations called ***pselect.***

I /O multiplexing is typically is used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario that was described in the previous paragraph.
- It is possible, but rare, ofr a client to handle multiple sockets at the same time. We show an example of this using ***select*** in the context of web client
- If a TCP server handles both a listening socket and its connected sockets, I / O multiplexing is normally used.
- IF a server handles both TCP and UDP, I/O multiplexing is normally used.
- If a server handles multiple services and perhaps multiple protocols, I/O multiplexing us normally used.

It is not restricted only to networking programme, it may be used in any nontrivial application as well.

I/O Models:

There are five I /O models in the Unix. These are:

a. Blocking I /O
b. Non blocking I / O
c. I/O Multiplexing (select and poll)
d. Signal driven I/O (SIGIO)
e. Asynchronous I/O (the Posix 1 aio_functions)

There are two distinct phases for an input operation.:

a. waiting for the data to be read and
b. copying the data from the kernel to the process.

Fo ran input operation on a socket the first step normally involves waiting for the data to arrive on the network. When the packet arrives, it is copied into buffer within the kernel. The second step is copying this data from the kernel's buffer into our applications buffer.

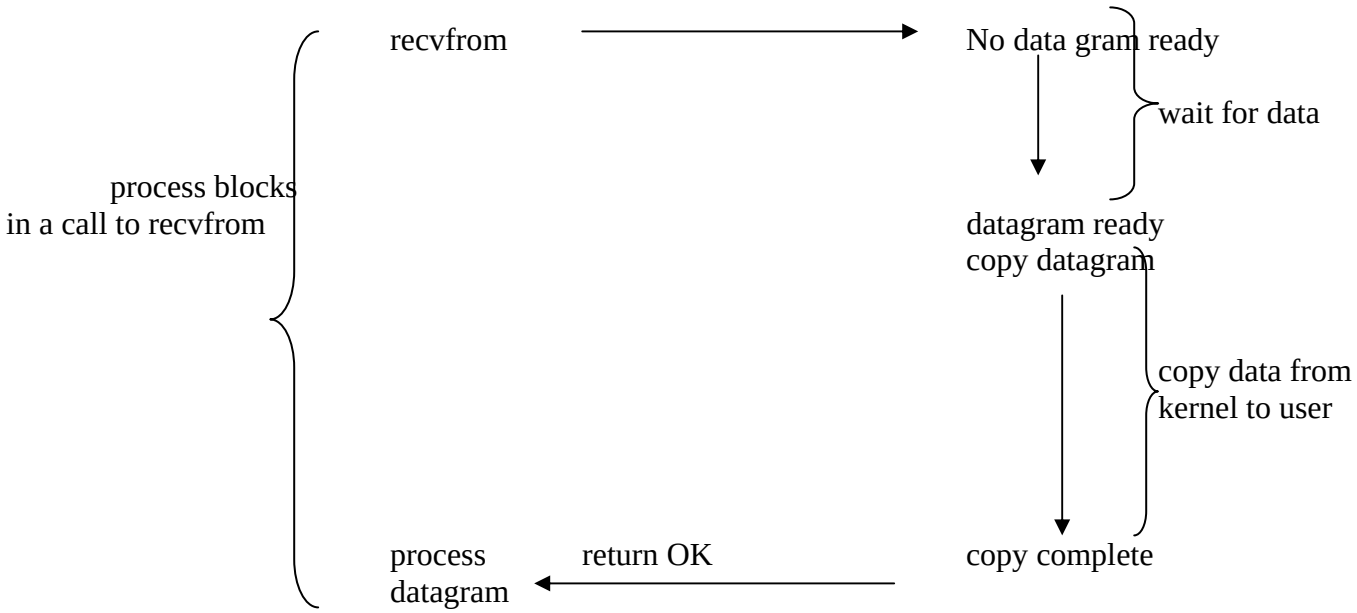
Blocking I/O Model :

The most prevalent model for I/O is the blocking I/O model, which we have used for all our examples so far in the text.. BY default, all sockets are blocking. Using a datagram socket for our examples we have the scenario as shown below. In UDP the concept of data being ready to be read is simple because either an entire datagram packet is received or not.

IN this example ***recvfrom*** as a system call as it differentiated between our application and the kernel.

The process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call

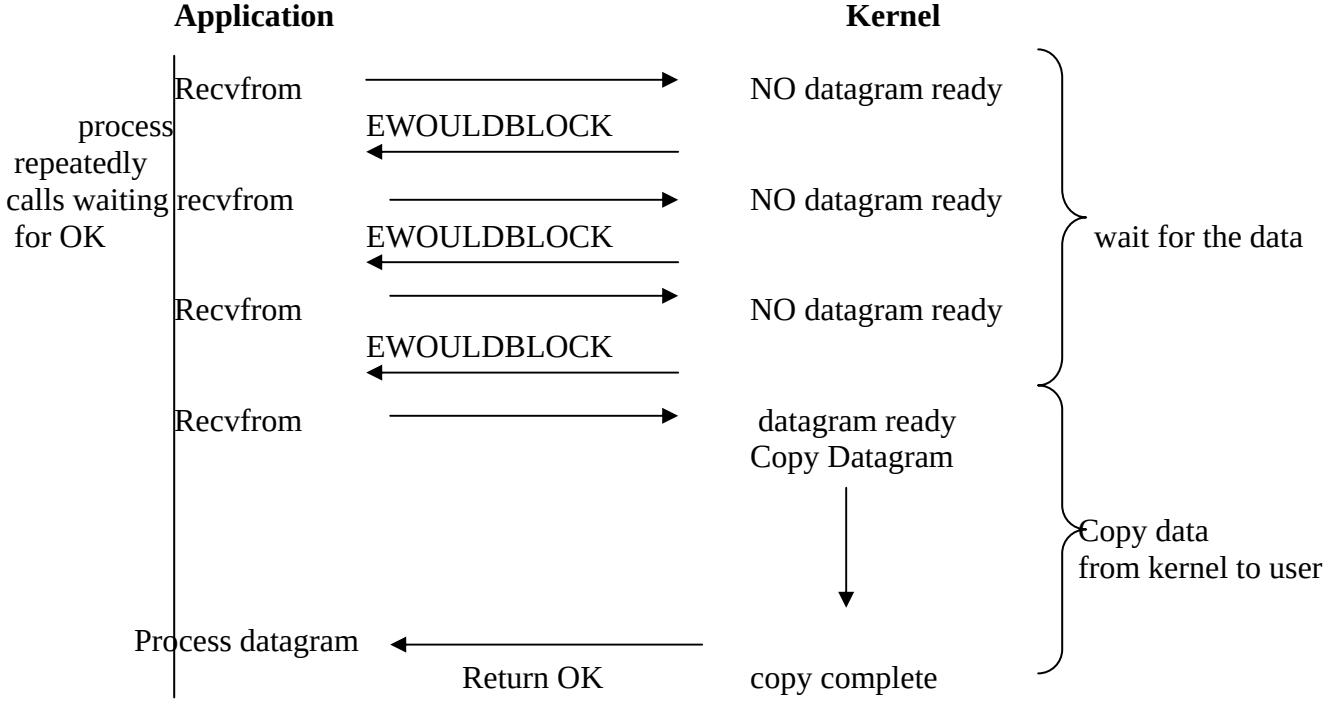
being interrupted by a signal. We say that our process is blocked the entire time from when it call recvfrom until it returns. When recvfrom returns OK, our application processes the datagram.



Blocking I/O Model.

Non Blocking I/O Model :

When the socket is set to non blocking, the kernel is told that "when I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep but return an error message instead." The following figure gives the details.



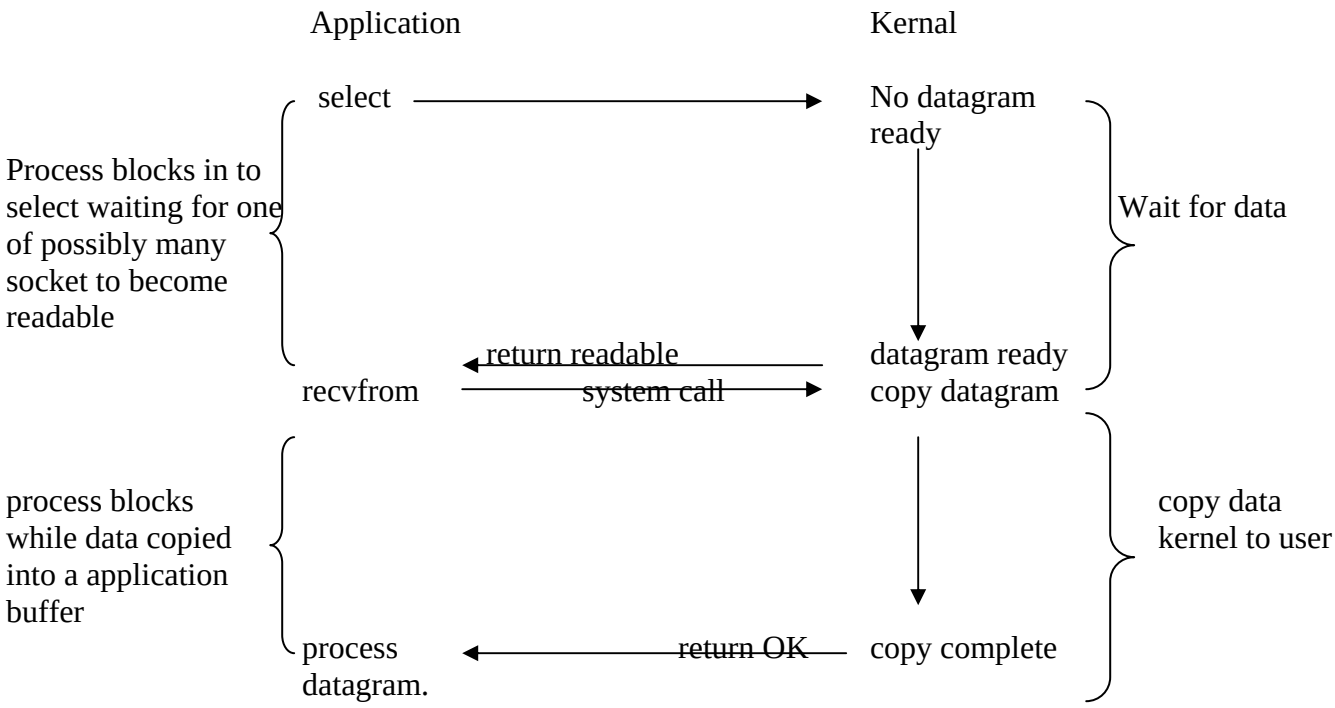
Non Blocking Model

During the first three times, when the ***recvfrom*** is called, there is no data to return, so the kernel immediately returns an error **EWOULDBLOCK.** Fourth time, when ***recvfrom*** is called, the datagram is ready, it is copied into our application buffer and the ***recvfrom*** returns OK. The application then process the data.

When the application puts the call ***recvfrom*** in a loop, on a non blocking descriptors like this, it is called polling. The continuation polling of the kernel is waste of CPU time. But this model is normally encountered on system that are dedicated to one function.

I / O Multiplexing Model :

IN this ***select*** or ***poll*** functions are called instead of ***recvfrom.*** The summary of I/O Multiplexing call is given in the following figure:



I / O Multiplexing Model

IN this the call to select is blocked waiting for the datagram socket to be readable. When select returns that the socket is readable, then the recvfrom called to copy the datagram into the application.

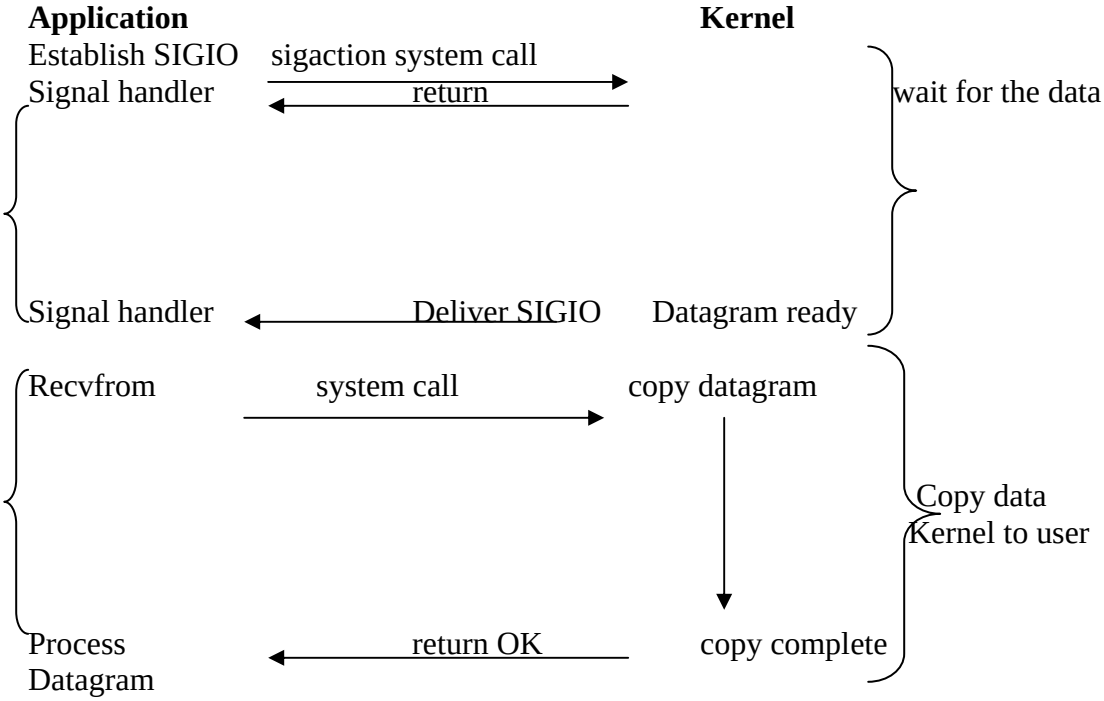
Compared to blocking model, the difference here is that the application can wait for more than one descriptors to be ready. The disadvantages is that select requires two system calls.

Signal Driven I/O Model

Signals are used to tell the kernel to notify applications with the **SIGIO** signal when the descriptor is ready. It is called **signal driven I/O** Model. The summary is shown in the following figure.

First enable the socket for the signal driven I/O and install a signal handler using the ***sigaction*** system call. The return from this system call is immediate and our process continuous, it is not block. When the datagram is ready to be ready, the SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be process, or we can notify the main lop and let it read the datagram.

The advantage I this model is that we are not blocked while waiting for the datagram top arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the dta is read to process or that the datagram is ready to be read.

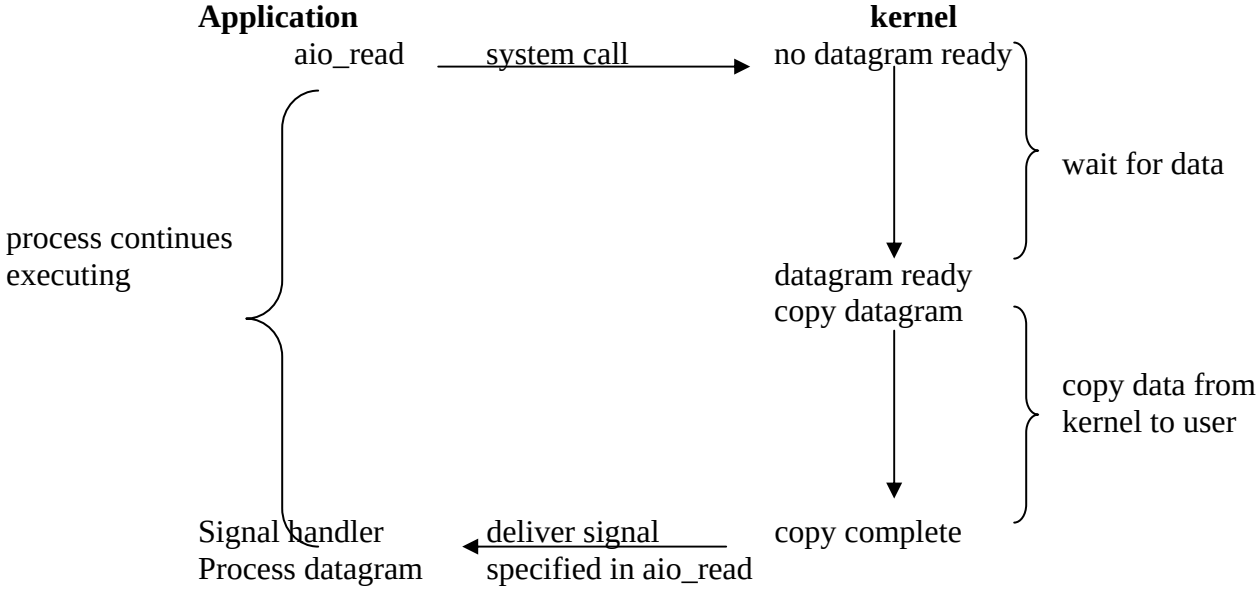


Signal Driven I/O Model

Asynchronous I/O Model

Asynchronous I/O are new with the 1993 edition of Posix 1g. In this the kernel is told to start operation and to notify when the entire operation (including the copy of the data from the kernel to our buffer) is over.

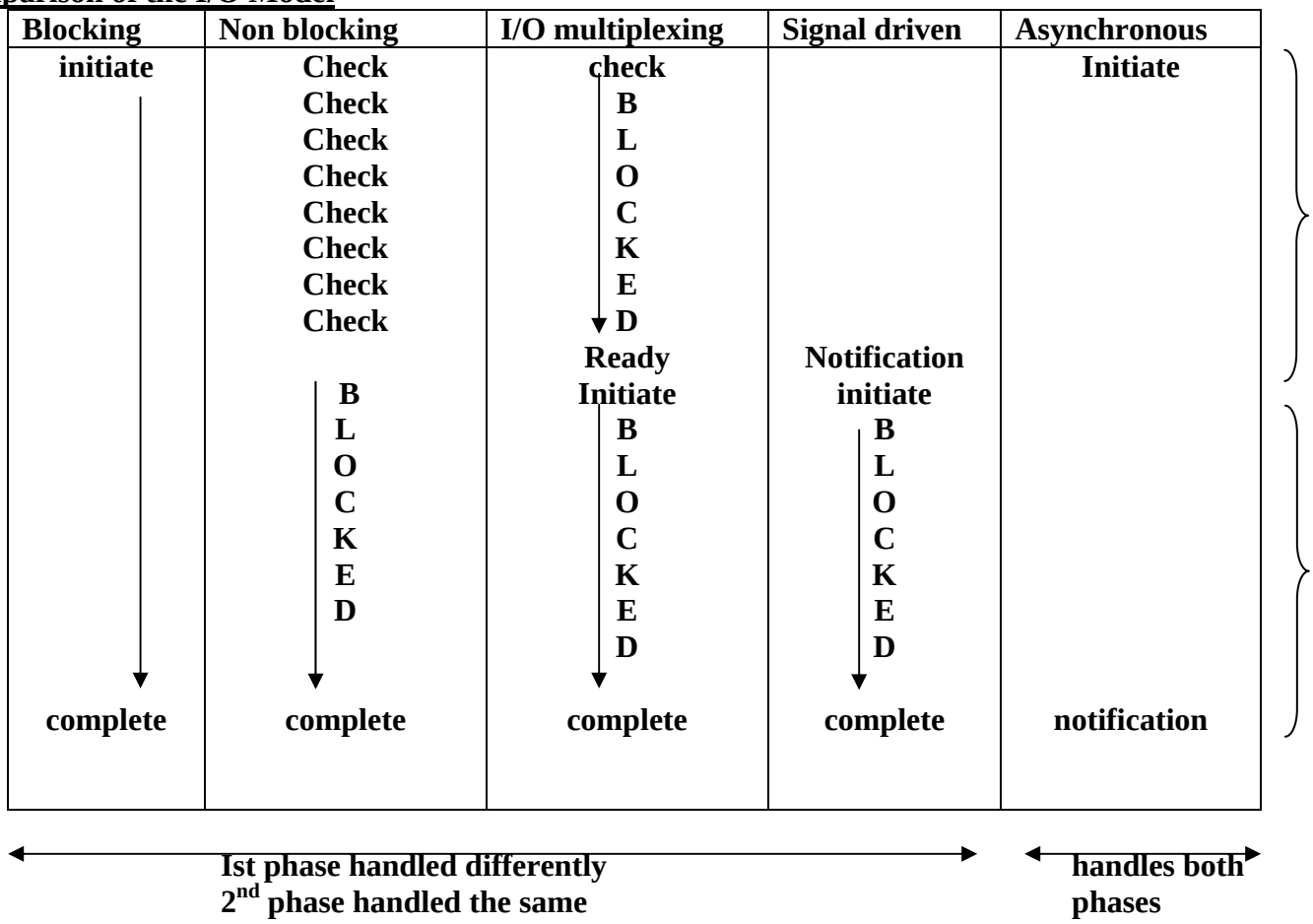
The main difference between this and the signal driven I/O model in the previous section is that with the signal driven I/O the kernel tells when the I/O operation can be initiated. But with asynchronous I/O, the kernel tells us when an I/O operation is complete. It is summarized as given below:



Asynchronous I/O Model

The function ***aio_read*** is called and the descriptor, buffer pointer, buffer size, file offset and how to notify when the entire operation is complete are passed to the kernel. The system calls returns immediately and our process is not blocked waiting for the I./O to complete. It is expected that the kernel will generate some signal when the operation is complete. The signal is generated only when the data is copied completely in the application buffer.

Comparison of the I/O Model



The main difference between the four models is the first phase as the second phase in the first four models is the same. The process is blocked in a cal to ***recvfrom*** while the data is copied from the kernel to the caller's buffer. Asynchronous I/O however, handles both the phases and is different form the first four.

Synchronous vs asynchronous:

A synchronous I/O operation causes the requesting process to be blocked until that I?O operation is completes.

An asynchronous I/O operation does not cause the requesting process to be blocked.

Based on this definition, the first four are synchronous as the actual I/O operation blocks the process. Only asynchronous I/O model matches the asynchronous I/O definition.

Select Function :

The select function allows the process to instruct the kernel to wait for any one of the multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call select and tell the kernel to return only

- any of the descriptors in the set { 1,4,5} are ready for reading, or
- any of the descriptors in the set {2,7} are ready for writing, or

- any of the descriptors in the set {1,4 } have an exception condition pending or
- after 10.2. seconds have elapsed.

That the kernel is told in what descriptors we are interested in (for reading, writing or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets: any descriptor can be tested using select.

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

Consider the final arguments: This tells the kernel how long to wait for one of the specified descriptors to become ready. A **timeval** structure specifies the number of seconds and microseconds**.**

```
Struct timeval {
        long    tv_sec;   /* seconds */
        long    tv_usec;  /* micros seconds */
        }
```

There are three possibilities:

- **wait forever**: return only when one of the specified descriptors is ready for I/O . For this, we specify the timeout argument as a **null pointer**.
- **Wait upto a fixed amount of time**: return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the **timeval** structure pointer to by the **timeout** argument.
- **Do not wait at all**: return immediately after checking the descriptors. This is called **polling**. To specify this, the timeout argument must point to a timeval structure and the timer value must be **zero**

In case of first two scenario, the wait is normally interrupted if the process catches a signal and returns from the signal handler.

The const qualifier on the timeout argument means it is not modified by select on return. For example, if we specify a time limit of 10 sec, and select returns before the timer expires, with one or more descriptors ready or with an error of EINTR, the timeval structure is not updated with the number of seconds remaining when the functions returns.

The three middle arguments readset, writeset and excepset specify the descriptors that we want the kernel to **test** for reading, writing and exception conditions.

The two exceptions conditions currently supported are:

- The arrival of out of bound data for a socket.
- The control status information to be read from the master side of pseudo terminal

The ***maxfdp1*** argument specifies the number of descriptors to be tested. Its value is the maximum descriptors to be tested plus one. The descriptors 0,1,2, through and including ***maxfdp1*** –1 are tested. The constant FD_SETSIZE defined by including <sys /select.h>, is the number of descriptors in the fd_set data type. It s value is often 1024, but few programs use that many descriptors. The ***maxfdp1*** argument forces us to calculate the largest descriptors that we are interested in and then tell the kernel this value. For 1,4,5 the ***maxfdp1*** is 6.

The design problem is how to specify one or more descriptors values for each of these three arguments. **Select** uses **descriptor sets**, typically an array of integers – a32 by 32 array- of 1024 descriptors

set. The descriptors are programmed for initializing /reading / writing and for closing using the following macros:

```
void FD_ZERO (fd_set   * fset)          /* clear all bits in fset */
void FD_SET (int fd,  fd_set   * fset)  /* Turn on the bit for fd in  fset*/
void FD_CLR (int fd,  fd_set   * fset)  /*turn off the bit for fd in fset
void FD_ISSET (int fd,  fd_set   * fset) /* is the bit for fd on in fset? */
```

For example to define a variable of type fd_set and then turn on the bits for descriptors, we write

```
fd_set  rset;
FD_ZERO (&rset);   /* initialize  the set; all bits to zet */
FD_SET (1, &rset);  /* turn on bit for fd 1
FD_SET (4, &rset );  /* turn on  bit for fd 4; */
```

If the descriptors are initialized to zero, unexpected results are likely to come.

Middle arguments to select, readset, writeset, or exceptset can be specified as null pointer, if we are not interest in that condition. The *maxfdp1* arguments specifies the number of descriptors to be tested. Its value is the maximum descriptors to be tested, plus one (hence the name maxfdp1). The descriptors 0,1,2 up through and including *maxfdp1 – 1* are tested.

Select function modifies the descriptor sets pointed by the readset, writset and excepset pointers. These three arguments are value-result arguments. When we call the function, we specify the values of the descriptors that we are interested in and on return the result indicates which descriptors are ready. We use FD_ISSET macros return to test a specific descriptor in an fd_set structure. Any descriptors that is not ready on return will have its corresponding bit cleared in the descriptors set.

str_cli Function :

The ***str_cli*** Function is rewritten using the select function as shown below:

```
# include "unp.h"
 void str_cli (FILE *fp, int socket)
        {
        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))
                        {       if (readline (sockfd, recvline, MAXLINE)==0)
                                err_quit ("str_cli: server terminated prematurely");
                        fputs (recvline, stdout);
                        }
                if (FD_ISSET(fileno(fp), &rset))
                        {if (fgets (sendline,  MAXLINE, fp)==NULL)
                        return;
                        writen(sockfd, sendline, strlen (sendline));
                        }
                }
        }
```

IN the above code the descriptors set is initialized to zero using FD_ZERO. Then, the descriptors, file pointer ***fp*** and socket ***sockfd*** are turned on using FD_SET. maxfdp1 is calculated from the descriptors list. **Select** function is called. IN this writeset pointer and exception set pointers are both NULL. Final time pointer is also NULL as the call is to be blocked until something is ready.

If on return from the select, socket is readable, the echoed line is read with ***readline*** and output by ***fputs.***

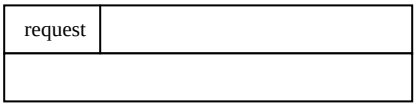
If the standard input is readable, a line is read by ***fget*** sand written to the sockets using ***writen.***
IN this although the four functions fgets, writen, readline and fputs are used, the order of flow within the function has changed. In this, instead of flow being driven by the call to fgets, it is driven by the call to select.

This has enhanced the robustness of the client.

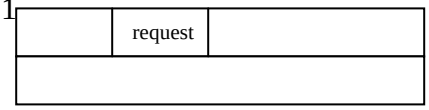
Batch Input:

The echo client server, works in a stop and wait mode. That is , it sends a line to the server and then waits for the reply. This amount of time is one RTT (Round Trip Time) plus the server's processing time. If we consider the network between the client and the server as a full duplex pipe with requests from the client to server, and replies in the reverse direction, then the following shows the stop and wait mode.

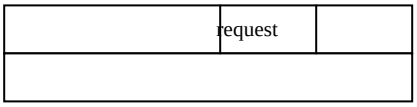
Time 0



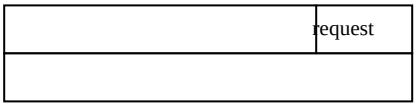
time1



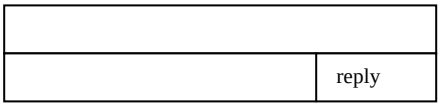
Time 2



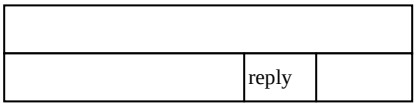
time 3



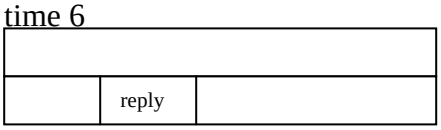
Time 4



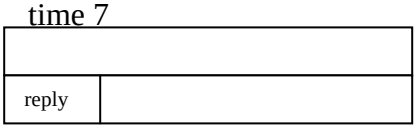
time 5



time 6



time 7



The request is sent by the client at time 0 and we assume RTT of 8 units of time. The reply sent at a time 4 is received at time 7. It is assumed that there is no serving processing time and that the size of the request is the same as the reply. Also, TCP acknowledgment are ignored.

But as there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full duplex, in this example we are only using one- eighth of the pipe capacity. This stop and wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, we can easily run our client in a batch mode. When the input is redirected as in client server example, the output file is always same.

To see what is happening in the batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7 as shown below:

Time 7

Request8	Request7	Request6	Request5
Reply 1	Reply 2	Reply 3	Reply 4

Time 8

Request9	Request8	Request7	Request6
Reply 2	Reply 3	Reply 4	Reply 5

Filling the pipe between the client and the sever : batch Mode;

Now to understand the problem with the **str_cli,** let the input file contains only nine lines. The last line is sent at time 8 as shown above. But we cannot close the connection after writing this request, because there are still other requests and rep0les in the pipe. The cause of the problem is our handling of end of file on input. The function returns to the main function, which then terminates. But in a batch mode, an end of file on the input does not imply. That we have finished reading from the socket. There might still requests on the way to the server or replies on the way to back from the server.

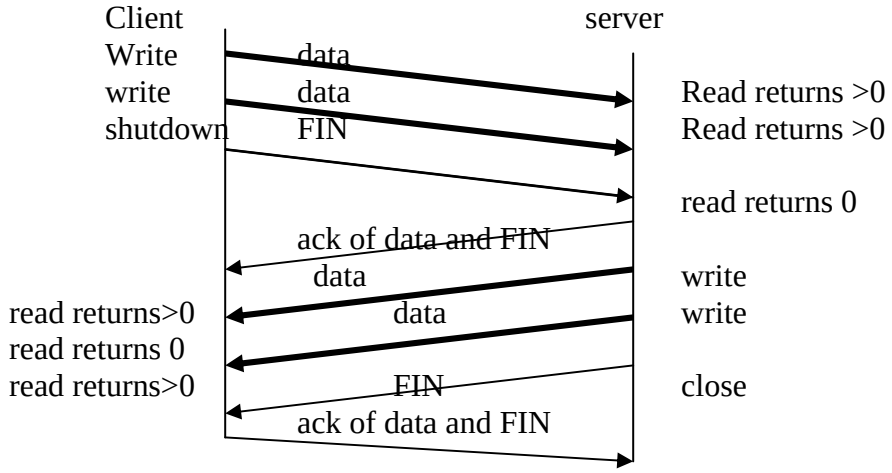
What we need is a way to close one half of the TCP connection. That is we want to send FIN to the server, telling it we have finished sending data, but leave the socket descriptors for reading. This is done with the ***shutdown*** function.

shutdown function.

The normal way to terminate a network function is to call the close function. But there are two limitations with close that can be avoided with the ***shutdown*** function.

1. Close decrements the descriptors' reference count and closes the socket only if the count reaches 0. With ***shutdown*** function, we can initiate TCP's normal connection termination sequence regardless of the reference count.

2. Close terminates both directions of data transfer, reading and writing. Since a TCP connections is full duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function. Following figure shows the typical scenario



Syntax of shutdown () function

int shutdown (int sockfd, int howto);

Returns : 0 if OK, -1 on error.

The action of the function depends on the value of the ***howto*** argument.

SHUT_RD – The read half of the connection is closed. NO more data can be received on the socket and any data currently in the socket receive buffer is discarded. He process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP sockets is acknowledged and then silently dropped.

SHUT_WR : The write half of the connection is closed. IN the case of TCP, this is called half close. Any data currently in the socket

SHUT_RW: Read half and write half of the connections are both closed.

The following code gives the str_cli function that is modified by using shutdown function:

```
# include "unp.h"
 void str_cli (FILE *fp, int socket)
        {
        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);
                if (FD_ISSET(sockfd, &rset))
                        {       if (readline (sockfd, recvline, MAXLINE)==0)
                                { if ( stdineof ==0 )
                                        return;
                                else
                                        err_quit ("str_cli: server terminated prematurely");
                                }
                        fputs (recvline, stdout);
                        }
                if (FD_ISSET(fileno(fp), &rset))
                        { if (fgets (sendline,  MAXLINE, fp)==NULL)
                                {stdineof = 1;
                                shetdown (sockfd, SHUT_WR);
                                FD_CLR (fileno(fp), &rset);
                                continue;
                                }
                        writen(sockfd, sendline, strlen (sendline));
                        }
                }
        }
```

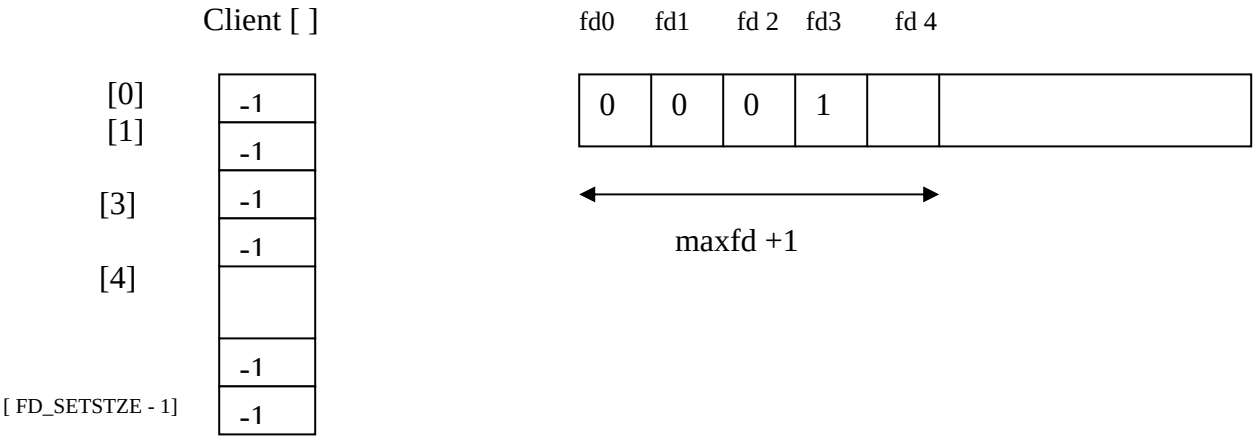
IN this ***stdineof*** in is a new flag that is initialized to 0. As long as this flag is 0, each time around the main loop we **select** on standard input for readability.

When we read the end of file on the socket, if we have already encountered an of file on standard input, this is the normal termination and the function returns. But if the end of file is not encountered on the standard input, the server process is prematurely terminated/

When we encounter the end of file on standard input, the new flag is set to 1 and we call shutdown with a second argument of SHUT_WR to send the FIN.

TCP Echo Server with IO multiplexing:

The TCP echo server can use the select function to handle any number clients, instead of forking one child per client. This needs to create two sets of data structures. These are : read descriptors set and connected socket descriptors set as shown below:



The server maintains only read descriptors set, which is shown above. When the server is started in the foreground, the descriptors 0, 1 and 2 are set to standard input, output and error. Therefore the available descriptors for the listening socket is 3.

Another client descriptors set is also shown. It is shows the number of the connected socket descriptors for each client. All elements of this array are initialized to –1.

The only non zero entry in the descriptors set is the entry of listening socket and the first argument to select will be 4.

When the first client establishes a connection with our server, the listening descriptors becomes readable and the server calls accept. The new connected descriptor returned by accept will be given, as per the assumption . From now on the server must remember the new connected socket in its client array and the connected socket must be added to the descriptor set. The updated data set is shown below.

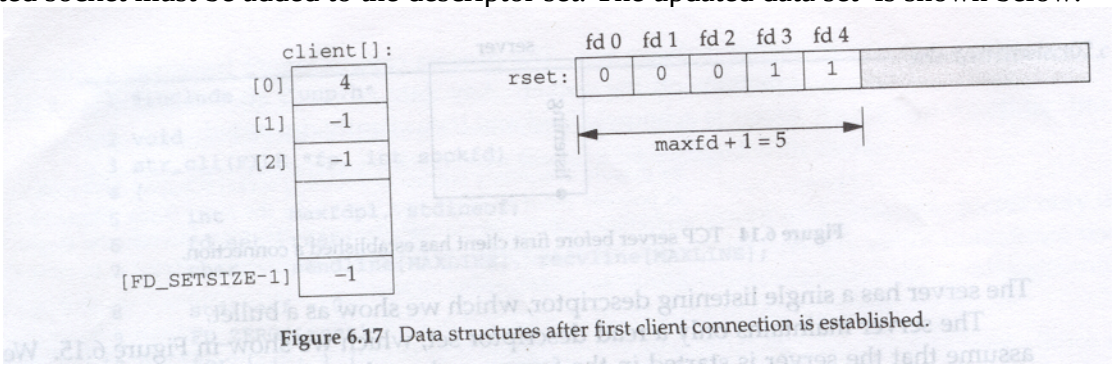


Figure 6.17 Data structures after first client connection is established.

Some time later a second client establishes a connection and we have the scenario as shown below:

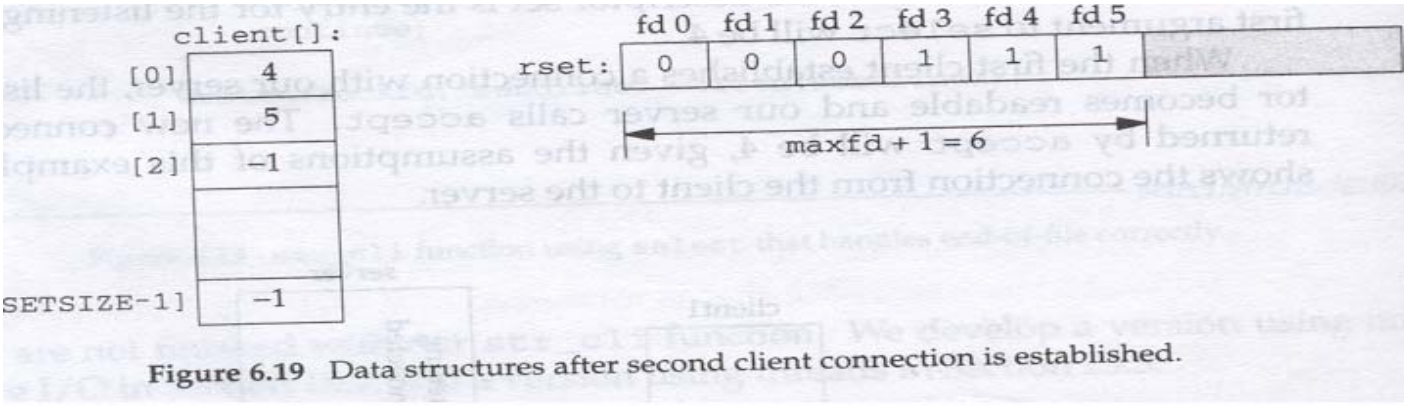


Figure 6.19 Data structures after second client connection is established.

The new connected socket which we assume as 5, must be remembered, giving the data structure as shown above.

When the first client terminates its connection, the client TCP sends FIN, makes descriptors in the server readable. When our server reads this connected socket, readline returns 0. We then close this colet and update our data structure accordingly. The value of the client[0] is set –1. and descriptors 4 in the descriptors is set to 0. The value of mxfd does not change.

That is as the client arrive we record their connected socket descriptor in the first available entry in the client array () the first entry with a value of –1. We must also add the connected socket to the descripto set. The variable maxi is the highest index in the current value of the first argument select. The only limit on the number of descriptors allowed for this process by the kernel. The first half server side programme is given below: