Lecture 8: TCP Sockets

(11/04/2009)

<u>Lecture Outline</u>

1. Overview

(1) Time line and functions involved in a TCP communication session: Fig. 4.1, p.96.

(2) *socket* function

a. Purpose:

- ∗ Specify the type of protocol desired (Internet TCP, Internet UDP, XNS SPP etc.)
- ∗ Reserve a socket descriptor, similar to file descriptor.

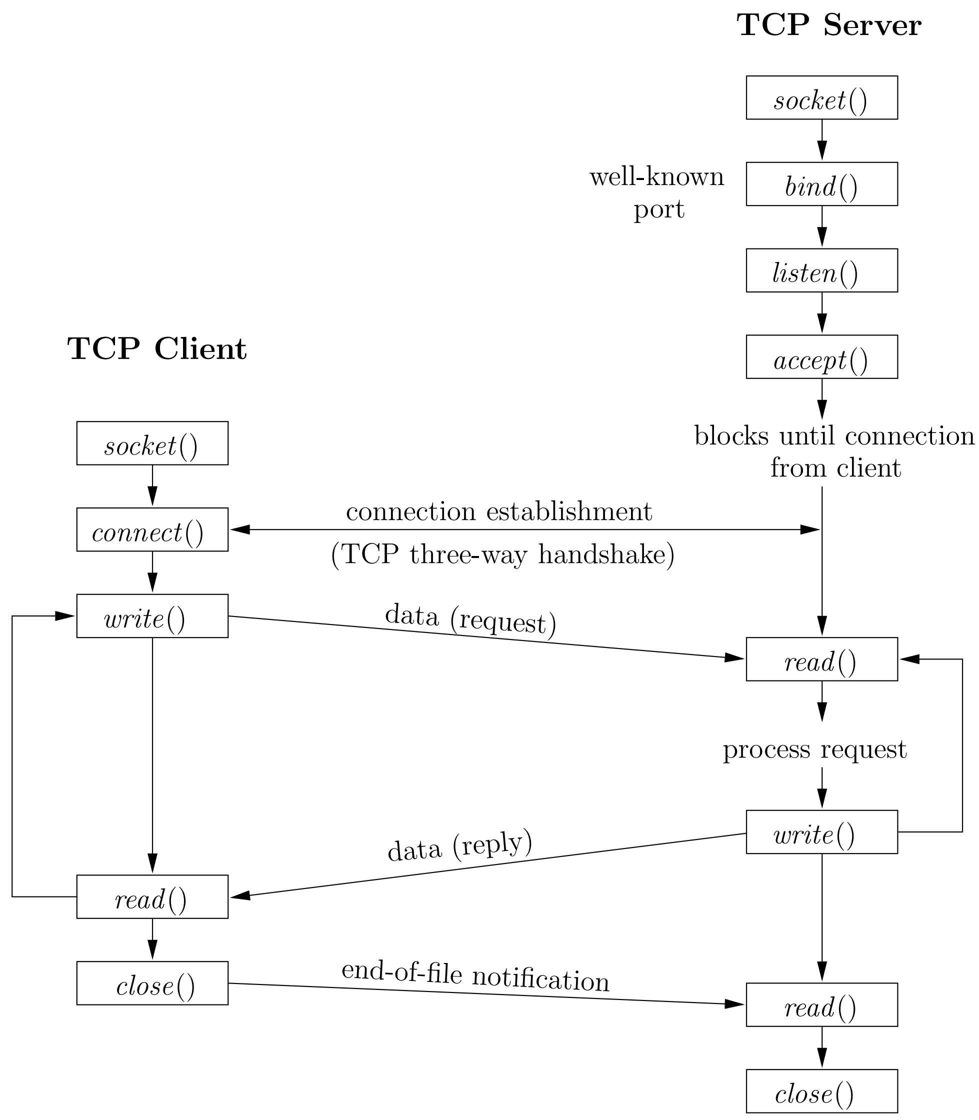

Figure 78: Socket functions for elementary TCP client-server. (Fig.4.1,p.96)

b. Syntax:

> #include <sys/types.h>
> #include <sys/socket.h>
> int socket(int *family*, int *type*, int *protocol*);

c. The *family* is one of (Fig.4.2, p.97)

family	*Description*
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key sockets (Chapter 19)

Figure 4.2 Protocol *family* constants for *socket* function

AF stands for "address family". Another prefix is PF_, which stands for "protocol family". They can be used interchangeably.

d. The *type* is one of (Fig.4.3, p.97)

type	*Description*
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Figure 4.3 *type* of socket for *socket* function

BSD provides also a raw socket interface to the IMP. An IMP is an *interface message processor*, providing intelligent packet switching service.

e. The *protocol* argument should be set to the specific protocol type as shown in Fig.4.4, or 0 to select the system's default that is determined by the combination of *family* and *type*.

Protocol	*Description*
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Figure 4.4 *protocol* of sockets for `AF_INET` or `AF_INET6`

f. Not all *family* and *type* combinations are valid (Fig. 4.5, p.97).

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP|SCTP	TCP|SCTP	yes		
SOCK_DGRAM	UDP	UDP	yes		
SOCK_SEQPACKET	SCTP	SCTP	yes		
SOCK_RAW	IPv4	IPv6		yes	yes

Figure 4.5 Combination of *family* and *type* for the *socket* function

(3) *connect* function

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

a. Returns 0 if successful, -1 otherwise.

b. For a connection-oriented protocol like TCP, this function initiates the three-way handshake process.

   (a) This function will return only when the connection is established or an error occurs.

   (b) There are several possible error cases:

      · If the client TCP doesn't receive response to its SYN segment, an ETIMEDOUT error is returned (This is a soft error and is returned to the TCP, not the client.). The action from TCP then is implementation dependent. For 4.4BSD, its TCP will send second SYN 6 seconds later, another 24 seconds later, a final SYN another 48 seconds later. That gives a total 78 seconds (There are two timers initiated, with the first timer set to 75 seconds, the second is a retransmission timer set as 6 seconds). After that an (hard) error is returned.

      · If the response from the server to client's SYN segment is RST, that normally means there is no corresponding service at the server end. This is a *hard error* and the error ECONNREFUSED is returned to the client as soon as the RST is received.

      · RST is a type of TCP segment (shown in TCP header) that is sent by TCP (both client and server can send) to indicate something is wrong. Three conditions that generate RST: when a SYN arrives for a port that has not listening server, when TCP wants to abort an existing connection, and when TCP receives a segment for a connection that does not exit.

      · If the client's SYN elicits an ICMP *destination unreachable* from some intermediate router, that is considered as a *soft error*. The TCP will saves the message but keep sending the SYNs with the same time between each SYN. If no response is received after some amount time, TCP will return the saved message as an `EHOSTUNREACH` or `ENETUNREACH` error.

c. The client does not have to *bind* (the function that will be introduced next) a local address before calling *connect*. The connection typically causes the following four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*, *foreign-addr*, *foreign-process*.

d. A connectionless client can also use *connect* call, with different functions.

   (a) The call in this case only stores the *servaddr* specified, so that the system knows where to send future data that the process writes to the *sockfd* descriptor.

   (b) Also only datagrams from this address will be received by the socket.

   (c) In this case the call returns immediately and there is no actual exchange of messages.

   (d) The advantage of using this call for a connectionless client that we don't need to specify the destination address for every datagram that we send. We can use *read*, *write, recv*, and *send* functions.

   (e) Another feature for connectionless clients: if the datagram protocol supports notification for invalid addresses, then the implementation can inform the user process if it sends a datagram to an invalid address (BSD4.2 has "connection refuses" messages).

e. NOTE: for both the *bind* and *connect* calls, no need for the protocol as an argument. In fact the protocol is available from two places: the AF_xxx is always in the first two bytes of the socket address structure. They both require a socket descriptor, which is associated with a single protocol family.

f. Using the daytimetcpcli to demonstrate various errors: p.100,101.

(4) *bind* function

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

The second argument is a pointer to a protocol-specific address and the third argument is the size of the address structure. Three uses of *bind*.

a. Returns 0 if successful, -1 otherwise.

b. Server register their well-known address with the system. It tells the system that "this is my address and any messages received for this address are to be given to me". Both connection-oriented and connectionless servers need to do this before accepting client requests.

c. A client can register a specific address for itself.

d. A server or client can uses a *wildcard* (the constant *INADDR_ANY*, to be seen used in Chapter 5) when specifying the IP address field of the socket structure to be used with *bind*. It can also specify a value zero for the port number field in the socket structure. Fig.4.6 (p.102) summarizes the rules when the kernel executing a *bind* function:

Process specifies		Result
IP address	Port	
wildcard	0	kernel chooses IP address and port
wildcard	nonzero	kernel chooses IP address, processes specifies port
local IP address	0	process specifies IP address, kernel chooses port
local IP address	nonzero	process specifies IP address and port

Figure 4.6 Result when specifying IP address and/or port number to *bind*

e. A connectionless client needs to assure that the system assigns it some unique address, so the other end has a valid return address to send its replies.

f. The *protocol* argument is set to 0 for most applications.

g. This call fills in the *local-addr* and *local-process* elements of the association 5-tuple.

(5) *listen* function

int listen(int *sockfd*, int *backlog*);

a. Returns 0 if successful, -1 otherwise.

b. Used by a connection-oriented server to indicate that it is willing to receive connections. Usually executed after both the *socket* and *bind* functions are invoked.

c. Two queues maintained by kernel for a listening socket (p.105, Fig.4.7):

   (a) An *incomplete queue* – contains an entry for each SYN that has arrived from a client. The server has not completed the three-way handshake process. The socket is in the SYN_RCVD state (Fig.2.4).

   (b) An *completed connection queue* – contains an entry for each client for which the server has completed the three-way handshake process. The socket is in the ESTABLISHED state.

d. In the time that it takes for a server to handle a request of an *accept*, it is possible for additional connection requests to arrive. All of them are put into the incomplete queue. A return from the accept function will remove an entry from the incomplete queue and put it at the end of the completed queue (Fig.4-8,p.105).

e. The second argument *backlog* specifies the sum of sizes of the two queues.

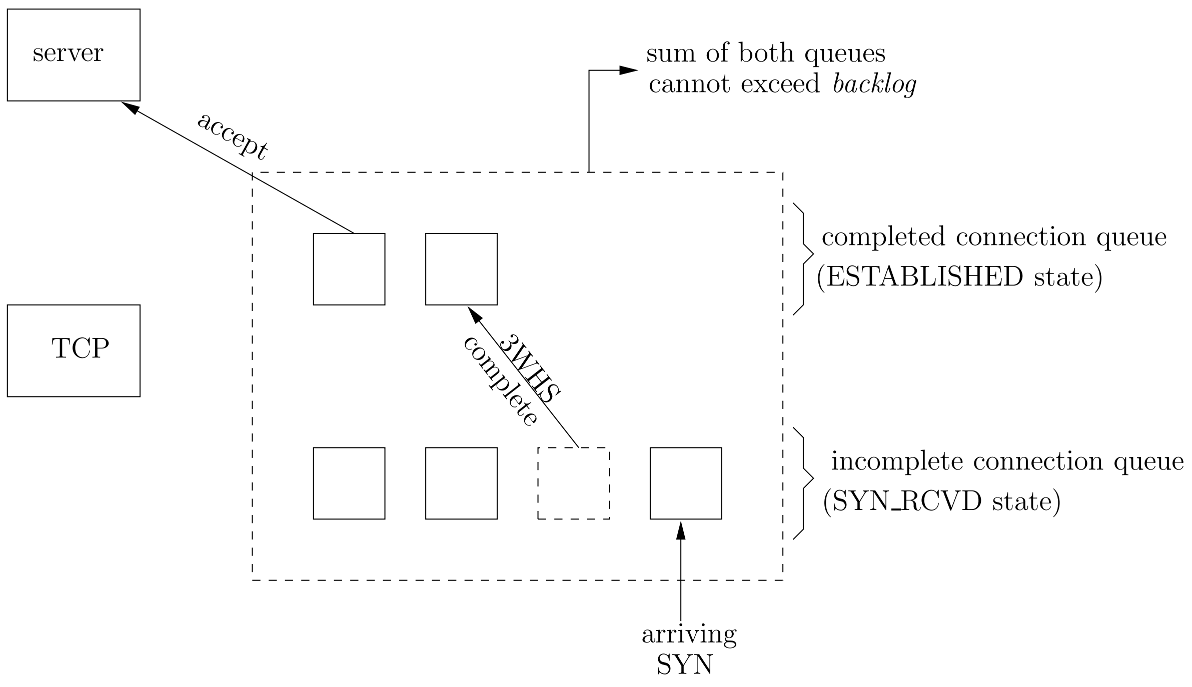

Figure 79: The two queues maintained by TCP for a listening socket. (Fig.4.7,p.105)

f. Historical value of BSD for this parameter is 5 (from 4.2BSD). 4.3BSD multiples the value by 5, i.e. if the given value is 5, the maximum value would be 8.

g. Actual number of queued connections for different implementations: Fig.4.10, p.108

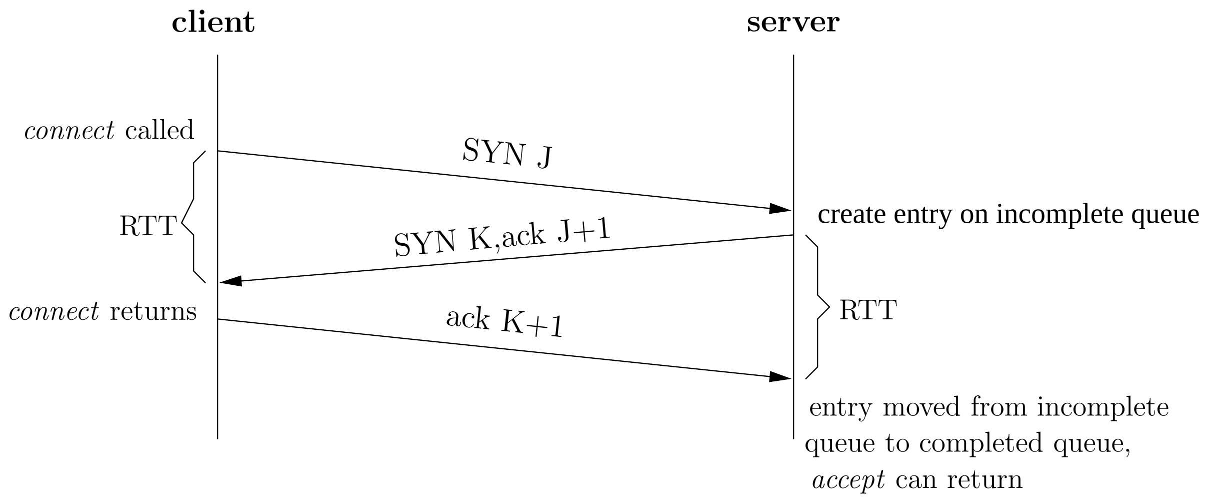
client
server
connect called
SYN J
create entry on incomplete queue
RTT
SYN K,ack J+1
RTT
connect returns
ack K+1
entry moved from incomplete queue to completed queue, accept can return

Figure 80: TCP three-way handshake and the two queues for a listening socket (Fig.4.8,p.105)

	Maximum actual number of queued connections				
	MacOS 10.2.6			FreeBSD 4.8	
backlog	AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

Figure 4.10 Actual number of queued connections for values of *backlog*

h. More about the *backlog* parameter: an experimental results: (Fig. 4.9, p.97 of the 2nd edition of the 2nd textbook)

(a) The results were from sampling the sizes of an HTTP server socket for two hours with a sampling interval 84 ms during the middle of a weekday.

(b) Two arrays were used, one array i for the incomplete queue and ther other c for the complete queue.

(c) During each sampling, if the number of entries in incomplete queue is k, then $x[k]$ is incremented by 1. Similarly, if the number of entries in complete queue is m, then $y[m]$ is incremented by 1.

#entries on queue	Incomplete queue	Complete queue
0	3,033	90,358
1	7,158	107
2	10,551	59
3	12,960	52
4	11,949	38
5	9,836	27
6	7,754	31
7	6,165	22
8	4,829	30
9	3,687	35
10	2,674	30
11	1,893	25
12	1,431	29
13	1,083	25
14	1,065	49
15	980	7
16	784	
17	696	
18	514	
19	392	
20	294	
21	248	
22	161	
23	152	
24	121	
25	77	
26	48	
27	33	
28	79	
29	78	
30	90	
31	70	
32	29	
33	16	
34	4	
	90,924	90,924

Figure 4.9 (of 2nd ed. of 2nd txbk) Number of entries on incomplete and completed connection queues

(6) *accept* function

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peer, socklen_t *addrlen);
```

a. *accept* takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there is no connection requests pending, this call blocks the caller until one arrives.

a. The *peer* and *addrlen* are used to return the address of the connected peer process (the peer).

b. The argument *addrlen* is a **value-result** parameter: the caller sets its value before the call, and the system stores a result in it when returned. The caller sets addrlen as the size of the buffer, while the function changes this value on return to the actual amount stored in the buffer.

c. This function automatically creates a new socket descriptor, which is usually passed to a child process.

d. Three values are returned:

   ∗ An integer value, either indicating an error (if less than 0), or representing a new socket descriptor.

   ∗ The address of the client process (in *peer*).

   ∗ The actual size of the buffer.

e. Example of value-result parameters: Fig. 4.11, p.110, a daytime server that uses the accept function to print out the value of a client's IP number and port number.

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

int
main(int argc, char **argv)
{
   int               listenfd, connfd;
   socklen_t         len;
   struct sockaddr_in   servaddr, cliaddr;
   char            buff[MAXLINE];
   time_t            ticks;

   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(13);   /* daytime server */
   Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

   Listen(listenfd, LISTENQ);

   for ( ; ; ) {
      len = sizeof(cliaddr);
      connfd = Accept(listenfd, (SA *) &cliaddr, &len);
      printf("connection from %s, port %d\n",
            Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
            ntohs(cliaddr.sin_port));

      ticks = time(NULL);
      snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
      Write(connfd, buff, strlen(buff));

      Close(connfd);
   }
}
```

Figure 4.11 Daytime server that prints client IP address and port.

(7) *fork* and *exec* functions

a. *fork* call, creates a new process. The creating process – the parent process; the created process – the child process:

   #include <unistd.h>
   pid_t fork(void);

   ∗ If unsuccessful, a single value -1 is returned. If successful, it returns twice: once to the child with value 0, once to the parent process with the PID of the child.

   ∗ The two share the same code and continue from the point of creation.

   ∗ Files opened are shared by both parent and child.

   ∗ After creation, the child process obtains a copy of its parent's data segment. The values copied from the parent process: real user ID; real group ID; effective user ID; effective group ID; process group ID; terminal group ID; root dir; CWD; signal handling settings; file mode creation mode;

b. *exec* functions

∗ A child process can invokes any program through one of these six functions. This is the only way to do it. The *fork* function is often called together with *exec* call to run a new program. There are six different versions of *exec* function:

```
#include <unistd.h>
pid_t fork(void);
int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, const char *argv[]);
int execle(const char *pathname, const char *arg0, ...
                /* (char *) 0, const char *envp[] */);
int execve(const char *pathname, const char *argv[], char const *envp[]);
int execlp(const char *filename, const char *arg0, ... /* char *) 0 */);
int execvp(const char *filename, const char *argv[]);
```

∗ Explanations:

· The first, third, and the fifth functions provide a variable length of arguments.

· The second, fourth, and the sixth passing parameters through arrays of pointers. The last element of the array must be a null pointer.

· The fifth and the sixth specify the *name* of the file to be executed. A search (through the environment variable PATH) will be performed if the file is not in CWD. The other four use path names and hence will not do the search.

· The 1,2, 5,6 functions do not specify an explicit environment pointer. The environment is built from the env variable *environ*. The other two specify such a list. Must terminate with a null pointer.

∗ Relationship among the six functions: Fig.4.12, p.113. They are all based on the implementation of the *execlp* function.

(8) Concurrent server (revisited)

a. Code for a typical concurrent server: Fig.4.13, p.114 (change the iterative server in Fig.4.11,p.110 to a concurrent server).

```
pid_t    pid;
int      listenfd, connfd;

listenfd = Socket( ... );
```

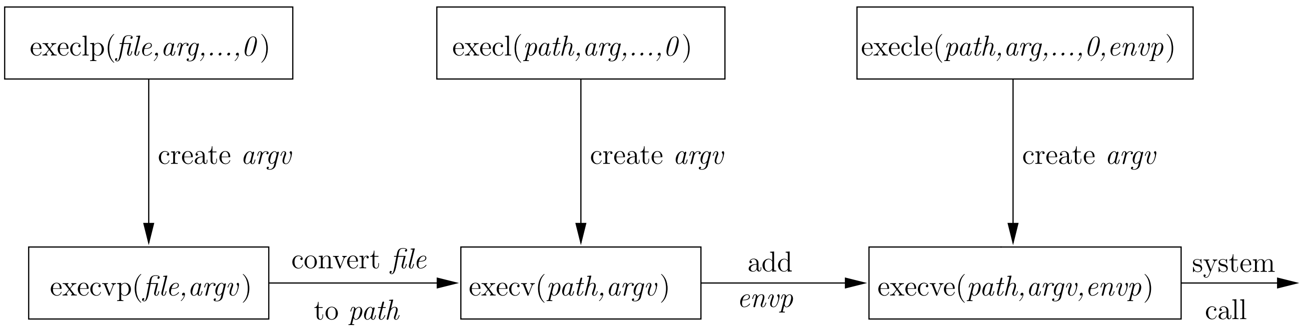

Figure 81: Relationship among the six *exec* functions. (Fig.4.12,p.113)

```
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
   connfd = Accept(listenfd, ... );     /* probably blocks */
   if ( (pid = Fork()) == 0) {
      Close(listenfd);    /* child closes listening socket */
      doit(connfd);       /* process the request */
      Close(connfd);      /* done with the client */
      exit(0);            /* child terminates */
   }
   Close(connfd);         /* parent closes connected socket */
}
```

Figure 4.13 Outline for typical concurrent server.

b. Illustration of operations of concurrent servers (Fig.4.14,4.15,p.115, Fig.4.16,4.17,p.116)
   (a) Fig.4.14: the status of client-server before call to *accept* function.

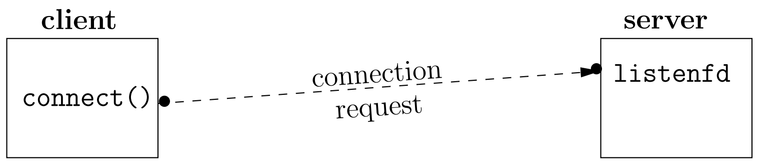

Figure 82: Status of client-server before call to *accept*. (Fig.4.14,p.115)

   (b) Fig.4.15: the status of client-server after a call to *accept* returns.
   (c) Fig.4.16: the status of client-server after a call to *fork* successfully returns.
   (d) Fig.4.17: the status of client-server after parent and child *close* appropriate sockets.

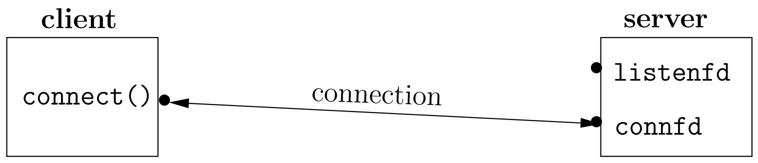

Figure 83: Status of client-server after return from *accept*. (Fig.4.15,p.115)

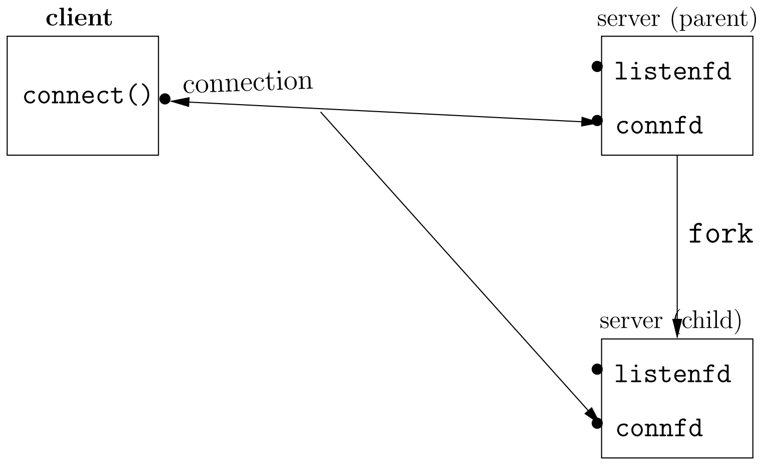

Figure 84: Status of client-server before after *fork* returns. (Fig.4.16,p.116)

(9) *close* function

int close(int *fd*);

a. It is a normal UNIX *close* function. Applied to a socket descriptor, the TCP termination sequence (4 message exchanges) will be initiated (active close).

b. For socket associated with a reliable delivery protocol (TCP or SPP), the system must make sure that any data within the kernel that still has to be transmitted or acknowledged, is sent.

c. Normally the system returns from close immediately, but the kernel will still try to send any data already queued.

d. Reference count of descriptors

   (a) Each descriptor (file or other types) has a *reference count* that is initialized to 1.

   (b) When a new process is generated, the new process gets a copy of each open descriptors. The reference count is incremented by one.

   (c) When a process closes a socket, the reference count of that socket descriptor is reduced by one. The kernel will actually close a connection when the reference count of the descriptor is zero.

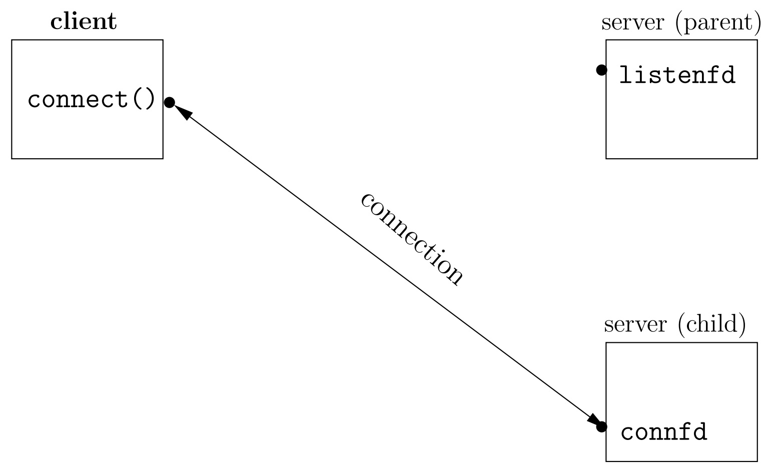

Figure 85: Status of client-server after parent and child *close* appropriate sockets. (Fig.4.17,p.116)

e. The function *shutdown* (to be discussed in Chapter 6) allow an application to have more control over the termination operation.

f. The action can be controled with *SO_LINGER* socket option.

(10) *getsockname* and *getpeername* functions

#include <sys/socket.h>
int getsockname(int *sockfd*, struct sockaddr $*$*localaddr*, socklen_t $*$*addrlen*);
int getpeername(int *sockfd*, struct sockaddr $*$*peeraddr*, socklen_t $*$*addrlen*);

a. *getsockname* function

   $*$ Return the "name" of the local process, i.e. the *local address*, and *local process* elements of the 5-tuple.

   $*$ *addrlen* again is a value-result argument.

b. *getpeername* function

   $*$ Return the "name" of the peer process, i.e. the *foreign address*, and *foreign process* elements of the 5-tuple.

   $*$ *addrlen* is a value-result argument.

   $*$ Can this be used for connectionless socket?

c. Use of these two functions: p.119.

   (a) A TCP client that does not call *bind* function before calling *connect* function can use getsockname to obtain the local IP and port nubmer after *connect* successfully returns (Recall: the kernel will assign an IP address and port number to such a client).

(b) After calling *bind* with port number of 0 (asking the kernel to choose port number), a client can use getsockname to obtain the local port number.

(c) The *getsockname* function can be used to obtain the protocol family information (Fig.4.19).

(d) A TCP server that uses *INADDR_ANY* can find out its specific local IP address after a client makes a connection.

(e) The only way for a child process of a server to know the identification (IP address and port number) of the client is through the *getpeername* function.

· The so called Internet super daemon process *inetd* (details discussed in Ch.13): Fig.4.18, p.119

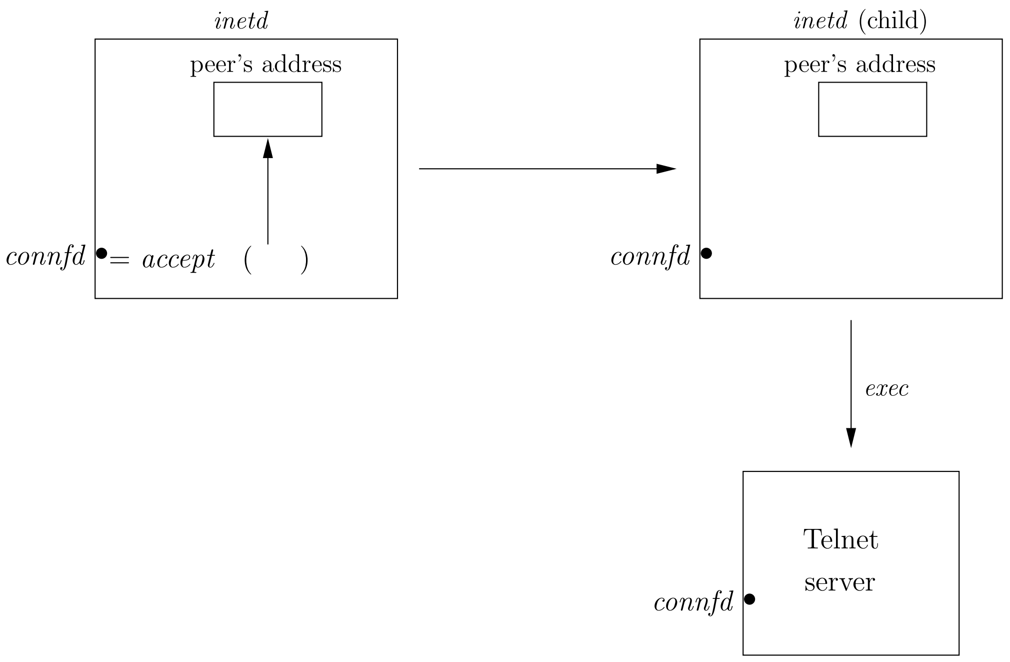

Figure 86: Example of *inetd* spawning a server. (Fig.4.18,p.119)

· A program that obtains the address family of a socket: Fig.4.19, p.119

```
#include   "unp.h"

int
sockfd_to_family(int sockfd)
{
   struct sockaddr_storage ss;
   socklen_t   len;

   len = size(ss);
```

```
    if (getsockname(sockfd, (SA *) ss, &len) < 0)
       return(-1);
    return(ss.ss_family);
}
```

Figure 4.19 Return address family of a socket

2. TCP client/server example (Chapter 5)

(0) Outline: a simpel client-server example that uses the previously discussed socket related functions (Fig.5.1, p.121).

   a. The clients reads a line from its stdin, and write through a socket to the server.
   b. The server reads a line from its socket and sends back whatever it read to the client.
   c. The client receives the reply and printed it on its stdout.

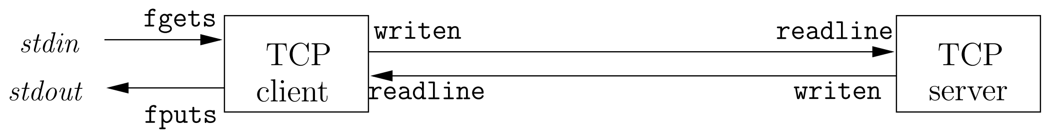

Figure 87: Simple echo client and server. (Fig.5.1,p.121)

(1) The TCP echo server: Fig.5.2, p.123.

```
#include   "unp.h"

int
main(int argc, char **argv)
{
   int                 listenfd, connfd;
   pid_t              childpid;
   socklen_t           clilen;
   struct sockaddr_in   cliaddr, servaddr;

   listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

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

   Listen(listenfd, LISTENQ);

   for ( ; ; ) {
      clilen = sizeof(cliaddr);
      connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

      if ( (childpid = Fork()) == 0) {   /* child process */
         Close(listenfd);   /* close listening socket */
         str_echo(connfd);   /* process the request */
         exit(0);
      }
      Close(connfd);         /* parent closes connected socket */
   }
}
```

Figure 5.2 TCP echo server (improved in Figure 5.12).

a. It is a concurrent server.

b. The child calls the procedure *str_echo* to perform the duty.

c. The *str_echo* procedure (p.124, Fig.5.3): called by the TCP echo server.

```
#include   "unp.h"

void
str_echo(int sockfd)
{
   ssize_t      n;
   char      buf[MAXLINE];

 again:
   while ( ( n = read (sockfd, buf, MAXLINE) ) > 0 )
      Writen(sockfd, buf, n);

   if (n < 0 && errno == EINTR)
      goto again;
   else if (n < 0)
```

```
        err_sys("str_echo: read error");
}
```

Figure 5.3 *str_echo* function: echo lines on a socket.

(2) The TCP echo client: Fig.5.4, p.124.

```
#include "unp.h"

int
main(int argc, char **argv)
{
    int         sockfd;
    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));

    str_cli(stdin, sockfd);      /* do it all */

    exit(0);
}
```

Figure 5.4 TCP echo client.

a. It is very similar to the daytime client.

b. It calls the procedure *str_cli* function to handle the communication.

c. The *str_cli* procedure (p.125, Fig.5.5): called by the TCP echo client.

```
#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);
   }
}
```

Figure 5.5 *str_cli* function: client processing loop.

(3) Normal startup (p.126-127)

a. Start the server as a background (daemon) process;

b. Use *netstat* command to find out the status and port number info of the server;

c. Testing the server by making a client connection from the same host:

   (a) The client initially is blocketd in attempting to get a line from stdin;

   (b) The server, when returning from the accept function, will fork a child process, which in turn calls the str_echo function;

   (c) The server itself goes back to execute accept function again.

d. Use *netstat* again to observe the status of the connection and port number info of the client;

e. Use *ps* command to observe the processes for server, the child of server, and the client.

(4) Normal termination (p.128-129)

a. A control-D input will cause the client to terminate the connection;

b. Use *netstat* to observe the new status of the connection;

c. Steps involved in a normal connection termination:

   (a) EOF (ctl-D) will return a null pointer for *fgets*. The function *str_cli* returns;

(b) The client main function will terminate by calling *exit* function;

(c) The child of the server returns 0 from readline function, which causes the str_echo function to return to the child's main function;

(d) The server child terminates by calling *exit*;

(e) The kernel on the client side closes all descriptors of the client process. This closing action sends a FIN to the server, which causes the server to respond by sending an ACK. At this time the client is in FIN_WAIT_1 state and the server is in CLOSE_WAIT state;

(f) All open descriptors in the server child are closed, which causes the final two segments FIN to be sent, and a final ACK sent by the client. At this time, the client is in TIME_WAIT state;

(g) The server kernel will send a SIGCHLD signal to the server process, which is ignored because the server process doesn't specify how to handle such signal. The child process enters the zombie state. (A zombie process is a process which has terminated, but whose termination information is still maintained by the kernel so that the parent of the zombie can retrieve it late on).

3. Discussions of the TCP client-server example

(1) POSIX signal handling

a. Signals. A signal is also called a software interrupt. A signal is a notification from the kernel to a process about the occurrence of certain event. Example of signals: I/O ready signal; termination signal; interrupt signal; SIGCHLD signal (sent by the kernel to a parent process to notify the termination of a child process of the parent).

b. Signal handling:

(a) Every signal has a *disposition*, which is also called the *action* associated with that signal. The kernel has *default* disposition for every signal supported.

(b) A process can define a function, called *signal handler*, for a signal. Once declared, the kernel will call that function for every occurrence of the specified signal (called *catching a signal*). There are two signals SIGKILL and SIGSTOP that cannot be handled by processes themselves.

(c) Signal handlers can be specified by using the old *signal* function, or the newer *sigaction* function.

(d) A signal can also be blocked. A process call use variable, called *signal mask*, to specify which signals to be blocked. The function *sigprocmask* allows a process to do that;

(e) Syntax and semantics of the *signal* function.

void (*signal(int sig, void (*function)(int))) (int);

(f) Implementing the *signal* function using *sigaction* function: Fig.5.6, p.130.

```
#include   "unp.h"

Sigfunc *
signal(int signo, Sigfunc *func)
{
   struct sigaction   act, oact;

   act.sa_handler = func;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (signo == SIGALRM) {
#ifdef   SA_INTERRUPT
      act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
#endif
   } else {
#ifdef   SA_RESTART
      act.sa_flags |= SA_RESTART;      /* SVR4, 44BSD */
#endif
   }
   if (sigaction(signo, &act, &oact) < 0)
      return(SIG_ERR);
   return(oact.sa_handler);
}
```

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

c. POSIX signal semantics (p.132)

(a) Once a signal is installed, it remains in effect until explictly changed;

(b) When a signal is being handled, another signal of the same type, and all signals specified in the signal mask, will be blocked.

(c) Blocked signals of the same type are not queued. I.e., if multiple signals of the same type occured more than once, only one will be kept by the kernel.

(2) Handling SIGCHLD signal

a. Problems with zombie processes: consumption of system resources;

b. Solution: retrieve the termination status of a child process as early as possible after the child terminates.

c. We can create a simple SIGCHLD signal handler to handle the termination of child process for the serer: Fig.5.7, p.133. The function *sig_chld* simply calls the function *wait* to retrieve the termination information of the child.

```
#include   "unp.h"

void
sig_chld(int signo)
{
   pid_t   pid;
   int      stat;

   pid = wait(&stat);
   printf("child %d terminated\n", pid);
   return;
}
```

Figure 5.7 Version of `SIGCHLD` signal handler that calls `wait`.

d. New behavior of the server after the signal handler for SIGCHLD is incorporated (p.133)

   (a) Once ctl-D is typed, the output from the signal handler is displayed.

   (b) The server itself aborts because the signal SIGCHLD is being delivered while it is waiting and being blocked at the *accept* function. This return from the accept causes

e. Handling interrupted system calls (functions);

   (a) `Slow functions`: Some networking functions, such as *accept, read, write*, are called *slow functions* because they can potentially block the calling process indefinitely. In other words, these functions may never return.

   (b) The standard behavior of Unix systems is that, when a process is blocked due to a slow function, a signal delivery will cause error `EINTR` returned as value of that function.

   (c) Observations: receiving an `EINTR` error from a slow function while an application is blocked is not an error from the perspective of the application.

   (d) Some implementations will automatically restart a slow function after a signal is delivered. Some others (most of BSD variants) will not automatically restart.

(e) We can handle this problem by adding some code (p.124) to examine the error variable after returning from a slow function and continue if error is equal to `EINTR`.

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

(f) The above approach will not work for the *connect* function (why?). Another method has to be used (non-blocking, Ch.15).

(3) *wait* and *waitpid* functions

a. They are used to allow a parent process wait for the termination and to inspect termination info of a child process. The *waitpid* function allows a process to have more control.

b. Syntax

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

c. *wait* function: it allows a parent to wait for the termination of any of its child processes.

(a) The calling parent process is blocked if none of the child processes has terminated.

(b) If one of the child processes has terminated (and its termination info has not been retrieved) the function will return immediately.

(c) If the calling process does not have any child processes (terminated or executing), the call returns -1.

d. *waitpid* function:

(a) It allows a parent to specify which process to wait;

(b) The last parameter allows more control over how to wait. The most commonly used option is WNOHANG, which allows a parent to return immediately if no terminated child processes.

e. Differences between *wait* and *waitpid*.

(a) A modified client that will make five connections at one time: illustration of connection: Fig.5.8, p.136

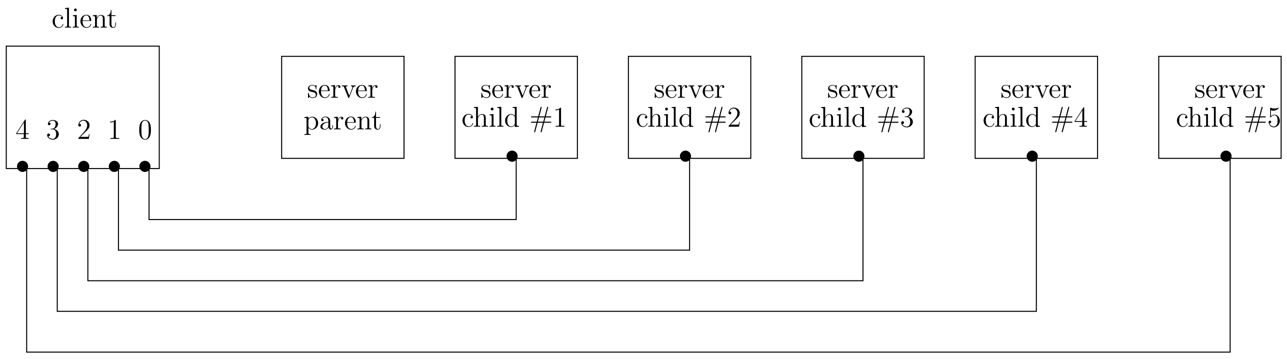

Figure 88: Client with five established connections to same concurrent server. (Fig.5.8,p.136)

(b) Code of the modified client: Fig. 5.9, p.136.

```
#include   "unp.h"

int
main(int argc, char **argv)
{
   int                i, sockfd[5];
   struct sockaddr_in   servaddr;

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

   for (i = 0; i < 5; i++) {
      sockfd[i] = 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[i], (SA *) &servaddr, sizeof(servaddr));
   }
   str_cli(stdin, sockfd[0]);      /* do it all */
   exit(0);
}
```

Figure 5.8 TCP client that establishes five connections with server.

(c) When the client terminates, all open descriptors are closed by the client kernel and all five connections are closed at approximately same time. Hence five FIN segments are sent to server (Fig.5.10, p.126).

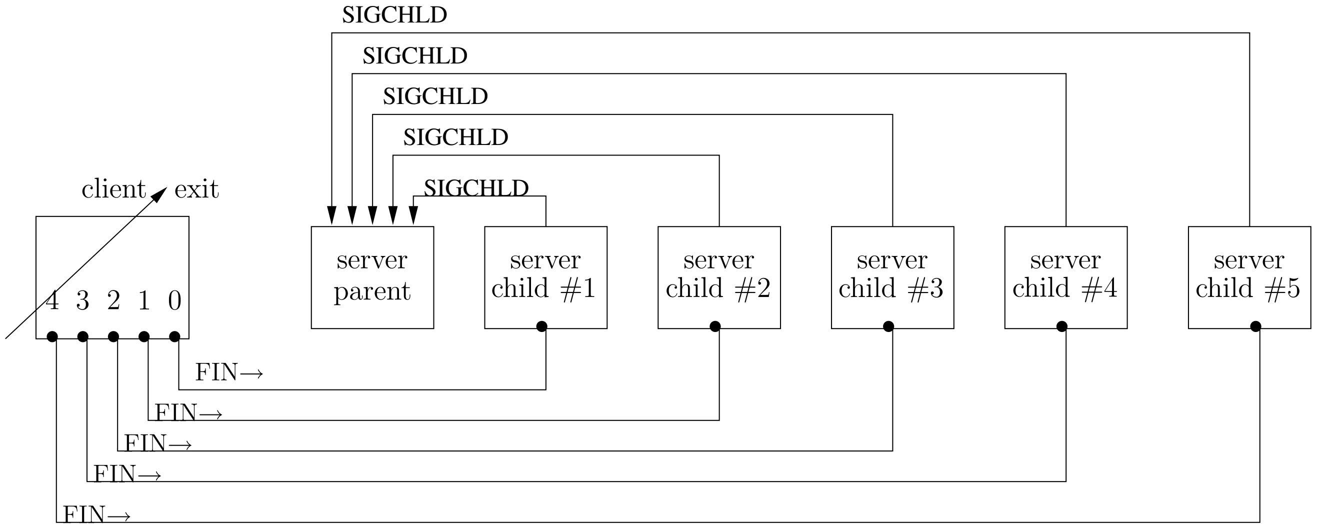

Figure 89: Client terminates, closing all five connections, terminating all five children. (Fig.5.10,p.126)

(d) Problem occurs because the delivery of the same type of signals multiple times at approximately same time: only one caught, and four others are lost. This results in four zombie processes!

(e) An improved version of the server that can correctly handle the above mentioned problems: Fig.5.11, p.138 and Fig. 5.12, p.138-139.

· The new signal handler for SIGCHLD calls waitpid in a loop, with WNOHANG option. This allows all five signal to be handled systematically.

```
#include   "unp.h"

void
sig_chld(int signo)
{
   pid_t   pid;
   int      stat;

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

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

```
#include   "unp.h"

int
main(int argc, char **argv)
{
   int                listenfd, connfd;
   pid_t             childpid;
   socklen_t          clilen;
   struct sockaddr_in   cliaddr, servaddr;
   void             sig_chld(int);

   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);

   Signal(SIGCHLD, sig_chld);   /* must call waitpid() */

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

      if ( (childpid = Fork()) == 0) {   /* child process */
         Close(listenfd);   /* close listening socket */
         str_echo(connfd);   /* process the request */
         exit(0);
```

```
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

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

f. Lessons learned from this section:
   (a) Slow functions can cause un-expected problems;
   (b) Signals are tricky to deal with in general;
   (c) Deep system and implementation knowledge sometimes are needed to correctly and reliably handle a specific signal.

(4) Connection aborts before *accept* returns

a. The client may abort a connection request by sending an RST (reset) message before a connection is established. In this case, another (nonfatal) error will return from an `accept` function call. This might be caused by killing a client process before a connection is established.

b. Illustration: Fig.5.13, p.140.

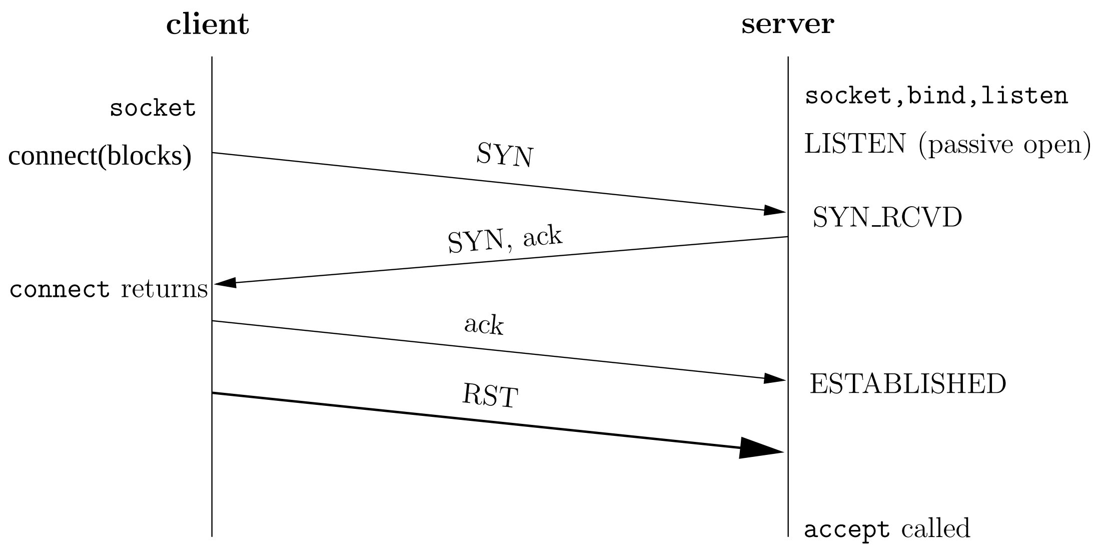

Figure 90: Receiving an RST for an ESTABLISHED connection before `accept` is called (Fig.5.13,p.140)

c. How to handle this error situation is implementation dependent.
   (a) BSD variants handle this completely inside kernel, and user processes will not even see this (i.e. no error returns)

(b) Most SVR4 based implementations will return an error. How to handle this error is again dependent on implementations.

(5) Termination of server process: kill the child process on server side after connection established.

a. A FIN segment will be sent from the server kernel to the client. The client's kernel will respond with an ACK message. This happens when the client is blocked at the *fgets* function. In other words, the client process is not aware of the arrival of the FIN segment from the server child;

b. On the server side, a SIGCHLD signal will be delivered to the server process and will be handled correctly in the modified version of server;

c. Use *netstat* on a Solaris machine at the client side shows that client is in CLOSE_WAIT state. Notice the difference output format of *netstat* command on Solaris;

d. Use *netstat* on the server machine showed that the server is in *FIN_WAIT_2* state, implying that the first two segments in TCP termination have been exchanged so far;

e. On the client side, if we type another input line quickly enough, we can still send another line to the server from the client. A response from the server is still received! This is possible because receiving an FIN from the server just means that the server will not initiate any new segment transmission. This is called a *half closed* socket. The server (child) at that state can still receive incoming segments;

f. But a second line sent by the client to the server will encounter the *server terminated prematurely* error message. This response is produced by the client kernel (a RST error) since the requested server child process has terminated by the time the second segment is received by the client kernel. Therefore, this RST is only seen (i.e. received) by the client kernel, not by the client process;

g. Such a RST segment will cause the *readline* function to return a value 0 (i.e. EOF), which is unexpected. The client process then quits with a "server terminiated prematurely" message;

h. Root of the above described problem: the client is being blocked at an input function (*fgets*) when a FIN from server child arrives. The client process hence is not properly informed of the arrival of the FIN segment. The client has to deal with two descriptors in an alternative fashion: a socket descriptor and a fild descriptor;

i. Solution: use a technique that allows the client to be able to monitor more than one descriptor simultaneously. The technique should allow the client to handle an I/O event that occurs at any one of the monitored descriptors at any time and act accordingly. The two functions *select* (from BSD) and *poll* (from SVR4) will provide this ability (Chapter 6).

(6) SIGPIPE signal: consequences of writing to a closed socket

a. A client may attempt to write to a closed socket. That could happen when a client in our example performs two *write*s to the server for each single input line. The first write will incur the RST segment;

b. Modified client function *str_cli* that writes twice: Fig.5.14, p.143. The function first writes the 1st byte of the input line and then takes a one second pause. It then sends the rest of the input line;

```
#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);
    }
}
```

Figure 5.14 `str_cli` that calls `writen` twice.

c. Consequence: when a process writes to a socket that has received a RST, the kernel will signal a SIGPIPE signal to the process as response. The default disposition for SIGPIPE is to terminate the receiving process;

d. A process can elect to specify a signal handler to *catch* or to specify explicitly to ignore SIGPIPE. In both cases, a write will return a EPIPE error;

e. Results of running the modified *str_cli*:

  (a) (p.133) After the server child is killed; the input line *bye* does not get echoed back;

  (b) In Korn shell, using the command

  echo $?

  , which returns the return value of last executed command, returns value 269. Using the head file */usr/include/sys/signal.h* shows that SIGPIPE has value 13. Under bsdi, it adds 256 to all SIG??? values listed in signal.h file.

  (c) Different UNIX implementations might have different return value. Under Digital UNIX 4.0, Solaris 2.5.1, UNIXWare 2.2.1, KornShell would return 141 (128+13);

  (d) How to handle SIGPIPE depents applications.

    · Ignore it might be a good choice if nothing special is required by the application.

    · Otherwise we might have to provide a signal handler to do special handling for it.

(7) Crash of server host

a. Simulating crash of server host

  (a) First we start the server and client on separate hosts. Let them perform several rounds of communications.

  (b) We then disconnect the server from the network. This simulates the both cases where the server crashes after establishing communications with the client, and where the server is unreachable from the client.

b. The client sends a line to the server. The client will then be blocked on the *readline* function call. There are two possible outcomes:

  (a) Because the server crashes, the TCP component of the client kernel will timeout and perform repeated retransmissions until it final reaches a predefined TCP timeout value. At that time the TCP component at the client's kernel will give up and return ETIMEDOUT error to the readline function call. BSD based UNIX system will retransmit 12 times, which will result in a total 9 minute waiting interval before giving up $((22.5 * 2) * 12 = 540)$

  (b) Some intermediate routers might determine that the server host is not reachable and hence send an ICMP *destination unreachable* message to the client host. In this case, the client kernel will return a EHOSTUNREACH or ENETUNREACH error to the readline function call.

c. How to find out the server crashes ealier (before we send a data segment and timeout for response): use the *SO_KEEPALIVE* socket option (Chapter 7.5).

(8) Crash and rebooting of server host

a. Make a successful connection from the client to the server. Then shutdown the server. Then Reboot the server. Then let the client send a line to the server.

b. When that line arrives at the server host, which carries connection information obtained before the server host crashes, the server kernel does not have that obsolete connection information (such as port # of the client process, sequence number of the segment). The TCP at the server host will respond with a RST, which will cause readline to return the error ECONNRESET;

c. If it is important for the client to be aware of the crash and rebooting of the server host, the SO_KEEPALIVE socket option has to be applied on the client host side.

(9) Shutdown of server host

a. The different between this case and the case of server host crashing is that the server host is shutdown by the server host operator;

b. The server host kernel will send a SIGTERM signal to each process. SIGTERM is catchable and if a process is well coded, it should catch this signal to terminate gracefully. This includes launching an active closing on each open descriptors;

c. If SIGTERM is not caught by the server process, a subsequent SIGKILL signal will terminate the process. This would cause the same sequence of actions as for termination of server process (case 9 in this notes).

(10) Summary of TCP example

a. Fig. 5.15, p.146 shows a TCP connection session from client's perspective. The client has to specify the foreign address (server's IP), foreign service (server's port number);

(a) The client normally does not specify its local address and/or local service (port number), although it can elect to do so by calling *bind* before *connect*;

(b) If the client does not choose its local IP address, the client host will choose one for the client based on its current routing information.

(c) If a client didn't choose its local address and port number before connection, it can obtain its local address and port number using the *getpeername* function;

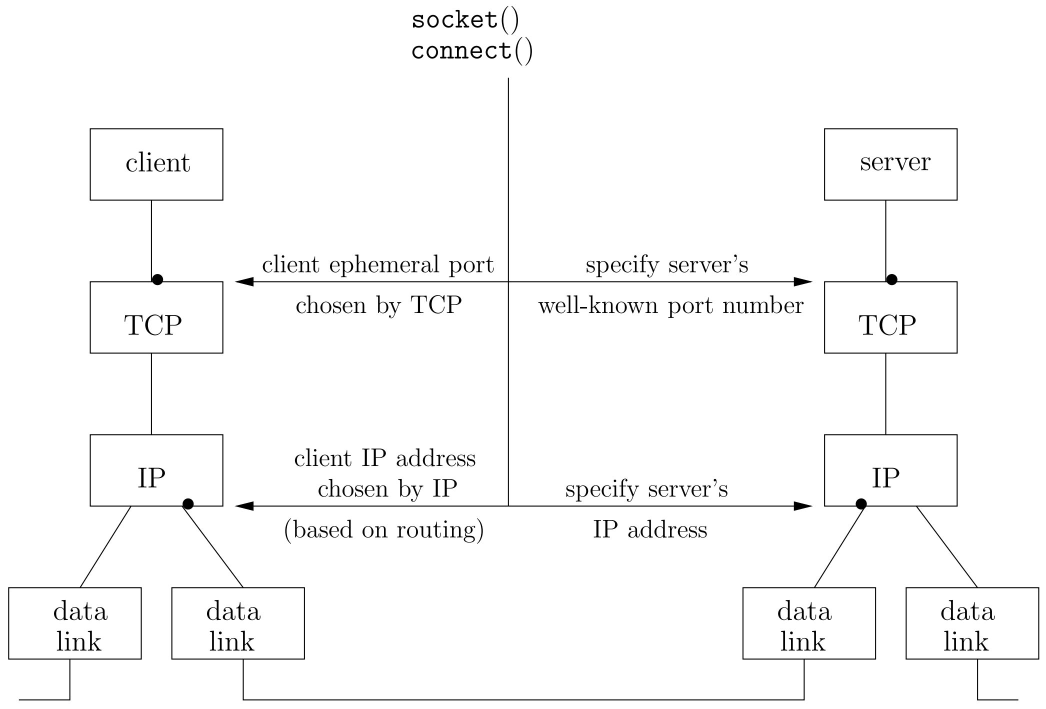

Figure 91: Summary of TCP client-server from client's perspective. (Fig.5.15,p.146)

(d) The client must specify the server host's IP address and the server's well-known port number in order to initiate a connection.

b. Fig. 5.16, p.147 shows a TCP connection session from server's perspective.

(a) The server normally specify its local address (server's IP) as a wildcard (i.e. the INADDR_ANY constant) so that clients can connect to the server process using any of the server host's IP addresses.

(b) The server process has to specify its local service (server's well-known port number) that has been agreed by the clients;

(c) The foreign address and port number of the client are returned to the server by the *accept* function;

(d) A server child (for a concurrent server) can obtain the client's address and port number using the *getpeername* function.

(11) Handling formatted messages

a. Data format might be important for many TCP applications, although both the client and server in the client-server example so far ignore the format of received messages.

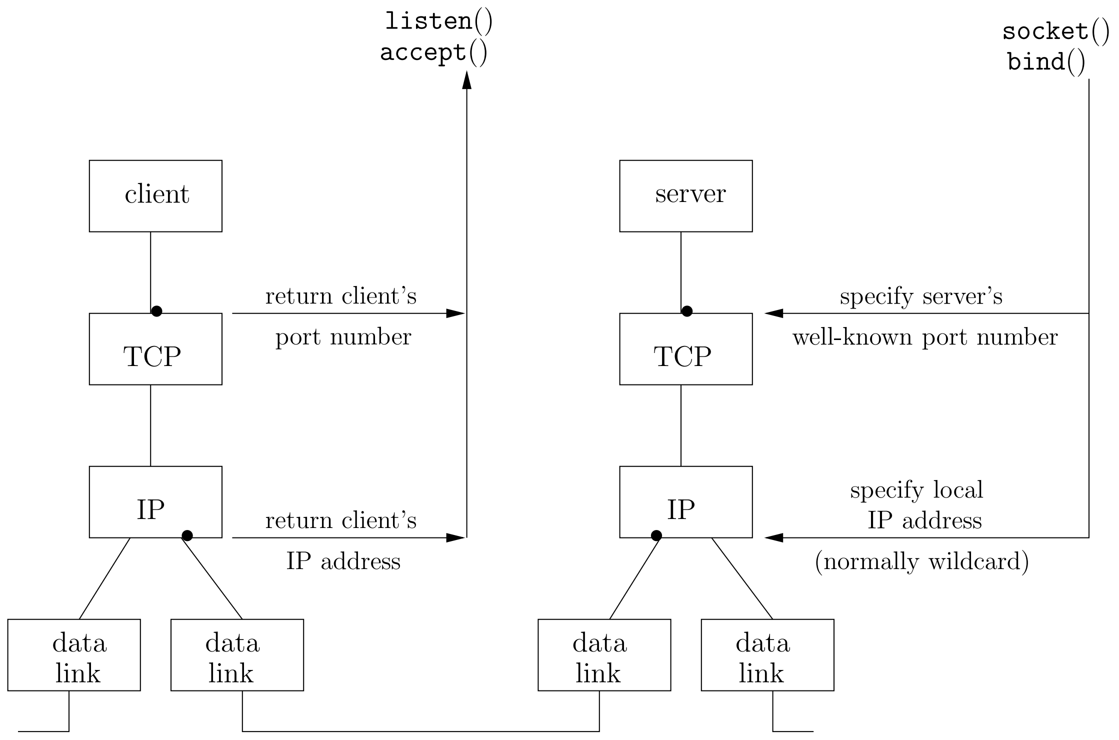

Figure 92: Summary of TCP client-server from server's perspective. (Fig.5.16,p.137)

b. The formats of request messages and reply messages depend on individual applications.

c. Example: passing text strings and interpreting data format implicitly.

  (a) The server now expects that each line sent by the client contains two integers separated by white space. The server will extract the two integers and send the sum of them back to the client;

  (b) The only change needed in the *str_echo* function (Fig.5.17, p.148).

```
#include   "unp.h"

void
str_echo(int sockfd)
{
   long      arg1, arg2;
   ssize_t      n;
   char      line[MAXLINE];

   for ( ; ; ) {
      if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
         return;      /* connection closed by other end */
      if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
         snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
      else
         snprintf(line, sizeof(line), "input error\n");

      n = strlen(line);
      Writen(sockfd, line, n);
   }
}
```

Figure 5.17 `str_echo.h` function that adds two numbers.

In this case, byte ordering of client and server hosts does not come into picture (why?).

d. Passing structure data

   (a) Each line sent by the client now contains a structure with two fields of type *long* (Fig.5.18, 5.19, p.148,149);

```
struct args {
  long   arg1;
  long   arg2;
};
struct result {
  long   sum;
};
```

Figure 5.18 `sum.h` header.

```
#include   "unp.h"
#include   "sum.h"

void
str_cli(FILE *fp, int sockfd)
{
   char          sendline[MAXLINE];
   struct args      args;
   struct result   result;

   while (Fgets(sendline, MAXLINE, fp) != NULL) {
      if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
         printf("invalid input: %s", sendline);
         continue;
```

```
    }
    Writen(sockfd, &args, sizeof(args));
    if (Readn(sockfd, &result, sizeof(result)) == 0)
       err_quit("str_cli: server terminated prematurely");
    printf("%ld\n", result.sum);
 }
}
```

Figure 5.19 `str_cli` function that sends two binary integers to server.

(b) The new *str_echo* function (Fig.5.20, p.149) now extracts the structure;

```
#include   "unp.h"
#include   "sum.h"

void
str_echo(int sockfd)
{
   ssize_t          n;
   struct args       args;
   struct result    result;

   for ( ; ; ) {
      if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
         return;      /* connection closed by other end */
      result.sum = args.arg1 + args.arg2;
      Writen(sockfd, &result, sizeof(result));
   }
}
```

Figure 5.20 `str_echo` function that adds two binary integers.

(c) Problem: because different architectures have different byte ordering and size for the data type *long*, unexpected outcome is possible.

· If the server and client run on hosts of the same architecture, we get correct results: p.139, the server host is Solaris, the client host is SUNOS5;

· If the server and client run on hosts with different architecture with different byte ordering, or different size for *long*, we get incorrect results: p.150, the server host is Solaris (big endian), the client host is bsdi (little endian);

(d) Solutions:

· Passing text, and let the processes themselves determine the meaning of the text (case (1) above);

· Using an explicitly defined binary format of supported data types. An example is the XDR (external data representation) of SUN RPC package.

4. Lecture summary

(1) Introduced elementary TCP socket functions.

(2) Introduced in details our first TCP client-server example.

(3) Discussed various problems of and possible fixes to the first version of the TCP client-server example.