Lecture 2 Overview

- Last Lecture
  - TCP/UDP
- This Lecture
  - Sockets introduction
  - Elementary TCP sockets
  - TCP Client-Server example
  - Source: Chapters 3, 4, 5
- Next Lecture
  - I/O multiplexing
  - Source: Chapter 6

Socket Address Structure

- Socket API is the popular fundamental networking API in Unix/Linux
- BSD POSIX socket API
  - POSIX.1g, approved in 2000.

```
struct in_addr {
  in_addr_t   s_addr;          /* 32-bit IPv4 address */
                               /* network byte ordered */
};

struct sockaddr_in {
  uint8_t          sin_len;    /* length of structure (16) */
  sa_family_t      sin_family; /* AF_INET */
  in_port_t        sin_port;   /* 16-bit TCP or UDP port number */
                               /* network byte ordered */
  struct in_addr   sin_addr;   /* 32-bit IPv4 address */
                               /* network byte ordered */
  char             sin_zero[8]; /* unused */
};
```

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

Posix.1g Data Types

Datatype	Description	Header
int8_t	signed 8-bit integer	<sys/types.h>
uint8_t	unsigned 8-bit integer	<sys/types.h>
int16_t	signed 16-bit integer	<sys/types.h>
uint16_t	unsigned 16-bit integer	<sys/types.h>
int32_t	signed 32-bit integer	<sys/types.h>
uint32_t	unsigned 32-bit integer	<sys/types.h>
sa_family_t	address family of socket address structure	<sys/socket.h>
socklen_t	length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

Figure 3.2 Datatypes required by Posix.1g.

IPv6 Socket Address

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t        sin6_len;       /* length of this struct (24) */
  sa_family_t    sin6_family;    /* AF_INET6 */
  in_port_t      sin6_port;      /* transport layer port# */
                                 /* network byte ordered */
  uint32_t       sin6_flowinfo;  /* priority & flow label */
                                 /* network byte ordered */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
};
```

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`.

Value-Result Arguments

- connect(sockfd, &serv, sizeof(serv));

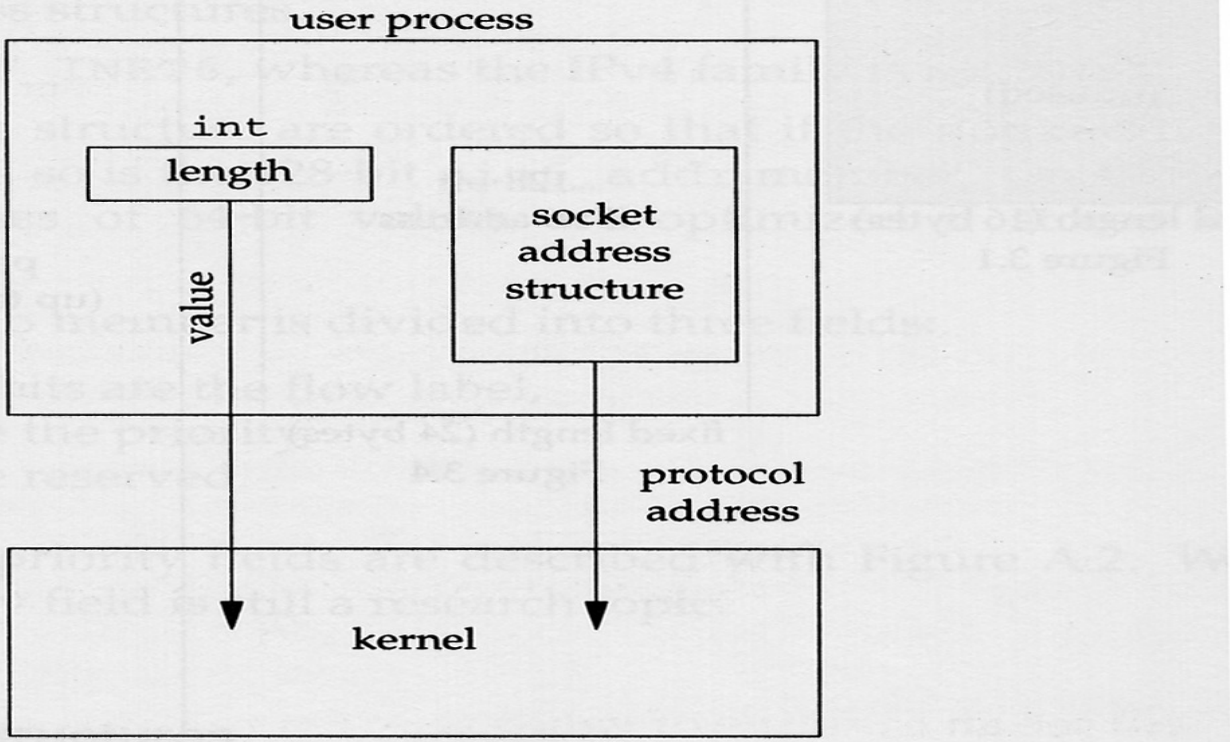

Figure 3.6 Socket address structure passed from process to kernel.

Value-Result arguments (cont.)

- accept(sockfd, &from, &len);

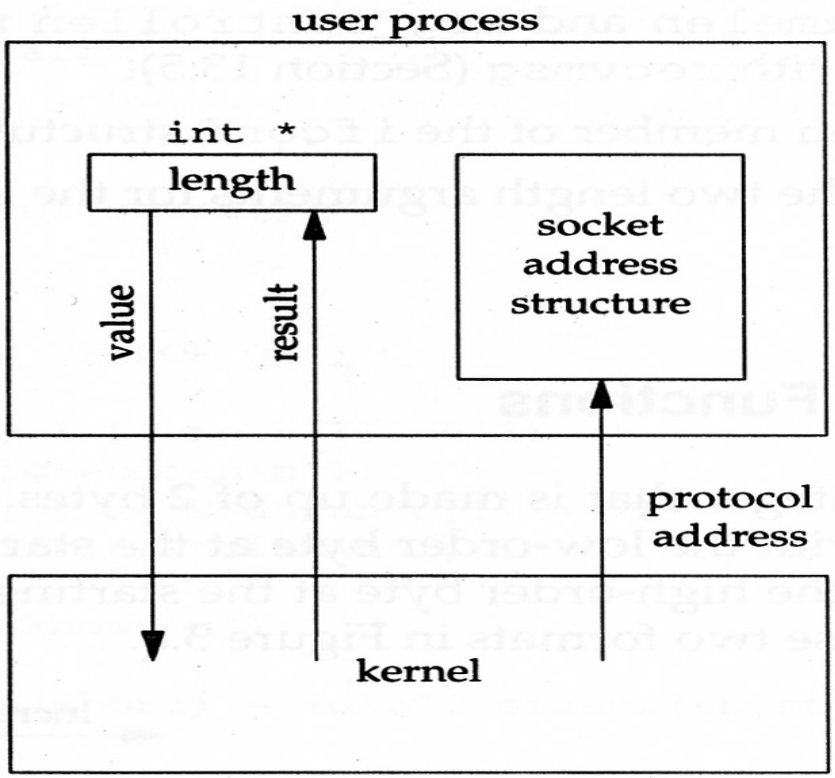

Figure 3.7 Socket address structure passed from kernel to process.

Byte order

- Little-endian and big-endian

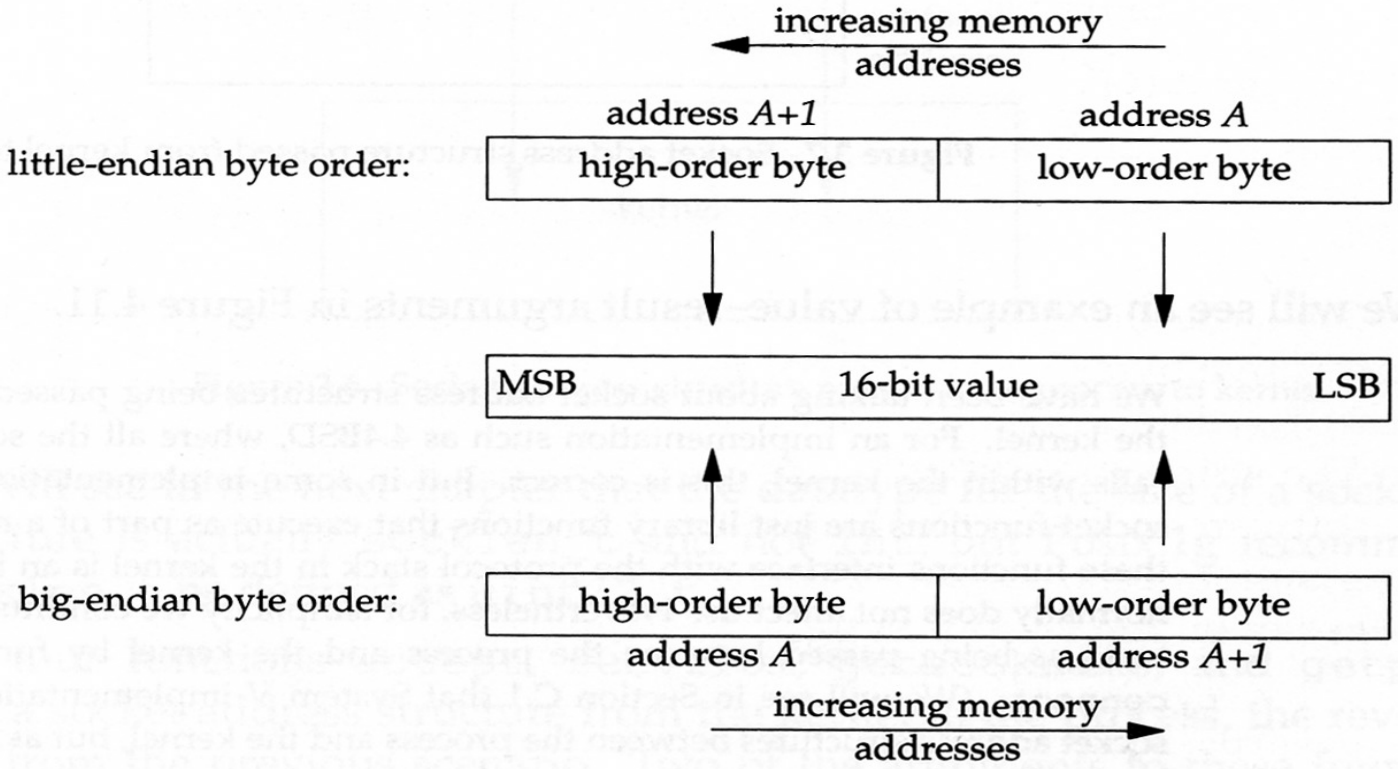

Figure 3.8 Little-endian byte order and big-endian byte order for a 16-bit integer.

Byte ordering functions

- Network byte order
  - Big-endian in Internet protocol suite
  - Not all protocols use big-endian byte order
- Byte ordering functions
  - uint16_t htons(uint16_t host16bitvalue);
  - uint32_t htonl(uint32_t host32bitvalue);
  - uint16_t ntohs(uint16_t net16bitvalue);
  - uint32_t ntohl(uint32_t net32bitvalue);

Byte manipulation functions

- Berkeley-derived
  - void bzero(void *dest, size_t nbytes);
  - void bcopy(const void *src, void *dest, size_t nbytes);
  - int bcmp(const void &ptr1, const void *ptr2, size_t nbytes);
- ANSI C
  - void *memset(void *dest, int c, size_t len);
  - void *memcpy(void *dest, const void *src, size_t nbytes);
  - int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

Address conversion functions

- Traditional functions
  - int inet_aton(const char *strptr, struct in_addr *addrptr);

    -- convert network address from string to binary from in network byte order
  - char *inet_ntoa(struct in_addr inaddr);

    -- converts address from binary form to a string in IPv4 dotted-decimal notation.
  - int inet_pton(int af, const char *src, void *dst);

    -- converts the character string src into a network address in the af family
  - const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);

    -- converts the address structure src in the af family into a character string
  - #define INET_ADDRSTRLEN 16 /* IPv4 dotted –decimal*/
  - #define INET6_ADDRSTRLEN 46 /*IPv6 hex string*/

Address conversion functions

- How to make functions independent of protocols?

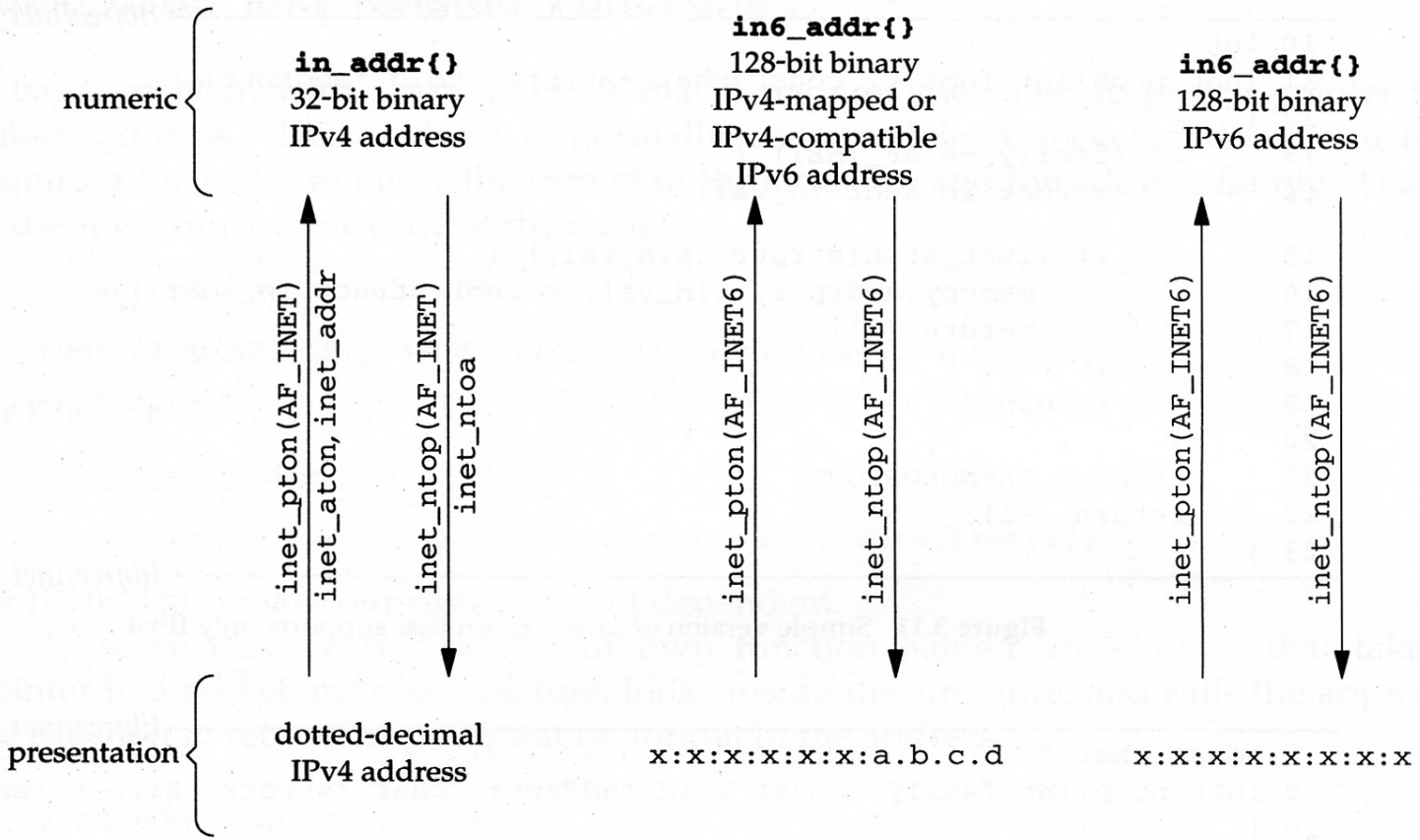

Figure 3.10 Summary of address conversion functions.

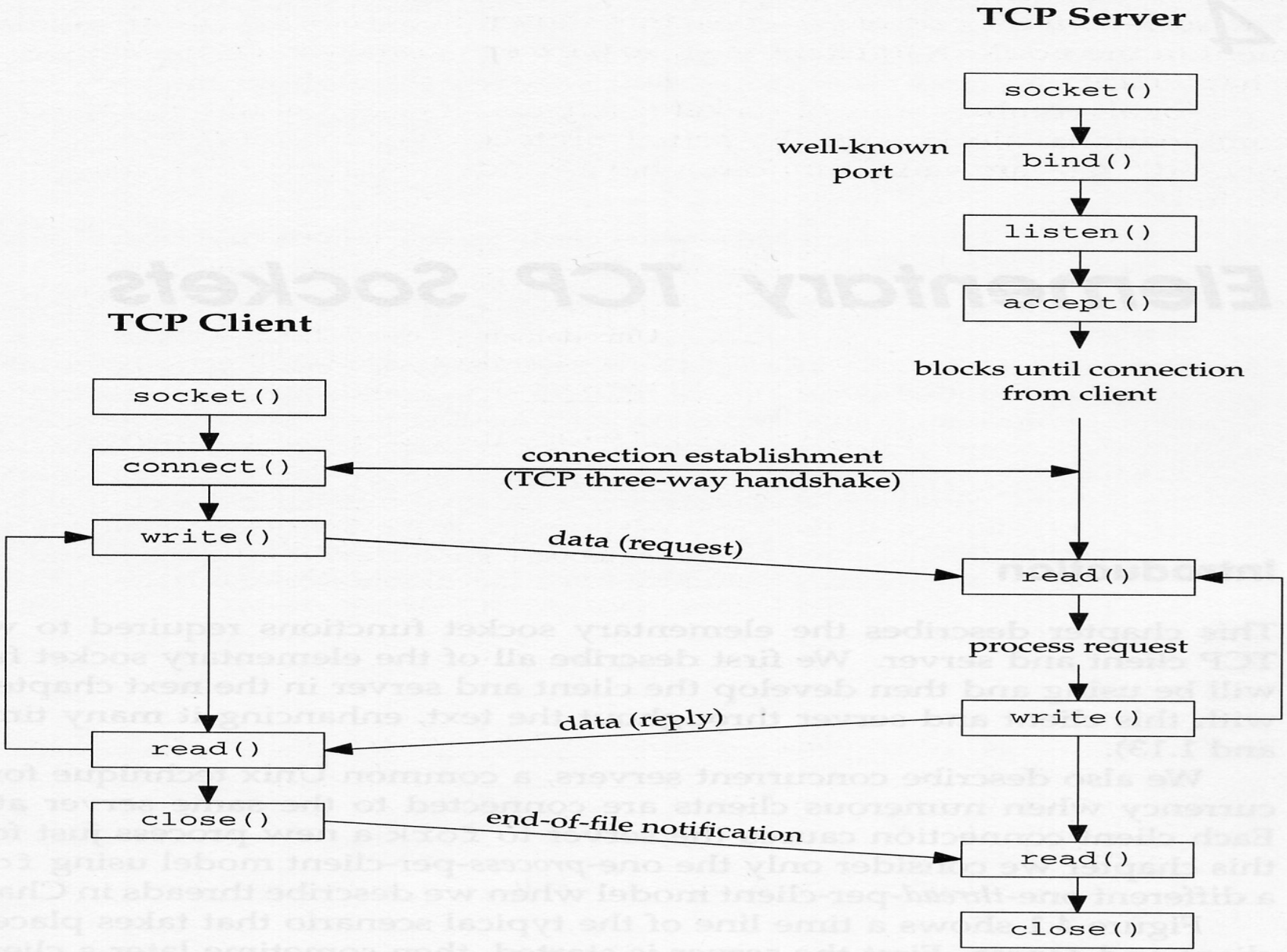

Figure 4.1 Socket functions for elementary TCP client–server.

Socket

- Socket function is used to create a socket for communication

```
int socket(int family, int type,int protocol)
```

  - *family*: specifies the protocol family
    - AF_INET (IPv4 protocols)
    - AF_INET6 (IPv6 protocols),
    - AF_LOCAL (Unix domain protocols)
    - AF_ROUTE (Routing sockets)
    - AF_KEY (Key socket)
  - *type*: specify the type of the socket
    - SOCK_STREAM (stream socket)
    - SOCK_DGRAM (datagram socket)
    - SOCK_RAW (raw socket)
    - SOCK_SEQPACKET (sequenced packet socket)
  - *Protocol:* specify a specific communication protocol
    - IPPROTO_TCP (TCP transport protocol)
    - IPPROTO_UPD (UDP transport protocol)
    - 0: the system' default for a given combination of `family` and `type`
  - *Returned value:* sockfd (socket descriptor)
    - >=0 succeed
    - -1, error

Socket (cont.)

- AF_xxx vs PF_xxx
  - AF_xxx stands for address family
  - PF_xxx stands for protocol family
  - The intent was that a single protocol family might support multiple address families.
  - Currently they have the same set of values

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Figure 4.4 Combinations of *family* and *type* for the `socket` function.

Bind

- The `Bind` function is used to assign a local protocol address to a socket.

```
int bind(int sockfd,const struct sockaddr *myaddr,
                socklen_t addrlen)
```

  - Returns: 0 if OK, -1 on error
  - Assigns a local protocol address to a socket
    - Combination of IP address (32 or 128 bit) and 16-bit TCP or UDP port number
  - *Bind* specifies a port number, and IP address, both or neither

Generic Socket Address Structure

- Socket address structure is always passed by reference.
- How to declare the type of pointer that is passed
- ANSI C has a generic pointer type: void *
- Generic socket address structure for IPv4

```
struct sockaddr {
        uint8_t         sa_len;
        sa_family_t     sa_family; /*address family: AF_xxx*/
        char            sa_data[14];/*protocol specific address*/
};
```

- Generic socket address structure for IPv6

```
struct sockaddr_storage {
        uint8_t         sa_len;
        sa_family_t     sa_family; /*address family: AF_xxx*/
};
```

Generic Socket Address Structure

Function prototype:

```
int bind(int, struct sockaddr *, socklen_t)
```

Using bind:

```
struct sockaddr_in serv;
/* fill in serv{} */
bind(sockdf, (struct sockaddr *) &serv, sizeof(serv));
```

Bind (cont)

- Scenario of use of bind
  - Server programs bind their well-known port
  - A process can bind a specific IP address to its socket (the IP address must belong to an interface on the host),
    - For a TCP client, the IP address is used as source IP address
    - For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

Bind (cont.)

- Important points
  - Normally a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used.
  - If a TCP server doesn't bind an IP address to its socket (i.e. using wildcard INADDR_ANY), the kernel uses the destination IP address of the client's SYN segment as the server's source IP address.
  - If we specify a port number of 0, the kernel chooses an ephemeral port when *bind* is called.
  - If we specify a wildcard IP address, the kernel doesn't choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP)
  - For IPv4, the constant *INADDR_ANY* should be used for wildcard address
  - To get kernel chosen port or IP address, use *getsockname*.

Listen

- The `listen` function called only by a TCP server

  `int listen(int sockfd, int backlog)`

  - Called after *socket* and *bind*, and must be called before *accept*.
  - *listen* converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
  - Moves the socket from CLOSED to LISTEN
  - Backlog specifies the maximum number of connections that the kernel should queue for this socket
- Two queues maintained in the kernel
  - Incomplete connection queue: in SYN_RCVD state
  - Completed connection queue: in ESTABLISHED state

Listen (cont.)

- When *accept* is called, an entry is cleared from the completed connection queue, or wait (sleep) if the queue is empty

Figure 4.6 depicts these two queues for a given listening socket.

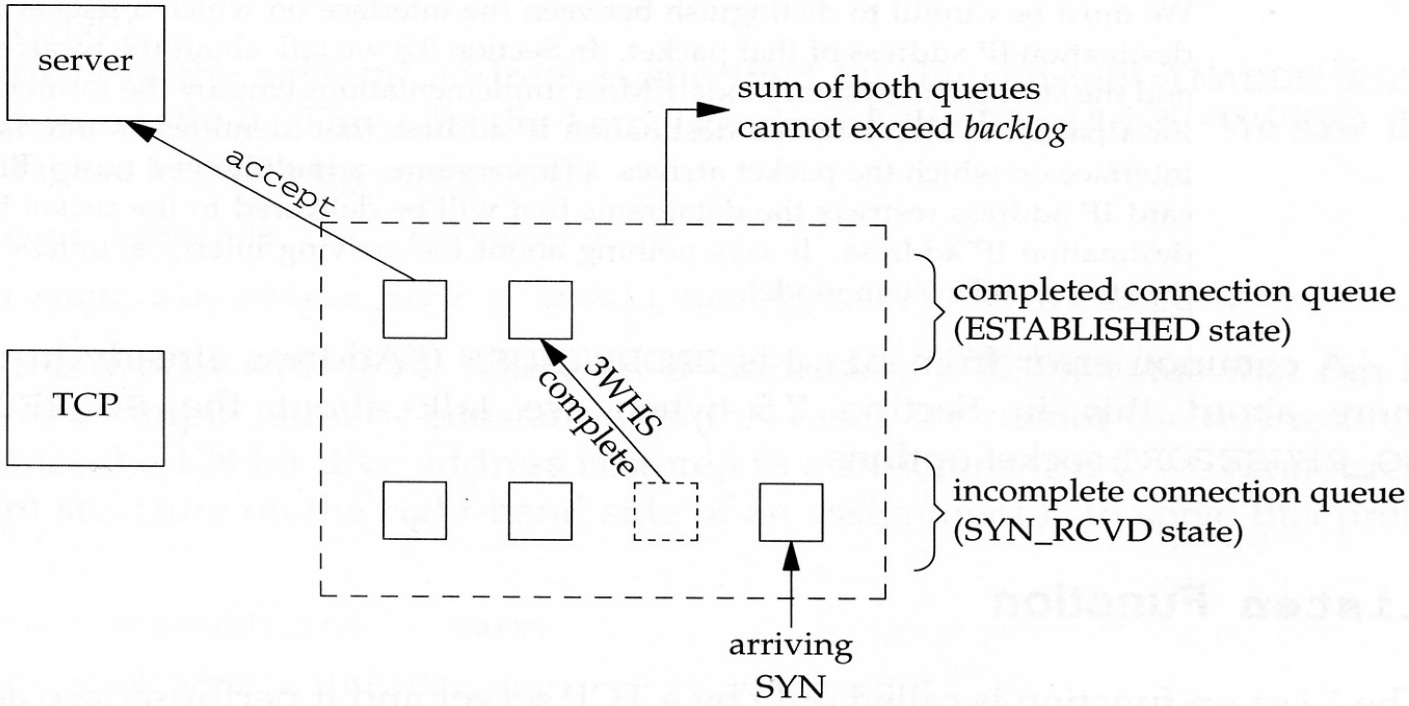

Figure 4.6 The two queues maintained by TCP for a listening socket.

backlog

- *backlog* is historically regarded as the maximum value for the sum of the two queues
- BSD adds a factor 1.5 to *backlog*, but other OSes may treat it differently
- Don't specify a *backlog* of 0, as different implementations interpret this differently
- If the 3-way handshake completes normally, an entry remains on the incomplete queue for RTT (roundtrip time)
- What value should the application specify?
  - Can use environment variable to override the value specified by the user
- Most of the time (99.4%) the complete queue is empty while there is a long incomplete queue.
- If the queues are full, the new arrived SYN is ignored
- Data after 3-way handshake completes but before calling *accept* are buffered

Connect

- `Connect` function is used by a TCP client to establish a connection with a TCP server.

```
int connect(int sockfd, const struct sockaddr *servaddr,
            socklen_t addrlen)
```

  - The socket address structure must contain the IP address and port number of the server
  - Returns: 0 if OK, -1 on error
- In case of TCP socket, *connect* initiates TCP's three-way handshake. The function returns only when the connection is established or an error occurs
- Possible returned errors
  - `ETIMEDOUT:` receive no response to the SYN segment
  - `ENETUNREACH:` destination unreachable
  - …

Connect (cont.)

- If `connect` succeeds, the socket moves to ESTABLISHED state; otherwise the socket moves to CLOSED state

- If connect fails, we can't call `connect` again on the same socket, but have to create a new socket with `socket` function.

Accept

- The `accept` function is called by a TCP server to return the next completed connection from the front of the completed connection queue; if the queue is empty, the process is put to sleep (assuming a blocking socket)

```
int accept(int sockfd, struct sockaddr *cliaddr,
           socklen_t *addrlen)
```

  - Returns: nonnegative descriptor if OK, -1 on error
  - *cliaddr* and *addrlen* are used to return the protocol address of the connected peer process.
  - *addrlen* is a value-result argument.
  - If not interested in the protocol address of the client, both can be set to null pointers.
  - The returned brand new socket is called the *connected* socket, while the argument *sockfd* is called the *listening* socket.

fork

- `Fork` is the only way in Unix to create a new process.

```
pid_t fork(void)
```

  - Create a copy of the process
  - Returns:
    - child's process ID in the parent
    - 0 in the child
    - -1 on error
    - Note: it is called *once* but returns *twice*
  - All descriptors open in the parent before the call to *fork* are shared with the child after *fork* returns.
  - Two typical uses of fork
    - A process copies itself so that one copy handles an operation while the other copy does another task
    - A process wants to execute another program

exec

- There are six *exec* functions
  - *exec* replaces the current process image with the new program file and this program normally starts at the *main* function
  - These functions return to the caller only if an error occurs
  - The difference in the six functions is
    - Whether the program file to execute is specified by a *filename* or a *pathname*. If it is a *filename*, the PATH environment variable is used. If there is a "/" in the name, PATH not used
    - Whether the arguments to the new program are listed one by one or referenced through an array of pointers
    - Whether the environment of the calling process is passed to the new program or a new environment is specified

exec (cont)

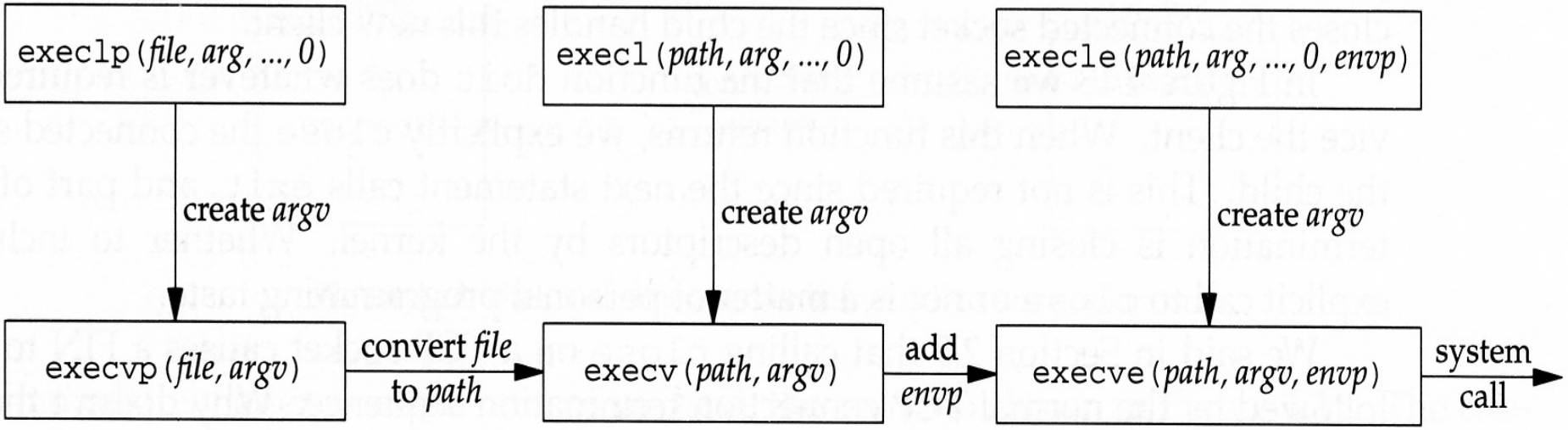

Figure 4.12 Relationship among the six exec functions.

- Descriptors opened in the process before calling *exec* normally remain open across the *exec*
  - This can be disabled by using fcntl to set the FD_CLOEXEC descriptor flag.

close

- `int close(int sockfd)`
  - Returns: 0 if OK, -1 on error
  - The socket is no longer usable by the process.
  - But already queued data will be sent to the other end before the TCP termination sequence takes place.
- Descriptor reference count
  - Used to track how many processes are using the descriptor.
  - Only when the count becomes 0, does TCP initiates the connection termination sequence on the socket descriptor
  - It is important for a process to close descriptors after using them; otherwise they will remain open for the life of the process

A Simple TCP Daytime Client

```
#include       "unp.h"
int
main(int argc, char **argv)
{
      int                     sockfd, n;
      struct sockaddr_in6     servaddr;
      char                    recvline[MAXLINE + 1];

      if (argc != 2)
            err_quit("usage: a.out <IPaddress>");
      if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)     err_sys("socket error");

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin6_family = AF_INET6;
      servaddr.sin6_port   = htons(13);/* daytime server */

      if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
            err_quit("inet_pton error for %s", argv[1]);

      if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)     err_sys("connect error");

      while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
            recvline[n] = 0;        /* null terminate */
            if (fputs(recvline, stdout) == EOF)
                  err_sys("fputs error");
      }
      if (n < 0)      err_sys("read error");

      exit(0);
}
```

Create a TCP socket (`socket`)

Specify server's IP address and port

Connect to the server (`connect`)

Send request or receive reply (`send & recv`)

Terminate program (close socket)

A Simple TCP Server

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

int main(int argc, char **argv)
{
 intlistenfd, connfd;
 struct sockaddr_inservaddr;
 charbuff[MAXLINE];
 time_tticks;

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 ( ; ; ) {
  connfd = Accept(listenfd, (SA *) NULL, NULL);
   ticks = time(NULL);
   snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
  Write(connfd, buff, strlen(buff));
  Close(connfd);
      }
}
```

Create a TCP socket (`socket`)

Specify server's IP address and port

Bind socket with local port (`Bind`)

Convert the socket to listening socket (`Listen`)

Accept client connection (`Accept`)

Receive or reply (`send & recv`)

Terminate connection (`Close`)

A Simple TCP Server

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

int main(int argc, char **argv)
{
 intlistenfd, connfd;
 struct sockaddr_inservaddr;
 charbuff[MAXLINE];
 time_tticks;

 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 ( ; ; ) {
   connfd = Accept(listenfd, (SA *) NULL, NULL);

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

   Close(connfd);
      }
}
```

An interactive server
Not a concurrent server

Concurrent Servers

```
listenfd = socket( . . .);
bind( . . .);
listen(listenfd . . .);
for ( ; ; ) {
    connfd = accept(listenfd  . . . );
    if ((pid = fork()) == 0) {
        close(listenfd);
        -- process request (connfd)--
        close(connfd);
        exit(0);
    }
    close(connfd);
}
```

Concurrent Servers (cont)

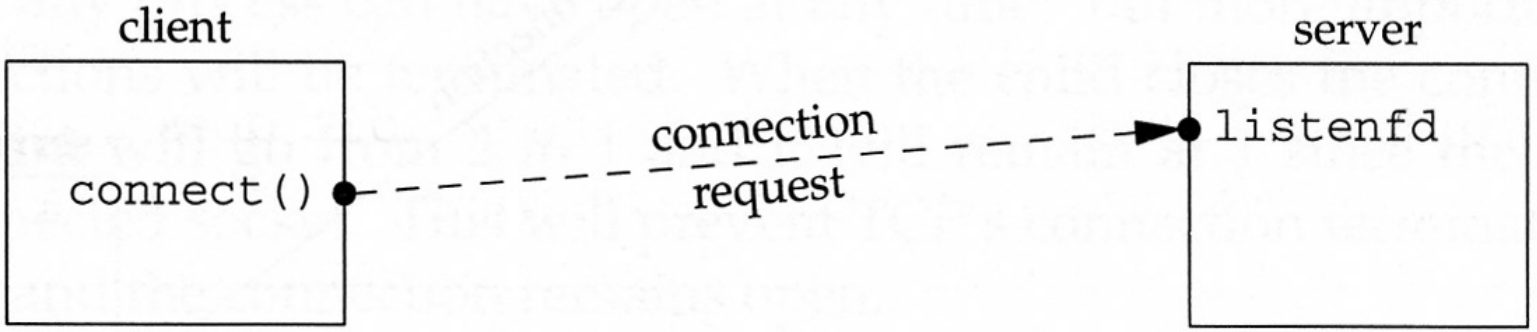

Figure 4.14 Status of client–server before call to `accept` returns.

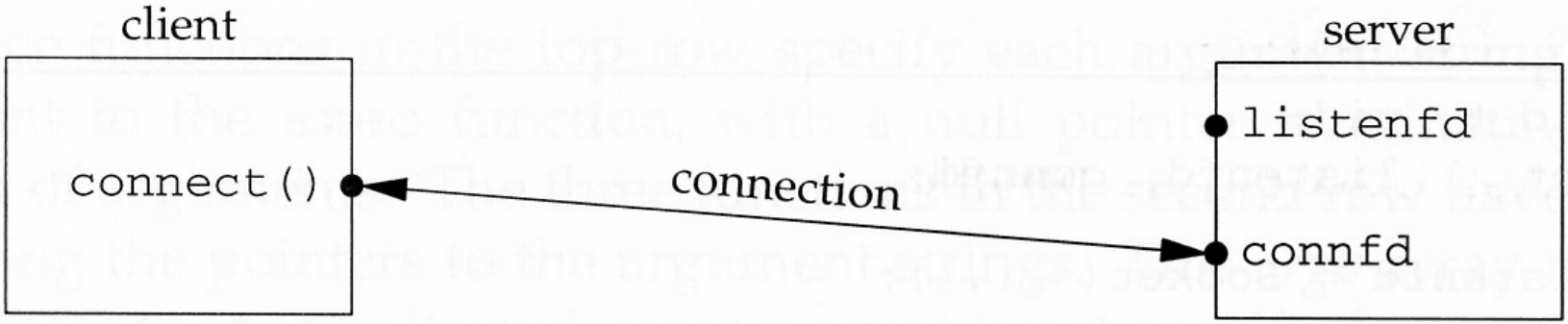

Figure 4.15 Status of client–server after return from `accept`.

Concurrent Servers (cont.)

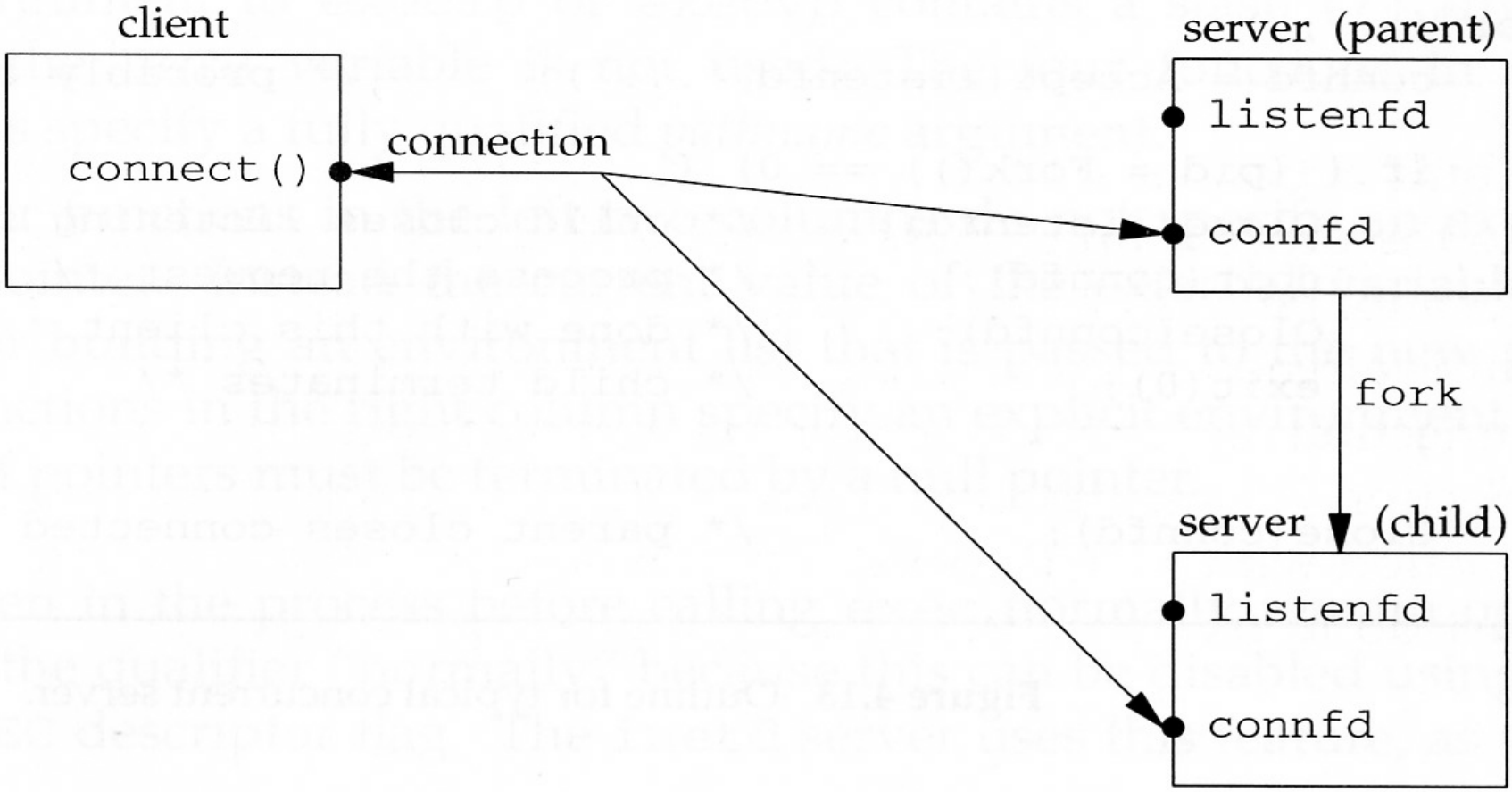

Figure 4.16 Status of client–server after `fork` returns.

Concurrent Servers (cont.)

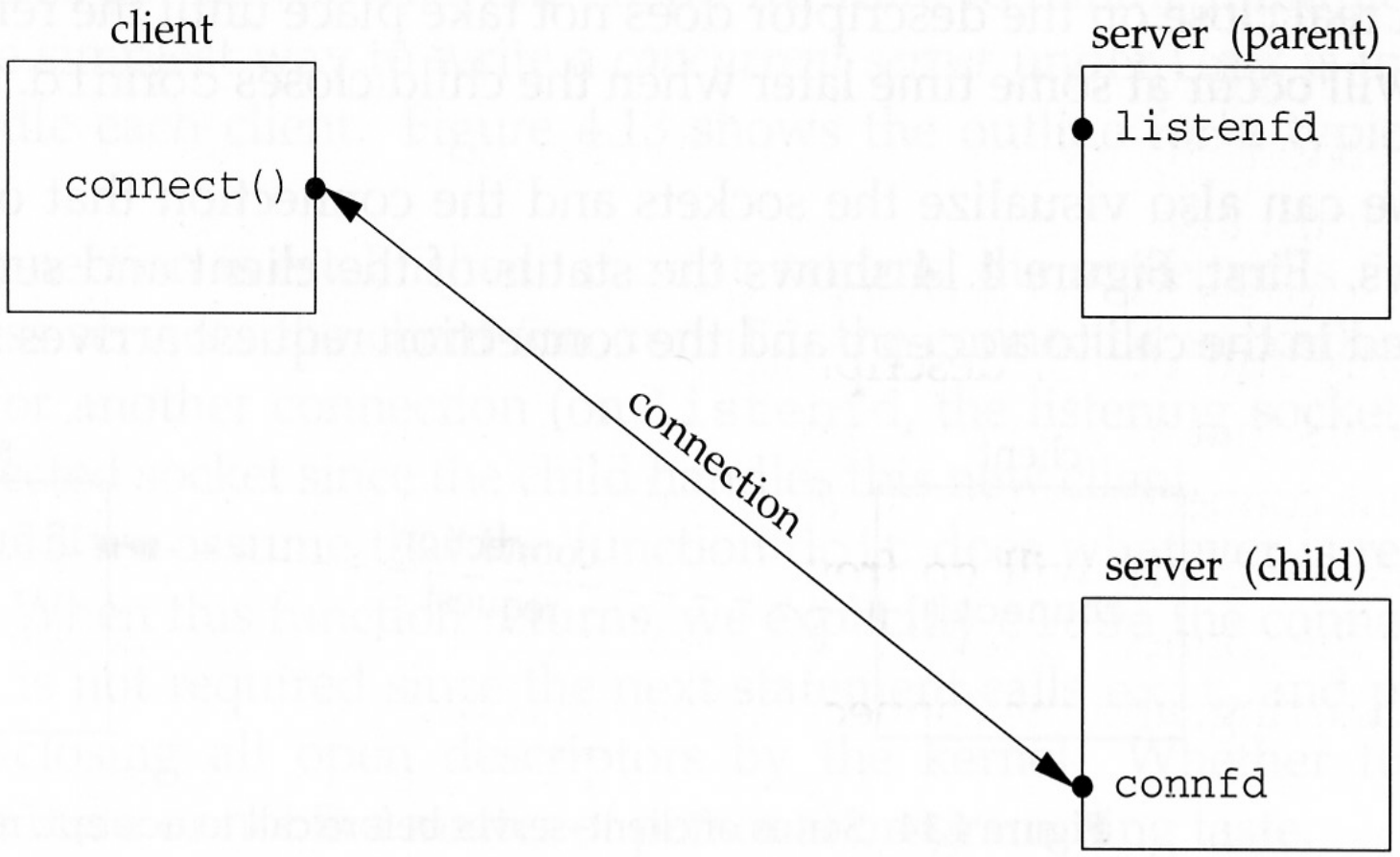

Figure 4.17 Status of client–server after parent and child close appropriate sockets.

getsockname and getpeername

- These two functions return either the local protocol address associated with a socket or the foreign protocol address associated with a socket
  - `int getsockname(int sockfd,struct sockaddr *localaddr, socklen_t *addrlen)`
  - `int getpeername(int sockfd,struct sockaddr *peeraddr, socklen_t *addrlen)`
  - Both return: 0 if OK, -1 on error
- Scenario of using them
  - Find the ephemeral port number for a process
  - Find address family of a socket by an *exec*ed process
  - Find the address of the bound interface in a server
  - Find the address of the peer in a server process

TCP Client/Server Example

- A simple echo server: description
  - The client reads a line of text from its standard input and writes the line to the server
  - The server reads the line from its network input and echoes the line back to the client
  - The client reads the echoed line and prints it on its standard output

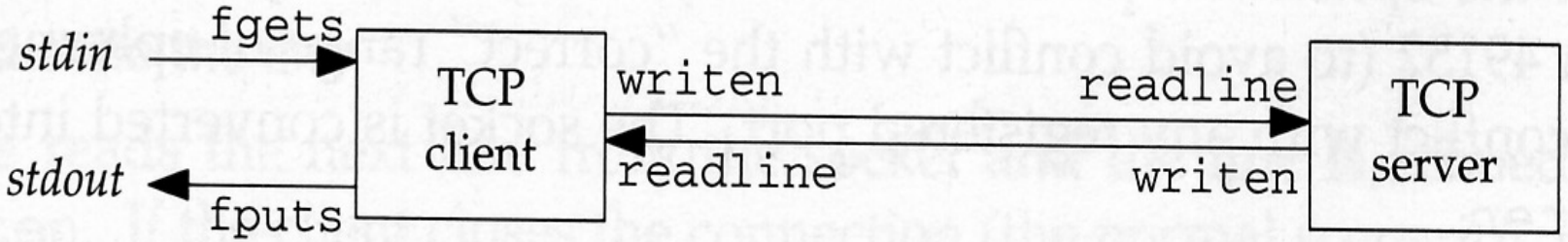

Figure 5.1 Simple echo client and server.

Program flow in Server (1)

- *main*
  - Create socket, bind server's well-known port
  - Wait for client connection to complete
  - Concurrent server (using *fork*)
  - Source code
    - *Appendix 1*
    - *Lab2018/tcpcliserv/tcpserv01.c*
- *str_echo* function
  - Read a line and echo the line
  - Source code
    - *Appendix 2*
    - *Lab2018/lib/str_echo.c*

Program flow in Client (1)

- *main* (in *tcpcliserv/tcpcli01.c*)
  - Create socket, fill in Internet socket address structure
  - Connect to server
  - Source code
    - *Appendix 3*
    - *Lab2018/tcpcliserv/tcpcli01.c*
- *str_cli* function
  - Read a line, write to server
  - Read echoed line from server, write to standard output
  - Return to *main* when end-of-file or error.
  - Source code
    - *Appendix 4*
    - *Lab2018/lib/str_cli.c*

Normal Termination

- Zombie process (defunct)
  - process state Z for zombie
- When child process terminates, OS sends SIGCHLD signal to the parent. If there is no action (i.e. wait for the child process) for the signal in the parent, the child process becomes zombie.
- When a process terminates, its child processes are handed over to *init* process (pid 1) as child processes. *init* will clean up the zombies.
- Major problem for long running servers.

Goal

- Catch SIGCHILD signal when forking child processes
- Handle interrupted system calls when we catch signals
- A correctly coded handler for SIGCHILD so zombies are not left around

Posix signal handling 1

- A signal is a notification to a process that an event has occurred
  - Also called software interrupts
  - Usually occur asynchronously
  - Use **man 7 signal** to google all signals in Linux
- Signals can be sent
  - By one process to another process (or to itself), e.g. SIGKILL signal to a process
  - By the kernel to a process (e.g. SIGCHLD signal)
- Commonly used signals
  - SIGALRM, SIGHUP, SIGPIPE, SIGIO, …

Posix signal handling 2

- Every signal has an action associated with it
  - Use *sigaction* function to set a handler function to catch the signal. The function is called when the signal occurs.
    - SIGKILL and SIGSTOP can not be caught
  - Ignore the signal by setting the handler SIG_IGN
    - SIGKILL and SIGSTOP can not be ignored
  - Set the default action by setting the handler SIG_DFL. The default is normally to terminate a process on the receipt of a signal, with certain signals also generating a core image of the process (e.g. *abort* signal)

Posix signal handling 3

- Prototype of signal functions
  - *void (*func)(int)*
  - ```
    struct sigaction {
        void (*sa_handler)(int);
        void (*sa_sigaction)
              (int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags
     }
    ```
  - ```
    int sigaction(int signum,
        const  struct  sigaction  *act,
        struct sigaction *oldact);
    ```

Example code for signal handling 1

- Signal Function

```
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, 4.4BSD */
#endif
        }
        if (sigaction(signo, &act, &oact) < 0)
                return(SIG_ERR);
        return(oact.sa_handler);
}
```

Example code for signal handling 2

- Handler function for SIGCHLD

```
void sig_chld(int signo) {
    pid_t pid;
    int stat;
    pid = wait(&stat);
    printf("Child %d terminated\n", pid);
    return;
    }
```

- Check the difference between tcpserv01.c and tcpserv03.c (appendix 5)

```c
#include    "unp.h"
#include <stdio.h>
#define NOTDEF 1

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
    pid_t               childpid;
    socklen_t           clilen;
    struct sockaddr_in  cliaddr, servaddr;
    char                addbuf[10];

    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);

        printf("new client joined: %s, port %d\n",
               Inet_ntop(AF_INET, &cliaddr.sin_addr, &addbuf, 10),
               ntohs(cliaddr.sin_port));


        childpid = Fork();
        if (childpid==0) {/* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        else { /*parent process*/
            printf("new child process created to handle the new client:
                PID=%d\n",childpid);
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

```c
//
//  str_echo.c
//
//
//  Created by Yawen Chen on 26/07/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#include    "unp.h"

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

    for ( ; ; ) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
            return;     /* connection closed by other end */

        Writen(sockfd, line, n);
    }
}

```

```c
#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);
}
```

```c
//
//  str_cli.c
//
//
//  Created by Yawen Chen on 26/07/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#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);
    }
}

```

```c
#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);

    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 */
    }
}

```