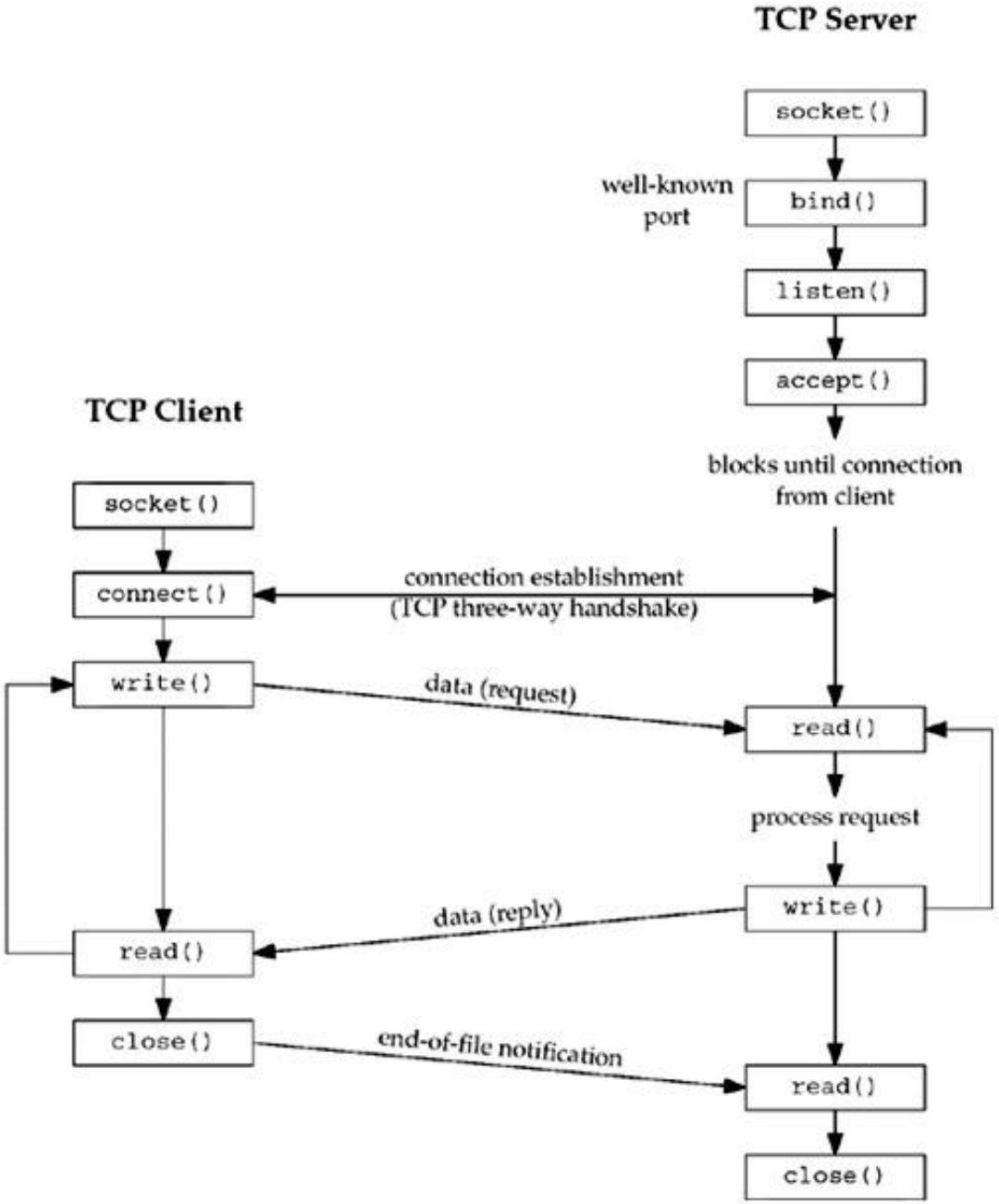
Elementary TCP Sockets

- *socket* function
- *connect* function
- *bind* function
- *listen* function
- *accept* function
- *fork* and *exec* functions
- Concurrent servers
- *close* function
- *getsockname* and *getpeername* functions

Figure 4.1. Socket functions for elementary TCP client/server.



socket Function

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
                    returns: nonnegative descriptor if OK, -1 on error
```

Family	Description
AF_INET	IPv4
AF_INET6	IPv6
AF_LOCAL	Unix domain protocols ~ IPC
AF_ROUTE	Routing sockets ~ appls and kernel
AF_KEY	Key socket

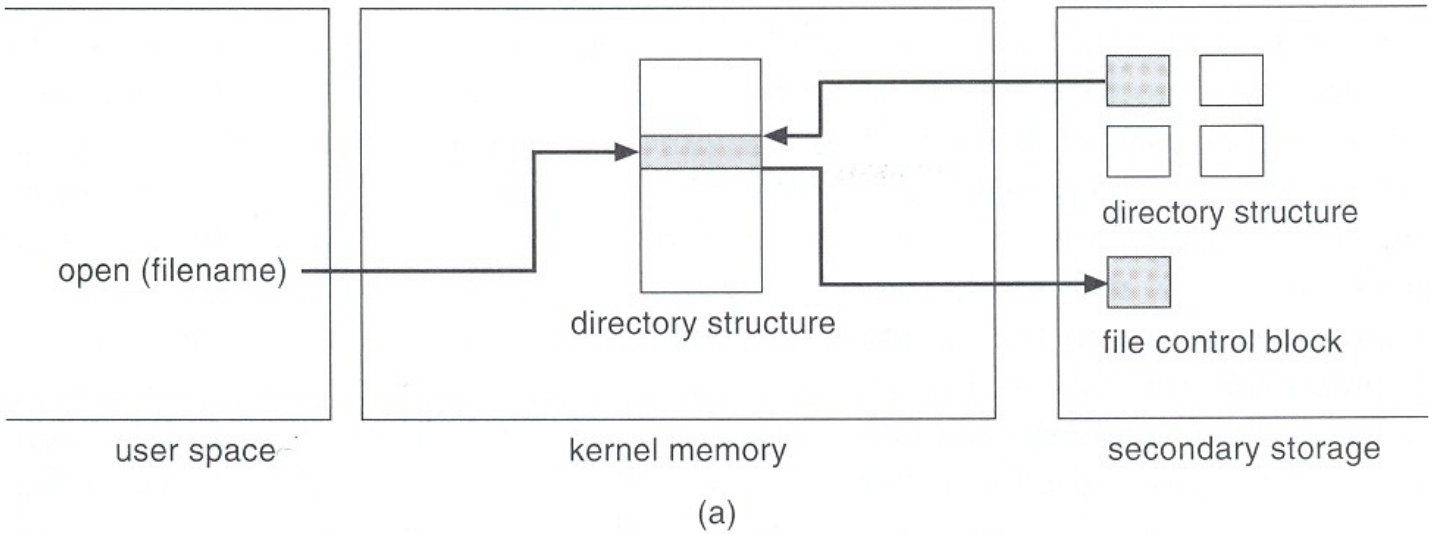
Type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket
SOCK_PACKET	datalink (Linux)

Note that not all combinations of family and type are valid.

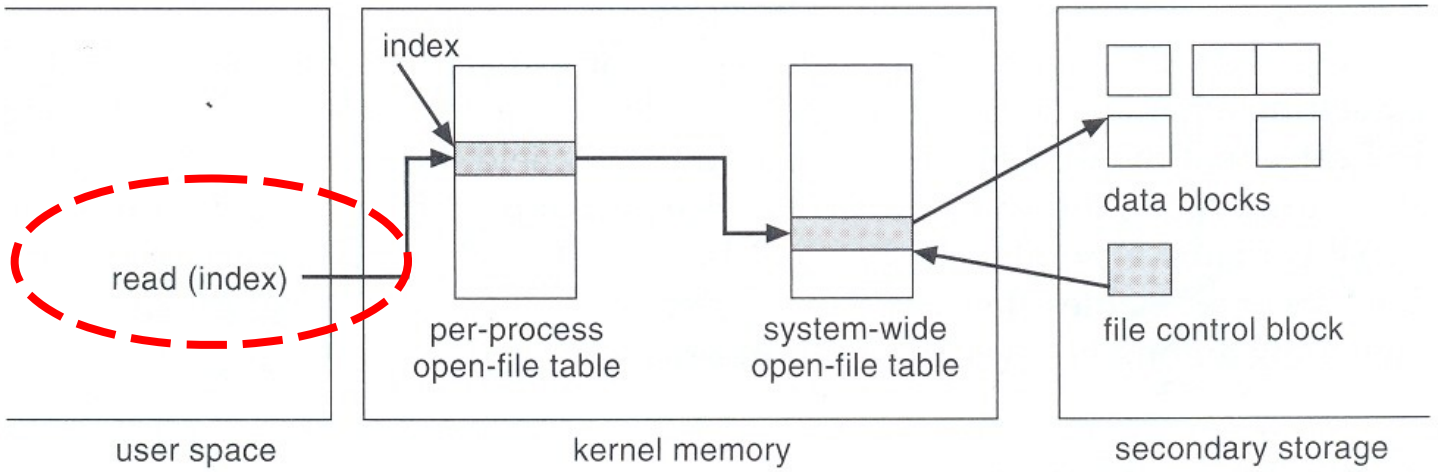
Figure 4.5. Combinations of *family* and *type* for the `socket` function.

	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

What is a socket/file descriptor?



(a)



(b)

connect Function: Three-Way Handshake

#include <sys/socket.h>
int connect (int *sockfd*, const struct sockaddr **servaddr*, socklen_t *addrlen*);
returns: 0 of OK, -1 on error
(If connect fails, the SYN_SENT socket is no longer usable.)

bind before connect : The kernel chooses the source IP, if necessary, and an ephemeral port (for the client).

Hard error: RST received in response to client TCP's SYN (server not running) returns ECONNREFUSED

Soft error: 1. no response to client TCP's SYN, retx SYN, timeout after 75 sec (in 4.4BSD), returns ETIMEOUT
2. ICMP dest unreachable received in response to client TCP's SYN (may be due to transient routing problem), retx SYN, timeout after 75 sec, returns EHOSTUNREACH

bind Function
assigning a local protocol address to a sock

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
                                                    returns: 0 if OK, -1 on error
```

Usually servers bind themselves to their well-known ports. However, RPC servers let kernel choose ephemeral ports which are then registered with the *RPC port mapper*.

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

bind Function (cont.)

For a host to provide Web servers to **multiple organizations**:

Method A: **Aliased IP addresses**

1. Alias multiple IP addresses to a single interface (*ifconfig*).
2. Each server process binds to the IP addr for its organization.

(Demultiplexing to a given server process is done by kernel.)

Method B: **Wildcard IP address**

1. A single server binds to the wildcard IP addr.
2. The server calls *getsockname* to obtain dest IP from the client.
3. The server handles the client request based on the dest IP.

listen Function

converting an unconnected socket into a passive socket

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
                    returns: 0 if OK,-1 on error
```

For a given listening socket, the kernel maintains two queues and *backlog*, which may be overridden by the environment variable LISTENQ, specifies the maximum value for the *sum* of both queues. However, different implementations interpret differently.

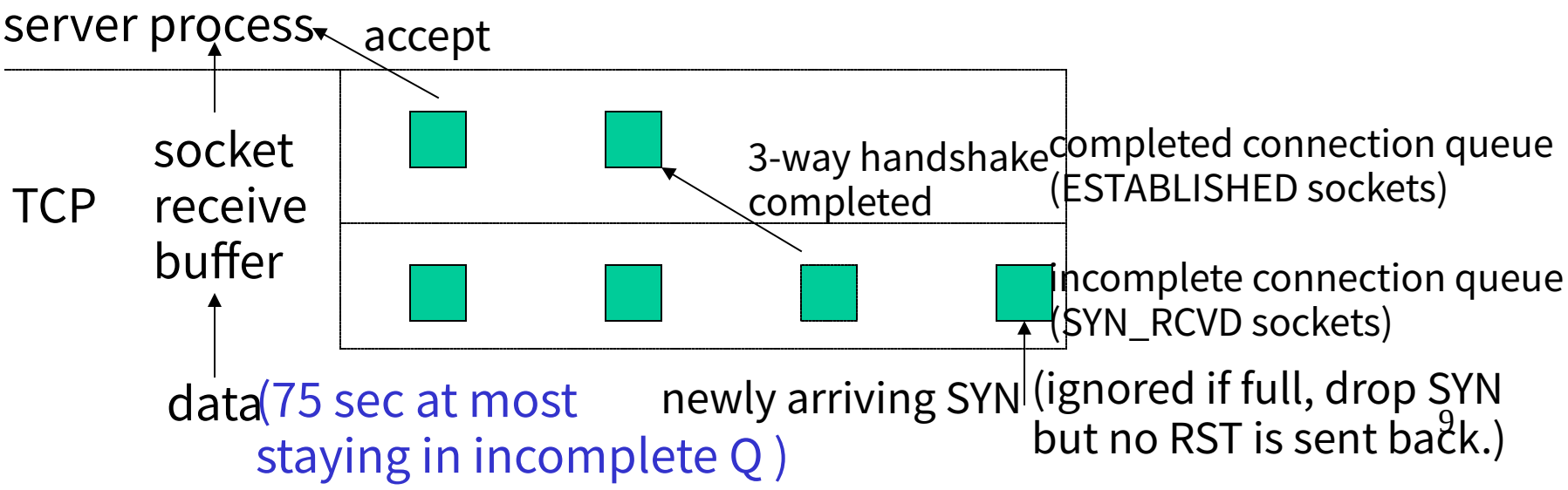
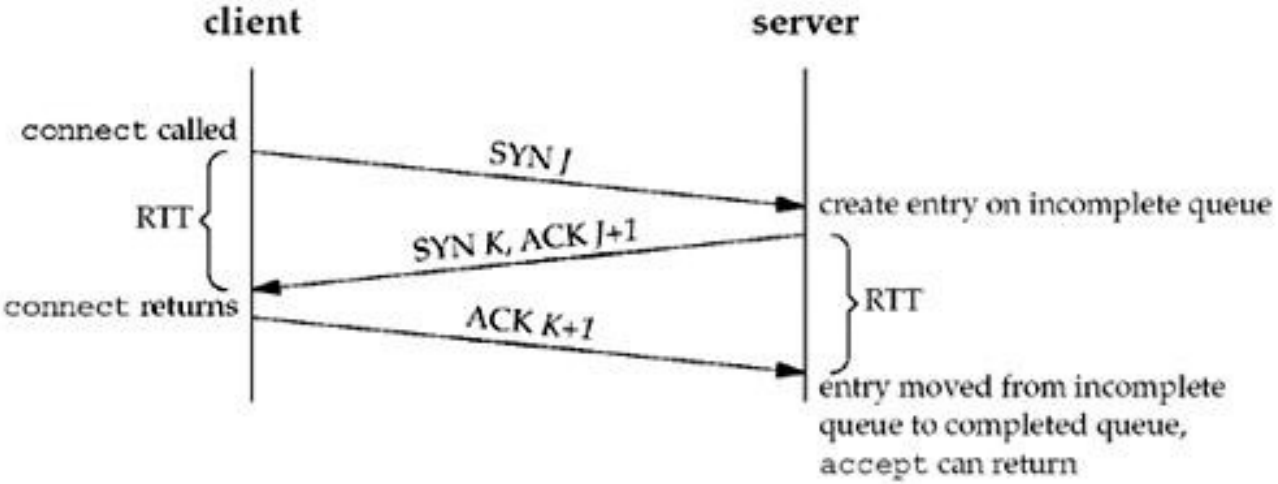


Figure 4.8. TCP three-way handshake and the two queues for a listening socket.



listen Function (cont.)

SYN Flooding : a type of attack due to "backlog"

1. Send SYNs at a high rate to the victim to fill up the incomplete connection queue for one or more TCP ports.
2. The source IP address of each SYN is set to a random number (IP spoofing)
3. Legitimate SYNs are not queued, i.e. ignored.

Thus, the "backlog" should specify the max number of
completed connections for a listening socket.

But there must be a limit like backlog for incomplete Q;
otherwise, kernel memory will be exhausted leading

accept Function

returning the new descriptor of next completed connection

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
                                    returns: nonnegative descriptor if OK, -1 on error
```

sockfd: listening socket
accept returns: connected socket
cliaddr, addrlen: value-result arguments

To examine and display client IP address and port:

```
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));
```

fork and *exec* Functions

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

returns: 0 in child, process ID of child in parent, -1 on error

(*called-once-return-twice*)

Inheritance: All descriptors open in the parent before fork, e.g. the *connected socket*, are shared with the child.

Two typical uses of fork:

1. to make another copy (e.g. network servers)
2. to execute another program (e.g. shells)

fork and *exec* Functions (cont.)

```
int execl (const char *pathname, const char *arg0, .... /* (char *) 0 */);
int execv (const char *pathname, char *const argv[ ]);
int execle (const char *pathname, const char *arg0, ... /* (char *)0,
        char *const envp[ ] */);
int execve (const char *pathname, char *const argv[ ], char *const envp[ ]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */);
int execvp (const char *filename, char *const argv[ ]);
```

All return: -1 on error, no return on success

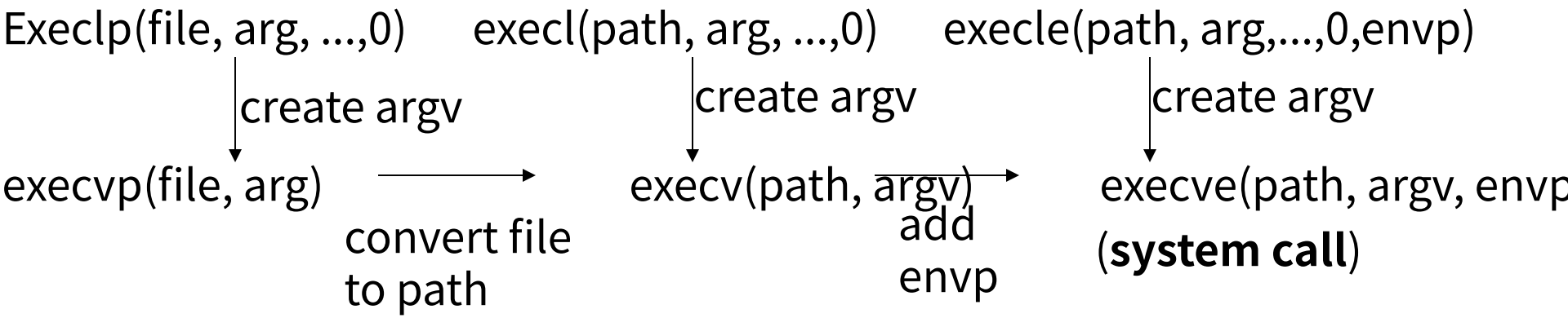


Figure 4.11 Daytime server that prints client IP address and port

intro/daytimetcpsrv1.c

```
 1 #include    "unp.h" 2
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr, cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     servaddr.sin_port = htons(13);  /* daytime server */

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

17     Listen(listenfd, LISTENQ);

18     for ( ; ; ) {
```

Iterative Servers

```
10      time_t ticks;

11      listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12      bzero(&servaddr, sizeof(servaddr));
13      servaddr.sin_family = AF_INET;
14      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15      servaddr.sin_port = htons(13);  /* daytime server */

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

17      Listen(listenfd, LISTENQ);

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

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

27          Close(connfd);
28      }
29  }
```

Concurrent Servers: Outline

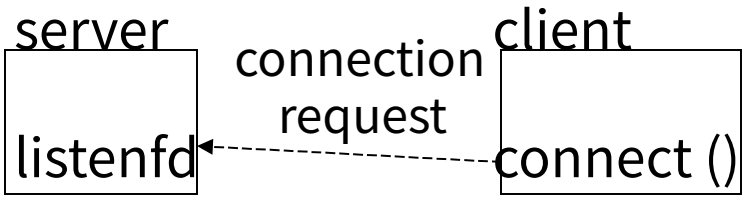
```
pid_t   pid;
int     listenfd, connfd;

listenfd = Socket (...);
        /* fill in socket_in{} with server's well-known port */
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 this client */
                exit (0);           /* child terminates */
        }
        Close (connfd);           /* parent closes connected socket */
}
```

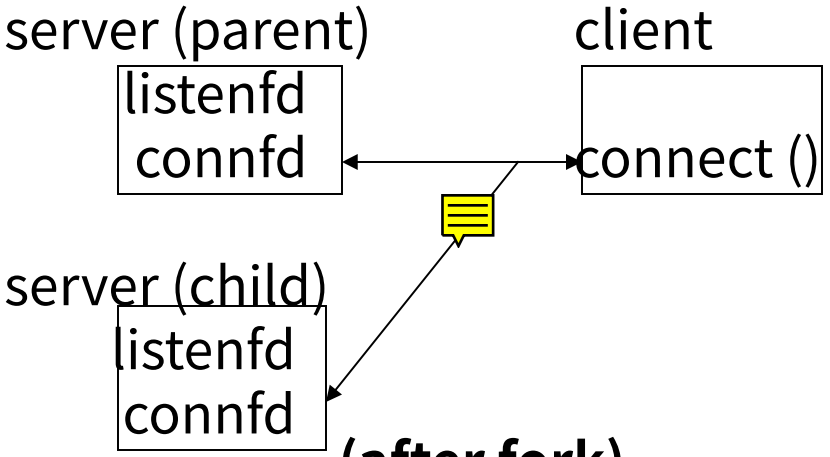
Concurrent Servers: Shared Descriptors

Why doesn’t the *close* of *connfd* by the parent terminate its connection with the client? ---- Every file or socket has a **reference count** in the *file table*

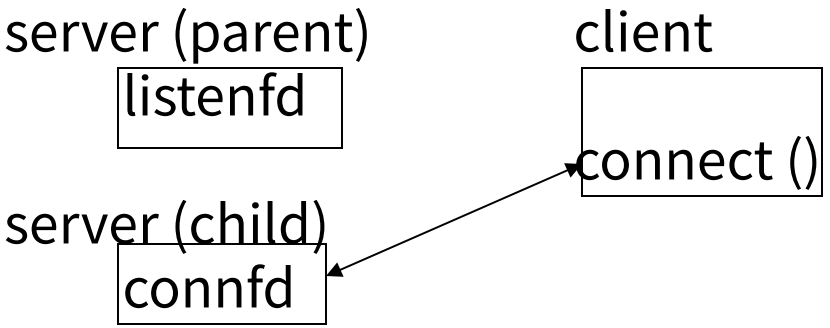


(before accept)

(after accept)



(after fork)



(duplicated sockets closed)

close Function

```
#include <unistd.h>
int close (int sockfd);
        returns: 0 if OK, -1 on error;
```

Default action of close: (may be changed by SO_LINGER socket option)

1. mark closed and return immediately
2. TCP tries to send queued data
3. normal 4-packet termination sequence

What if the parent does not close connected socket?

1. Run out of descriptors
2. None of the client connections will be terminated.
   (reference count remains at 1, termination sequence cannot occur)

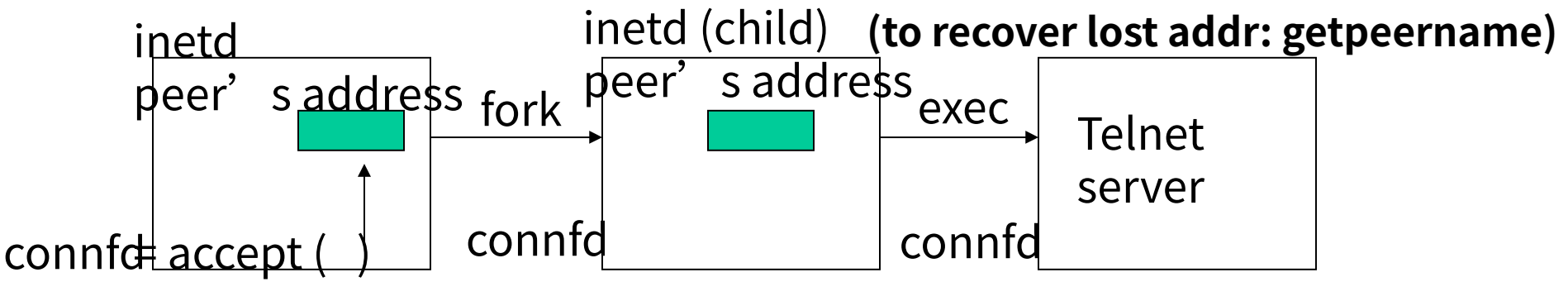
getsockname and *getpeername* Functions

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

returns: 0 if OK, -1 on error

Where are these two functions needed?

1. TCP client that does not call bind but need to know local IP and assigned po
2. To obtain address family of a socket
3. TCP server that binds the wildcard IP, but needs to know assigned local IP
4. A *exec*ed server that needs to obtain the identity of the client



To know *connfd* after exec: 1. Always setting descriptors 0, 1, 2 to be the connected socket before exec. 2. Pass it as a command-line arg.

Figure 4.19 Return the address family of a socket.

lib/sockfd_to_family.c

```
 1 #include    "unp.h"
 2 int
 3 sockfd_to_family(int sockfd)
 4 {
 5     struct sockaddr_storage ss;
 6     socklen_t len;

 7     len = sizeof(ss);
 8     if (getsockname(sockfd, (SA *) &ss, &len) < 0)
 9         return (-1);
10     return (ss.ss_family);
11 }
```