Network Programming:
Ch. 4 Elementary TCP Sockets

Li-Hsing Yen

NYCU

Ver. 1.0.0

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

Some materials in these slides are taken from Prof. Ying-Dar Lin with his permission of usage

socket Function

normally 0 (except for raw sockets)

```
#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 (TCP)
SOCK_DGRAM	datagram socket (UDP)
SOCK_RAW	raw socket
SOCK_PACKET	datalink (Linux)

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

`connect` Function

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

- Used by a TCP client to establish a connection with a TCP server
- It initiates TCP’s three-way handshake
- The socket address structure must contain the IP address and port number of the server
- The client does not have to call `bind`
  - The kernel chooses the source IP, if necessary, and an ephemeral port (for the client).

Possible Errors

retransmits

- no response to client TCP's SYN, retx SYN, timeout after 75 sec (in 4.4 BSD), returns ETIMEOUT → Time Out
- **Hard error**: RST received in response to client TCP's SYN (server program not running)
  - returns ECONNREFUSED → Connection Refused
- **Soft error**: ICMP dest. unreachable received in response to client TCP's SYN (maybe due to transient routing problem), retx SYN, timeout after 75 sec, returns EHOSTUNREACH or ENETUNREACH

EHOSTUNREACH → Host Unreachable

ENETUNREACH → Network Unreachable

`bind` Function (server only)

generic

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

- Assign a local protocol address to a socket
  - With the Internet protocol, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.
- With TCP, calling bind lets us specify a port number, an IP address, both, or neither.

Possible Address/Port Combinations

Process specifies		
IP address	port	Result
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

Wildcard IP address in IPv4

```
struct sockaddr_in          servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Wildcard IP address in IPv6

```
struct sockaddr_in6        serv;
serv.sin6_addr = in6addr_any;
```

const defined in `<netinet/in.h>`

`bind` Function (cont.)

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

Method A: **Aliased IP addresses (one server for each IP address)**

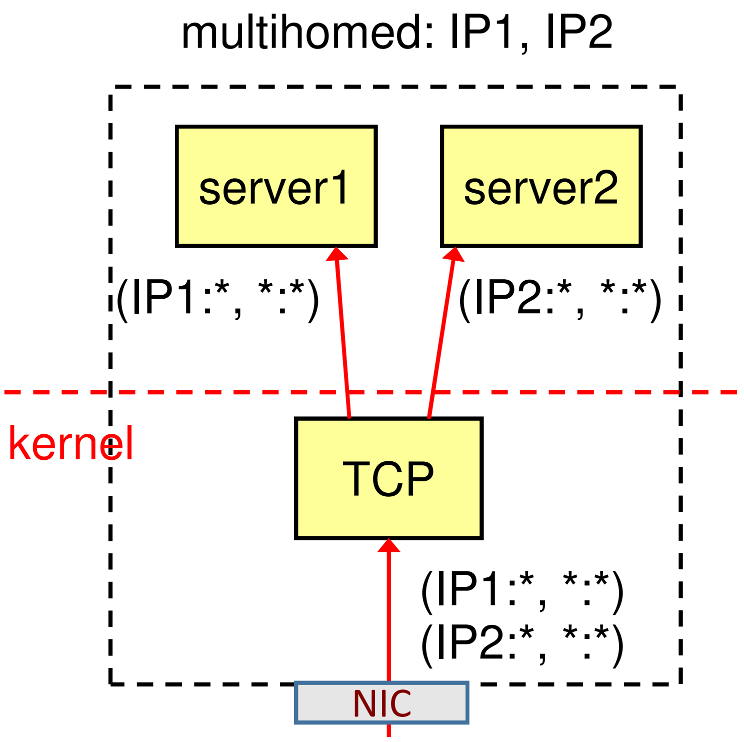
1. Alias multiple IP addresses to a single interface (*ifconfig*).
2. Each server process binds to the IP addr for its organization.

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

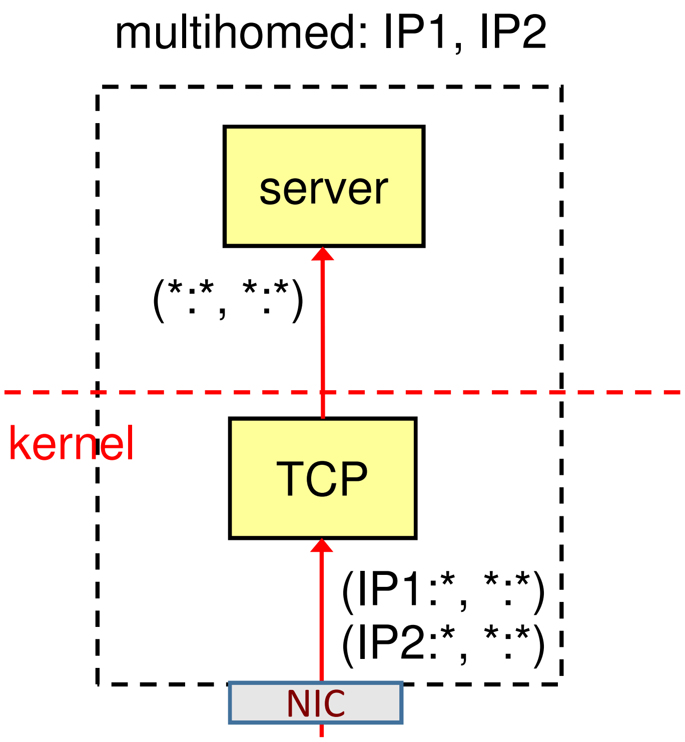
Method B: **Wildcard IP address (single server for all IP addresses)**

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.

Aliased IP addresses



Wildcard IP addresses



listen Function (server only)

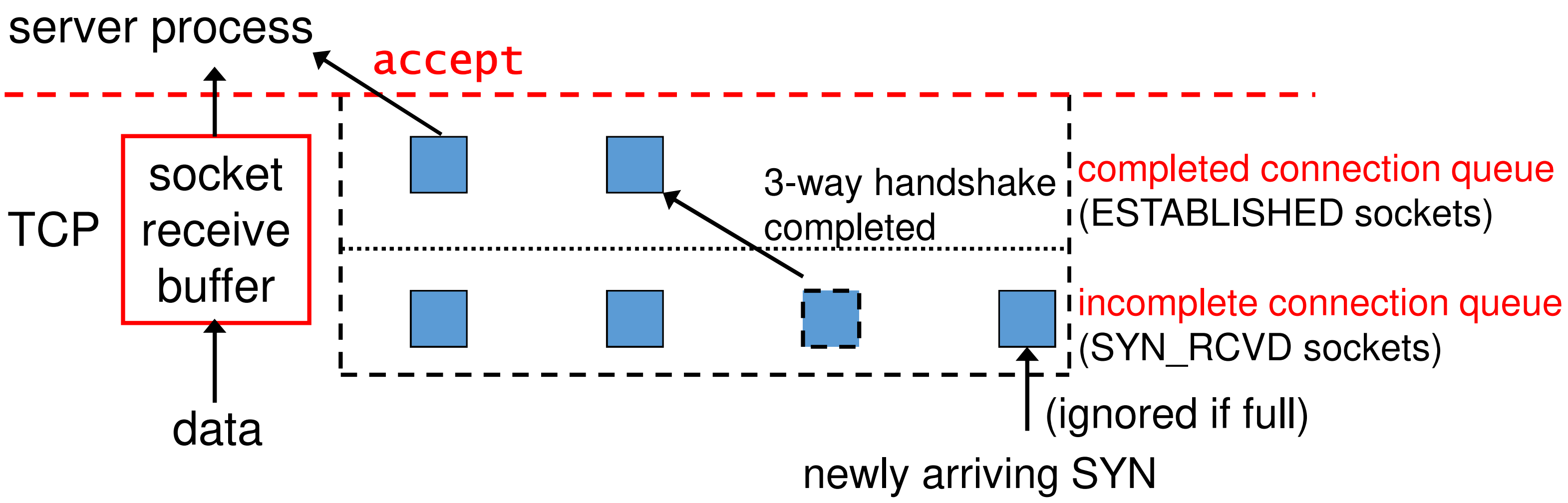
```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
        returns: 0 if OK,-1 on error
```

- called only by a TCP server
- It converts an unconnected socket into a passive socket (listening socket)
  - A socket created by the `socket` function is assumed to be an active socket (a client socket that will issue a `connect`)

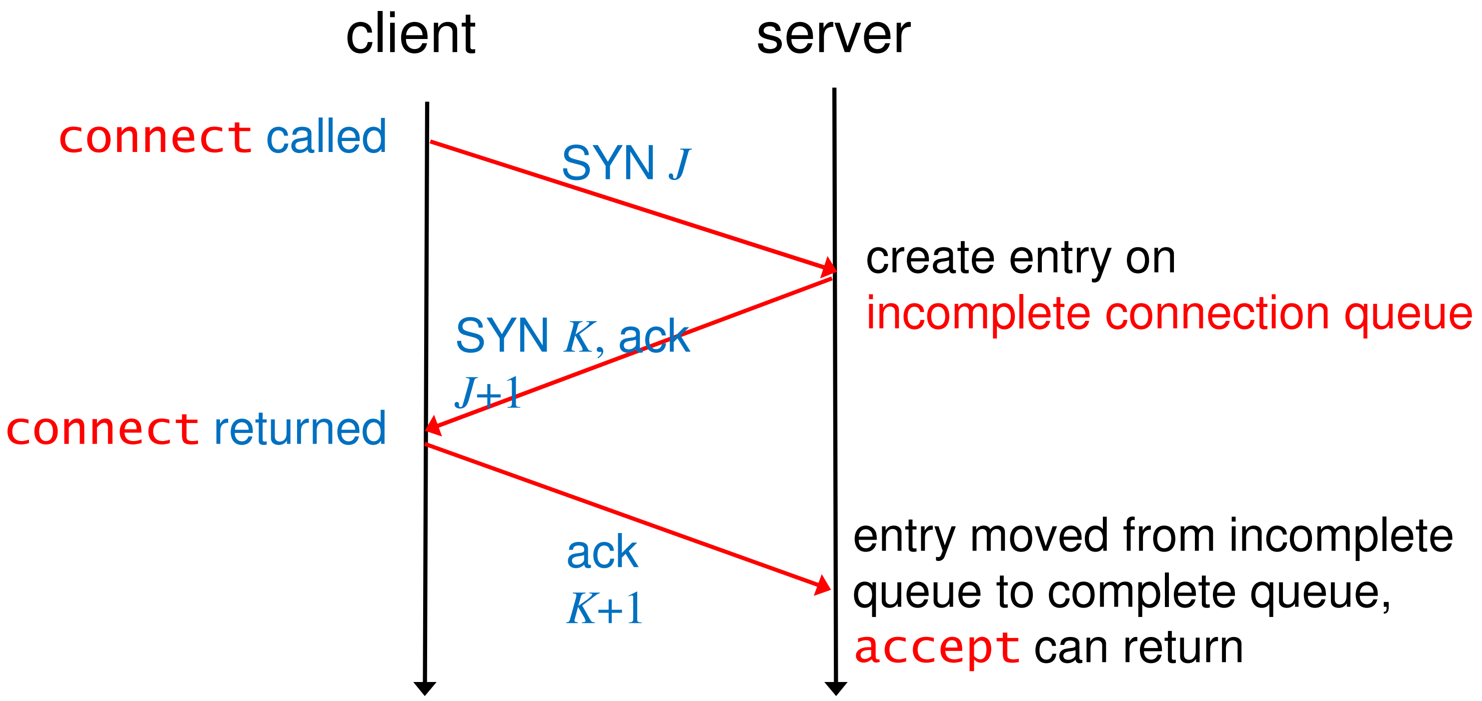
About The *backlog* Parameter

- For a given listening socket, the kernel maintains two queues
- Backlog specifies the max for the sum of both queues

historically



Three-Way Handshake and the Two Queues



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. Other legitimate SYNs are not queued, i.e. ignored.

Let the kernel handle SYN flooding; the “backlog” should only specify the max number of completed connections for a listening socket.

accept Function (server only)

```
#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 fd

cliaddr, *addrlen*: value-result arguments

connected socket
(storage allocated beforehand)

- called by a TCP server
- returns the next completed connection from the front of the completed connection queue (socket is automatically created by kernel)
- If the completed connection queue is empty, the process is put to sleep

Listening Socket vs. Connected Socket

- Listening socket
  - requested by the server program
  - Used to accept incoming connections
- Connected socket
  - created by the kernel for each connected connection (id returned by `accept`)
  - used to communicate with the connected client

Use `cliaddr` to Get Client’s Address

```
int  listenfd, connfd;
socklen_t  len;
struct sockaddr_in  servaddr, cliaddr;
char buff[MAXLINE];
…
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));
```

cast to a pointer to a generic socket

```
struct sockaddr_in {
  uint8_t           sin_len;
  sa_family_t       sin_family;
  in_port_t         sin_port;
  struct in_addr    sin_addr;
  char              sin_zero[8];
};
```

used for subsequent data exchange with this client

fork System Call

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

returns: 0 in child, process ID of child in parent, -1 on error
(*called-once-return-twice*)

- fork is the only way in Unix to create a new process (a child)
- Two typical uses of fork:
  1. to make another copy (e.g. network servers)
  2. to execute another program (e.g. shells)

More on fork

get parent's process id

- child can get the process ID of its parent by getppid
- All descriptors opened in the parent before fork, e.g. the listening and connected sockets, are shared with the child.

with reference count increased by 1

- Normally the child then reads and writes the connected socket and the parent closes the connected socket.

decreasing the ref. count by 1. resource won't be released as long as the ref. count > 0

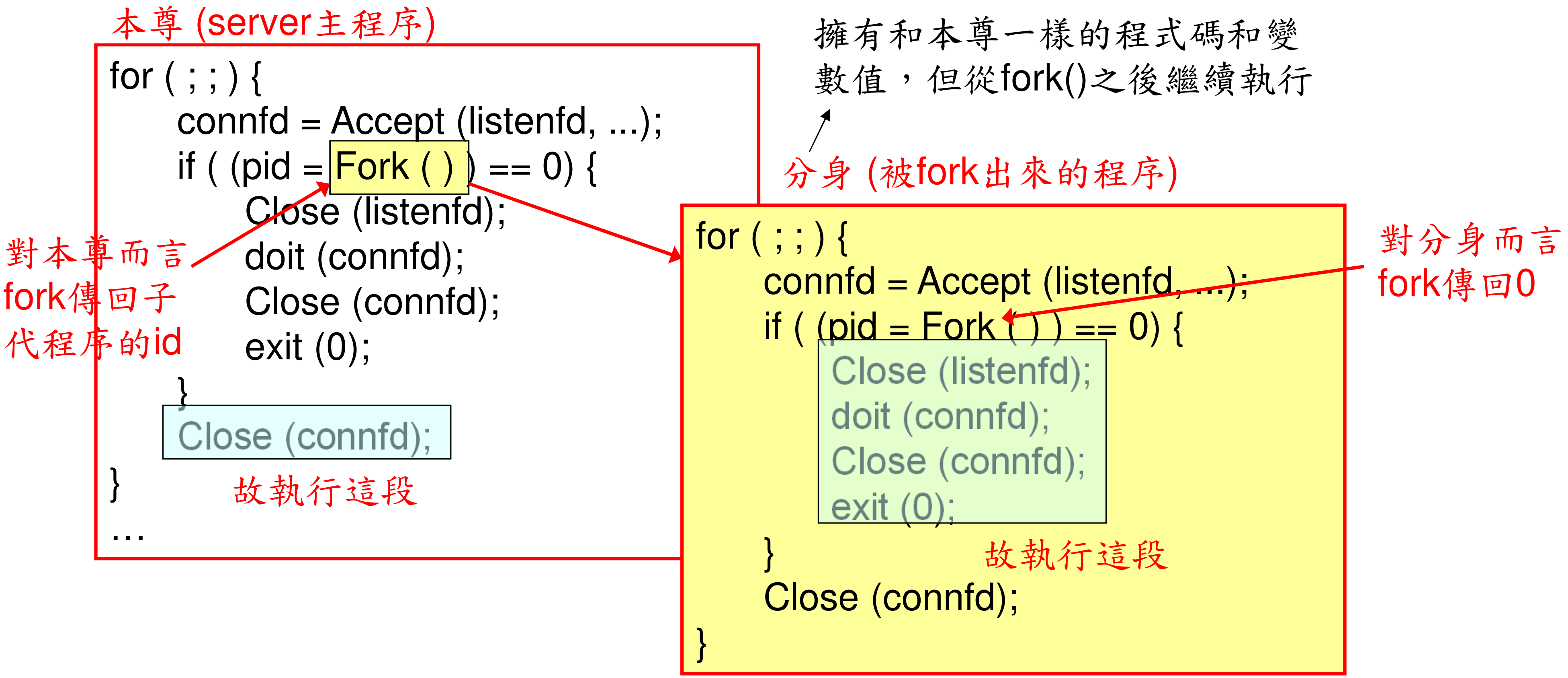
Concurrent Servers: Outline

```
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 */
}
```

fork 的本尊與分身



本尊 (server主程序)

```
for ( ; ; ) {
      connfd = Accept (listenfd, ...);
      if ( (pid = Fork ( ) ) == 0) {
            Close (listenfd);
            doit (connfd);
            Close (connfd);
            exit (0);
      }
      Close (connfd);
}
...
```

對本尊而言
fork傳回子
代程序的id

故執行這段

擁有和本尊一樣的程式碼和變
數值，但從fork()之後繼續執行

分身 (被fork出來的程序)

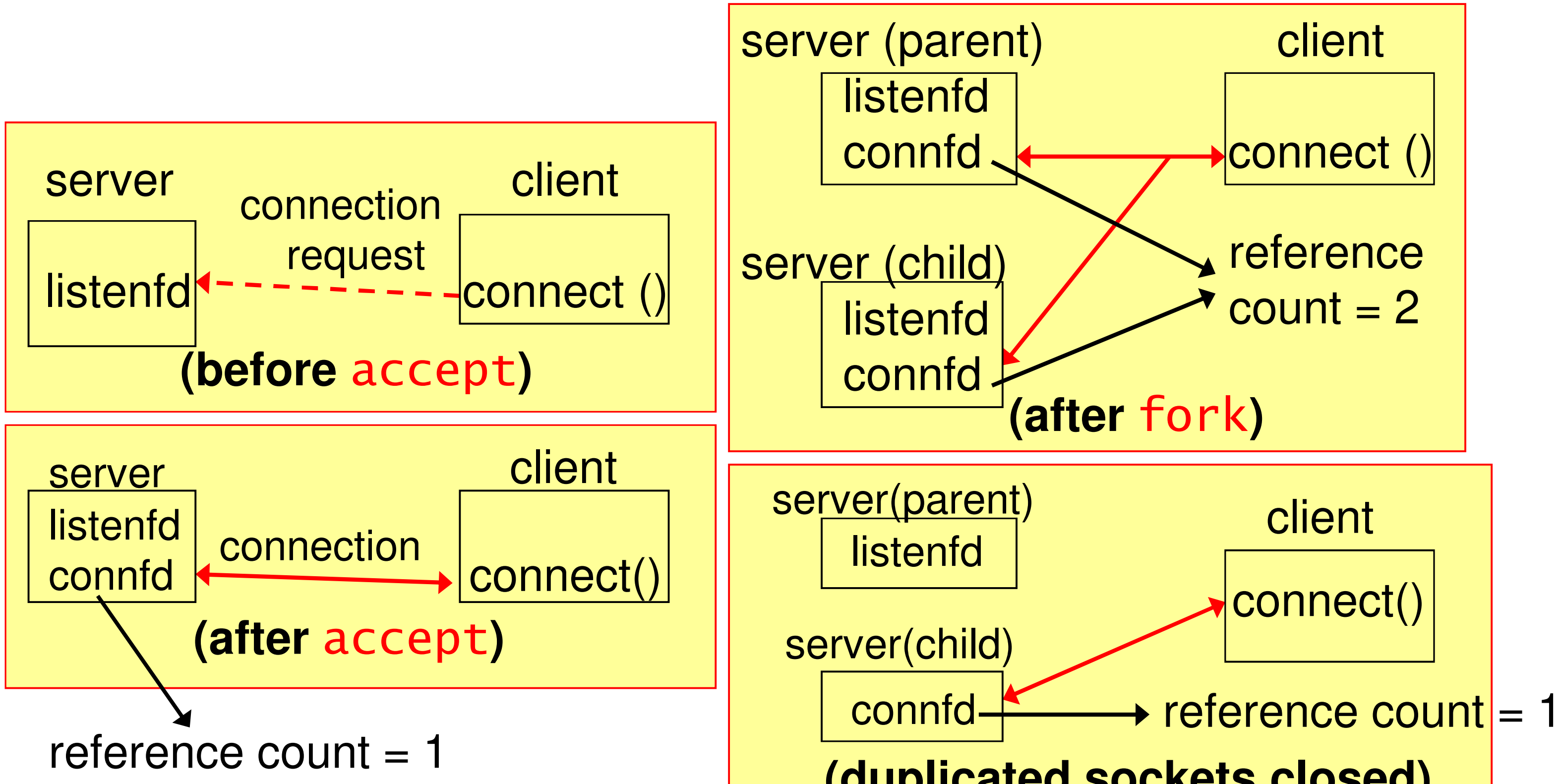
```
for ( ; ; ) {
      connfd = Accept (listenfd, ...);
      if ( (pid = Fork ( ) ) == 0) {
            Close (listenfd);
            doit (connfd);
            Close (connfd);
            exit (0);
      }
      Close (connfd);
}
```

對分身而言
fork傳回0

故執行這段

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*.



close Function

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

Default action of close in the kernel (may be changed by SO_LINGER socket option)

1. mark closed (decreasing the reference count by 1) and return immediately
2. TCP tries to send queued data (if any)
3. If the reference count of the socket is 0, perform normal 4-packet termination sequence

What if the parent does not close connected socket?

```
for ( ; ; ) {
        connfd = Accept (listenfd, ...);
        if ( (pid = Fork ( ) ) == 0) {
                Close (listenfd);
                doit (connfd);
                Close (connfd);
                exit (0);
        }
        Close (connfd);
}
…
```

如果本尊不執行這行，則所有的connected socket就算已被分身處理完畢(close掉)，也會因為reference count不為零而會繼續佔用系統資源

連線不會斷

最後server的系統資源終究會用完而無法再接受新的連線請求

但應用程式結束(exit)時，所開啟的資源會自動被系統釋放掉

getsockname (自己的) and getpeername (對方的) Functions

由sockfd得到socket address結構的內容

```
#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 know its local IP addr. and assigned port
2. To obtain address family of a socket ← connected socket
3. TCP server that binds the wildcard IP, but needs to know assigned local IP
4. An *exec*ed server that needs to obtain the identity of the client

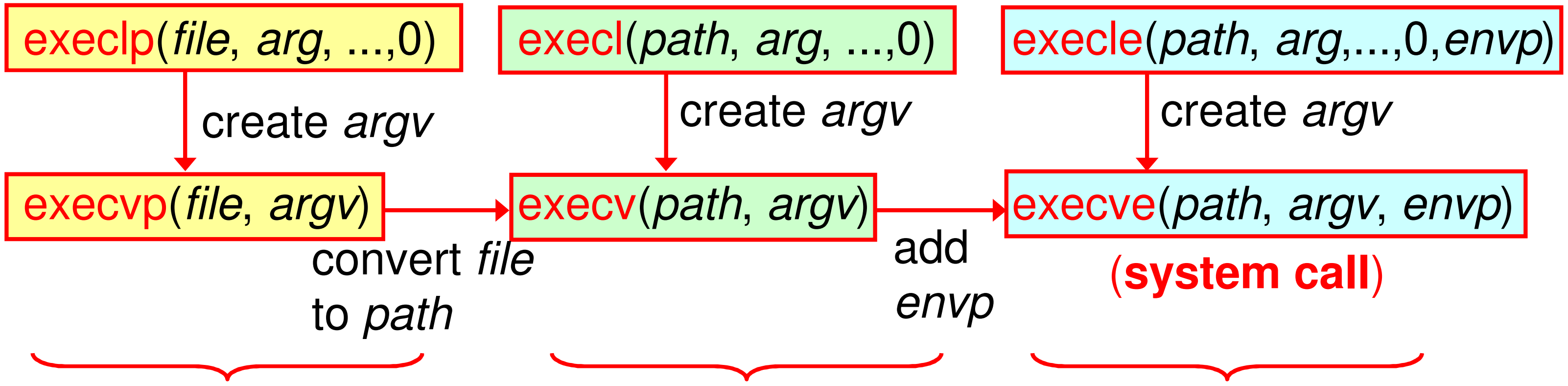
← 藉由傳遞sockfd達成

exec Function

- replaces the current process image with the new program file (載入新的程式碼覆蓋原本的)　 (用在child程序中)

```
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
```

The exec Family



execlp(*file*, *arg*, ...,0)

create *argv*

execvp(*file*, *argv*)

convert *file* to *path*

execl(*path*, *arg*, ...,0)

create *argv*

execv(*path*, *argv*)

add *envp*

execle(*path*, *arg*,...,0,*envp*)

create *argv*

execve(*path*, *argv*, *envp*)

(**system call**)

設定載入的程式檔名但
不指定程式檔的路徑

設定程式檔的路徑
但不設定環境變數

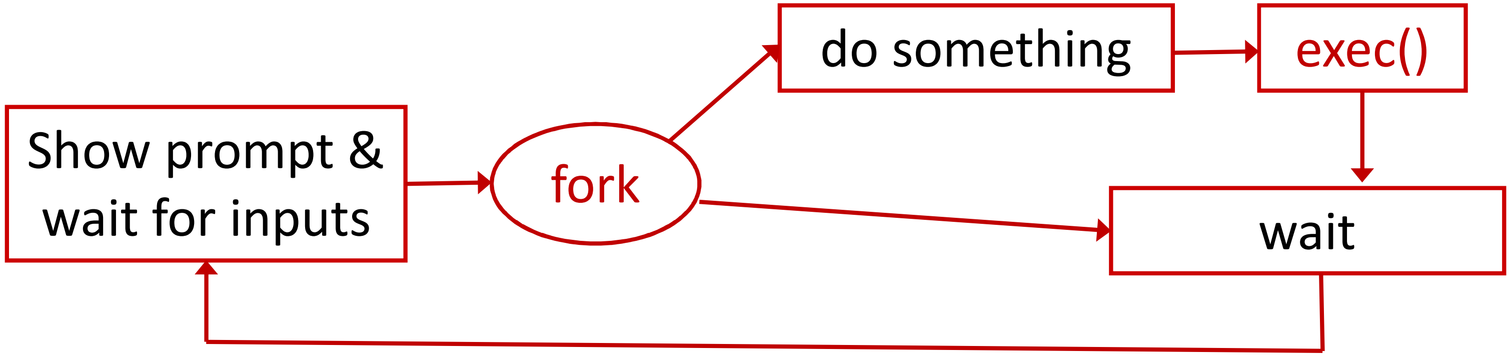
設定程式檔的路徑
及環境變數

Example Using **exec()**

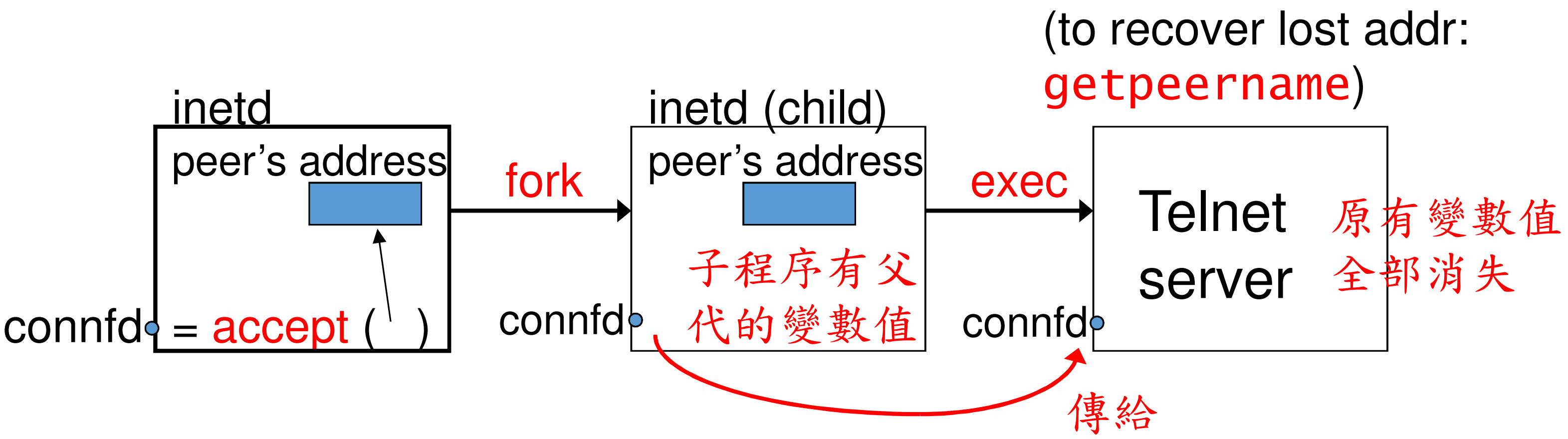
```
if ((rc = fork()) < 0) { // fork failed; exit
   fprintf(stderr, "fork failed\n");
   exit(1);
} else if (rc == 0) { // child (new process)
   printf("hello, I am child (pid:%d)\n", (int) getpid());
   char *myargs[3];
   myargs[0] = strdup("wc"); // program: "wc" (word count)
   myargs[1] = strdup("p3.c"); // argument: file to count
   myargs[2] = NULL; // marks end of array
   execvp(myargs[0], myargs); // runs word count
   printf("this shouldn’t print out");
} else { // parent goes down this path (main)
   int rc_wait = wait(NULL);
   printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait,
         (int) getpid());
}
```

Why Two Steps?

- the separation of fork() and exec() is essential in building a UNIX shell
- it lets the shell run code after the call to fork() but before the call to exec()



Example of inetd Spawning a Server



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 argument in exec