Network Computing & Programming

Lecture 3

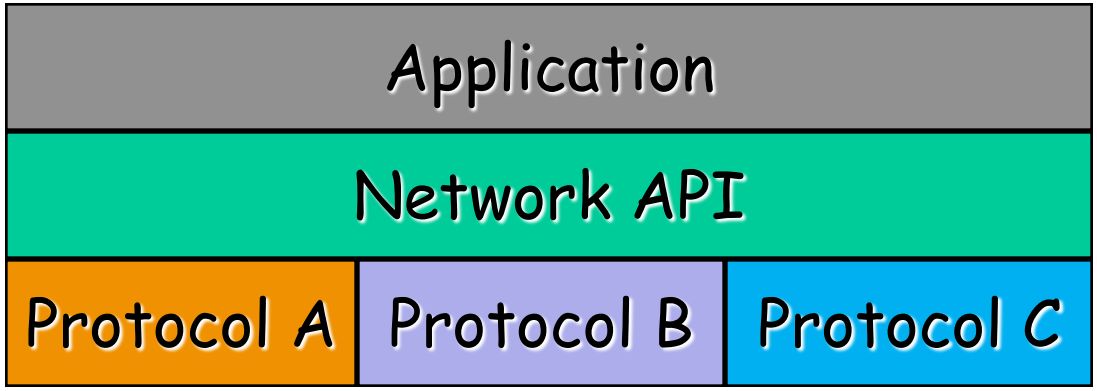
Socket Programming – First Stage

2019-2020-1

Slides are modified from Xiang(Shaun) Zhang

Network Application Programming Interface (API)

- Services that provide the interface between application and protocol software (often by the operating system).



Network API wish list

- **Generic Programming Interface**
  - Support multiple communication protocol suites (families).
  - Address (endpoint) representation independence.
  - Provide special services for Client and Server.
  - Support for message oriented and connection oriented communication.
  - Work with existing I/O services (when this makes sense).
  - Operating System independence.

TCP/IP

- TCP/IP does not include an API definition.

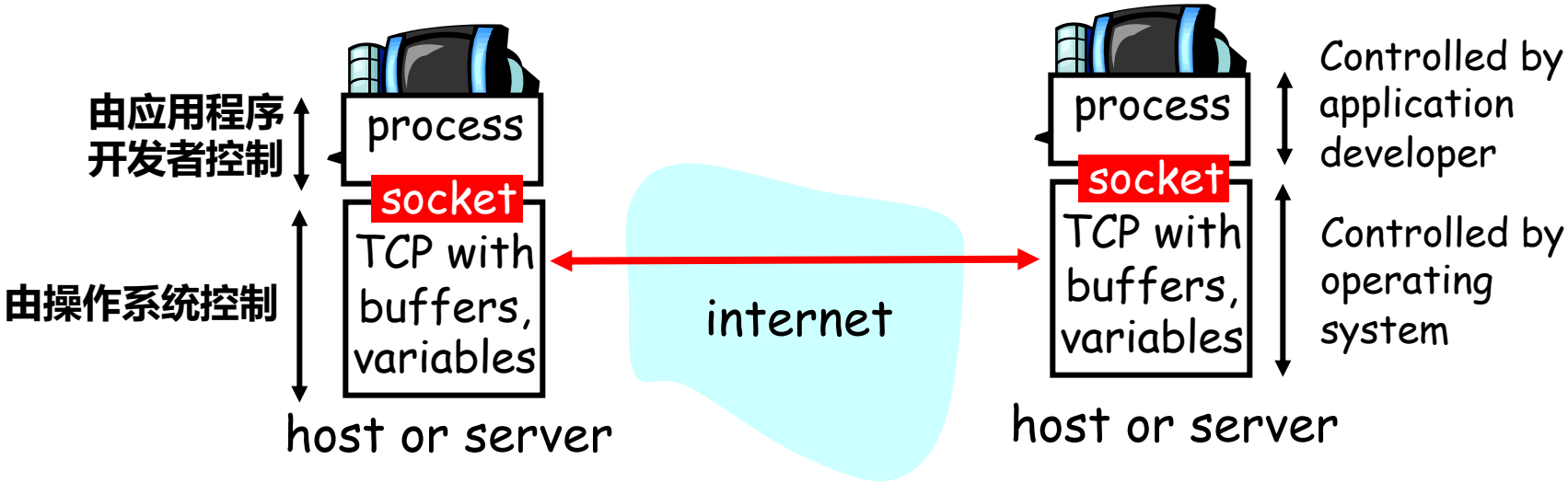
- There are a variety of APIs for use with TCP/IP:
  - Berkeley sockets
    - API for Internet sockets and Unix domain sockets (used for inter-process communication (IPC))
  - XTI(X/Open Transport Interface)/TLI (Transport Layer Interface), Unix System V
  - Winsock, Windows
  - MacTCP, Mac
  - Linux/POSIX Socket, basically Berkeley sockets
    - Portable Operating System Interface, POSIX

Functions needed:

- Specify local and remote communication endpoints
- Initiate a connection
- Wait for incoming connection
- Send and receive data
- Terminate a connection gracefully
- Error handling

Berkeley Sockets

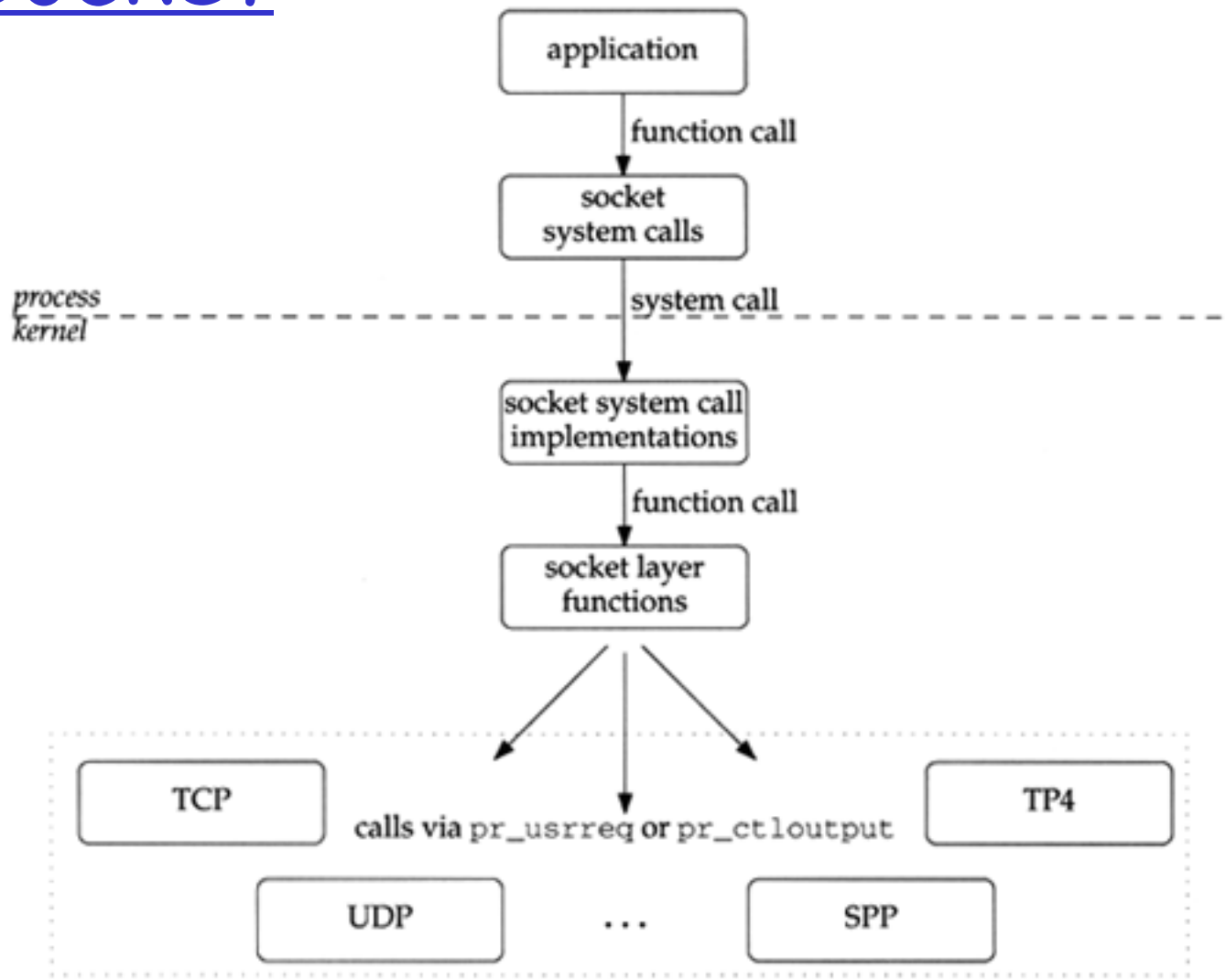
- Generic:
  - Support for multiple protocol families.
  - Address representation independence
- Uses existing I/O programming interface as much as possible.



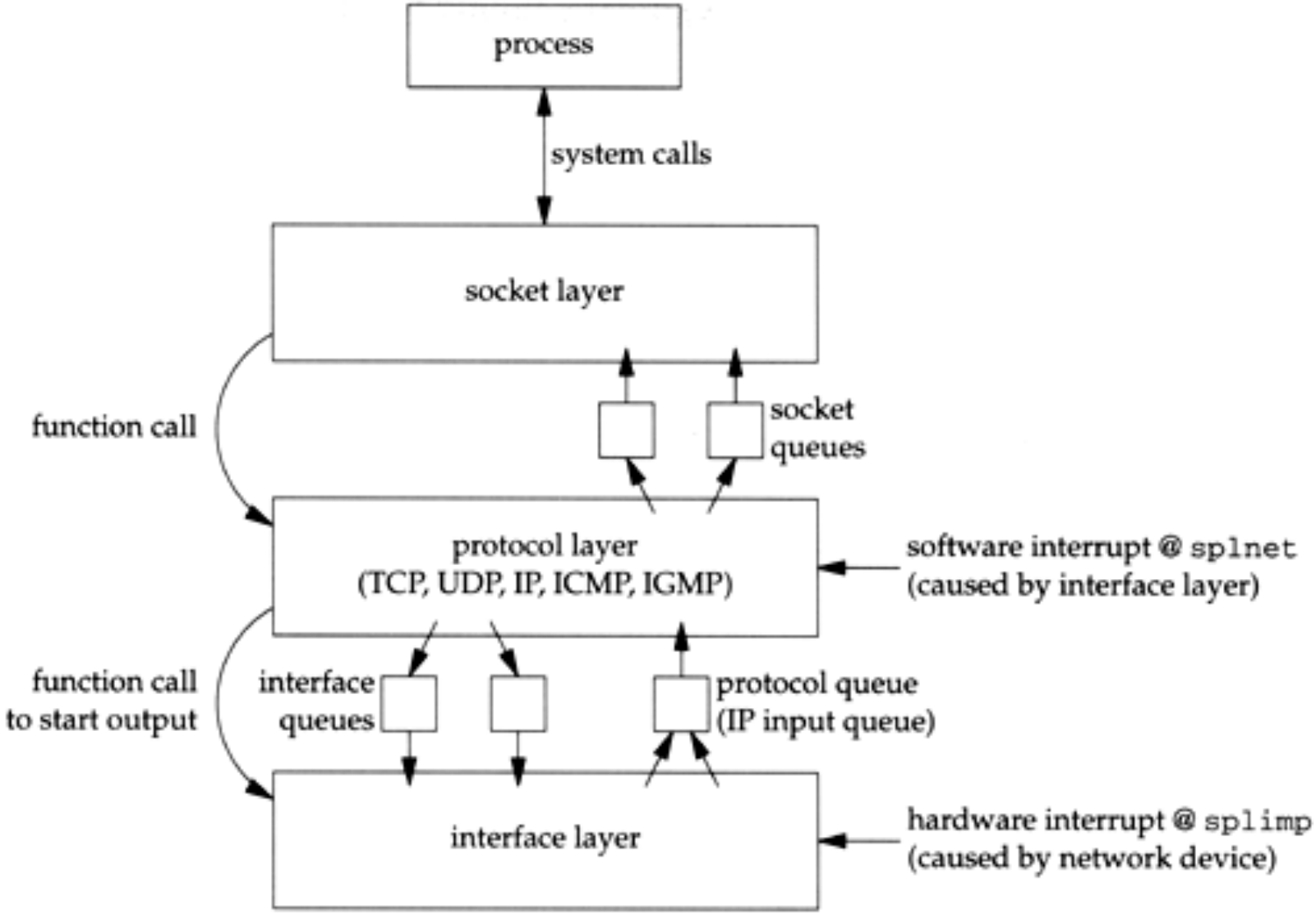
Socket

- A socket is an abstract representation of a communication endpoint.
- Sockets work with Unix I/O services just like files, pipes & FIFOs.
- Sockets (obviously) have special needs:
  - Establishing a connection
  - Specifying communication endpoint addresses

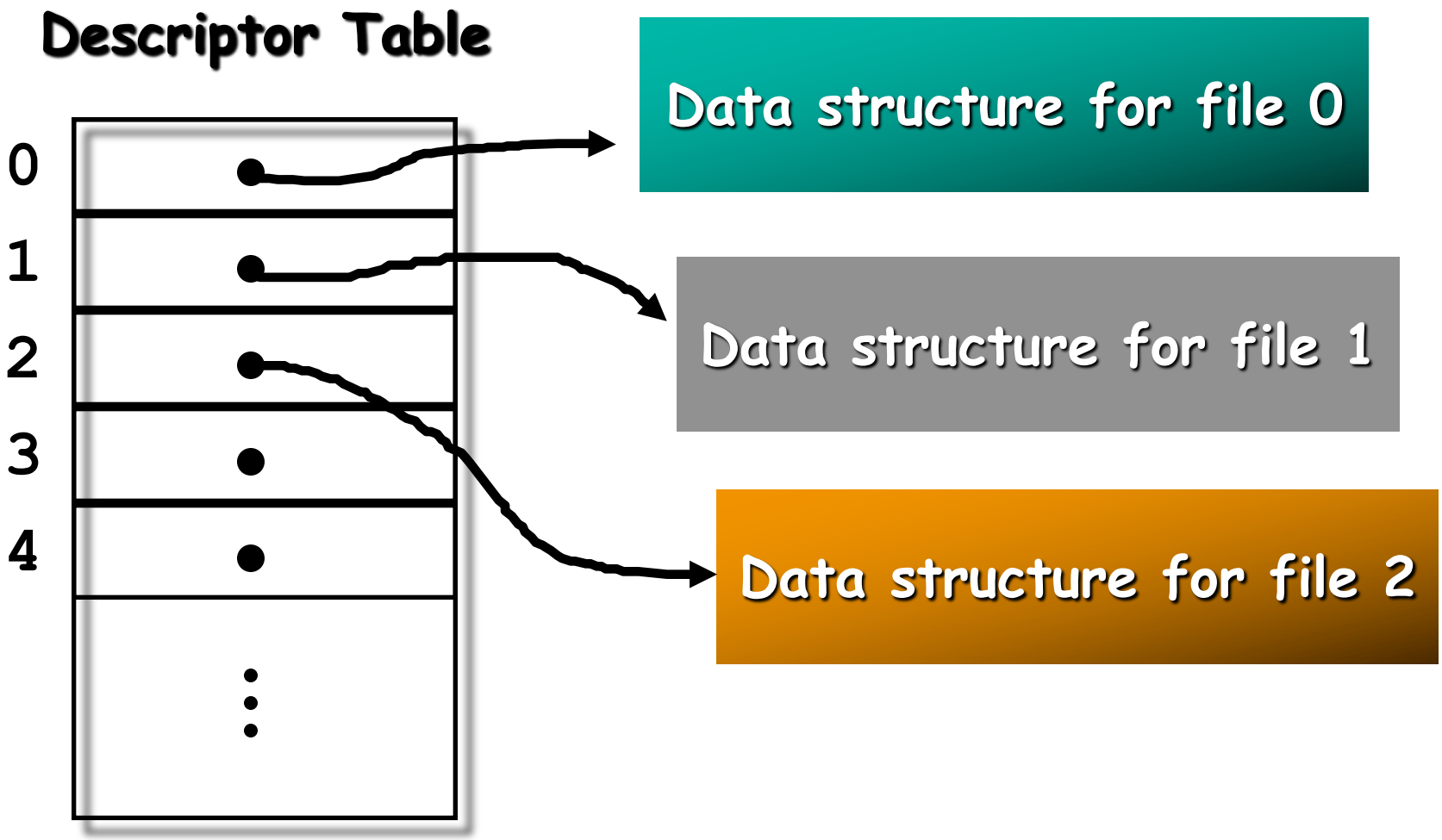
Socket


application
function call
socket
system calls
process
kernel
system call
socket system call
implementations
function call
socket layer
functions
TCP
calls via pr_usrreq or pr_ctloutput
TP4
UDP
. . .
SPP

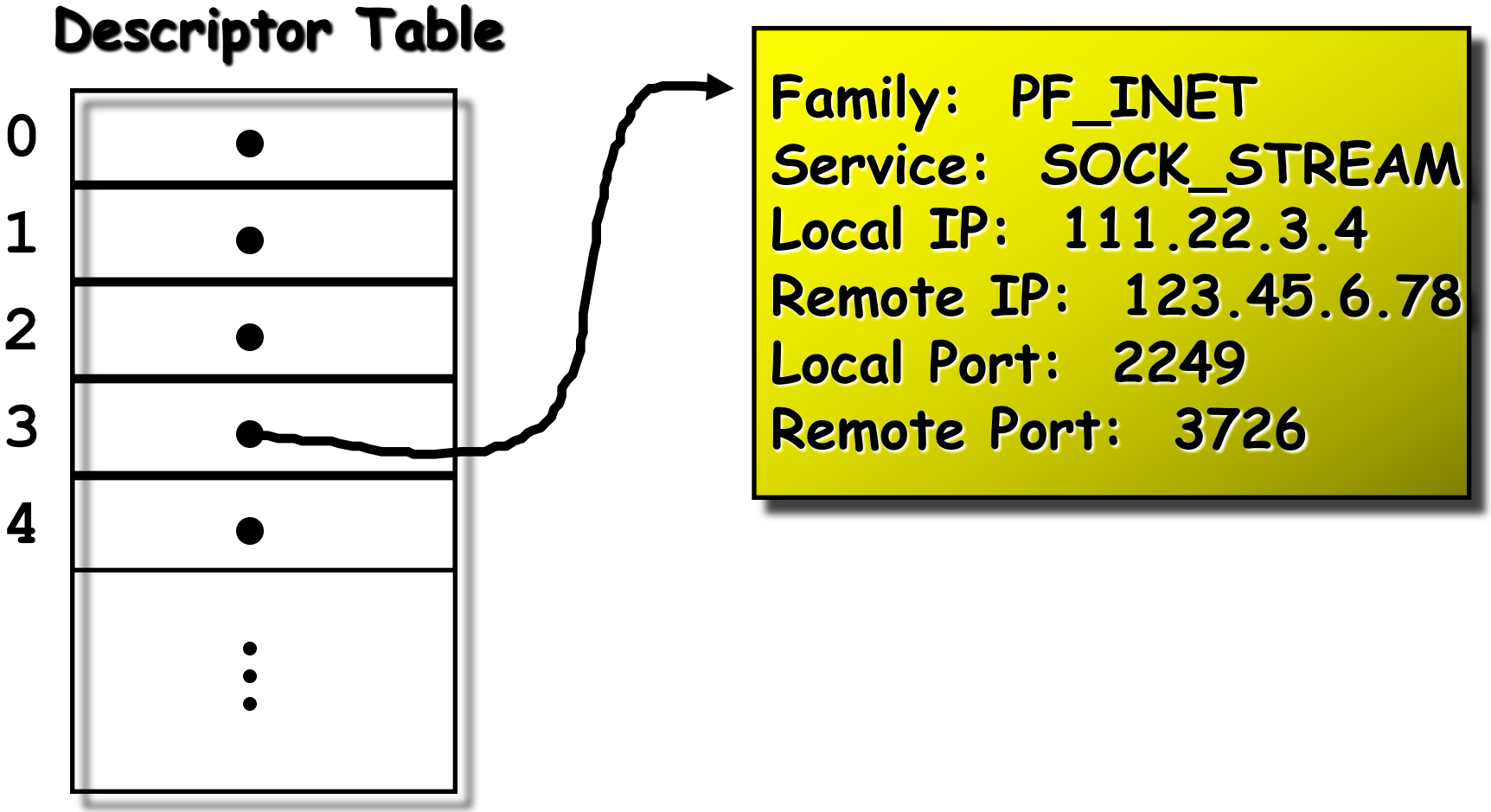
Socket


process
system calls
socket layer
function call
socket
queues
protocol layer
(TCP, UDP, IP, ICMP, IGMP)
software interrupt @ splnet
(caused by interface layer)
function call
to start output
interface
queues
protocol queue
(IP input queue)
interface layer
hardware interrupt @ splimp
(caused by network device)

Unix Descriptor Table



Socket Descriptor Data Structure



Creating a Socket

```
int socket(int family,int type,int proto);
```

- **family** specifies the protocol family (**PF_INET** for TCP/IP).
  - **AF_INET** vs. **PF_INET**
- **type** specifies the type of service (**SOCK_STREAM**, **SOCK_DGRAM**).
- **protocol** specifies the specific protocol (usually 0, which means *the default*).

socket()

- `socket()` returns a socket descriptor (small integer) or -1 on error.

- `socket()` allocates resources needed for a communication endpoint
  - but it does not deal with endpoint addressing.

Specifying an Endpoint Address

- Sockets API is generic.
- There must be a generic way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.
- Other protocol suites (families) may use other schemes.

Necessary Background Information: POSIX data types

POSIX (Portable Operating System Interface of UNIX)

`int8_t`	signed 8bit int
`uint8_t`	unsigned 8 bit int
`int16_t`	signed 16 bit int
`uint16_t`	unsigned 16 bit int
`int32_t`	signed 32 bit int
`uint32_t`	unsigned 32 bit int

`u_char, u_short, u_int, u_long` Obsolete

More POSIX data types

`sa_family_t`	address family
`socklen_t`	length of struct
`in_addr_t`	IPv4 address
`in_port_t`	IP port number

Generic socket addresses

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

Used by kernel

- **sa_family** specifies the address type.
- **sa_data** specifies the address value.

sockaddr

- An address that will allow me to use sockets to communicate with my family.
- address type **AF_FAMILY**
- address values:

Daughter	**1**
Wife	**2**
Mom	**3**
Dad	**4**
Sister	**5**
Brother	**6**

AF_FAMILY

- Initializing a sockaddr structure to point to Daughter:

```
struct sockaddr mary;

mary.sa_family  = AF_FAMILY;
mary.sa_data[0] = 1;
```

AF_INET

- For AF_FAMILY we only needed 1 byte to specify the address.

- For AF_INET we need:
  - 16 bit port number
  - 32 bit IP address

IPv4 only!

struct sockaddr_in (IPv4)

```
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];
};
struct in_addr {
  in_addr_t        s_addr;
};
```

Length of structure (16)

AF_INET

16 bit Port number

Make structure 16 bytes

32 bit IPv4 address

struct sockaddr_in (IPv6)

```
struct sockaddr_in6 {
  uint8_t          sin6_len;
  sa_family_t      sin6_family;
  in_port_t        sin6_port;
  uint32_t         sin6_flowinfo;
  struct in6_addr  sin6_addr;
  uint32_t         sin6_scope_id;
};
struct in6_addr {
   uint8_t         s6_addr[16];
};
```

Length of structure (28)

AF_INET6

Port number

Flow label

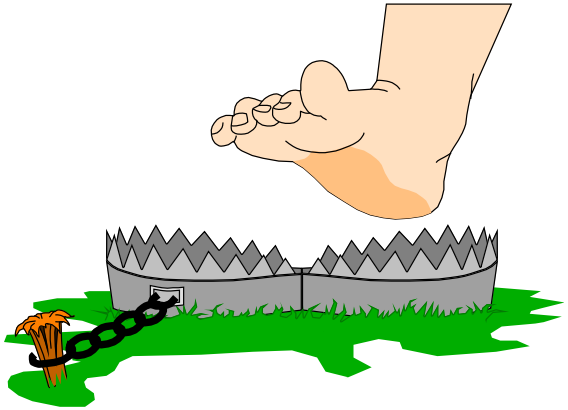
Scope of address

128 bit IPv6 address

Network Byte Order

- All values stored in a **`sockaddr_in`** must be in network byte order.
  - **`sin_port`** a TCP/IP port number.
  - **`sin_addr`** an IP address.

!!! Common Mistake !!!



Network Byte Order Functions

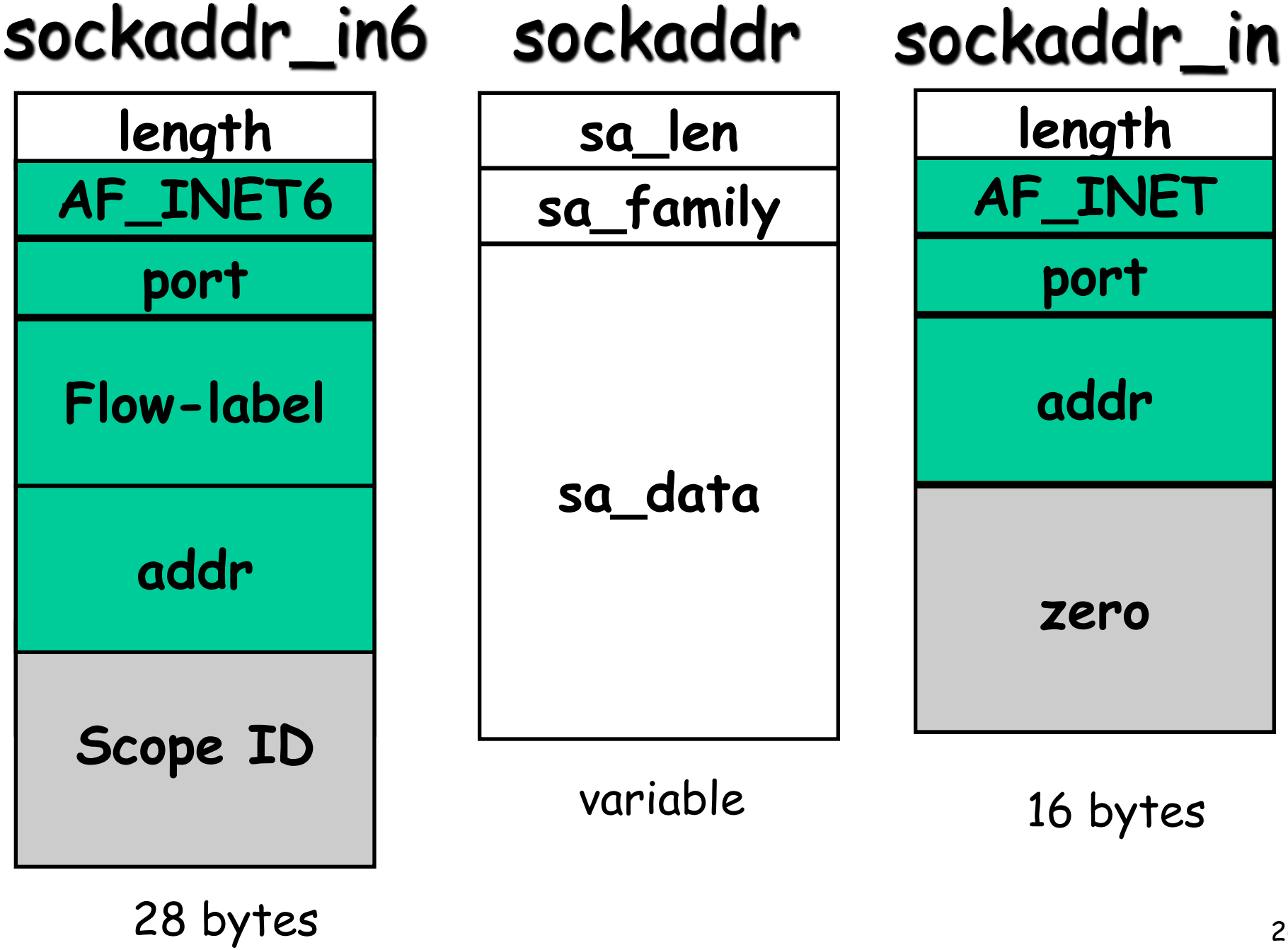
'h' : host byte order
'n' : network byte order
's' : short (16bit)
'l' : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint_16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```

TCP/IP Addresses

- We don't need to deal with **sockaddr** structures since we will only deal with a real protocol family.
- We can use **sockaddr_in** structures.
- BUT: The C functions that make up the sockets API expect structures of type **sockaddr**.


sockaddr_in6
length
AF_INET6
port
Flow-label
addr
Scope ID
28 bytes
sockaddr
sa_len
sa_family
sa_data
variable
sockaddr_in
length
AF_INET
port
addr
zero
16 bytes

Assigning An Address to A Socket

- The **bind()** is used to assign an address to an existing socket.

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

const!

- **bind** returns 0 if successful or -1 on error.

bind()

- Calling **bind()** assigns the address specified by the **sockaddr** structure to the socket descriptor.

- You can give **bind()** a **sockaddr_in** structure:

```
bind( mysock,
      (struct sockaddr*) &myaddr,
      sizeof(myaddr) );
```

bind() Example

```
int mysock,err;
struct sockaddr_in myaddr;

  mysock = socket(PF_INET,SOCK_STREAM,0);
  myaddr.sin_family = AF_INET;
  myaddr.sin_port = htons( portnum );
  myaddr.sin_addr = htonl( ipaddress);

  err=bind(mysock, (sockaddr *) &myaddr,
                 sizeof(myaddr));
```

Uses for `bind()`

- There are a number of uses for **`bind()`**:
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the O.S. to assign *any available* port number.
- Clients typically don't care what port they are assigned.
- When you call bind, you can tell it to assign you any available port

What Is My IP Address

- How can you find out what your IP address is so you can tell **`bind()`** ?

- There is no realistic way for you to know the right IP address to give bind()
  - what if the computer has multiple network interfaces?

- specify the IP address as: **`INADDR_ANY`**, this tells the OS to take care of things.

IPv4-only Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

- Convert **ASCII dotted-decimal IP address** to **network byte ordered 32(IPv4) bit value**.
- Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

- Convert **network byte ordered value** to **ASCII dotted-decimal** (a string).

IPv4&IPv6 Address Conversion

```
int inet_pton(int, const char *, void *);
```

- (address_family, string_ptr, address_ptr)
- Convert IP address string **presentation** to **network byte ordered 32 (IPv4) or 128(IPv6) bit value**.
- Returns 1 on success, -1 on failure, 0 on invalid input

```
char *inet_ntop(int,const void*,char*,size_t);
```

- (address_family, address_ptr, string_ptr, length)
- Convert **network byte ordered value** to IP address string **presentation**.
  - x:x:x:x:x:x:x:x or x:x:x:x:x:x:a.b.c.d

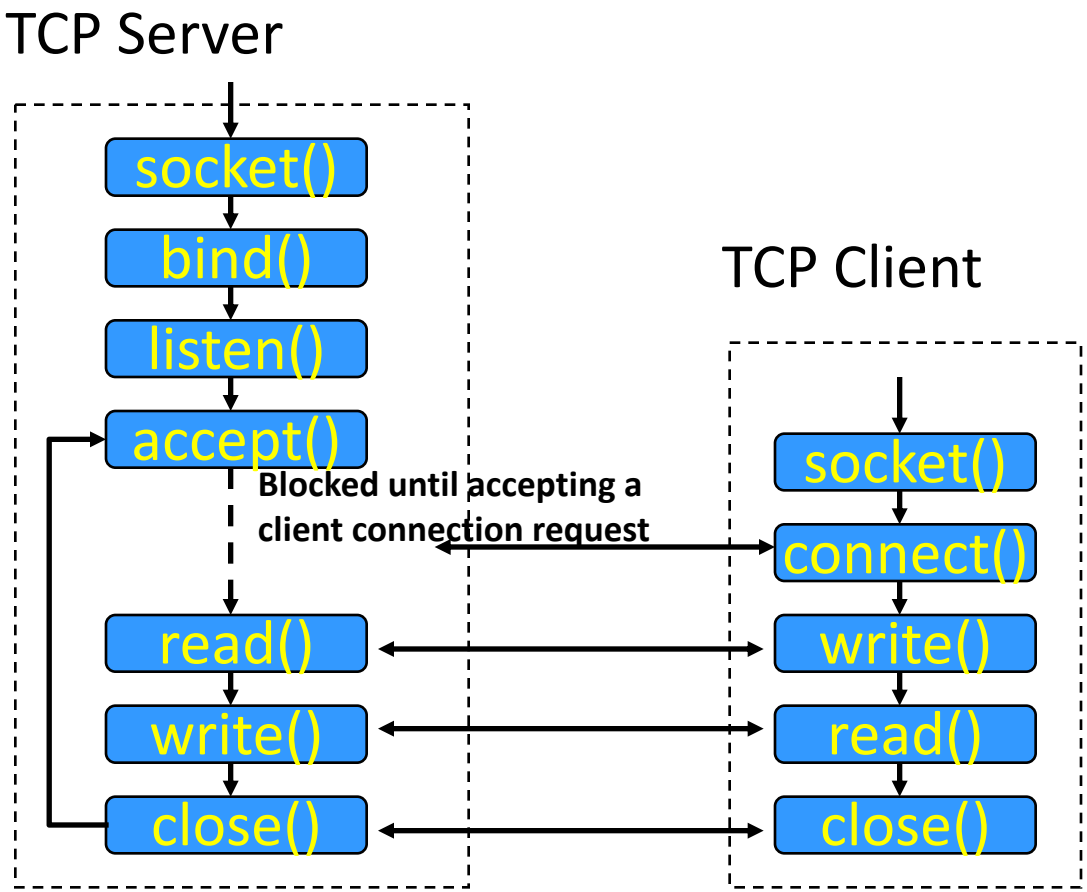
Other Socket Functions

- General Use
  - `read()`
  - `write()`
  - `close()`

- Connection-oriented (TCP)
  - `connect()`
  - `listen()`
  - `accept()`

- Connectionless (UDP)
  - `send()`
  - `recv()`

Basic TCP Socket Programming

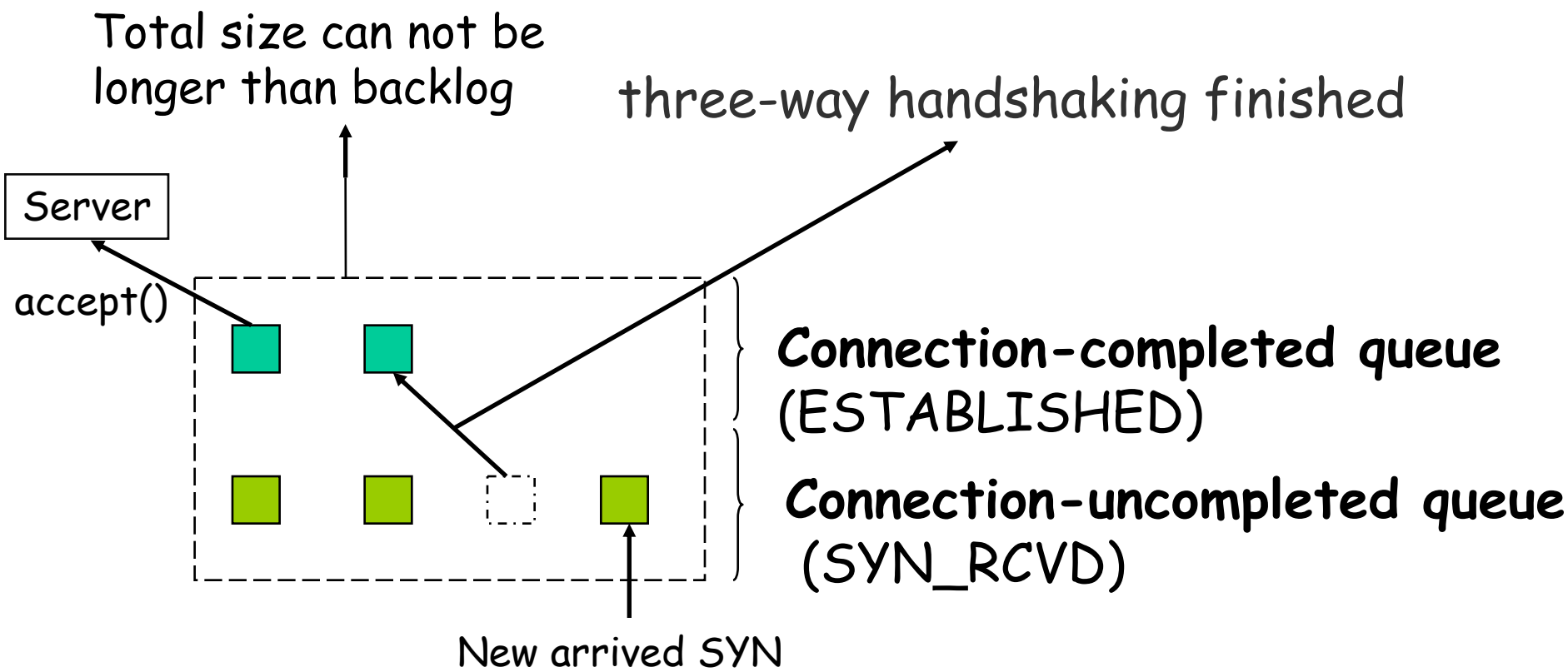


listen()

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

- Parameters
  - [in] socket
    - Specifies the file descriptor associated with the socket.
  - [in] backlog
    - To limit the number of outstanding connections in the socket's listen queue. Implementations shall support values of backlog up to SOMAXCONN, defined in <sys/socket.h>.
- Returns
  - Upon successful completions, **listen()** shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

listen()-backlog



Two Queues maintained by TCP

listen()-backlog

- Linux IPv4 Implementation
  - The size of **connection-uncompleted queue** is defined by **backlog**. But the value of backlog cannot be larger that the value stored in /proc/sys/net/core/somaxconn with default value 128.
  - The size of **connection-uncompleted queue** is defined by /proc/sys/net/ipv4/tcp_max_syn_backlog

accept()

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

- Accept an incoming connection on a listening socket
- Parameters
  - [in] sockfd
    - The listen()ing socket descriptor.
  - [out] addr
    - Filled in with the address of the site that's connecting to you.
  - [out] addrlen
    - Filled in with the sizeof() the structure returned in the addr parameter.
- Returns
  - The newly connected socket descriptor, or -1 on error, with errno set appropriately.

connect()

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

- Once building a socket descriptor with **socket()**, you can **connect()** that socket to a remote server.
- Parameters
  - [in] sockfd
    - The socket descriptor to
  - [in] addr
    - Filled in with the address of the site you wan to connecting to.
  - [in]addrlen
    - Specifies the length of the sockaddr structure pointed to by the address argument..
- Returns
  - 0 if succeed or -1 on error, with errno set appropriately.

send()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t
len, int flags);
```

- Send a message on a socket, used for TCP SOCK_STREAM connected sockets
- Parameters
  - [in] sockfd: Specifies the socket file descriptor.
  - [in] buf: Points to the buffer containing the message to send.
  - [in] len: Specifies the length of the message in bytes.
  - [in] flags: Specifies the type of message reception. Support for values other than 0 is not implemented yet.
- Returns
  - The number of bytes sent if succeed or -1 on error, with errno set appropriately.

sendto()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf,
size_t len, int flags, const struct sockaddr
*to, socklen_t tolen);
```

- Send a message on a socket, used for UDP SOCK_DGRAM unconnected datagram sockets.
- Parameters
  - [in] sockfd: Specifies the socket file descriptor.
  - [in] buf: Points to the buffer containing the message to send.
  - [in] len: Specifies the length of the message in bytes.
  - [in] flags: Specifies the type of message reception. Support for values other than 0 is not implemented yet.
  - [in] address: Points to a sockaddr structure containing the destination address. The length and format of the address depend on the address family of the socket.
  - [in] address_len: Specifies the length of the sockaddr structure

Socket核心函数- recv()

- 功能：
  ① 从TCP接收数据,返回实际接收数据长度，出错返回－1。
  ② 如果没有数据将阻塞，如果收到的数据大于缓存大小，多余数据将丢弃。
- 参数：
  ① Sockfd:套接字描述符
  ② Buf:指向内存块的指针
  ③ Buf_len:内存块大小，以字节为单位
  ④ flags:一般为0

int recv(int sockfd, void *buf, int buf_len,unsigned int flags);

Socket核心函数- recv()

int recv(int sockfd, void *buf, int buf_len,unsigned int flags);

e.g. recv(sockfd, buf, 8192, 0)

Socket核心函数- recvfrom()

- 功能：
  ① 从UDP接收数据，返回实际接收的字节数，失败返回－1
- 参数：
  ① Sockfd:套接字描述符
  ② buf:指向内存块的指针
  ③ buf_len:内存块大小，以字节为单位
  ④ flags:一般为0
  ⑤ from:远端的地址，IP地址和端口号
  ⑥ fromlen:远端地址长度

int recvfrom(int sockfd, void *buf, int buf_len, unsigned int flags, struct sockaddr *from, int fromlen)

Socket核心函数- recvfrom()

int recvfrom(int sockfd, void *buf, int buf_len, unsigned int flags, struct sockaddr *from, int fromlen)

e.g. recvfrom(sockfd, buf, 8192, 0, (struct sockaddr *)&address, sizeof(address));

Socket核心函数-close()

- 功能：
  - ① 关闭socket，实际上是关闭文件描述符
  - ② 如果只有一个进程使用，立即终止连接并撤销该套接字，如果多个进程共享该套接字，将引用数减一，如果引用数降到零，则撤销它。
- 参数：
  - ① Sockfd:套接字描述符

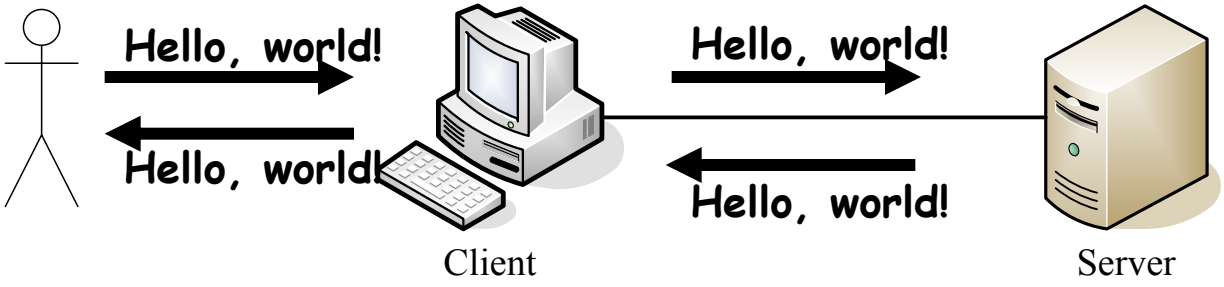
```
#include <unistd.h>

int close (int sockfd);

                                    Returns: 0 if OK, -1 on error
```

Iterative Server

- Echo process



Server

```
#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);
```

Server

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

    Listen(listenfd, LISTENQ);

    for ( ; ; )  {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
        str_echo(connfd);   /* process the request */
        Close(connfd);
    }
}
```

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");
}
```

Client

```
#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);
```

Client

```
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

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

    exit(0);
}
```

Client

```
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);
    }
}
```