网络编程期末复习参考

Macau University of Science and Technology

此文档包含的网络编程知识只覆盖了澳门科技大学，计算机科学与技术，大四上，网络编程课程，2024 年期末考试内容。这是我的个人笔记，仅供学习参考使用。

Sirius

2024.1.1

GitHub repository： https://github.com/violet857/NetworkCoding

Contents

ch1.1 Intro 2

ch1.2 IP_Basic 2

ch 2.1 TCP 4

ch 2.2 TCP 9

ch 3 UDP 13

ch 5 Socket_Options 15

ch 6 Multi-process 16

ch 9 Multi-thread 20

ch 7 Multiplexing 24

ch 8 Multiple Recipients 25

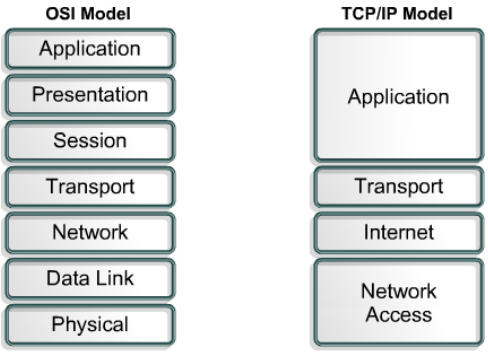
Hint：标黄是期末复习时说会考，标红是上课时说的会考，标蓝是可能会考，没标是补充内容
*选择题 10 个（2*10），问答题 8 个（5*6+10*2），编程题 2 个（20+10）（2024 年 1 月）*

ch 1.1 Intro

➢ 什么是 socket:

A socket is an abstract representation of a communication endpoint through which an application may send and receive data.

ch 1.2 IP Basic



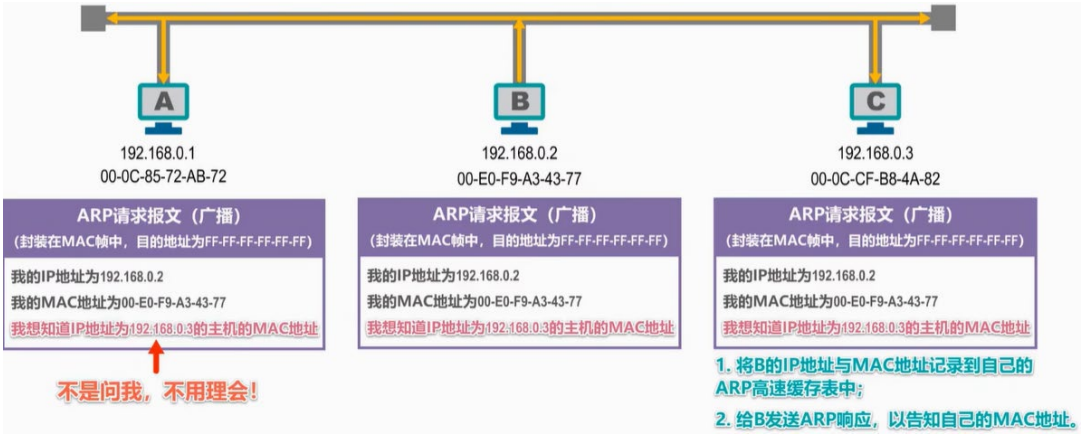
➢ 什么是 ARP 和 RARP:

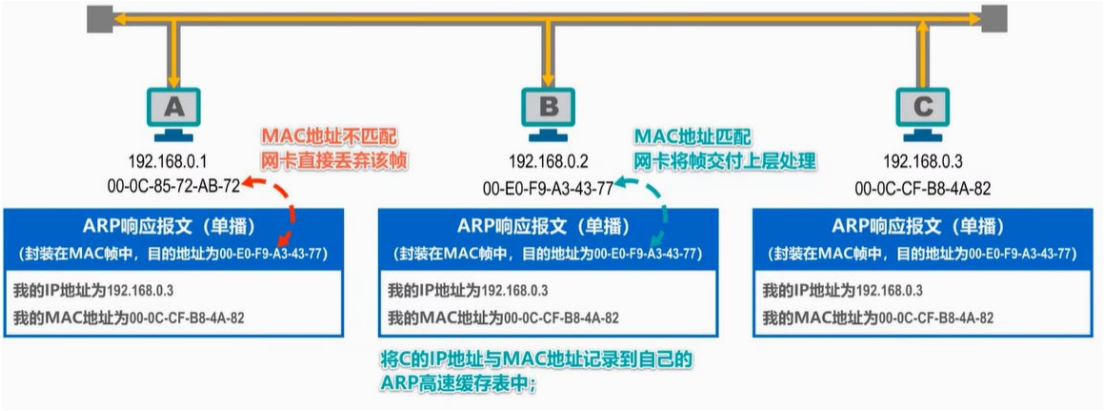
ARP 和 RARP 属于四层协议中的 Internet 层。都是基于广播实现的协议。

ARP: The process of finding the hardware address of a host given the IP address is called **Address Resolution Protocol**.

RARP: The process of finding out the IP address of a host given a hardware address is called **Reverse Address Resolution Protocol**.

➢ ARP 工作流程:





➢ IP 数据报分片：

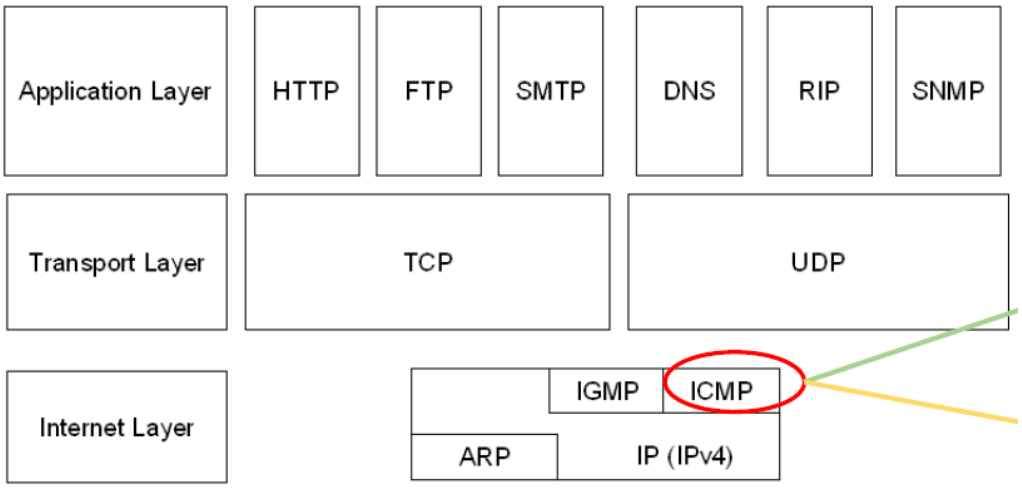
IP 数据报合并（reassembly）只发生在目的主机，如果有片段丢失会向 sender 发送 ICMP message。

以上格式为 IP 数据报分片的方式，FLAG 段中有 DF 和 MF 两个参数。

DF：为 0 时表示此 IP 数据报可以被分片；为 1 时表示不能被分片，如果长度大于 MTU（最大传输单元），则按 DF=0 处理，还是要分片。

MF：为 0 表示这是最后一个 IP 数据报或者没有被分片；为 1 时表示后面还有数据报片段。

➢ 协议对应 TCP/IP 层



记住哪个协议属于哪一层（特别是 ICMP）。

➢ IP 和 MAC 的区别

IP 地址用于网络层 Internet，标识网络和同一网络上的不同主机，互联网中是唯一的。

MAC 地址用于链路层 data link，局域网中标识网络接口硬件，控制帧 frame 的传输。

➢ Since IP address is globally unique, why do we need MAC address?

1. 公网 IP 全球唯一，但是私有 IP 地址会在不同的私有网络重复使用。2. 局域网内传输需要 MAC 地址传输帧 frame。3. IP 地址会改变，MAC 地址为硬件提供标识符。4. 允许使用 DHCP

等协议获取 IP 地址。

➢ TCP 和 UDP sockets 之间的不同

TCP socket: connection-oriented, reliable, congestion control.

UDP socket: connectionless, unreliable, without congestion control.

两者各有优势，TCP 适合可靠、按顺序的数据传输，如 FTP。UDP 适合低延迟和高性能的数据传送，如 VoIP（voice over Internet protocol）。

ch 2.1 TCP

1. Client 或 server 是一个 process，不是指代机器，一个机器可以具有多个 client 和 server。

2. sockaddr 和 sockaddr_in 是两种用于处理网络地址的结构，前者是通用类型，后者是指定互联网地址的特定结构，专用于 TCP/IP 网络，用于指定套接字的地址信息，包含 IP 地址和端口号。但在使用 bind（）等函数时需要将 sockaddr_in 指针转换为 sockaddr 指针，因为这些函数为多网络类型设计而不仅仅是 IP 网络。

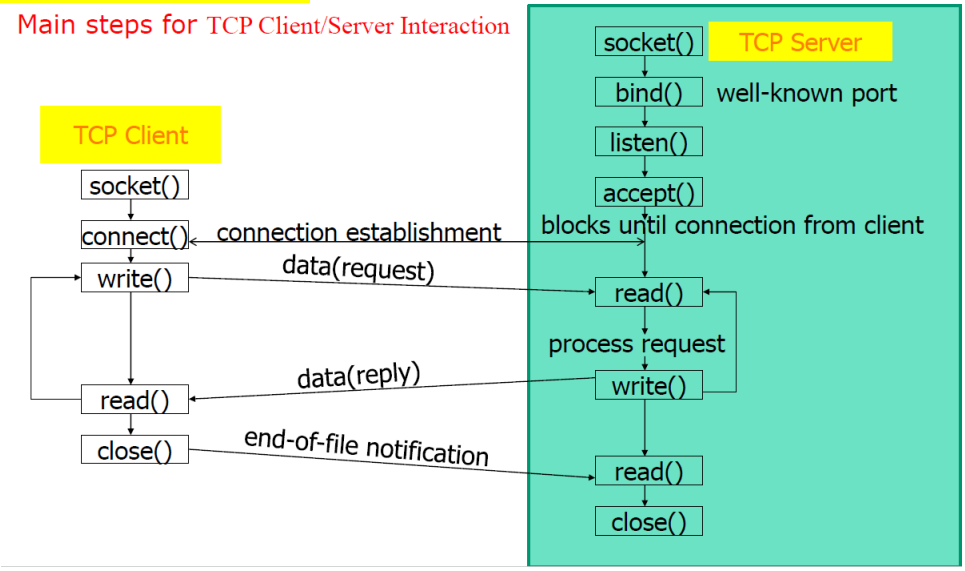
3. 设置 sockaddr_in 结构:

```
memset(&serv_addr, 0, sizeof(serv_addr)); // init
serv_addr.sin_family = AF_INET; // ipv4
serv_addr.sin_port = htons(1234); // port
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // ip
```

   赋值之前先初始化。addr.sin_family 指定 ipv4 或 ipv6 协议；addr.sin_port 指定要绑定或者连接的网络字节序的端口号；addr.sin_addr.s_addr 存储网络字节序的 IP 地址。

4. sin_addr 结构体专门用于存储一个 IPv4 地址。这个结构体只包含一个成员，通常命名为 s_addr，它是一个无符号长整数（unsigned long），用于存储网络字节序的 IPv4 地址。sockaddr_in 中用于存储 IP 地址的 sin_addr 成员就是 in_addr 类型。

➢ TCP 连接主要步骤及函数功能



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

此函数用于创建一个新的套接字，用于初始化网络通信的端点。create a socket

family：指定套接字使用的协议族，例如 AF_INET(PF_INET)用于 IPv4 互联网协议，AF_INET6 (PF_INET6)用于 IPv6。

type：指定套接字类型。 SOCK_STREAM 表示流套接字（用于 TCP），SOCK_DGRAM 表示数据报套接字（用于 UDP）。

protocol：指定具体的协议。通常情况下设置为 0，让操作系统自动选择 type 对应的默认协议。例如 TCP 或 UDP。

return：返回非负整数，即套接字描述符的 descriptor。如果调用失败返回-1。

```
int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        // 错误处理
    }
    // 使用 sockfd 进行其他网络操作...
    return 0;
}
```

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

此函数用于 server，将套接字与特定的本地 IP 地址和端口号绑定。assign local IP addr & port to a socket

sockfd：之前通过调用 socket() 函数创建的套接字描述符。

my_addr：这是一个指向 sockaddr 结构的结构体指针，该结构包含了套接字监听的 IP 地址和端口号。对于 IPv4，会使用 struct sockaddr_in 结构，然后将其转换为 struct sockaddr 类型的指针来传递给 bind()。

addrlen：这是 addr 指针指向的地址结构的长度。

return：调用成功返回 0，失败返回-1。

```
int main() {
    int sockfd;
    struct sockaddr_in addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        // 错误处理
    }
    addr.sin_family = AF_INET;// 指定协议 IPv4
    addr.sin_port = htons(8080); // 指定端口 8080，htons 为字节序转换，后面会提到
    addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // 绑定到指定 ip 地址，inet_addr 转换字节序
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) == -1) {// addr 要转换成 sockaddr 指针
        // 绑定失败的错误处理
```

```
    }
    // 其他操作，例如 listen()和 accept()
    return 0;
}
```

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

此函数用于 server，将一个未连接的（unconnected）套接字转换为被动的监听套接字，用于接受来自客户端的连接请求。Marks the socket as accepting connections

sockfd：之前通过调用 socket() 函数创建的套接字描述符，通过 bind() 函数绑定到一个本地地址。

backlog：指定未完成连接请求的最大数目。它定义了在套接字处理请求（accept()）之前，可以在内部队列中等待的未完成连接请求的数量。

return：调用成功返回 0，失败返回-1。

```
if (listen(sockfd, 10) == -1) { // 最大排队请求个数为 10
    // 错误处理
}
```

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

此函数用于 server，接受一个来自客户端的连接请求。从已建立的连接队列中接收一个待处理的客户端连接。accept a connection on a socket

sockfd：socket() 创建，bind() 绑定到本地地址，并通过 listen() 设置为监听模式的套接字。

addr：这是一个指向 struct sockaddr 结构的指针，用于接收连接的客户端的地址信息。在调用结束后，这个结构将被填充为客户端的地址信息。

addrlen：这是一个指向 socklen_t 变量的指针，该变量指定接收客户端地址信息的结构体大小。

return：成功时返回一个新的套接字描述符，用于与连接的客户端进行后续的通信（write 等操作）。这个新的套接字与最初的监听套接字是不同的。如果 accept() 失败，则返回-1。

```
struct sockaddr_in clnt_addr;
clnt_addr_size = sizeof(clnt_addr);
clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
if(clnt_sock == -1){
    // 错误处理
}
```

此函数会阻塞 server 进程，直到与 client 建立连接。

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

此函数用于 client，建立客户端与服务器的连接，初始化到服务器指定地址和端口的 TCP 连接。initiate a connection on a socket

sockfd：客户端先前通过调用 socket() 创建的套接字描述符。

addr：这是一个指向 struct sockaddr 的指针，其中包含了服务器的地址和端口信息。对于 IPv4 通常使用 struct sockaddr_in 结构，需要将其转换为 struct sockaddr 类型的指针。
addrlen：addr 指针指向的结构体的长度。
return：调用成功返回 0，失败返回-1。

```
sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
serv_addr.sin_port = htons(1234);
if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1){
    printf("connect() error!");
    exit(1);
}
```

此函数会阻塞 client 进程，直到第三次握手完成或失败。

```
int write (int fd, const void *buf, size_t count)
```

此函数用于向一个已建立连接的套接字发送数据。
fd：指定要写入数据的套接字。通常是通过 connect()（在客户端）或 accept()（在服务器端）连接成功后的套接字描述符。
buf：这是一个指向数据缓冲区的指针，write() 函数从这个缓冲区中读取数据并发送到套接字。
count：要写入套接字的字节数。
return：成功调用返回写入的字节数，出现错误则返回-1。

```
char message[]="Hello MUST!";
if(write(clnt_sock, message, sizeof(message)) == -1){
    // 错误处理
}
```

当数据缓冲区为空时，即 write()没有读取到任何数据或者输出缓冲区已满时，会阻塞主进程。
需要注意的是，写入不一定会立即发送，他会存储在发送队列中择机发出。

```
int read (int fd, void *buf, size_t count);
```

此函数用于向一个已建立连接的套接字接收数据。
fd：指定要读取数据的套接字。通常是通过 connect()（在客户端）或 accept()（在服务器端）连接成功后的套接字描述符。
buf：这是一个指向存放数据缓冲区的指针，用于存储从套接字接收的数据。
count：存放数据缓冲区的大小，即最大可读取的字节数。
return：成功调用返回实际读取的字节数，如果对端已关闭连接返回 0，出现错误则返回-1。

```
char message[50];
int str_len;
str_len = read(sock, message, sizeof(message)-1); //减 1 保留字符的终止字符位置
if (str_len == -1){
    // 错误处理
```

}
printf("Message from server: %s\n", message);
当套接字内数据为空时，即 read()没有读取到任何数据时，会阻塞主进程。

```
int close (int fd);
```

此函数用于关闭一个套接字。destroy a socket
fd：这是要关闭的套接字描述符。
return：成功调用返回 0，出现错误则返回-1。

```
if (close(sockfd) == -1) {
    // 错误处理
}
```

server 关闭监听套接字不会影响已建立连接的套接字，关闭已连接的套接字不会影响监听套接字。client 关闭套接字则会直接断开连接。

- 为什么只有 server 需要 bind?

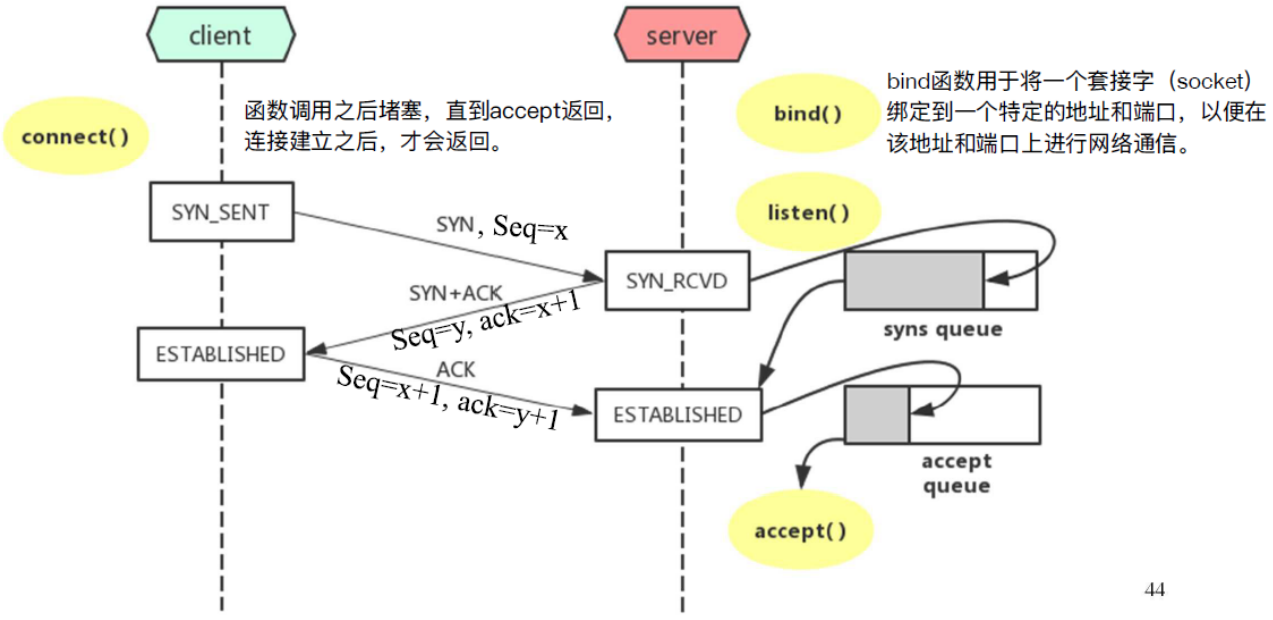
1. 服务器需要监听一个固定的地址和端口，固定的地址和端口使得客户端可以找到并连接到服务器。
2. 某些服务需要运行在指定端口，如 HTTP 端口为 80。
3. 客户端不需要固定端口，它们使用动态端口（也称为临时端口），这个端口在套接字建立连接时(socket())由操作系统自动分配。
4. Client 不需要显式地（explicitly）知道自己的 ip 地址和端口也可以通信，所以不需要 bind。

- Client 如何获得的 ip 和端口号?

when connect() runs, client socket get IP(host IP) and port(randomly assigned). 端口号通常避开了 well-known 端口号（0-1023）和 organization 端口号（如 1024-49151）。

- 与三次握手相关的函数?

connect()，bind()，listen()，accept()。



当 accept() 返回时表示三次握手已成功完成，服务器与客户端之间建立了 TCP 连接。在 accept()

返回后双方才可以开始数据传输。

- 与四次挥手相关的函数?

close() 和 shutdown()。程序调用 close()或者 shutdown()之后，开始执行四次挥手。

- 数据边界 data boundary

数据边界（Data Boundary）指的是对数据单元开始和结束的明确划分。

SOCK_STREAM：流（stream）套接字，用于 TCP，提供的是一种面向连接的、可靠的数据传输服务。特点是双向通信（Bidirectional）、可靠（Reliable）、有序（Sequenced）、无重复（Unduplicated）、无记录边界（Without record boundaries）。无记录边界即接收者需要自己判断数据的开始和结束，如打电话。

SOCK_DGRAM：数据报（datagram）套接字，用于 UDP。特点是无连接（Connectionless）、无序（Unordered）、不可靠（Unreliable）、双向通信（Bidirectional）、有记录边界（Record boundaries）。有记录边界即每个数据报都携带足够的信息来被独立处理。接收方可以明确知道每个数据报的开始和结束，如收到邮件或短信。

TCP 由于无记录边界，所以一段信息可以通过多次 read 获取。但在 UDP 中，如果一个数据报包含 100 字节的数据，但 read 只读取了 50 字节，那么在读取后剩下的 50 字节数据将会被丢弃，无法再次访问。

- shutdown()

在 close()调用之前还可以调用 shutdown()，用于实现优雅地关闭 TCP 套接字。与 close() 函数不同，shutdown() 不会释放套接字描述符，它仅仅改变套接字的可用状态，如允许关闭套接字的读、写或者读写两个方向 shutdown() changes the sending and receiving operations on a socket。在调用 shutdown() 之后，通常还需要调用 close() 来释放套接字和相关资源。

```
int shutdown (int sockfd, int how);
```

sockfd：需要关闭的套接字描述符。

how：指定关闭的方式。SHUT_RD（或者 0）：关闭套接字的读取部分（不再接收数据）；SHUT_WR（或者 1）：关闭套接字的写入部分（不再发送数据）； SHUT_RDWR（或者 2）：同时关闭套接字的读取和写入部分。

return：0 表示成功。-1 表示失败。

```
if (shutdown(sockfd, SHUT_WR) != 0) {   //关闭写入部分
    //错误处理
}
close(sockfd);
```

ch 2.2 TCP

- 端口号分类

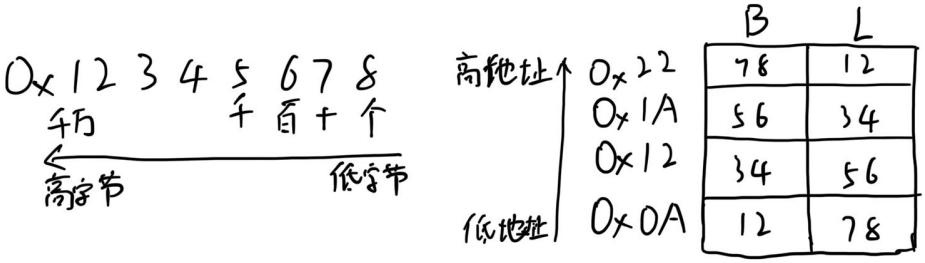
[1, 1023]的端口为知名端口号（well-known），不能用于绑定给 socket。[1024, 49151]为注册端口号（Registered），企业使用，用户可以使用但要检查是否已经被占用。[49192, 65535]为私有端口号（Private），用户可以随便使用。所以 socket 绑定端口号的范围为 [1024, 65535]。

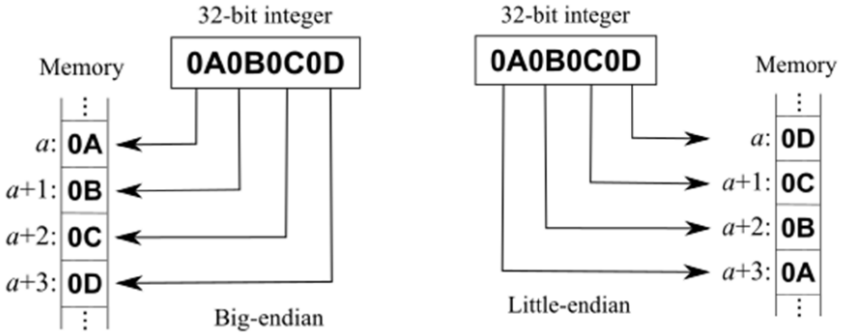
- 字节序 byte order

Hint：存储数据一般情况下是从低地址位开始存储。

大端序（Big-endian）：符合人类阅读习惯的顺序，从左向右存储。高位字节存储在低地址位。早期的处理器和网络字节顺序是大端序。

小端序（Little-endian）：从右向左存储，低字节存储在低地址位。X86 和 ARM 架构是小端序，因为 CPU 从低位开始计算，小端序中低地址和低字节相同，便于计算。





- 主机与网络字节序转换

主机字节序 host byte order 是小端序，网络字节序 network byte order 是大端序。

h: host;　　n: network;　　s: short;　　l: long;

htonl()：将无符号长整型数（如 uint32_t）从 host 转换为 network 字节序。

htons()：将无符号短整型数（如 uint16_t）从 host 转换为 network 字节序。

ntohl()：同理

ntohs()：同理

端口号使用整数存储时，转换用 htons()和 ntohs()；IPv4 地址如果使用 32 整数表示（不是点分十进制），可以使用 htonl()和 ntohl()转换。

- 为什么单字节字符不用考虑字节序问题?

when a data type only consists of one byte, there is nothing to reorder.

- 字符串形式地址转换

当 IP 地址使用点分十进制表示，并存储在字符串中时，需要使用以下函数转换字节序。

```
in_addr_t inet_addr (const char *cp);
```

将一个点分十进制的字符串形式的 IPv4 地址转换为其网络字节序的二进制形式。

cp：指向一个 C 字符串的指针，包含了一个标准的 IPv4 地址（如 "192.168.1.100"）。以下同理

return：通常是一个无符号长整型（unsigned long），表示网络字节序（大端序）的 IPv4 地址。不合法地址返回 INADDR_NONE（通常是 -1）。

```
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_addr.s_addr=inet_addr("192.168.1.100");
```

需要注意的是，此函数只适用于 ipv4 地址，可以判断地址是否合法，但是不能判断是否转换成功。推荐使用更现代的 inet_pton()。

```
int inet_aton (const char *cp, struct in_addr *inp);
```

作用与 inet_addr()相同，但是可以判断转换是否成功。

inp：指向一个 struct in_addr 结构的指针。如果转换成功，这个结构将被填充为相应的网络字节序的二进制地址。

```
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
if (inet_aton("192.168.1.100", &serv_addr.sin_addr.s_addr)) {
    // 转换成功
}
```

此函数也只能处理 IPv4 地址，能够判断转换是否成功。

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

将网络字节序的二进制 IPv4 地址转换成点分十进制的字符串形式。

in：是一个 struct in_addr 结构，其中包含了一个网络字节序的 IPv4 地址。

return：指向字符串的指针，该字符串表示了点分十进制的 IPv4 地址。如果有错误返回-1。

```
struct in_addr addr; //专门用于存储一个 IPv4 地址的结构体
addr.s_addr = htonl(0xC0A80164); // 192.168.1.100 在主机存储的整数形式
char *ip_str = inet_ntoa(addr);
```

存在静态内存问题，下面会提到。仅支持 IPv4 地址。

以下两个函数涉及 IPv6 内容，可能不会考:

```
int inet_pton (int af, const char *src, void *dst);
```

将点分十进制的字符串格式（对于 IPv4）或十六进制字符串格式（对于 IPv6）的 IP 地址转换为其网络字节序的二进制形式。其中 p 指的是 internet protocol, n 指的是 number。

af：指定地址族，IPv4 是 AF_INET，IPv6 是 AF_INET6。

src：指向点分十进制的字符串格式（对于 IPv4）或十六进制字符串格式（对于 IPv6）的 IP 地址的指针。

dst：指向目标结构的指针（对于 IPv4 是 struct in_addr，对于 IPv6 是 struct in6_addr），用于存储转换后的二进制地址。

return：成功时返回 1。输入不是有效的 IP 地址时返回 0。发生错误时返回 -1。

```
struct in_addr ipv4addr;
if (inet_pton(AF_INET, "192.168.1.1", &ipv4addr)) {
    // IPv4 地址转换成功
}
```

```
const char *inet_ntop (int af, const void * src, char * dst,
                                                socklen_t size);
```

将网络字节序的二进制形式转换为其点分十进制的字符串格式（对于 IPv4）或十六进制字符串格式（对于 IPv6）的 IP 地址。

af：指定地址族，IPv4 是 AF_INET，IPv6 是 AF_INET6。

src：指向目标结构的指针（对于 IPv4 是 struct in_addr，对于 IPv6 是 struct in6_addr），用于存储转换后的二进制地址。

dst：指向点分十进制的字符串格式（对于 IPv4）或十六进制字符串格式（对于 IPv6）的 IP 地址的指针。

size：指定目标存储的大小，以确保不会发生缓冲区溢出。

return：成功时返回指向目标存储的指针。发生错误时返回 NULL。

```
char addr_str[16]; // 对于 IPv4
struct in_addr ipv4addr;
if (inet_ntop(AF_INET, &ipv4addr, addr_str, sizeof(addr_str))) {
    printf("IPv4 address: %s\n", addr_str);
}
char addr_str6[46]; // 对于 IPv6
struct in6_addr ipv6addr;
if (inet_ntop(AF_INET6, &ipv6addr, addr_str6, sizeof(addr_str6))) {
    printf("IPv6 address: %s\n", addr_str6);
}
```

- 静态内存问题

inet_ntoa() 函数在将网络字节序的二进制 IPv4 地址转换成点分十进制字符串时，使用静态内存来存储转换后的字符串。这意味着每次调用 inet_ntoa() 时，它都会使用相同的内存位置来存储结果。

```
struct in_addr addr1, addr2;
addr1.s_addr = inet_addr ("192.168.1.1");
addr2.s_addr = inet_addr ("192.168.1.2");
const char *result1 = inet_ntoa(addr1);
const char *result2 = inet_ntoa(addr2);
printf("Address 1: %s\n", result1); // 期望输出 192.168.1.1，实际输出 192.168.1.2
printf("Address 2: %s\n", result2); // 期望输出 192.168.1.2，实际输出 192.168.1.2
```

- INADDR ANY

INADDR_ANY 是一个特殊的 IP 地址（0.0.0.0），表示 "任意地址" 或 "所有接口"。通常用作绑定（bind）操作的 IP 地址参数，允许服务器监听所有网络接口（例如多个网卡或 IP 地址）上的客户端连接请求。

```
struct sockaddr_in server_addr;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY); //使用 htonl，因为其实际上是个整数常量
```

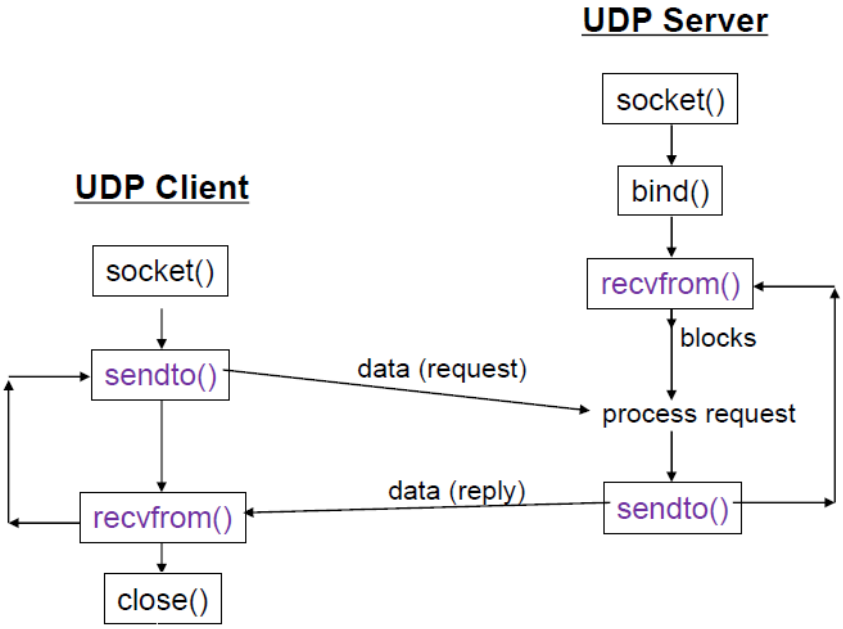
- atoi()

用于将字符串转换为整数。

char *str = "1234";

serv_addr.sin_port = htons(atoi(str));

ch 3 UDP



- 与 TCP 不同的函数

```
serv_sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

创建 socket 与 TCP 相同，除了 type 和 protocol 传入的参数。type 为 SOCK_DGRAM 指定套接字类型为数据报；protocol 为 IPPROTO_UDP 指定协议为 UDP（其实也可以为 0，让操作系统自动选择）。

```
ssize_t recvfrom (int sockfd, void *buf, size_t len, int flags,
                  struct sockaddr *src_addr, socklen_t *addrlen);
```

类似于 TCP 的 read，此函数用于从 UDP 套接字接收数据，并可以指定发送端的地址信息。

sockfd：UDP 套接字描述符。

buf：指向接收数据的缓冲区的指针。

len：缓冲区大小。

flags：指定调用方式的标志，通常设置为 0。

src_addr：指向 struct sockaddr 的指针，指明要接收谁的消息。

addrlen：指向存储 src_addr 地址长度的变量的指针。

return：调用成功它返回读取的字节数；连接已经关闭返回 0；发生错误返回-1。

char buffer[1501];

struct sockaddr_in client_addr;

```
socklen_t clnt_addr_size = sizeof(clnt_addr);
ssize_t received = recvfrom(sockfd, buffer, 1501, 0,
                            (struct sockaddr *)&client_addr, &clnt_addr_size);
if (received == -1) {
    // 错误处理
}
```

```
ssize_t sendto (int sockfd, const void *buf, size_t len, int
  flags, const struct sockaddr *dest_addr, socklen_t addrlen);
```

参数和返回值与 recvfrom 相同，唯一不同的是后两个参数用于指明要发送给谁。

```
char buffer[1501];
struct sockaddr_in server_addr;
socklen_t serv_addr_size = sizeof(server_addr);
ssize_t sent = sendto(sockfd, buffer, 1501, 0, (struct sockaddr *)&server_addr, &serv_addr_size);
if (sent == -1) {
    // 错误处理
}
```

recvfrom一次读一个 sendto 发送的数据块，如果 recvfrom 的 buffer 不够大，一次读不完接收到的数据块，recv 队列中剩余的数据会被丢掉且不会重传。所以一个 recvfrom 对应一个 sendto。让 recvfrom 的 flag = MSG_PEEK 时，接受数据但不会移除 RecvQ 中的缓冲数据，可以基于即将接收到的数据作出决定，如判断 recvfrom 的 buffer 够不够大。

- 与 TCP 相比缺少了什么？为什么？

缺少了 listen(), connect(), accept(), shutdown()。因为 UDP 不需要建立连接。

- UDP socket connection

在 UDP 中，可以调用 connect() 函数，它并不会建立一个真正的连接，只是设定默认的目标地址和端口，这样就可以使用 send() 和 recv() 函数，而不是像 sendto()和 recvfrom（）一样每次调用都需要指定目标。因此，UDP 中断开 connect()实际上只是取消之前设定的默认目标地址和端口。

```
int sockfd;
struct sockaddr_in serverAddr;
// 创建 UDP 套接字
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
// 设置服务器地址结构体
memset(&serverAddr, 0, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
// 使用connect 函数连接到服务器
if (connect(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
    //错误处理
```

```
}
// 发送和接收数据
send(sockfd, "Hello", 5, 0);
char buffer[1024];
recv(sockfd, buffer, sizeof(buffer), 0);
// 关闭套接字
close(sockfd);
```

ch 5 Socket Options

修改套接字的属性是为了向操作系统传达应用程序的意愿，但是不一定会被操作系统响应。有些 socket 的属性不可修改（因为只读）。

- 为什么 sock 描述符从 3 开始?

```
int serv_sock = socket(AF_INET, SOCK_DGRAM, 0); // serv_sock >= 3
```

程序启动时，通常会自动打开三个文件描述符：0 标准输入（stdin）；1 标准输出（stdout）；2 标准错误（stderr）。创建新的描述符时会分配下一个可用的最小文件描述符。

- getsockopt()

用于获取 socket 的属性的值。

```
int getsockopt (int sockfd, int level, int optname, void
                                        *optval, socklen_t *optlen);
```

sockfd：想要查询的套接字的描述符。
level：指定选项代码的级别。例如要指定查询套接字使用 SOL_SOCKET（课堂中只用过这一个参数）。
optname：需要获取的属性的名称。课堂中只练习过 SO_RCVBUF，SO_SNDBUF，SO_TYPE 三个参数，分别对应接收缓冲区大小、发送缓冲区大小、套接字类型。
optval：指向变量的指针，用于存储查询的属性的值。
optlen：存储属性值的区域的 size。
return：调用成功返回 0，错误返回 -1。

```
socklen_t len;
int rec_buff, snd_buff, soc_type, state;
len = sizeof(rec_buff);
if ((state = getsockopt(serv_sock, SOL_SOCKET, SO_RCVBUF, &rec_buff, &len)) == -1){
    // 错误处理
}
printf("receive buffer size: %d\n", rec_buff); // receive buffer size: 65536
if((state = getsockopt(serv_sock, SOL_SOCKET, SO_TYPE, &soc_type, &len)) == -1){
    // 错误处理
}
printf("Socket type: %d \n", soc_type); // Socket type: 1   (1 为 TCP 套接字，2 表示为 UDP)
```

- setsockopt()

用于修改套接字属性的值。

```
int setsockopt (int sockfd, int level, int optname, void
                                        *optval, socklen_t optlen);
```

所有参数和返回值与 getsocket()相同，唯一不同的是 optlen 不是指针类型。后三个参数用于修改 socket 而不是查询。如 SO_TYPE 就是只读属性，不可修改。

optval：指向一个变量，属性的值会被修改成这个变量的值。

snd_buff = snd_buff-1024;

state = setsockopt(serv_sock, SOL_SOCKET, SO_SNDBUF, &snd_buff, sizeof(snd_buff))

- buff size 可以修改吗？

可以修改。SO_RCVBUF 和 SO_SNDBUF 是可以修改的套接字选项，它们分别用于设置套接字接收缓冲区和发送缓冲区的大小。但是大多数操作系统都对这些缓冲区的大小有最小值和最大值的限制，如果设置的值超出了这个范围，它会被自动调整到这个范围内。但是 SO_TYPE 不可修改。

Value passed to setsockopt() is not guaranteed to be the new size of the socket buffer, even if the call apparently succeeds. Receiving and sending buffer of TCP and UDP can only be modified in a range which set by the operating system.

- SO_REUSEADDR

Default: 0；Reuse: set 1。SO_REUSEADDR = 1 时，即使原先的套接字处于 TIME_WAIT 状态（正在进行四次挥手），新套接字也可以立即绑定到相同的端口上。

```
int yes = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes)) == -1) {
    perror("setsockopt");
    // 处理错误
}
```

ch 6 Multi-process

- 并发式服务器实现的三种方式以及对应的 function

multi-process：fork(),wait(), waitpid()

multi-thread：pthread_create(), pthread_join(), pthread_detach()

multiplexing：select(), FD_ZERO(), FD_SET(), FD_ISSET(), FD_CLR()

- 程序与进程的区别

程序是储存在硬盘上的一组指令，当程序执行时就是进程 process。

- fork()

此函数会创建一个父进程的副本（包括 memory，register 等）称为子进程。

```
pid_t   fork (void);
```

return：在父进程中返回子进程的 PID。子进程中返回 0。

```
if (pid == -1) {
    // 错误处理
} else if (pid > 0) {
        // 父进程代码
    wait(NULL); // 等待子进程结束
} else {
    // 子进程代码
}
```

需要注意的是，子进程几乎与父进程一样，但是进程 ID（PID）、父进程 ID（PPID）不一样。并且子进程拥有 separate identical copy of parent's virtual address space，这意味着子进程进行修改变量值等操作时不会影响到父进程。

- 获取 PID

init() 的 PID 为 1，getpid（）获取该进程 ID，getppid() 获取该进程的父进程 ID。

- Orphan and Zombie

孤儿进程：When a parent process terminates before its child process, while child process is still running, the child process becomes an orphan process. 孤儿进程会被 init 进程（PID 为 1）收养。

```
pid_t pid = fork();
if (pid > 0) {
   // 父进程立即退出
   exit(0);
} else {
   // 子进程休眠一段时间
   sleep(10); // 孤儿进程在此期间被 init 进程收养
}
```

僵尸进程：When a process has finished execution, but its parent process has not yet called wait() or waitpid() to reclaim its status information, the process becomes a zombie process. 僵尸进程占用的资源非常少，主要是进程表中的一个条目，但过多的僵尸进程会占用系统资源。

```
pid_t pid = fork();
if (pid > 0) {
   // 父进程休眠，不立即等待子进程
   sleep(10);
   // 此时，子进程可能已经结束，成为僵尸进程
   wait(NULL); // 最后回收子进程资源
} else {
   // 子进程立即退出
   exit(0);
}
```

上述例子中，子进程执行 exit()后父进程没有立即调用 wait() 或 waitpid()，因此子进程会变成

僵尸进程，直到父进程调用 wait() 为止。注意，父进程结束时，它的僵尸子进程通常不会自动被回收。它们会被 init 接管。init 会定期等待（wait）其子进程，包括那些僵尸子进程，从而回收它们并释放资源。

➢ 处理僵尸进程

wait()和 waitpid()可以防止僵尸进程产生，如何实现的？他们使父进程等待并回收子进程的退出状态。They will make the parent process wait to reclaim the child process's exit state。

```
pid_t wait (int *status);
```

此函数会等待任意一个子进程结束。如果调用 wait() 时没有任何子进程已经结束，父进程将被阻塞，直到至少有一个子进程结束。

status：用于获取状态信息，可以传入 int 类型的指针，一般为 NULL 即可。

return：成功会返回被回收的子进程的 pid，失败会返回-1。

```
pid_t pid = fork();
if (pid > 0) {
   // 父进程代码
   wait(NULL); // 回收子进程资源
} else {
   // 子进程代码
   exit(0); // 子进程结束
}
```

```
pid_t waitpid (pid_t pid, int *status, int options);
```

与 wait() 唯一的区别就是，waitpid()是等待一个特定的子进程。而且通过设置参数，waitpid()可以在等待子进程时不阻塞父进程，父进程可以继续执行。

pid：>1 时等待 PID=pid 参数值的子进程；<-1 时等待同一进程组中 PID = pid 参数绝对值的子进程；=-1 时功能与 wait()相同，等待任意子进程；=0 时等待同一个进程组中的任何一个子进程。

status：与 wait()相同。

options：通常为 0，也可以设置成 WNOHANG，让 waitpid 以非阻塞方式运行。

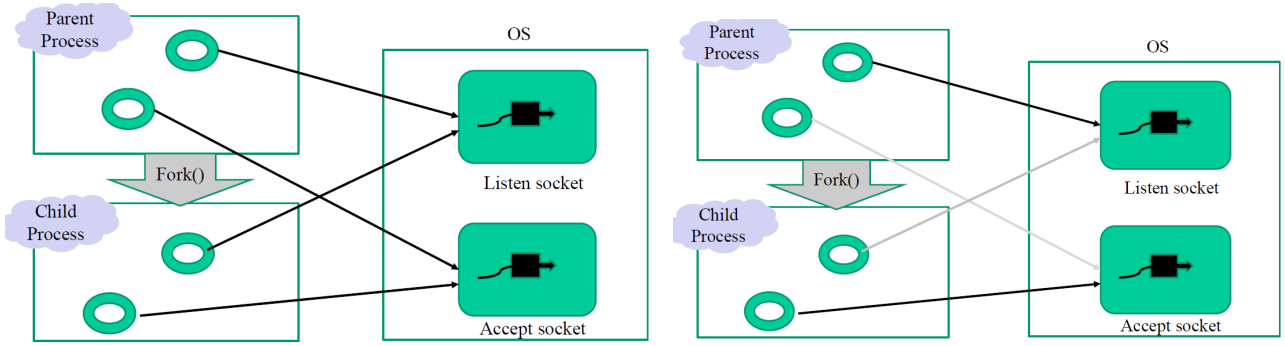
return：成功会返回被回收的子进程的 pid，失败会返回-1。如果 options = WNOHANG 并且子进程没有结束，会立即返回 0。

进程组 process group：是一个或多个进程的集合，用于进程管理和信号传递。每个进程组由一个唯一的进程组 ID（Process Group ID，简称 PGID）来标识。Linux 系统中，每个进程除了有一个唯一的进程 ID（PID）外，还属于一个进程组。子进程在创建时默认继承其父进程的进程组 ID。

```
pid_t pid = fork();
if (pid > 0) {
   // 父进程代码
   waitpid(pid, NULL, WNOHANG); // 以非阻塞方式回收特定子进程资源
} else {
   // 子进程代码
   exit(0);   }   // 子进程结束
```

➢ TCP echo server fork 注意事项

首先，一个 socket 可以有多个文件描述符；当一个 socket 有多个文件描述符时，close 其中一个文件描述符并不会关闭 socket；当关闭 socket 的最后一个描述符时 socket 才会关闭；fork 时会一同 fork 父进程的文件描述符。



在 client 尝试建立连接时，server 父进程会 fork 一个子进程用于处理 TCP 连接。我们需要关闭子进程的 listen socket 的描述符，关闭父进程的 accept socket 描述符。因为一个 server 子进程用于处理与一个 client 建立的 TCP 连接，并在 connection 断开后退出，它不需要监听新的 connect 请求；而 server 父进程不需要处理已经建立的 TCP 连接，它只需要监听是否有新的 connect 请求，并在接收到请求后 fork。

server 父进程代码示例：

```
clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
if (clnt_sock == -1){
        // 错误处理
}
processID = fork();
if (processID < 0){
    //错误处理
}else if (processID == 0){    // 子进程代码段
      close(serv_sock);   // 关闭子进程中的 listen socket 文件描述符
      HandleTCPClient(clnt_sock, clnt_addr);   //封装的函数，用于处理新建立的 TCP 连接
      exit(0);
}
printf("with child process: %d\n", (int)processID);
close(clnt_sock);   // 关闭父进程中的 accept socket 文件描述符
```

➢ Does close(clntSock) in the parent process destroy the clntSock in the child process?

关闭父进程中的 accept socket 文件描述符会破坏 socket 吗？No. The close() call in the parent process only affects the file descriptor in the parent process. The child process still has its own file descriptor, which remains open and valid.

➢ Does the parent process keep the new connection?

父进程会维护新的连接吗？NO，new connection is handled by the child process. Parent process continues to listen for new connections.

➢ 如果 fork 时不关闭 server 中的 accept socket 描述符会怎么样？

如果没有 close(clnt_sock)，父进程的文件描述符可能会被耗尽。因为如果不关闭，server 需要维护一个 listen socket 的文件描述符，每当有一个 client 连接时还要维护一个新的 accept socket 的文件描述符，当有足够多的 client 连接时就会耗尽父进程的文件描述符。

- sigaction()

sigaction 函数也可以用于回收子进程，与 wait()不同的是，wait()是被动地等待子进程结束，而 sigaction()可以在接收到子进程结束的信号时主动地回收子进程。
SIGCHLD 是子进程结束时通知父进程的信号。用法：

```
struct sigaction act;
act.sa_handler = read_childproc; // 设置接收到 SIGCHLD 信号时的处理函数
sigemptyset(&act.sa_mask); // 清空信号集，确保处理函数执行时不会阻塞其他的信号
act.sa_flags = 0;  // 等于 0 表示不处理特殊的信号量，不写也行
int state = sigaction(SIGCHLD, &act, 0); // 改变信号处理方式，state = 0 改变成功
if (state != 0){
    printf("Error occured!7 \n");
    exit(1);
}
void read_childproc(int sig){  // 信号处理函数，只能返回 void 并接受一个 int 参数
    if (sig != SIGCHLD){ // 如果是其他信号则不处理
        return;
    }
    pid_t pid;
    int status;
    pid = waitpid(-1, &status, WNOHANG);  // 以非阻塞的方式等待任意一个子进程
    printf("remove proc id: %d \n", pid);
}
// 其他函数，开始创建子进程接受 client 的请求
```

需要注意的是，sigaction()只要在创建子进程之前设置都是有效的。

ch 9 Multi-thread

- 主要函数

```
int pthread_create (pthread_t *thread, const pthread_attr_t
                    *attr, void *(*)(void *), void *arg);
```

pthread_create()会创建一个线程，与主线程共享 heap data 和 data area，但是拥有自己独立的 stack area。

thread：指向 pthread_t 类型变量的指针，作为新创建线程的标识符。pthread_create() 执行成功后，这个变量将保存新线程的 ID。

attr：用于指定各种线程属性，如栈大小、调度策略等。一般传递 NULL ，表示使用默认线程属性。

第三个参数：这是一个函数指针，指向由线程执行的函数。这个函数必须返回 void * 并接受一个 void * 参数。
arg：参数指针，指向传递给线程执行的函数的参数。
return：成功返回 0，失败返回错误号。

```
void *thread_main(void *arg) {
	int i;
	int cnt = *((int*)arg);
	for(i = 0; i < cnt; i++) {
		sleep(1);
		puts("running thread");
	}
	return NULL;
}
int main(int argc, char* argv[]) {
	pthread_t tid;
	int thread_param = 5;   // 传递给线程的参数
	if(pthread_create(&tid, NULL, thread_main, (void*)&thread_param) != 0) {
		puts("pthread_create() error");
		return -1;
	}
	sleep(10);
	puts("end of main");
	return 0;
}
/*Output:
running thread
running thread
running thread
running thread
running thread
end of main	*/
```

- What will happen if main process does not sleep for the end of thread?

程序会在创建的线程完成任务之前结束。在多线程程序中，当主函数结束时，整个进程及其所有线程都将终止。

- 线程等待

```
int pthread_join (pthread_t thread, void **retval);
```

实际上不知道线程什么时候会结束，此时就需要在主线程中调用 pthread_join，主线程被阻塞，直到指定的线程结束。使用 pthread_join() 可以防止主线程在其他线程完成之前结束。
thread：要等待的线程的标识符。通过 pthread_create() 创建线程时获取的 pthread_t 类型的值。
retval：指向指针的指针，用于获取被等待线程的返回值。如果不关心线程的返回值，可以传

NULL。
return：成功返回 0，失败返回错误码。

```
int main(int argc, char* argv[]) {
    pthread_t tid;
    int thread_param = 5;   // 传递给线程的参数
    if(pthread_create(&tid, NULL, thread_main, (void*)&thread_param) != 0) {
        // 错误处理
    }
    pthread_join(tid, NULL);   // 等待线程结束
    puts("end of main");
    return 0;
}
```

```
int pthread_detach (pthread_t thread);
```

用于将指定的线程置为 "分离" 状态。当一个线程处于分离状态时，它的资源在终止时会自动被回收，即不需要主线程显式地调用 pthread_join() 来等待线程结束并回收资源。
thread：要等待的线程的标识符。
return：成功返回 0，失败返回错误码。

```
pthread_create(&threadID, NULL, thread_main, (void*)&thread_param);
if (pthread_detach(threadID) != 0) {   // 将线程设置为分离状态
   perror("pthread_detach");
}
// 主线程可以继续它的执行，无需等待子线程结束
```

➢ 什么是 critical section?

临界区（Critical Section）是指一个程序、程序段或代码区域，它执行访问共享资源（如共享内存、共享文件等）的操作，并且这个区域的代码不能被多个线程同时执行。Critical section is a chunk of code in which, for correctness, it is necessary to ensure that only one thread of control can be in that section at a time.

```
void * thread_inc(void * arg)
{
    int i;
    for(i=0; i<50000000; i++)
        num+=1;
    return NULL;
}
void * thread_des(void * arg)
{
    int i;
    for(i=0; i<50000000; i++)
        num-=1;
    return NULL;
}
```

critical section

critical section

包含访问、修改共享资源（全局变量、文件、内存）的代码片段就是临界区。

➢ 使用 mutex 解决临界区问题

互斥锁 mutex 用于防止多个线程同时访问共享资源。当一个线程想要访问受保护的资源时，它

必须先获取对应的互斥锁（pthread_mutex_lock）。如果锁已经被另一个线程持有，请求锁的线程将被阻塞，直到锁可用。当线程完成了对受保护资源的访问后，它应该释放互斥锁（pthread_mutex_unlock），使其他线程能够获取锁并访问资源。

```
pthread_mutex_t lock; // 声明一个互斥锁
void* do_work(void* data) {
    pthread_mutex_lock(&lock); // 获取互斥锁
    // 执行涉及共享资源的操作
    pthread_mutex_unlock(&lock); // 释放互斥锁
    return NULL;
}
int main() {
    pthread_mutex_init(&lock, NULL); // 初始化互斥锁
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, &do_work, NULL);
    pthread_create(&thread2, NULL, &do_work, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    pthread_mutex_destroy(&lock); // 销毁互斥锁
    return 0;
}
```

- 使用 semaphore 解决临界区问题

信号量（Semaphore）与 mutex 不同，mutex 是谁锁谁开，不能控制访问临界区的顺序，信号量则是资源空闲时就可以访问，可以用于顺序执行。

```
sem_t semA; // 声明一个控制线程 A 的信号量
sem_t semB; // 控制线程 B 的信号量
void* threadA(void* arg) {
    sem_wait(&semA); // 等待信号量 semA
    // 线程 A 的临界区代码
    sem_post(&semB); // 允许线程 B 进入临界区
    return NULL;
}
void* threadB(void* arg) {
    sem_wait(&semB); // 等待信号量 semB
    // 线程 B 的临界区代码
    return NULL;
}
int main() {
    pthread_t tA, tB;
    sem_init(&semA, 0, 1); // 初始化 semA，第三个参数为 1，允许线程 A 首先进入
```

```
    sem_init(&semB, 0, 0); // 初始化，第二个参数设置为 0 表示只允许 1 个线程同时访问

    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);

    pthread_join(tA, NULL);
    pthread_join(tB, NULL);

    sem_destroy(&semA);   //销毁信号量
    sem_destroy(&semB);
    return 0;
}
```

- multi-process server 与 multi-thread server 的区别?

Multi-threaded Servers: memory sharing, consuming fewer resources, create more thread with one system call, termination takes less than process, don't need extra mechanism to communicate.

Multi-process Servers: isolated memory, consuming more resources, more than one system call to create more than one process, termination takes more than thread, needing IPC to communicate.

ch 7 Multiplexing

多路复用 Multiplexing 实现的并发服务器实际上不是严格意义上的并发，它只是实现了一个信道的复用。

- 五个步骤

Set file descriptors:
初始化文件描述符集合并添加需要监视的套接字。文件描述符是指代开放文件或套接字的整数标识符。使用 FD_ZERO 来初始化文件描述符集合，然后用 FD_SET 将特定的套接字（如服务器监听套接字和客户端套接字）添加到集合中。
Set scope for monitoring file descriptors:
确定 select() 函数应监视的文件描述符的范围，即在每次调用 select() 之前更新和设置文件描述符集合的当前状态。
Set timeval:
定义 select() 函数的超时时间，即函数等待文件描述符变为就绪状态的最大时间。
Call select():
执行 I/O 多路复用操作，select() 函数监视指定的文件描述符集合，等待其中一个或多个变为就绪状态（可读、可写）。
View the call results:
分析 select() 的返回值，确定还有哪些文件描述符是就绪的，并据此执行相应的读取、写入或错误处理操作。使用 FD_ISSET 确定哪些文件描述符就绪并处理它们。

- 函数

FD_ZERO(fd_set *set)初始化文件描述符集合。FD_SET(int fd, fd_set *set) 将一个 server 套接字描述符加入到集合中。FD_CLR(int fd, fd_set *set) 从文件描述符集合中移除一个套接字描述符。FD_ISSET(int fd, fd_set *set) 用于检查 select() 调用后，是否还有文件描述符为就绪状态，如果有就返回 1，执行相应的处理，例如读取数据、接受新连接等。

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
nfds：监控的文件描述符的数量，通常设置为监控的最大文件描述符加 1（1 + largest file descriptor）。因为此值表示的是文件描述符的个数，而不是最大文件描述符的值。
readfds：指针指向需要检查可读性的文件描述符集合。
writefds：指针指向需要检查可写性的文件描述符集合。
exceptfds：指针指向需要检查异常条件的文件描述符集合。
timeout：用于指定 select 调用的超时时间。可以设置为 NULL 以无限等待。
return：> 0 表示就绪的文件描述符的数量；= 0 表示超时，没有文件描述符就绪；< 0 发生错误。

- 示例

见 multiplexing_server.c。

ch 8 Multiple Recipients

- 广播、组播如何实现的？

Only UDP sockets can use broadcast and multicast services. 广播只能在同一局域网内传播 local scope，组播可以跨局域网（如一个学校内的多个 LAN）。但是大多数的 ISP（internet service providers）不支持跨互联网（Internet）的多播。

- 组播

使用组播时，需要修改 socket 的 option，执行加入/退出组播组的操作。

sender:

```
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&mul_adr, 0, sizeof(mul_adr));
mul_adr.sin_family = AF_INET;
mul_adr.sin_addr.s_addr = inet_addr(argv[1]); //设置组播地址（224.0.0.0 ~239.255.255.255）
mul_adr.sin_port = htons(atoi(argv[2]));  //设置发送到哪个端口
// 设置 option，加入组播，设置组播消息的跳数为 64，注意此处的 level 参数，只有组播使用
//  IPPROTO_IP，其余情况都是 SOL_SOCKET
int time_live = 64;
setsockopt(send_sock, IPPROTO_IP, IP_MULTICAST_TTL, (void *)&time_live, sizeof(time_live));
// 发送组播，UDP 使用 sendto
sendto(send_sock, buff, strlen(buff), 0, (struct sockaddr *)&mul_adr, sizeof(mul_adr));
```

receiver:

```
struct ip_mreq join_adr; // 声明一个 ip_mreq 结构体存储加入组播所需的信息
```

```
recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_addr.s_addr = htonl(INADDR_ANY); //不限制套接字的地址，即可以监听所有可用地址
adr.sin_port = htons(atoi(argv[2])); //设置监听端口
bind(recv_sock, (struct sockaddr *)&adr, sizeof(adr));  //将套接字与地址信息绑定
join_adr.imr_multiaddr.s_addr = inet_addr(argv[1]);  // 设置加入的组播地址
join_adr.imr_interface.s_addr = htonl(INADDR_ANY);  // 设置监听所有的端口
// 修改套接字 option，使得 IP_ADD_MEMBERSHIP 等于之前设置的组播信息
setsockopt(recv_sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void *)&join_adr, sizeof(join_adr));
// BUF_SIZE – 1 是为了保留字符串最后的结束符
str_len = recvfrom(recv_sock, buf, BUF_SIZE - 1, 0, NULL, 0);
```

- 广播

使用广播时，需要修改 sender 中的 socket 的 option，执行打开/关闭广播开关的操作。

sender：

```
struct sockaddr_in broadcast_addr;
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&broad_adr, 0, sizeof(broad_adr));
broadcast_addr.sin_family = AF_INET;
broad_adr.sin_addr.s_addr = inet_addr(argv[1]);  // 设置广播地址（IP 地址的主机段全为 1）
broad_adr.sin_port = htons(atoi(argv[2]));  //
// 设置 socket option，让 SO_BROADCAST = 1，打开 socket 广播开关
int so_brd = 1;
setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST, (void *)& so_brd, sizeof(so_brd));
//发送广播，UDP，所以使用 sendto
sendto(send_sock, buff, strlen(buff), 0, (struct sockaddr *)&broad_adr, sizeof(broad_adr));
```